Advanced concepts in Modelica and their implementation in VehProLib

Master's thesis
performed in **Vehicular Systems**

by
Otto Montell

Reg nr: LiTH-ISY-EX-3511-2004

10th May 2004

Advanced concepts in Modelica and their implementation in VehProLib

Master's thesis

performed in **Vehicular Systems**,
Dept. of Electrical Engineering
at **Linköpings universitet**

by **Otto Montell**

Reg nr: LiTH-ISY-EX-3511-2004

Supervisor: **Anders Fröberg**
Linköpings Universitet

Examiner: **Associate Professor Lars Eriksson**
Linköpings Universitet

Linköping, 10th May 2004

Avdelning, Institution
Division, Department

Vehicular Systems,
Dept. of Electrical Engineering
581 83 Linköping

Datum
Date

10th May 2004

Språk
Language
☐ Svenska/Swedish
☒ Engelska/English
☐ _____

Rapporttyp
Report category
☐ Licentiatavhandling
☒ Examensarbete
☐ C-uppsats
☐ D-uppsats
☐ Övrig rapport
☐ _____

ISBN
—

ISRN
LITH-ISY-EX-3511-2004

Serietitel och serienummer **ISSN**
Title of series, numbering —

URL för elektronisk version
http://www.vehicular.isy.liu.se
http://www.ep.liu.se/exjobb/isy/2004/3511/

Titel Avancerade koncept i Modelica och deras användning i VehProLib

Title Advanced concepts in Modelica and their implementation in VehProLib

Författare Otto Montell
Author

Sammanfattning
Abstract

VehProLib is one of many libraries being developed for the object oriented multi-domain language Modelica. The layout and the current status of the library are shown. The aims of the library are to provide the user with a number of different components with different levels of complexity. The components included range from mean value engine components to in-cylinder models. An efficient way to handle parameters using records is provided. Different bus systems are implemented and discussed. Furthermore are replaceable fluid models introduced in the library. It will be shown that Modelica is a very efficient way to create an advanced modelling library.

Nyckelord
Keywords Modelica, VehProLib, In-cylinder models, Mean value engine models, Fluid models

Abstract

VehProLib is one of many libraries being developed for the object oriented multi-domain language Modelica. The layout and the current status of the library are shown. The aims of the library are to provide the user with a number of different components with different levels of complexity. The components included range from mean value engine components to in-cylinder models. An efficient way to handle parameters using records is provided. Different bus systems are implemented and discussed. Furthermore are replaceable fluid models introduced in the library. It will be shown that Modelica is a very efficient way to create an advanced modelling library.

Keywords: Modelica, VehProLib, In-cylinder models, Mean value engine models, Fluid models

Acknowledgment

Thanks first and foremost to Associate Professor Lars Eriksson, ISY, for providing me with an interesting and challenging thesis work. His support and useful ideas about the development of the VehProLib is gratefully acknowledged. Thanks also to Ph.D. Student Anders Fröberg for supervising my thesis work, always being available as an discussion partner, and listen to me complaining about weird and ununderstandable error messages from Dymola. Thanks to Johanna Wallén and Daniel Claesson for all the fun moments in our "office" at the Division of Vehicular Systems. Last, but not least, thanks to all the people at the Division of Vehicular Systems for all the amusing moments and accepting me as one of the group.

Otto Montell
Linköping, 26th of April, 2004

Contents

Chapter 1

Introduction

This thesis is about Modelica in general and VehProLib in particular. This first chapter contains two parts, first an introduction and reading recommendation to the forthcoming chapters, and a part about the limitations of this thesis. In general are Modelica keywords written in **bold** font, and the name of models from VehProLib are written in `teletype` font.

The outline is as follows:

Chapter 2, Modelica This chapter is a brief introduction to Modelica, the intended reader is someone without or with little previous experience of Modelica. The more advanced user should consider reading the section about inheritance as it's of great importance. The chapter is finished by an example, an electrical circuit.

Chapter 3, VehProLib An introduction to the VehProLib is found here, central part is the section about the connector. The more interested reader should also read about the partial classes.

Chapter 4, Advanced concepts in Modelica This is the first of the two major chapters in this thesis. A number of concepts is introduced, replaceable components, parameter handling and the construction of a Signal Bus. As this is the central chapter it should be read by everyone.

Chapter 5, Medium models Here the replaceable gas model is introduced, the purpose is to give the user a number of gas models to chose from. By doing so the user can chose the level of complexity needed. This chapter should also be read by everyone.

Chapter 6, Concluding comments Personal comments and recommendations for future work.

The limitations that apply are the following:

Dymola All the models presented in the thesis has been simulated using Dymola 5.1b++, there might be some unforeseen compatibility issues with previous version of Dymola and other simulation environments.

Source code Most of the code fragments in the thesis has been edited for readability reasons, this include but is not limited to, all graphical code has been removed and the declarations statement has been shortened. The code in the Appendixes remains in their original form, except for graphical information.

Validation No real effort has been made to validate the models used in the library. They should instead be seen as an example what can be achieved using Modelica.

Chapter 2

Modelica

This chapter will give an brief introduction to the multi-domain modelling language Modelica, for further information please refer to [8].

Modelica is a standardized modelling language built around acausal modelling with mathematical equations. Modelica is a multidomain language, ie the model can be mix of several different domains, for example both electrical and mechanical domains are supported. Furthermore it's an object oriented language, constructs for extending and reusing models are provided.

2.1 Making models in Modelica

The Modelica language is being constructed by the Modelica Association. The aim is to construct a standard language for describing physical models. Because the language is not built around assignment, the equations can be written in their original form, the way they are usually found in textbooks. Modelica models are built from small models representing a small part of the bigger model. These models can then be joined together to form bigger and more advanced models. This concept can be illustrated by looking in the Electrical library of the Modelica Standard Library, the models found there are, small and simple components such as resistor and inductor. These components can then be joined together to form more complex circuits, note that this can be done in several layers.

Modelica provides a number of standard libraries, like the one mentioned above and for example the mechanical library. The user is free to construct his own libraries, both parts from the standard libraries and own models can be used. VehProLib is of course an example of a user created library. When the libraries and models needed are complete,

modelling will seem much like looking in a book, picking the components and setting the parameters to their appropriate values. One of the most important tasks in achieving this is the re-usability of the models. If two similar models is to be made, they should both utilize a generic model and only differ in there parameter settings.

2.2 The structure of models

A model in Modelica contains three parts, name, declarations and equations, written in the following way.

```
class name "comment"
  declarations;
equation
  equation;
end name;
```

The following example is taken from [9]. The first program of any programming language usually prints "Hello world" on the screen, but as this would make no sense in a modelling language, we will instead create a model to solve a simple differential equation.

$$\dot{x} = -a * x, \quad x(0) = 2, \quad a = 2 \tag{2.1}$$

The model is written in Modelica like this:

```
class solveDiff "Solves a differential equation"
  Real x(start=2, fixed = true);
  parameter Real a = 2;
equation
  der(x)=-a*x;
end solveDiff;
```

The following things can be noted, first the name of the model is solveDiff. In the declaration section the variable x is declared and its initial value is fixed to two. After that the parameter a is declared, both of them is of the type Real. To learn more about different types see [8]. In the equation part of the model equation 2.1 can be found, note that it's written in same form as above. Observe that there is no assignment here, one doesn't need to specify which are input and output. The causality of the equation is unspecified and only becomes specified when the equation is solved.

2.3 Restricted classes

In the Modelica language there are several restricted classes, each with different restrictions added to them. To use a restricted class simply

replace the keyword **class** with one of the following keywords: **records, connector, model, block, function, type, and package**.

2.3.1 Connectors

The most important of the restricted classes is the connector, because it is used to connect different models specifying the communication between the models and will determine much of design and modelling capabilities. The design of connector is a very important step in the design process, because all the following models will use the connector in one way or another. So an error or an overlooked signal, could lead to large and tiring re-writhing of the code. Transition between different domains are usually done by adding different connectors in a model, eg a rotational to transitional gear would include one rotational connector and one transitional connector, and a equation describing the conversion. Below is the code for the positive electrical connector in Modelica Standard Library used to connect electrical models:

```
connector PositivePin
  "Positive pin of an electric component"
  SI.Voltage v "Potential at the pin";
  flow SI.Current i "Current flowing into the pin";
end PositivePin;
```

Restriction that apply to the connector is that there can be no equations or algorithms. A connector can declare two types of variables, non-flow and flow. Flow variables is declared using the flow prefix as seen above.

Connections
Connection between models can be created using a connect statement, the connectors must be of the same type. The connect statement creates an acausal connection, the direction of the dataflow is not needed to be known. To create a causal connection, with determined direction of the data flow, a connect statement between a connector declared as input and one declared as output can be used.

Two types of equations can be created using the connect statement, depending on if the variable is declared flow or not.

1. Equality equation, for non flow variables.
2. Sum to zero, for flow variables.

If two electrical pins are declared, as pin_a and pin_b, the statement connect(pin_a,pin_b) would result in the following two equations,

$$pin\_a.v = pin\_b.v$$
$$pin\_a.i + pin\_b.i = 0$$

Obviously the connect statement creates the equations corresponding to Kirchhoff's law at a junction.

2.3.2 Packages and functions

These two classes are nice to know, but not as important as the connector. The packages class is used to create new libraries. All new libraries must contain a package class.The package class will set the name of the library and may also include declarations of constants and other classes.

The function class is a fixed causality block, input-output block. Each component in the interface must have either an input or output attribute. Secondly there may be no equation in the equations block, instead algorithms are used. Algorithms are fixed casuality and not dependent on time, so in functions the value is assigned to the output variable, indicated by the use of := instead of the ordinary =. Calling a function requires that all components specified as input must be provided by the calling model, because functions are re-initialized every time a function call is made. A simple example of a function can look like this:

```
function mean "Calculates the mean of two numbers"
  input Real x;
  input Real y;
  output Real z;

algorithm
  z := (x+y)/2;
end mean;
```

2.4 Inheritance

One of the more powerful features of Modelica is the possibility to inherit and extend the properties of that class. The basic class known as a superclass is extended to create a more specific class with more detailed properties, this class is known as a subclass. The subclass will inherit all properties of the superclass, such as declaration and equations. The contents of the superclass is reused or in other words inherited by the subclass. A common approach is to declare the superclass **Partial**, this tells the user that the class isn't complete yet and needs to be extended, ie given more content. This process will be illustrated using a model of a resistor from Electrical Library in the Modelica Standard Library.

```
partial model OnePort
  "Component with two electrical pins p
```

```
  and n and current i from p to n"

  SI.Voltage v
    "Voltage drop between the two pins";
  SI.Current i
    "Current flowing from pin p to pin n";
  PositivePin p;
  NegativePin n;
equation
  v = p.v - n.v;
  0 = p.i + n.i;
  i = p.i;
end OnePort;
```

This model is used as a superclass for the resistor model below and a number of other models. Note that the OnePort is partial, as it is lacking any relationship between the current and the voltage. This relationship would in this case be the well known Ohm's law, U=RI, so if this relationship is added a resistor can implemented in the following way,

```
model Resistor "Ideal linear electrical resistor"
  extends Interfaces.OnePort;
  parameter SI.Resistance R=1 "Resistance";
equation
  R*i = v;
end Resistor;
```

Note that this model then in turn can be extended, to for example a Temperature dependent electrical resistor by specifying an equation for the resistance dependence on the temperature.

2.5 Example

The purpose of this example is to show how modelling is made in Modelica, and to give something to compare with the problem will first be solved using Simulink. The circuit to be modelled is complex enough to show the strength of Modelica, but anyone with limited knowledge of electrical circuits should have no problem understanding the different steps.

The circuit which is to be modelled contains one voltage source V, a resistor R, an inductor L and a capacitor C. The inductor and capacitor is in parallel and the other two is in series with them, figure 2.2 shows the layout of the circuit.

To be able to model the circuit in Simulink one must first obtain the equations describing the circuit. In this example it's sufficient to use such well-know relations as Kirchhoff's and Ohm's laws. For bigger and more complex systems one might have to resort to bond-graphs to get the equations, for more information about bond-graphs see [6]. The following equations describe the circuit in the example,

$$\begin{aligned} U - U_R - U_L &= 0 \\ U_R &= Ri \\ i &= i_C + i_L \\ U_L &= C\frac{di_L}{dt} \\ i_C &= L\frac{dU_C}{dt} \\ U_L &= U_C \end{aligned} \tag{2.2}$$

Figure 2.1 shows the implementation of the equations in Simulink. As

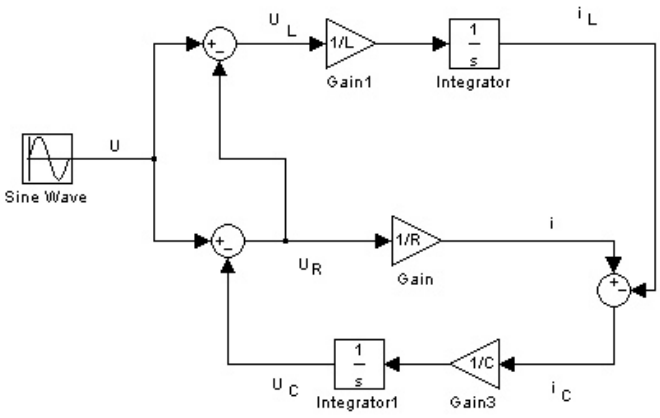

Figure 2.1: Circuit modelled in Simulink.

can be seen the physical structure of the circuit is not retained, instead the blocks are arranged in computational order. The block diagram is equivalent to assign statements for the derivatives of the two state variables (i_L and U_C).

2.5.1 Equation based solution

If one was to write Modelica code for the circuit, following the guidelines shown in the first code fragment in this chapter, the resulting code may look something like below. The first thing that should be noted is that the equations are written in same form as in equation 2.2, there is no need to manipulate the equations by hand. This might seem like a feasible approach but a number of difficulties arise, foremost it is still very difficult to understand what the code is a model of. Secondly it is very error prone as one needs to write every equation by hand, this is of course not a big problem in this case, but if a more complex circuit was to be modelled this would be a very tedious work. Because of these difficulties another approach must be found.

```
model Circuit
  package SI = Modelica.SIunits;
  package Math = Modelica.Math;
  constant Real PI=Modelica.Constants.PI;

  parameter SI.Resistance R=50;
  parameter SI.Capacitance C=0.05;
  parameter SI.Inductance L=0.1;
  parameter SI.Voltage A=10;

  SI.Current i, iL, iC;
  SI.Voltage U, UC, UR, UL;
equation
  U - UR - UL = 0;
  UR = R*i;
  i = iC + iL;
  UL = L*der(iL);
  iC = C*der(UC);
  UC = UL;
  U = A*Math.sin(2*PI*time);
end Circuit;
```

2.5.2 Component based solution

This approach requires that a model is made for every component required. Once these models are complete, the modelling work will be very easy. Figure 2.2 shows same electrical circuit modelled in Modelica using components from the Electrical library of the Modelica Standard Library. As can be seen the physical structure is maintained, there is no pre-determined directions for the signals. This model tells the program what is to be simulated, and the program then in turn decides how this done. The source code generated for the example is shown

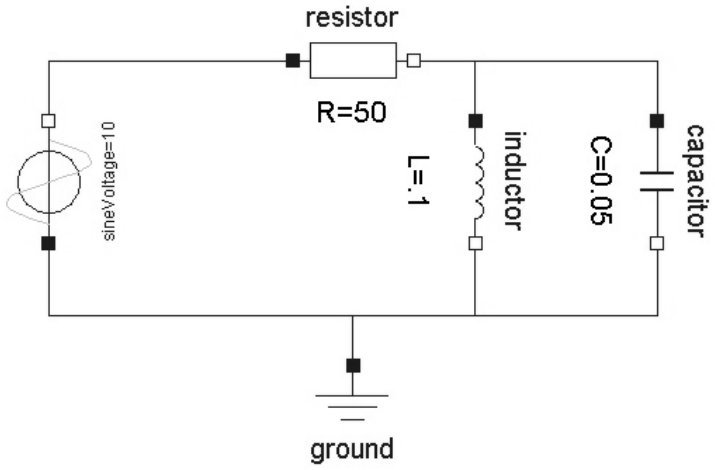

Figure 2.2: Circuit modelled in Modelica.

below.

```
model Circuit
  Ground ground;
  Capacitor capacitor(C=0.05);
  Inductor inductor(L=.1);
  Resistor resistor(R=50);
  SineVoltage sineVoltage(V=10);
equation
  connect(sineVoltage.n, resistor.p);
  connect(resistor.n, inductor.p);
  connect(capacitor.p, resistor.n);
  connect(inductor.n, ground.p);
  connect(capacitor.n, ground.p);
  connect(sineVoltage.p, ground.p);
end Circuit;
```

The name of the model is Circuit, and it consists of five parts, ground, resistor, inductor, capacitor and voltage source. The parameters are set in the code directly following the declaration of the component, and can be changed directly in the code. In the equation part of the model, the connections between the different parts is specified. The connect statement can be written in any order, it is possible to flip a component and for example instead connect capacitor.p to ground.p.

A final remark, this code has been edited for readability reasons, the lines specifying the graphical layout of the circuit has been removed as they are not important.

Chapter 3

VehProLib

In this chapter the basic models and connectors used in the VehProLib[1] library is presented. Some of the topics from the last chapter will exemplified and some new will be added.

3.1 Introduction to VehProLib

The purpose of the library is to provide the user with an easy to use and highly replaceable set of models, to be used as a stand alone library or in cooperation with other Modelica libraries. Currently the library contains the following packages, see figure 3.1, note that some of the packages contain packages them self. Please note that this is the structure as of right now, packages may be added and old ones may be removed as the development progresses.

The packages contain models of different parts of an engine and driveline, as well as some examples to provide the user with ideas how to use the library. In the following sections some of the packages will be presented. The aim of the library is that it should comply to all Modelica standards, so it can be used with ease with other libraries, foremost the VehicleDynamics Library and the Powertrain Library. The interfaces on models in question are implemented using the standard Modelica Rotational Library, so this should not pose any problem. Another aim was the replaceability of the component models, one should be able to start with a simple component model and if further knowledge is needed, simply exchange the component to an more advanced model of the *same* component. These aims were originally stated by Eriksson in [3].

[1] VehProLib stands for Vehicle Propulsion Library, and it is being developed by Division of Vehicular Systems at Linköping University. Point of contact is Associate Professor Lars Eriksson, larer@isy.liu.se

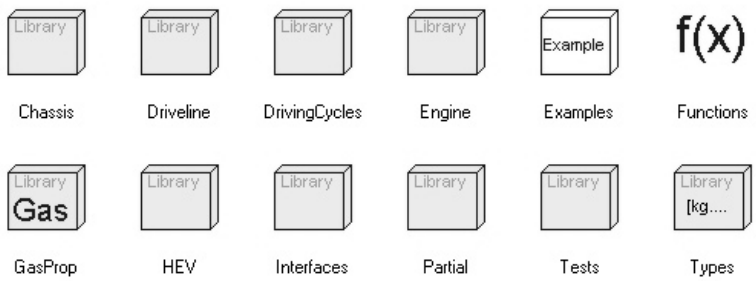

Figure 3.1: Packages included in the VehProLib library.

3.2 The connector, FlowCut

There are several different connectors available in the library. All of them can be found in the `VehProLib.Interfaces` package. The most important one is the flow connector. Other available connectors are `FlowCut_liquid`, the `Geometry` connector and finally the `SignalBus` connector. The flow connector is an enhanced version of the pWHT-connector described in [11], the modifications aim to support the use of gases with multiple components.

```
connector FlowCut
  "Baseclass for standard connector"
  package SI = Modelica.SIunits;
  parameter Integer n=1 "Number of gascomponents";
  SI.Pressure p(nominal=100000, start=100000)
    "Pressure sensed by the connector";
  SI.Temperature T(nominal=500, start=300)
    "Temperature sensed by the connector";
  SI.MassFraction x[n](
    max=1,
    min=0,
    nominal=0) "Mass Fraction";
  flow SI.HeatFlowRate H
    "Enthalpy flowing through the connector";
  flow SI.MassFlowRate Wx[n]
    "Mass flow of gas components";
  flow SI.MassFlowRate W
    "Mass flow through the connector";
end FlowCut;
```

As can be seen there are three intensity variables, namely pressure, temperature and mass fraction. The mass fraction variable, $x[n]$, describes the mass fraction of the ingoing components in the fluid sensed by the connector. The size of the array $x[n]$ is set by parameter n, n being the number of components in the fluid. If a single component fluid is used, $x[n]$ is of course equal to one. The flow variables has been expanded with $Wx[n]$, mass flow through the connector of each ingoing component. The W variable is no longer strictly necessary as the sum of $Wx[n]$ should always equal W, but W is kept because the total mass flow is often of greater interest than the partial mass flows. Some limitations apply to the use of this connector. It's not possible to connect components completely at random, it is for example not possible to connect two restrictions in parallel and then followed by another restriction. The reason for this is that there might be a temperature difference between the two restriction in parallel. The solution is to always put a control volume between restrictions.

This connector should not be used directly, instead two connectors, `FlowCut_i` and `FlowCut_o` are provided, they both inherit the `FlowCut` connector and each add a different icon. The reason for this is, that it should be easy to tell the difference between them in a connection diagram.

3.3 Partial models

The library is built around fairly large amount of partial models, this provides the user with a basic structure on which to build more advanced models. All models with a working fluid originate either from a model called `OnePin` or `TwoPin`. As the names suggests the `OnePin` is the superclass for all models with just one pin, for example the `Ambient` model and the `TwoPin` is the superclass for models with two pins. All models of working fluids originate from the `GasPropBase` class. Mechanical models like the connecting rod and such, don't have any superclasses in the library. These partial models can all be found in `VehProLib.Partial` library, note that there are also some partial models in other parts of the package, for example partial engine models are found in `VehProLib.Engine.Partial`.

Figure 3.2 shows the overall structure of the library, all models above the dashed line are partial models and must be extended. The boxes below the line don't represent actual models found in the library in all cases, they should instead be seen as an indication of which type of models that are constructed using that partial class.

The `TwoPin` is in turn extended to two different partial models, the `TwoPinStatic` and the `TwoPinDynamic`. The `TwoPinStatic` is superclass for all restrictions and the `TwoPinDynamic` is superclass for all

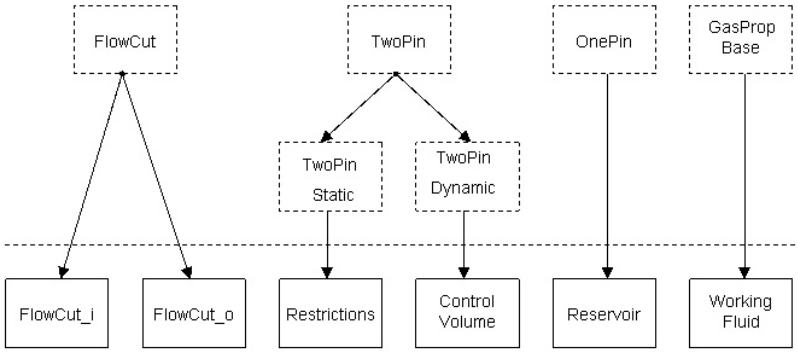

Figure 3.2: Relationships between models, partial models are dashed. Boxes below the line don't represent actual models in all cases, they should be seen as an indication of which type of models that are constructed using that partial class.

control volumes. The `TwoPin` model contains two connectors and a model for the working fluid. No equations is provided at this stage. Some basic equations is added in the next step. The `TwoPinStatic` adds equations so the flow variables will sum to zero as there can be no build up of mass or energy in a restriction. This is not true for the `TwoPinDynamic` model. Instead here are all intensity variables set equal, ie both connectors will sense the same pressure and so forth.

When working with partial models and inheritance it's always important to consider, how many steps are required to achieve a working model. If to many step are used it will become difficult to understand what each step accomplish. Here only two steps is used, and the partial models are clearly defined, each step adding a distinct ability to the model. An example of a model that is to finely fragmented is the `MvemCylinder` model in `VehProLib.Engine` library, where another three partial models are used before a working model is obtained.

3.4 Control volumes and restrictions

Both the control volumes and restrictions must be modified to handle multi-component working fluids. In the case of restrictions an equation specifying which mass ratio array to use, the task is simply to pick out the upstream one. The control volume needs a more advanced balance equation. To handle the changes in the mass ratio array that might result if different initial mass ratios are specified.

The control volume implemented in VehProLib library is idealized

to model the mixing of inflowing gases as instantaneous. This results in an uniform pressure and temperature through out the entire volume. The volume acts like an buffer between restrictions, holding mass and energy. Depending on the approximations used to model the volume, the number of states needed are of course different. In the simplest models, where temperature is considered constant and no mixtures are implemented, only one state will be needed, the pressure in the volume. The model used in the library contains at least two states. One state to model the energy in the fluid and one state for the mass of each component in the working fluid.

Two equations are modified to be able to handle mixture of fluids,

$$\frac{dmx[n]}{dt} = \sum Wx[n] \tag{3.1}$$

$$mx[n] = m * x[n] \tag{3.2}$$

Equation 3.1 is a mass balance equation for each component in the mixture, simply stating that derivative of the mass equals the sum of the different mass flows, flows into the volume taken with a positive sign. Equation 3.2 states that the array $x[n]$ is the the ratio between the partial masses and the total mass. The code for the standard control volume is shown below, as expected it is extended from the `TwoPinDynamic` class. In the equation field the two balance equations are implemented, and three equation for calculating pressure, total mass and the mass ratio array. Initial values for the masses are set using ideal gas law, by specifying initial temperature, pressure and mass ratios. This imposes a restriction on the medium models that can be used. If non ideal gas models are to be implemented, the equations must be revised. Currently all medium models are considered ideal gases, but it can be noted that the partial class `GasPropBase` which is the superclass for all medium models, doesn't assume that the gas is an ideal gas.

```
model ControlVolume
  "Standard control volume,
    mass, mass fraction and energy balance"
  extends VehProLib.Partial.TwoPinDynamic;
  parameter SI.Volume V=1/1000;
  SI.Mass m;
  SI.Mass mx[g.n](
    start=((pInit*V)/(286*TInit)*g.xInit);

equation
```

```
  der(g.u)*m + g.u*(i.W + o.W) =
    i.H + o.H;
  der(mx) = i.Wx + o.Wx;

  m = sum(mx);
  mx = m*g.x;
  p*V = m*g.R*T;
end ControlVolume;
```

As the pressure drops over a restriction in engine can be substantial it is not sufficient to idealize the flow to an incompressible medium. When modelling flow through a restriction, it must be taken into account that the flow through can't reach speeds above the speed of sound. The point when this happens is when the pressure ratio reach a certain value, known as the critical pressure ratio. If the pressure is increased beyond this point, the flow will become choked and the flow will not increase even if the pressure ratio continues to increase. The flow through the restriction is modelled using equation 3.3.

$$\dot{m} = \psi \frac{p_i}{\sqrt{RT}} \tag{3.3}$$

$$\psi = \sqrt{\frac{2\gamma}{\gamma - 1}\left[\left(\frac{p_o}{p_i}\right)^{\left(\frac{2}{\gamma}\right)} - \left(\frac{p_o}{p_i}\right)^{\left(\frac{\gamma+1}{\gamma}\right)}\right]} \tag{3.4}$$

$$\left(\frac{p_o}{p_i}\right)_{critical} = \left(\frac{2}{\gamma + 1}\right)^{\left(\frac{\gamma}{\gamma-1}\right)} \tag{3.5}$$

For pressure ratios smaller than the critical value given by 3.5, the critical pressure ratio is used instead of the actual pressure ratio in 3.4. A complete deduction of these equations can be found in [5]. In order to be able to use these equations with a mixture, a number of modifications must be made. First the correct mass ratio array must be computed, this is done by comparing the pressures sensed by each connector, and as the flow will always be from the higher to lower pressure, the mass fraction array at the highest pressure is chosen. Equation 3.3 is modified to calculate the partial mass flows instead by multiplying with the upstream mass ratio array, and the total mass flow is given by summing $Wx[n]$.

```
model StandardRestriction
  "Standard compressible restriction model"
  extends VehProLib.Partial.TwoPinStatic;
  parameter Real Cd=0.7;
  SI.Area A;
  Real pRatio;
```

```
  Real pRatioPsi;
  Real Psi;
equation

  pRatio = if i.p > o.p then o.p/i.p else i.p/o.p;
  g.T = if i.p > o.p then i.T else o.T;
  g.p = if i.p > o.p then i.p else o.p;
  g.x = if i.p > o.p then i.x else o.x;
  pRatioPsi =
    if pRatio <
      (2/(g.gamma + 1))^((g.gamma/(g.gamma - 1)))
      then
      (2/(g.gamma + 1))^((g.gamma/(g.gamma - 1)))
    else pRatio;
  Psi = LinRoot((2*g.gamma)/(g.gamma - 1)*
    (pRatioPsi^((2/g.gamma)) -
    pRatioPsi^(((g.gamma + 1)/g.gamma))), 0.01);
  i.Wx = if i.p > o.p then
    ((i.p/LinRoot(g.R*i.T, 0.01)*Cd)*A)*Psi
    else
    -((o.p/LinRoot(g.R*o.T, 0.01)*Cd)*A)*Psi*g.x;
  i.H = g.h*i.W;
  i.W = sum(i.Wx);
end StandardRestriction;
```

The code for the standard restriction implemented rather straight forward, the first four equation sets the correct values for the medium model depending on flow direction. The next equation is a check comparing the actual pressure ratio against the critical pressure ratio, compare equation 3.5. The equation yielding the mass flow rate has been modified with two extra variables, one discharge coefficient C_d and the area of restriction A. The discharge coefficient C_d is a correction term used to correct the theoretical flow value for the effects of the Reynolds number. The Reynolds number describes velocity profile of the medium, the flow is considerably smaller near the boundaries of the flow field. A reasonable number for the C_d term is about 0.7 for a standard restriction.

3.5 Included components

Figure 3.1 shows all top libraries included in VehProLib, the libraries contain both models and full examples. The structure is based on the physical properties of the model. For example a model of a wheel would be found in the Chassis library. To be able to maintain an intuitive structure is very important, the user should be able to foresee where

a model can be found before the need arises. The different libraries includes a number of different models which will here be given a brief introduction.

`Chassis` The chassis library contains models for wheels, carbody and so forth. The main uses of the library is to model a complete vehicle, for example in fuel consumption studies. The chassis library is a very basic library, the models included are simple. If a more detailed model is needed one should consider using the VehicleDynamics library [1] instead, where more detailed models are provided.

`Driveline` In this library models of the powertrain can be found, for example gearbox and differential gear. In the package a model of a driver is also provided, the model is used to "drive" the car according to a predetermined drive cycle. This library is also in some aspects a rather basic library, and a more advanced library with about the same capabilities is the Powertrain library [10] from DLR.

`DrivingCycles` Here different drivecycles for complete vehicle studies are provided. As of now only one cycle is implemented, the New European Drivecycle (NEDC). The cycles are implemented in the form of a look up tables.

`Engine` This is the biggest and most important library. There are a number of different engine components stored. When modelling engines there are basically two different approaches, the in-cylinder models and the mean value engine models (MVEM). Both methods has been implemented in the library. The in-cylinder model implemented is a single zone, zero-dimensional model, and depending on the working fluid chosen different combustion models are provided. Single zone means that there is only one zone in the combustion chamber, there is no separation of burned and unburned gases and so forth. The zero-dimensional structure follows naturally since there is only one zone modelled there can be no geometrical interpretation of the zones. Heattransfer is implemented using both equations developed by Woschni [5] and Hohenberg [5], the use of heattransfer is optional so comparing studies can be performed. The combustion is described using the standard Vibe-function. The model of the complete cylinder, `VehProLib.Engine.CylWValves` is implemented in true multi-domain configuration. The fluid equations are separated from the mechanical ones in different subcomponents. In-cylinder models are very detailed but slow to simulate, so in order to reduce simulation time MVEM models are provided.

The MVEM models are typically used in complete vehicle simulations or in the design of control systems. Since all cylinder models share the same interface `VehProLib.Interfaces.Cylinder`, they are easily replaceable, so the complexity of the model can be varied by a flip of a switch, simply inserting the right model in its appropriate place.

`Examples` A number of examples showing how different components can be put to together and simulate different engine configurations.

`Functions` Contains math functions useful to the entire library.

`GasProp` All medium models available for the library is found here. The purpose of this package is to have a number of different models for thermodynamic properties of the fluid available to the user, each with a different levels of complexity. The fluid models are easily replaceable, so the level of complexity can be changed with little effort. This package will be examined in great detail in the Medium models chapter of this thesis.

`HEV` This library contains different models used in simulations of Hybrid and Electrical vehicles. For further information about this library please refer to the master thesis written by Wallén [13].

`Interfaces` This library contains the different interfaces used by VehProLib. An interface can be both a single connector and a model containing several different connectors. Interfaces containing several connectors are for example the `VehProLib.Interfaces.Cylinder` which contains two FlowCut connectors, one rotational and one FlowCut_liquid, this interface is used for models of a cylinder both in-cylinder and MVEM.

`Partial` Almost all partial models in VehProLib can be found here, this is the very foundation of the entire library.

`Tests` When a new model is added to the library they should be tested, reassuring they can function with all other parts of the library. Such tests should be placed in this library. Unfortunately no such tests are available right now, so an effort should be put in to complete and implement some test models.

`Types` The standard collection of units is extended with some units specially needed for the VehProLib library. The definitions of them are found in this section.

Chapter 4

Advanced concepts in Modelica

In this chapter some of the more advanced concepts I have tried will be presented. First an introduction to the replaceable concept in Modelica, and its uses in the VehProLib library. Following that a discussion about different approaches to handling parameters in Modelica, and the implementation in VehProLib. After that a section covering, different ways to implement a Signal Bus. All this will then be exemplified with a walk through of the `MvemTestBench` model, where most of the concepts covered here is implemented.

4.1 Replaceable components

In the previous chapter the concept of inheritance was introduced, and the uses in VehProLib was shown. In order to maximize the reusability of the models a number of different partial classes were constructed, for example the `TwoPinStatic` from which all restrictions are extended from.

This concept will be expanded with another powerful feature of the Modelica language, the possibility to replace an entire model with a subtype of that model. This section will only give a somewhat brief introduction to the concept, adequate to understand how it's used in VehProLib. For a more extensive discussion please refer to [8]. In order to understand the limitations and possibilities of this concept, one must first understand how a subtype is defined in Modelica. At a first glance one might think that a subtype is classified by inheritance in Modelica, but this is not the case, they are only a way to create a subtype relationship. Instead subtypes are defined by that they share the same public components as the superclass. To clarify, two models

exists A and B. If A contains at least all public components of B, then A is said to be a subtype of B. Note, and this is very important, that A must not have inherited B. The concept is illustrated in a number of code fragments of an incomplete circuit shown below.

```
model Circuit
  replaceable Resistor component1(R=100);
  replaceable Inductor component2(L=0.1)
    extends OnePort;
  replaceable Resistor component3(R=200)
    extends OnePort;
  connect(...);
end Circuit;
```

This code fragment defines an incomplete circuit with three different components, all of them is declared as replaceable, meaning that they can be replaced by a subtype of that class. The parameter values specified in the first circuit are retained, and only the new parameters must be specified, ie in this case Temp and C. One can of course also change parameters already specified except if they are declared as final. Then the parameter can no longer be changed by modification, including redeclaration.

```
model AnotherCircuit =
  Circuit(redeclare TempResistor component1(
            Temp=25),
          redeclare Capacitance component3(
            C=0.05));
```

In this circuit two components has been redeclared, component1 and component3 as illustrated by figure 4.1. Component1 is redeclared as a TempResistor, which is a subtype of the Resistor model. The TempResistor shares all the public components of the Resistor, in this case only the value of the resistance R and the two electrical connectors. Note that TempResistor has not inherited the Resistor model, because one need to replace Ohm's law with some other relationship between the current and the voltage. In order to redeclare component3 as a Capacitance one must change the constraining class, in this case it's changed to the OnePort model instead of the Resistor model, when this is done the component can be redeclared as any subtype of the OnePort model.

The equivalent circuit can of cause modelled not using the replaceable concept as shown below.

```
model AnotherCircuit
  TempResistor component1(R=100, Temp=25);
  Inductor component2(L=0.1);
```

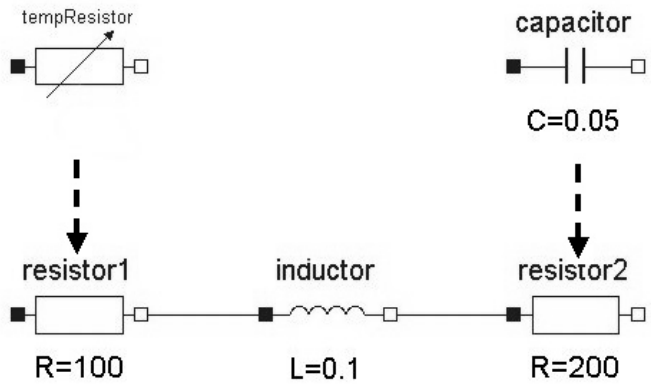

Figure 4.1: Original circuit and effects of the redeclarations.

```
  Capacitance component3(C=0.05);
  connect(...);
end AnotherCircuit;
```

When the Resistor model was replaced by the TempResistor model, this method seemed very appealing, but a number of difficulties arise if one wishes to replace *all* the Resistors models with the TempResistor model. First one must know the total number of components declared as replaceable, and one must know the name of every single component one wishes to redeclare. The solution to this problem is to take the redeclaration to a higher level. Instead of redeclaring every single component, one could instead redeclare the model from which the resistors are declared. In the next example a replaceable model called MyResistor is created, and the model MyResistor is declared to be a model of a resistor.

```
model Circuit
  replaceable model MyResistor = Resistor;
  MyResistor component1;
  Inductor component2;
  MyResistor component3;
  connect(...);
end Circuit;
```

The two components declared as MyResistor will in this case be declared as resistors, as indicated by the first line of the code fragment.

If one wishes to redeclare the Resistors as TempResistors this is simply done by redeclaring the model MyResistor to a model of a TempResistor, see figure 4.2 for an graphical illustration of the process.

```
model AnotherCircuit =
  Circuit(
    redeclare model MyResistor = TempResistor);
```

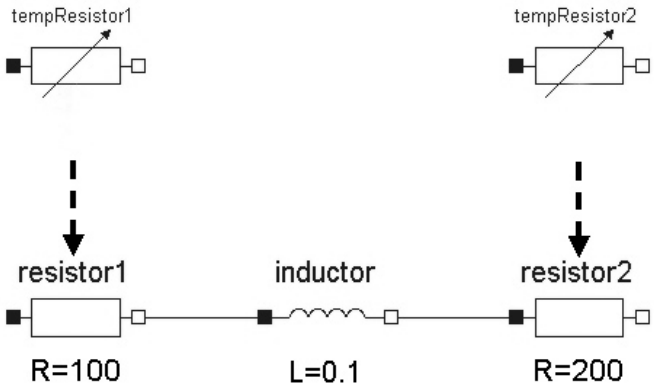

Figure 4.2: Original circuit and effects of the redeclarations.

The equivalent model of the circuit not using any replaceable components is modelled like this.

```
model AnotherCircuit
  TempResistor component1;
  Inductor component2;
  TempResistor component3;
  connect(...);
end AnotherCircuit;
```

One can even replace entire packages at the time, this is not used anywhere in VehProLib, so this will not be covered here. The interested reader can look in the Modelica Media library, when it becomes publicly available, where this is used extensively.

The main uses of the possibility to replace a component is probably in creating models for studying specific effects of a certain component. This is found in a number of models in VehProLib. One example is to

enable the study of the effects from heattransfer from the cylinder. One model with an adiabatic (no heattransfer) cylinder model is created, then one simply replaces that model with a model with heattransfer. The advantages of this approach is that there is no need to write two virtually identical models, instead one model is created and only the parts being redeclared is needed in the second model. Compare the two code fragments above, the one using replaceable and the one not using it.

The point of replacing an entire model instead of replacing components is not so easily understood. If this method is to be considered, a component must exist in several places in a model and one must wish to replace all of them at the same time. The best example of such situation is the fluid models in the VehProLib, a model of the fluid exist in almost every component and when one wishes to redeclare the fluid model it's convenient to do so at one place instead of being forced to use the standard redeclare in every model containing a model of the fluid. How this is done in practice will be shown in the last chapter of this thesis.

4.2 Handling parameters

In the models shown so far the parameters has been set in the declaration section of the model, but as models get bigger and more complex this is no longer a satisfactory way of handling the parameters settings. When a large and complex model is created a need to collect all the parameters in one place arises. The first reason for this is the need to quickly be able to view all the parameters at the same time, thus eliminating the need to open and view several different submodels of the larger model. Secondly, in a model a parameter can be used by several different components and there must be a way to ensure that they are always set to the same value. This is the most important reason to collect parameters in one place. Another important reason is to create a more user friendly environment, if one wishes to change a parameter, it is much easier just to change it at single place instead of being forced to remember where the parameter is actually used. Finally it should be possible to replace the parameters with a different set of parameter, thus enabling a way to create a library of different parameter sets representing, for example different geometrical layouts of a cylinder.

Two ways of creating such parameter sets will be presented here, first the approach used by Tiller in [12], and then by using one of the restricted classes of Modelica, namely **record**.

The main idea behind Tiller's way of representing parameters, is to create two models, one generic model for the physical system to be

modelled and one model containing the parameters needed. To communicate between the models two specially designed connectors are used. One is input only type connector, meaning it can only receive information, the other one is naturally an output connector, sending information to the input connector. The code fragments below show a straight forward implementation of two connectors needed. In this case only two parameters can be transmitted, but this can of course be extended to any number.

```
connector EngineGeometryRequired
  "Connector to receive geometry information"
  input SI.Length bore "Engine bore";
  input SI.Length stroke "Engine stroke";
end EngineGeometryRequired;

connector EngineGeometryProvided
  "Connector to send geometry information"
  output SI.Length bore "Engine bore";
  output SI.Length stroke "Engine stroke";
end EngineGeometryProvided;
```

To access the parameters one simply include an input connector, give it a name, for example geometry. Then parameters can be access using the dot notation like this, *geometry.bore*, would return the value of the component bore on the connector. In the other end, where one wishes to send parameters to the connector, the method is very similar, simply replace the input connector with an output connector as shown in the code fragment below.

```
model GeometrySource
  EngineGeometryProvided geometry;
  parameter SI.Length bore = 0.09;
  parameter SI.Length stroke = 0.09;
equation
  geometry.bore = bore;
  geometry.stroke = stroke;
end GeometrySource;
```

The model GeometrySource should always be declared replaceable when it's used, so it can easily be replaced by any other set of parameters. The advantages of this approach is that since the parameters are stored in a ordinary model, one can write equations in the equation field. This can be used to check the values of the parameters before they are accepted, for example one can set an upper and lower limit of acceptable values. On the downside, two problems can be identified. The first problem is that one can't use the models without providing any parameters, there can be no default values stored in the models, because

the parameters are not declared in the model, they are only provided via the connector. A second problem is the need to write a lot of connect statements, this will in turn reduce that readability of the code, as connect statements are no longer reserved for actual physical flows.

To address these shortcomings the restricted class **record** was introduced. The limitations that apply to the use of the restricted class **record** is that it may not contain any equations and it can not be used in connections. When one wishes to use a parameter record it is simply included in the model, and its components is referred to using the usual dot notation. The code for the parameter set using a **record** is very similar to one used in GeometrySource above, as can be seen below.

```
record EngineGeometry
  parameter SI.Length bore;
  parameter SI.Length stroke;
end EngineGeometry;
```

The common way to implement a library of parameters sets is to declare a empty record, only specifying which parameters it contains. A new record is then created by extending from the empty record and providing the values for the parameters.

```
record SAAB95
  extend EngineGeometry(
    bore = 0.09,
    stroke = 0.09);
end SAAB95;
```

To change the record in a model is done in the same way as any other parameters is changed, on the line the model is declared, simply write which parameter record to use within parenthesis. Compare with for example the code fragments from the previous section.

```
model car
  Engine.MvemEngine engine(data = SAAB95());
  ...
end car;
```

For this to work a record of the correct type must exist in the model. In this case, in the MvemEngine model one should include the record EngineGeometry. The advantages of using parameters records are basically the disadvantages pointed out using Tiller's method using connectors. Here there is no need to write connect statement, so all connect statement represent actual flows. The parameters can be given default values, so testing is simpler as one can test a model without including additional models. On the downside is that there is no possibility to

check the parameters before they are accepted as there is not allowed to be any equations in the record class.

The conclusion of this section is that records is the recommended way of storing parameters, as they give a clearer view of the model, not clouding it with extra connect statements. Some Dymola specific advantages exist also, there is a much better support for handling record in the user interface than using models with parameters. The method using records is also recommended by Modelica so this should also be a strong argument for using it.

4.3 The Signal Bus

Signal connections in technical system can quickly get very complicated and hard to overview, to remedy this a bus system is often introduced collecting all signals to a common signal bus system. This procedure can be mimicked with Modelica in a number of ways. Two different ways will be presented here, first the implementation used in the PowerTrain library created by Dymola, the second implementation is used by Ford in their in-house engine modelling libraries.

The signal bus system suggested by Dymola in [4] is based around a specially designed connector which acts like a bus. To represent this graphicly the connector has been drawn out to form a oblong rectangle, this is of course of no importance but only a way of providing a resemblance with an actual bus. If one tries to use a connector as a bus right away, a problem will quickly be identified, it's not possible to connect to a single variable on the connector. Every model that uses the bus must know the entire bus and the only way of achieving this is to include a bus connector, which means that none of the standard models can be used. So for the bus connector to be of any practical use, a single variable connector must be constructed. To solve this problem three single variable connectors are provided, one for Real, Integer and Boolean type of variables.

```
connector RealPort
  replaceable type SignalType = Real;
  extends SignalType;
end RealPort;
```

The use of the replaceable signal type is not strictly necessary. The purpose of this is to allow a redeclaration of type so the signal will be given its proper unit. In the code fragment below this is used to redeclare the type of the signal vehicleSpeed from Real to Velocity, and by doing so the variable will be displayed with correct unit.

```
connector Bus
  RealPort vehicleSpeed(redeclare type SignalType
    = SI.Velocity);
  IntegerPort Gear;
  ...
end Bus;
```

For every signal to be transmitted on the bus a connector of the appropriate type is added. Connection can now be achieved both to the entire bus system using another bus connector or to a single variable using one of the single variable connectors. One small problem is left to be tackled, most models have an ordinary OutPort or InPort to be used in connections. They can not be connected directly to the bus, so a bus adaptor is needed to send and receive signals from the bus. All the models used in conjunction with the bus, can be found in `Modelica.Blocks.Interfaces` of the Modelica Standard Library.

To summarize, to send a real variable to the bus requires two additional components, first a standard OutPort, then a bus adaptor for the conversion to a RealPort connector. The adaptor is then connected to the single variable connector on the bus connector. To send real variable from the bus would require two more components, an InPort and an separate adaptor. The advantages of using this implementation is that it's easy to add signals, one must only add an extra connector on the bus, and the extension is complete. The negative side, has already been introduced, it requires a number of extra models to work and all of them must be connected using connect statements. Remember that the purpose of introducing the bus was to collect all the signals, reducing the number connection lines needed. In this only a partial success is achieved, the signals can be transmitted on the bus between models, but inside the model the bus must be split up to access the individual signals. One problem arises when replaceable components are introduced in models. Different components may require a different set of control signals and thus a different bus configuration. This problem can not be easily handled using this version of the bus, one must manually reconfigure the bus, adding or removing signals. Another solution is to maintain the bus and use a separate model to put out dummy values on the part of the bus not currently in use.

The implementation presented by Tiller in [7] tries to address this problem. To avoid the problem, something called the SignalBus idiom is introduced. In this approach, all signals associated with each subsystem are grouped together on a "master" bus on the top-level of the model. The components only need to be aware of the specific signals they require and not all the signals on the master bus.

```
connector SignalBus
```

```
  (graphics)
end SignalBus;
```

First one creates an empty connector, and give it a special graphical notation so it can easily be identified in the models. This is not necessary, only a help to the user.

```
model LambdaControl
  outer ControlBus eng_control_bus;
  ...
protected
  connector ControlBus
    extends SignalBus;
    Real Lambda;
    Real LambdaControl;
  end ControlBus;
end LambdaContol;
```

Inside each component requiring control signals, in this case the lambda control, a specific bus type for that component is declared. The bus is declared as protected, clearly showing it's for internal use only. The bus definition should include only the signals required by the component. This bus can be instantiated with the outer qualifier, the name of the instance should be that of the master bus in the top-level model, where the signals reside.

```
model Car;
  inner EngineMaster eng_control_bus;
  ...
  EngineControlUnit ECU "Contains Lambdacontrol";
protected
  connector EngineMaster
    extends SignalBus;
    Real Lambda;
    Real LambdaControl;
    Real IdleControl;
    Real EngineSpeed;
  end EngineMaster;
end Car;
```

The top-level model, should contain a master bus type containing the union of all the subsystems buses and a inner instance must be declared. This implementation has the following advantages over other bus implementation, it avoids the necessity to place connectors at each level. This is a big advantage since every time a component is changed it might require modifications to connectors at every level of the model. The outer bus must only be a subtype of the matching inner bus, the

component must only declare the signals it's interested in. In the code fragment, this is clearly seen, the lambda control only requires two signals, but the master bus contains four signals. The signals are accessed in the same way as in the previous examples, using the standard dot notation, for example eng_control_bus.Lambda. One disadvantage using this implementation is the more complex way of adding new signals, here one needs to add the signal at two different locations. First in component where the new signal is needed, and then in the top-level model.

In VehProLib the latter implementation has been implemented, reasons for this is that it requires no connectors at the sub-levels. This results in a clearer view of the model. The same reason as the record was chosen as the primary way of handling parameters. The second thing that influenced the choice was the need to take care of an under used bus, which requires a special model in the first implementation putting out dummy values on not used signals, this is not a problem in the latter implementation since the components only declare the signals they are interested in.

Some final remarks about the use of buses in Modelica. It should be possible to simulate for example an engine without being forced to use the bus. The models should not require the presence of the control systems in order to function. On the other hand, the models should automatically start using a control system if the connection to control bus is detected.

```
model MvemCylinder
  if cardinality(eng_control_bus) == 0 then
    eng_control_bus.LambdaControl = 1;
  end if;
  ...
end MvemCylinder;
```

The code fragment shows how this can be achieved using the **cardinality** keyword, cardinality(eng_control_bus) returns the number of occurrences of eng_control_bus in connect statements. So if the cardinality statement returns zero, no connections has been made on the bus, and no control system can be present. The model then mimics the control system, by setting an appropriate value on the signals effected by the control system. Unfortunately this method can not detect if the right control system has been connected, so this is left to the user, to see to that the control system meets the the demands of the model.

4.4 The MvemTestBench Model

This example will serve as an illustration of most of the concept covered in this chapter. The model is `MvemTestBench` and it's a model assembly for testing MVEM engines. The model consists of six different parts, as can be seen in figure 4.3. The engine will run at a constant speed set by the dynamometer and the signal source can be used to alter the throttle setting. The engine control system has two functions implemented, first it provides lambda control for the engine, secondly there is an idle running function. The lambda control influences the amount of injected fuel, always maintaining a lambda value close to one. The idle running steps in and controls the throttle if the throttle setting is to low to maintain idle running.

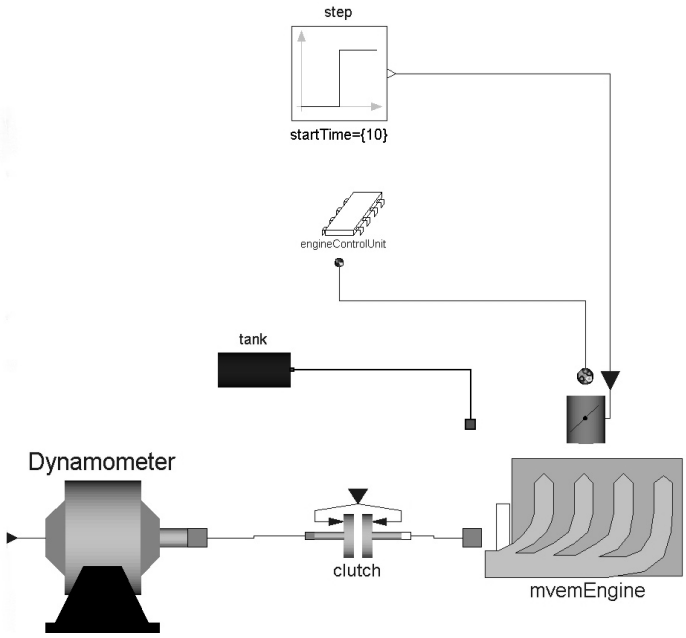

Figure 4.3: The MvemTestBench model.

The engine and the control system communicate on a bus implemented following the lines laid out in the previous section. The bus can declare a total of four signals. Components in the engine will put out the current lambda value and the speed of the engine, the control system will return two control signals, one effecting the throttle and one the fuel injection. The engine control system can be removed without any need to modify the engine, as all effected components in the engine are implemented using cardinality statement as illustrated

in the previous section.

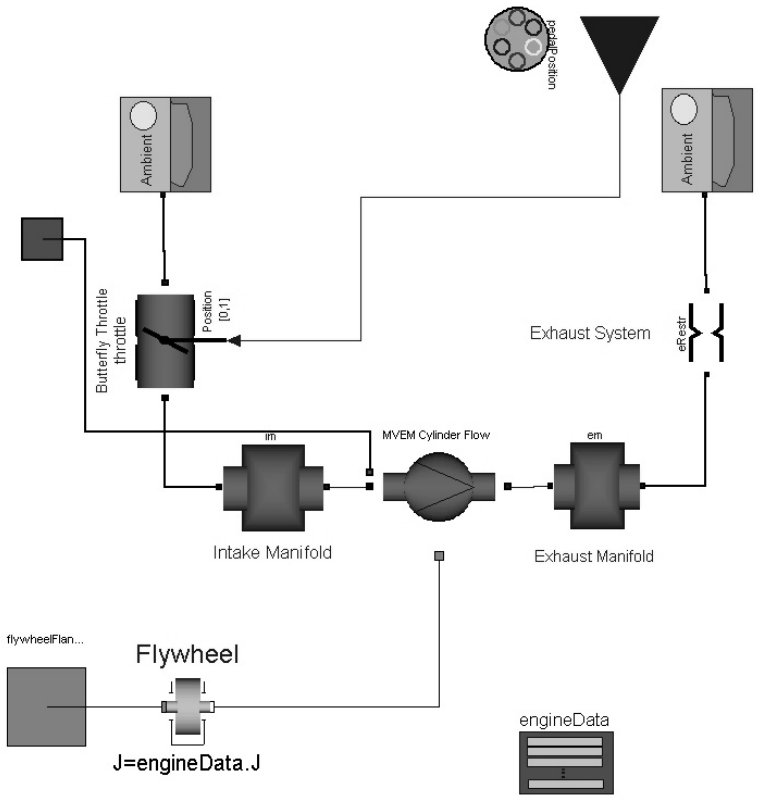

Figure 4.4: Layout of MvemEngine model. The connector for the signal bus is visible in top of the figure and the parameter record can be seen in the bottom.

The layout of the MVEM engine is shown in figure 4.4. All parameters used to specify the engine is collected in a record. The record is declared replaceable so the engine can easily be reconfigured. Two possible ways exist, first one can use one of the available engine record, or one can change the individual parameters of the record, and by doing so creating a new engine. One more choice is available to the user, a number of different medium models is implemented in VehProLib. The medium models range from a very simple fluid model with constant c_p to multi-component fluids. The user is free to use any of these models and the choice is not affected by any other choices made. All choices are perpendicular to each other.

Chapter 5

Medium models

Different engine simulations require a different level of detail. This is achieved by providing the user with a number of different models to choose from, for example in-cylinder and mean value engine models. This should also be possible when calculating the thermodynamic properties of the fluid. One should be able to choose from a number of models describing the fluid. This concept was introduced by Batteh, et al. in [2].

5.1 The replaceable Medium Model

The purpose of the medium models is to calculate the thermodynamic properties of the content of for example a control volume. The span of the complexity can be considerable, in one case one can treat the medium flowing through the engine as single-component gas with fixed c_p, but in other cases one perhaps wishes to use a multi-component gas which would require tracking of several chemical species. At first glance, there would seem like one would be forced to have a different set of component models for each medium model. This would result in a totaly unacceptable situation. A deeper look into the problem would however reveal that the thermodynamic properties of the medium are orthogonal to the equations of the different processes inside the engine. So if one formulates the models correctly the choice of medium can be made independently of the components used in the model.

This is achieved by creating a partial model called `GasPropBase` which describes the functionality the different medium models must possess. The `GasPropBase` model consists of a number of parameters and variables each medium model must have, and some basic relations that are valid for all models. The source code for the model can be found in Appendix A. It can be noted that the `GasPropBase` contains

no assumptions about ideal gases. The different medium models in the library are then extended from the `GasPropBase` model.

Because the medium-specific information is wholly contained within in a replaceable model, the working fluid specification can be changed at a single place, preferably at the top-level of the model. This functionality is achieved by using the possibility to declare replaceable models, see the second example of how to use replaceable in chapter four. If one where to use that approach straight away, it would not yield any good result. In that example the model was changed from within the component itself, but in this case one would like to be able to change the model from outside the different components. This requires the introduction of the inner/outer prefix, inner/outer is used in Modelica to create global variables. So if one wishes to create a global variable one uses the inner prefix when declaring the variable at the top-level, this variable can then be accessed by any component within by simply declaring a variable with the same name using the outer prefix.

So in every component that requires a fluid component, a model called Gasmedium is declared, the model is declared using the outer prefix. The Gasmedium model is then used to declare a fluid component called g.

```
outer model Gasmedium = GasPropBase;
  Gasmedium g;
```

Then at the top-level a replaceable model with the same name is declared, this time using the inner prefix instead. The result will be that the Gasmedium model will be the same through out the entire model, so if it is changed at the top-level it will also be changed in the under-lying components.

```
inner replaceable model Gasmedium =
  PerfectGas extends GasPropBase;
```

The Gasmedium model can be changed into any medium model extended from the `GasPropBase` model, in this case a model called `PerfectGas` is chosen. This choice is then reflected down through out the model hierarchy.

5.2 Implemented medium models

The implemented medium models in VehProLib can be group together in three distinct groups. First single-component fluids, secondly multi-component fluids and finally models of real substances, such as hydrogen and oxygen. The models of real substances are then used to build up more advanced multi-component fluids. In this section the

three groups will be presented in order, starting with simple single-component fluid. The structure of the different medium models is illustrated by figure 5.1, please note that this figure is by no means complete, it only serves as an illustration of the structure. There is a number of models implemented not present in the figure. On the top of

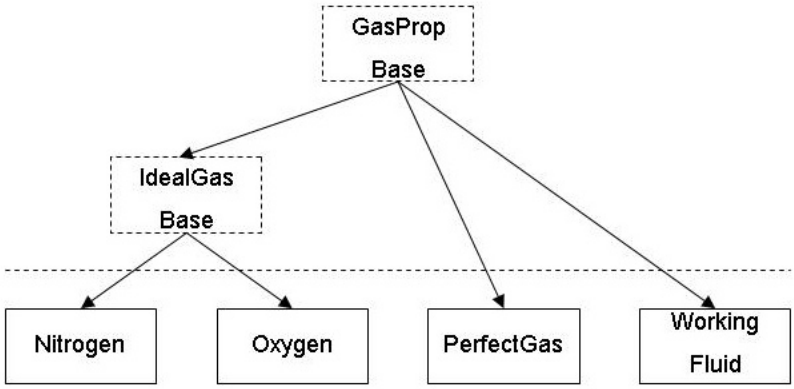

Figure 5.1: Structure of Medium models, partial models are dashed.

the figure the superclass `GasPropbase` is found, this class is then used to create all medium models. The second partial model `IdealGasBase` is used to create models of actual substances as shown in the figure. The other two models shown represent two existing medium models. The `PerfectGas` is a simple single-component fluid, whereas the `Working-Fluid` is a complex multi-component fluid model.

5.2.1 Single-component fluids

Two single-components are implemented in VehProLib as of now, one very simple with constant c_p and one with a linear temperature dependent c_p. The source code for the first model is shown below, a number of things can be noted.

```
model PerfectGas
  "Very simple gasmodel, cp(T) = cp0"
  extends VehProLib.Partial.GasPropBase(
    MolarMass=29/1000,
    final n=1,
    final xInit={1},
    final xBurned={1});
```

```
  parameter SI.SpecificHeatCapacityAtConstantPressure
    SpecHeat_p=1200
    "Specific heat capacity at constant pressure";
equation
  c_p = SpecHeat_p;
  p*v = R*T;
  c_v = c_p - R;
  h = c_p*T;
end PerfectGas;
```

First the model is extended from `GasPropBase` as expected, and four parameters is given its values. These four are the $molarmass$, n, $xInit$ and $xBurned$. $Molarmass$ needs no further explanation, n is the number of components in the fluid, $xInit$ sets the initial values for the mass fraction array (since n was equal to one in this case, $xInit$ must also be equal to one), $xBurned$ yields the new mass fraction array after fluid has been burned when using a MVEM model. In the equation field four equations are found, first an equation to fix c_p to the value given by $SpecHeat\_p$. The second is the ideal gas law, then the relationship between the specific heats, and last an equation for calculating the mass specific enthalpy. The things that can be improved with this implementation is a more advanced method to calculate the value of c_p, instead of using a fix value, one can use some sort of mathematical relation between temperature and c_p as mentioned above.

5.2.2 Multi-component fluids

Two multi-component fluids are available to the user, the source code for the simpler one of them is found in Appendix B. The code is similar to the one used for single-component fluids, n, $xInit$ and $xBurned$ are all found in this model too. The value of n is here greater than one, so $xInit$ and $xBurned$ are true vectors. Following that are the different fluid components declared, in the example three components are used, but this can be expanded to any number. The different components are single-component fluids implemented following the lines laid out in the previous section. In the source code it can be noted that the `PerfectGas` model is used for one of the components. The galine_position parameter is used to point out the position of the gasoline in the mass fraction array, the parameter is used when fuel is injected to the fluid. All the components senses the same pressure and temperature, and will calculate their own thermodynamic properties. The thermodynamic properties of the composite fluid is then calculated by taking the mean of the different components weighted by the mass fraction array, see the equation field of the source code. Multi-component fluids are also able to calculate their own relative fuel/air ratio, this is used in combustion

processes. The use of multi-component fluids has a number advantages over single-component fluids, the combustion process can be more accurately modelled, one component can be modelled to resemble air and another one to resemble exhaust. A number of phenomenon can be captured, for example the residual gas fraction in the cylinder.

5.2.3 Substance models

Substance models are used in the more advanced multi-component fluid models, such as the `WorkingFluid` model. The `WorkingFluid` model has seven different substances included. The substance models are extended from the `IdealGasBase` model. All the equations are found in the `IdealGasBase` model, the only thing effected by the extension is from which parameter record the substance specific data is read. The source code for `IdealGasBase` can be found in Appendix C. Models of gasoline in vapor phase require a different set of equations, so those models don't use the `IdealGasBase` as a superclass. The thermodynamic properties are computed using the formulaes and data found in [5]. In the library seven different substances are provided, six models of different gases and one of gasoline vapor.

5.2.4 Comparison between different fluid models

In order to evaluate the differences in performance of the different fluid models in the library, the model `ControlVolumeTest` can be used. The test model will increase the pressure by opening a valve after two seconds. The pressure will then be reduced in two steps at four and eigth seconds. Things to consider are the time required to complete the simulation and the differences in the variables that can be observed. In figure 5.2, the result from two simulation are shown. The model `PerfectGas` is the simplest fluid model in the library. It is a single-component fluid with c_p fixed to a constant value. The other model is called `WorkingFluid`, this is the most advanced model implemented. Seven different components are included in the model (O_2,N_2,H_2O,CO_2,CO,H_2,CH). The thermodynamic properties are calculated using data from NASA tables. The time required to complete the simulation is 0.531 seconds for the `WorkingFluid` model and 0.061 seconds when using the `PerfectGas` model. The more complex fluid is, as can be seen about ten times slower to simulate. In the top figure of figure 5.2, the pressure of the fluid inside the control volume is plotted. No visible differences can be seen in the figure. However some differences can be detected during the transients. The response for the `PerfectGas` is a little bit slower than the `WorkingFluid`. In the bottom figure, the pressure is instead plotted. Here is the differences more apparent, the `PerfectGas` will yield a lower temperature when the temperature is above 293

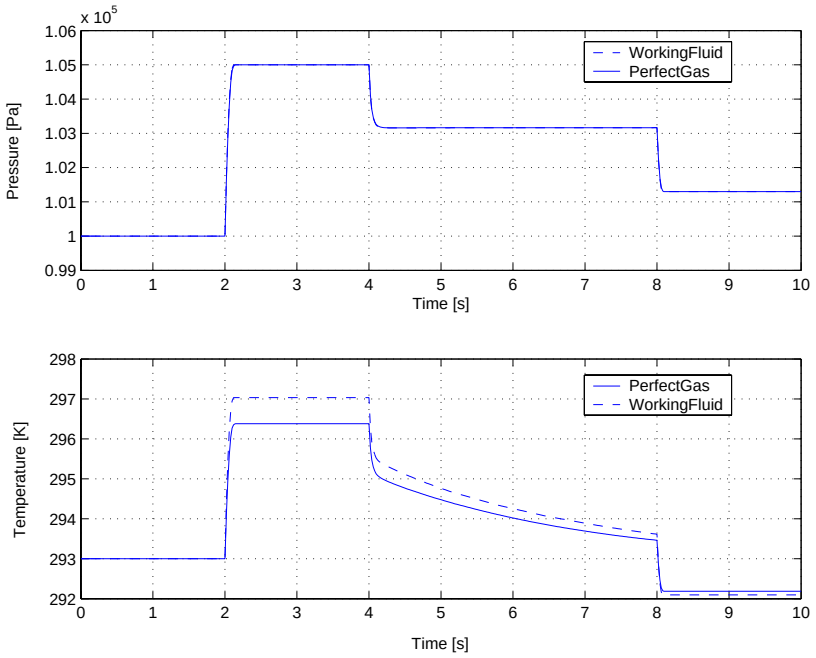

Figure 5.2: Comparison between two different fluid models.

K and vice versa. The `WorkingFluid` model is very accurate and the other models should try to capture the behavior of this model. The two figures show that the `PerfectGas` model capture the overall behavior of the system very well and the simulation time is reduced considerably. However will the ability for the `PerfectGas` model to produce good results decrease as the temperature span increases.

5.3 Limitations on the uses of medium models

The purpose of implementing a number of different medium models, was to allow the user to chose the level of complexity in the simulations. One of the goals was to be able to chose the fluid independently from the components used in the model. Unfortunately this is not completely true in the current implementation. When MVEM models are used all the information required by the fluid is completely contained within the fluid, so the choice of fluid model can be made totally independent. In in-cylinder models, this is not the case. Problems will arise in a number of situations, first in the injector model. If single-

component fluid is used with the injector model, it's not possible to inject fuel and simultaneous track the amount of fuel injected. The reason for this is that the fuel will be added to the only present component and it can not be singled out at a later stage. Problems will also arise in the combustion chamber when using in-cylinder models. The combustion chamber is basically an enhanced control volume, where the mass balance equation (3.1), is replaced with an equation that allows for mass flow between different components. If a single-component fluid is used there can be no flow between components, but if a multi-component fluid this must be possible. So the mass balance equation must be changed to fit each of the different possible choices. The model of the combustion chamber used with single-component fluids is found in Appendix D. This should be compared with the combustion chamber model used with the most advanced fluid found in Appendix E. As can be seen the level complexity regarding the mass balance equation varies substantially, from a single line to over ten lines in the code. This is one of the major shortcoming of the library, and an effort should be made to fix this problem.

5.4 The CylWValves model

This example will show some of the results that can be obtained using in-cylinder models combined with a multi-component fluid. For this simulation a model called `CylWValves` is used. This is an in-cylinder model, modelled using a true multi-domain configuration. The layout of the model can be seen in figure 5.3. In the model are three different domains represented, mechanical, fluid and heat transfer. The transition between mechanical and fluid domains is located in the piston model. The pressure of the fluid is converted to a force acting on the connecting rod. The transition from fluid to heat transfer is located in the combustion chamber of the model. The heat is transported to a model of the cylinder wall, which will be heated by the combustions. All the parameters are collected in a record called `cylinderParameters`. Parameters available to the user are, for example geometrical layout of cylinder. Valve timing for both inlet and exhaust valve can be varied by changing the appropriate parameters. The result from a simulation using a fluid with three components is shown in figure 5.4. The three components of the fluid are air, gasoline vapor and exhaust gas. The figure shows two consecutive combustions in the chamber, the combustions occur at approximately 0.735 and 0.775 seconds into the simulation. The four variables in the figure are the total and partial masses of the ingoing components. The total mass will increase during the intake stroke, remain constant during compression and expansion strokes and decrease during the exhaust stroke. During the intake stroke gasoline

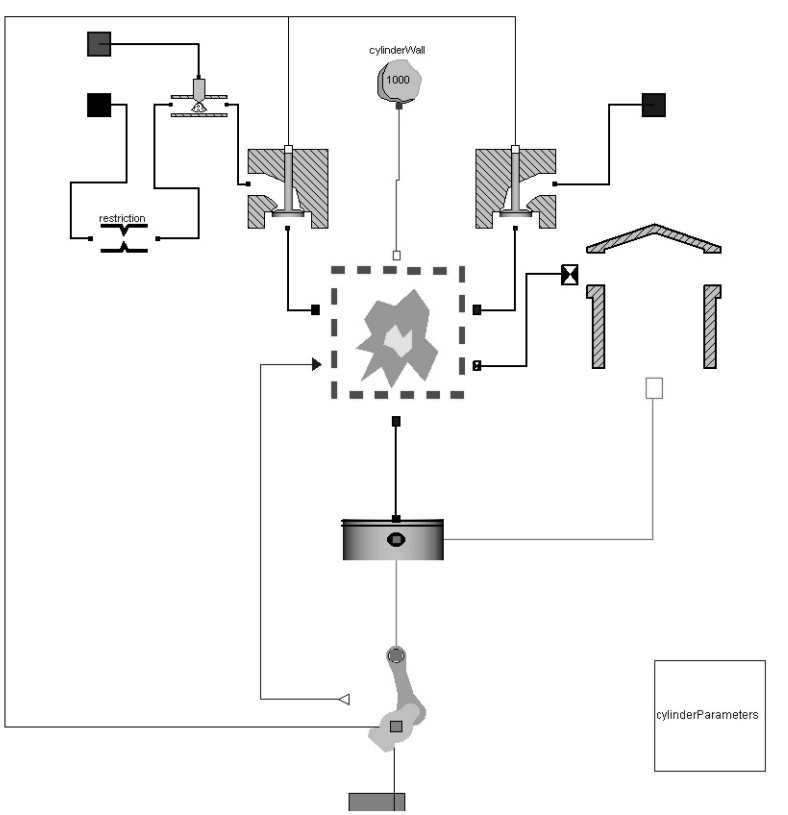

Figure 5.3: Complete cylinder model with valves.

and air will be inducted into the cylinder. This can be clearly seen in the figure at about 0.72 and 0.76 seconds. During the intake stroke the residual gas can also be seen in the figure. During the combustion the air and gasoline will be burned and form exhaust. The first combustion is lean, there is not sufficient gasoline to consume all the air in the cylinder. This will result in that air is still present in the cylinder after the combustion is complete. This can be seen at 0.74 seconds. The second combustion is stoichiometric, all the air is consumed in the combustion.

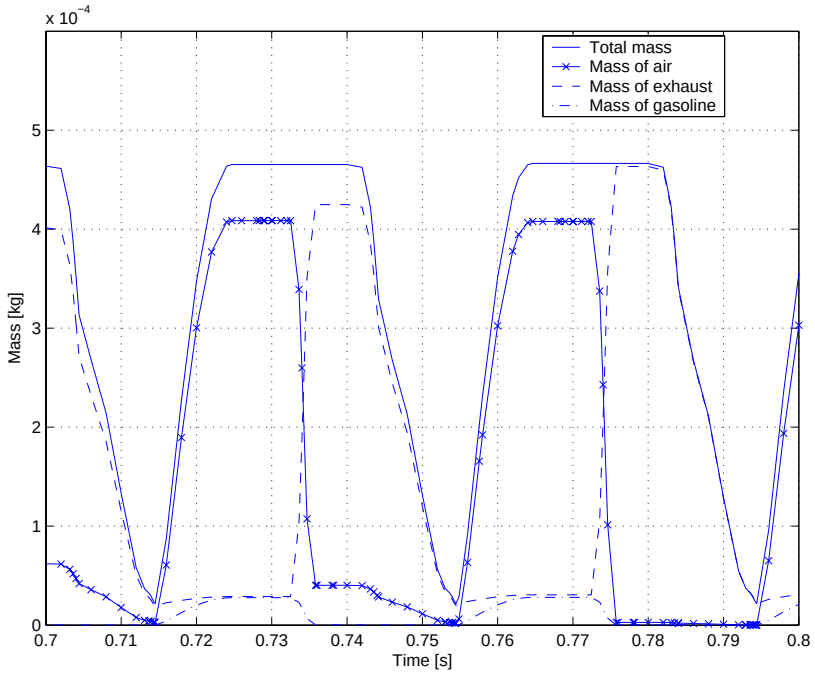

Figure 5.4: Gas exchange process inside the cylinder. Two consecutive cobustions are shown. The first is lean, air is present in the chamber after the combustion is complete. The second is stoichiometric, all the air is consumed in the combustion.

Chapter 6

Concluding comments

In this last chapter some of the conclusions drawn in the previous chapter are summarized and some recommendations for future work are given.

6.1 Summary and Recommendations

In chapter one, the strengths of Modelica were shown, those are the possibility to write equations in the form they are found and the ability to collect equations to form models, which can be reused. The example in the last section of the chapter illustrates these points nicely. Chapter two continued with the introduction of the VehProLib, and the discussion about the design of connectors and partial models. These are the two most important design issues, the connector because its wide spread use, almost every component in the library uses the connector. The partial models dictate most of the capabilities in terms of the ability to use replaceable components. One more consideration should be pointed out, the level of fragmentation, each partial model should add an distinct ability. The two following chapters deal with more advanced issues of VehProLib.

In chapter four an introduction to the replaceable concept was provided along with a discussion about parameters and signal buses. The recommended way to handle parameters is to use parameter records. Reasons for this are, increased readability of the code and no need to provide extra connectors. One drawback can be noted when using records, the values of the parameters can not be tested before they are accepted. The signal bus suggested by Tiller was choosen before the one provided by Modelica for mainly two reasons. First it is not necessary to use adaptors to access the signals on the bus thus increasing the readability of the code. Secondly and probably the most important

reason is the large problems associated with replaceable components when using the standard bus. Those problems were the need modify the connector to fit the needs of the new model and to take care of left over signals on the bus. One negative aspect of the selected bus can be pointed out, when new signals are to be added modifications must be made at numerous positions in the code instead of at just one place.

The Medium models chapter deals with the need to be able to choose from a number of different models describing the thermodynamic properties of the fluid. The models provided range from very simple single-component fluids to multi-component fluids. The resons for adding multi-component fluids to the library were, first the possibility to represent the fluid using one component for unburned fluid and another for burned. Both residual gas fraction and lean combustions phenomenons can be captured as shown in section 5.4. When replacable fluid models were introduced one problem was identified, it is not possible to choose the fluid model independently from the component models when working with in-cylinder models, as pointed out in the last part of chapter five. The main reason for this is the need to modify the massbalance equation in the combustion chamber to fit the different fluid models.

It can be concluded that Modelica met the test, it was possible to implement such advanced concepts as a signal bus in a feasible way. All goals except one were achieved, the thing left to be tackled is the problem with the fluid models.

6.2 Future work

The library is still in need of development, some of the areas that needs to be addressed are:

- Implement and incorporate more engine components, turbo charger models are for example completely missing. Exhaust gas recycling could be added with little effort and take great advantage of the multi-component fluid models.
- Test models, all models to be added to the library should be tested to ensure that they work with all other parts of the library. Today no such test models are implemented, this is a very important issue to be solved
- The fluid models, the problem with the incomplete separation of fluid and component models must be solved.
- Evaluate the performance of the Modelica Media, and see if it should replace the current fluid models in the library.

References

[1] Johan Andreasson. VehicleDynamics library. In *3rd International Modelica Conference, Proceedings*, pages 11–18, Linköping, Sweden, November 2003.

[2] John Batteh, Michael Tiller, and Charles Newman. Simulating of Engine Systems in Modelica. In *3rd International Modelica Conference, Proceedings*, pages 139–148, Linköping, Sweden, November 2003.

[3] Lars Eriksson. VehProLib - Vehicle Propulsion Library. Library development. In *3rd International Modelica Conference, Proceedings*, pages 249–256, Linköping, Sweden, November 2003.

[4] German Aerospace Center (DLR), Oberpfaffenhofen, Germany. *PowerTrain Library Tutorial*, 1.0 edition, December 2002.

[5] John B. Heywood. *Internal Combustion Engine Fundamentals*. Automotive Technology Series. McGraw-Hill International Editions, New York, U.S., 1 edition, 1988.

[6] Lennart Ljung and Torkel Glad. Modellbygge och Simulering TSRT 62. Linköping, Sweden, 2002. Course material, Linköpings Universitet, Sweden.

[7] Tiller M., Tobler W.E., and Kuang M. Evaluating Engine Contributions to HEV Driveline Vibrations. In *2nd International Modelica Conference, Proceedings*, pages 19–24, Oberpfaffenhofen, Germany, March 2002.

[8] Modelica Association. *Modelica - A Unified Object-Oriented Language for Physical Systems Modeling Language Specification*, 2.0 edition, July 2002.

[9] Eva-Lena Lengquist Sandelin, Susanna Monemar, Peter Fritzon, and Peter Bunus. DrModelica - An Interactive Tutoring Enviroment for Modelica. In *3rd International Modelica Conference, Proceedings*, pages 125–136, Linköping, Sweden, November 2003.

[10] Christian Schweiger. PowerTrain Library 1.0. In *Modelica Automotive Workshop*, Dearborn, USA, November 2002.

[11] Alexander Stankovic. Modelling of air flows in automotive engines using modelica. Master's thesis, Linköpings Universitet, SE-581 83 Linköping, 2000.

[12] M. Tiller. *Introduction to Physical Modeling with Modelica*. Kluwer Academic Publishers, 2001.

[13] Johanna Wallén. Modelling of Components for Conventional Car and Hybrid Electric Vehicle in Modelica. Master's thesis, Linköpings Universitet, SE-581 83 Linköping, 2004.

Appendix A

The GasPropBase model

```
partial model GasPropBase
  "Base class for gas properties"
  package SI = Modelica.SIunits;
  SI.Pressure p(
    nominal=100000,
    min=5000,
    max=20000000) "Gas pressure";
  SI.SpecificVolume v(nominal=0.5, start=1)
    "Specific volume";
  SI.Temperature T(
    start=400,
    nominal=500,
    min=200,
    max=5000) "Gas temperature";
  SI.Density rho(nominal=2) "Gas density";
  parameter Integer n=1 "Number of gas components";
  parameter SI.MassFraction xInit[n]
    "Initial mass fractions of gas";
  parameter SI.MassFraction xBurned[n]
    "Mass fractions after burn complete";
  SI.MassFraction x[n](
    max=1,
    min=0,
    nominal=0.5) "Mass Fraction";
  SI.SpecificEnthalpy h(
    min=-1.0e8,
    max=1.e8,
    nominal=1.e6) "Mass specific enthalpy";
  SI.SpecificEnergy u(
    min=-1.0e8,
```

```
    max=1.e8,
    nominal=1.e6) "Mass specific internal energy";
  Real R(final unit="J/(kg.K)", nominal=200)
    "Gas constant";
  SI.MolarMass MolarMass(nominal=3/100)
    "Molar mass";
  SI.SpecificHeatCapacityAtConstantPressure c_p(
    nominal=1200)
    "Specific heat capacity at constant pressure";
  SI.SpecificHeatCapacityAtConstantVolume c_v(
    nominal=1000)
    "Specific heat capacity at constant volume";
  SI.RatioOfSpecificHeatCapacities gamma(
    nominal=1.35)
    "Ratio of specific heats";
equation
  R = Modelica.Constants.R/MolarMass;
  gamma = c_p/c_v;
  h = u + p*v;
  rho*v = 1;
end GasPropBase;
```

Appendix B

The SimpleWorkingFluid model

```
model SimpleWorkingFluid
  "Gasmodel for working fluid with three
   gascomponents"
  extends Partial.GasPropBase(
    final n=3,
    xInit={1,0,0},
    xBurned={0,1,0});

  PerfectGas air(final x={1});
  IdealGas exhaust(final x={1});
  GasolineVapor gasolineVapor(final x={1});
  final parameter Real gasoline_position[n]={0,0,1}
    "Pointer to gasolineVapor fraction";
  Real phi;
equation

  MolarMass = air.MolarMass*x[1] +
  exhaust.MolarMass*x[2] +
  gasolineVapor.MolarMass*x[3];
  c_p = air.c_p*x[1] + exhaust.c_p*x[2] +
  gasolineVapor.c_p*x[3];
  c_v = air.c_v*x[1] + exhaust.c_v*x[2] +
  gasolineVapor.c_v*x[3];
  h = air.h*x[1] + exhaust.h*x[2] +
  gasolineVapor.h*x[3];
  rho = air.rho*x[1] + exhaust.rho*x[2] +
  gasolineVapor.rho*x[3];
```

```
  phi = if noEvent(x[1] > Modelica.Constants.small)
  then (x[3]*gasolineVapor.AFs)/x[1] else 10;

  T = air.T;
  p = air.p;

  T = exhaust.T;
  p = exhaust.p;

  T = gasolineVapor.T;
  p = gasolineVapor.p;
end SimpleWorkingFluid;
```

Appendix C

The IdealGasBase model

```
partial model IdealGasBase
  "Basemodel for gasproperties calculations"
  extends Partial.GasPropBase(
    final n=1,
    final xInit={1},
    final xBurned={1});
  //All parameters from Heywood p.131 table 4.10
protected
  parameter Real a_low[:]=gasDataBase.a_low[:];
  parameter Real a_high[:]=gasDataBase.a_high[:];
  parameter SI.Temperature T_limit=
    gasDataBase.T_limit
    "Switching temperature between
    burned/unburned properties";
public
  replaceable parameter GasData.GasDataBase
    gasDataBase;
equation
  MolarMass = gasDataBase.MolarMass;
  p*v = R*T;
  c_p = if T < T_limit then R*(a_low[1] +
    a_low[2]*T + a_low[3]*T^2 + a_low[4]*T^3 +
    a_low[5]*T^4)
    else
    R*(a_high[1] + a_high[2]*T +a_high[3]*T^2 +
    a_high[4]*T^3 + a_high[5]*T^4);
  c_v = c_p - R;
  h = if T < T_limit then R*T*(a_low[1] +
    a_low[2]/2*T + a_low[3]/3*T^2 + a_low[
    4]/4*T^3 + a_low[5]/5*T^4 + a_low[6]/T)
```

```
    else
    R*T*(a_high[1] + a_high[2]/2*T
    + a_high[3]/3*T^2 + a_high[4]/4*T^3 +
    a_high[5]/5*T^4 + a_high[6]/T);
end IdealGasBase;
```

Appendix D

The OpenCylinder model

```
partial model OpenCylinder
  "Simple engine cylinder model"
  extends VehProLib.Partial.TwoPinDynamic;
  constant Real PI=Modelica.Constants.PI;
  parameter Types.Degree ign=-20
  "Spark advance, before TDC";
  parameter Types.Degree deltaTheta=60
  "Duration of combustion";
  parameter Real lambda=1
  "Relative air/fuel ratio";

  SI.Volume V;
  SI.Mass m;
  SI.Mass mx[g.n](
    start=(pInit*V)/(286*TInit)*g.xInit,
    fixed=false,
    nominal=1e-6);
  SI.Energy U(start=100, fixed=false);
  SI.Angle theta;
  SI.AngularVelocity omega;
  SI.Power dW "Work produced on the piston";
  SI.Energy Qin(start=0, fixed=true)
  "Chemical energy in fuel";
  SI.Energy Qtot(start=0, fixed=true)
  "Total heat release";
  SI.HeatFlowRate dQch "Chemical heat release";
  SI.HeatFlowRate dQht
```

```
  "Heat transfer from the
  combustion chamber";
  Real mfb(start=0, fixed=true)
  "Mass fraction burned";
  SI.Area A "Cylinder wall area";
  SI.Angle thetaOne
  "Crank angle connected to the revolution";
  Real dxb(start=0, fixed=true)
  "Differentiated mass fraction burned";
  Interfaces.FlowCut_liquid i_fuel;
  Interfaces.FlowCut_o piston(n=g.n);
  Modelica.Blocks.Interfaces.InPort inPort_theta;
  VehProLib.Interfaces.Geometry geometry;
  Modelica.Thermal.HeatTransfer.
  Interfaces.HeatPort_b heatPort_b;

algorithm
  thetaOne := mod(theta, 4*PI);
equation
  piston.p = g.p;
  piston.T = g.T;
  piston.x = g.x;
  piston.Wx = zeros(g.n);
  heatPort_b.Q_dot = -dQht;
  //Heat flowing out of system
  theta = inPort_theta.signal[1];

  V = geometry.V;
  A = geometry.A;

  der(theta) = omega;
  der(Qin) = i_fuel.W*i_fuel.q_LHV;
  der(mfb) = if noEvent(Qin > 0) then
  dQch/Qin else 0;
  i_fuel.W = if i.W > 0 then
  i.W/(i_fuel.AFs*lambda) else 0;

  when thetaOne > PI and mfb > 0.5 then
    reinit(Qin, 0);
    reinit(mfb, 0);
  end when;

  //Energybalance
  der(U) = i.H + o.H - dW + dQch - dQht;
```

```
  // Massfraction balance
  der(mx) = i.Wx + o.Wx + i_fuel.W*{1};

  m = sum(mx);
  g.x*m = mx;
  g.u*m = U;
  p*V = m*g.R*T;

  dW = p*der(V);
  dxb = Functions.vibeDer(
  theta, ign/180*PI, deltaTheta/180*PI)*omega;
  dQch = Qin*dxb;
  der(Qtot) = dQch;
end OpenCylinder;
```

Appendix E

The OpenCylinderAdv model

```
model OpenCylinderAdv
  "Engine cylinder model, note can only be
  used when workingfluid is choosen as medium"
  extends VehProLib.Partial.TwoPinDynamic;
  constant Real PI=Modelica.Constants.PI;
  parameter Types.Degree ign=-20
  "Spark advance, before TDC";
  parameter Types.Degree deltaTheta=60
  "Duration of combustion";
  // Heattransfer
  constant SI.Temperature Tivc=350
  "Temperature at inlet valve closing";
  parameter Real C1=0.1 "Tuning constants";
  parameter Real C2=1 "Tuning constants";

  Real mx_burnstart[g.n](start=zeros(g.n))
    "Mass of gascomponents at start of burn";
  Real phi "Relative fuel/air ratio";
  SI.Volume V;
  SI.Mass m;
  SI.Mass mx[g.n](
    start=(pInit*V)/(286*TInit)*g.xInit,
    fixed=false,
    nominal=1e-7);
  SI.Energy U(start=100, fixed=false);
  SI.Energy Qin(start=0, fixed=true)
  "Chemical energy in fuel";
```

```
  SI.Angle theta;
  SI.AngularVelocity omega;
  SI.Power dW "Work produced on the piston";
  SI.HeatFlowRate dQht
  "Heat transfer from the combustion chamber";
  SI.HeatFlowRate dQch "Chemical heat release";
  Real dxb(start=0, fixed=true)
  "Differentiated mass fraction burned";
  Real mfb(start=0, fixed=true)
  "Mass fraction burned";
  SI.Area A "Cylinder wall area";
  SI.Angle thetaOne
  "Crank angle connected to the revolution";
  parameter SI.Temperature deltaT=1
  "Delta used in approx. of dc/dT";
  Real c(
    nominal=0.1,
    min=0,
    max=1);
  Real c_dot;
  // Heattransfer
  SI.Temperature Twall "Cylinder wall temperature";
  SI.Velocity meanSpeed "Mean piston speed";
  Real h "Heat transfer coefficient";
  Interfaces.FlowCut_o piston(n=g.n);
  Modelica.Blocks.Interfaces.InPort inPort_theta;
  VehProLib.Interfaces.Geometry geometry;
  Modelica.Thermal.HeatTransfer.
  Interfaces.HeatPort_b heatPort;

algorithm
  thetaOne := mod(theta, 4*PI);
  //Fix massratio at start of burn
  when dxb < 0.001 and dxb > 0.0005 then
    mx_burnstart := mx;
  end when;
equation
  piston.p = g.p;
  piston.T = g.T;
  piston.x = g.x;
  piston.Wx = zeros(g.n);
  heatPort_b.Q_dot = -dQht;
  //Heat flowing out of system
  Twall = heatPort_b.T;
```

```
meanSpeed = ((omega/2)/PI*geometry.stroke)*2;
theta = inPort_theta.signal[1];
h = Functions.woschniHT(p, meanSpeed, 5*mfb,
 ((geometry.Vc*geometry.r_c)/geometry.V)^(g.gamma),
 Tivc, T, C1, C2, geometry.r_c,geometry.bore);
if cardinality(heatPort_b) == 0 then
  dQht = 0;
else
  dQht = (h*A)*(T - Twall);
end if;

V = geometry.V;
A = geometry.A;

der(theta) = omega;
phi = g.phi;

c = Functions.Heywood_c(g.T, phi,
  g.gasolineVapor.epsilon);
c_dot = (Functions.Heywood_c(g.T + deltaT,
  phi, g.gasolineVapor.epsilon) -
  Functions.Heywood_c(g.T - deltaT, phi,
  g.gasolineVapor.epsilon))/(2*deltaT)*der(T);

//Energybalance
der(U) = i.H + o.H - dW - dQht + dQch;

if phi <= 1 then
  der(mx[1]) = i.Wx[1] + o.Wx[1];
  der(mx[2]) = i.Wx[2] + o.Wx[2]
  - mx_burnstart[2]*phi*dxb;
  der(mx[3]) = i.Wx[3] + o.Wx[3]
  - mx_burnstart[3]*dxb;
  der(mx[4]) = i.Wx[4] + o.Wx[4]
  + 2*(1 - g.gasolineVapor.epsilon)
  *phi*mx_burnstart[2]*dxb;
  der(mx[5]) = i.Wx[5] + o.Wx[5]
  + g.gasolineVapor.epsilon*phi*
  mx_burnstart[2]*dxb;
  der(mx[6]) = i.Wx[6] + o.Wx[6];
  der(mx[7]) = i.Wx[7] + o.Wx[7];
else
  der(mx[1]) = i.Wx[1] + o.Wx[1];
  der(mx[2]) = i.Wx[2] + o.Wx[2]
```

```
    - mx_burnstart[2]*dxb;
    der(mx[3]) = i.Wx[3] + o.Wx[3]
    - mx_burnstart[3]*dxb;
    der(mx[4]) = i.Wx[4] + o.Wx[4]
    + (2*(1 - g.gasolineVapor.epsilon*phi) + c)
    *mx_burnstart[2]*dxb + c_dot*mx_burnstart[2]*mfb;
    der(mx[5]) = i.Wx[5] + o.Wx[5]
    + (g.gasolineVapor.epsilon*phi - c)
    *mx_burnstart[2]*dxb - c_dot*mx_burnstart[2]*mfb;
    der(mx[6]) = i.Wx[6] + o.Wx[6]
    + (2*(phi - 1) - c)*mx_burnstart[2]
    *dxb -c_dot*mx_burnstart[2]*mfb;
    der(mx[7]) = i.Wx[7] + o.Wx[7]
    + c*mx_burnstart[2]*dxb
    + c_dot*mx_burnstart[2]*mfb;
  end if;

  m = sum(mx);
  g.x*m = mx;
  g.u*m = U;
  p*V = m*g.R*T;

  when thetaOne > PI and mfb > 0.5 then
    reinit(Qin, 0);
    reinit(mfb, 0);
  end when;

  dW = p*der(V);
  der(Qin) = i.Wx*g.gasoline_position
    *g.gasolineVapor.q_LHV;
  dQch = Qin*dxb;
  dxb = if noEvent(Qin > 1) then
    Functions.vibeDer(theta, ign/180*PI,
    deltaTheta/180*PI)*omega else 0;
  der(mfb) = if noEvent(Qin > 1) then
    dQch/Qin else 0;

end OpenCylinderAdv;
```

Copyright

Svenska

English