2005

Proceedings of the 4th International Modelica Conference

March 7-8, 2005
Hamburg University of Technology
Hamburg-Harburg, Germany

Gerhard Schmitz (editor)

Volume 2

organized by
The Modelica Association and
the Department of Thermodynamics, Hamburg University of Technology

All papers of this conference can be downloaded from
http://www.Modelica.org/Conference2005/papers/

Proceedings of Modelica' 2005
Hamburg University of Technology,
Hamburg-Harburg, Germany, March 2005
Editor:
Prof. Dr.-Ing. G. Schmitz
Published by:
The Modelica Association (http://www.Modelica.org) and
the Department of Thermodynamics,
Hamburg University of Technology (http://www.tt.tu-harburg.de)
Printed by:
COPY-DRUCK
Gesellschaft für Digital- und Offsetdruck mbH

Preface

The first Modelica Conference took place October 2000 in Lund, Sweden. Since then, Modelica has been more and more established as a preferred modelling language for complex multi-domain systems. This is indicated by the high number of registrations from industry and science for the 4th International Modelica Conference which is held between March 7th and 8th 2005 at Hamburg University of Technology (TUHH). But it is also indicated by the number of excellent papers submitted to the program committee which made the task of selecting papers for oral and poster presentation very difficult and, last but not least, by the exhibition during the conference at which around 10 companies will present themselves. The proceedings contain the papers of the 60 oral presentations and 9 poster presentations given at the conference. The ability of Modelica as a multi-domain simulation language is demonstrated impressively by the various fields that are covered, e.g. digital electronic devices, hybrid electric power trains, waste water processes or thermodynamic applications.

With the special features of the Modelica language, e.g. object-oriented modelling and the ability to reuse and exchange models, Modelica has become – among other things – a further step towards of an integrated engineering design process. In some fields Modelica is being used as a standard platform for model exchange between suppliers and OEM's, for example in case of vehicle air conditioning systems.

A key issue for the success of Modelica is the continuous development of the Modelica language by the Modelica Association under strict observance of backward compatibility to previous versions. The broad base of private and institutional members of the Modelica Association as a non-profit organization ensures language stability and security in software investments.

The Modelica Conference 2005 was organized by the Modelica Association and by the Department of Thermodynamics of Hamburg University of Technology (TUHH), Germany. Together with the entire team of the local organizing committee I would like to wish all participants an excellent and fruitful conference.

Hamburg-Harburg, March 1, 2005

Gerhard Schmitz

Program Committee

- Prof. Gerhard Schmitz, Hamburg University of Technology, Germany (Program chair).
- Prof. Bernhard Bachmann, University of Applied Sciences Bielefeld, Germany.
- Dr. Francesco Casella, Politecnico di Milano, Italy.
- Dr. Hilding Elmqvist, Dynasim AB, Sweden.
- Prof. Peter Fritzson, University of Linköping, Sweden
- Prof. Martin Otter, DLR, Germany
- Dr. Michael Tiller, Ford Motor Company, USA
- Dr. Hubertus Tummescheit, Scynamics HB, Sweden

Local Organizers

- Gerhard Schmitz
- Katrin Prölß
- Wilson Casas
- Henning Knigge
- Jens Vasel
- Stefan Wischhusen
- TuTech Innovation GmbH

Contents

Volume 1

Index of Authors

Session 4a

Automotive Simulation II

Modeling an automotive power train and electrical power supply for HiL applications using Modelica

Lutz Morawietz[1] Silvio Risse[1] Thomas Christ[2]
Hans Zellbeck[1] Hans-Christian Reuss[3]

[1]Dresden University of Technology
Institute of Combustion Engines and Automotive Engineering
George-Baehr-Str. 1c, 01062 Dresden
{morawietz, risse, zellbeck}@ivk.tu-dresden.de

[2] BMW Group
Energy management und CO_2
80788 München
Thomas.Christ@bmw.de

[3] University of Stuttgart
Institute for Internal Combustion Engines and Automotitve Engineering
Pfaffenwaldring 12, 70569 Stuttgart
reuss@ivk.tu-stuttgart.de

Abstract

Fuel consumption and emissions are key issues in automotive development. An intelligent energy management helps to improve both factors. Tools for developing new management strategies can be off-line simulation as well as Hardware-in-the-Loop (HiL) simulation. This paper gives an overview over a joint project between Dresden University of Technology and the BMW Group. In this project an `EnergyLibrary` containing power train and electrical power net models is improved.

The paper will describe the thermodynamic model of the internal combustion engine (ICE), the alternator model, and the implemented `NeuralNetworkLibrary` in more detail. On the HiL test bench component measurements and new energy management strategies for the electrical power net can be tested.

1 Introduction

Legal regulations on fuel consumption and the rising need of comfort and safety are the main issues in automotive development.

One main field of interest is the reduction of fuel consumption by optimizing the auxiliary units, the warm-up behavior of the internal combustion engine (ICE) and the electrical power net [4].

The electrical energy management controls the energy generation, distribution and storage in the electrical power net. It enhances the robustness of the system and is capable of reducing the fuel consumption.

The interaction between the electrical power net and the drive train occurs at the alternator. A rising demand of electrical energy leads to an increased driving torque of the alternator and therefore to an increased fuel consumption. The alternator's torque demand plays an important role in the dynamic behavior of the ICE, especially at idle speed.

For fuel consumption the thermal behavior of the ICE plays another dominant role. Between 10-20% of the fuel during the New European Driving Cycle (NEDC) is used for the warm up.

This paper describes some enhancements made to the BMW model library used for energy flow prediction. The models are derived from measurements generated on an engine test bench, a Hardware-in-the-Loop (HiL) test bench, and during vehicle testing.

The presented results were developed in a joint project of Technische Universität Dresden and BMW Group.

2 Approach and Tools

The goal is to model the electrical power net, the auxiliary units, and the ICE including its thermodynamic behavior using Modelica/Dymola.

The overall vehicle model consists of objects of many different physical domains such as electronics, mechanics, thermodynamics, hydraulics, pneumatics as well as control systems and is described in section 3.1.

The model parameters are gained using different measurement environments:

(i) on an ICE test bench the ICE itself and the auxiliary units are measured

(ii) in a test vehicle all internal car data (e.g. CAN), electrical currents and voltages, pressures, temperatures and torques of engine and auxiliary units can be recorded.

(iii) HiL test bench for the electrical power net where single components of the power net, especially the alternator, can be examined

Using these test environments, realistic inputs for the HiL test bench and model validation can be generated.

The HiL test bench is especially important since one can efficiently test new components and control strategies in real time. The interaction with other subsystems in a car can be analyzed.

In real time environments high simulation speed is crucial. Therefore the models were adapted accordingly by eliminating unnecessary dynamics and expensive computations. Most of the model behavior is represented using look-up tables or Neural Networks.

At BMW more and more system simulations are carried out in object oriented simulation environments. Modelica is currently in use for the simulation of the electrical power net, the air conditioning, the fuel consumption and hybrid drive trains.

3 Model

As mentioned earlier fuel consumption is the main focus of the simulations done in this project. In the first stage the car is simplified to its longitudinal dynamics. The model frame work, the thermodynamic engine model, and the power net are described.

3.1 EnergyLibrary

The developed model library, shown in figure 1 is divided into four main packages:

(i) `DriveLine` includes `Engine`, `GearBox`, `Tank`, `Axles`, and `CargoSystems`

(ii) `DriveEnvironment` includes `Tracks`, `Driver`, and `BusSystem`

(iii) `AuxiliaryUnits` include `BeltDrive`, `ClimateUnit`, `HydraulicUnits`, and `PowerNet`

(iv) `Blocks` include non Modelica standard blocks

Each component model (e.g. the ICE) consists of the following packages:

(i) `Basis` including templates for icons, connector (e.g. rotational flanges, bus), and component specific sub models

(ii) `BusSystem` including the bus signals that the component uses for its communication

(iii) `Models` including various types of models (e.g. various warm up models). Those models can be selected via `Choice`.

(iv) `Record` which represents the parameter structure for the most commonly used models.

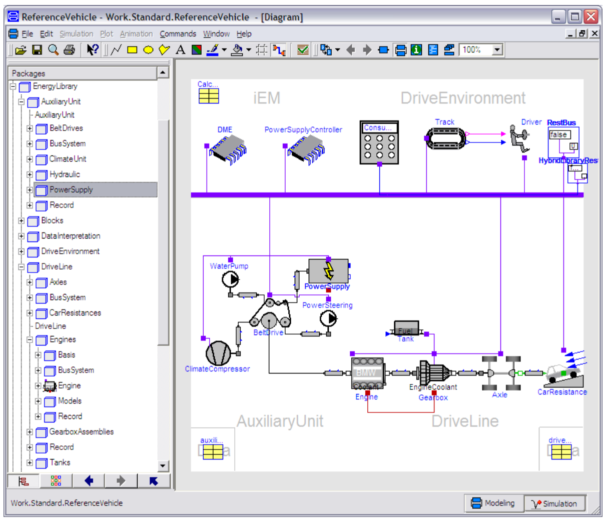


Figure 1: Library structure and vehicle model

The main model is assembled by the `Choice` blocks of each component model. That way structural changes in the model can be made comfortably.[1] The top level of the car model and the library structure can be seen in figure 1.

[1]For example a change between automatic and manual `GearBox`

The `Bus` and `Record` structure are assembled in a similar tree structure by instantiation. Thus the bus signals are grouped after simulation in a concise tree structure as shown in figure 2. We established the same tree structure in the model library, the simulation model, the parameters, and the bus system.

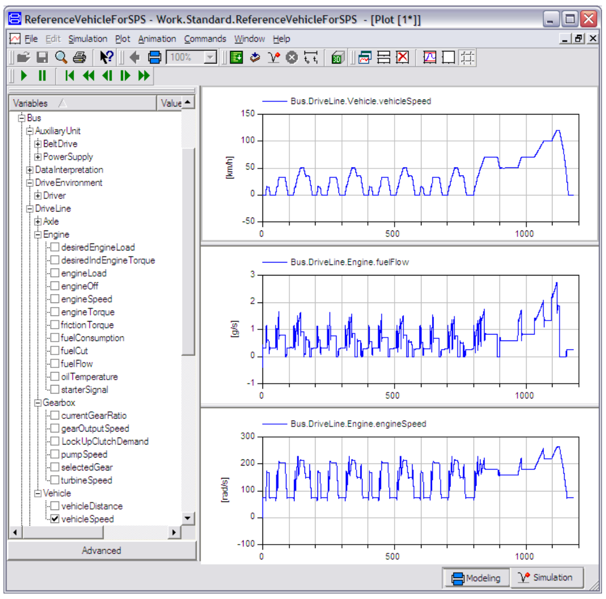


Figure 2: Simulation of the New European Drive Cycle showing the structured `Bus`

The model parameters are managed by a parameter database which also allows pre-processing.

3.2 Thermodynamic Model of an ICE

The temperature has a major impact on the behavior of the ICE and the auxiliary units. To be able to simulate the warm-up phase a thermodynamic model of the ICE is necessary. Additionally several environmental conditions and different auxiliary loads have to be taken into account.

Torque Balance

An essential requirement of the model is the determination of the relevant torques taking into account different auxiliary and engine loads. According to [3] the torque balance is given by the following equation:

$$0 = T_{ind} - T_{fric} - T_{aux} - T_{eff} \qquad (1)$$

The indicated engine torque T_{ind} is defined as the possible drive torque which theoretically can be used if the engine is mechanically free of losses. T_{eff} represents the torque used for the vehicle movement. Beside the losses caused by the engine friction T_{fric} the torque of the auxiliary systems T_{aux} has to be considered. The load torque of the auxiliary systems is determined for every relevant unit in separate models. Additionally for each torque a temperature dependence is included. These dependencies are caused by the tribological behavior.

Thermodynamic Behavior

The thermal behavior of an ICE is defined by its heat capacities, heat transfers and thermal conductivities as well as its surrounding conditions. The heat capacities can be divided into two major groups. There are constant heat capacities which arise from engine construction and varying heat capacities following from fluid systems. For the latter ones the oil and the coolant circuit are relevant. The oil temperature has a direct influence on the engine friction T_{fric}. Therefore it plays an essential role in ICE fuel consumption.

Although different thermodynamic libraries for modelling fluid flow already exist in Modelica, none of them seem able to be run on a real time platform. Due to that a more basic `FluidFlowLibrary` was developed. All fluid systems are described by a stationary pressure drop model and defined as an incompressible single medium. Because phase changing of the medium appears only locally during standard driving cycles (e.g. NEDC, FTP75) it is not taken into consideration. This has negligible influence on the thermal behavior of the overall system.

In the `FluidFlowLibrary` mainly `TwoPort`-components are used. The connectors are reduced to the variables: Pressure p, temperature T and the flow variables mass flow rate $\dot{m}$ and heat flow rate $\dot{Q}$.

```
connector Port_A
   Modelica.SIunits.Pressure p;
   flow Modelica.SIunits.MassFlowRate mdot;
   Modelica.SIunits.Temperature T;
   flow Modelica.SIunits.HeatFlowRate Qdot;
end Port_A;
```

Besides to the two connectors for the fluid transfer, a heat port `HeatPort_A` from the standard Modelica library was inserted. For each of the control volumes mass and energy balance equations are applied. The internal energy ΔU is calculated with the help of the enthalpy flows $\dot{Q}_{in}$, $\dot{Q}_{out}$ at the connectors and the heat losses $\dot{Q}_{heat}$ at the volume boundary. As a reference value the mean temperature T_{mean} of the volume is used. Work due to change of volume is not taken into account.

```
model volume
...
equation
  ...
  //energy balance equation
  dU=Qindot + Qoutdot + Qheatdot;

  // heat transfer equations
  Qindot = Port_A.mdot*cp*Port_A.T;
  Qoutdot = Port_B.mdot*cp*Port_B.T;

  //equations for heat loss
  Qheatdot=HeatPort_A.Q_dot;
  Tmean=HeatPort_A.T;

  //equation for the internal energy
  dU = m*cp*der(Tmean);
  ...
  end heater;
```

The media qualities depending on temperature, e.g. density, specific heat capacities c_p, are computed for each volume separately. Either look-up tables or polynomial functions are used for these calculations [6].

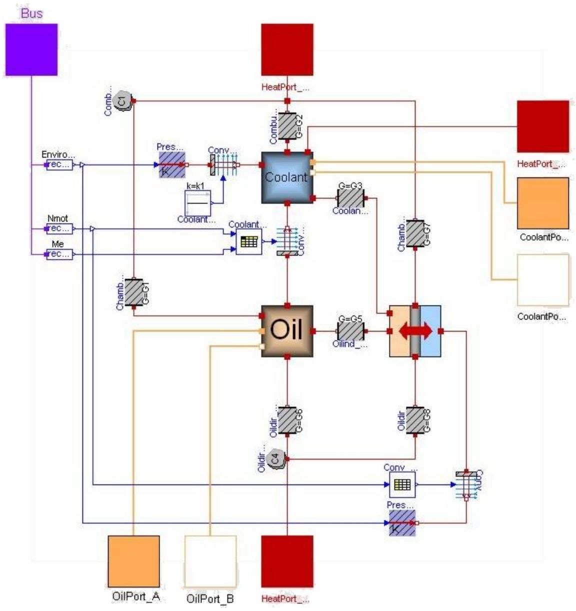


Figure 3: Thermodynamic network of the ICE model

Figure 3 shows a part of the thermodynamic network model. It consists of several heat capacities (i.e. oil, coolant, solid material of the engine) and their heat transfers. The contact to the rest of the ICE model and its environment is defined by the standard `HeatPorts`, the `Bus` connectors and the earlier described ports for the fluid flows (orange). These connectors allow the exchange of oil and coolant between ICE and other components in the cooling circuit.

3.3 Electrical Power Net

The components of the electrical power net can be divided into four parts:

(i) generators: alternator

(ii) storage devices: battery, double layer capacitor

(iii) converter: DC/DC converter

(iv) loads: seat heating, fan, driving light

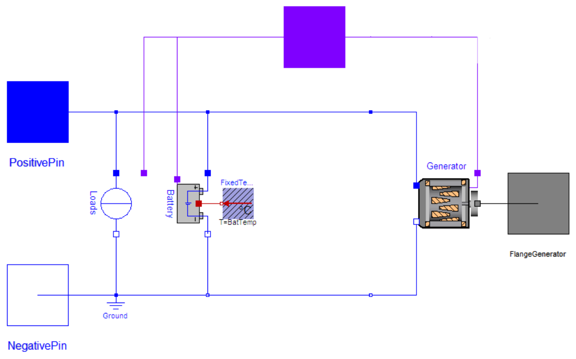


Figure 4: Object diagram of a simplified power net

Figure 4 shows the model of a simplified power net. The `Loads` are modelled in three simple ways:

ohmic resistance the resistance is independent of voltage and current

power sink the consumed power is independent of voltage and current

current sink the consumed current is independent of voltage

More effort is needed modelling storage and distribution devices and power sources.

Alternator model

In a conventional power net the alternator is the only source for electrical power. With regards to modelling two aspects are of interest:

(i) Fuel consumption caused by the alternator

(ii) Computation of the charge-balance

(iii) Dynamic behavior at changing electrical loads

The dynamic behavior of the generated current influences the voltage stability of the power net. It also causes a dynamic torque which affects the ICE. Close to idle speeds this torque causes unwanted disturbances in the engines smoothness.

In modern alternators the so-called *Load Response Control* is used to improve the engine smoothness. It limits the slew rate of the generator current.

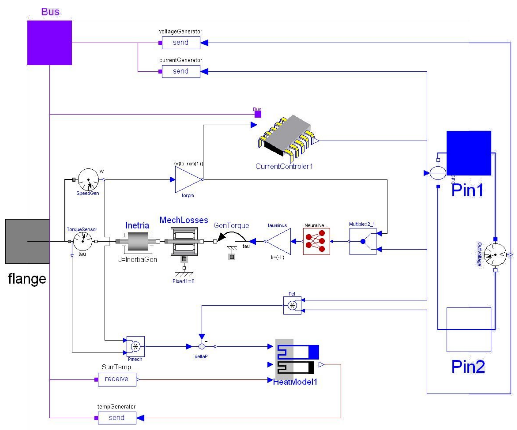


Figure 5: Model of an alternator

Figure 5 shows the connection between the thermal, mechanical, electrical and control part of the alternator.

The alternator `CurrentController` compares the reference voltage and the actual voltage and controls the current between the electrical pins.

Part of the `CurrentController` is the *Load Response Control* implemented employing a rate limiter:

```
model RateLimiter "Rate limiter"
  extends Modelica.Blocks.Interfaces.SISO;
  parameter Real RR
        "Maximum rising slew rate";
  parameter Real FR
        "Maximum falling slew rate";
  parameter Real Ts=0.01 "Sampling time";
  Real x "auxiliary variable";

equation

  if initial() then
    y = u;
    x = 0;
  end if;

 der(x) = (u - y)/Ts;
  der(y) = smooth(0,noEvent(
  if der(x) > RR then RR else if der(x) < FR
        then FR else der(x)));
end RateLimiter;
```

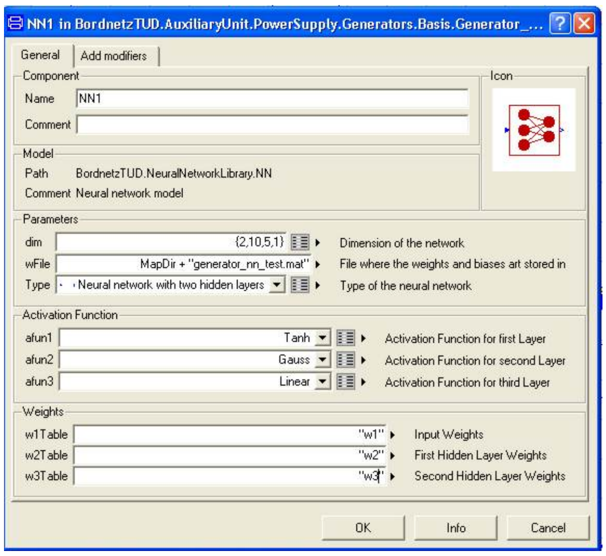
In order to determine the slew rate the input signal has to be derived once. For discontinuous input functions the derivation is not defined at all times. Therefore we introduce the auxiliary variable `x` and the sampling time `Ts`.

The link between the electrical and mechanical domain is realized employing a Neural Network. It evaluates the alternator torque in depending on its current and revolution speed. The `MechanicalLosses` model includes all bearing friction and fan losses.

4 Neural Network Library

At system level mechatronic components mostly exhibit strong nonlinear behavior. Often this behavior is hard to describe in a mathematical way. Even if it is described mathematically the models need a high computational power and are not usable on a real time platform. In this case another way of describing this behavior can be realized with the help of look-up tables or Neural Networks [5]. In general Neural Networks require less memory than look-up tables but employing lager networks the computational effort will rise. The lower need of memory is of greater interest for real time simulation.

To be able to use Neural Networks in Dymola a `NeuralNetworkLibrary` was developed. It can be used to simulate feed-forward networks with up to two hidden layers.


NN1 in BordnetzTUD.AuxiliaryUnit.PowerSupply.Generators.Basis.Generator_...

General | Add modifiers

Component
Name: NN1
Comment:

Model
Path: BordnetzTUD.NeuralNetworkLibrary.NN
Comment: Neural network model

Parameters
dim: {2,10,5,1} — Dimension of the network
wFile: MapDir + "generator_nn_test.mat" — File where the weights and biases art stored in
Type: Neural network with two hidden layers — Type of the neural network

Activation Function
afun1: Tanh — Activation Function for first Layer
afun2: Gauss — Activation Function for second Layer
afun3: Linear — Activation Function for third Layer

Weights
w1Table: "w1" — Input Weights
w2Table: "w2" — First Hidden Layer Weights
w3Table: "w3" — Second Hidden Layer Weights

Figure 6: Parameters of the Neural Network Model

In figure 6 the parameter window of a neural network can be seen. The dimension of the Neural Network is defined as an array named `dim`. The user can choose from zero to two hidden layers. Every layer has its own activation function. The weights and bias values are loaded from a Matlab file during initialization of the process. The input and the output are vector ports. The size of the vector depends on the number of input and output neurons, respectively.

First tests were performed at a dSPACE system to estimate the performance of the Neural Networks in comparison to look-up tables. Therefore a real time simulation was carried out on the HiL test bench described in section 5. The look-up table in the generator model (figure 5) was replaced by a fully connected Neural Network with two input neurons, one output neuron and two hidden layers. The first layer consists of 20 neurons, the second one of 10. As activation functions the hyperbolic tangent was used.

Using the explicit Euler method for inline integration the computational need of the Neural Network is only slightly higher than for look-up tables. But further work needs to be done varying the size of the Neural Network and using different activation functions.

5 Dymola in a HiL environment

For measuring component parameters and testing strategies for energy management a Hardware-in-Loop (HiL) test bench was built. It is kept modular so that the boundary between hardware and software can be shifted in a certain range. With this concept it is possible to cut free the complex system of the electrical power net to different degrees.

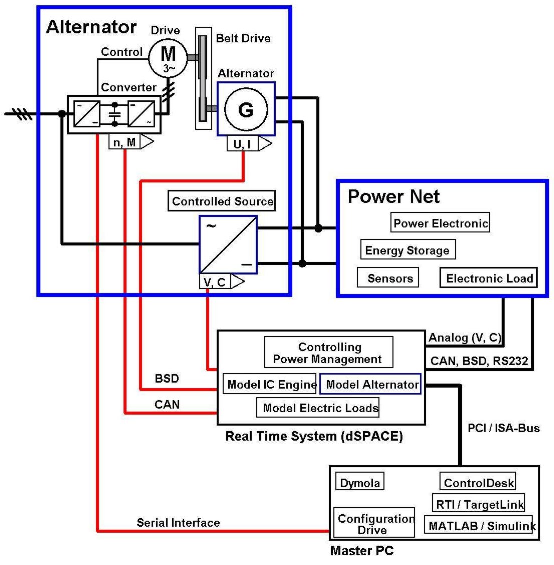


Figure 7: Schematic structure of the HiL test bench

In figure 7 a schematic of the test bench structure can be seen. Two power sources are implemented. On one hand there is a physical alternator driven by an electric motor. On the other hand a programmable current source can be used. To emulate the alternator behavior the model described in section 3.3 is used.

A dual power net can be realized by using a system of three busbars. This way various topologies can be build up using power sources, storage devices and power electronics.

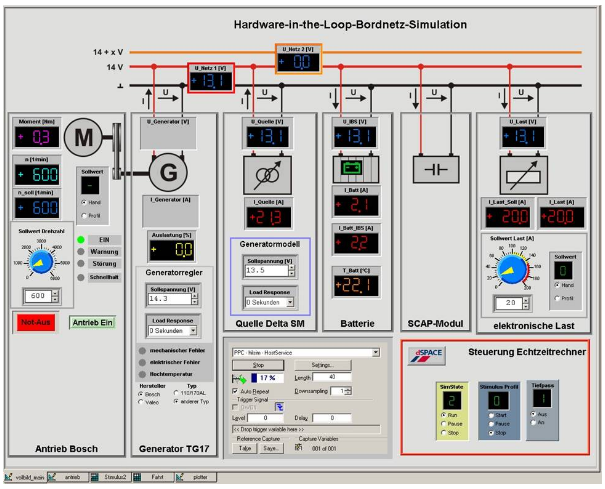


Figure 8: Operating and monitoring the HiL test bench with dSPACE ControlDesk

The HiL computer is based on a *dSPACE modular hardware* system. The core of this system is a *DS1005* where the models are computed on a 500 MHz PowerPC. Further, the *dSPACE expansion box* includes several boards for analog and digital signal I/O, CAN interfaces and digital signal processing. The user interface to the real time system is given by the software *dSPACE ControlDesk*. In figure 8 a screen shot can be seen. [1]

The task of the *dSPACE system* is both to monitor and control the HiL test bench and to simulate models under real time conditions. An electronic controlled load replaces the electrical consumer load. Its input is derived from profiles measured in a test vehicle or from the models described in section 3.3. The other necessary input data to the HiL environment, e.g. engine speed, surrounding temperature, control voltage of the generator, are gained in a similar way either by simulation or measurement.

The main goals for the HiL environment are:

(i) compare the alternator behavior to the alternator model

(ii) implement the model of the ICE including thermal behavior

(iii) test energy management strategies

So far the step size for the integration at the real time system is set to 1 ms.

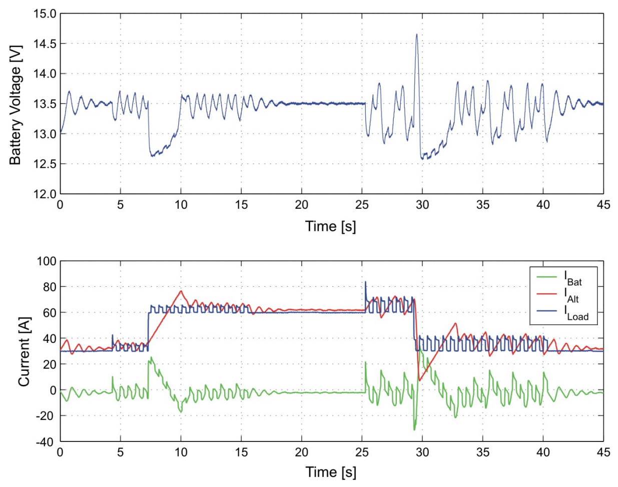


Figure 9: Example of an HiL simulation result

As an example for HiL simulation figure 9 shows an experiment using the alternator model. In this case the components of the power net are the controlled load, a real battery and the alternator model controlling the electronic power source. As load current a measured blinker current with an additional load step at 7 s was taken. The slew rate of the alternator current was limited by the earlier described *Load Response Control*. The difference between load and alternator current has to be equalized by the battery current. During this time the voltage drops from alternator voltage (13.5 V) to the battery voltage (12.6 V).

6 Conclusions

The existing model libraries for automotive power train and power net were extended for better thermal engine modeling and electrical real time simulation. The simulation platform enables us to develop new energy management strategies and test them under realistic conditions. For real time application the Modelica/Dymola models are included in a Simulink/dSpace environment. First tests are done replacing commonly used look-up tables by Neural Networks to reduce the size of the real time code without generating computational overhead.

In automotive every day use the Modelica/Dymola environment has proved to be a useful tool.

References

[1] dSPACE: http//www.dspace.de

[2] Dymola: Dymola User Manual, Version 5.3a. Dynasim AB, http://www.dynasim.se, 2004

[3] Heywood, J.: Internal Combustion Engine Fundamentals. McGraw-Hill, Inc., 1988

[4] Meir, F., Bertram, M., Christ, T.,Reuss, H.-C., Morawietz, L., Büchner, S.: Energiemanagement des elektrischen Bordnetzes im Kraftfahrzeug - Werkzeuge der Funktionsentwicklung. Proceedings of the VDE Congress, Berlin, 2004

[5] Krug, C.: Ein Beitrag zur dynamischen Modellierung des Verbrennungsmotors für Ausgaben der Echtzeit-Simulation. München, Germany, PhD thesis, Department of Car-electornics and electrics, Dresden University of Technology, 2003

[6] Verein Deutscher Ingenieure: VDI-Wärmeatlas. Springer Verlag, 2002

Modelica in Automotive Simulations - Powernet Voltage Control during Engine Idle

Erik Surewaard
Ford Forschungszentrum Aachen
Suesterfeldstrasse 200
52072 Aachen, Germany
esurewa1@ford.com

Marc Thele
RWTH Aachen University
Jaegerstrasse 17-19
52066 Aachen, Germany
marc.thele@isea.rwth-aachen.de

Abstract

Due to the increasing electric power demand of future vehicles, problems may be expected with the voltage stability of the powernet. In conventional vehicles, control of the powernet voltage can be lost when the loads in the electric powernet request more power than can be supplied by the generator. In that case voltage control of the powernet will be lost since the generator will not be able to follow the voltage setpoint. The voltage of the electric powernet will drop to and follow the battery voltage. This will go impaired with undesired voltage fluctuations resulting in light flicker and blower motor fluctuations, which can be noticed by the vehicle occupants. This will have a negative effect on customer perception.

This paper describes both physical plant models and control algorithms, which can be used for simulation of the electric powernet. By making use of the ModelicaVMA structure that has been set-up by Tiller *et. al.* in [1], a simulation model is set-up including detailed models of: (i) a battery, (ii) generator, (iii) heated front windscreen and (iv) an internal combustion engine. In the case of de-icing the front windscreen during engine idle, simulations have been performed to investigate what the effect is of (i) engine idle speed control and (ii) load switching compared with the conventional situation.

1 Introduction

The average electric power drawn in a conventional vehicle, shows a rising trend over the years. This is on the one hand caused by the growing amount of electric comfort loads but also by the electrification of vehicle chassis functions, *e.g.* Electric Power Assisted Steering (EPAS) and Electronic Damping Control (EDC). Since the electric powernet will get an increasingly important function in the vehicle, electric load models will also become increasingly important for future vehicle simulations. That the Modelica language can be of great benefit in electric powernet simulations has already been shown by the author in [2], [3] and [4]. That Modelica is also suitable for control system development will be shown in an application for powernet voltage control during engine idle. Based on the ModelicaVMA structure, this paper describes physical models and control algorithms, which can be used to investigate voltage stability of the electric powernet during engine idle.

2 Powernet Voltage Stability

Fig. 1 shows a schematic of the powernet of a conventional vehicle in which the generator voltage is the only variable that can be actively controlled.

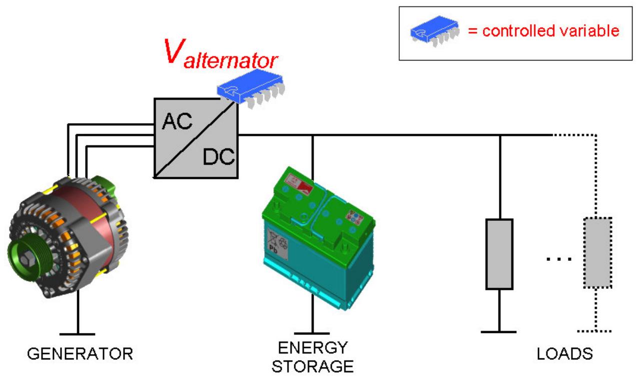


Fig. 1 Powernet topology for a conventional vehicle in which only the generator voltage can be controlled

Dependent on the amount of requested electric power by the loads, two powernet states can occur:

1. The powernet state in which the requested electric power is LOWER than the maximum that can be supplied by the generator. In this state the powernet voltage will be close to the voltage setpoint of the generator: in a conventional vehicle the voltage setpoint (*e.g.* 13.7V) of the generator is above the open circuit voltage of the battery (12.7V). Therefore the battery will be charged continuously in this state.
2. The powernet state in which the requested electric power is HIGHER than the maximum that can be supplied by the generator. In this state the additional requested power will be drawn from the battery. The saturated generator will not be able to follow the given setpoint and control of the powernet voltage will be lost: the powernet voltage will be determined by the battery. This will go hand in hand with voltage fluctuations which can be noticeable to the vehicle occupants: *e.g.* light flicker, changes in noise generated by the blower fan.

In conventional vehicles, the demand of electric power is usually highest during winter time when the engine is idling and the heated front screen is activated. The generator is usually saturated in this situation resulting in a loss of control of the powernet voltage. This paper will investigate means to prevent loosing voltage control in this situation by using idle speed control or reducing the electric power to a load.

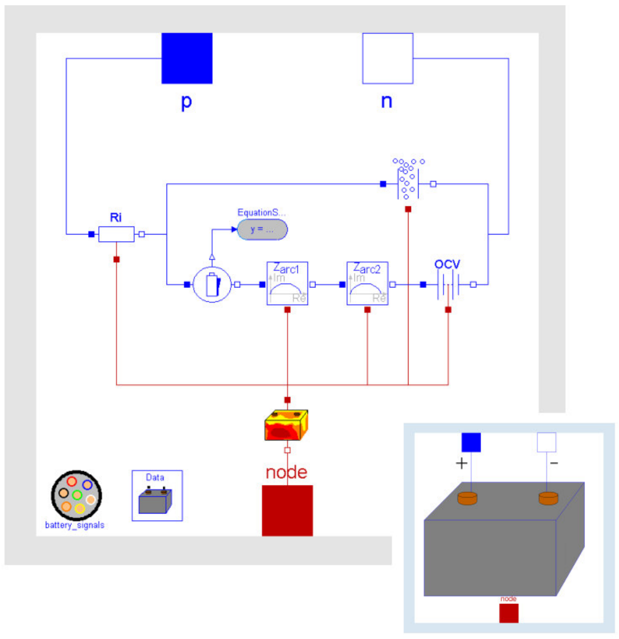


Fig. 2 Battery model as described in [3]

3 Battery

Fig. 2 shows the battery model that has been described by Surewaard in [3]. The issue with this model is that overcharging is not well described since the overcharging process is difficult to investigate with impedance measurements. The first step that is taken in the improvement of the battery model is to include the mass transport processes, which have been described by Thele in [5]. Mass transport processes will have a significant effect on the equilibrium voltage, also known as Open Circuit Voltage (OCV), of the battery. Detailed information on the Inclusion and research on the overcharging reaction is still in progress at the moment.

For a lead-acid (PbA) battery the discharge reaction that occurs at the positive electrode surface is the reaction of sulphuric acid and lead dioxide into lead sulphate and water whilst consuming two free electrons:

$$PbO_2 + HSO_4^- + 3H^+ + 2e^- \rightarrow PbSO_4 + 2H_2O$$

The discharge reaction that takes place at the negative electrode surface, is that lead and sulphuric acid react to lead sulphate and two hydrogen protons, forming two free electrons:

$$Pb + HSO_4^- + H^+ \rightarrow PbSO_4 + 2H^+ + 2e^-$$

During charging above-mentioned reaction will take place in the opposite direction.

The reasons for including mass transport processes in the battery model are the following processes:

- ACID FORMATION/CONSUMPTION AND DIFFUSION
  During discharging, sulphuric acid is consumed at the contact surface of both the positive and negative electrode. During charging sulphuric acid is formed at the electrode surfaces, which are in direct contact with the electrolyte. The differences in concentration will result in mass transport (diffusion) of sulphuric acid.

- CHARGE MIGRATION
  This is the movement of charged particles (ions) due to the electric field that exists between the positive and negative electrode. HSO_4^- ions and electrons will be attracted by the positive electrode, H^+ protons will be attracted by the negative electrode.

3.1 Electrode Equilibrium Potential and Acid Formation/Consumption

The equilibrium potential of the positive and negative electrode is dependent on the acid concentration. According to Bode in [6], the molarity of sulphuric acid can be calculated from the sulphuric acid concentration by:

$$m = 1.00 \cdot 10^3 \cdot C + 3.55 \cdot 10^4 \cdot C^2 + 2.17 \cdot 10^6 \cdot C^3 + 2.06 \cdot 10^8 \cdot C^4 \quad (1)$$

in which C represents the acid concentration and m the molality. The electrode potential can now be calculated by the following equations:

$$u_+ = 1.628 + 0.0739 \cdot \log(m) + 0.0331 \cdot \log^2(m) + 0.0432 \cdot \log^3(m) + 0.0216 \cdot \log^4(m) \quad (2a)$$

$$u_- = -0.295 - 0.0736 \cdot \log(m) - 0.0305 \cdot \log^2(m) - 0.0305 \cdot \log^3(m) - 0.0120 \cdot \log^4(m) \quad (2b)$$

Acid formation/consumption at the electrode surface has been modeled with the following equations:

$$q_+ = \frac{3 - 2 \cdot t_+}{d \cdot A \cdot F \cdot 3600} \cdot i \quad (3a)$$

$$q_- = \frac{2 \cdot t_+ - 1}{d \cdot A \cdot F \cdot 3600} \cdot (-i) \quad (3b)$$

in which q_+ and q_- are the flows of acid ions, t_+ the transfer number for cations, d the electrode thickness, A the effective electrode surface, F the Faraday constant and i the current.

The above-mentioned equations have been implemented in a Modelica model for the positive and negative electrode of which Fig. 3 shows both top level icons.

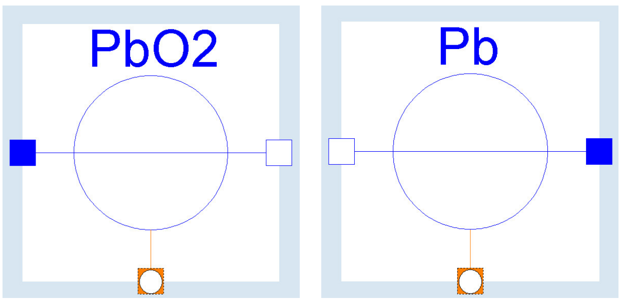


Fig. 3 Electrode equilibrium potentials which are dependent on acid concentration: (a) positive electrode, (b) negative electrode

In the figures the lower connector is the 'concentration' connector which has been defined as:

```
connector ConcentrationNode
  Modelica.SIunits.Concentration C
    "Concentration [mol/m3]";
  flow Real q(final unit="mol/s")
    "Diffusive flow";
end ConcentrationNode;
```

3.2 Electrode Porosity

Both the positive and the negative electrode have a porous structure, thereby increasing the surface area of the electrode with the bulk electrolyte. Due to the reaction that occurs during (dis)charging, the porosity of both electrodes will change. The maximum porosity of the electrode, *i.e.* the maximum amount of open space in the electrode, will be reached when the electrode is fully charged. Assuming the electrode porosity to be dependent on the battery State of Charge (SOC) and having a volume change as function of the amount of discharged energy, the electrode porosity has been described in a Modelica model. The top level icon of this model is displayed in Fig. 4a.

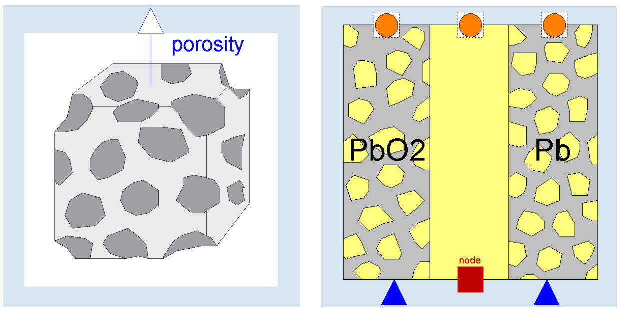


Fig. 4 Top level icon of the (a) electrode porosity model and (b) electrode-electrolyte diffusion model

3.3 Acid Diffusion and Charge Migration

For acid diffusion and charge migration, complex diffusion equations have been set-up based on equations described by Thele in [5]. These equations have been implemented in a Modelica model. The Modelica model, including all equations for acid diffusion and charge migration, is displayed in Fig. 4b. Three concentration connectors can be seen which from left to right represent (i) the positive electrode, (ii) the bulk electrolyte and (iii) the negative electrode. Since the diffusion processes are temperature dependent, the model contains a thermal connector. The inputs to the electrodes are their specific porosities, of which the model is discussed in the previous section.

3.4 Complete OCV Model

By combining the submodels that have been described in the previous subsections, the OCV model can be constructed, which is based on mass transport processes. This model is displayed in Fig. 5.

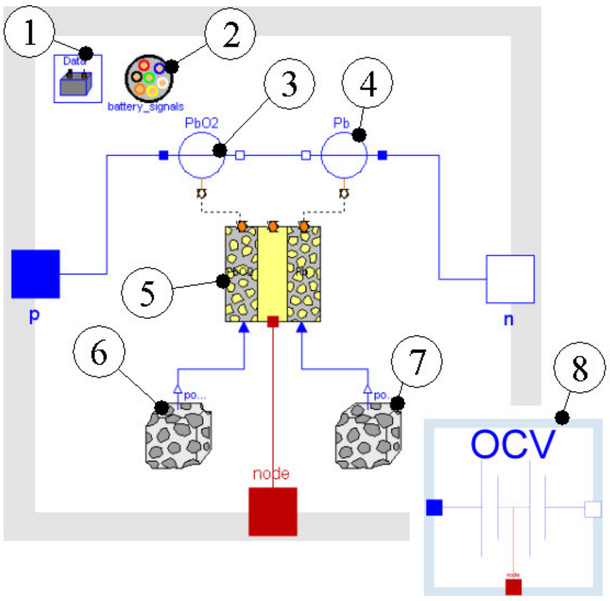
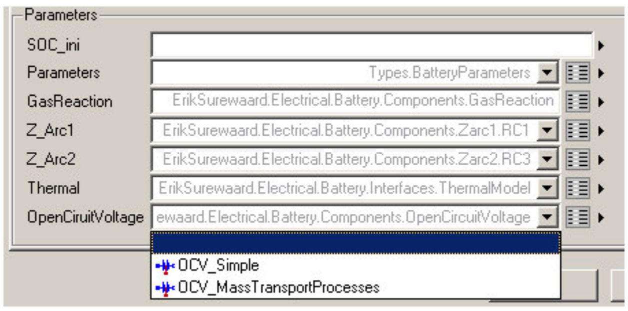


Fig. 5 OCV model including mass transport processes

The numbered submodels in Fig. 5 are a representation of: (1) battery parameters, (2) battery state information to be used for the porosity determination, (3) and (4) the equilibrium potential of respectively the positive and negative electrode, (5) the diffusion processes and charge migration, (6) and (7) the porosity of respectively the positive and negative electrode. Item (8) represents the top level icon.

The battery model that is described by Surewaard in [3] is extended with the OCV model based on mass transport processes by defining it as a replaceable. The parameter window for the battery model, including the replaceable OCV model, is displayed in Fig. 6. The already 'OCV_Simple' model is extended by the model described in this paper: 'OCV_MassTransportProcesses'.


Parameters

SOC_ini	
Parameters	Types.BatteryParameters
GasReaction	ErikSurewaard.Electrical.Battery.Components.GasReaction
Z_Arc1	ErikSurewaard.Electrical.Battery.Components.Zarc1.RC1
Z_Arc2	ErikSurewaard.Electrical.Battery.Components.Zarc2.RC3
Thermal	ErikSurewaard.Electrical.Battery.Interfaces.ThermalModel
OpenCiruitVoltage	ewaard.Electrical.Battery.Components.OpenCircuitVoltage

OCV_Simple
OCV_MassTransportProcesses

Fig. 6 Zoom of the parameter window of the battery model in which the OCV replaceable is highlighted

4 Heated Front Windscreen

A component in the electric powernet, which consumes a significant amount of electric power and has a relatively long thermal time constant, is the electrical heated front windscreen.

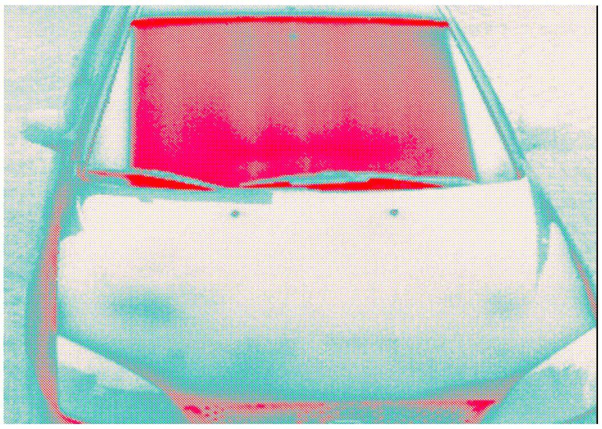


Fig. 7 Electric heated front windscreen: the area that is heated

The electric heated front windscreen basically consists of a sandwich of materials: a Polyvinyl Butyral (PVB) layer on which the tungsten heating wires are placed, sandwiched between two glass layers. A schematic of the layered structure is displayed in Fig. 8.

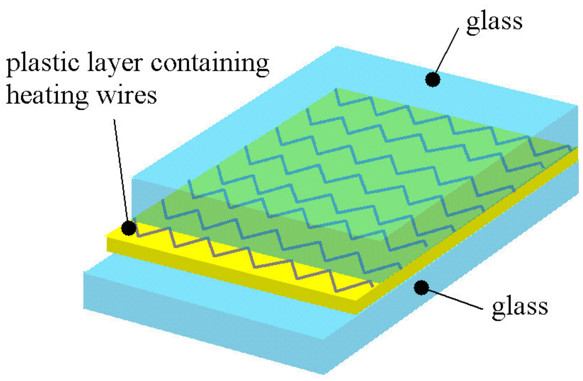


Fig. 8 Schematic overview of the different layers in an electric heated windscreen

The two-dimensional heat transfer model that is set-up for the electric heated windscreen is displayed in Fig. 9. Basically each material layer is modeled by taking the thermal mass in the center of the layer. The thermal mass is connected to the outer surface of the layer by it by two thermal conduction elements, each having half the thickness of the total layer.

Apart from components of the `Modelica.Thermal.HeatTransfer` library, two new components have been developed: (1) a thermal mass representing the ice layer including the phase change from ice to water at 0°C, and (2) a thermal wire component, which converts electric power to a heat flow. The left and right thermal connectors in Fig. 9 represent respectively the inner and outer surroundings of the vehicle. Heat transfer between the screen surface and the surroundings takes place via both convection and thermal radiation.

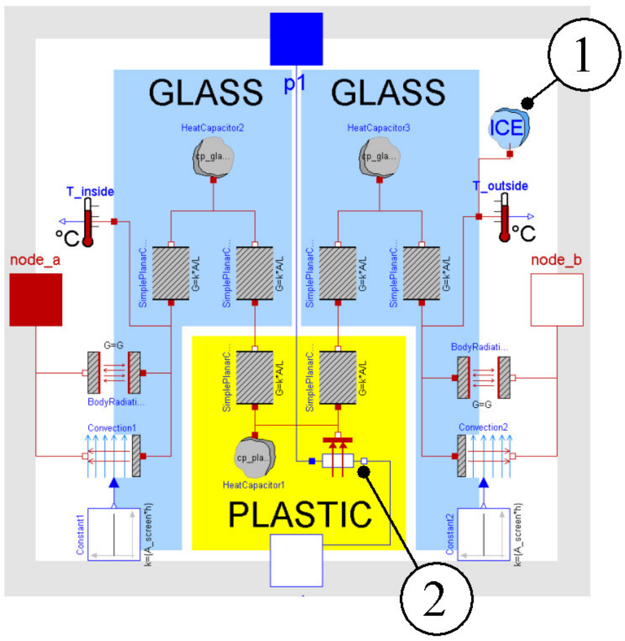


Fig. 9 Electro-thermal model for the electric heated windscreen

For the material constants and dimensions, use has been made of data supplied by the manufacturer of the heated screen and from literature. The model is validated by comparing simulated data with in-vehicle measured data with the electric heated windscreen active. The windscreen temperature is measured with a thermocouple attached to the inner surface of the windscreen (Fig. 10).

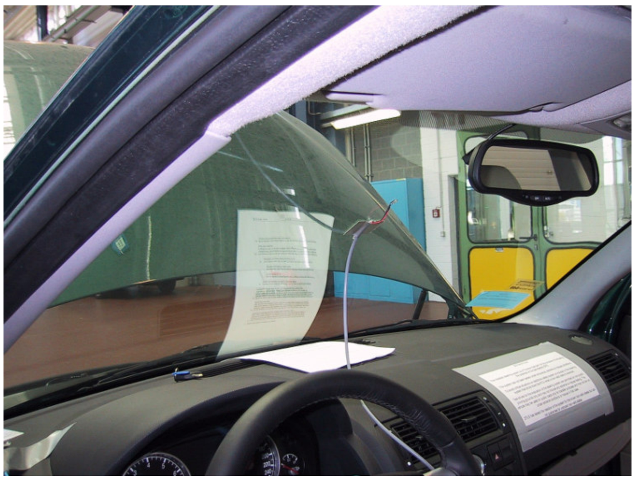
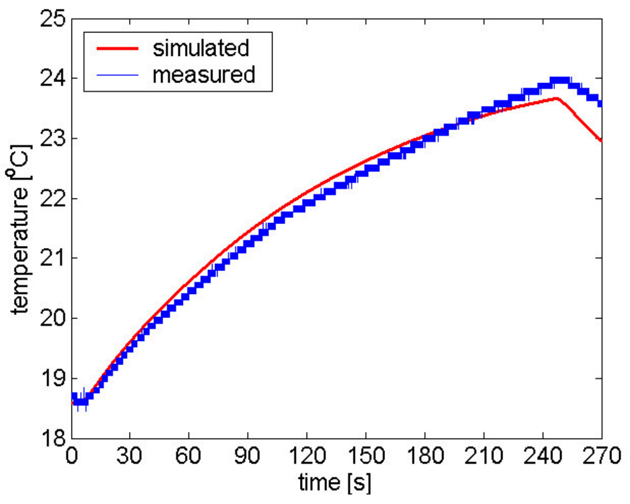
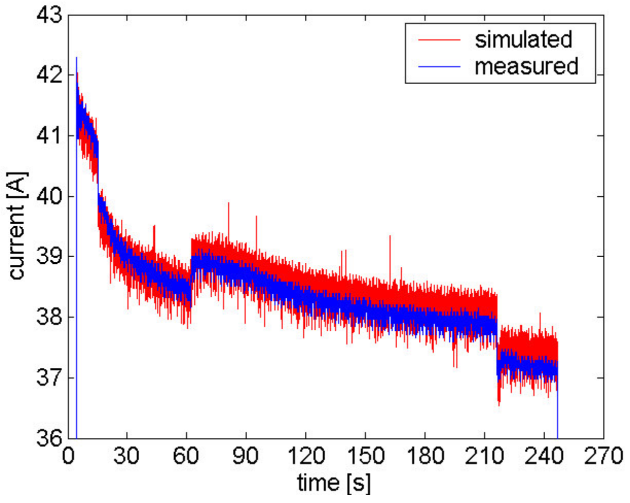


Fig. 10 Thermocouple attached to the inner surface of the windscreen

A measurement has been performed where the windscreen was initially at room temperature (approx. 18.5°C). The measured voltage at the terminals of the windscreen is used as input for the model. The simulated current and temperature have been compared with the measured current and temperature. The comparison of these results is displayed in Fig. 11 (heated windscreen was active in the timeframe between 5 and 250 seconds).



(a)



(b)

Fig. 11 Comparison between simulated and measured data of the heated windscreen: (a) temperature, (b) current

It can be seen from Fig. 11 that the simulated results correspond well with the measured data.

5 ModelicaVMA - Vehicle Idle Model

For the simulations, use will be made of the Vehicle Model Architecture (VMA), which is based on the description by Tiller in [1]. Since we are interested in the engine idle state of the vehicle, the top level VMA model can be simplified to the one displayed in Fig. 12. The idle speed model contains the following physical plant and controller models: (1) driver, (2a) accessory drive, (2b) accessories controller, (3a) powerplant, (3b) powerplant controller, (4) transmission, (5a) electrical system, (5b) electrical system controller and (6) top level controller. The models of above-mentioned subsystems will be discussed in the following subsections.

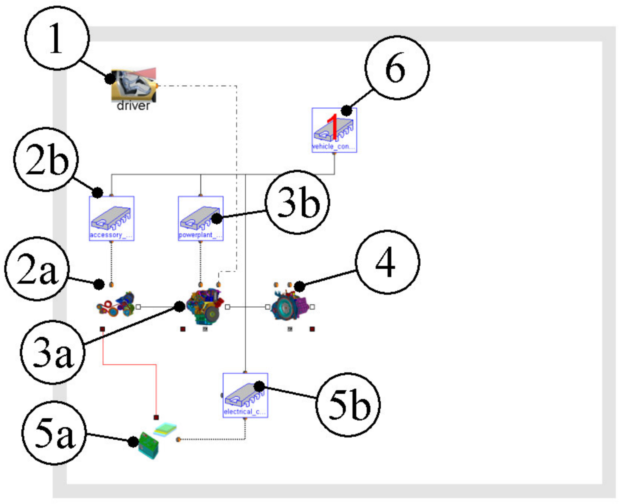


Fig. 12 Top level VMA model for the engine idle state

5.1 Driver

The driver subsystem is represented by item (1) in Fig. 12. Since the simulation will be performed with the engine idling, the driver subsystem will output a closed throttle position.

5.2 Accessories

The accessory subsystem and the accessory controller are represented by respectively items (2a) and (2b) in Fig. 12. The accessory subsystem includes a table lookup based generator model and also includes the belt losses. The top level icon of the accessory subsystem is displayed in Fig. 13a.

5.3 Powerplant

The powerplant subsystem and controller are represented by respectively items (3a) and (3b). An existing Simulink based model of an engine including its controllers (*e.g.* idle speed controller), which is used for fuel economy simulations at Ford, is converted to ModelicaVMA. The top level icon of the powerplant subystem is displayed in Fig. 13b.

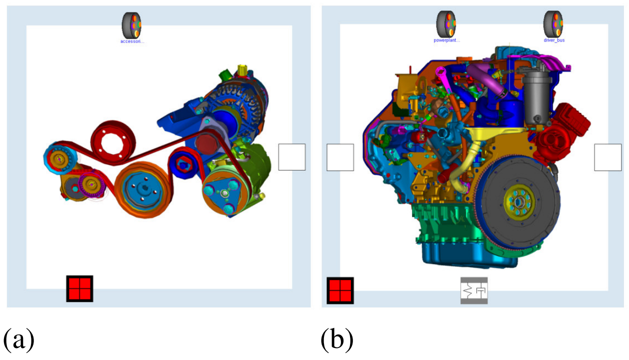


Fig. 13 Top level icons of (a) the accessory subsystem and (b) the powerplant subsystem

5.4 Electrical

The electrical subsystem and controller are represented by respectively items (4a) and (4b) in Fig. 12. The model of the electrical subsystem is displayed in more detail in Fig. 14 and includes the following models: (1) activation signal for the electrical windscreen, (2) switch of the electrical windscreen, (3) controllable PWM switch, (4) residual electrical loads, (5) battery, (6) heated windscreen and (7) top level icon of the electrical subsystem.

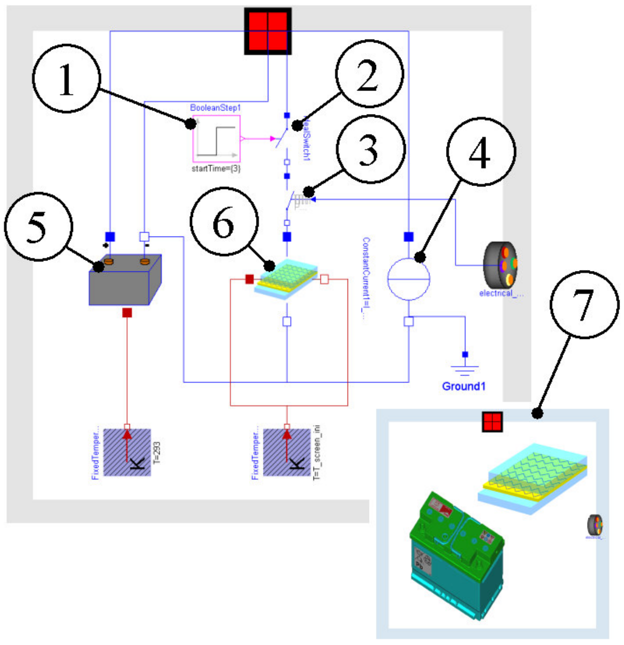


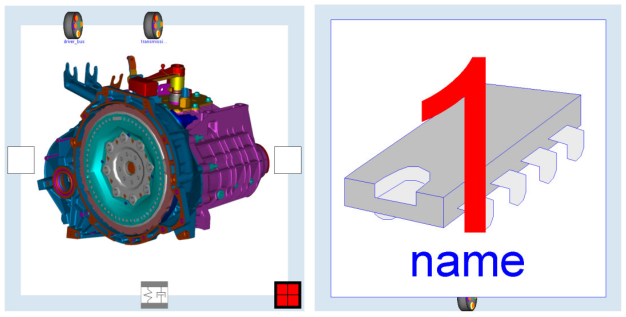
Fig. 14 Electrical subsystem containing battery, switches, heated windscreen and residual electrical loads

The strategy is to initiate the simulation with the heated windscreen inactive. After 3 seconds, the switch of the electrical windscreen will be closed so that it becomes active. The residual electrical loads are approximated by in-vehicle measured loads (having the electric heated screen inactive): 55A from 0-

100s, 40A during 100-200s and 20A after 200s. The reason for the high load current during the first 200 seconds is that the glow plugs are active (Diesel engine).

5.5 Transmission

The transmission subsystem is represented by item (4) in Fig. 12. For the idle speed simulations, it is modeled by having a closed clutch having the neutral gear engaged. The transmission subsystem includes both the engine and gearbox sided inertia and a table lookup model for the spinning losses in the neutral gear. The top level icon of the transmission subsystem is displayed in Fig. 15a.



(a) (b)

Fig. 15 Top level icons of (a) the transmission subsystem and (b) a top level controller implementation

5.6 Top level Controller

The top level controller is represented by item (6) in Fig. 12. Three top level controller models have been developed, each having a different voltage control strategy. The voltage control strategies will be discussed in the following section. The top level icon of on of the controllers is displayed in Fig. 15b.

6 Voltage Control Strategies

The top level controller, which is also known as the Vehicle System Controller (VSC), will control the idle speed of the engine, the voltage setpoint of the generator and if available the PWM frequency of the electric heated windscreen. Three control strategies will be investigated:

6.1 Strategy 1 - Conventional

Conventional 'strategy' where the idle speed is independent of the saturation of the generator. The engine idle speed setpoint will be 750 rpm.

NOTE: the PWM switch as displayed by item (3) in Fig. 14 is not used in this strategy since it will not be available in a conventional vehicle.

6.2 Strategy 2 - Idle Speed Control

This strategy is based on the fact that the maximum generator output current can be increased if the idle speed is increased. The principle is displayed in Fig. 16. When the idle speed would be kept constant at say 750 rpm, the maximum output current will be approximately 70 A. When however the idle speed would be increased, the maximum generator output current also increases. Raising the idle speed above 1500 rpm would no benefits for the generator displayed in Fig. 16.

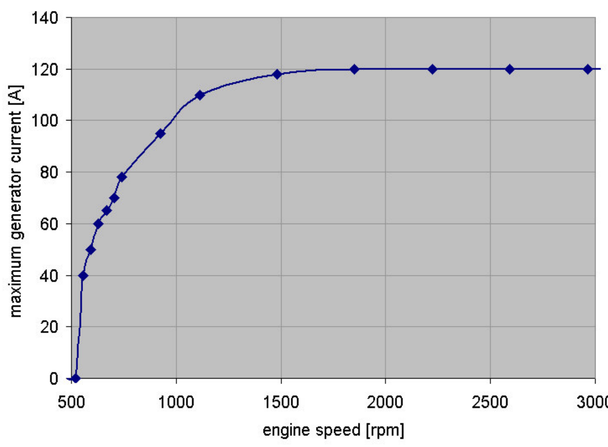


Fig. 16 Maximum generator output current as function of engine speed

The strategy that is implemented is that if the vehicle is in idle, the engine idle speed is controlled to reach a generator saturation of 95%. The window in which the idle speed is allowed to be changed is limited by a lower boundary of *e.g.* 750 rpm and a upper boundary of *e.g.* 1500 rpm (cf. Fig. 16).

NOTE: the PWM switch as displayed by item (3) in Fig. 14 is not used in this strategy since it will not be available in a conventional vehicle.

6.3 Strategy 3 - Pulse Width Modulation

Pulse Width Modulation (PWM) switches can be added to specific electric loads in the powernet as has been proposed by for instance Graf in [7]. The powernet layout for this variant is displayed in Fig. 17.

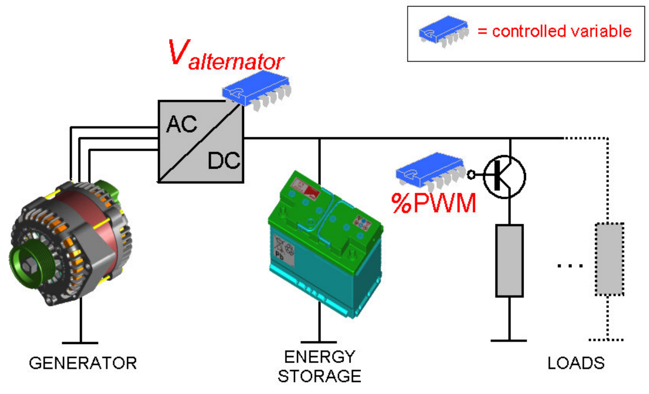


Fig. 17 Powernet variant in which next to the generator voltage also the electric power to a (group of) loads can be controlled

By adding a controllable PWM switch, an additional control variable is introduced in the electric powernet next to the generator voltage. By reducing the electric power flowing to (a group of) loads, the total requested electric power can be controlled. In this way exceeding the maximum electric power that can be delivered by the generator can be prevented. According to Rienks [8], an approach is to add PWM switches to comfort loads, *e.g.* seat heating and screen heating, since these loads have a relative long time constant. Temporarily reducing the amount of electric power flowing to comfort loads will not affect customer acceptance as badly as loosing control of the powernet voltage and by that cause for instance light flickering.

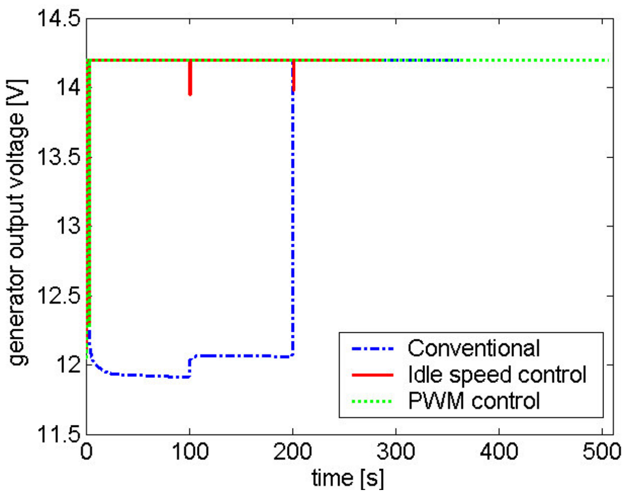
In strategy 3, the PWM switch to the electric heated screen is controlled in such a way that the maximum output current of the generator is not exceeded: the power flowing to the heated screen is reduced to prevent the generator to saturate and loose control of the powernet voltage. The engine idle speed setpoint is kept constant at 750 rpm in this strategy. The setpoint for the generator setpoint is set to 95%.

7 Simulation Results

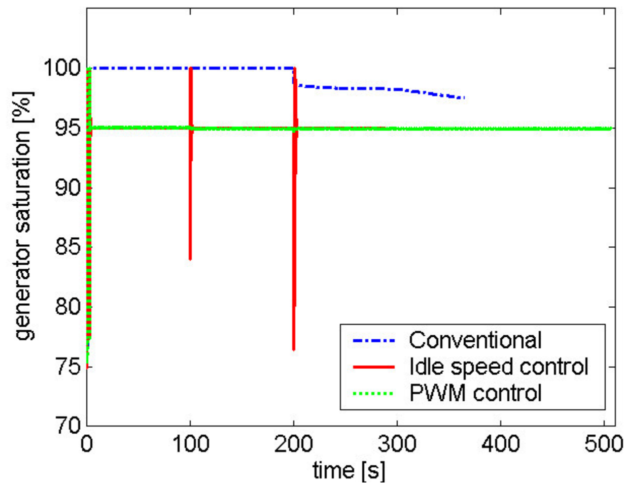
One of the advantages of using the ModelicaVMA structure is the fact that all models can be redefined since they are defined as replaceable. The simulation results that will be described in this sections, have been obtained by making use of the ModelicaVMA structure with the subsystem and controller models from Section 5 and the control strategies from Section 6. For the model parameters, use has been made of real vehicle data (Diesel engine, 120A generator and a 400W electrical heated front screen). A 12V lead-acid battery parameter set has further been used in the simulations: the initial battery State of Charge (SOC) is taken 70% and the initial temperature equal to that of the frozen windscreen: -3°C.

For the simulations the initial condition is a vehicle that has a frozen windscreen (-3°C, 200µm thick ice layer). The electrical heated windscreen will be activated after 3 seconds after the simulation is initiated. The simulation will be stopped when the outer surface temperature of the heated windscreen has reached 3°C. The complete simulation will take place with the engine in idle state. The output data of interest is (i) the cumulative fuel consumption, (ii) the voltage of the powernet, (iii) the engine idle speed and (iv) the time before the outer surface temperature of the windscreen has reached 3°C. The results for the three different strategies are as follows:

7.1 Voltage Stability and Generator Saturation



(a)



(b)

Fig. 18 (a) Powernet voltage for the three different strategies, (b) generator saturation

It can be seen from Fig. 18a that operating the electric heated windscreen with the conventional strategy will result in loosing control of the powernet voltage. It can be seen in Fig. 18b that this is caused by the fact that the generator is saturated in this case. The amount of power requested by the electric powernet exceeds in the conventional case the power that can be delivered by the generator . Therefore the generator setpoint can not be followed and the powernet voltage will drop to the battery voltage. After 200 seconds the glow plugs will become inactive and the generator is from this point on able to supply the total requested electric power and therefore follow its voltage setpoint.

With the two other strategies (*i.e.* idle speed control and PWM control), the generator saturation can be controlled to 95% and therefore the voltage of the powernet can be maintained at 14.2V.

7.2 Effect of Engine Idle Speed Control

Since the second strategy makes use of engine idle speed control to increase the generator output, it is interesting to see the difference in the engine speed for the three strategies. Fig. 19 shows this. Where the engine speed remains 750 rpm with the conventional and the PWM controlled strategy, the engine speed with the idle speed control strategy is increased to improve the generator output. The stepped decrease of the engine speed can be explained by the residual loads (cf. Section 5.4): 55A from 0-100s, 40A during 100-200s and 20A after 200s.

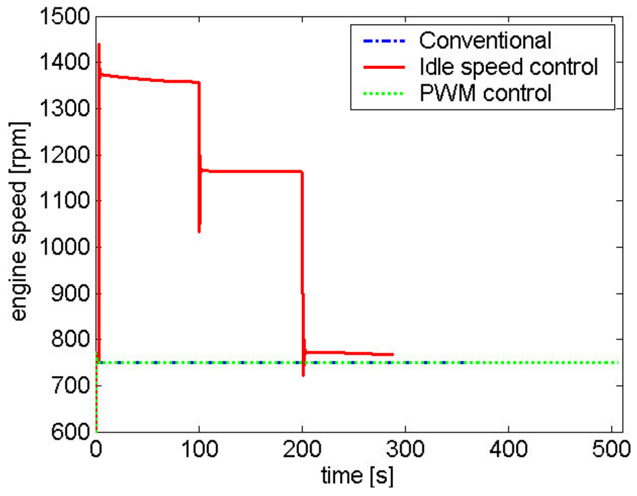


Fig. 19 The effect of engine idle speed control

7.3 Effect of PWM Control

As discussed in Section 6.3, adding a PWM switch to the heated windscreen can be used to reduce the load and therefore maintain powernet voltage stability. Dependent on the amount of power requested, the generator saturation is controlled to 95% by changing the PWM frequency of the electric heated windscreen. Fig. 20 shows the simulated PWM frequency. Again the steps are caused by the reduction of the residual loads (as also explained in the previous section).

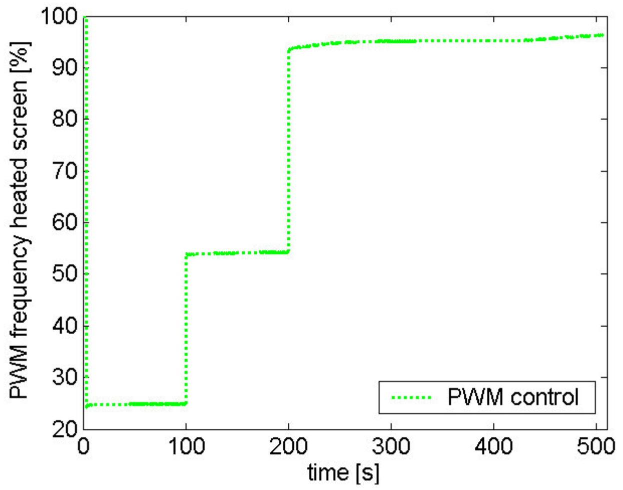


Fig. 20 Percentage of maximum load that the electric heated windscreen is operated with (PWM)

7.4 Heating Performance and Fuel Economy

Other factors of interest are the time before the temperature of the outer surface of the heated screen has reached 3°C and how many fuel is used until this point is reached. Fig. 21 shows the temperature of the outer surface. It can be clearly seen that there is a significant difference between the three strategies: the strategy with idle speed control is the fastest (total time is 290 seconds), followed by the conventional strategy (total time is 365 seconds) and the PWM controlled strategy (total time is 507 seconds).

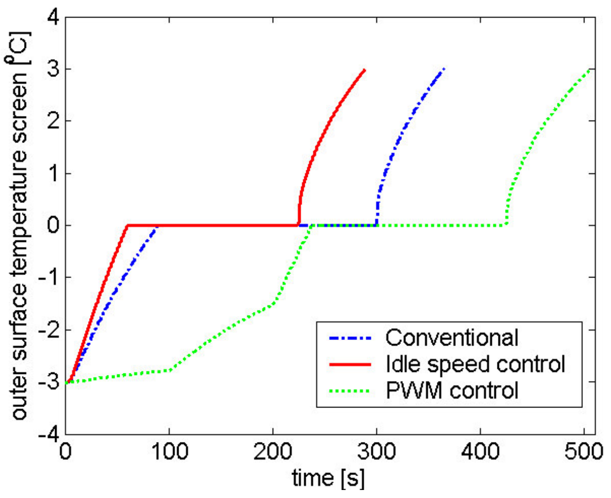


Fig. 21 Outer surface temperature of the electric heated windscreen.

The cumulative fuel consumption during the heating of the frontscreen is displayed in Fig. 22. The gradient for the idle speed control can be explained by the fact that the fuel consumption at higher engine speeds is larger (due to engine / transmission / generator losses). The total amount of fuel used is: 50 gram for the conventional strategy, 52 gram for idle speed control and 69 gram for the PWM controlled case.

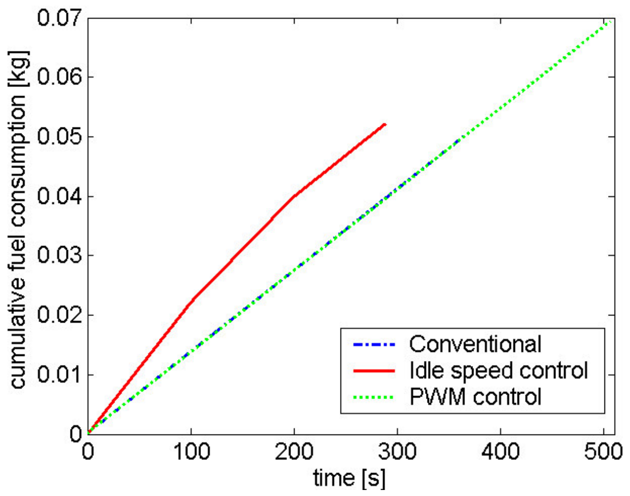


Fig. 22 Cumulative fuel consumption during the heating process

7.5 Summarized Results

The following table summurizes the results from the previous subsections.

Table 1 Summarized results

	Conv.	Idle speed	PWM
Stable voltage? (yes/no)	NO	YES	YES
Time to reach 3°C [s]	365	290	590
Fuel used [g]	50	52	69

The idle speed control is evidently the preferred solution: no additional components are needed (as PWM switches). By controlling the idle speed, the voltage can be kept stable, the (time) performance of the heated windscreen can be increased and that all without paying a fuel penalty compared with the conventional strategy.

Acknowledgements

The authors wish to thank Eckhard Karden, battery expert of the Energy Management Group of Ford Motor Company and Dirk Uwe Sauer of the RWTH Aachen University for their great ideas in battery modeling. Also Daniel Kok, Michael Tiller, Serve Ploumen, Maurice Rienks and Thomas Gerhards are thanked for their support and ideas.

References

1. **Tiller, M., Bowles, P.** and **Dempsey, M.,** "Development of a Vehicle Model Architecture in Modelica", ", Proceedings of the 3rd International Modelica Conference, pp. 75-85, Linköpig Sweden, 2003 http://www.modelica.org/Conference2003/papers /h32_vehicle_Tiller.pdf
2. **Surewaard, E., Tiller, M.** and **Linzen, D.**, "A Comparison of Different Methods for Battery and Supercapacitor Modeling", SAE paper 2003-01-2290, 2003
3. **Surewaard, E., Karden, E.** and **Tiller, M.**, "Advanced Electric Storage System Modeling in Modelica", Proceedings of the 3rd International Modelica Conference, pp. 95-102, Linköpig Sweden, 2003 http://www.modelica.org/Conference2003/papers /h11_Surewaard.pdf
4. **Surewaard, E., Kok, D.,** and **Tiller, M.**, "Engine Cranking: Advanced Modeling and an Investigation of the Influence of the Initial Crank Angle and Inertia", SAE paper 2004-01-1875, 2004
5. **Thele, M., Buller, S., Sauer, D.U., De Doncker, R.W.** and **Karden, E.**, "Hybrid Modeling of Lead-Acid Batteries in Frequency and Time Domain", Journal of Power Sources, Article in Press, 2005
6. **Bode, H.,** "Lead-acid Batteries", Wiley, 1977
7. **Graf, A.**, "Power Semiconductors to Drive Energy Management", Second Aachener Elektroniksymposium, Institut fuer Kraftfahrwesen, Aachen, 2004
8. **Rienks, M**. and **Kok, D.**, "Development and Implementation of Electrical Power Distribution Management", Second Aachener Elektroniksymposium, Institut fuer Kraftfahrwesen, Aachen, 2004

Inversion of Vehicle Steering Dynamics with Modelica/Dymola

Tilman Bünte[1] Akin Sahin[2] Naim Bajcinca[1]
[1] German Aerospace Center (DLR), Institute of Robotics and Mechatronics, Oberpfaffenhofen, D-82230 Wessling, Germany
[2] University of Siegen

Abstract

The task of steering a vehicle is an exercise which is usually considered hierarchically in terms of the two subtasks *path planning* and *path following*. With the driver in the loop some essential man dependent tasks such as sensing, information processing, and motor function affect the steering quality. In case of simulations, the same applies correspondingly for driver models. In this paper the aim is to investigate vehicle steering dynamics independent of any driver-related properties. The path is therefore assumed given by a *reference trajectory* together with a speed profile. The steering angle which is necessary for exact or at least approximate path following is sought after. This allows for plausible comparative assessment of different vehicle's steering dynamics in terms of the demanded steering effort for a certain maneuver. On the other hand, this approach requires dynamic inversion of vehicle steering dynamics which represents the main focus of this paper. Two vehicle models, the common single track model and a detailed model from the Modelica vehicle dynamics library are investigated. Since exact inversion of the detailed vehicle model turns out not to be feasible, approximate inversion is accomplished by means of a novel control structure called *inverse disturbance observer*. Simulations of a double lane change maneuver are conducted for illustration. Finally, wavelet power spectra of the steering angle signal are used for steering effort assessment.

1 Introduction

In the usual way of simulating vehicle models, a driver module provides inputs to the vehicle in terms of the steering wheel angle and gas/brake pedal position. As a result of this forward simulation, a trajectory of the vehicle is obtained. With *inverse simulation of vehicle steering dynamics* for a given desired trajectory and velocity profile, the aim is computation of the steering wheel angle input required from the driver.

Reference trajectories may be defined in terms of the curvature ρ as a function of the arc length λ. The reference trajectory of a *double lane change* maneuver is presented as an example. For tracking the reference path with a lateral displacement τ, instead of Cartesian coordinates a trajectory based coordinate system (λ, τ) is employed. In section 2, the representation of reference trajectories and the trajectory based coordinate system are explained in detail. For developing and investigation of the concept of vehicle steering dynamics inversion, two vehicle models are considered: the common linear *single track model* and a *detailed vehicle model* from the Modelica vehicle dynamics library. These models are introduced in section 3.

If some requirements like regularity and uniqueness of solutions hold, inverse models may be obtained in Modelica by simply providing equations for the outputs and removing an adequate number of equations for the original inputs. The perfect inverse of the detailed vehicle model from the Modelica vehicle dynamics library (using rigid linkages for the suspensions) is easily achieved. However, it turns out that the detailed vehicle model is non-minimum phase. Therefore, the inverse vehicle model is unstable and can not be simulated. To overcome this problem, as a trade-off we use approximate inversion of models, such that the resulting system is stable. For this purpose, a novel high gain control scheme, the *inverse disturbance observer* [1] is utilized. The inverse disturbance observer combines exact inversion of a simplified model as feedforward control and high-gain feedback for robust tracking performance. Simulation results for a double lane change maneuver illustrate the effectiveness of the applied approach in section 4.

Steering dynamics of different vehicles may be compared in terms of the steering inputs being necessary to perform a specific maneuver. The objective is to establish a method which can be used to assess the steering dynamics of vehicles with specific modifications like active steering control. Therefore, in section 5 the double lane change steering inputs are compared for two single track models with significantly different loading. For analyzing the steering efforts of the two vehicles, wavelet transform is applied. Conclusions on easiness or difficulty for a driver when driving these cars can be drawn from wavelet power spectra.

2 Reference trajectories and coordinate system for path tracking

For inverse vehicle simulations investigated in this paper, the vehicle's speed and a reference trajectory for the vehicle's position are given. The reference point on the vehicle representing its position is assumed to be located at the center of the front axle. With perfect inversion, this reference point exactly follows the reference trajectory, otherwise the task is to make the lateral displacement from the reference trajectory as small as possible. Therefore, this problem is closely related to the problem of path tracking for automatic car steering.

2.1 Reference trajectories

In this paper, the reference trajectory is defined in Cartesian coordinates $(x_{ref}(\lambda), y_{ref}(\lambda))$ as a function of the arc length λ. Any reasonable trajectory of a vehicle cruising at finite speed may be assumed continuous and at least twice differentiable. With $\rho(\lambda)$ and $\phi(\lambda)$ denoting the curvature and the track angle respectively, the following relations hold:

$$\begin{bmatrix} \phi' \\ x'_{ref} \\ y'_{ref} \end{bmatrix} := \frac{\mathrm{d}}{\mathrm{d}\lambda} \begin{bmatrix} \phi \\ x_{ref} \\ y_{ref} \end{bmatrix} = \begin{bmatrix} \rho \\ \cos(\phi) \\ \sin(\phi) \end{bmatrix} \tag{1}$$

Our approach is to start from a definition of $\rho(\lambda)$ and solve (1) for ϕ, x_{ref}, and y_{ref} using appropriate initial conditions. See Fig. 1 for an exemplary definition of $\rho(\lambda)$ and the resulting reference trajectory (x_{ref}, y_{ref}) for a *double lane change* maneuver.

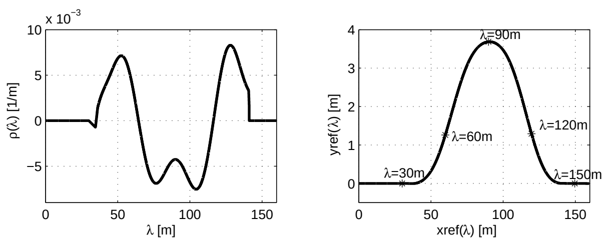
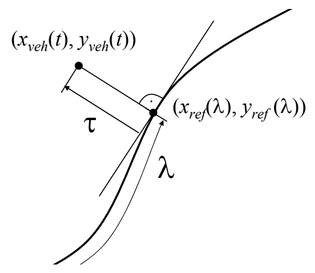


Figure 1: Curvature (left) and reference trajectory (right) for a double lane change.

2.2 Coordinate system for path tracking

For the mathematics involved with the path tracking problem, it is not expedient to describe the vehicle's position with Cartesian coordinates. Therefore, rather a trajectory based coordinate system (λ, τ) is employed, see Fig. 2. It consists of the arc length λ referring to the point $(x_{ref}(\lambda), y_{ref}(\lambda))$ on the reference trajectory which is closest to the vehicle and the lateral displacement τ, also referred to as tracking error. That is, τ is the signed closest perpendicular distance to the reference trajectory.



$(x_{veh}(t), y_{veh}(t))$
$(x_{ref}(\lambda), y_{ref}(\lambda))$
τ
λ

Figure 2: Vehicle position in trajectory based coordinates (λ,τ).

A coordinate transformation between Cartesian coordinates (x_{veh}, y_{veh}) and trajectory based coordinates (λ, τ) needs to be accomplished. The unit vector $[-y'_{ref}, x'_{ref}]^T$ is perpendicular to the reference trajectory and is oriented to the left hand side of the trajectory. Hence, the distance between the position of the vehicle and the reference trajectory may be written as

$$\begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} := \begin{bmatrix} x_{veh} - x_{ref} \\ y_{veh} - y_{ref} \end{bmatrix} = \tau \begin{bmatrix} -y'_{ref} \\ x'_{ref} \end{bmatrix}. \tag{2}$$

The coordinate transformation can be done in the following way: Elimination of τ in (2) yields the nonlinear equation

$$\Delta x x'_{ref} + \Delta y y'_{ref} = 0 \tag{3}$$

which can be solved for λ.

Using the fact $x'^2_{ref} + y'^2_{ref} = 1$ (see (1)) together with (2) yields

$$\tau = \Delta y \, x'_{ref} - \Delta x \, y'_{ref}. \tag{4}$$

Multiple solutions may exist for equation (3). Only the closest solution where $|\tau|$ has its minimum value is relevant and is to be selected. This ambiguity makes evident that the introduced trajectory based coordinate system is only suitable in a sufficiently narrow vicinity of the reference trajectory. This assumption holds, since accurate path tracking is aimed at.

Later, the coordinate transformation will be considered a part of the vehicle models. Linearization, as may be necessary, is done in the following way. A virtual object exactly following the reference trajectory as defined in section 2.1 senses a lateral acceleration given by

$$a_{yref} = \rho(\lambda)\,\dot{\lambda}^2 \tag{5}$$

with $\dot{\lambda}$ denoting the object's speed. Under the assumption of small tracking error τ and small chassis side slip angle the lateral acceleration of a vehicle closely tracking the reference trajectory with speed v (entailing $v \approx \dot{\lambda}$) can therefore be represented by

$$a_{yveh} = a_{yref} + \ddot{\tau}. \tag{6}$$

Hence,

$$\tau = \frac{1}{s^2}\left(a_{yveh} - a_{yref}\right). \tag{7}$$

2.3 Implementation in Modelica

During the simulation the actual value of λ needs to be solved from equation (3) for each integration step. This is automatically done by Dymola, provided that $x_{ref}(\lambda), y_{ref}(\lambda), x'_{ref}(\lambda)$ and $y'_{ref}(\lambda)$ are known. Therefore, in our Modelica model we provide look-up tables depending on λ that contain values for x_{ref}, y_{ref}, and ϕ each with the derivative w.r.t. λ. These look-up tables are pre-calculated from (1) in Matlab, saved to mat-files, and used in Modelica/Dymola for interpolation at simulation time. According to our experience, the selection of the proper solution of (3) does not cause any problems since the solution for λ is continuously and monotonically increasing along the followed reference trajectory.

A special problem occurs with the simulation of perfectly inverted vehicle models (see section 4.1). In general, for inverse simulations executed in Dymola the given output where required needs to be differentiated one or multiple times w.r.t. time. The look-up tables we use, however, only provide derivatives w.r.t. λ since the reference trajectory does not depend on time. Therefore, if needed the time derivatives are supplied by special functions[1]. They are calculated from the actual value of $\dot{\lambda}$ and the corresponding derivatives w.r.t. λ. If necessary, higher derivatives w.r.t. λ are supplied in extra columns in the look-up tables. As an example the Modelica code

```
dxdlambda = TableFunc.y(tableIDintx, 3,lambda);
```

is used to retrieve x'_{ref} from the look-up table (referred to by its identifier *tableIDintx*, 3rd column stores first derivative) for the actual value of λ. This is the used package:

```
package TableFunc
 function y // here y means a generic output
   input Integer ID, index;
   input Real u;
   output Real y;
 external "C" y=
       dymTableIpo1_my(ID,index,u);
   annotation (derivative=ydot);
 end y;
 function ydot
   input Integer ID,index;
   input Real u,dudt;
   output Real dydt;
 protected
   Real dydu;
 algorithm
   dydu :=
       dymTableIpo_my(ID,index+1,u);
   dydt := dydu*dudt;
   annotation
       (derivative(order=2)=yddot);
 end ydot;
 function yddot
 ... // analogous to ydot
 end yddot;
 function dymTableIpo_my
   input Integer ID,index;
   input Real u;
   output Real y;
 external "C" y=
       dymTableIpo1_my(ID,index,u);
 end dymTableIpo_my;
end TableFunc;
```

The C function *dymTableIpo1_my* provides the table look up. It corresponds to *dymTableIpo1* which can be found in *dymtable.c* in the Dymola source directory. Note the annotations. The standard way of differentiating inputs from look-up tables is thus replaced by use of the function *ydot* (*yddot* respectively) while applying the chain rule.

[1]The authors thank Andreas Pfeiffer (DLR) for his helpful support.

3 Vehicle Models

Vehicle steering dynamics in this paper are explored using two models with essentially different levels of detail. Firstly, for basic considerations the very simple *single track model* is implemented in Modelica. Secondly, for more advanced investigations, a *detailed vehicle model* from the Modelica vehicle dynamics library is used. In both cases, the trajectory which normally is the output of a driving maneuver is defined together with a speed profile and the necessary steering input is asked for. Therefore, both the reference trajectory and the coordinate transformation as defined in section 2 are added to the model description.

3.1 The linear single track model

The single track model [2] is a simple linear vehicle model commonly used in the analysis and control design of lateral and yaw dynamics. The wheels of the each axle are considered lumped together in the center of the vehicle. The roll, pitch, and heave motions are neglected. In Fig. 3 the single track model is illustrated. Its major variables and geometric

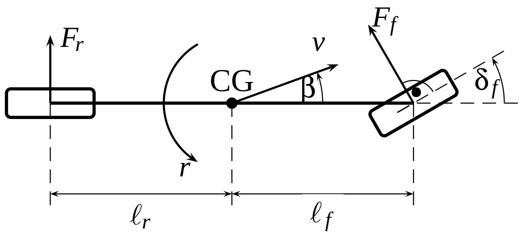


Figure 3: Single track model.

parameters are

$F_f (F_r)$	lateral wheel force at front (rear) wheel
ψ	yaw angle
$r = \dot{\psi}$	yaw rate
β	chassis side slip angle at center of gravity (CG)
v	speed, i.e. magnitude of velocity vector at CG
$\ell_f (\ell_r)$	distance from front (rear) axle to CG
i_L	steering gear ratio
δ_f	front wheel steering angle
$\delta_S = i_L \delta_f$	steering wheel angle

Linearizing the tire force characteristics lateral wheel forces at the front and rear wheels can be written as

$$F_f(\alpha_f) = \mu c_{f0} \alpha_f, \quad F_r(\alpha_r) = \mu c_{r0} \alpha_r \tag{8}$$

with c_{f0}, c_{r0} being the tire cornering stiffnesses at the front and the rear wheels, μ the road adhesion factor and α_f and α_r the tire side slip angles at the front and the rear wheels given by

$$\alpha_f = \delta_f - \left(\beta + \frac{\ell_f}{v} r\right), \quad \alpha_r = -\left(\beta - \frac{\ell_r}{v} r\right) \tag{9}$$

The mass of the vehicle is m and J is the moment of inertia w.r.t. a vertical axis through the CG. Under the assumptions of small side slip and steering angles and slowly varying velocity the linearized equations of motion are

$$\begin{bmatrix} mv(\dot{\beta} + r) \\ J\dot{r} \end{bmatrix} = \begin{bmatrix} F_f + F_r \\ F_f \ell_f - F_r \ell_r \end{bmatrix} \tag{10}$$

The lateral acceleration of the vehicle at the front axle is

$$a_{yveh} \quad = \quad v(\dot{\beta} + r) + \dot{r} \ell_f. \tag{11}$$

For linear considerations, (7) may be used for the lateral position w.r.t. the trajectory based coordinate system. Otherwise, the velocity of the vehicle's CG is

$$\begin{bmatrix} \dot{x}_{CG} \\ \dot{y}_{CG} \end{bmatrix} = v \begin{bmatrix} \cos(\psi + \beta) \\ \sin(\psi + \beta) \end{bmatrix} \tag{12}$$

The front axle (i.e. vehicle reference point) position is

$$\begin{bmatrix} x_{veh} \\ y_{veh} \end{bmatrix} = \begin{bmatrix} x_{CG} \\ y_{CG} \end{bmatrix} + \ell_f \begin{bmatrix} \cos(\psi) \\ \sin(\psi) \end{bmatrix} \tag{13}$$

The *single track model* will be used later as a simple substitute for the fully detailed standard vehicle dynamics model from the Modelica vehicle dynamics library [3] (which is parametrized as a BMW 3-series car by default). The corresponding parameters for the single track model were determined in [4] and they are also used here: $i_L = 16.94$, l_f=1.0203m, l_r=1.5297m, $m = 1482.9$kg, $J = 2200$kg m^2, $c_{f0} = 91776$ N/rad and $c_{r0} = 77576$ N/rad. Only dry road conditions are considered here, therefore $\mu = 1$.

3.2 Detailed vehicle model

The vehicle dynamics library [3] of Modelica provides models for vehicle dynamics simulation. It consists of a detailed mathematical model comprising the multibody differential equations. Since this library is freely available, documented and well known to the Modelica user community, no further details are stated here. In this paper the Modelica vehicle model described in [4] is used. The standard *chassis level 2* vehicle model is completed by the *simple power train* model and brakes. Furthermore, a PI speed controller sets an adequate gas/brake pedal position and makes the vehicle accurately follow a desired speed profile. Finally, a wheel slip controller approximates the function of an antilock braking system (ABS). In the sequel, this model will be referred to as the *detailed vehicle model*.

4 Perfect and approximate inversion of vehicle steering dynamics

The vehicle models used in this paper (see section 3) are considered as SISO (single input single output) systems with the steering wheel angle δ_S being the input and the lateral displacement τ from the reference trajectory being the output. The ideal conception of the model inversion process (referred to as *perfect inversion*) is to obtain a steering wheel angle signal such that the lateral displacement τ is always zero. Simulations executed with perfectly inverted models are denoted *inverse simulations* here. Inversion of the longitudinal dynamics (i.e. speed) may in general be considered as well. Here, however, we focus on steering (i.e. lateral) dynamics. Along the way, the vehicle speed v is set or controlled to match a given profile $v(\lambda)$ or alternatively $v(t)$. Hence, speed is rather considered a set varying parameter than an input or output. If perfect steering dynamics inversion is not possible, *approximate inversion* is aimed at. That is, the resulting lateral displacement τ and steering wheel angle error respectively should be as small as possible.

For both models, single track model and detailed model, we first try to achieve perfect inversion. As will be shown, this is possible for the single track model. In contrast, perfect inversion of the detailed model turns out not to be feasible in terms of a converging simulation. Therefore, a novel high gain control scheme is applied to approximately invert the detailed model. This approach may incidentally also be applied to the task of high fidelity path tracking for real world automatic car steering.

In the course of this section, simulations of the inverted models are conducted for the purpose of illustration. Exemplarily, the double lane change maneuver introduced with Fig. 1 is considered with a constant speed of 20m/s .

4.1 Perfect inversion of the vehicle models in Modelica/Dymola

The option of perfect inversion of Modelica models has already been exploited in a number of applications such as automatic generation of control laws for the control of aircraft [5] or industrial robots [6]. Inverse models may be obtained in Modelica by simply providing equations for the outputs and relaxing an adequate number of equations for the original inputs. As pointed out in [6], the derivation of the inverse system equations may require to differentiate certain parts of the model equations. Therefore, the model equations need to be continuous and differentiable. Moreover, since it may be necessary to the differentiate the given output signals too, their time derivatives must exist and be provided up to a certain order. Therefore, as explained in section 2.3, look-up tables for the trajectory variables and their derivatives w.r.t. λ are provided in the models together with functions to form the respective time derivatives.

With nonlinear models, for a given output not necessarily any solution in terms of input functions does exist. On the other hand, multiple solutions may exist for the same inverse simulation problem. So far, we have not worked on these questions. We have rather assumed conditions (i.e. moderate lateral acceleration) which do not cause corresponding problems.

One necessary condition for perfect inversion is that the considered input/output dynamics of the model is minimum phase. Otherwise the inverted model is not stable and therefore inverse simulation is not feasible.

4.1.1 Perfect inversion of the single track model

For investigating the perfect inversion of the single track model, the implementation of its equations and its parameters in Modelica as described in section 3.1 is employed. The model includes the reference trajectory look-up tables for the double lane change maneuver and the coordinate transformation (3),(4) introduced in section 2. The set of equations is completed by $\tau = 0$ and $v = 20$m/s and thus the number of equations matches the number of unknowns. The model can be successfully translated and simulated. The resulting front steering angle δ_f is shown in Fig. 4. The parameters of the *light vehicle* are those given in section 3.1. For comparison, the simulation is repeated with a *heavy vehicle*. Its parameters are the same except for double values of mass m and inertia J.

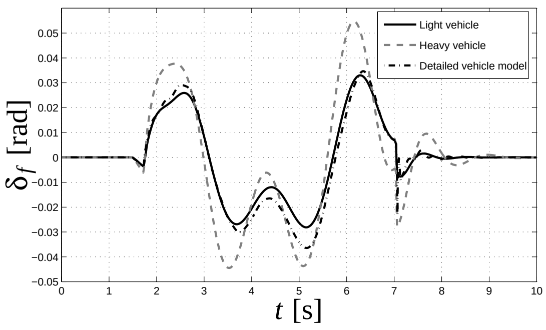


Figure 4: Front steering angle δ_f for the double lane change maneuver (v = 20m/s) obtained by inverse simulation of the single track model. Two parameter sets are used: *light vehicle* and *heavy vehicle*. Also results for the approximately inverted detailed vehicle model (see section 4.2.2) are shown.

4.1.2 Perfect inversion of the detailed vehicle model

The detailed vehicle model is inverted in the same way by adding the equation $\tau = 0$ and by setting the target value for speed control to 20m/s. The steering wheel angle is relaxed, i.e. any direct equation for driver steering input is removed.

We attempted to invert models with different suspensions. With the *SimpleSuspension* the translation of the model was successful. However, the integration in Dymola terminated 0.13s after start of the simulation due to missing convergence of the corrector. With the *MacPhersonSuspension2* Dymola was not able to differentiate some of the model equations, therefore, this inverse model could not be translated successfully. The last-mentioned problem was not investigated further since we found out, that the detailed vehicle model is non-minimum phase which causes stability problems at any rate when simulating its inverse. This is also the reason why the inverse simulation using *SimpleSuspension* did not converge.

For illustration of the non-minimum phase dynamics, the pole-zero-map of the transfer function from steering wheel angle δ_S to lateral displacement τ was investigated. The transfer function was obtained by linearisation about straight driving ($x(\lambda) = \lambda$, $y(\lambda) = 0$, $\delta_S = 0$, $r = 0$, $\psi = 0$, $\tau = 0$, $\dot{\lambda} = v = 20$m/s). The pole-zero map reveals a fast zero at $s \approx 90$ in the right half plane. The corresponding non-minimum phase behavior can be explained by the suspension construction of the steered front axle. It can briefly be depicted imagining an idle vehicle at zero speed. If the steering wheel is turned then the front end of the car moves slightly to the opposite direction due to the suspension's caster characteristic. In normal drive operation, this effect superimposes with the remaining vehicle steering dynamics and results in non-minimum phase behavior. When inverting the model, the right half plane zero becomes a fast unstable pole which makes simulation of the perfectly inverted model impossible. Therefore, in the next section a stable approximately inverted model will be generated using accurate path tracking control. For this purpose, a novel control structure denoted *inverse disturbance observer* is employed.

4.2 Approximate inversion of the detailed vehicle model

4.2.1 Inverse disturbance observer

The inverse disturbance observer (IDOB) was recently introduced in [1] as a modification of the common disturbance observer (DOB) structure. Basically, both

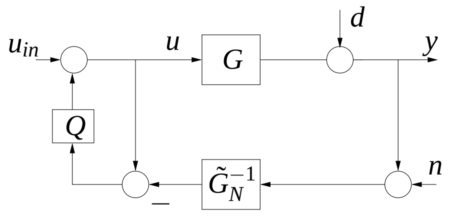


Figure 5: DOB scheme.

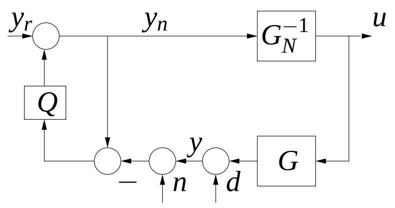


Figure 6: IDOB scheme.

DOB (see Fig. 5) and IDOB (see Fig. 6) are two degree of freedom control structures that combine high-gain and exact model inversion facilities in a simple configuration. The design parameters are an invertible nominal model G_N ($\tilde{G}_N$ respectively) approximating the plant dynamics G (which is assumed to be stable) and a Q-filter which commonly has unity gain and low-pass properties. Compared with DOB, in the IDOB structure the block positions of the plant G and the nominal model G_N are simply interchanged (which partly gives a different meaning to the involved signals). Therefore, with IDOB the inverted nominal model G_N^{-1} is in the feedforward part instead of the feedback as it is the case with DOB.

DOB and IDOB structures are used for different purposes. The aim of the traditional DOB is matching the dynamics of the controlled system to a nominal model G_N. However, in case of IDOB the aim is matching the closed loop dynamics to G^{-1}. Therefore, the IDOB control structure is especially applicable for dynamic model inversion (in this case G represents the model to be inverted) and output tracking problems (in this case G represents a plant).

IDOB combines the facilities of feedforward control using an inverted nominal model of the plant and high gain feedback in a very simple structure while preserving the advantages of each principle. In the IDOB structure G_N^{-1} acts as a feedforward control. The term $G_N^{-1}(s) \cdot y_r(s)$ provides the main portion of the plant input $u(s)$ where y_r is the setpoint for y. The subordinate positive gain feedback loop containing the Q-filter forces this approximate inversion signal to converge to the perfect inversion signal and also provides robustness to the inversion process due to its high gain feedback feature.

The IDOB structure serves as an approximate model inversion method for a model G if the relation between the signal y_r and the plant input u is considered:

$$\frac{u}{y_r} = \frac{1}{G_N(1-Q)+GQ} \tag{14}$$

Recall that Q is a low pass filter with unity gain. The frequency interval between zero and the bandwidth of Q is denoted the *frequency operating domain* of the IDOB. In the frequency operating domain, $Q \to 1$ holds and therefore, $u \to G^{-1} y_r$. At high frequencies, the gain of Q tends to zero, therefore $u \to G_N^{-1} y_r$ which at least provides the input signal based on the model G_N. In the case that G is non-minimum phase and G_N is a minimum phase approximation for G, then by proper choice of the bandwidth of Q the stability of the IDOB system can be ensured. In practice, the bandwidth of Q will be chosen according to a compromise between (robust) stability and (robust) performance.

It can be concluded from (14) that for approximating perfect model inversion $u = G^{-1} y_r$ one of the following two criteria would be sufficient:

$$Q \to 1 \;\text{ or }\; G_N \to G \tag{15}$$

The IDOB structure combines the facilities of both high gain (subordinate loop with $Q \to 1$) and inversion with feedforward control $G_N \to G$ in the same structure. Also it is important to notice that with the IDOB structure, the approximate inverse of the model G is obtained without inverting the model explicitly.

On the other hand, considering y as the output of the system, IDOB becomes a plant controller for output tracking:

$$\frac{y}{y_r} = \frac{G}{G_N(1-Q)+GQ} \tag{16}$$

In the frequency operating domain, $Q \to 1$ holds and therefore, $y \to y_r$ i.e. good output tracking is achieved.

Due to its similar structure, the IDOB holds the known robustness properties of the disturbance observer in terms of disturbance and measurement noise rejection. Hence, the sensitivity (S) and complementary sensitivity (T) functions are the same as with DOB:

$$S = \frac{y}{d} = \frac{G_N(1-Q)}{G_N(1-Q)+GQ} \tag{17}$$

$$T = \frac{y}{-n} = 1 - S = \frac{GQ}{G_N(1-Q)+GQ} \tag{18}$$

Within the IDOB frequency operating domain ($Q \to 1$), disturbances are attenuated ($S \to 0$). For high frequencies ($Q \to 0$), noise is attenuated ($T \to 0$).

4.2.2 Application of IDOB for approximate inversion of the detailed vehicle model

As was shown in the last section, the IDOB needs a nominal model G_N. For approximate inversion of the detailed vehicle model by means of IDOB, the single track model is adopted as nominal model. It is easily invertible as already demonstrated in section 4.1.1. The actual single track model parameters (see section 3.1) were determined for good approximation of the detailed vehicle model [4].

However, the IDOB may not directly be applied to approximately invert the whole vehicle model since IDOB requires a stable plant but the vehicle dynamics with steering wheel angle δ_S as input and lateral

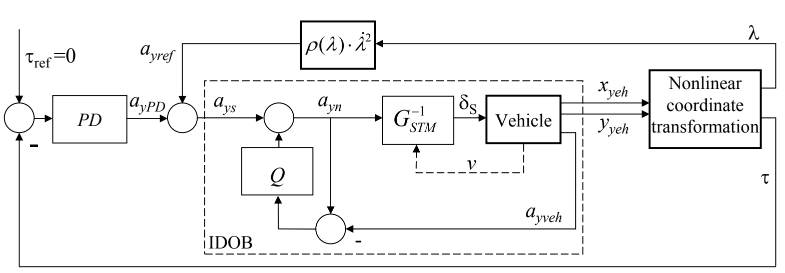


Figure 7: Path tracking control with IDOB.

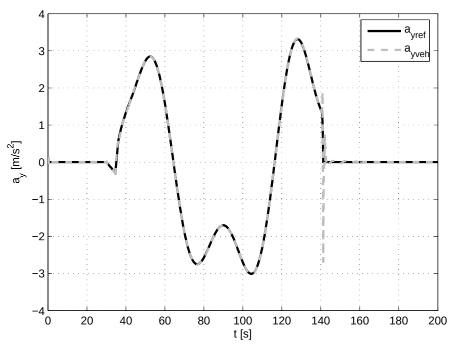


Figure 8: Lateral acceleration control with IDOB.

displacement τ as output involves two integrators, see (7). Based on (5)-(7), for approximate inversion of the whole detailed vehicle model we adopt a hierarchical control structure according to Fig. 7:

$$a_{ys} = a_{yref} - (K_d s + K_p)\tau \qquad (19)$$

$$a_{yveh} = G_{IDOB} \cdot a_{ys} \qquad (20)$$

A subordinate high bandwidth IDOB is used to make $a_{yveh} \to a_{ys}$. The Q-Filter is chosen a first order low-pass filter with a 0.03s time constant. An outer PD control loop with lower bandwidth compensates for the remaining tracking error τ. In the IDOB structure, henceforth only the stable part of the vehicle dynamics with output a_{yveh} is considered. $a_{y,s}$ is the set point for the inner IDOB loop and G_{STM} represents the single track model adopted as nominal model which corresponds to eqns. (8)-(11) . Note that the speed parameter of G_{STM}^{-1} is scheduled with the actual speed of the detailed vehicle model. δ_S is the steering wheel angle signal which is in the focus of interest. The reference lateral acceleration a_{yref} may be considered as a known external disturbance. Therefore disturbance feedforward compensation is applied according to Fig. 7. The resulting transfer function to τ is

$$\frac{\tau}{a_{yref}} = \frac{G_{IDOB} - 1}{s^2 + G_{IDOB}(K_d s + K_p)}. \qquad (21)$$

Assuming that the bandwidth of the IDOB transfer function is sufficiently high ($G_{IDOB} \to 1$), the bandwidth and damping of the outer PD control loop may directly be affected by the PD parameters which are chosen as $K_d = 12$, $K_p = 36$.

Fig. 8 shows a simulation result of the approximately inverted detailed vehicle model performing the double lane change maneuver. The results are presented in terms of the actual vehicle lateral acceleration a_{yveh} which well tracks the reference lateral acceleration a_{yref} by virtue of the proposed IDOB based control. Remarkably, in this simulation the absolute value of the lateral displacement τ is less than 1.5mm (not depicted here). The steering wheel angle obtained is shown in Fig. 4 and can be well compared to the light vehicle single track model.

5 Comparative assessment of the steering dynamics using model inversion

In order to track a given trajectory with a given velocity profile, different vehicles potentially need different steering efforts. Therefore, using the inverse simulation results, steering dynamics of different vehicles may be compared in terms of the required steering efforts.

To illustrate our approach, the light vehicle and the heavy vehicle from section 4.1.1 are compared. The steering angles of these two models necessary to perform the double lane change maneuver with a constant speed of 20 m/s were given in Fig. 4. As it may be seen in this figure, the magnitude of the heavy vehicle steering angle is larger than that of the light vehicle during the maneuver. Moreover, especially in the time interval ca. $[3s, 6s]$ it is recognizable that the heavy vehicle needs to be steered slightly earlier than the light vehicle to follow the reference trajectory. That is, the look-ahead-time the driver needs to drive the heavy vehicle is larger compared to the light vehicle.

In the remainder of this section a method is established to quantify the conclusions mentioned above on the magnitude and look-ahead-time. Wavelets are used for appropriate time-frequency analysis of the steering angle signals.

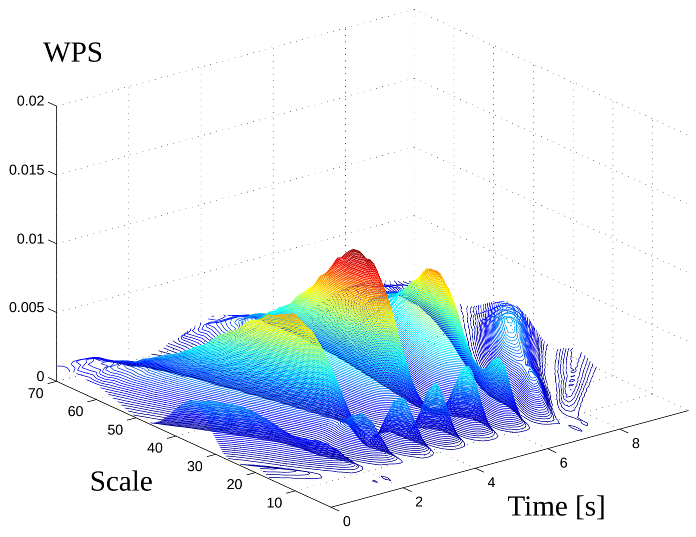


Figure 9: Wavelet power spectrum of the light vehicle steering angle using Morlet wavelet function.

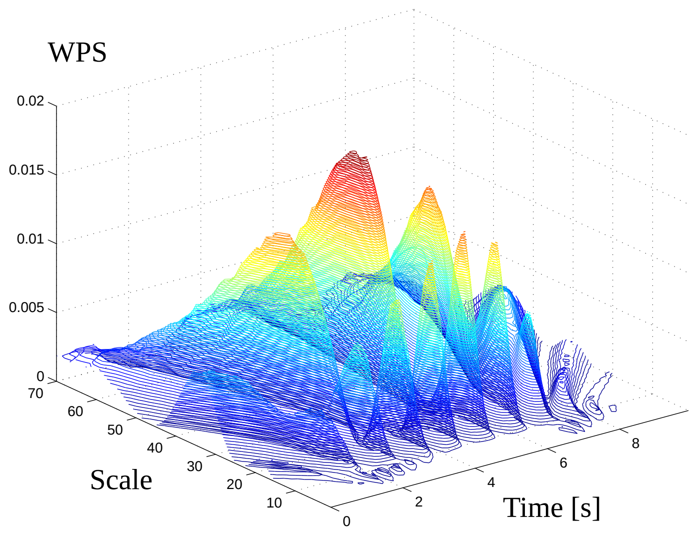


Figure 10: Wavelet power spectrum of the heavy vehicle steering angle with Morlet wavelet function.

5.1 Wavelet transform

Time-frequency analyses map the time domain signals into a two dimensional representation of energy versus time and frequency. Wavelet transform is a time-frequency analysis method that breaks a signal down into its constituent parts, wavelets, for analysis. Wavelets are oscillatory, scalable functions which are non-zero only within a limited spatial and Fourier regime. In the continuous wavelet transform, which is used in this paper, a wavelet is translated (time-shifted) through the signal. At each instant (i.e. time grid point) it is compared with the signal by means of evaluating the time integral of their product. This procedure is repeated for a grid of wavelets with different time scales. As a result, coefficients representing the similarity between sections of the signal and the scaled wavelet are produced. More detailed information on wavelets and wavelet transform may be found in [7], [8]. The wavelet transform returns a time-scale representation of the signal instead of the time-frequency representation. The scale is proportional to the reciprocal of the frequency. Large scales correspond to small frequencies and vice versa.

The single track model steering angle signals from Fig. 4 are now compared in terms of wavelet power spectra. At every instant, the time-scaled wavelet that locally best matches with the steering signal yields the maximum wavelet power spectrum value. Therefore, the local frequency content of the signal can be estimated from the scale value at which a local maximum occurs.

5.1.1 Wavelet transform of the steering angle signals

One of the basic problems in wavelet transform is choosing the appropriate wavelet function for the analysis of a given signal. In the wavelet transform of the steering signals, *Morlet* wavelet function is used, since it is recommended [9] for the analysis of time signals with smooth variations. In Figures 9 and 10 wavelet power spectra (WPS) of the steering angles of the light and heavy vehicles are given, respectively.

The two WPS are quite similar in terms of the scale and time locations of the local maxima, i.e. both signals have similar frequency contents at corresponding

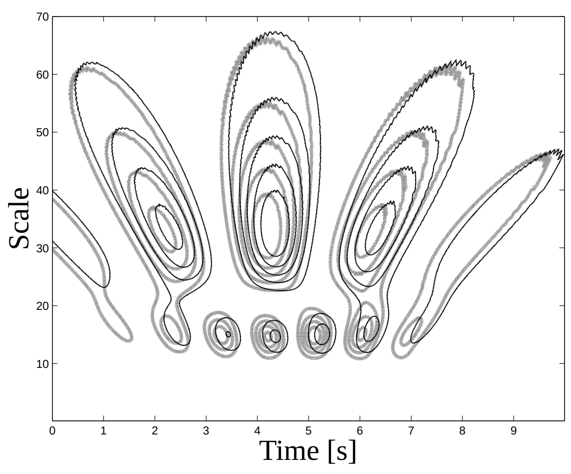


Figure 11: Wavelet power spectrum of the light vehicle steering angle (black lines) and eavy vehicle steering angle (gray lines) with Morlet wavelet function.

instants. However, almost throughout the entire time-scale domain, WPS of the heavy vehicle steering angle has higher power values compared to the light vehicle. This shows that the heavy vehicle needs more steering amplitude compared to the light vehicle all along the maneuver. This result coincides with the previous observation in the time-domain.

In Fig. 11 both WPS's are drawn in the same 2-D plot to make the differences between them better visible. The time of the local maxima can be more easily detected in Fig. 11. The power contour lines of the heavy vehicle steering angle are shifted to earlier instants by about 0.14s-0.18s compared to the light vehicle. This shows that the look-ahead-time needed to steer the heavy vehicle is correspondingly larger than for the light vehicle.

Using continuous wavelet transform with the Morlet wavelet function we are thus able to quantify the conclusions we already made from the time-domain plot of the steering angles. Another aspect in the comparison of the steering efforts is the frequency content of the signals. In Fig. 11 it may be noticed that there are mainly two accumulations of local maximum scale values which are at scale values 14 and 35. The scale values 14 and 35 corresponds 0.58 Hz and 0.23 Hz respectively which are frequencies that prevail in the signals. From Figs. (9, 10 it may be noticed that the steering angle of the heavy vehicle contains relatively higher power values at scale 14.

In other words, the heavy vehicle has to be steered with higher amplitudes, with a relatively larger portion of high frequencies and earlier (i.e. with more look-ahead-time) than the light vehicle. Hence, we conclude that the heavy vehicle is more difficult to drive.

6 Conclusions

Exact inversion of simulation models in principle is supported by Modelica/Dymola. However, it may be the case that models do not comply with the requirements to make inversion feasible. If so, approximate inversion may be an expedient way to still achieve useful results. High fidelity path tracking was demonstrated by means of the inverse disturbance observer based control. This provides a pretty accurate approximation of the steering angle signal which would result in perfect tracking. The time-scale wavelet power spectrum of the steering angle signal is an adequate basis for assessment of the steering effort.

Acknowledgment

The authors are grateful to Professor Wolfgang Wiechert from the University of Siegen Department of Simulation for academic supervision of the work [10] resumed in this paper.

References

[1] N. Bajcinca and T. Bünte, "A novel control structure for dynamic inversion and tracking," *Accepted for IFAC World Congress*, 2005.

[2] P. Riekert and T. Schunck, "Zur Fahrmechanik des gummibereiften Kraftfahrzeugs," *Ingenieur Archiv*, vol. 11, pp. 210–224, 1940.

[3] J. Andreasson, "Vehicle dynamics library," *Proceedings of the 3rd International Modelica Conference*, pp. 11–18, 2003.

[4] S. Heller and T. Bünte, "Modelica vehicle dynamics library: Implementation of driving maneuvers and a controller for active car steering," in *Proc. 3rd International Modelica Conference*, (Linköping, Sweden), 2003.

[5] G. Looye, "Design of robust autopilot control laws with nonlinear dynamics inversion," *Automatisierungtechnik*, pp. 523–531, 2001.

[6] M. Thümmel, M. Otter, and J. Bals, "Control of robots with elastic joints based on automatic generation of inverse dynamics models," *Proc. of 2001 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 925–930, 2001.

[7] R. Polikar, *The wavelet tutorial*. http://engineering.rowan.edu/∼polikar/WAVELETS/WTtutorial.html.

[8] S. Qian and D. Chen, *Joint Time-Frequency Analysis, Methods and Applications*. Upper Saddle River, NJ USA: Prentice-Hall, Inc., 1996.

[9] M. O. Domingues, "On wavelet techniques in atmospheric sciences," *First Latin American Advanced School on Space Environment*, 2004.

[10] A. Sahin, *Inversion of vehicle dynamics with Modelica*. University of Siegen, 2005. M.Sc. thesis.

Session 4b

Thermodynamic Systems III

Analysis of thermal storage systems using Modelica

Wolf-Dieter Steinmann Jochen Buschle
German Aerospace Center
Institute of Technical Thermodynamics
Pfaffenwaldring 38-40, 70569 Stuttgart

Abstract

Modelica is used for the analysis of different kinds of thermal storage system for applications in power plants and process industry. The analysis includes concepts using sensible heat storage media, latent heat systems and steam accumulators. The temperature range for these systems is between 200°C – 400°C, the maximum thermal power is 100MW. For the various storage systems physical models are implemented in Modelica. Modelica is also used for system analysis simulating the interaction of the storage unit with the other components. The results of this system analysis help to improve the efficiency y of storage systems significantly.

Keywords:thermal energy storage; solar power plant; steam accumulator

1 Introduction

Thermal energy storage systems are a promising option for improving the efficiency of power plants and process heat utilization in industry. These systems represent an additional tool for energy management in thermal processes by bridging the gap between demand and availability of energy. At the present time the range of proven storage systems for providing thermal energy at temperatures exceeding 100°C is limited. Various solutions have been proposed, the selection of a concept strongly depends on the characteristics of the process. The aim of current research project is to develop storage systems for commercial applications. For three different basic storage concepts Modelica is used to identify the interaction of the storage unit with the other components of the systems. Using models from the library TechThermo Modelica proves to be an effective tool for the analysis of the dynamics of energy storage systems.

2 Storage systems for solar thermal power plants

2.1 Storage systems using sensible heat

Solarthermal power plants use concentrated solar insolation to drive a thermodynamic power cycle [1]. Today’s commercial systems use trough shaped mirrors to heat a synthetic oil flowing in absorber pipes located in the focus line (Figure 1). At temperature up to 390°C the heat transported by the oil is used to generate steam to drive a turbine. The total electric capacity of these parabolic trough power plants operated in California is 350MW, by continuous optimisation the costs for electricity have been reduced to 0,14US$/kWh, so parabolic trough power plants are the most economic system for large scale generation of electricity from solar energy.

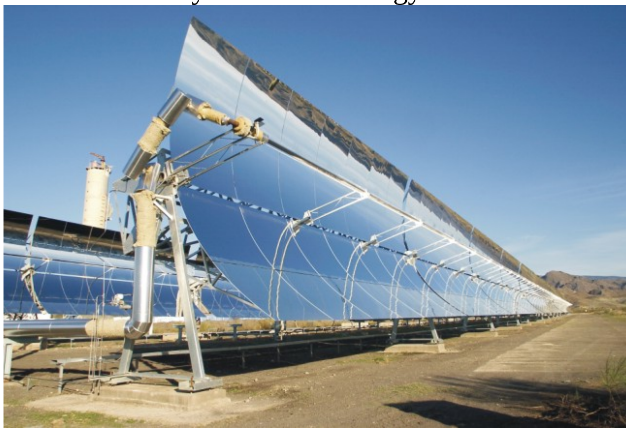


Figure 1: Parabolic trough collectors at solar test center near Almeria, Spain.

In recent years significant research activities have been initiated in Europe to improve the parabolic trough technology to promote a market introduction in areas like the Mediterranean region. Important components for increasing the efficiency of these solarthermal systems are systems for the storage of thermal energy. These storage systems help to reduce the dependence on the course of solar insolation.

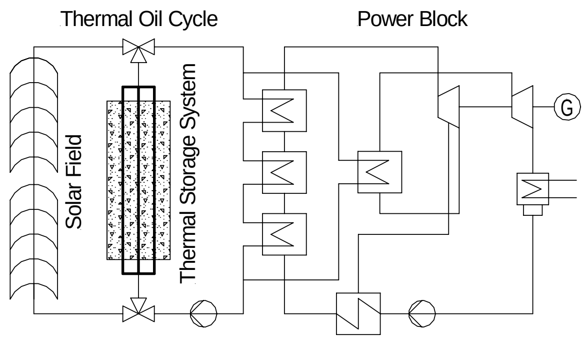


Figure 2: Schematic of parabolic trough power plant with integrated storage unit

One storage concept is based on sensible heat storage in solid media [2]. A tubular heat exchanger is integrated into the storage volume. During the charge process, hot oil from the solar collectors is used to heat up the storage mass; during the discharge cycle cold oil enters the storage unit and is heated up. Modelica is used for the simulation of the storage unit [3]. The results provide the basis for the design of a storage test facility and are applied for the development of operation strategies. This proves to be an ideal application of Modelica since

- the system combines a detailed physical model (spatial discretized storage unit) with less detailed models for the power cycle and the solar collectors
- Modelica allows continuous and discrete event modelling which is necessary for the analysis of the transition from charging to discharging
- the characteristic duration of a charge/discharge cycle is in the range of 24 hours; the computing time needed by Modelica is less than 0.1% of the simulated time interval.
- the analysis requires modifications of the structure of the power plant. The graphical interface of Modelica allows a quick variation of the number and interaction of components.
- although the storage unit represents a non-conventional component, it can be modelled by combination of well known fundamental physical processes; the extent of required additional modelling is small

TechThermo was used for the modelling of the storage unit integrated in the parabolic trough power plant. More than 90% from the lines forming the source code the complete model were taken from TechThermo; the additional modelling mostly defines characteristics of the charge/discharge process. Figure 3 shows the first model level representing the complete power plant with the three main components and a control unit that defines the mass flows in the system. The focus of the analysis lies on the storage model. Due to economic aspects low cost materials like concrete are used as storage mass. Since these candidate materials usually exhibit low values for heat conductivity the temperature of the storage mass is not homogenous.

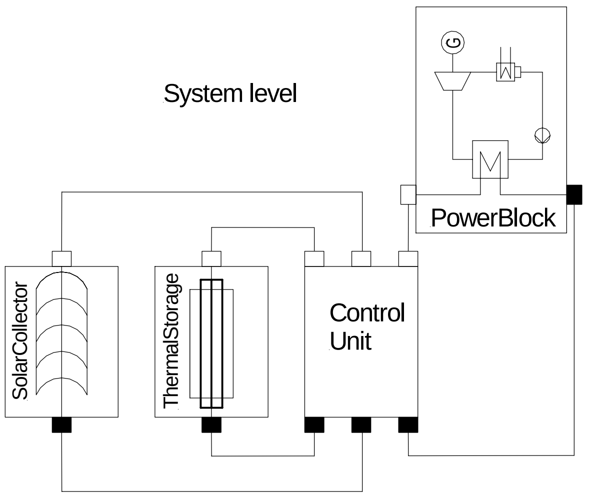


Figure 3: Main components of the Modelica model of parabolic trough power plant with integrated storage unit

Figure 4 shows the model of the storage unit: the system is assumed to be composed of parallel tubes surrounded by storage material, the radial temperature distribution and the flow field inside the tube should be identical for all tubes at the same axial position. Since characteristic lengths of the storage unit are in the range of about 500m the assumption of a radially symmetric temperature distribution around the tubes is necessary to avoid a significant increase in computing time resulting from a three dimensional calculation of the temperature inside the storage mass. The errors resulting from this assumption seem to be acceptable. The storage tube is discretized in axial direction. Modelica offers the declaration of arrays of components which are interconnected, spatial discretization is done by connecting models for a storage segment of length dz in series. The number of elements depends on the length of the storage unit and varies between 50 and 100. The build up of the model of the storage segment is shown. The model is composed of a model for the fluid volume, the tube and the surrounding storage material, heat is transferred between these three models in radial direction. There is also a heat and mass flow in axial direction to the neighbouring segments. The model for the surrounding storage material includes a spatial discretization in radial direction to account for the limited thermal heat conductivity of the storage mass.

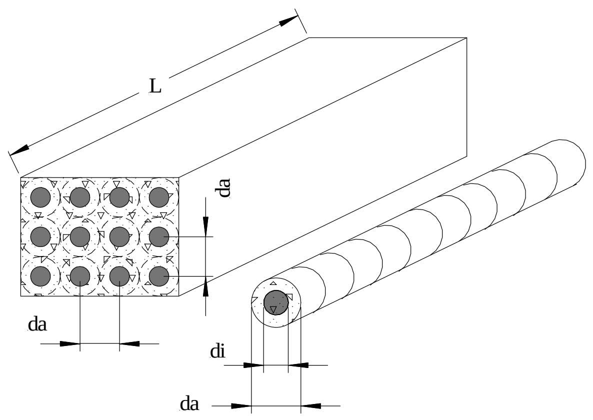


Figure 4: Physical storage model composed of parallel tubes discretized in axial direction

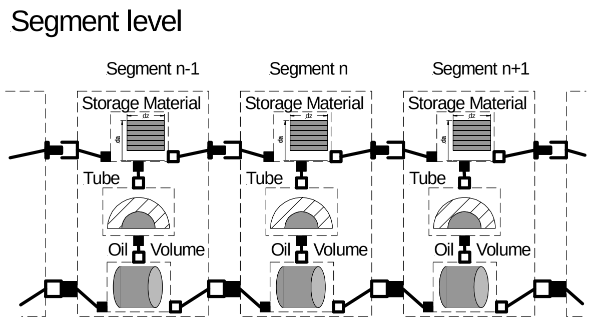


Figure5: Cut-out of the model of a single pipe of the storage system discretized in axial direction by serial connection of models for a pipe segment

Figure.5 shows the internal build up of the model for the fluid volume: the model is composed of a component including the conservation laws for mass and energy, two models calculating friction pressure loss and a model for the convective heat transfer between fluid and tube wall. These components are completed by a property model providing the correlations between the thermal state variables.

The storage model was used to identify the influence of material and geometry parameters and provide the basis for an economic optimisation. Figure 6 is an exemplary diagram for the time course of temperature at various radial positions in the storage material. The simulated time interval includes a charge cycle, a break and a discharge cycle.

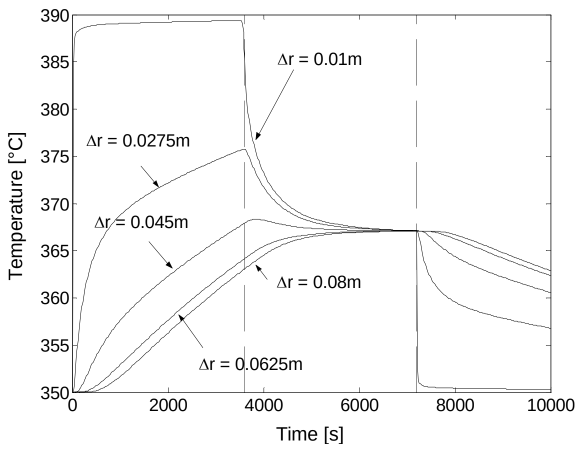


Figure 6: Example for results of storage simulation: temperature of storage material at various radial positions

The energy provided by the storage unit during discharge is used to generate electricity, so the analysis has to consider the transformation of heat into mechanical work by the Rankine cycle. The Rankine cycle demands heat at different temperature levels, the temperature of the oil flowing back to the storage unit is also dependent on the temperature of the oil at the exit of the storage unit. Modelica was used for the simulation of the complete power plant including storage and solar collectors. Figure7 shows the electric energy provided during the discharge process for different configurations of the storage unit. The total storage mass remains constant. The Modelica results show that an optimised operation strategy can lead to an increase of storage capacity of about 200%. This improvement is achieved by an adjustment of the storage unit to the specific requirements of solar collectors and the power cycle [3].

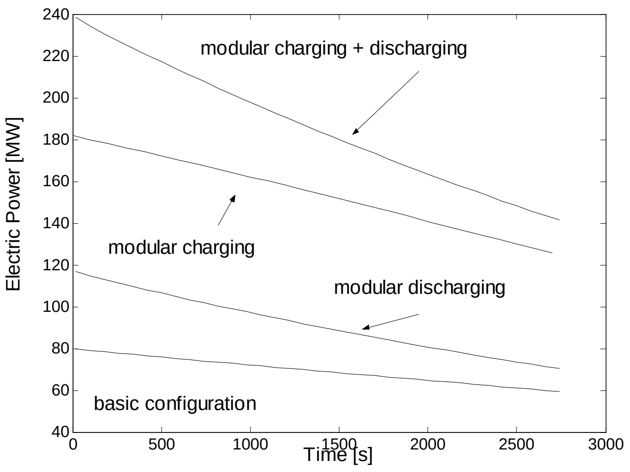


Figure7: Example for system simulation: electric power provided by the power plant during the discharge process.

The identification of this option to reduce the investment costs for the storage unit was mainly possible due to the simulation results.

2.2 Storage systems using latent heat storage media

Solarthermal power plants using thermal oil as heat transfer medium in parabolic trough absorber pipes have been optimized in recent years, a further progress demands the modification of the basic principle. The direct steam generation (DSG) in the absorber-pipes improves the efficiency of the parabolic trough technology by eliminating the synthetic oil and the heat exchanger and increasing the maximum process temperature [4]. The estimated reduction in electricity generation costs is in the range of 25%.

In DSG systems more than 50% of the thermal energy is needed for the evaporation process which takes places at nearly constant temperature. Regarding second law efficiency, a DSG system must be able to store and release thermal energy at nearly constant temperature, sensible storage systems can't be used. Instead, the utilization of latent heat from the melting / solidification process seems a promising concept for constant temperature storage systems.

First concepts for latent heat storage systems a similar to the sensible heat storage systems using concrete: a heat exchanger is embedded in the storage material (phase change material - PCM). Instead of the thermal oil, steam flows in the heat exchanger.

Modelica is also used for the initial analysis of the PCM –storage system. The first model is a modified version of the model for sensible heat storage: the concrete storage material is replaced by the model for the PCM storage material, the thermal oil in the pipe volume is replaced by a steam flow. Due to the reuse of already existing models, the development time for the first model of the PCM storage model could be reduced significantly.

3 Steam accumulator systems

Due to its high volumetric heat capacity and low mass specific costs water represents an ideal storage medium. Unfortunately, it can't be applied under atmospheric conditions at temperatures exceeding 100°C. In order to extend the application range of water based thermal storage system, water was stored in pressurized vessels to increase the saturation temperature. These storage systems are called steam accumulators since usually they are intended for supplying saturated steam [5].

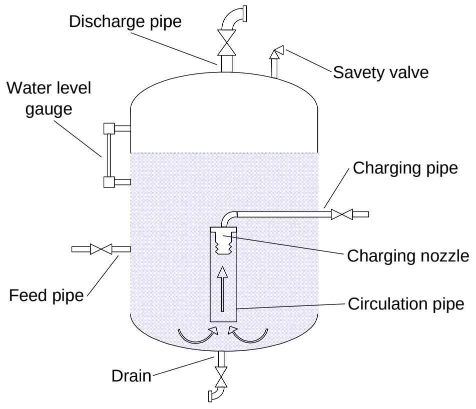


Figure 8: Cross section steam accumulator

Figure8 shows the cross-section of a steam accumulator. Most of the volume is filled by the liquid phase that is covered by the saturated steam phase at the top of the vessel.

Both phases are in thermodynamic equilibrium. If the steam is discharged directly from the accumulator, steam is produced by evaporation from the boiling liquid part. The latent heat of evaporation is cooling down the content of the storage vessel. This leads to a new thermodynamic equilibrium and accordingly to a lower pressure. To charge the accumulator steam is brought into intimate contact with the water content, in order to distribute the heat, released from the condensing steam, uniformly throughout the liquid.

The main parts of a steam storage installation are:

- Storage vessel for holding the storage medium
- Devices for charging and discharging the steam
- Accessories for carrying out the storage operation
- Regulators for the automatic control of the storage installation

Storage Vessels

The production costs of the vessel are the most important item in the total cost. For this reason the design of the storage vessel is central to the layout of the installation as a whole. The best shape provides minimum weight, is simplest to produce and takes up the least floor space. From considerations of strength the storage vessels are best made circular in cross-section, i. e. their basic shape is that of a cylinder. The ends are elliptic or hemispherical in shape. In practice an average length-to-diameter ratio of 4 has been found to be the best.

3.1 Charging and Discharging Devices

Indirect steam accumulators use a liquid as the storage medium, so that the steam must be condensed to be stored. This can be achieved by blowing it into the liquid contained in the accumulator. The incoming steam bubbles condense in the liquid or pass into the steam space, depending on the thermodynamic equilibrium in the vessel. The bubbles which rise to the steam space increase the pressure and lead to a higher saturation temperature, so that the next bubble might condense. To use the entire storage content, the charging process requires circulation. Ruth invented a method that consists of nozzles which turn the flow of steam upwards. The nozzles are surrounded by a circulation pipe, wherein the water flows upwards. The minimum temperature loss is composed of the difference between the steam space and the uppermost liquid layer and the difference between the saturation temperatures due to the additional pressure of the water at lower depths. Depending on the accumulator pressure and the steam intake there is a certain depth for the nozzles which minimizes the overall temperature loss. To avoid introducing charging steam into the storage vessel itself an external condenser and evaporator can be used.

3.2 Accessories

All storage installations require efficient thermal insulation to reduce cooling losses to an economic level. The fittings on the pressure vessel itself are the safety valve, the anti-vacuum valve and the blowdown valve. The thermal expansion of the vessel can be considerable in all directions and simultaneous adjustment must be provided in the piping by smooth or curved pipe bends or by bellow-type compensators. Measuring instruments for indicating the charging state in the accumulator are of special importance.

3.3 Regulators

To maintain a certain state in the accumulator or in the piping system regulation by valves is required. The regulator can be acting as a reducing valve, opening with falling pressure in the downstream controlled piping. It can also be acting as an overflow valve with increasing pressure in upstream controlled piping.

Figure 9 shows the Modelica model for the varying-pressure accumulator. The central part of the model is the vessel. In the vessel the mass and energy balance for an open control volume is solved. The volumetric and caloric properties are calculated within the equation of state model that is connected to the vessel model via a thermal state connector. All connectors are defined in the TechThermo library. To represent the mass of the vessel shell a heat capacity is connected due a thermal resistance to the vessel. The pressure loss of the mass flow during charging the accumulator is represented by two models. The first is used to calculate the static pressure increase below the water line in the vessel. The second model computes the pressure loss of a flow due friction with a coefficient called Zeta.

3.4 Simulation Model and Results

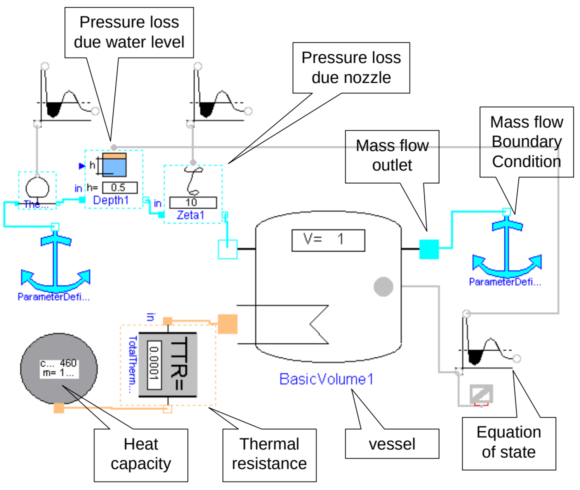


Figure 9: Modelica model of steam accumulator

3.5 Mass and energy balance in the vessel

The energy equation for a control volume that relates energy and mass flows has following form

$$\frac{dE}{dt} = \dot{m}_{in} \cdot \left(\frac{w_{in}^2}{2} + g \cdot z_{in} + u_{in} + \frac{p_{in}}{\rho_{in}} \right)$$

$$- \dot{m}_{out} \cdot \left(\frac{w_{out}^2}{2} + g \cdot z_{out} + u_{out} + \frac{p_{out}}{\rho_{out}} \right) + \dot{Q} - \dot{W}_e$$

If we assume that the changes in kinetic and potential energy are zero and there is no external work we obtain an equation for the internal energy of the control volume

$$\frac{dU}{dt} = h_{in} \cdot \dot{m}_{in} - m_{out} \cdot \dot{m}_{out} + \dot{Q}$$

Conservation of mass means that the change of mass in the control volume must equal the difference between the mass entering the system and the mass leaving the system

$$\frac{dm}{dt} = \dot{m}_{in} - \dot{m}_{out}$$

The specific internal energy in the control volume consists of the internal energy of the liquid part and the internal energy of the vapour part

$$u = (1-x) \cdot u_{liq} + x \cdot u_{vap}$$

with the steam quality x in the following form

$$x = \frac{m_{vap}}{m_{liq} + m_{vap}}$$

3.6 Thermophysical properties of steam

The volumetric and caloric properties are calculated with the Soave-Redlich-Kwong cubic equation of state, the departure function for the cubic equation of state, the Antoine saturation pressure correlation and the enthalpy of the ideal gas. The results are within an error of 5 %.

A closed system that consists of a liquid and its vapour in thermodynamic equilibrium has 1 degree of freedom. So if e.g. the temperature is known the pressure, the enthalpy, etc. can be calculated in the following way.

For a known temperature the pressure is given by the Antoine pressure correlation

$$\ln(p) = A_A - \frac{B_A}{T + C_A}$$

With the known temperature and the calculated pressure the specific volume of the liquid part and the vapour part is received from the Soave-Redlich-Kwong cubic equation of state

$$p = \frac{RT}{v-b} - \frac{a(T)}{v(v+b)}$$

by rearrangement to the normal form

$$v^3 - \left[\frac{RT}{p}\right] \cdot v^2 + \left[\frac{a(T) - RTb}{p} - b^2\right] \cdot v - \left[\frac{a(T)b}{p}\right] = 0$$

and using Cardano’s method. R and b are material dependent constants. a(T) is a temperature dependent variable.

The enthalpy is estimated with the departure function for the Soave-Redlich-Kwong cubic equation of state

$$h - h^\circ = \left[T \cdot \frac{\partial a(T)}{\partial T} - a(T)\right] \cdot \frac{1}{b} \cdot \ln\left(\frac{v+b}{v}\right) - \frac{a(T)}{v+b} + R \cdot T \cdot \left(\frac{v}{v-b} - 1\right)$$

and a polynomial equation for the enthalpy of the ideal gas

$$h - h_{T_0} = A_{IG}(T - T_0) + \frac{B_{IG}}{2}(T^2 - T_0{}^2) + \frac{C_{IG}}{3}(T_3 - T_0{}^3) + \frac{D_{IG}}{4}(T_4 - T_0{}^4)$$

For the enthalpy of the liquid part the results are not satisfying. A better approach is to calculate the latent heat of evaporation and subtract it from the enthalpy of the vapour. The latent heat of evaporation is received from the Antoine vapour pressure correlation in combination with the Clausius-Clapeyron equation

$$\Delta h_{lv} = \frac{B_A \cdot \Delta v_{lv}}{T} \cdot \left(\frac{T}{T + C_A}\right)^2 \cdot e^{A_A - \frac{B_A}{T + C_A}}$$

The differentiation between the state of superheated steam and the state of wet steam is realised by an if-clause.

```
if (x > 1 and p < p_sat) then
   x = 1;
else
   p = p_sat;
end if;
```

If the steam is superheated the steam quality x is fixed to 1. Else the system pressure is calculated with the saturation pressure correlation.

3.7 First Results

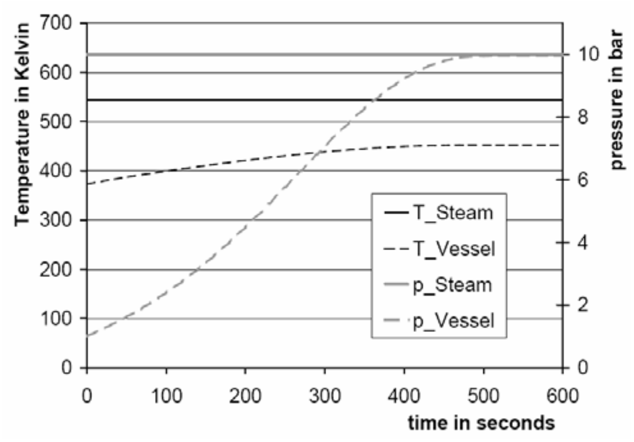


Figure 10: Temperature and pressure rise in steam accumulator during charging process

In Figure 10 first simulation results of the varying-pressure accumulator model are shown. The vessel is charged with superheated steam at a temperature of about 550 Kelvin and a pressure of 10 bar. The initial temperature of the vessel is 373 Kelvin. If the vessel is in thermodynamic equilibrium the temperature of the vessel will not exceed the saturation temperature belonging to the pressure in the vessel. As it can be seen, limiting for the charging procedure is the pressure of the superheated steam. A bigger amount of energy could be stored if an indirect charging device is used.

4 Conclusions

In particular the results of the system analysis of storage units prove to be a very useful tool for the optimization. For a selected application, the thermal energy provided by the storage system must be evaluated regarding the requirements of the specific process. Often, the duration of a charging / discharging cycle often exceeds durations of 24h. The capability of Modelica to simulate efficiently the transient behaviour of systems over such periods offers an important option for optimization.

Further development will also include steam accumulator with integrated phase change material. This concept is intended to increase the storage capacity of steam accumulators. Here, most of the needed models are already available from the current simulation projects.

References

[1] Winter, C.-J ,C.-J., Sizmann, R. L., Vant-Hull, L. L.: Solar Power Plants, Springer-Verlag, Berlin, 1991

[2] Dinter, F., Geyer, M., Tamme, R.:Thermal Energy Storage for Commercial Applications, Springer-Verlag, Berlin 1990

[3] Tamme, R., Laing, D., Steinmann, W.-D.:Advanced Thermal Energy Storage Technology for Parabolic Trough, ASME J. of Solar Energy Engineering 126 (2004), pp.794-800.

[4] Eck M., Zarza E., Eickhoff M., Rheinländer J., Valenzuela L.: Applied Research concerning the Direct Steam Generation in Parabolic Troughs, Solar Energy, Vol. 74 (2003) pp. 341-351

[5] Beckmann, G., Gilli, P.V. “Thermal Energy Storage“, Springer Verlag, 1984

Exergy-analysis of a direct-evaporating cooling plant with heat reclaim

Stefan Wischhusen[1]   Gerhard Schmitz
Hamburg University of Technology
Department of Thermodynamics
Denickestr. 17, D-21073 Hamburg
[1]wischhusen@tuhh.de

Abstract

In this paper the modelling of a direct-evaporating two-stage cooling plant with the refrigerant ammonia (R717) will be described. The plant model is used to determine the power consumption as well as the possible heat reclaim to the domestic hot water system of the plant. In a sensitivity study important control parameters of the plant are evaluated for electricity, water and natural gas consumption. One characteristic operating point is investigated in an exergy-analysis [1, 2] to find potential for energy savings.

Keywords: Refrigeration, Exergy-analysis, Heat recovery, Ammonia, R717, Two-stage system

1 Introduction

Probably the largest application for industrial refrigeration is cooling and freezing of food. Large plants are needed to provide refrigeration throughout all seasons of a year covering all production steps during the processing, storage and transportation. Most of the plants are built in a direct-evaporating architecture where the refrigerant is evaporated in each cold storage or consumer. In contrast to that, indirect evaporation with a secondary cooling agent is used for air conditioning systems in large buildings. The reason for this is a lower pressure loss for liquid media in extensive pipework.

Historically, ammonia (R717) is one of the best known refrigerants in industrial applications and it has suitable properties like a high evaporation heat at moderate densities and a range of feasible saturation pressures at common working temperatures (especially with regard to low temperature applications). An economical advantage is its low price in comparison to other refrigerants. Drawbacks are the flammability and toxicity. Experiences go back to the 19th century when David Boyle (1873) and later Carl von Linde (1876) developed the first compression chillers using ammonia. The chiller created by Linde was used in breweries for cooling beer.

Since industrial refrigeration plants are operated many hours per year the energy consumption is relatively high and therefore capital investment for increasing the efficiency returns faster than in plain air conditioning plants which are just seasonally used. A dynamic simulation is carried out because of the high refrigerant and water capacity of the plant.

2 Refrigeration Plant Topology and Functioning

In order to provide cooling capacity at two temperature levels (-10 °C and -35 °C) the compression of the working fluid is separated into two stages: The high pressure (13.5 bar/2.91 bar) and the low pressure (2.91 bar/0.9 bar) cycle, displayed in Fig. 1. The low pressure compression provided by two screw compressors ("Booster", $1 \rightarrow 2$) can be operated independently while one high pressure compressor is always needed to reject the waste heat over the condensers. Therefore, a higher cooling capacity is always necessary on the high pressure side with three screw compressors installed ($3 \rightarrow 4$). The waste heat is mainly rejected to the ambience by evaporative condensers, which incorporate air and water for evaporative cooling ($5 \rightarrow 6,7$).

Since there is a high demand for domestic hot water (DHW) in the plant during production times (mass

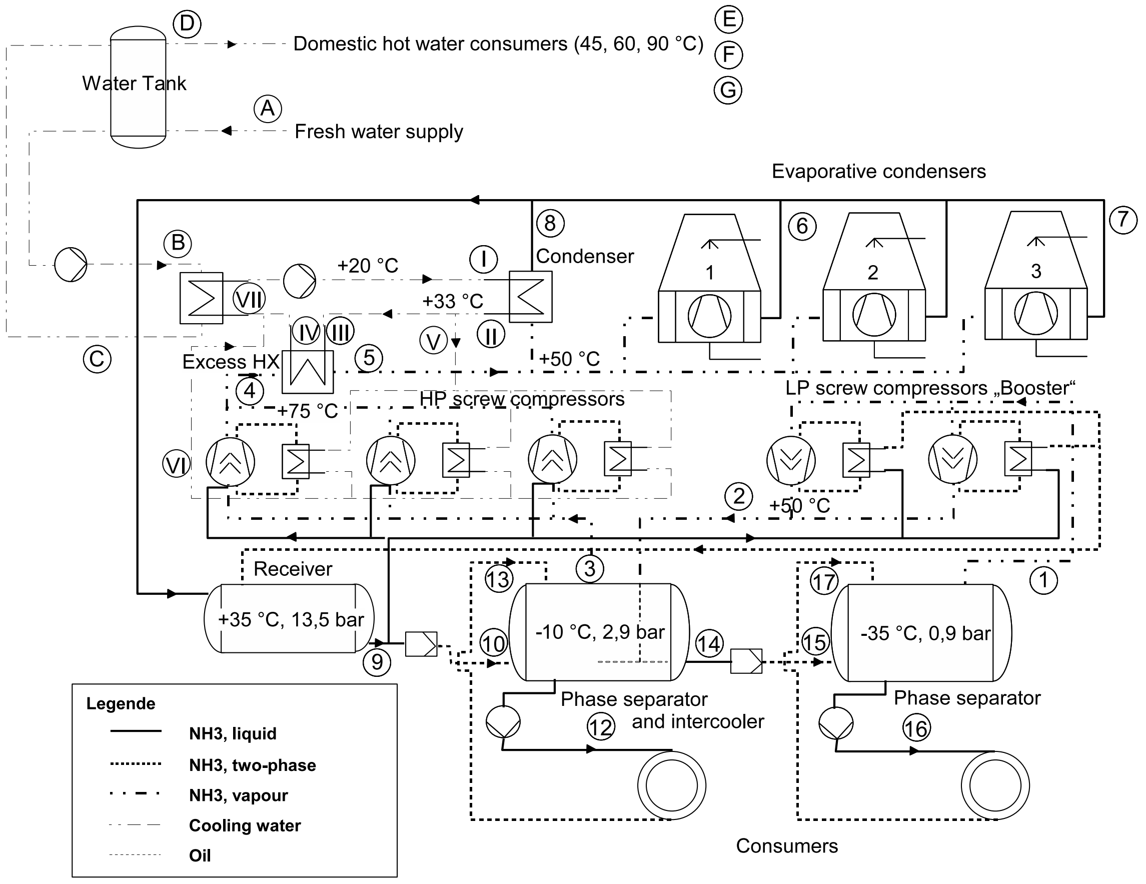


Figure 1: Simplified plant schematic including the refrigeration and domestic hot water system

flow rates are as high as 8 kg/s) it is convenient to recover waste heat by cooling compressor oil and high pressure gas in a water-cooled excess heat exchanger ($4 \rightarrow 5$) and a water-cooled condenser ($5 \rightarrow 8$). Those heat exchangers (HX) are of shell and tube type. Following German domestic water ordinance the heat exchangers have to be cooled indirectly to avoid a contamination with ammonia in case of leakage. Subsequently, the condensate flow is fed into the high pressure receiver (V=2 m^3, $8 \rightarrow 9$) where it can be tapped for expansion or cooling the low stage and high stage screws (9). The latter compressors ($3 \rightarrow 4$) just draw liquid ammonia when the cooling water temperature is too high to ensure an oil temperature of 48 °C. Like oil the liquid ammonia may be injected into the suction side of the compressor to decrease the outlet temperature of the compressed gas. To remove the oil fraction from the superheated refrigerant vapour, an oil separator for each compressor is necessary which is also used as a tank storage.

Unlike in one-stage refrigeration systems, the expanded refrigerant is first stored in a phase separator (V=11 m^3, each) to remove flash gas. This component is essential in order to supply pure liquid medium to the pumps and evaporators of each stage. The separating vessel on the intermediate pressure level (10,13 $\rightarrow$ 3,12,14) is equipped with an intercooler because of the superheated low stage gas which needs to be cooled down to saturation conditions before it can be compressed again by the screws of the second stage.

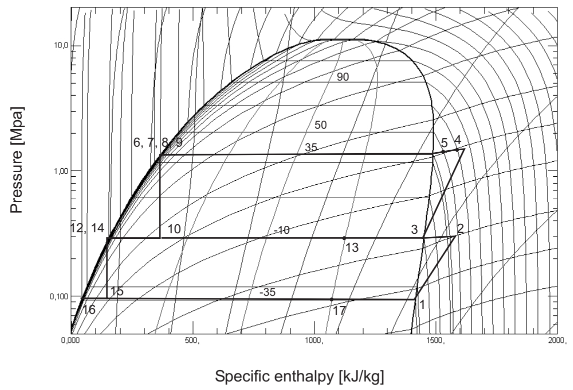


Figure 2: p,h-diagram for R717 of the two-stage refrigeration system – Arabic numerals with regard to Fig. 1

In applications with temperatures below 4 °C a defrost system has to be applied to each cooling coil which is in contact with (humid) air. For this purpose superheated refrigerant tapped before the excess HX is used occasionally by switching a valve at each evaporator.

The total cooling capacity of the screw compressors is 847 kW on the low stage (-35 °C) and 2308 kW on the high pressure side whereof 1461 kW are available for -10 °C consumers. The rated power consumption of the compressor motors sums up to 200 kW for the Boosters and 693 kW on the high stage. Accordingly, the evaporative and water-cooled condensers have a cooling capacity of 3910 kW (incl. desuperheat HX) at a saturation temperature of +35 °C.

The capacity control of the system is realised by a variable valve in the screw compressors which can throttle the effective mass flow rate at constant speed. The lowest continuous working point is limited to 10 % of the rated capacity. Below that operation point the motor is driven in an on-off procedure. Internally, the oil flow is adjusted so that a constant oil inlet temperature can be provided. The mass flow rate of oil is almost of the same magnitude as the refrigerant flow to ensure a sufficient lubrification, sealing and cooling. All compressors are organised in a load dependent cascade, operating as many machines as needed.

In an analogous manner the three evaporative condensers are enabled in an pressure dependent cascade at operating points ranging from 9 to 12 bar. At low pressures, the spray water pumps are activated followed by the ventilation of the cooling towers. The mass flow rate through the parallel condensers and cooling towers is adjusted naturally since a lower heat transfer rate leads to higher pressure losses due to the rising resistance in one branch.

The refrigeration process is also shown in a logarithmic p,h-diagram for NH_3 in Fig. 2.

3 Boundary Conditions and Measurements

Since the cooling demand is changing dynamically the plant is not driven continuously but in a typical load profile (see Fig. 4) which is dominated on the low pressure side (-35 °C) by shock-cooling of food entities (2250 hours/a) and on the medium pressure side by cooling storage rooms at -10 °C (8760 hours/a) (see Fig. 3). The load profile of the low temperature consumers varies between 30 and 1100 kW_{th} and for the normal cooling between 100 and 2300 kW_{th}. Thanks to the data measurement of the plant's operator the hourly power and water consumption (see Fig. 4) as well as the product flow of the plant is known and it is considered as boundary conditions for the system simulation. Since the unknown cooling requirement is an important input variable of the load dependent simulation it has to be calculated from known and assumed variables like the power consumption and the product flow. On the low temperature side (LT, 16 → 17) the refrigeration load can be estimated by the following equation:

$$\dot{Q}_{0,\,LT} = \dot{n}_{prod} \cdot 11\ \mathrm{kJ} + \dot{Q}_{0,\,aux} \qquad (1)$$

where $\dot{n}_{prod}$ denotes the flow rate of product. Each product entity has a heat capacity of 11 kJ in the corresponding temperature range and $\dot{Q}_{0,\,aux}$ stands for the smaller amount of additional refrigeration which averages 30 kW. The refrigeration load for room cooling $\dot{Q}_{0,\,MT}$ (12 → 13) at an evaporation temperature of ϑ_0 =-10 °C results from the following equation

$$\dot{Q}_{0,\,MT} = P_{el} \cdot \overline{COP} - \dot{Q}_{0,\,LT} \qquad (2)$$

applying an average coefficient of performance $\overline{COP}$ =3. Despite the fact that the peak load of both stages sums up to 3,400 kW (see Fig. 3) their occurrence is separated. The highest total load is not larger than 2,450 kW.

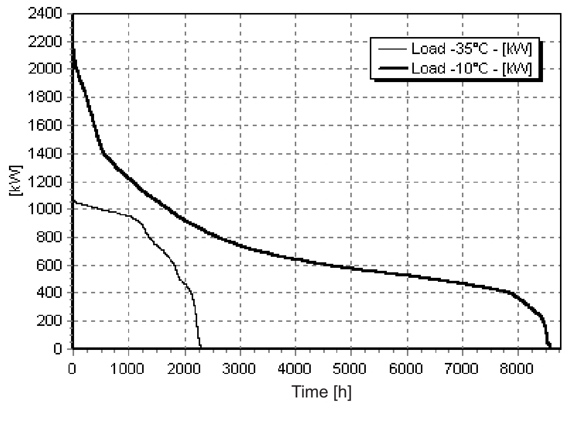
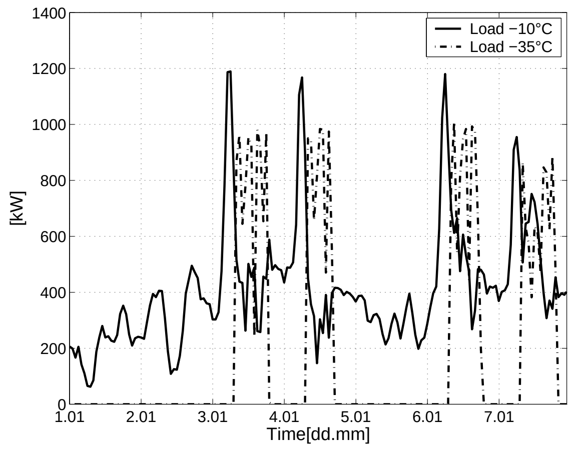


Figure 3: Annual load duration curve for refrigeration

Not available are ambient conditions for that time, so that weather data from a test reference year of the corresponding region in Germany has been used to



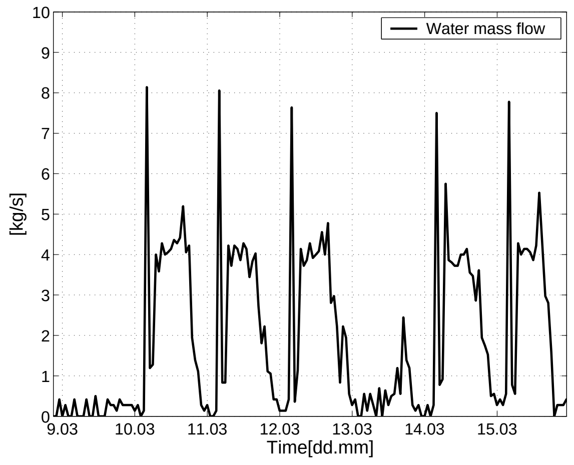


Figure 4: Typical cooling requirement (t.) and DHW consumption profile (b.)

calculate the performance of the cooling tower model (temperature and relative humidity as inputs). The temperature of the fresh water was assumed to change in a sinusoidal way between 10 °C at the beginning of spring and 13 °C in late summer.

The mass flow rate of consumed domestic hot water (E,F,G) is also dynamically changing. The highest flow rates occur at the beginning of each production day (see Fig. 4). Those days are contemporaneously characterised by a high cooling demand on the low temperature side because of the necessary product cooling. This fact combined with a considerable water demand during production times results in a very worthwhile potential for heat recovery. On the other hand the mass flow rate of water in the meantime is not high enough to provide a sufficient condensation and cooling capacity.

Important for an economical analysis of an existing plant are the energy and media prices which are listed below:

- Electricity cost: 70 €/MWh (Compressors, cooling towers, pumps),
- Gas price: 35 €/MWh (DHW supply),
- Fresh water cost: 0.89 €/m^3 (Cooling towers),
- Charge for waste water: 2.29 €/m^3 (Cooling towers).

4 Modelling of Plant and Components

First of all it should be pointed out, that the modelling in this case was focused on the simulation of the refrigeration plant with the integration of the heat recovery. The models for hydronic systems have been supplied by the model libraries of HKSim [3, 4, 5]. Pfafferott has shown that a dynamic simulation of mobile refrigeration systems is possible [6]. He used Modelica for modelling of thermohydraulic elements integrating dynamic energy and mass balances and a quasistatic impulse balance. Unfortunately, such detailed component models are not suitable for complex systems, especially when long simulation periods are investigated. The typical period in the current project is one week and more in order to detect improvements and present them in a financial suitable resolution.

When applying the Finite Volume method in fluid modelling it is important to have a medium property model for all technical relevant states. This is given by a fundamental equation of state which was elaborated by Baehr and Tillner-Roth for a few important refrigerants [7] also including R717. The two-phase region has to be modelled by polynomial functions which depend on one thermodynamic state variable (T or p). It is known that the simulation of the gaseous and two-phase region can be rather efficiently performed, when the density ρ (or the specific volume v, resp.) and the temperature T are used as states and inputs to the highly non-linear equations. The dimensionless Helmholtz-function is defined as:

$$\Phi := f(T,v)\cdot\frac{1}{RT}. \qquad (3)$$

Provisions have to be made with regard to the calculation of liquid state properties. In this region the simulation may become stiff since small changes (or even integration failures) in temperature at nearly

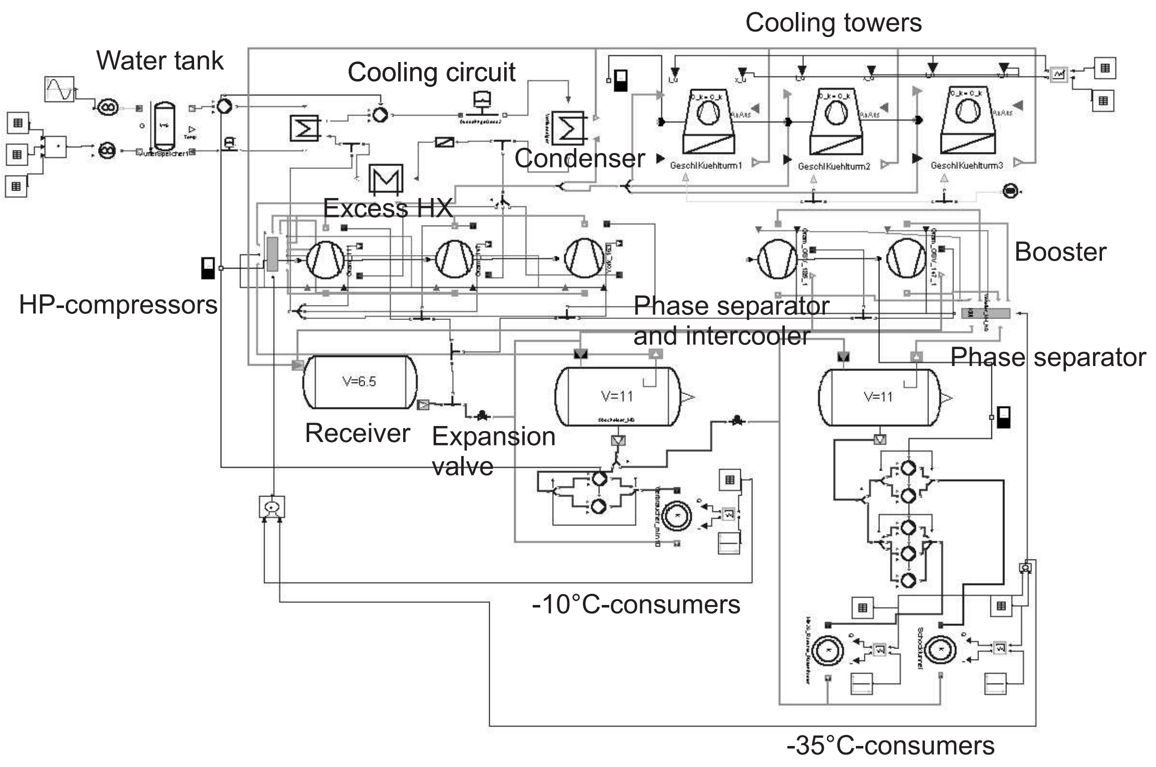


Figure 5: Plant model integrating the two stage refrigeration cycles, cooling water circuit and fresh water tank

constant density result in large pressure gradients. Those gradients lead to small system time constants due to the linkage of momentum and mass and energy balance. This is one reason why a state variable selection of pressure p and specific enthalpy h is generally preferred. However, with those states an iterative calculation of T and v is necessary during the simulation because the complex property functions can not be transformed symbolically, yet. In order to optimise the simulation also with respect to simulation speed it was decided to use a component related formulation of the balance equations in this project especially regarding components containing liquid refrigerant (e. g., refrigerant pumps and other hydraulic elements). One important and simplifying assumption is, that industrial plants are operated more or less continuously even though with variable utilisation factor. Therefore, heat and mass dissipation is not taken into account. From this fact follows that the feed ducts of the evaporators are always passed through with liquid medium. For those elements the incompressible formulation of the mass and energy balance [4] may be used with a constant specific heat capacity c =4,500 J/(kg·K) and a constant density of ρ =650 kg/m^3.

Very important for achieving a fast and stable simulation is also a component related momentum balance which should be as simple as possible. A momentum balance is always needed when a mass shift inside the system due to pressure gradients has to be calculated. In other words: It can be expressed ideally and more efficient if the mass transfer is guarded by a superior control system. For example, the mass flow through the expansion valve of each stage is set in order to realise a constant liquid fill level in the following phase separator.

The mass flow rates through the parallel passes of the condenser and cooling towers just depend on the pressure loss across each branch (in steady state always the same value) which is defined by hydraulic pressure drop correlations. For the quasistatic momentum balance follows:

$$0 = p_{in} - p_{out} - \Delta p_{loss} \tag{4}$$

$$\Delta p_{loss} = \Delta p_{100} \cdot \left(\frac{\dot{m}}{\dot{m}_{100}} \right)^2 \cdot \frac{\rho_{100}}{\rho} . \tag{5}$$

All parameters indicated by 100 in Eq. 5 refer to one characteristic operation point. The density factor can not be neglected when the liquid fraction is variable or a dry out of the heat exchanger is possible (here: condenser). This is often the case in the actual plant when the cooling water temperature rises due to a low domestic hot water consumption.

Moreover, component models which show a phase change like condensers and evaporators should not be separated into multiple volumes to avoid too many events during simulation. A promising approach could be a Moving-Boundary-Model [8] although it was not implemented in this work because of the frequent dry out of the condenser and the load dependent evaporator model ($\dot{Q}_0$ is an input variable).

The heat transfer rate from the refrigerant to the liquid water in the water-cooled condenser is calculated by a quasistatic efficiency calculation (P_1-NTU) from [9].

$$\dot{Q}_{liq} = P_1 \cdot C_{min} \cdot \left(T_{in}^{liq} - T_{out}^{liq}\right) \quad , \quad C = c \cdot \dot{m} \qquad (6)$$

$$P_1 = \begin{cases} \dfrac{2}{1+\frac{C_{min}}{C_{max}}+\sqrt{1+\frac{C_{min}}{C_{max}}^2}\cdot\coth\left(\sqrt{1+\frac{C_{min}}{C_{max}}^2}\cdot\frac{NTU}{2}\right)} \quad , \\ \quad \mathrm{C}_{max} > 0 \;\;\&\;\; \frac{C_{min}}{C_{max}} < 1 \;\;\&\;\; \mathrm{C}_{min} > 0\,. \\ \dfrac{1}{1+\coth\left(\frac{NTU}{\sqrt{2}}\right)} \quad , \\ \quad \mathrm{C}_{max} > 0 \;\;\&\;\; \frac{C_{min}}{C_{max}} >= 1 \;\;\&\;\; \mathrm{C}_{min} > 0\,. \end{cases} \qquad (7)$$

Since the specific heat capacity at constant pressure c_p is equal to infinity in the two-phase region, a crossing function has to be implemented realising a "chatter-free" solution when liquid or vapour content is high. Good experiences were made with a tanh-function changing its value and derivation steadily at vapour qualities x =0… 0.05 and x =0.95… 1. The value of the function is multiplied with the property value for the specific heat capacity of the property model.

The compressors are modelled in a Super-Model approach integrating the base compressor model, an oil separator, the water-cooled oil heat exchanger (fixed properties for liquid oil) and the auxiliary liquid ammonia injection (see Fig. 6). Instead, the booster model incorporates an oil cooling heat exchanger permanently fed with ammonia.

A determining factor for the power consumption of the plant is the efficiency of the compressor. The so called coefficient of performance (COP) mainly depends on

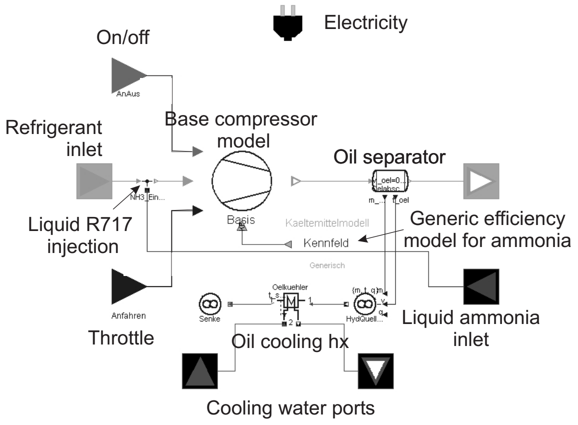


Figure 6: Diagram layer of a high pressure screw compressor model with integrated oil separator, oil cooler and ammonia injection

the part load control (part load factor φ) and the thermodynamic properties of the refrigerant as well as the thermodynamic states in the suction and discharge chamber. The latter mainly result from the actual heat transfer of all components in the cycle. With regard to this plant the suction (index *suc*) and discharge (index *dis*) pressure is defined by the capacitive component models (e. g., the excess heat exchanger and the phase separator in the high pressure cycle, see Fig. 5). In order to calculate the power consumption the mentioned thermodynamic variables are considered in the calculation of the total efficiency of the compressor.

$$\begin{aligned} COP \;\; = \;\; & f_{pl}\left(\Delta p, \varphi\left(p_{suc}, p_{dis}, \frac{\dot{m}}{\dot{m}_{max}}\right)\right) \\ & \cdot f_{th}\left(p_{suc}, p_{dis}, \dot{Q}_0^{nom}, P_{el}^{nom}, \eta_{mech}\right) \end{aligned} \qquad (8)$$

The part load function f_{pl} may be derived from manufacturer data or from literature [10]. For the calculation of the rated performance (index *rat*) at variable suction and discharge pressures (f_{th}) a determination of the refrigerant's properties (specific enthalpies h, entropies s and densities ρ) is carried out. In contrast to the *rated* performance at **full** mass flow rate and **variable** pressures the *nominal* performance denotes **one** rated operating point at **constant** pressures.

$$f_{th} = \frac{h_{0,\,in} - h_{0,\,out}}{h_{dis}^{is} - h_{suc}} \cdot \eta_{is} \cdot \eta_{mech} \qquad (9)$$

$$\eta_{is} = \frac{\dot{m}^{rat} \cdot \left(h_{dis}^{is} - h_{suc}\right)^{rat}}{P_{el}^{nom}\eta_{mech}} \qquad (10)$$

$$\dot{m}^{rat} = \frac{\rho_{suc}}{\rho_{suc}^{rat}} \cdot \dot{m}^{nom} \qquad (11)$$

$$\dot{m}^{nom} = \frac{\dot{Q}_0^{nom}}{h_{0,\,in} - h_{0,\,out}} \quad (12)$$

$$h_{dis}^{is} = h\left(s\left(h_{suc}, p_{suc}\right), p_{dis}\right) \quad (13)$$

It is assumed that the isentropic efficiency of the compression at nominal mass flow rate $\dot{m}^{nom}$ is nearly constant for all operating points. In addition, the available enthalpy of evaporation is assumed to be ideally used.

$$h_{0,\,in} = h_{liq}(p_c) \quad (14)$$

$$h_{0,\,out} = h_{vap}(p_0) \quad (15)$$

At very low cooling requirements (10 % of $\dot{Q}_0^{rat}$) the control system of the compressors stops the continous operation and activates a two-point control with a minimum mass flow rate.

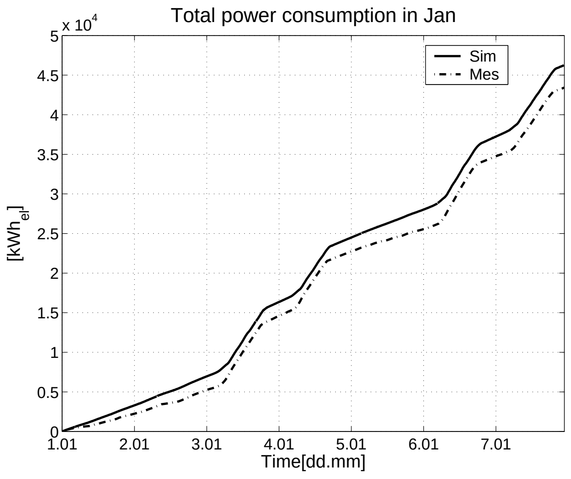
For achieving an efficient simulation only the largest capacities in the cycle were modelled by control volumes. Those components are the phase separators (each 11 m^3) and the high pressure receiver (2.3 m^3). Additionally, the high pressure heat exchangers were also modelled by using dynamic mass and energy balances in order to stabilise the solution of the non-linear system of equations during simulation. The modelling of the intercooler functionality of the phase separator on the intermediate pressure level is realised by mixing of all inbound enthalpy flows and computing saturated enthalpies for all outgoing mass flows.

A very demanding component from the modelling point of view is the evaporative condenser which has three fluid fluxes moving in different directions (Refrigerant: horizontal, air: bottom-top, water: top-bottom). A detailed model is described by [11, 12]. More applied to the needs of complex energy system simulations seems to be the approach of Stabat and Marchio [13] which offers a promising approach and some successful validation.

The model of this study is even more simplified by using the assumption that the air outlet condition equals always the mean temperature between the entering refrigerant and the wet bulb temperature while the relative humidity is constant. The cooling capacity can be adjusted by a variable mass flow rate of air. The supplied characteristic curve for the ventilation yields the power consumption of the motor.

4.1 Validation of the Plant Model

For the validation of the plant model measurement data has been supplied by the plant operator. The data displays the power, domestic water consumption and waste water flow in an hourly interval. Moreover, some offline-information was collected on a visit of the plant while the production was on (high cooling requirement for -35 °C-consumers).



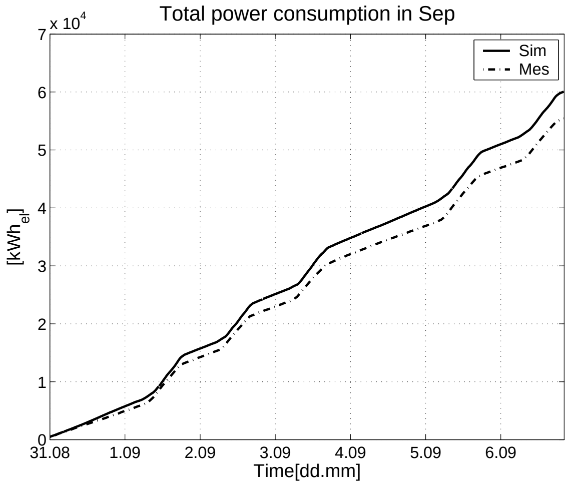


Figure 7: Comparison of the total power consumption in January (t.) and September (b.) with simulation results

The comparison of the power consumption shows a good agreement. In both simulated periods of one week in January and September respectively the simulation result is slightly higher than the measurement. The relative deviation is less than 7.7 % (Fig. 7). Obviously, the power consumption of the plant is overpredicted when the production cooling is off .

Item	**Measurement** 29.04.04, 13:00	**Simulation** 23.05.03, 13:00
Heat reclaim	ϑ [°C]	ϑ [°C]
HX "Excess", water	25 / 38	29 / 37
HX "Condenser", water	20 / 28	17 / 29
HP screw 1, oil	52	45
HP screw 1, gas	70	60
HP screw 3, oil	55	45
HP screw 3, gas	68	63
Receiver	p_c [bar]	p_c [bar]
Conden. pressure	11.5	11.7
Aux.		
Ambient temp.	20 °C	20 °C

Table 1: Comparison of temperatures and pressures for one operating point with comparable boundary and load conditions

In Tab. 1 some temperatures and pressures displayed by onboard information systems or thermometers are listed for one operating point in April. Those values were compared to the corresponding values of the plant simulation at a similar load condition of the previous year. Especially, the simulated saturation pressure in the condensers, responsible for the attainable heat recovery, matches the value of the measurement. The same applies for the cooling water temperatures in the excess and condensing heat exchanger. A greater deviation can be seen in the gas and oil temperatures of the high pressure compressors. It must be pointed out that the position of the oil temperature sensor could not be clarified. Hence, the model of the oil cooling unit was not calibrated again but the parameters of the plant documentation were used.

5 Exergy-analysis of the Refrigeration System

For estimating savings potential it is important to know where the dominating loss mechanisms of a process are located. Such losses may be noticed in form of heat transfer, power decrease, mixing and pressure resistances. For the purpose of a clear description of process efficiencies it is necessary to define how much of an energy portion can be transformed into any other form of energy. For example, it is not possible to transfer heat from a cooler to a warmer volume in order to produce power. It is even not permitted by the second law of thermodynamics to completely turn heat into power by reducing the temperature of a medium to ambient conditions. The exergy E represents that part of energy which is technically useful and can be extracted without restrictions to work. The specific exergy e is expressed by:

$$e = \underbrace{h - h_0 - T_0\,(s - s_0)}_{\text{thermal}} + \underbrace{0.5(c^2 - c_0^2)}_{\text{kinetic}} + \underbrace{g(H - H_0)}_{\text{potential}} \quad (16)$$

The (specific) exergy always depends on the definition of ambient conditions indicated by the index 0. It is not always trivial to select the "correct" ambience model and the discussion about this issue is not finished, yet. Nevertheless, the exergy represents a powerful tool for analysing energy systems.

Item	**Total change of exergy flow** $\Delta\dot{E}^{tot}$ [kW]	**Inner cost flow** $\dot{K}^{i}$ [€/h]
$1 \rightarrow 2$	-62.15	-4.60
$3 \rightarrow 4$	-142.42	-10.58
$4 \rightarrow 5$	-12.10	-0.90
$5 \rightarrow 6$	-92.02	-6.83
$5 \rightarrow 7$	-44.30	-3.29
$5 \rightarrow 8$	-8.48	-0.63
$6,7,8 \rightarrow 9$	2.50	0.19
$9 \rightarrow 10$	-23.98	-1.78
$10,2,13 \rightarrow 3,14$	-11.96	-0.89
$12 \rightarrow 13$	-47.18	-3.50
$14 \rightarrow 15$	-4.24	-0.31
$15,17 \rightarrow 1,16$	-0.76	-0.06
$16 \rightarrow 17$	-195.33	-14.50
Total	-642,42	-47.68

Table 2: Inner costs resulting from exergy losses without heat reclaim for one hour continuous operation (see Fig. 1 for items) – The specific cost for exergy is 0.074 €/kWh$_{ex}$

In order to express the exergy losses in the corresponding components in terms of hourly costs the change of exergy is calculated first for an characteristic operating point with active production cooling. The cooling requirement is 485 kW for storage rooms (-10 °C) and 985 kW for production (-35 °C). At the same time a domestic hot water consumption of 4.5 kg/s takes place. Kinetic and potential forms of exergy are neglected and the reference point is set to p_0 = 1 bar

Item	**Thermal change of exergy flow** $\Delta\dot{E}^i$ [kW]	**Power consump.** P_{el} [kW]	**Change of enthalpy flow** $\Delta\dot{H}$ [kW]	**Water consump.** $\dot{m}_{fr}$ [kg/s]	**Waste water flow** $\dot{m}_A$ [kg/s]	**Outer cost flow** $\dot{K}^o$ [€/h]
$1 \rightarrow 2$	104.15	166.30	145.63	0.00	0.00	11.64
$3 \rightarrow 4$	255.18	397.60	173.09	0.00	0.00	27.83
$4 \rightarrow 5$	-12.10	0.00	-80.64	0.00	0.00	0.00
$5 \rightarrow 6$	-53.02	39.00	-1056.08	0.31	0.08	4.58
$5 \rightarrow 7$	-25.10	19.20	-494.50	0.15	0.04	2.21
$5 \rightarrow 8$	-8.48	0.00	-166.95	0.00	0.00	0.00
$6,7,8 \rightarrow 9$	2.50	0.00	0.00	0.00	0.00	0.00
$9 \rightarrow 10$	-23.98	0.00	0.00	0.00	0.00	0.00
$10,2,13 \rightarrow 3,14$	-11.96	0.00	0.00	0.00	0.00	0.00
$12 \rightarrow 13$	-41.18	6.00	485.76	0.00	0.00	0.42
$14 \rightarrow 15$	-4.056	0.00	0.00	0.00	0.00	0.00
$15,17 \rightarrow 1,16$	-0.76	0.00	0.00	0.00	0.00	0.00
$16 \rightarrow 17$	-181.03	14.30	983.26	0.00	0.00	1.00
VII $\rightarrow$ I	-19.03	0.70	-372.68	0.00	0.00	0.05
I $\rightarrow$ II	5.46	0.00	169.24	0.00	0.00	0.00
III $\rightarrow$ IV	5.15	0.00	87.59	0.00	0.00	0.00
V $\rightarrow$ VI	10.52	0.00	115.04	0.00	0.00	0.00
IV,VI $\rightarrow$ VII	-2.08	0.00	0.00	0.00	0.00	0.00
B $\rightarrow$ C	13.42	0.70	370.97	0.00	0.00	0.05
C $\rightarrow$ D	-0.68	0.00	0.00	0.00	0.00	0.00
D $\rightarrow$ E	-8.97	0.00	106.32	0.00	0.00	40.93
D $\rightarrow$ F	-25.42	0.00	241.12	0.00	0.00	92.83
D $\rightarrow$ G	-12.85	0.00	89.23	0.00	0.00	34.35

Table 3: Change of exergy and enthalpy flow rate for the refrigerant, power and water consumption, waste water mass flow rate and outer cost flow invested - power rates and media prices are listed in Sec. 3

and ϑ_0=12.4 °C (fresh water temperature entering the plant). The change of exergy and enthalpy with regard to the refrigerant or cooling water flow is shown in Tab. 3. An increase of exergy ($\Delta\dot{E}^i > 0$) happens in the compressors and in those heat exchanger volumes which show a rising temperature (cooling water HX). Under the assumption of continuous operation for one hour the plant consumes a total of 643.8 kWh exergy in form of electricity. This effort has to be compared to the thermal profit of the plant which is defined by the exergy provided by the evaporators and the water tank to the DHW system. Hence, the exergetic efficiency ζ follows from the ratio of the actual thermal advantage to the total exergy input ($\sum P_{el}$):

$$\zeta = \frac{\sum \dot{E}_Q}{\sum P_{el}} = \frac{|\Delta\dot{E}^i_{12,13}| + |\Delta\dot{E}^i_{16,17}| + \Delta\dot{E}^i_{B,D}}{\sum P_{el}} = 0.36 \tag{17}$$

This value is more plausible than the COP which equals 2.28 at the same time. If an economical analysis shall be carried out it is possible to combine the change of exergy flow rate $\Delta\dot{E}^i$ with outer cost flows $\dot{K}^o$ (see Tab. 3) resulting from power and water consumption. This method is described as “exergy costing” by Bejan [1]. In a simplifying approach it can be postulated that all outer costs are divided by the exergy input in order to calculate the specific costs of exergy. With this average value the costs of internal losses are expressed (see Tab. 2). Generally, the initial costs (e. g. capital investment) should also be included but in this case an operating plant is considered and it should be investigated how the efficiency could be improved without installing new components. Therefore, the task was not to compare different components with different initial costs and thus this contribution was neglected.

The total amount of all costs for this operation mode is 48 €/h. The largest cost centre in terms of exergy destruction is encountered in the cooling towers (10 €/h) followed by the expansion valves (2.10 €/h) and the phase separator of the high stage (0.89 €/h) due to the internal heat transfer. Hence, financial savings can be obtained by reducing exergy destruction in the evaporative condensers (e. g. by lowering the saturation pressure or increasing the mass flow rate through the water-cooled condenser).

Item	**Cost flow** $\dot{K}^i$ [€/h]	**Revenue** $\dot{G}$ [€/h]	**Virt. op. profit** $\dot{P}$ [€/h]
$1 \rightarrow 2$	-4.60	0.00	-4.60
$3 \rightarrow 4$	-10.58	4.39	-6.19
$4 \rightarrow 5$	-0.90	3.34	2.44
$5 \rightarrow 6$	-6.83	0.00	-6.83
$5 \rightarrow 7$	-3.29	0.00	-3.29
$5 \rightarrow 8$	-0.63	6.46	5.83
$6,7,8 \rightarrow 9$	0.19	0.00	0.19
$9 \rightarrow 10$	-1.78	0.00	-1.78
$10,2,13 \rightarrow 3,14$	-0.89	0.00	-0.89
$12 \rightarrow 13$	-3.50	0.00	-3.50
$14 \rightarrow 15$	-0.31	0.00	-0.31
$15,17 \rightarrow 1,16$	-0.06	0.00	-0.06
$16 \rightarrow 17$	-14.50	0.00	-14.50
Total	-47.68	14.20	-33.48

Table 4: Costs due to exergy losses and destruction, revenue of heat recovery and virtual operating profit in comparison to conventional system for one hour continuous operation – The specific gain of recovered heat is 0.039 €/kWh$_{th}$

Useful changes of exergy in the evaporators cost 18 €/h so that this can be considered as the minimum running cost level if the insulation of the rooms or other consumers could not be further improved.

Up to now the positive impact of the heat reclaim is missing in this study. To attain the total balance of costs and revenues the gas savings from the DHW system are propagated *upstream*. By means of cooling water a total of 371 kW waste heat is recovered from the ammonia or oil, respectively. This is almost 20 % of the waste heat produced by the cycle. 115 kW are contributed by the oil coolers (6 %) and 80 kW (4 %) by the excess heat exchangers. Helmke [14] even states a potential of 7.4 % for excess heat and 9.2 % for oil cooling (only high pressure screws).

The actual heat recovery depends strongly on the availability of cooling water which has only low temperatures in case of DHW consumption. In order to supply the demand of hot water (45 °C, 60 °C and 90 °C) a heating capacity of 807 kW is needed. Assuming an efficiency of 90 % for heating and specific heat costs of 0.039 €/kWh$_{th}$ the running costs of a conventional plant would be 31.07 €/h. Taking the recovered heat into account the costs for natural gas drop to under 17 €/h and therefore the heat exchangers for heat reclaim are not representing a loss of 1.50 €/h but a *virtual operating profit* of 8.20 €/h. In addition, the oil coolers also show a revenue of 4.39 €/h reducing the loss of the HP screws from 10.58 €/h to 6.19 €/h. A complete coverage of the DHW supply by the refrigeration system is not possible as long as temperatures of more than 60 °C are needed. But it would be possible to realise higher savings (see Tab. 4) if a consumer of 1.913 kW at a low temperature of approx. 30 °C could be found or if the temperature level of the high pressure cycle could be increased **during** production. Moreover, in future low-exergy consumers and storage systems will be available for heating systems and buildings so that more energy can be saved. Currently, 47.68 €/h have to be invested in the refrigeration system (and 16.77 €/h in the DHW system, resp.).

6 Improvement Measures and Component Optimisation

From the exergy-analysis follows that a higher condensation pressure offers a higher potential for heat reclaim. This is only worthwhile if the DHW consumption is high enough. Therefore a mass flow depending control for the cooling towers is implemented. During production the saturation temperature is lifted by a throttling of the tower ventilation from a max. value of 10 bar to 14 bar. Outside production times it is important to achieve low condensation pressures (min. 7 bar) in order to reduce the power consumption (see Fig.8).

Additionally, the compressor cascade is changed so that the base requirement in winter is provided by the smallest compressor because the COP is generally better with a higher part load factor. In summer the cascade order remains the same because the base load for cooling storage rooms is often higher than the maximum capacity of the smallest compressor.

For both periods in summer and winter the running costs can be reduced by 4 % (see Tab. 5) due to the increased heat recovery (≈ 20 %) at a slightly higher power consumption (1 to 2 %). For those savings there are basically no large investments needed. Compared to the running costs of the refrigeration plant (≈ 200.000 €/a) for one year the possible reduction is 5.000 €/a.

The simulation of one week takes 12 hours on a fast PC (3 GHz processor) due to on/off-control of the

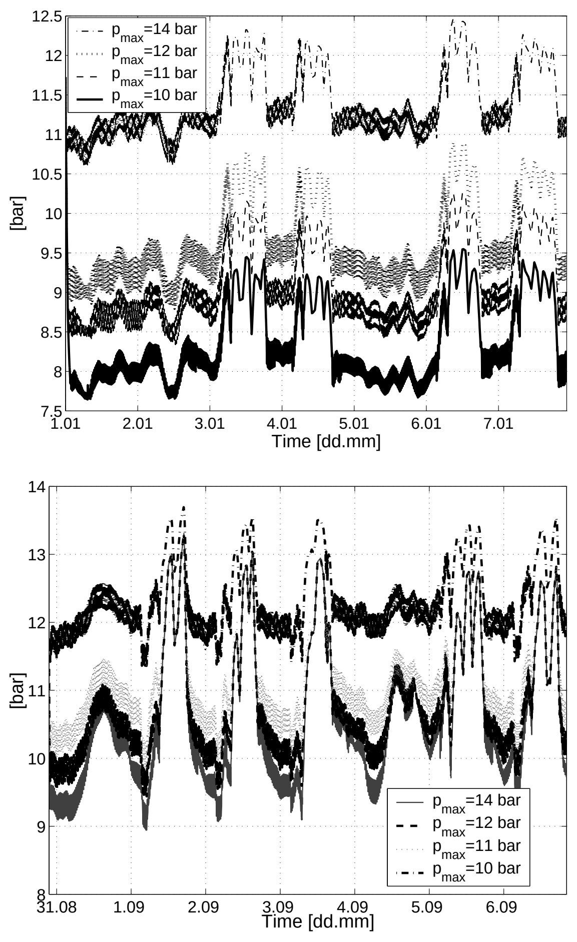


Figure 8: Impact of the variable cooling tower control on the condensation pressure in January (t.) and September (b.)

compressors at part load. If frequent events could be avoided by a part load function or if a higher plant utilisation is considered the simulation time would be reduced to approximately 1 hour.

7 Conclusions and Outlook

It is possible to simulate even complex refrigeration plants for longer periods like weeks with a high amount of unsteady events resulting from a 2-point-control of some components. A modelling approach aiming to further reduce events during simulation (e. g., performance curves for on/off-controlled elements) would yield faster simulation times for calculating balances of whole years. Attaining this goal is important since the boundary conditions profile (test reference year) has a dominating influence on the total power consumption.

Mode	Jan. act.	Jan. mod.	Sep. act.	Sep. mod.
Power con. [kWh]	46.285	46.887	60.123	61.319
Fresh water [m^3]	118	126	346	310
Waste water [m^3]	39	42	115	103
Heat reclaim [kWh]	22.795	27.127	29.966	34.241
Costs [€]	2.557	2.446	3.627	3.487
Rel. dev. [%]	-	-4.3	-	-3.9

Table 5: Comparison of power and water consumption, heat reclaim and running costs for one week in January and September of the actual and modified plant

When considering multiple consumers (e. g., refrigeration at different temperature levels, DHW) the system's control is a key factor for realising an efficient plant operation. In this paper, it was shown that in even well-designed plants incorporating state-of-the-art subcomponents savings are attainable without much capital investment. The transient simulation offers a method for a holistic analysis of technical systems throughout the product-life-cycle. In combination with an exergy-analysis it is possible to find optimisation potential for characteristic operating points. Basically, the implementation of the exergy method is easy when necessary medium properties are provided by the control volume models (see Eq. 16). Nevertheless, the evaluation may become tedious for complex dynamic systems especially when economical constraints (energy or exergy costs) have to be considered. For this purpose capable validation and evaluation methods must be implemented to concisely provide the information needed for drawing correct conclusions and finding effective improvement measures.

References

[1] A. Bejan, G. Tsatsaronis, and M. Moran. *Thermal Design and Optimization*. John Wiley & Sons, New York/Chichester/Brisbane, 1996.

[2] V. M. Brodyansky, M. V. Sorin, and P. Le Goff. *The efficiency of industrial processes: Exergy analysis and optimization*. Elsevier, Amsterdam / London / New York, 1994.

[3] St. Wischhusen and G. Schmitz. Transient simulation as an economical analysis method for energy supply

systems for building or industry. *Applied Thermal Engineering*, 24:2157 – 2170, 2004.

[4] St. Wischhusen and G. Schmitz. Numerical Simulation of Complex Cooling and Heating Systems. In *Proceedings of the 2nd Int. Modelica-Conference*, pages pp. 183–191, Oberpfaffenhofen, 2002. Modelica Association.

[5] St. Wischhusen, B. Lüdemann, and G. Schmitz. Halbhermetische hubkolbenverdichter. In *Proceedings of the 3rd Modelica Conference*, pages pp. 259–267, Linköping, Schweden, 2003. Modelica Association.

[6] T. Pfafferott. *Dynamische Simulation von* CO_2*-Kälteprozessen für mobile Anwendungen*. Shaker Verlag, Aachen, 2005.

[7] H.-D. Baehr and R. Tillner-Roth. *Thermodynamische Eigenschaften umweltverträglicher Kältemittel*. Springer Verlag, Berlin, 1995.

[8] J. M. Jensen. *Dynamic modeling of Thermo-Fluid Systems - With focus on evaporators for refrigeration*. Department of Mechanical Engineering, Technical University of Denmark, Lyngby (Dänemark), 2003.

[9] R. Shah and D. Sekulić. *Fundamentals of Heat Exchanger Design*. John Wiley & Sons, Hoboken, USA, 2003.

[10] W. Stoecker. *Industrial Refrigeration Handbook*. McGraw–Hill, New York/San Francisco/Washington D. C., USA, 1998.

[11] W. Leidenfrost and B. Korenic. Experimentelle Überprüfung einer Berechnungsmethode für die Leistungsvoraussage verdunstungsgekühlter Kondensatoren. *Brennstoff–Wärme–Kraft*, 34(1):9–14, 1982.

[12] H.-M. Hellmann. *Untersuchung und einheitliche Berechnung des Betriebsverhaltens von Verdunstungskühlern und –kondensatoren*. Phd thesis, Technische Universität Braunschweig, Braunschweig, 1991.

[13] P. Stabat and D. Marchio. Simplified model for indirect-contact evaporative cooling-tower behaviour. *Applied Energy*, 78:433–451, 2004.

[14] T. Helmke. NH_3-Verbundkälteanlagen mit Schraubenverdichtern kleiner Leistung. *KK - Die Kälte- und Klimatechnik*, (5), 2000.

Application of the Modelica library *WasteWater* for optimization purposes

T. Ziehn*, G. Reichl*, E. Arnold**

* Technische Universität Ilmenau
Department of Automation and
Systems Engineering
P.O. Box 10 05 65, 98684 Ilmenau, Germany
tilo.ziehn@stud.tu-ilmenau.de,
gerald.reichl@tu-ilmenau.de

** Fraunhofer-IITB
Application Center Systems Technology
Am Vogelherd 50, 98693 Ilmenau, Germany
eckhard.arnold@ast.iitb.fraunhofer.de

Abstract

The following work is a contribution to dynamic optimal control strategies of an activated sludge model. The model is applied to the waste water treatment plant (WWTP) in Jena, Germany. The model is implemented using the Dymola software package with the application of the free available `Modelica` library *WasteWater*. On the basis of this model open-loop and closed-loop (Model Predictive Control MPC) optimizations are applied and the results are evaluated. The main focus is on the variable operating costs of the WWTP.

Keywords: *waste water treatment, dynamic optimization, model predictive control*

1 Introduction and problem description

Nowadays new waste water treatment plants are designed for reliability and safety, not for operational cost efficiency. Further more sustainable sewage management, which is subject to increasing legal requirements, plays an important role. Therefore, the application of simulation and optimization methods to the waste water purification process is necessary. The principal purposes are the reduction of the load for the environment (adherence to the limits of the effluent parameters) with simultaneous minimization of the waste water treatment costs.
The free available `Modelica` library *WasteWater* [1] was developed and is successfully applied to the WWTP Jena. In this plant blowers and pumps are controlled by simple SISO control loops with standard controllers (usually two-point controllers).
The WWTP Jena is comperatively well equiped with various on-line measurement devices including COD, NH_4^+-N, NO_3^--N and PO_4^{3-}-P. Only on the basis of this equipment, investigations concerning dynamic control strategies become possible.
The cleaning achievement of a WWTP can be evaluated with the help of the effluent parameters of the receiving water. Some of these effluent parameters (e.g. NH_4^+-N) of the WWTP Jena are considerably below legal limits. As a result the operating costs are higher than is necessary. The electrical energy costs of the blowers and pumps, as well as the sludge disposal costs, represent the main part of the variable costs which are required to operate the WWTP Jena.
The aim of this work is to find optimal trajectories for blowers and pumps by using dynamic optimization methods. Mainly dry weather scenarios were chosen from the stored data for simulation and optimization purposes. As a result the minimization of the operation costs of the WWTP, with simultaneous adherence to the limits of the effluent parameters, becomes possible. These results are used as a basis for further investigations concerning closed-loop operation (Model Predictive Control MPC).

2 Modelling of the WWTP Jena

The dynamic model of the WWTP is of crucial importance for the model-based optimization. There is a multiplicity of mathematical models, which describe the waste water purification process. However, these models are almost exclusively applied for simulation purposes. Due to the complexity of the used model an object-oriented approach is worthwhile.

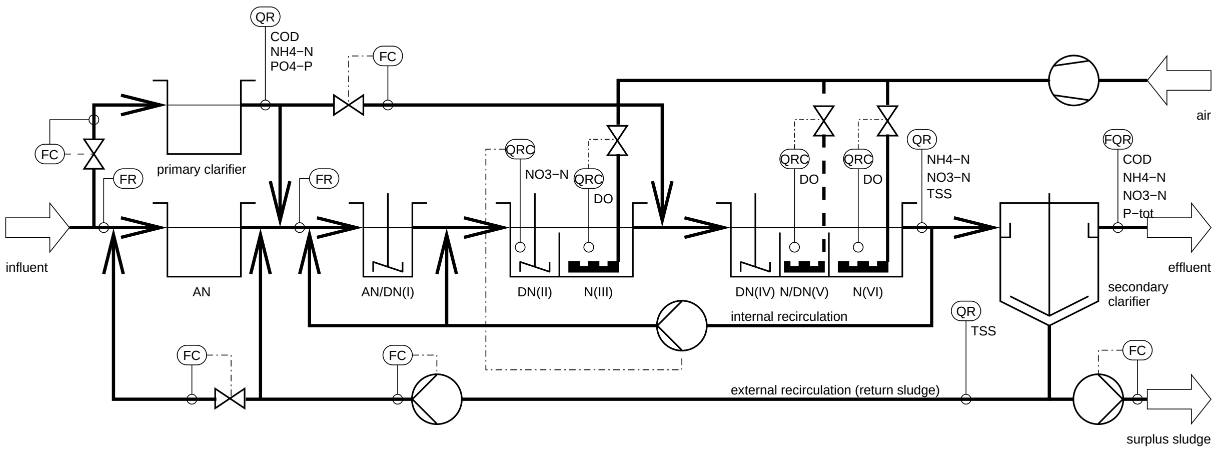


Figure 1: Simplified sytem of the Jena WWTP

The programming language `Modelica` represents this object-oriented approach. In this context the software package Dymola was used for simulation, since the C++ code, generated automatically by Dymola, is particularly suited for optimization purposes. The free `Modelica` library *WasteWater* was used for the implementation of the model of the WWTP Jena. The library contains the Activate Sludge Models (ASM) No. 1 to No. 3 [2] and different multi-layer models for secondary clarifier.

The WWTP Jena is characterised by a connection size of 145,000 people equivalent (p.e.), a cascade-type denitrification with pre-clarification, biological and chemical phosphorus removal and sludge digestion. The bio-gas produced by sludge fouling is used in a block-type thermal power station of 2x250 kW. A simplified system of the plant is shown in Figure 1.

Existing control loops (usually two-point controllers) are isolated in and/or removed from the model for the optimization process, with the goal being to find optimal trajectories (e. g. the control of the blowers for the air supply into the nitrification tanks). Both, the physical limits of the control variables (blowers, pumps) and the limits of the effluent parameters (e. g. maximum 1 mg NH_4^+/l) must be kept.

The model of the WWTP Jena is implemented by using the Activated Sludge Model No. 2d of the IWA task group, which is part of the `Modelica` library *WasteWater*. A summerised description of the model reads:

$$\frac{dS_i}{dt} = (S_{i,in} - S_i)\frac{Q_{in}}{V} - r_i, \qquad i \in \{F, A, NH, NO, PO, I, ALK, N\} \tag{1}$$

$$\frac{dS_O}{dt} = (S_{O,in} - S_O)\frac{Q_{in}}{V} + r_O + r_{air}, \tag{2}$$

$$\frac{dX_i}{dt} = (X_{i,in} - X_i)\frac{Q_{in}}{V} - r_i, \qquad i \in \{I, S, H, PAO, PP, PHA, AUT, TSS, MeOH, MeP\} \tag{3}$$

In equation (1) the index i stands for the different dissolved concentrations such as inert organic matter (S_I), substrate ($S_F + S_A$), nitrate nitrogen (S_{NO}), etc., and in equation (3) for the particular concentrations, which are amongst others the heterotrophic (X_H) and autotrophic (X_{AUT}) biomass. Variables subscripted by index 'in', e.g. $S_{i,in}$, indicate concentrations in the flow Q_{in} entering a considered tank. Equation (2) describes the balance of the dissolved oxygen and has an additional term for the oxygen uptake (aeration r_{air}) caused by the blowers. The reaction rates r_i and r_O in the balance equations (1) - (3) are given by the model matrix of the ASM No. 2d. It models 19 relevant concentrations (state variables) and 21 processes per tank. The WWTP Jena model is described by approximatly 3000 equations and 250 state variables. The complete description and development is available in [2] .

The object oriented approach implemented in `Modelica` combines the advantages of a hierachical model structure and the reusability of model components in a multi-domain modelling environment of complex dynamic systems. The model components such as nitrification tank, secondary clarifier, blower, flow mixer, divider and so on have to be defined for all types of ASM models because of differing variables and the number of variables. The components are characterised by internal variables declared independently of other components, and by connectors linking the components.

3 Optimal control problem

The optimization based control requires the dynamic model of the WWTP (developed in section 2). The whole model and control problem transformation procedure is done automatically by exporting a compiled flat model representation of the WWTP in C++ from Dymola [3], that can be used by the optimization solver Hqp/Omuses [4], [6].

3.1 Open-loop problem

According to the problem described in section 1 different nonlinear optimal control problems can be formulated and solved taking into account different objective functionals. The main aim of the optimization is to reduce operating costs, e.g. the electrical energy cost taking into account the time-dependent electricity tariff, respectively. The objective functional is minimised with respect to the constraints. Restrictions for the state variables results from legal limits of effluent concentrations of the WWTP to the receiving water. The controls are limited due to the maximum installed pump and blower capacity.
Using the multi-stage control parameterisation technique described in [4], the continuous optimal control problem is approximated by a constrained discrete-time optimal control problem, that reads:

$$\begin{aligned} J \;=\; & F(\mathbf{x}^K) + \sum_{k=0}^{K} \Big[\rho_e^k f_{o,e}^k(\mathbf{x}^k, \mathbf{u}^k, \mathbf{z}^k) + f_{soft}^k + \\ & \rho_s^k f_{o,s}^k(\mathbf{x}^k, \mathbf{u}^k, \mathbf{z}^k) + \rho_o^k c_o f_{o,c}^k(\mathbf{x}^k, \mathbf{u}^k, \mathbf{z}^k) \Big] \end{aligned} \quad (4)$$

with $f_{o,e}$ - electrical energy costs resulting from time-dependent tariff and electrical energy demand for blower and pump operation, $f_{o,s}$ - sludge disposal costs depending on waste sludge flow rate and composition, taking into account profit from bio-gas utilisation, $c_o f_{o,c}$ - chemical dosage costs (negligible), f_{soft}^k - penalty term for soft constraints (slack variables), $\rho_e^k, \rho_s^k, \rho_o^k$ - weighting factors, $\mathbf{x}^k$ - model state variables, $\mathbf{u}^k$ - control inputs and $\mathbf{z}^k$ - non-controllable inputs (e.g. inflow and concentrations).
The electrical energy costs can be formulated as follows:

$$f_{o,e} = \int_{t_0}^{t_e} c(t) \cdot p_{el}(t) dt \quad (5)$$

with $c(t)$ - time dependent tariff and p_{el} - electrical energy depending on the air supply into the nitrification tanks. A linear connection between the electrical energy and the amount of air flow into the nitrification tanks could be determined. The blowers responsible for the air supply into the nitrification tanks N(III), N/DN(V) and N(VI) (Figure 1) represent the main part of the variable electrical energy costs. The costs resulting from pumps are negligible.
The constrained discrete-time optimal control problem is numerically solved as a large-scale and structured nonlinear programming problem in the state and control variables.

3.2 Closed-loop problem

The Model Predictive Control (MPC) offers the possibility to merge an optimal control problem solution into a closed-loop. One substantial advantage in comparison to simple control loops is in the treatment of constraints for the controls and the state variables. In this work a nonlinear MPC is considered with an economic cost function. There is no setpoint because of external input dynamics. The MPC algorithm can be summerised in the following steps:

1. Set: $\hat{k} = 0$
2. State estimation: Get the current states $\mathbf{x}^{\hat{k}|\hat{k}}$ using past measurements, e.g. Nonlinear Moving Horizon State Estimator (NMHSE).
3. Prediction: Get a prediction of the non-controllable inputs $\mathbf{z}^{k|\hat{k}}$ (reads: inputs at the time k calculated at the time $\hat{k}$), $k \in [\hat{k}, \hat{k}+K-1]$ for a given horizon K (prediction horizon) using the process model (summerised description equations (1) - (3)).
4. Optimal control problem: Calculate the future control signal $\mathbf{u}^{k|\hat{k}}, k \in [\hat{k}, \hat{k}+K-1]$ by solving an optimal control problem (reference to section 3.1 and equation (4)) with subject to the process model and the constraints.
5. Applying: Send the control signal $\mathbf{u}^{\hat{k}|\hat{k}}$ to the process.
6. Shift: $\hat{k} = \hat{k} + 1$ and go back to step 2.

Many linear MPC approaches have found successful applications and important issues such as online computation, robustness and stability are well addressed. Within nonlinear MPC, research is still in progress. Therefore, and since the implemented WWTP Jena model has more than 250 state variables with only 10 measureable outputs, the following application of the

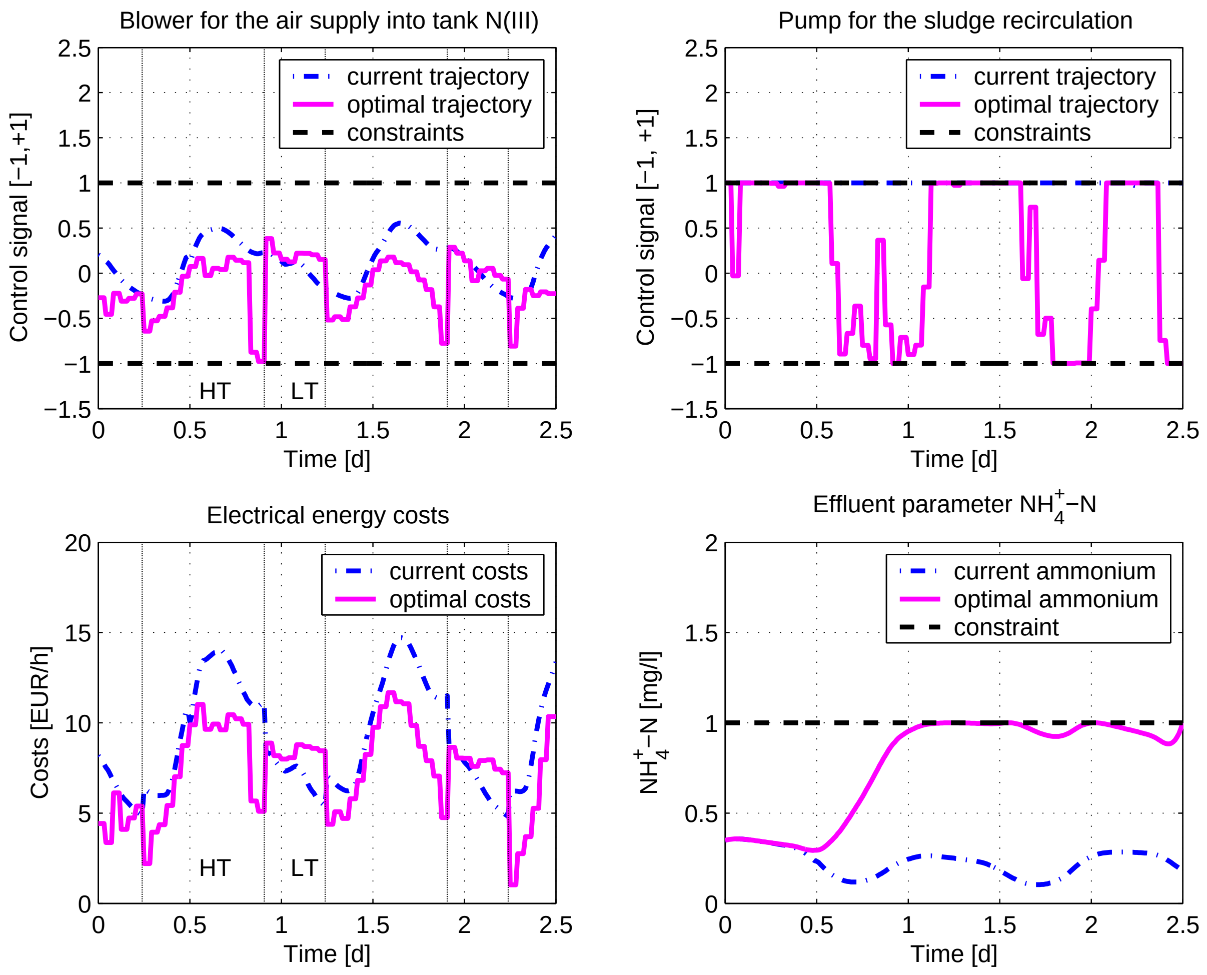


Figure 2: Open-loop optimization results (control signals, electrical energy costs and effluent value for NH_4^+-N)

MPC to the model of the WWTP Jena is only considered on nominal conditions. It is assumed that an exact model of the existing plant is beeing used without any disturbances and that all state variables are available. Because of these assumptions the state estimation is not considered in this work. However, a succesful approach of the nonlinear moving horizon state estimator to an activated sludge model can be found in [7] and an application of MPC for an ASM No. 1 can be found in [8].

For further investigations on the WWTP Jena model it is advisable to reduce the number of state variables. This can be done for example by using a simple secondary clarifier instead of a multi-layer secondary clarifier.

Another simplification is made by assuming that the non-controllable inputs can be determined exactly and so the prediction (step 3) is not applied. Only on-line measured data of the WWTP Jena is used. When applying the MPC algorithm to the real plant investigations regarding the prediction of the non-controlable inputs (inflow and concentrations) are still necessary. In this context it is also important to examine how prediction errors affect the results of the applied trajectories. The legal limits of the effluent parameters of the WWTP must always be guaranteed.

4 Results

Primarily dry weather scenarios are considered in the open-loop investigations. The main point is the minimization of the electrical energy costs taking into account the effluent parameter limits of the WWTP.

Figure 2 shows the results of an open-loop optimization process. The optimal trajectory of one of the blowers is presented in comparison to the trajectory resulting from the control by the basic control loops (Figure 2 upper left). The different electrical energy tariffs (HT - high tariff and LT - low tariff) are marked by vertical lines. A piecewise constant approximation of the control variables was chosen to support

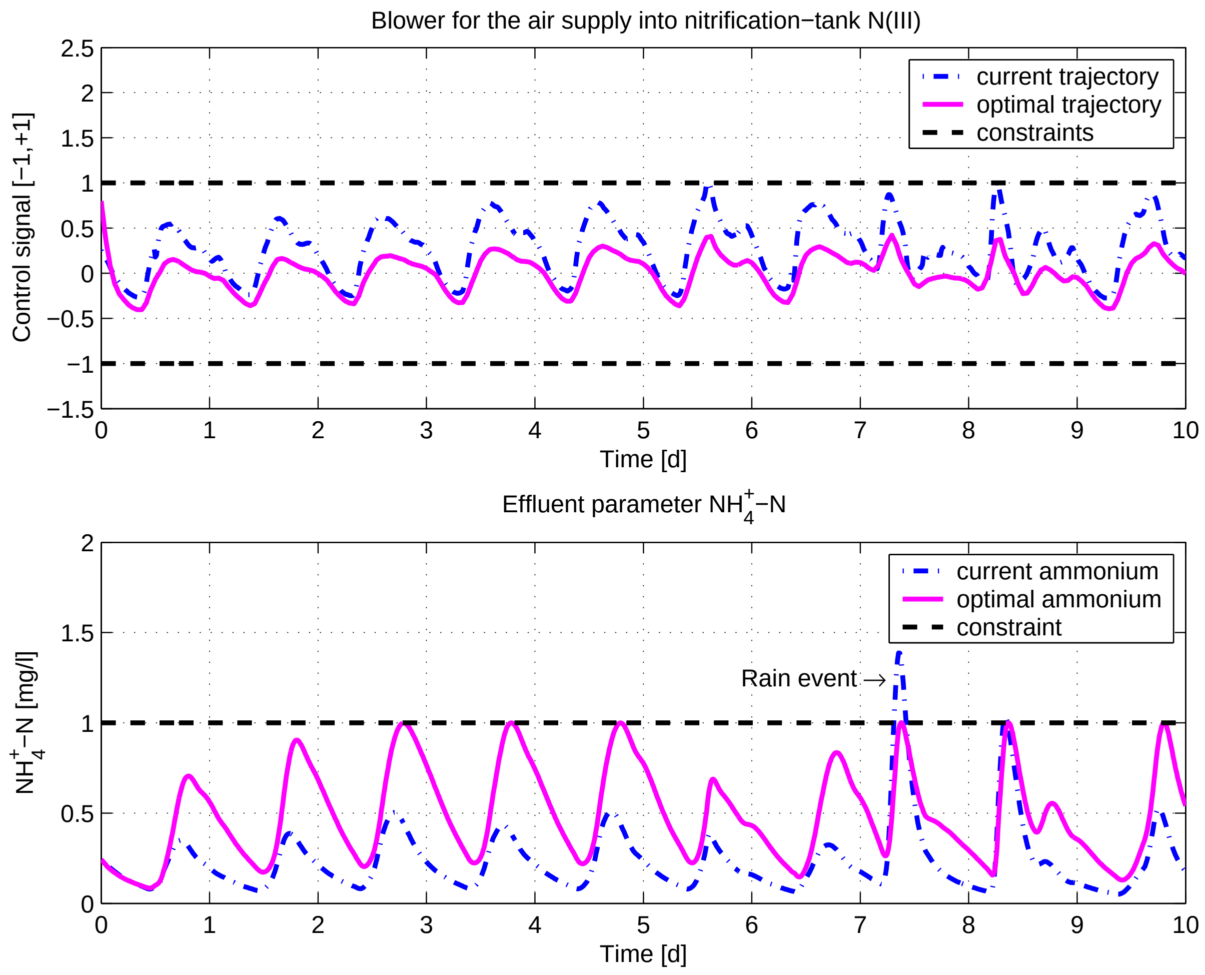


Figure 3: MPC results (effluent value for ammonium and control signal of the blower for the air supply)

the use of the tariffs more efficiently. In the right lower part of Figure 2 the effluent parameter of ammonium (NH_4^+-N) is represented. It shows that the chosen upper constraint for ammonium (1 mg NH_4^+-N/l) becomes active. With optimal trajectories of the blowers the electrical energy costs can be reduced by more than 10 % (Figure 2 lower right). It is even possible to achieve a higher reduction of the electrical energy costs (up to 15 %) by including additional control variables (e.g. the pump for the sludge recirculation, Figure 2 upper right) in the optimization process.

The legal limit for the effluent parameter NH_4^+-N is about 10 mg/l and therefore much higher than the applied limit of 1 mg/l. However, all investigations are considered with dry weather scenarios. The limit for NH_4^+-N is reduced in order to guarantee that the legal limit can be kept even under uncertain conditions. With a higher limit for NH_4^+-N (e.g. 5 mg/l) the electrical energy costs can be reduced by more than 20 %.

In addition the sludge costs are taken into account and the control variable responsible for the sludge removal is included in the optimization process. However, it is necessary to limit the total suspended solids (TSS) in the secondary clarifier, otherwise too much sludge would be removed from the whole process. The investigations also consider the bio-gas utilization and the profit obtained. Further results can be found in [5].

The results of the open loop investigations (due to different scenarios) can be merged with the help of the Model Predictive Control (MPC) into a closed-loop. The influence of the prediction horizon and the stepsize on the results is examined. For all further investigations a data record is used, which contains dry weather data and a rain event.

It can be shown that the prediction horizon is of little importance for optimization results, taking into account the electrical energy costs. Different prediction horizons from 0.5 days up to 5 days are examined, with stepsizes from 0.25 days up to 1 day. The results (electrical energy costs) are nearly the same. Figure 3 shows the results with a prediction horizon of 2 days and a stepsize of 1 day. The upper part of Figure 3 rep-

resents the control signal of one of the blowers. The resulting effluent parameter NH_4^+-N is presented in the lower part of Figure 3. With occurrence of the rain event (day 8) the chosen upper constraint can not be kept by using the implemented simple control loops. Only by applying Model Predictive Control strategies can the constraint for NH_4^+-N be kept.
In the case of dry weather (day 1 to day 5) it is possible to save approxamately 11 % on electrical energy costs. During the entire investigation period of 10 days it is even possible to save more than 12 % with the optimal trajectories. The following control variables are taken into account: the control of the blowers, the control of the recirculation of the sludge and the control of the return sludge. With the control of the return sludge and the recirculation it becomes possible to shift activated sludge at short notice (from the biology into the secondary clarification and in reverse). However, altogether no TSS is extracted from the process.
Also investigations concerning all costs (electric energy costs and sludge costs) are examined. In this case the optimization horizon has substantial influence on the results. The control for the sludge removal is of substantial importance. In the case that the rain event is not predicted early enough too much sludge is extracted from the process and with the occurrence of the rain event it is not possible to keep the necessary cleaning achievements (constraints, e.g. 1 mg NH_4^+-N/l) of the water since insufficient biomass is available in the tanks. Only prediction horizons smaller than 6 days could be investigated since the computer capacity was exhausted at this point.

5 Summery and conclusion

In this work the dynamic optimization method was applied to a WWTP model. With the help of the library *WasteWater* an ASM No. 2d model of the WWTP Jena was examined and evaluated. Different scenarios were provided and mainly dry weather data were used for simulation and optimization. The influence of different control variables was investigated by using open-loop optimizations. These results could be used in a closed-loop by applying Model Predictive Control strategies. It could be shown that regarding the electrical energy costs more than 10 % can be saved in comparision to existing simple control loops. However, to apply the results and the MPC strategy to the real plant further investigations are still necessary. The prediction of the non-controllable inputs is not implemented yet or the state estimation for the state variables.

References

[1] G. Reichl. *WasteWater* - a Library for Modeling and Simulation of Wastewater treatment plants. In: Proceedings of the 3rd Modelica Conference, Linköping, Sweden, Modelica Association, 3-4 November 2003.

[2] M. Henze, W. Gujer, T. Mino and M. v. Loosdrecht. Activated sludge models ASM1, ASM2, ASM2d and ASM3. Technical report, IWA task group, 2000.

[3] H. Elmquist et al. Dymola - dynamic modeling laboratory. User's manual. Dynasim AB. Sweden, 2001.

[4] R. Franke and E. Arnold. The solver Omuses/HQP for structured large-scale constrained optimization: algorithm, implementation, and example application, Sixth SIAM Conference on Optimization, Atlanta, 1999.

[5] T. Ziehn. Untersuchungen zur optimalen Steuerung der KA Jena mit Hilfe der Modelica-Bibliothek *WasteWater*. Ilmenau, Germany: Unpublished report, Dept. of Automation and Systems Engineering, Technische Universität Ilmenau, 2004.

[6] R. Franke. Formulation of dynamic optimization problems using Modelica and their efficient solution. Proceedings of the 2nd International Modelica Conference, Oberpfaffenhofen, 2002.

[7] E. Arnold and S. Dietze. Nonlinear moving horizon state estimation of an activated sludge model. In F. G. Filip, I. Dumitrache, and S. S. Iliescu, editors, Large Scale Systems: Theory and Applications. 9th IFAC/IFORS/IMACS/IFIP Symposium, pages 554-559, Bucharest, Romania, 2001.

[8] E. Arnold, S. Dietze, and G. Reichl. Application of model-predictive control for long-term operation of a municipal wastewater treatment plant. In K. Konarczak and D. Trawicki, editors, Technology, Automation and Control of Wastewater and Drinking Water Systems, TiASWiK'02, pages 211-216, Gdansk-Sobieszewo, Poland, 2002.

Session 4c

Tools I

Real-Time Simulation of Modelica Models under Linux / RTAI

Gianni Ferretti   Marco Gritti   Gianantonio Magnani   Gianpaolo Rizzi   Paolo Rocco
Politecnico di Milano, Dipartimento di Elettronica e Informazione
Via Ponzio 34/5, 20133 Milano

Abstract

The paper presents a concept and its implementation software modules to obtain ready to run real-time simulation code directly from Modelica models. Basically, a Modelica special model building block has been developed supporting the definition of the real-time input /output variables, their communication with external tasks or systems (e.g. a real hardware and software controller), and the scheduling of the periodic execution of the simulation task. The special module links to the real-time operating system (Linux with extension RTAI) through a special purpose C library. The real-time simulation of a 7-DOF space robotic arm is presented as a test case.

1 Introduction

Real-time simulation systems are mainly used for testing and check out of control electronics and other components of complex systems ("hardware-in-the-loop" simulation), like power plants, aircraft, vehicles, as well as for training of plant operators, aircraft pilots, and astronauts.
In real-time simulators model inputs must be acquired from external devices each sample time and model equations must be solved within fixed time intervals, so that a selected subset of computed variables can be output the next sampling time. To implement real-time communication with external world and to schedule model execution exactly each sampling time, the simulation software relies on system primitives whose calls are added to the model solution code. Usually, an effort is also necessary to simplify model equations most demanding from the point of view of computational burden.
Commercial tools exist that allow to adapt off-line models to real-time simulations on dedicated hardware. A typical situation consists in porting Simulink models to dedicated hardware using the Matlab Real Time Workshop [2] or the dSPACE TargetLink [6]. Simulink and also Dymola [1] models can be interfaced with dSPACE hardware to allow hardware in the loop simulations. Tools [6] exist also to assist the production of special Simulink models whose simulation can be run on multi-processor hardware.
On the other side, research efforts are spent to port the simulators obtained with open-source modeling tools like MBDyn [3] on real-time [5] possibly distributed platforms, like RTnet [4]. And also, efforts are spent to generate parallel code from Modelica models [10][11], to be eventually executed on supercomputer platforms [12].
This paper deals with the problem of obtaining ready to run real-time simulation code directly from Modelica models, so that already available models can be executed in real-time, and all the powerful Modelica libraries and the features and tools of a Modelica editor / compiler, such as Dymola, can be exploited for the development of new models.
The way this goal has been achieved is illustrated in this paper. In Section 2 the simulation platform requirements and main features are discussed. In Section 3 the proposed real time extension to Modelica is described. In Section 4 a test case of the real-time software modules is presented. A detailed open loop model of the Spider arm, a seven degrees of freedom Italian space manipulator, is exploited. The real-time simulated robot arm can also be controlled through a real-time software controller running on a second workstation. A short description of this software controller is given in Section 5.

2 The simulation platform

2.1 Requirements

The real-time dynamic simulation software should:

- satisfy the constraints of periodic real-time execution;
- be able to interface itself with external processes, possibly with hardware;

- be easily derived from models developed for the off-line simulation.

2.2 Real-time execution platform

The purpose of obtaining a real-time application imposes the choice of an operating system capable of supporting the execution of real-time processes.
The Linux operating system extended with the Real Time Application Interface (RTAI) [7] has been chosen. This operating system supports the execution of real-time processes, it is open-source, and it is widely used among the scientific community and in the European research centers.

2.3 Interoperability issues

Many custom libraries are available for Linux / RTAI. For the purposes of this project, the COMEDI (COntrol and MEasurement Device Interface) [8] library, developed by the open-source community, is particularly interesting. By means of a set of standard interfaces, COMEDI allows to manage the communication with hardware boards, and so provides a valid support for the data exchange on hardware channels. Actually, a device driver equipped with the COMEDI interface (see Section 5.2) has been exploited. This driver allows the access to an Ethernet board masking it as an acquisition board for analog and digital signals.

2.4 Real-time model generation

The dynamic models for the off-line simulation were developed in Dymola [1], a Modelica editor and compiler. In order to make as easy as possible the production of real-time models, it has been investigated the possibility of deriving the real-time models directly from the off-line models developed in Dymola. After having analyzed the features of Dymola, it has been found possible to use it also for the development and the compilation of real-time models. So, it is possible to both reuse the off line models by rapidly adapting them to the real-time simulation, and to build up some new model from scratch, with the advantage to use all the model libraries and the graphical instruments of such application.

2.5 Numerical integration issues

The features and the performances of the real-time numerical solvers available in Dymola have been analyzed in order to determine the most appropriate algorithm for the case study. It has been found that in order to obtain the maximum processing speed and to avoid the risk of a non convergent solution it is advisable to select the Inline Integration method applied to the Implicit Euler algorithm.

3 Real-time Modelica extension

3.1 Real-time components

A Modelica component has been developed that allows to transform any Modelica model into a model suitable for real-time simulation. This Modelica component, is a Modelica `block` that is called *ModRTAI*, and it can be used in the graphical user interface of Dymola as a normal building block of simulation models. ModRTAI uses a library of functions, called *LibRTAI*, which has been also developed within this project. LibRTAI has been entirely developed in the C language and contains the functions which allow the simulation code to access the RTAI and COMEDI libraries.
More precisely, by adding to the model development environment both a ModRTAI component and the LibRTAI library it is possible to perform the activities mentioned in the project requirements:

- Periodic execution at precise clock ticks of the simulation task, through access to the RTAI application programming interface;
- Management of the communication with the external world, through access to the interface of a COMEDI driver.

3.2 Porting a model to real-time

By means of the two modules which have been developed it is possible to obtain a Modelica model suitable for real-time simulation under Linux / RTAI. Given an off-line Modelica model, the ModRTAI block allows to select the signals which are input to the model (to be acquired from the external world), and the signals which are the output of the model (to be sent to the external world). The ModRTAI block also allows to set a parameter that states the frequency of the periodic task running the simulation. This parameter indicates how much time is left to the numerical solver to evaluate the transient related to one sampling interval. Only if such parameter is equal to the sampling interval, the simulation time should match the physical time elapsed since the instant in which the simulation itself was started; otherwise the simulation time develops slower than the physical time of a factor equal to the ratio between the

task scheduling period and the sampling step of the transient.
The Modelica model has to be compiled with Dymola under Linux, having selected the Inline Integration option and the Implicit Euler as integration algorithm. The functions library LibRTAI should be installed on the workstation onto which the model is compiled. The simulator which is generated in such way is called Dymosim, as a normal simulation binary file obtained by means of compiling with Dymola a Modelica model. The only difference is that the Dymosim binary obtained in such way is runnable in soft[1] real-time under Linux / RTAI. Figure 1 resumes the steps for porting a Dymola model into real-time.

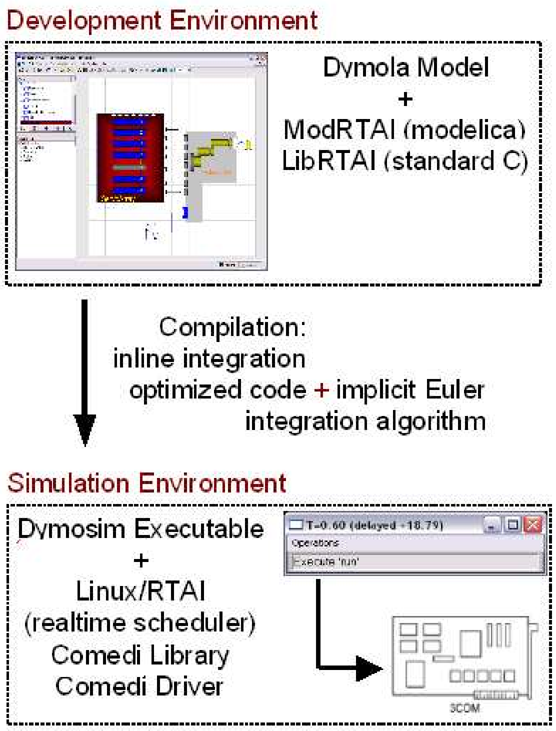


Figure 1: Development of a real-time Modelica model using Dymola

In order to allow the data exchange with external processes, the Dymosim simulator should be run on a workstation on which the COMEDI library is installed, and the drivers of the peripheral board onto which should flow the data signals in input and output to the simulator are installed too. By now, the simulator has been tested using an Ethernet communication board, which directs the signals to another workstation. The Ethernet board, equipped with drivers with a COMEDI interface, emulates a data acquisition board, as described in Section 5.2.
The entire procedure has been tested using a model of the Spider robotic arm with detailed descriptions of motors and transmissions (see Section 4), which was developed in a former research [14][13]. In this model the outputs are the seven motor positions, while in input the model receives the seven motor current set-points, and a digital signal controlling the brakes that in the home position block the motors axes. Experimental results about the performance of the Dymosim real-time simulator obtained with this model will be illustrated in Section 4.

[1]In Linux / RTAI a *hard* real-time process runs in the Linux kernel space and has more strict timing constraints, while a *soft* real-time process runs in the Linux user space and has more loose timing constraints. In Section 4 it is explained why the soft real-time solution has been preferred.

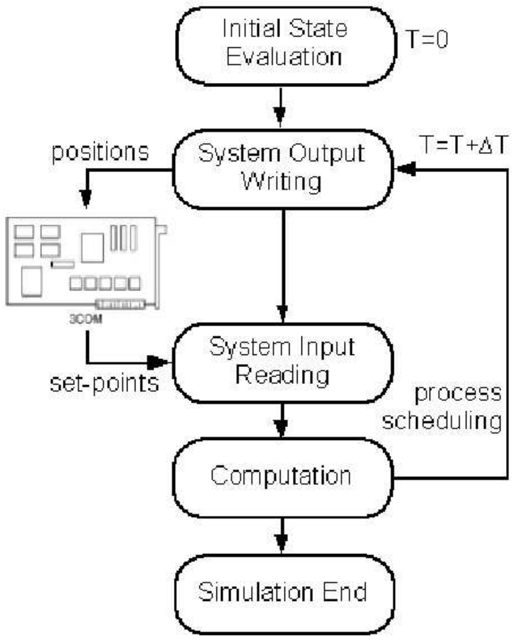


Figure 2: The simulation process execution cycle.

3.3 The simulator process

The real-time execution of the simulation code has the purpose of computing, at the chosen frequency, the solution that represents the evolution of the model state. The output of the model should be passed to an external control unit, which could be another workstation running the control procedure, or a dedicated hardware controller. The control system, on the basis of the received data, computes a control action that is sent to the workstation on which the real-time simulator is running. The simulator evaluates the new model state on the basis of the received input. This elementary cycle, illustrated in Figure 2, is executed till the end of the simulation.

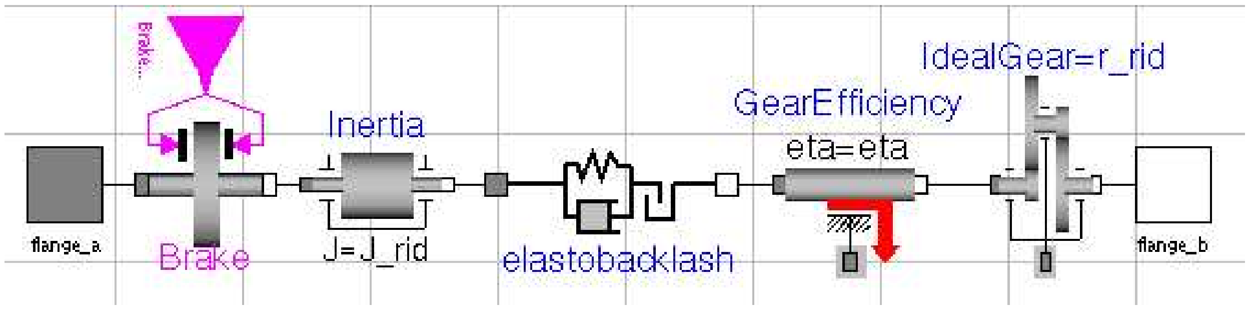


Figure 3: Modelica scheme of the transmission of the real-time Spider robot model.

At every wakeup of the periodic process, the simulator has to evaluate within a maximum time span[2] the next model status, and the corresponding outputs. This time span is equal to the scheduling period of the simulator process. Since the control loop should be closed on a digital (either hardware or software) regulator, it is mandatory that at any wakeup the simulator process is fed with inputs sampled at a constant frequency, and in the same way it yields as output some signal sampled at a constant frequency. This leads to the necessity to use a fixed step algorithm for the numerical integration, just as the Implicit Euler algorithm is.
As it has been said in Section 3, in order to synchronize with real-time a standard Dymosim process, and in order to exchange the model inputs and outputs with the external world, some calls to the external C functions implemented in the LibRTAI have been added in the ModRTAI block. These functions are:

- `RTAIGetInputSample(...)`
- `RTAIGetData(...)`
- `RTAIPutOutputSample(...)`

At every step, when the numerical solver tries to solve all the equations of the model, these three C functions are called in this order. The first and the third ones assign the external input variables to the vector of model inputs, and the vector of model outputs to the external output variables. By means of these two functions, the data are only put in some internal buffer of the LibRTAI module. Instead, the *real* data exchange with the external process / hardware is done by `RTAIGetData(...)`.

[2]There is no way to set a constraint on a Dymola numerical integrator forcing it to yield a result at every step within a physical maximum time. The missed deadline is checked by comparing the theoretical time in which the result should have been yield with the time in which the Dymosim process has actually released the control.

If `initial()` or `terminal()` are true in the Modelica code, `RTAIGetData(...)` respectively performs the initialization or the finalization of the the real-time process associated with Dymosim; otherwise such function performs the real data exchange and then it suspends the process.
During the initialization, the RTAI `rt_task_init(...)` function is called to initialize a new RTAI task, and the RTAI `rt_task_make_periodic(...)` function is called to make this task a periodic one. Other RTAI APIs are called to set the real-time scheduler, and to initialize and start the RTAI timer at the chosen frequency. During the finalization, the `rt_task_delete(...)` function is called, and the RTAI timer is stop. During a normal periodic call, the RTAI `rt_task_wait_period()` is called at the end of `RTAIGetData(...)` in order to suspend the periodic task.

4 Experimental results

As it has been said in Section 3.2, the proposed system has been tested on a detailed model of the Spider robotic arm [13][14] consisting of a total of more than 12,000 equations listed at compile time. The robot model includes a seven degrees of freedom mechanical chain, built with the old Modelica MultiBody library, and an array of seven servomechanisms, each of which featuring the dynamics of a brushless two-phase motor, an analog current controller, an elastic transmission with backlash and a brake on the motor axis. A detailed scheme of the elastic transmission used in the servomechanisms is shown in Figure 3.
The tests have been done running the simulation on a workstation, and a control action playback on another workstation. More precisely, the control action was not computed during the simulation, but it was a record of the control action of a simulated control system dur-

ing an off-line simulation of the same command given to the robot. The real-time model has been tested on a 3GHz Pentium IV workstation with 512Kb of 2^{nd} level cache. This processor ensures enough computational power to simulate the model with an integration step of $1ms$[3] and a wakeup period of $1.5ms$ for the corresponding scheduled process.
It has been experienced an average of 1% of faults, i.e. periods in which the simulator has not been able to evaluate the corresponding model transient. The graph in Figure 4 shows the CPU time not used by the simulator at each period, during the first 2 seconds of a simulation. A `move` command was simulated, and the command execution was started at $1.1s$, which explains why at this instant the free CPU time dramatically decreases.

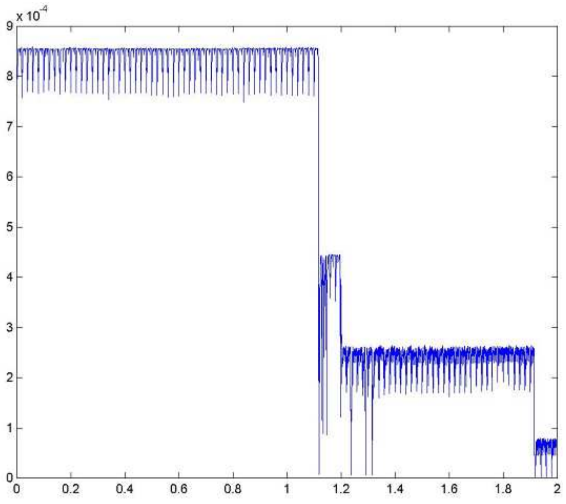


Figure 4: CPU idle time during simulation. Simulation time (in seconds) on x-axis; CPU idle time (in seconds) on y-axis.

This result proves that the model is still not ready to be simulated in a time equal to the physical time on the testing workstation, since the transient computation occupies half of the CPU time if the robot is still, but practically all the CPU time if the robot is moving. In order to reach the purpose to simulate with a scheduling period of $1ms$, it is mandatory to simplify the model, or to use a more powerful workstation.
The graph in Figure 5 shows the physical time span between two subsequent process wakeups in the same simulation as before. Figure 6 shows a detail of the graph in Figure 5. The maximum variation of the wakeup period is of 40%, while the average variation is of 1.34%. The picks of variation in the scheduling period are a consequence of the beginning of large transients, due to the motion start, but the continuous period variation during the movement and also before the beginning of the movement, are due to the soft real-time nature of the chosen scheduler.

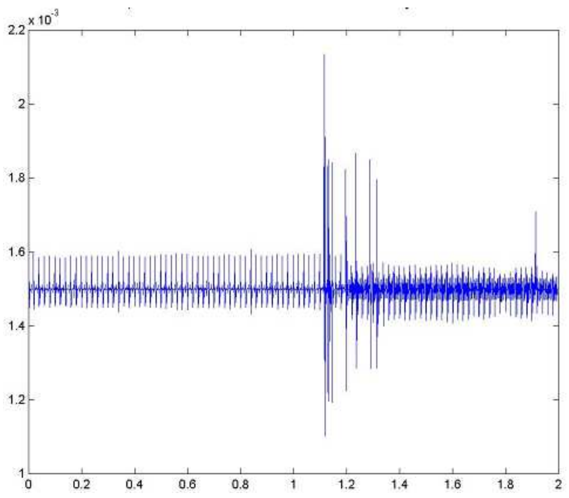


Figure 5: Dymosim task scheduling interval. Simulation time (in seconds) on x-axis; scheduling interval (in seconds) on y-axis.

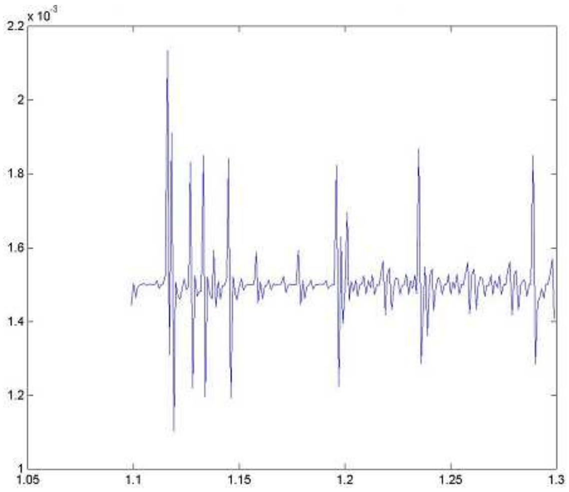


Figure 6: Dymosim task scheduling interval: detail. Simulation time (in seconds) on x-axis; scheduling interval (in seconds) on y-axis.

The choice of a soft real-time scheduler has been imposed by the nature of the executable binary code generated by Dymola. In fact, the Dymosim native code executes some operating system calls that do not exploit the RTAI API[4]. The OS calls of Dymosim impose a continual switch from a real-time context to a non-

[3]This is equal to the sampling frequency of a typical axis control cycle of an industrial robot controller.

[4]For example, all mathematical functions in the numerical integrator do not call the RTAI API.

realtime context. If the simulator is declared hard real-time at process startup, the continual context switch is from hard real-time to non-realtime, while if the simulator is declared soft real-time at process startup, the continual context switch is from soft real-time to non-realtime. The first kind of switch is much more time consuming than the second kind, and leads to worse performances.

5 Real-time control

5.1 The software control application

Within the same project framework, a real-time software control system [16] has been developed which emulates several functionalities of robot controllers. This control application can be easily adapted to interact with simulations of robotic arms driven at joint level. Thanks to the adoption of COMEDI drivers (which have standard interfaces, as stated in section Section 5.2), the control system can control without distinction a physical system, or a model based simulation of the system itself, provided that the two systems have the same number of input and output channels, disposed in the same order.
The software control system can be coupled with the real-time simulator, and each one of these two applications can be used as test bench when adding new features to the other one. So, the software control system can be used to test new and more refined robot models, and to analyze their behavior, if compared to the behavior of the corresponding real robot, while the real-time simulator can be used to test some innovating control solutions, without taking the risk of damaging the robot hardware.
The Linux / RTAI operating system has been chosen for the control application too, for the same reasons explained in Section 2.2. To support the design of the control application, the OROCOS (Open RObot COntrol Software) [9] framework has been chosen. The control application is by now capable of executing the position control in joint space for a six[5] degrees of freedom robot. The OROCOS control application can execute a standard control cycle, with signals exchanged in Real-Time with the controlled system; moreover, it can publish the variables internal to the controller and the signals received from the controlled process. Internal variables are published to non-realtime applications external to the controller, for reporting purposes. Also, the OROCOS control application can accept the robot motion commands from a program script, or from some external non-realtime application.

[5]Due to this limitation, the cross tests with the Spider model have been done blocking the seventh joint of the robot.

5.2 The closed-loop data acquisition

Both the control and the simulation applications should be able to transmit and to acquire signals on a hardware communication channel. In order to make any application unaware of the presence of hardware or software on the other side of the control loop it has been decided to implement COMEDI drivers for the communication boards. The COMEDI package has been chosen because it is an open-source product widely used in the field of automation. Indeed COMEDI provides a standard for drivers of DAQ (Digital AcQuisition boards) under Linux.
A COMEDI driver for 3COM 3C90x(B) [15] Ethernet boards and a COMEDI driver for COMAU BIT3 AT CARD [15] boards of the COMAU C3G-9000 controller have been developed. Both boards are accessible from real-time processes: the first one is used by now for the data exchange between the OROCOS control system and the Dymosim simulated process, while the second one (whose driver is still in a test phase) will be used for the data exchange with all robots supported by the COMAU C3G-9000 controller.

6 Conclusions and future work

A design concept and the related implementation software for obtaining real time simulation code from standard Modelica / Dymola models and related software has been presented. They permit to develop ready to run real time simulation code by fully exploiting the powerful libraries and tools that Modelica / Dymola make available for the model development phase. The real time simulation of a detailed model of a 7-DOF space arm has been afforded as a test bench for the software, and has proved its versatility and correctness.
With reference to the class of mechatronic systems models, additional work has to be spent to speed up model execution by refining or simplifying models of those phenomena that most affect the computational burden, like, for instance, non linear friction at low speed. Indeed, while the model exploited for the actual tests can be simulated in a time which is of the same order of magnitude of physical time on a high-end mono-processor system, models including friction

equations are much slower and cannot be proposed for the purposes of real-time simulations. An alternative approach would be to move on a multi-processor platform, provided that a Modelica compiler tool for parallel code generation is adopted.

Acknowledgments

The authors would like to thank the Italian Space Agency and the Ministry of Instruction, University, and Research for the support to this research, in relation to the I/R/217/02 ASI contract and the OASYS project.

References

[1] Dymola Multi-Engineering Modeling and Simulation [Online]. Available: `http://www.dynasim.se/`

[2] Real-Time Workshop: Generate optimized, portable, and customizable code from Simulink models [Online]. Available: `http://www.mathworks.com/products/rtw/`

[3] MBDyn - MultiBody Dynamics Software [Online]. Available: `http://www.aero.polimi.it/~mbdyn/`

[4] RTnet - Hard Real-Time Networking for Linux / RTAI [Online]. Available: `http://www.rts.uni-hannover.de/rtnet/`

[5] Attolico M, Masarati P. A Multibody User-Space Hard Real-Time Environment for the Simulation of Space Robots. In Proceedings of the 5^{th} Real-Time Linux Workshop 2003, Valencia, Spain, Real-Time Linux Foundation, November 9-11, 2003.

[6] dSPACE - Solutions for Control [Online]. Available: `http://www.dspaceinc.com/`

[7] Cloutier P, Mantegazza P, Papacharalambous S, Soanes I, Hughes S, Yaghmour K: DIAPM-RTAI Position Paper. In Proceedings of the 2^{nd} Real-Time Linux Workshop 2000, Orlando, Florida, USA, Real-Time Linux Foundation, 27-30 November, 2000.

[8] COMEDI - The Linux Control and Measurment Device Interface [Online]. Available: `http://www.comedi.org/`

[9] OROCOS - Open Robot Control Software [Online]. Available: `http://www.orocos.org/`

[10] Aronsson P, Fritzson P. Parallel Code Generation in MathModelica / An Object Oriented Component Based Simulation Environment (Conference paper). In Proceedings of the Parallel / High Performance Object-Oriented Scientific Computing Workshop, POOSC01 at OOPSLA01, Tampa Bay, Florida, USA, 14-18 October, 2001.

[11] Aronsson P, Fritzson P. Multiprocessor Scheduling of Simulation Code from Modelica Models (Conference paper) in Proceedings of the 2^{nd} International Modelica Conference, DLR, Oberpfaffenhofen, Germany, Modelica Association, 18-19 March 2002.

[12] Nyström K, Aronsson P, Fritzson P. GridModelica - A Modeling and Simulation Framework for the Grid (Conference paper). In Proceedings of the 45^{th} Conference on Simulation and Modelling, Copenhagen, Danemark, Scandinavian Simulation Society, 23-24 September 2004.

[13] Ferretti G, Gritti M, Magnani G, Rocco P, Viganò L. Object-Oriented Modeling of a Space Robotic Manipulator. In Proceedings of the 8^{th} ESA Workshop on Advanced Space Technologies for Robotics and Automation 2004, Noordwijk, The Netherlands, ESTEC, 2-4 November 2004.

[14] Viganò L. Modellistica del Braccio Robotico Europa con Analisi del Controllo nello Spazio Operativo [in Italian]. Milano, Italy: Master's Thesis, Politecnico di Milano, 2003.

[15] Minazzi P. Sviluppo di Driver COMEDI in Ambiente Linux / RTAI [in Italian]. Milano, Italy: Master's Thesis, Politecnico di Milano, 2004.

[16] Cappellini S, Consonni A. Progetto e Realizzazione di un'Applicazione Real-Time a Componenti per il Controllo di Robot Manipolatori [in Italian]. Milano, Italy: Master's Thesis, Politecnico di Milano, 2004.

SCICOS: a general purpose modeling and simulation environment

M. Najafi[‡] , S. Furic[*], R. Nikoukhah [†‡]

Abstract

Partial support for Modelica is now provided by the general purpose dynamical system simulator Scicos. In particular it is now possible to use component models in Scicos diagrams where the dynamics of the component has been described in Modelica. This paper presents this new extension of Scicos.

KEYWORDS: Dynamic system simulation; Simulation software; Component level modeling; Scilab; Modelica

1 Introduction

Scicos is a Scilab[1] toolbox for modeling and simulation of dynamical systems [1, 2]. Scicos provides a hierarchical graphical editor for the construction of complex dynamical systems, a simulator and a code generator. For many applications, the Scilab/Scicos environment provides a free open-source alternative to Matlab/Simulink and MatrixX.

Very general dynamical systems, including hybrid systems, can be modeled in Scicos [3, 4, 5, 6]. A typical Scicos diagram is presented in Fig. 1. This diagram is used to evaluate the performance of an observer by simulation; the simulation results are given in Fig. 2.

The model of Fig. 1 is composed of ”explicit” blocks, i.e., block with explicitly identified inputs and outputs. Modeling with such blocks is called system level modeling. Component level modeling, on the other hand, allows the use of ”implicit” blocks which are blocks with port connections which a-priori are not labeled as inputs or outputs [7]. Implicit blocks are essential for constructing models which include physical components such as resistors, capacitors, etc., in electricity, or pipes, nozzles, etc., in hydraulics. They are also useful in many other areas such as mechanics and thermodynamics. In Modelica community implicit blocks are called acausal [13].

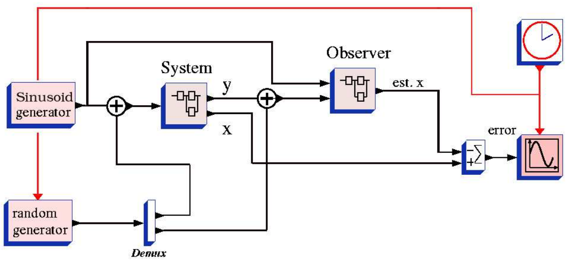


Figure 1: A system modeled in Scicos.

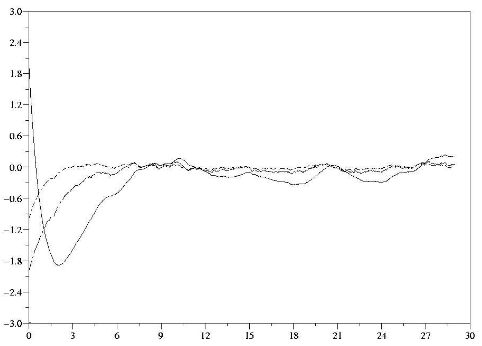


Figure 2: Simulation result of model of Fig. 1 in Scicos.

Contrary to explicit blocks, implicit blocks cannot be modeled as black box objects. The equations realizing the behavior of an implicit block must be available to the compiler for system reduction and code generation. To describe the behavior of these blocks in Scicos, the Modelica language has been adopted.

[*]IMAGINE (`www.amesim.com`)

[†]Corresponding author (ramine.nikoukhah@inria.fr)

[‡]M. Najafi, R. Nikoukhah are with INRIA-Rocquencourt, Domaine de Voluceau, 78153 Le Chesnay Cedex, France

[1]Scilab is a free open-source software for scientific computation, see www.scilab.org and www.scicos.org.

2 Modelica and Scicos

Even though Modelica is a rich language having the capacity to handle continuous-time and discrete-time behaviors, for the start, we are mainly using Modelica and implicit blocks to model continuous-time dynamics; only minimal support is provided for discrete-time behavior. The discrete-time behavior, in the Scicos environment, is provided via explicit blocks.

The addition of implicit blocks has been done without changing significantly Scicos formalism. Even though implicit blocks can be used anywhere inside a Scicos diagram, they are grouped and replaced with a single block in a precompilation phase [7]. The mechanism, which can be compared to the way an `amesim`[2] or `Dymola`[3] model is integrated in Simulink, is completely transparent to the user.

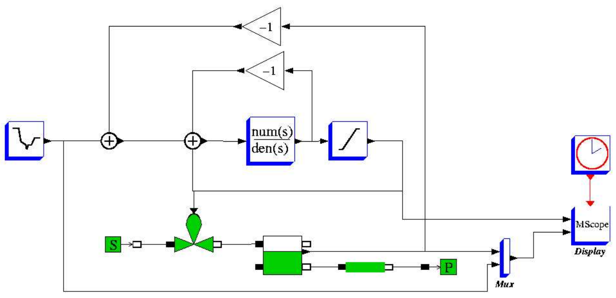


Figure 3: Scicos diagram containing both types of blocks.

Consider for example the Scicos diagram in Fig. 3. Here we have a fluid level control system. To model this system in a natural way, a hydraulic source, a regulated valve, a container, a tube, and a well have been used. The container has a built-in level sensor which makes the interface with the explicit part of the system, similarly the valve is regulated through an input signal from the explicit part of model. The controller and the display mechanism have been implemented using explicit blocks and the blocks in gray are implicit blocks that have been developed in Modelica language.

2.1 Scicos architecture

To illustrate our method, a simple flowchart given in Fig. 4 shows how Scicos and Modelica interact. A designer can select blocks from either standard or implicit toolboxes. Blocks in implicit toolbox have been developed using Modelica language. As shown in the flowchart, if the model contains an implicit block, after a series of automatic preprocessing steps, implicit part of model is abstracted into a standard block with explicit input/outputs; the resulting model can then be simulated by Scicos [7].

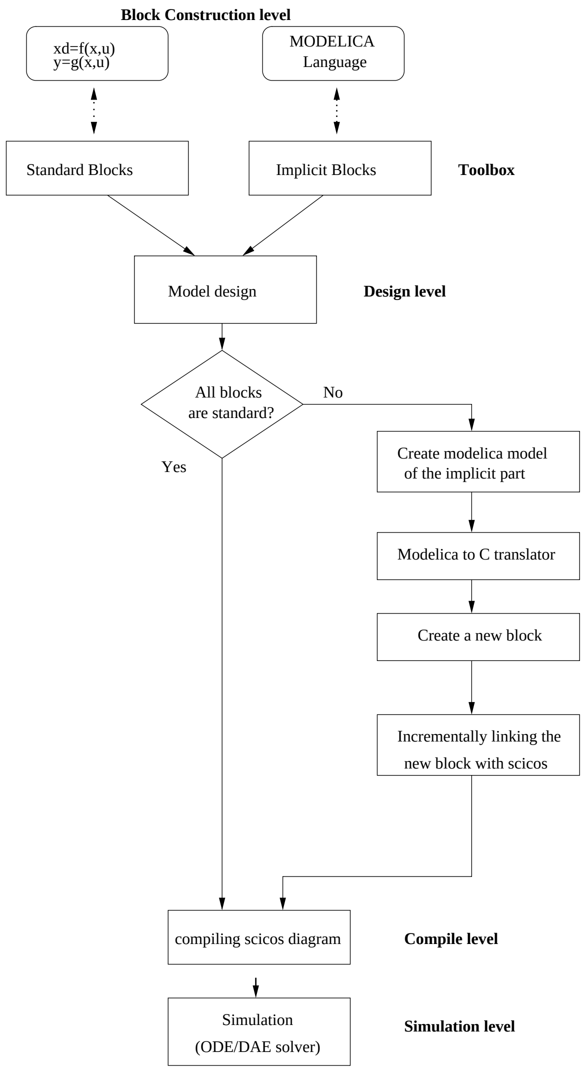


Figure 4: system flowchart

2.2 Available implicit toolboxes

To be able to use implicit blocks in addition to explicit ones in Scicos, several new features have been added to Scicos. So far, only two palettes with implicit blocks are available for testing purposes: the electrical and the thermodynamics palettes. The thermo-hydraulic toolbox and available blocks are shown in Fig. 5.

[2]www.amesim.com

[3]www.dymola.com

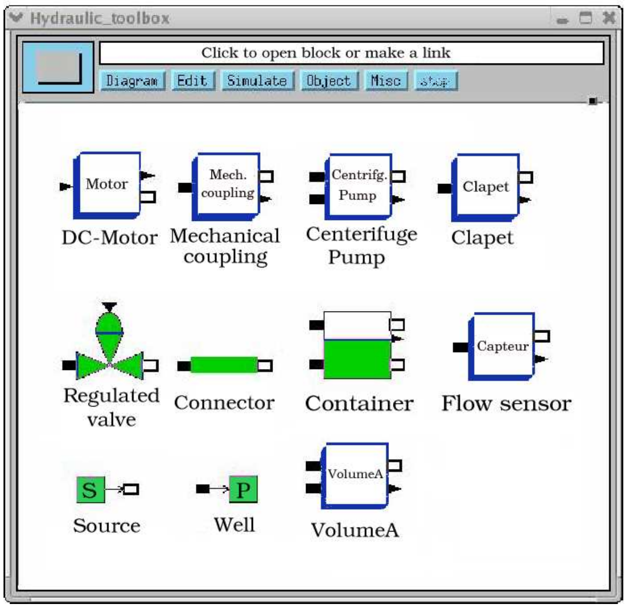


Figure 5: Thematic toolbox

2.3 New link and port type

Implicit blocks or components are interfaced via special links associated with physical quantities such as current or voltage in electronics, or, flow or pressure in hydraulics. It would be meaningless for a link representing a voltage to be connected to another link representing the output value of a PID controller. To distinguish between these two, two different link types have been defined: explicit and implicit links that interconnect explicit and implicit ports respectively. In Fig. 6 we have a hydraulic container which has four implicit ports (marked IP) representing liquid outlets and an explicit port (marked EO) representing a liquid level sensor output.

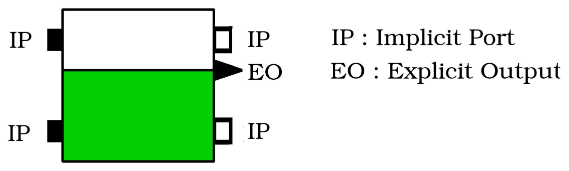


Figure 6: An implicit block can have implicit and explicit ports.

2.4 Compiling a mixed diagram

To compile and simulate a model containing implicit blocks, Scicos groups all implicit blocks into a single block having explicit inputs and outputs. Then it generates a Modelica code expressing the behavior of the new block and save it in a temporary file. This file is then processed by `modelicac` [4] which translates this Modelica code into a C-code describing the behavior of the new Scicos block. Once the C-code is compiled and incrementally linked in Scilab, Scicos sees this new block as a standard explicit block; see Fig. 7. At the end of this procedure, the model is no longer implicit because all blocks are standard explicit blocks, so the model can be compiled and simulated as usual. It should be noted that this procedure is completely transparent to the user [7].

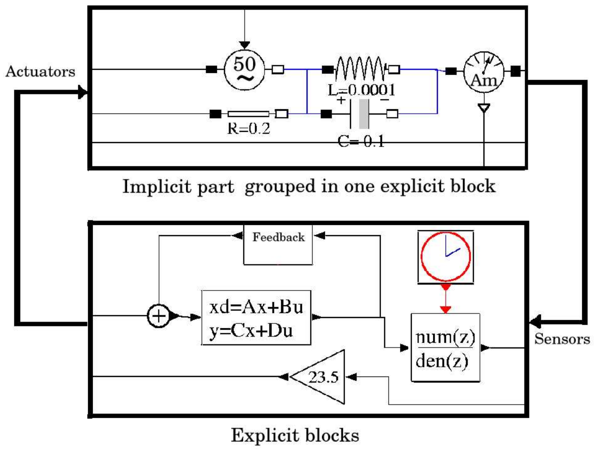


Figure 7: In a precompilation phase, all implicit blocks are grouped to form an explicit block.

2.5 New numerical Solver

Most of the time, after generating C code for implicit part of the model, we end up with a set of Differential-Algebraic Equations (DAE) that no longer can be integrated via ordinary differential equation solvers. It is for this reason that the DAE solver DASKR [8, 9, 10, 11, 12] has been incorporated into Scicos.

3 Modelicac, a Modelica compiler

Modelicac (acronym of "Modelica compiler") is a compiler for the subset of the Modelica language we felt necessary to handle in order to cover the needs of simulating hybrid models under Scicos. Modelicac is an external tool, i.e. it is independent of Scilab, so

[4] A Modelica compiler and C code generator written in `Objective Caml` and included in the Scilab distribution.

one may use it like an ordinary compiler e.g., like a C compiler. By default, modelicac comes with a module that generates C code for the Scilab target. However, since modelicac is free and open source, it is possible to develop a code generator for another target.

3.1 Modelicac development

Modelicac has been developed in Objective Caml[5] which is a functional programming language developed at INRIA since 1985. This language is distributed with two compiler-development tools (i.e., Ocamllex and Ocamlyacc) that offer some nice facilities to build compilers. Furthermore the Objective Caml compiler is free and open source, that's why we adopted it to develop modelicac [16].

3.2 Modelica compilation using modelicac

Modelicac is invoked for two purposes: compiling basic models from libraries and generating code for the target simulation environment. To fulfill the first task, like generating an object file with a C compiler, modelicac is invoked with the appropriate options from the command line to generate an object file with ”*.moc” extension to be used later. The second task of modelicac is compiling the ”main” Modelica model (here provided by Scicos) and generating a code for the target (here, a C code). In this phase instead of generating an object file, modelicac performs several simplification steps to generate a code as compact as possible. In Fig. 8 a simple flowchart shows how modelicac generates a C file from modelica model of a Scicos diagram.

3.3 Supported Modelica subset

As said previously, the current version of modelicac (1.x.x) does not handle the full set of Modelica language constructs. It actually allows only the description of physical models at ”equation” level. A physical model is built as the aggregation of sub-models or basic types with constraints between variables, and explicit event declarations (”When”). Currently modelicac has the following main limitations:

- Only ”Real” data type is supported.
- Inheritance is not currently supported.
- ”Algorithm” is not supported but it can be defined as an external C function.

[5]caml.inria.fr

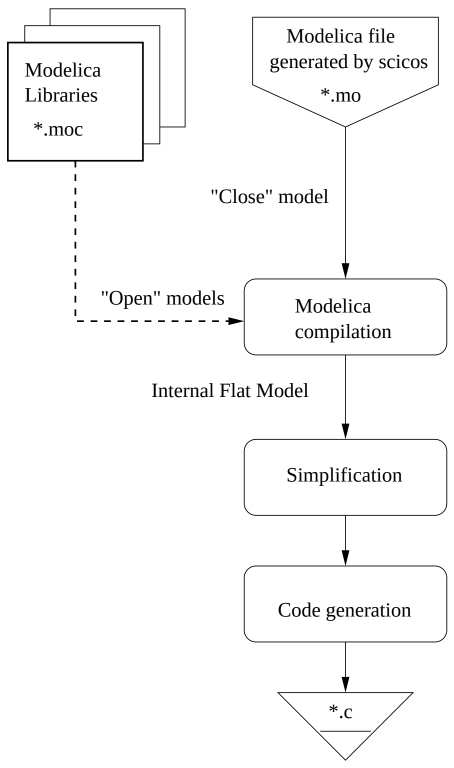


Figure 8: Modelicac translation flowchart

3.3.1 Modelica source files

Modelica source files must contain only one class declaration, introduced either by the ”class” keyword or by the ”function” keyword. So a Modelica source file may define one of the following things:

- An ”open” model is a model with free variables. There are more variables than equations (e.g. the model of a resistor in electrical library). The open models are introduced by the ”class” keyword,
- A ”close” model is a model with equal number of variables and constraints. It is also introduced by the ”class” keyword,
- An external function, introduced by the ”function” keyword.

Of course, only closed models can be simulated. In order to find classes in the host file system hierarchy, it is required that the name of the file be the same as the name of the enclosed class. To compile the ”close” model modelicac searches the libraries used in the current compilation directory and also in user-defined directories.

The following source code describes a simple resistor enclosed in a ”Resistor.mo” file:

```
class Resistor
  Pin p, n;
  parameter Real R "Resistance";
```

```
equation
  R*p.i = p.v - n.v;
  p.i = -n.i;
end Resistor;
```

An instance of ”Resistor” has two ”connectors” (”p” and ”n”), that have their own potentials and flows variables (here, the voltage and the current, respectively). A resistor has also a resistance parameter imposed by the component through the first equation. The second equation simply states that the current that flows in the resistor through ”p” is equal to the current that flows out through ”n”.

3.4 Model simplification

The following tasks are fulfilled by modelicac to simplify and generate a C source file from a Modelica source code and library object code files:

• Obtaining a flat model by replacing an aggregation of sub-models by the set of all their variables and equations merged together and replacing connection equations by ordinary equations. Symbolic manipulations in modelicac are performed using classical acyclic graph manipulation techniques

• Simplification of trivial or unnecessary equations using symbolic manipulations e.g. in the following system

$$\begin{cases} \cos(x)+\sin(y)=0 \\ \cos(x)-\sin(y)=0 \\ z-x-y=0 \\ f(x,y,z,v)=0 \end{cases}$$

the first two equations are fully non-linear and only the numerical solver can solve the system for x and y. But the third equation is trivial and z can be obtained in terms of x and y, so in the rest of equations z is replaced by $x+y$. Most of the variables used to connect Modelica components (”connection variables”), are eliminated in this way.

• Causality analysis, i.e. computation of system’s Jacobian matrix. It will be explained further.

3.4.1 Causality analysis

Causality analysis performs a few operations in order to find the so-called ”strongly connected components” of a system of equations viewed as a directed bipartite graph [13]:

1. Constructing a bipartite graph whose nodes on the left represent variables in the system and whose nodes on the right represent constraints between variables (i.e., equations). There is an edge between a left-side node and a right-side node if and only if the variable represented by the left-side node appears in the equation represented by the right-side node.

2. Finding a coupling (using the Ford and Fulkerson method for instance)

3. Giving the edges an orientation depending on the results of the previous step. Edges that link two coupled nodes are all oriented in a given direction (either left-to-right or right-to-left) and the other ones in the opposite direction.

4. Finding strongly connected components in the resulting oriented graph (using Tarjan’s algorithm for instance)

5. Sorting the resulting nodes in a topological order.

Each strongly connected component represents a subsystem of the whole system and it is now possible to perform symbolic simplification steps in order to reduce the number of variables in the system.

3.4.2 Modelicac simplification strategy

Symbolic simplifications typically involve variants of the Gauss method (to solve linear systems) and simple symbolic simplification methods based on a set of predetermined patterns (for efficiency reasons) to try to solve the remaining equations. In modelicac we focused on the second class of simplification methods. The problem when trying to solve a set of non-linear equations is to determine a coupling in the bipartite graph described above that triggers as many simplifications as possible. So the Ford and Fulkerson (or equivalent) method is not enough for our purposes: instead of taking the first encountered coupling, we want in addition that the coupling satisfy a given criterion (e.g. maximizing the potential number of simplifications in the system). Hence the use of a variant of the Hungarian method which can be seen in modelicac as a method for finding a coupling based on an additional constraint called the ”satisfaction”. Practically, that is done in modelicac by associating a set of pairs (variable, weight) with each equation: given an equation, each weight indicates whether the equation is ”easy” to solve with respect to its associated variable or not. For instance, if an equation contains only one variable, the weight associated with that variable is low whereas the weight associated with any other variable is infinite. Since modelicac associates low weights with variables that appear in linear systems, the Hungarian method ”discovers” linear systems by itself and symbolic substitution techniques, when applied to those linear systems, achieve the same effect as Gaussian elimination. Even though we did not consider the Gaussian elimination algorithms in modelicac, we got good results.

3.5 A complete (yet simple) example

First, we present the Modelica source code of a few electrical models from electrical library and then show how to use these models to construct and compile elaborated electrical models with modelicac.

3.5.1 Connectors

In Scicos libraries ”connectors” are the most basic open models. Each particular domain (e.g., electrical, hydraulic, etc.) has a its own connectors that connect two or more models and exchange quantities. The are two connector types:
• Internal connectors, that allow connection of two Modelica components, such as ”p” and ”n” pins used in electrical resistor model.

```
class Pin
  Real v;
  flow Real i;
end Pin;
```

• External connectors, that allow communication of a Modelica component with an external environment (Explicit part of model in Scicos environment, for instance). Instances of ”InPutPort” and ”OutPutPort” are examples of these connectros types

```
class InPutPort
  input Real vi;
end InPutPort;

class OutPort
  output Real v;
end OutPort;
```

These types of connectors are used in sensor and actuator blocks that can be seen in Fig. 3 and 9.

3.5.2 Electrical component classes

These models include the ideal resistor, capacitor, inductor, sinusoidal voltage source and ground.

```
class Ground "Ground"
  Pin p;
equation
  p.v = 0;
end Ground;

class Capacitor
  Pin p, n;
  Real v;
  Real i;
  parameter Real C "Capacitance";
equation
  C*der(v) = i;
  v = p.v - n.v;
  0 = p.i + n.i;
  i = p.i;
end Capacitor;

class Inductor "Ideal electrical inductor"
  Pin p, n;
  Real v;
  Real i;
  parameter Real L "Inductance";
equation
  v = L*der(i);
  v = p.v - n.v;
  0 = p.i + n.i;
  i = p.i;
end Inductor;

class VsourceAC "Sin-wave voltage source"
  Pin p, n;
  Real v;
  Real i;
  parameter Real VA=220 "Amplitude";
  parameter Real f=50 "Frequency";
  parameter Real PI=3.1415926 "PI";
equation
  v = VA*2*PI*f*time;
  v = p.v - n.v;
  0 = p.i + n.i;
  i = p.i;
end VsourceAC;
```

3.5.3 ”Main class”

In order to perform the simulation of an electrical circuit one normally has to describe the circuit using Modelica by defining the components involved (i.e. giving their names and the value of their parameters) and the connections to establish. Then, modelicac should be invoked with the appropriate options and arguments. This task is done by Scicos, provided that the appropriate library exist in Scicos;
In fact it is not necessary to write any Modelica code to build a circuit: one can assemble components using the Scicos editor and then Scicos automatically builds the Modelica source code from the graphical specifica-

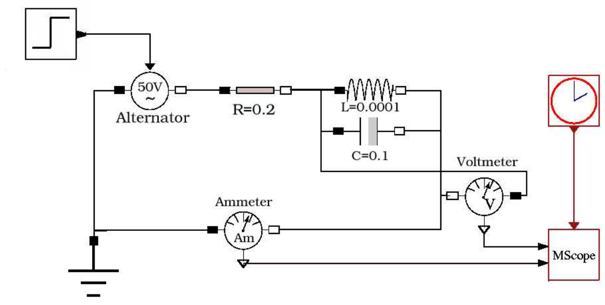


Figure 9: An electrical circuit modeled in Scicos.

tion and invokes modelicac to convert Modelica code into C code. In Fig. 9 there is a model of an electrical circuit modeled in Scicos. Here is its Modelica class automatically generated by Scicos:

```
class imppart_rlc
  parameter Real P1=0.0001;
  parameter Real P2=0.1;
  parameter Real P3=25.0;
  parameter Real P4=0.2;
  parameter Real P5=50.0;
  Inductor      B1(L=P1);
  Capacitor     B2(C=P2, v(start=P3));
  Ground        B3;
  VoltageSensor         B4;
  CurrentSensor         B5;
  Resistor      B6(R=P4);
  VVsourceAC    B7(f=P5);
  OutPutPort    B8;
  OutPutPort    B9;
  InPutPort     B10;
equation
  connect (B5.p,B3.p);
  connect (B7.p,B3.p);
  connect (B2.p,B1.p);
  connect (B4.p,B1.p);
  connect (B6.n,B1.p);
  connect (B2.n,B1.n);
  connect (B4.n,B1.n);
  connect (B5.n,B1.n);
  connect (B7.n,B6.p);
  B4.v = B8.vi;
  B5.i = B9.vi;
  B10.vo = B7.VA;
end imppart_rlc;
```

For this model modelicac generates a C code. This C code is incrementally linked with Scicos to be used as a standard block.

```
/*
number of discrete variables = 0
number of variables = 3
number of inputs = 1
number of outputs = 2
number of modes = 0
number of zero-crossings = 0
I/O direct dependency = false
*/

#include <math.h>
#include <scicos/scicos_block.h>

void rlc(scicos_block *block, int flag)
{
   double *rpar = block->rpar;
   double *z = block->z;
   double *x = block->x;
   double *xd = block->xd;
   double **y = block->outptr;
   double **u = block->inptr;
   double *g = block->g;
   double *res = block->res;
   int *jroot = block->jroot;
   int *mode = block->mode;
   int nevprt = block->nevprt;
   int property[3];
    /* Intermediate variables */
   double v0,v1;

   if (flag == 0) {
     res[0] = x[1]-xd[0]*rpar[0];
     res[1] = x[0]+xd[1]*rpar[1]-x[2];
     v1=get_scicos_time();
     res[2] = x[1]+x[2]*rpar[3]+sin(6.28318530718*v1*rpar[4])*u[0][0];
   } else if (flag == 1) {
       if (get_phase_simulation() == 1) {
          y[0][0] = x[1]; /* main.B8.vo */
          y[1][0] = -x[2]; /* main.B9.vo */
       } else {
          y[0][0] = x[1]; /* main.B8.vo */
          y[1][0] = -x[2]; /* main.B9.vo */
       }
   } else if (flag == 2 && nevprt < 0) {
   } else if (flag == 4) {
       x[0] = 0.0;     /* main.B1.i */
       x[1] = rpar[2]; /* main.B2.v */
       x[2] = 0.0;     /* main.B6.p.i */
       Set_Jacobian_flag(1);
   } else if (flag == 6) {
   } else if (flag == 7) {
       property[0] = 1;   /* main.B1.i (state variable) */
       property[1] = 1;   /* main.B2.v (state variable) */
       property[2] = -1;  /* main.B6.p.i (algebraic variable) */
       set_pointer_xproperty(property);
   } else if (flag == 9) {
   } else if (flag == 10) {
       v0 = Get_Jacobian_parameter();
       res[0] = -rpar[0]*v0;
       res[1] = 1.0;
       res[2] = 0.0;
       res[3] = 1.0;
       res[4] = rpar[1]*v0;
       res[5] = 1.0;
       res[6] = 0.0;
       res[7] = -1.0;
       res[8] = rpar[3];
       res[9] = 0.0;
       res[10] = 0.0;
       res[11] = sin(6.28318530718*get_scicos_time()*rpar[4])
       res[12] = 0.0;
       res[13] = 0.0;
       res[14] = 1.0;
       res[15] = 0.0;
       res[16] = 0.0;
       res[17] = -1.0;
       res[18] = 0.0;
       res[19] = 0.0;
       set_block_error(0);
   }
        return;
}
```

Conclusion

The use of Modelica in Scicos provides a versatile modeling and simulation tool.

References

[1] C. Bunks, J. P. Chancelier, F. Delebecque, C. Gomez(ed.), M. Goursat, R. Nikoukhah and

S. Steer, *Engineering and Scientific Computing with Scilab,* Birkhauser, 1999.

[2] J. P. Chancelier, F. Delebecque, C. Gomez, M. Goursat, R. Nikoukhah, and S. Steer, *Introduction à Scilab*, Springer-Verlag, 2002.

[3] R. Nikoukhah and S. Steer,*Scicos a dynamic system builder and simulator*, IEEE INTERNATIONAL CONFERENCE ON CACSD, Dearborn, Michigan, 1996.

[4] R. Nikoukhah and S. Steer, *Hybrid systems: modeling and simulation*, COSY: MATHEMATICAL MODELLING OF COMPLEX SYSTEM, Lund, Sweden, Sept. 1996.

[5] A. Benveniste, *Compositional and Uniform Modeling of Hybrid Systems,* IEEE Trans. Automat. Control, AC-43, 1998.

[6] R. Nikoukhah and S. Steer, *Scicos: A Dynamic System Builder and Simulator, User's Guide - Version 1.0,* INRIA Technical Report, RT-0207, June 1997.

[7] M. Najafi, A.Azil, and R. Nikoukhah, *Extending scicos from system to component level simulation*, ESMc2004 international conference, Paris, France, October 2004.

[8] K. E. Brenan, S. L. Campbell, and L. R. Petzold, *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations,* SIAM publication, Philadelphia, 1996.

[9] A. C. Hindmarsh, ”LSODE and LSODI, Two New Initial Value Ordinary Differential Equation Solvers”, *ACM-Signum Newsletter*, Vol. 15, 1980, pp. 10–11.

[10] L. R. Petzold, ”Automatic selection of methods for solving stiff and nonstiff systems of ordinary differential equations”, *SIAM J. Sci. Stat. Comput.*, No. 4, 1983.

[11] L. R. Petzold. ”A Description of DASSL: A Differential/Algebraic System Solver”, *In Proceedings of the 10th IMACS World Congress*, Montreal, 1982, pp. 8-13.

[12] P. N. Brown, A. C. Hindmarsh, and L. R. Petzold, *Consistent Initial Condition Calculation for Differential-Algebraic Systems,* SIAM J. SCI. COMP., NO. 19, 1998.

[13] P. FRITZSON,*Principles of Object-Oriented Modeling and Simulation with Modelica 2.1,* WILEY-IEEE PRESS, 2004

[14] S.E. MATTSSON, H. ELMQVIST, M. OTTER, AND H. OLSSON, ”INITIALIZATION OF HYBRID DIFFERENTIAL-ALGEBRAIC EQUATIONS IN MODELICA 2.0”, *2nd Inter. Modelica Conference 2002*, DYNASIM AB, SWEDEN AND DLR OBERPFAFFENHOFEN, GERMANY, 2002, PP. 9–15.

[15] R. NIKOUKHAH AND S. STEER, ”CONDITIONING IN HYBRID SYSTEM FORMALISM”, *International Conference Automation of Mixed Processes:A D P M Hybrid Dynamic Systems*, DORTMUND, GERMANY, 2000.

[16] WEIS, PIERRE , LEROY, XAVIER, *Le Langage CAML*, 2ND ED , DUNOD PRESS, 1999

Model Validation and the Modelica Language

Dr. Richard Dorling
Advanced Dynamic Systems
Datum House, Commerce Road, Lynch Wood, Peterborough, PE2 6LR, UK

Abstract

Model validation is a crucial aspect of the development of any dynamic system that uses computer aided engineering (CAE).

The ease of applying model validation techniques is dependent on the structure of the model, and this is often dependent on the CAE tool used.

The Modelica language is both well-structured, and independent of any CAE tool. As such it exhibits many features that make it ideal in the application of model validation techniques.

This paper considers various aspects of the Modelica language and how they relate to the implementation of a model validation tool.

The validation of a vehicle model is presented as an example of how the features of the Modelica language are used in the validation process.

Finally, future developments of the Modelica language that would enhance the performance of any model validation tool are identified.

1 Introduction

Model validation should be an essential ingredient in any dynamic system development that uses Computer Aided Engineering (CAE) methods.

Comprehensive checks against test data should be made for even the most rudimentary model in order to identify any errors and invalid assumptions in the model. These should then be corrected in order to gain sufficient confidence in the CAE results.

The term ‘model verification’ is used to describe the process by which the behaviour of the model is checked against test data.

The term ‘model validation’ is used to describe the process by which a model, and/or a real system, is corrected so that the model and system’s performance match.

Model verification is thus a prerequisite for, and an essential part of model validation.

The problems with implementing model validation stem from the limitations of existing model verification techniques. These revolve around the comparison of simulation results with test data.

1.1 Existing approach to model verification

The comparison of simulation results against test data can be carried out in two ways: either using time histories; or using frequency responses/statistical data.

The comparison of time histories is notoriously difficult due to the following drawbacks:

1. All inputs to the model must be known.
2. Any errors in a model get magnified during the simulation since they are integrated to generate future time histories.
3. Any error space, calculated by taking the difference between the real and virtual results, will not normally have a single minima.

The above drawbacks in model verification mean that any attempt at model validation must, at best, utilise slow global optimisation tools. The requirement for assumptions to be made for unknown inputs will also have a detrimental impact on the model validation process.

Although the comparison of frequency responses and statistical data generally overcome these drawbacks, both methods suffer from two further shortcomings:

1. Lack of resolution.
2. Concealment of non-linear effects by statistical processing or Fourier transformation.

These particular shortcomings result in poor model verification and therefore subsequent attempts at model validation will not characterise the complex behaviour of a dynamic system. This detailed information is vital when analysing system performance.

1.2 Example, a spring-mass system

Consider a mass, on a non-linear spring and damper, subject to a disturbance input at the free end as shown in figure 1. This system has a very simple mathematic representation, and yet highlights all the limitations of current model verification techniques and their impact on model validation.

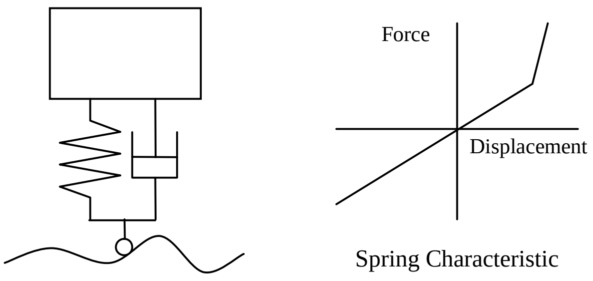


Figure 1: Example dynamic system

For the purpose of this example it will be assumed that the only measurement made during a test of the real system is that of the acceleration of the mass.

Considering the use of time history methods, and assuming the input to the system is known, it can immediately be shown that the error space between the measured and simulated accelerations does not have a single minima when the mass parameter is varied (figure 2).

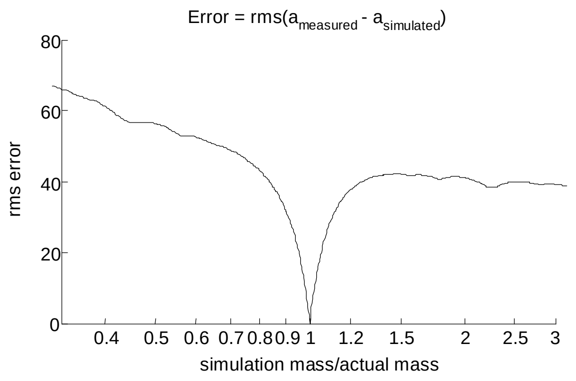


Figure 2: Verification error against mass

Considering frequency response methods, it can be seen that the frequency response of the system does not clearly represent the non-linearity of the spring characteristic, since it is lost within the measurement noise (figure 3).

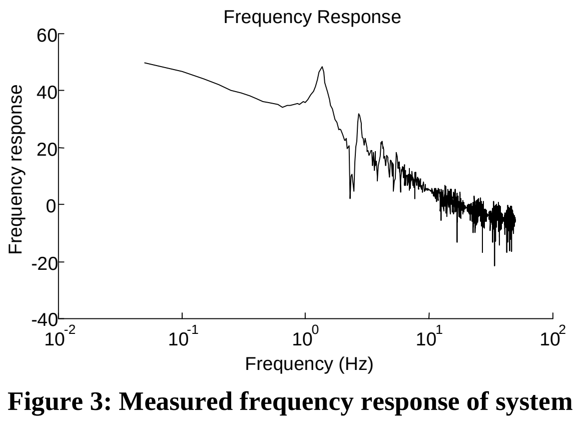


Figure 3: Measured frequency response of system

The more complex a system, the more problematic the limitations associated with current model verification techniques become.

These limitations mean that implementing a model validation technique for a complex dynamic system is almost impossible.

In order to develop a proper method for model validation, a robust model verification technique must first be developed that does not exhibit any of the drawbacks described above.

1.3 A new approach to model verification

The limitations of current model verification techniques all stem from the fact that it is the results of a model simulation that are compared to test data, rather than the model itself.

Any new approach to model verification must therefore compare the model itself to test data rather than the outputs of any analysis. This requires that the model must be independent of the analysis carried out on that model.

In order to achieve this, the model must be driven directly by the test data, and not by a set of inputs predetermined by the formulation of the model.

It is always possible to reformulate any model so that it can be driven by any arbitrary data, as long as there is sufficient data measured about the real system; it is just a matter of matching the number of equations to the number of pieces of information required to solve those equations.

When model verification is approached in this way, the requirement to know all inputs to the model is immediately removed. Instead, these inputs are derived as part of the model verification process.

In addition, the test data to be used can be differentiated and integrated prior to any analysis. As a result, errors do not get magnified by integration within the model and error spaces become well-behaved, exhibiting a single minima.

Finally as this approach is not statistical in nature, and does not use Fourier transforms, non-linearities are preserved, as is the resolution of the data.

The Modelica language has several features that are ideally suited to implementing a model verification technique based on the approach outlined above. Firstly, the equations of the model can be rearranged and manipulated as required. Secondly, the Modelica language includes only information about the model itself and not the analysis of that model.

1.4 A new approach to model validation

The model verification approach outlined in section 1.3 gives a measure of how well the model matches the physical system. This measure can be thought of as determining the 'error' between the model and the physical system, and as stated above this measure is 'well-behaved' exhibiting only a single minima at the point where the behaviour of the model and physical system agree.

The aim of any model validation process is to ensure that the behaviour of the model of a system matches that of its physical counterpart. In this approach, this is achieved by minimising the above error to an acceptable level.

In order to completely validate a model it is necessary to address two distinct issues: parameter identification and model structure development.

Parameter identification is concerned with tuning the parameters of the model to minimise the error between the model and physical system, whereas model structure development examines whether changes to the components or equations of the model would reduce this error.

By using the model verification approach discussed above, in conjunction with a fully parameterised model that allows replacement and updating of its individual components, it is possible to implement both parameter identification and model structure development techniques.

The Modelica language has several features that enable both parameter identification and model structure development to be implemented. This ensures that a robust and comprehensive method of model validation can be applied to Modelica models.

2 Features of the Modelica language relevant to Model Validation

There are many features of the Modelica language that make it ideal for implementing the approach to model verification and model validation described above. These features are discussed under six headings. The first four relate to requirements identified in sections 1.3 and 1.4:

- For model verification it is important that the model is separate from any analysis of that model.
- For parameter identification the model must be fully parameterised.
- For model structure development components must be replaceable with other representations of that component.
- For model structure development the equations of the model must be available in an accessible form.

The following additional features are necessary for the control of the model validation process:

- In order to produce a meaningful result from the model verification analysis, weighting factors must be applied to quantities within the model
- In order to control the model validation process, attributes need to be associated with the quantities, components and equations of the model.

2.1 Separation of the model and analysis

A model in which simulation and/or integration are integral parts of the model is of limited use when implementing model validation technologies, as these models will inherently suffer from the problems outlined in section 1.1.

Although the Modelica language has been designed to produce models that will be simulated (integrated against time) the formulation of these models does not explicitly require this. Furthermore there is no necessity for the integrators to be included within the formulation of the model.

Modelica models can therefore be analysed in ways other than by simulation techniques, such as techniques based on the model verification and validation approaches described above.

2.2 Model parameterisation

Full model parameterisation requires that any parameter of the model can be given a new value and the model re-executed, without the need for extensive recompilation. In this way the sensitivity of the model to parametric changes can be quickly assessed by the model validation analysis.

Parameters within a Modelica model are identified as such by the keyword **parameter,** used in the definition of that quantity. This allows such quantities to be treated appropriately, and the model checked for consistency.

This information can also be used to produce a full parameterisation of the model. Such a parameterisation would require that each parameter of the model was stored independently of the equations of the model, rather than hard coded into those equations.

There is no specific requirement in the Modelica language for models to be parameterised, and therefore this becomes an implementation issue; the language itself carries all the necessary information for a full parameterisation of the model.

2.3 Replaceable components

Being able to replace the model of a component with different variations of that model allows different component models to be assessed as to their suitability to model a particular physical sub-system.

This, in turn, allows intelligent trade offs to be made between the simplicity of individual sub-component models and the accuracy of the model as a whole.

Replaceable components have been part of Modelica since its first documented version. These allow components or sub-components of a model to be replaced with different variations of the same component. For example a model of a resistor can be replaced with a model of temperature sensitive resistor.

2.4 Availability of model equations in an accessible form

In a Modelica model, components are described by a series of equations. These equations are written in a standard form that can be interpreted, simplified and re-arranged as necessary.

The accessibility to these equations means that changes to the basic formulation of the model can be made, and the impact of such changes assessed with respect to the behaviour of the model.

In Modelica models, equations are grouped together under the definitions of the components that they represent. This makes the user-selection of equations to be assessed straightforward.

2.5 The association of weighting factors with quantities within the model

In order to generate a representative error from the model verification process described in section 1.3, it is important that all the errors across the model are scaled appropriately during the model verification process. This ensures that all measurement errors and all modelling errors are treated equivalently, rather than giving emphases to those quantities with a high nominal value.

For example, in a car, if both suspension force and wheel movements were measured as part of a test, it would be incorrect to assume that an error of 1N in the suspension force was equivalent to 1m of deflection in the suspension. In fact, it may be more appropriate to make 5kN of suspension force equivalent to 10mm of suspension deflection. Weighting factors of 5000 and 0.01 would therefore be applied to these quantities respectively. In fact, in most cases it makes sense to weight a given quantities error by the nominal value of that quantity.

Such scaling values can usually be determined from the nominal attribute of quantities within the model, but this is sometimes not the case, as will be discussed in section 4.

2.6 Attributes for control of the model validation process

Control attributes are required to mark quantities, components and equations as being included or excluded from the model validation process.

In certain situations it does not make sense for a quantity, a component or an equation of a component to be assessed as part of a model validation exercise. For example, it does not make sense to change an equation that directly implements a physical law, such as Newton’s Second Law.

In such cases, it is useful to mark these quantities, components, and equations within the model, so that they can be automatically excluded from any model validation analysis.

Annotations, with their relatively free form, are an ideal way of marking such quantities, equations and components, and feeding extra information about the model into the model validation process.

Using annotations in such a way enables all the important information about the validation process that relates to the model to be contained within the model itself.

3 Example: Validation of a model of a Racing Car.

The validation of a model of a racing car against test data is presented. The technique used is based on the model validation approach described above.

The model used for the validation exercise is shown graphically in figure 4.

The data used for the validation is taken from a test of the car on a chassis dynamic test rig and consists of data for:

- Forces applied by aerodynamic loading actuators
- Displacements and Forces applied by road input actuators

- Accelerations of the body and wheels
- Forces in the four pushrods
- Displacements of the four dampers
- Height, roll and pitch of the body of the car

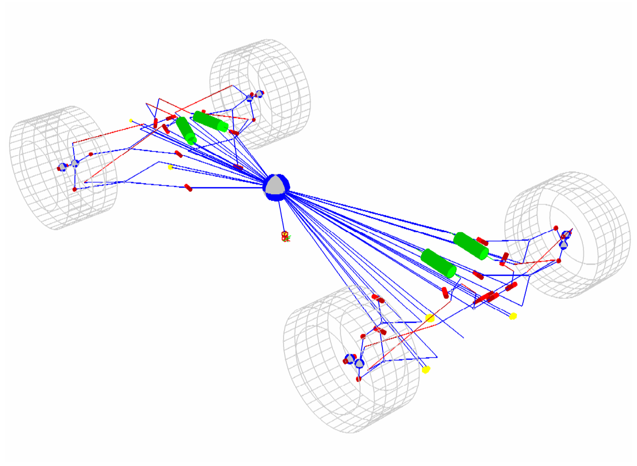


Figure 4: Model of a racing car.

The model validation technique used here is, in general, an iterative process where the largest causes of error are removed first, followed by the second largest, and so on. Each time a source of error is removed, other smaller errors become apparent in the model.

In this example, one iteration of parameter identification and one iteration of model structure development is presented, together with the overall model validation results.

3.1 Parameter Identification

Using the model verification technique described in section 1.3, a time history of the error between the model and measured data can be derived.

This error is analysed with respect to the model, using a cause and effect analysis, to determine which parameters are most likely to be causing the error between the model and the physical system. Furthermore the sensitivity of each parameter to the error is also calculated.

Table 1 shows a sample of results of such an analysis.

Table 1: Results of Parameter Identification analysis

Parameter	Correlation	Sensitivity
Car.Front.AntiRollBar.Stiffness	0.945	165.7
Car.Front.Suspension.Stiffness	0.856	1443
Car.Rear.AntiRollBar.Stiffness	0.832	-40.54
⋮	⋮	⋮

The values in the column labelled ‘correlation’ determine the likelihood that a particular parameter is causing the error between the model and the physical system. In this case therefore, it is most likely that the error is caused by the front anti-roll bar, however it may also be caused by the front springs or the rear anti-roll bar.

The choice of parameter can usually be made using the sensitivity index and engineering judgement. This parameter can then be adjusted, either automatically, or by the user, to a new and better value.

A new table of results similar to table 1, but with one of the modelling errors removed can then be calculated and the next iteration made.

3.2 Model Structure Identification

As an example of the development of the model structure, the rear dampers of the vehicle will be considered.

The model of these non-linear dampers includes an equation that specifies the force generated by each damper due to the velocity across the dampers.

The characteristics of this equation can be checked by removing the equation from the model, or turning the equation off, and treating the force generated by the damper as a continuous input to the model.

The model verification process will then generate a damper force against time that minimises the error between the model and real system.

The true characteristic of the non-linear damper can be examined by plotting the damper force against damper velocity. The form of the original equation can be checked and revised by considering the shape of this characteristic versus that generated by the equation it replaced.

The results for the model validation example are shown in figure 5, with both the left and right damper characteristics plotted.

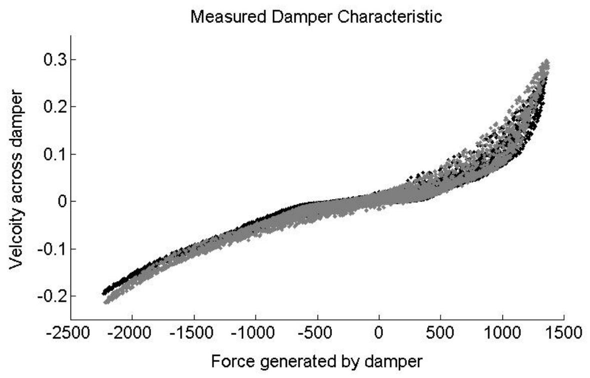


Figure 5: Identified Damper characteristic

There is good agreement between both the left and right dampers, and the expected characteristic.

Comparing the verification errors for the model with the damper equation turned on and turned off enables the error attributable to that component to be quantified.

3.3 Model Validation Results

In this example model validation, an initial error of 0.3 was reduces to a final error of 0.01. The model validation process was stopped at this point as it was decided that the model was of sufficient accuracy for its intended purpose.

The reduction in error can be broken down into 3 categories, listed in table 2.

Table 2: Errors with model

Error due to incorrect calibration of sensors	0.18
Error due to incorrect parameters	0.07
Error due to invalid assumptions	0.04

4 Future Developments of the Modelica Language for model validation

Although the Modelica language has many features that make it an excellent choice for modelling systems that will be validated against test data, it was originally designed for modelling systems to be simulated. As such there are two areas where small changes or developments of the language would improve the validation of Modelica models; these are extensions to the Real attributes and annotation specification.

Two further issues are discussed: the interpretation of Modelica models and the pre-resolution of constraint equations. The resolution of these issues would benefit both model validation and simulation tools.

4.1 Extension of the Real type, to include a sensitivity attribute

Although weighting factors can be currently entered using the **nominal** attribute of Real quantities, there is no possibility to enter sensitivity values. There are two cases in which sensitivity is an issue that need to be properly addressed when applying model validation techniques.

Firstly, some quantities have a large value, but a small range; for example, damper lengths as measured on racing cars. A damper may have a static length of approximately 20 cm, but a variation in this length of between 18 and 22 cm. In this example, the nominal value of 20 cm should not be used as a weighting factor for the error in the damper length; instead a value of 4 cm should be used. If it was possible to attribute a sensitivity of 4 cm to this quantity, this could be used to weight any errors attributed to the damper length.

Secondly, it is possible to have quantities that although have a large range, are very sensitive to variations in their value. Good examples of such quantities are the wheel speeds of a racing car and the forward velocity of that car. When calculating the tyre force, using standard models such as Pacejka, the difference is taken between the forward velocity of the car, and the velocity of the tread of the tyre relative to the vehicle. This gives the slip of the tyre on the ground. Changes in slip of only 1% of the speed of the vehicle can result in large forces being generated by the tyre. In this case the nominal value for the velocity of the car would be 50 m/s, whereas the sensitivity should be 1% of this at 0.5m/s.

For simulation tools sensitivity has not been a problem as the equations are all solved to 0, and tolerances, both relative and absolute are used to determine accuracy. However, such sensitivities may be useful for giving the user a greater degree of control of tolerances in specific situations.

4.2 Specific annotations for model validation

In section 2.6, attributes for equations and components were discussed. It was noted that the **annotation** mechanism in Modelica is suited to the requirements of indicating which equations were to be included in any model validation analysis.

As with the drawing of graphics and icons for components, it would be useful to include any specification of annotations within the Modelica specification, so that all model validation tools could use a common source of models.

4.3 Interpretation of Modelica models

Another area of development of the Modelica language is to consider whether it is possible to interpret (or Just-In-Time compile) Modelica models rather than pre-compile them.

For the purposes of model validation, interpretation offers many benefits over compilation as changes to equations within the model can be assessed more quickly without the need for recompilation of the model.

4.4 Pre-resolution of constraints

It is possible in some cases to pre-solve constraint equations and generate lookup tables for their solutions.-

For example, a double wishbone suspension on a race car, if modelled with solid elements, will have 5 states and 4 constraints. These constraint equations can be easily removed and lookup tables inserted. This reduces the size of matrices within the model validation tool and speeds up the entire model validation process.

5 Conclusions

Model validation should be an essential ingredient in any dynamic system development that uses Computer Aided Engineering (CAE) methods. The choice of CAE tools should reflect this, and so it is important with any modelling language development, such as that of the Modelica language, to consider whether the underlying structure of the language is suitable for the implementation of such techniques.

The Modelica language uniquely combines several features that make it an excellent base for the implementation of model validation techniques.

References

[1] Dorling R.J. ADS Trackside User Manual. Advanced Dynamic Systems Limited, 2004.

[2] Modelica Association. Modelica – A Unified Object-Orientated Language for Physical Systems Modeling – Language Specification, Version 2.1

Session 5a

Engines

Monte Carlo Simulations for Evaluating Engine NVH Robustness

John J. Batteh Michael M. Tiller Adam Goodman
Ford Motor Company, Research and Advanced Engineering
{jbatteh, mtiller}@ford.com, akgood@umich.edu

Abstract

This paper describes the use of a design-oriented engine cycle simulation model in Modelica for evaluating robustness of engine NVH (Noise, Vibration, and Harshness) to noise factors. This paper highlights the novel use of the cycle simulation model for analytic robustness studies using Monte Carlo simulations. The Monte Carlo simulations allow the robustness of a statistically significant engine population to be examined upfront in the product development process. The paper also discusses a flexible, extensible tool that was developed in conjunction with the Modelica models to streamline the description, execution, and results post-processing from the simulations.

Keywords: engine cycle simulation; NVH; Monte Carlo simulations; Fourier analysis

1 Introduction

Engine NVH is typically one of the vehicle attributes that contributes strongly to customer satisfaction and perceived quality [1]. The customer experiences the NVH characteristics of the vehicle via multi-sensory feedback. Radiated noise, steering wheel vibration, and seat track vibration are just a few of the common audible and tactile feedback mechanisms.

The vehicle NVH characteristics related to the engine result from the coupling of the dynamic engine torque with the transfer characteristics of the vehicle. The vehicle transfer function is affected by many different system-level design attributes, such as the design of the engine mounts and engine block, the vehicle frame design, and the vehicle stiffness, just to name a few. The design of the individual components and the overall system design are crucial for the development of a system that meets the functional requirements while maintaining acceptable NVH characteristics.

NVH evaluation is often performed on hardware components, primarily early prototypes. Due to the cost and limited availability of prototypes early in the design process, the evaluations are necessarily restricted in scope and usually at the nominal design. While more extensive evaluations can be performed on hardware later in the design process, the impact of the evaluations on the design is often limited due to the additional cost and potential program timing impact of design changes further downstream in the product development process. Furthermore, evaluating the impact of the manufacturing process and capability on the vehicle NVH is extremely difficult with prototype hardware due to the lack of a statistically significant population.

A robust product design requires the evaluation of the nominal design performance and sensitivity with respect to noise factors. This paper presents an analytic approach for evaluating engine NVH robustness to noise factors. The advantages of analytic NVH evaluation are many. Analytic evaluations are a cost-effective way of assessing NVH attributes upfront in the design process where changes are most easily accommodated. In addition to being costly, "cut and try" hardware experimentation can be extremely resource-intensive and time-consuming. Analytic evaluations can provide data in a more timely manner and allow for streamlining the NVH audits via batch simulations, parallel computing, and automated data collection and post-processing. Because even the noise factors can be set and accurately measured in the analytic models, the resulting data can clearly show the impact and interactions between the various factors. In addition, an analytic robustness evaluation can easily include the impact of manufacturing capability on the resulting NVH, thereby providing the opportunity for an optimal design including the effects of the manufacturing process for a statistically significant population representative of that in the hands of the customers. Furthermore, a detailed knowledge of the nominal design and its sensitivity to noise factors can lead to the feeding forward of additional requirements or control actions for the manufacturing process by the product development team to ensure robust product delivery to the customer.

2 Engine NVH Analysis

Engine NVH can be quantified in many different ways. One typical way of assessing engine NVH due to combustion torque is via calculation of the standard deviation of Indicated Mean Effective Pressure (SDIMEP) in the engine, essentially a measure the variation in the combustion event amongst the various cylinders. An engine with good NVH characteristics would typically have uniform combustion and thus similar power output from the various cylinders. However, past work has shown that SDIMEP does not often correlate well with engine NVH metrics observed by the customer [1]. A new technique for evaluating combustion variation called Combustion Torque Uniformity [1]-[2] is applied in this work. This approach examines the frequency content of the engine torque to analyze combustion non-uniformity. This section describes the models and techniques used to analytically calculate Combustion Torque Uniformity metrics for a V6 engine subject to noise factors.

2.1 Cycle Simulation

At the heart of the analytic engine NVH methodology is the cycle simulation model. This model describes the detailed thermodynamics of the breathing, compression, combustion, and expansion of the gas mixture. Figure 1 shows the Modelica representation of the GESIM predictive cycle simulation model [3]-[4].

The details of the cycle simulation submodels influence the types of noise factors that can be considered in the NVH analysis. The cycle simulation model used in this study includes the following submodels:

- Multi-zone, predictive combustion based on thermodynamics and bulk fluid motion
- Pseudo-species formulation with detailed mixture property calculations for the thermodynamic media
- Gas exchange across the valves with the valve lift kinematically determined from the cam position and the valve lash
- Detailed thermal response models for the block, head, and piston
- Intake reservoir boundary conditions including the pressure, temperature, and composition

GESIM has previously been used to simulate cycle-to-cycle variability based on factors related to the physics of early flame development [5].

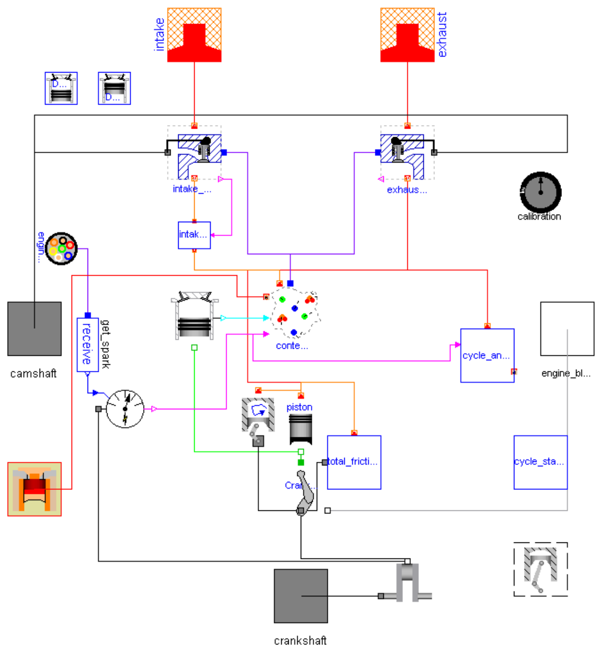


Figure 1. GESIM cycle simulation cylinder

The cycle simulation model shown in Figure 1 can be inserted into predefined engine templates to simulate multi-cylinder engines [4], [6]. The multi-cylinder engine templates use a `replaceable` cylinder model that is instantiated locally. The specific engine to be simulated is created by extending the appropriate engine template (*i.e.* single cylinder, I4, V6, *etc.*) and redeclaring the cylinder model. Figure 2 shows the dual-plenum V6 engine configuration with each cylinder as the GESIM cycle simulation model shown in Figure 1.

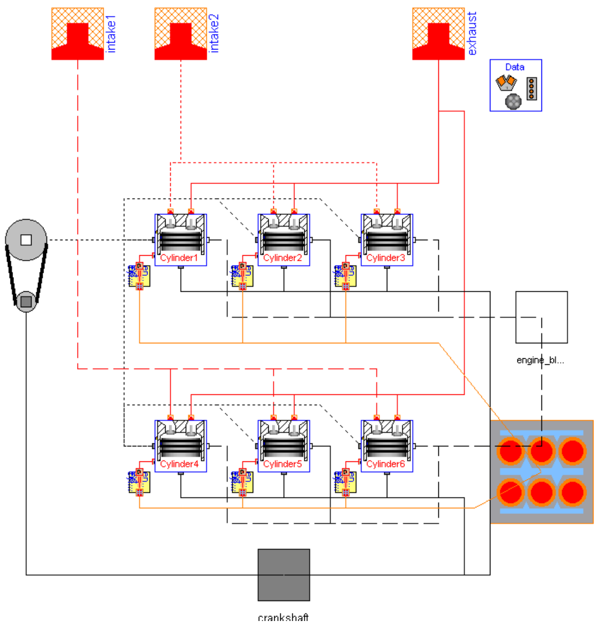


Figure 2. V6 (dual plenum) engine configuration

The test case for the engine NVH simulations is shown in Figure 3. This model is an extension of our existing flexible dyno template [4] and has previously been used for detailed powertrain NVH simulations [7]. Figure 4 shows the code required to modify the dyno template to simulate a V6 engine with GESIM cylinders. The engine geometry data is specified by the `redeclare` of the EngineData package. The user defines the test conditions to be simulated by specifying the engine speed, spark timing, and intake conditions via the engine controller. An additional block was added to the top-level model to perform the Fourier analysis of the resulting engine torque and will be discussed in greater detail in the next section.

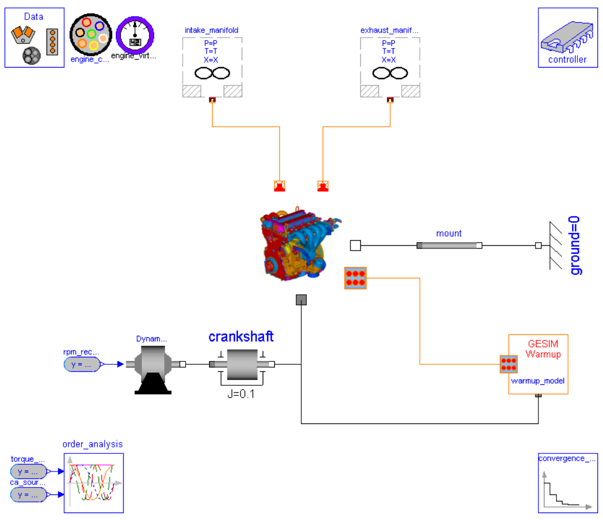


Figure 3. Engine NVH test case

```
model EngineNVH
  extends DynoSetup(
  redeclare package EngineData = MyEngine,
  redeclare model CylinderModel = GESIM,
  redeclare model Configuration =
    V6DualPlenum(redeclare model
                  CylinderModel= CylinderModel));
…
end EngineNVH;
```

Figure 4. Code excerpt from engine NVH test case

2.2 Fourier Analysis

The Combustion Torque Uniformity technique [1]-[2] analyzes the harmonics of the engine torque waveform. The torque order content above the 0th order and less than the firing frequency is computed via Fourier decomposition. The 0th order torque content represents the work done on the crankshaft while the magnitude of the other harmonics are non-zero as a result of non-uniform combustion events. The magnitude (A_n) and phase (ϕ_n) of the nth order harmonic [1] is given by the following Fourier representation:

$$A_n \cos(n\theta_i + \phi_n) \qquad (1)$$

where θ_i is the crank angle. The code excerpt in Figure 5 illustrates the Modelica implementation of the discrete Fourier transform calculations.

```
model OrderAnalysis
…
  for m in 1:num_order loop
    s_sum := 0;
    c_sum := 0;
    for i in 1:no_pts loop
      s_sum := s_sum + waveform_sample[i]*
        sin(4*pi*order[m]*(i - 1)/no_pts)*2/no_pts;
      c_sum := c_sum + waveform_sample[i]*
        cos(4*pi*order[m]*(i - 1)/no_pts)*2/no_pts;
    end for;
    s_sum_temp[m] := s_sum;
    c_sum_temp[m] := c_sum;
    mag[m] := if (order[m] == 0 or order[m] == no_pts)
       then 0.5*((s_sum)^2 + (c_sum)^2)^0.5 else
       ((s_sum)^2 + (c_sum)^2)^0.5;
    phase[m] := if atan2(-s_sum, c_sum)*rad2deg < 0
       then 360 + atan2(-s_sum, c_sum)*rad2deg else
       atan2(-s_sum, c_sum)*rad2deg;
  end for;
…
end OrderAnalysis;
```

Figure 5. Code excerpt from order analysis model

The analysis can be performed either on the torque from a multi-cylinder engine model or via the superposition of the calculations from individual cylinders. The following equation can be used to calculate the contribution to the engine harmonic from an individual cylinder based on the cylinder phasing and firing order:

$$A_n \cos(\phi_n - \psi_i n) \qquad (2)$$

where A_n is the magnitude of the nth order harmonic for the ith cylinder, ϕ_n is the phase of nth order harmonic for the ith cylinder, and ψ_i is the firing angle for the ith cylinder.

As an example of a typical engine torque signature, Figure 6 shows the simulated engine torque from one firing cycle of a uniform V6 engine along with the torque from cylinder 3. The engine torque shows the 6 distinct firings and superposition of the resulting torque pulses from the individual cylinders. Note that the torque pulse from each cylinder is identical as there was no noise introduced in the geometry or operating conditions for the individual cylinders. Figure 7 shows the simulated engine torque for the same V6 engine as in Figure 6 but with the introduction of noise into the operation conditions such that cylinders 1-3 are still operating at a stoichiometric air-fuel (AF) ratio of 14.6 while cylinders 4-6 are now operating lean at AF=16. In comparing both the engine and cylinder torques from Figure 6 and Figure 7, it is clear that the resulting torque signatures are different and could lead to the excitation of different vehicle NVH modes when coupled with the transfer function of the vehicle. Table 1 shows the torque harmonics for Cylinders 3 (AF=14.6) and 4 (AF=16) from the engine in Figure 7. Note that the lean cylinder has lower torque magnitudes as expected.

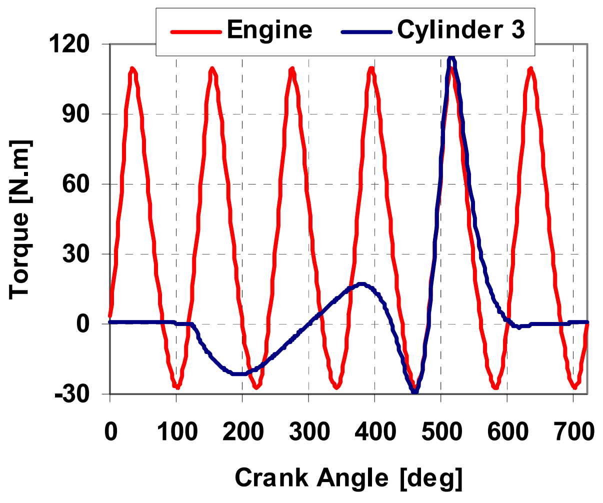


Figure 6. Torques from a uniform V6 engine

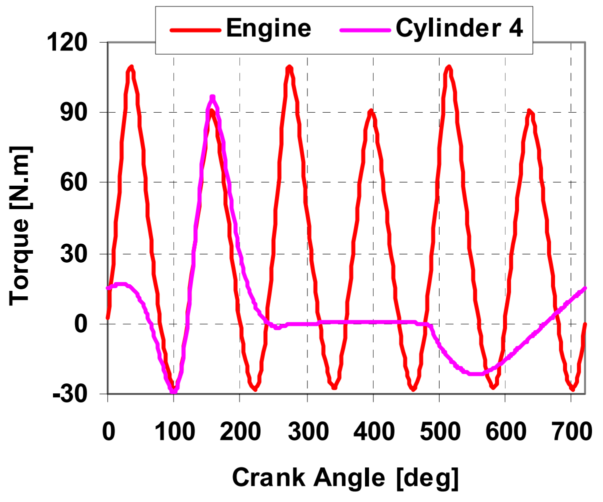


Figure 7. Torques from a V6 engine with cylinders 1-3 operating at AF=14.6 and cylinders 4-6 at AF=16

Table 1. Cylinder torque harmonics calculations

Torque Harmonics		Cyl. 3 AF=14.63	Cyl. 4 AF=16
0	Mag [N.m]	5.64	4.72
	Phase [deg]	0	0
0.5	Mag [N.m]	20.61	18.81
	Phase [deg]	336.3	335.8
1	Mag [N.m]	7.79	6.10
	Phase [deg]	332.7	335.1
1.5	Mag [N.m]	17.15	15.98
	Phase [deg]	265.9	262.1
2	Mag [N.m]	14.91	13.57
	Phase [deg]	266.7	264.0
2.5	Mag [N.m]	14.57	13.36
	Phase [deg]	247.9	245.0
3	Mag [N.m]	10.82	9.55
	Phase [deg]	245.6	243.4

2.3 Methodology

A key advantage to analytical engine NVH analysis is the ability to evaluate a statistically significant engine population. Rather than simulate a large number of V6 engines with a multi-cylinder engine model subject to various noise factors in the individual cylinders, it is far more computationally efficient to simulate a large number of single cylinder engines using the Monte Carlo method [8] to choose the value of the noise factor(s) for each run and then "virtually" assemble the single cylinders into a V6 engine. The following methodology was used to perform the analytic engine NVH analysis (see Figure 8):

1. For the given engine, build and calibrate the single cylinder cycle simulation model.
2. For the operating condition of interest, configure the Monte Carlo simulations by determining the noise factors and their distributions (*i.e.* from manufacturing process capability, *etc.*).
3. Perform the Monte Carlo simulations with the single cylinder model to generate a library of single cylinder results.
4. Assemble multi-cylinder engines from library of single cylinder results.
5. Analyze engine population and calculate statistics of interest.

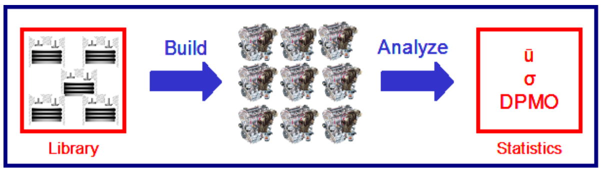


Figure 8. Engine assembly methodology

3 NestedAnalysis Toolkit

While considerable effort is made in these kinds of analyses to capture the appropriate level of detail in the models, it is important not to lose sight of the bigger picture. Ultimately, the simulation of the underlying model is simply a means of generating data. Such data can then be used in a variety of ways. For example, frequently models are used as the basis of a design optimization process whereby the simulation evaluates perspective designs according to a set of objectives or constraints.

For this reason, an analysis tool, called the NestedAnalysis Toolkit, has been developed internally that can be used to construct models of the complete analysis. The term "nested" refers to the fact that complete analysis is composed of a hierarchy of other analyses. For example, in quality and robustness analyses it is quite common to try and minimize the variation of a products performance with respect to uncontrollable noise factors.

Figure 9 shows how such an analysis could be represented graphically. At the bottom of the hierarchy are individual simulations. Each simulation represents slightly different conditions. The conditions are generated automatically based on statistical information about the noise factors being considered and their impact on the inputs to the simulation. After these simulations are completed, the results are compiled and analyzed to produce statistical information. This statistical information can then be used in subsequent analyses (*i.e.* an optimization process in this case) to minimize the variation.

The construction of these analyses was originally described in Python [9]. The descriptions were essentially declarative in nature, and the analysis engine would use the declarative descriptions of the analysis to coordinate the complete analysis. In order to make the toolkit more useful to end users, a graphical user interface was developed. Using the new interface, users constructed the analyses graphically in a hierarchical manner. The goal of the user interface was to show users a representation that was intuitive, like the overview shown in Figure 9. Figure 10 provides screenshots from a sample Monte Carlo analysis of a Dymola[1] [10] transient model, in this case the `TwoMasses` example from the Modelica Standard Thermal library.

While support for running simulations generated from Dymola was a key feature of the toolkit, the toolkit architecture was developed to accommodate a range of different types of analyses. For example, support for using Excel spreadsheets within the framework was easily added. Furthermore, each plug-in added to the framework was developed specifically to support each analysis type. So, for example, the support for Dymola simulations was able to automatically construct lists of input parameters and results and display them for the user to choose as either inputs or outputs in the nested analysis. In addition, the toolkit has an extensible architecture for adding node analyses. Currently the architecture supports Monte Carlo analysis, full factorial Design of Experiments (DOE), and some optimization functionality. The vision is to be able to develop new analysis plug-ins as needed. Some examples of possible additional analysis plug-ins are sensitivity analysis, Latin hypercube sampling, and fractional factorial DOEs.

There is a fundamental philosophical principle in our toolkit that bears some explanation. We do not rely on the simulation tools themselves to provide these capabilities. There are two reasons for this approach. First, we want our analysis capabilities (Monte Carlo analyses, optimization, *etc.*) to work with multiple tools, not just Dymola. Furthermore, we do not want to distract simulation tool developers with functionality that we consider to be "above" the simulator. That being said, we recognize that there are also great advantages to having these capabilities integrated into a simulation tool as well.

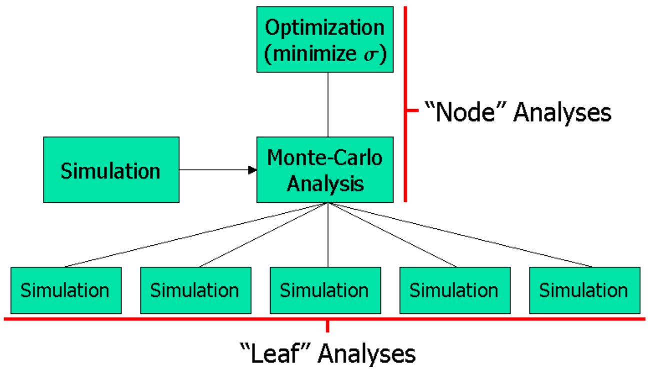
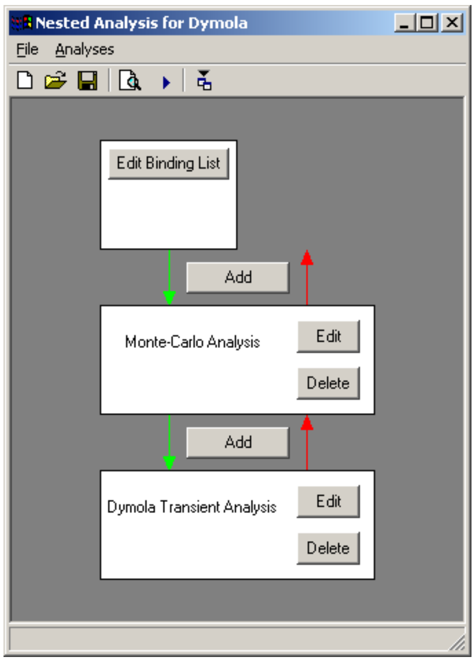
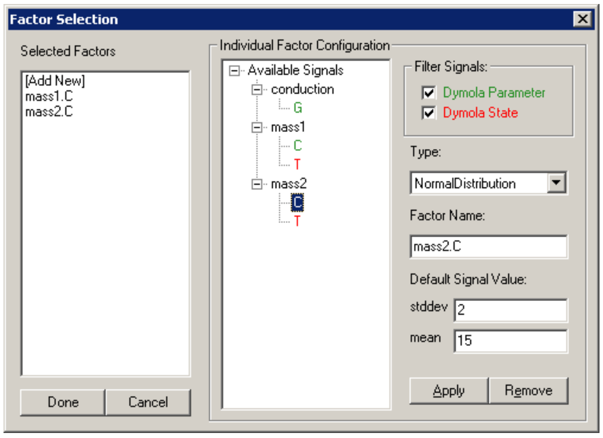


Figure 9. Building blocks for hierarchical analyses

[1] Dymola is a trademark of Dynasim AB



(a) Analysis construction



[Add New]
mass1.C
mass2.C

Available Signals
- conduction
  - G
- mass1
  - C
  - T
- mass2
  - C
  - T

☑ Dymola Parameter
☑ Dymola State

Type: NormalDistribution

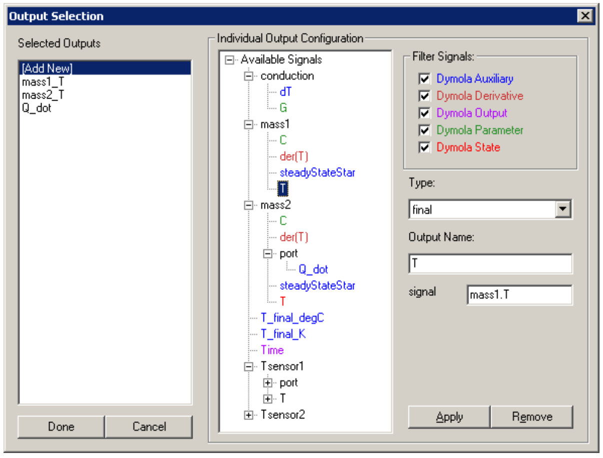
Factor Name: mass2.C

Default Signal Value:

stddev: 2

mean: 15

(b) Monte Carlo factor selection



[Add New]
mass1_T
mass2_T
Q_dot

Available Signals
- conduction
  - dT
  - G
- mass1
  - C
  - der(T)
  - steadyStateStar
  - T
- mass2
  - C
  - der(T)
  - port
    - Q_dot
  - steadyStateStar
  - T
- T_final_degC
- T_final_K
- Time
- Tsensor1
  - port
  - T
- Tsensor2

☑ Dymola Auxiliary
☑ Dymola Derivative
☑ Dymola Output
☑ Dymola Parameter
☑ Dymola State

Type: final

Output Name: T

signal: mass1.T

(c) Output variable selection

Figure 10. NestedAnalysis Toolkit GUI screenshots for a Monte Carlo analysis of a Dymola transient model

4 Results

The methodology described in Section 2.3 was used to simulate the effects of valve lash variation in a sample V6 engine at a fixed operating condition. Valve lash is an important design variable as it affects the timing and duration of the valve events, the maximum valve lift, and the overlap between valves. Thus, it directly impacts breathing, mixture preparation, and combustion. Since there is some variation in the lash of each cylinder in the assembled engine due to the manufacturing process, it is highly desirable to understand the NVH effects of this variability in the engine population.

The NestedAnalysis Toolkit described in Section 3 was used to establish and perform the simulations with Dymola [10]. One hundred single cylinder Monte Carlo simulations were performed with a normal distribution for the variation in valve lash. From the library of 100 single cylinder runs, 10,000 V6 engines were assembled and analyzed. Each engine was assembled by randomly choosing 6 cylinders from the library (see Figure 8). To determine an appropriate number of engines to assemble to represent the engine population, the number of assembled engines was increased until the overall engine population statistics converged. To examine the sensitivity of the engine population to the process capability, the Monte Carlo simulations were conducted with four different standard deviations for valve lash. Figure 11 shows three of the simulated valve lash distributions (note that all distributions are normal but with different standard deviations).

Figure 12 shows the results of the engine population analyses for the various valve lash distributions. Figure 12a-b shows the histograms of the 1.5 order torque in the assembled engine populations for $\sigma = 0.02$ mm and 0.01 mm, respectively. Figure 12c shows the engine population statistics for 1.5 order torque. Note that as the standard deviation of the valve lash increases, there is both a larger mean 1.5 order torque and more variability in the engine population as indicated by the error bars showing $\pm 1\sigma$ levels. Understanding the sensitivity of the engine population NVH characteristics to valve lash leads to the ability to optimize the lash centering and manufacturing process capability to optimize engine population robustness. Furthermore, given a specification on the various torque magnitudes, the analysis would also yield information as to the fraction of engines in the population that would meet the specifications.

While the sample simulations shown here considered a single noise factor with a normal distribution, the approach is general and can be used with multiple noise factors and a variety of distributions. Currently the Monte Carlo analysis plug-in supports normal, log normal, uniform, beta, exponential, gamma, and Pareto distributions, and the additions of new, user-defined distributions are trivial. Furthermore, simulations with multiple noise factors can be analyzed with existing statistical techniques to identify main effects and interactions between the factors.

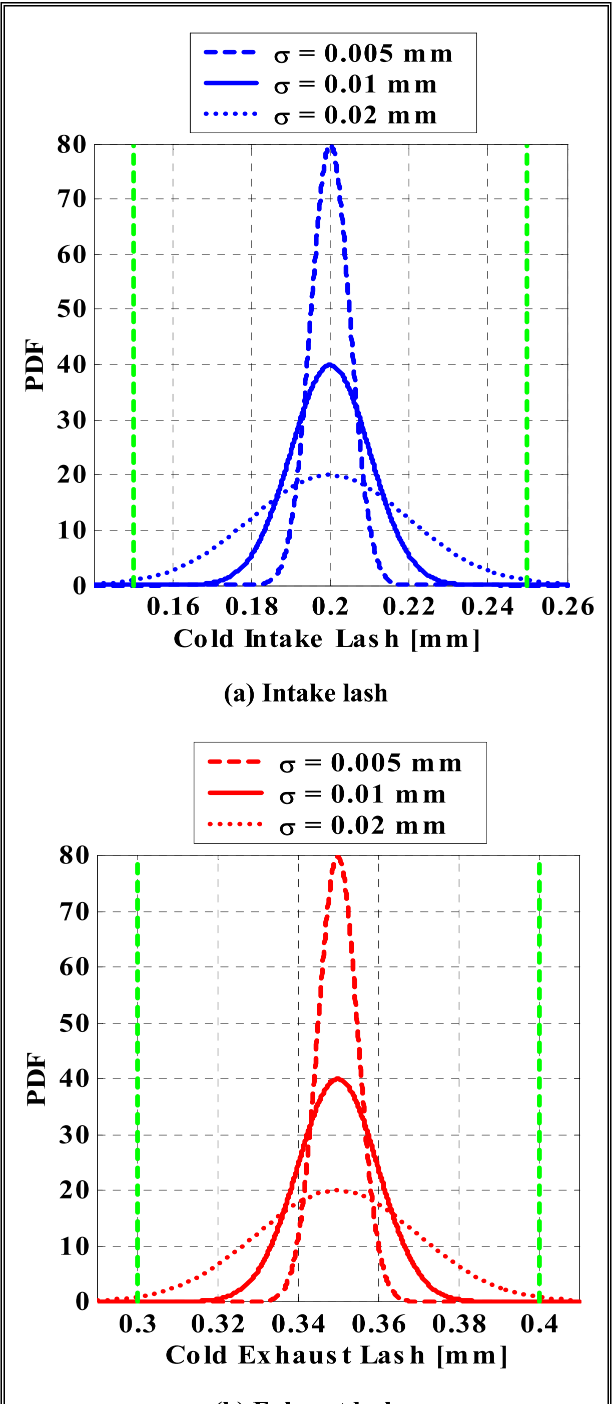


Figure 11. Simulated distributions for valve lash

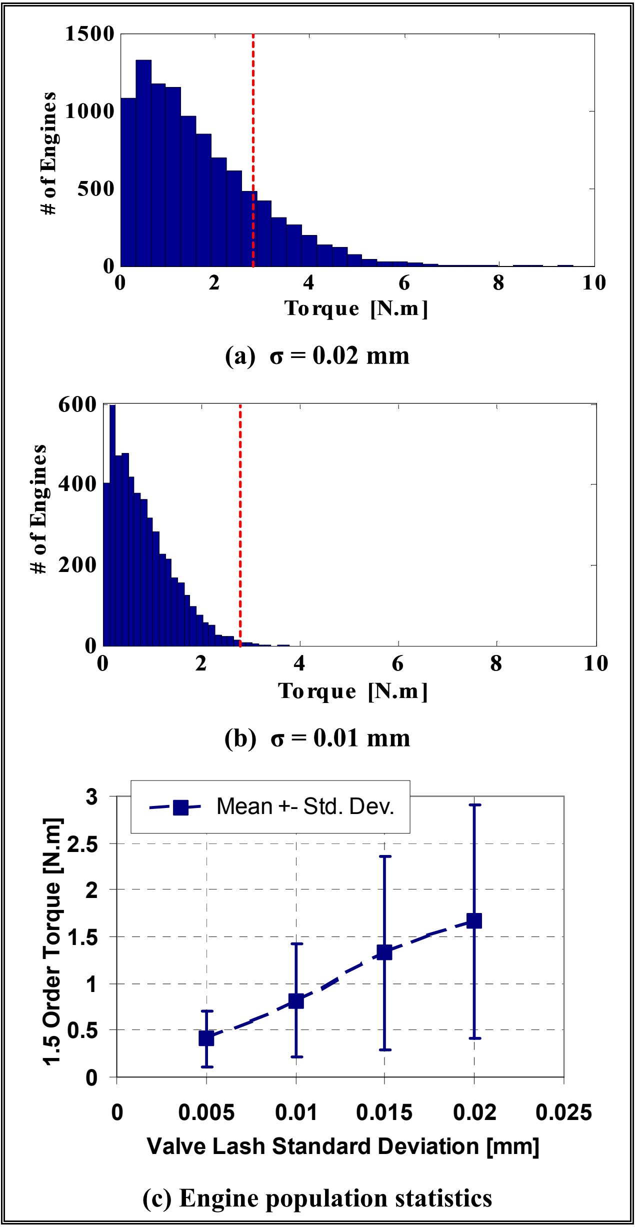


Figure 12. 1.5 order torque analysis for engine population due to valve lash variation

5 Conclusions

This paper discusses a methodology for analytical NVH simulations using the Combustion Torque Uniformity technique. A novel simulation approach using a design-oriented cycle simulation model and the Monte Carlo method for simulating the effects of noise factors allows the robustness of a statistically significant engine population to be analyzed upfront in the design process. In addition, the approach allows for multiple noise factors to be simulated according to various distributions to examine design sensitivities and interaction effects. The ability to analytically simulate an entire engine population leads to the opportunity for the optimization of the engine design coupled with the manufacturing process capability to deliver the most robust product to the customer. Furthermore, the flexible, descriptive NestedAnalysis Toolkit has been developed to streamline the description, execution, and results post-processing from these sorts of robustness studies.

References

[1] Stout, J.L., Mancini, M., Host, R., and Hancock, K., 2003, "Combustion Uniformity as a Measure for Engine Idle NVH," *SAE-2003-01-1429*, Society of Automotive Engineers.

[2] Stout, J.L., 2001, "Engine Excitation Decomposition Methods and V Engines Results," *SAE-2001-01-1595*, Society of Automotive Engineers.

[3] Newman, C., Batteh, J., and Tiller, M., 2002, "Spark-Ignited-Engine Cycle Simulation in Modelica", 2nd International Modelica Conference Proceedings, pp. 133-142, http://modelica.org/Conference2002/papers/p17_Newman.pdf

[4] Batteh, J., Tiller, M., and Newman, C., 2003, "Simulation of Engine Systems in Modelica", 3rd International Modelica Conference Proceedings, pp. 139-148, http://www.modelica.org/events/Conference2003/papers/h34_Batteh.pdf

[5] Brehob, D.D. and Newman, C.E., 1992, "Monte Carlo Simulation of Cycle by Cycle Variability", *SAE922165*, Society of Automotive Engineers.

[6] Bowles, P. and Batteh, J., 2003, "A Transient, Multi-Cylinder Engine Model Using Modelica", *SAE-2003-01-3127*, Society of Automotive Engineers.

[7] Tiller, M., Tobler, W.E., and Kuang, M., 2002, "Evaluating Engine Contributions to HEV Driveline Vibrations", 2nd International Modelica Conference Proceedings, pp. 19-24, http://modelica.org/Conference2002/papers/p03_Tiller.pdf

[8] Metropolis, N. and Ulam, S., 1949, "The Monte Carlo Method", *Journal of the American Statistical Association*, 44 (247), pp. 335-341.

[9] Lutz, M. and Ascher, D., 1999, *Learning Python*, 1st Ed., Cambridge: O'Reilly and Associates.

[10] Dymola. Dynasim AB, Lund, Sweden, http://www.dynasim.com.

Dynamic Simulation of a Free-Piston Linear Alternator in Modelica

Sven-Erik Pohl[*] Markus Gräf[†]

German Aerospace Center (DLR), Institute of Vehicle Concepts
Pfaffenwaldring 38-40, 70569 Stuttgart

Abstract

This paper presents the modeling and simulation of a novel development of a free-piston engine in Modelica. The investigated concept is a combination of a combustion process and a linear alternator designed to provide clean, efficient energy in a compact engine. To study the features of free-piston engines a Modelica library is outlined containing basic and advanced component models. Detailed sub-models are investigated in order to design and improve hardware components. Control strategies are developed and dynamically tested within the Modelica simulation. Dymola/Modelica was found to be the best tool to examine the dynamic system behavior.

Keywords: free-piston engine, linear alternator, power electronics, control strategies

1 Introduction

The free-piston linear alternator proposed by the German Aerospace Center (DLR) - Institute of Vehicle Concepts (IFK) combines a two stroke combustion engine with a linear alternator. An adjustable gas spring is used to reset the piston assembly (Figure 1). The engine is designed to enable new degrees of freedom for advanced optimization of the combustion process. In contrast to conventional crankshaft engines the free-piston design offers mainly three degrees of freedom to improve engine performance:

- variable stroke
- variable compression ratio
- variable piston velocity

[*]sven.pohl@dlr.de
[†]markus.graef@dlr.de

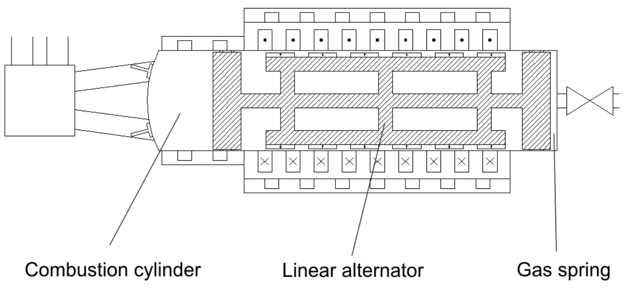


Figure 1: The free-piston linear alternator concept

These key features allow for designing a combustion process with low emissions and development towards homogenous charge combustion ignition (HCCI). The variable stroke and variable compression ratio can be used to optimize the combustion process for part load conditions. The goal is to achieve a compact electric power engine with high efficiency and reduced emissions at low costs. The free-piston linear alternator aims towards automotive application as auxiliary power unit as well as power generator in hybrid electric vehicles.

2 Modeling Objectives

At IFK a hardware demonstrator is currently being built to investigate the functionality of the free-piston linear alternator. In parallel to the hardware components a dynamic simulation model of the complete system is developed using Modelica. A Modelica library is outlined with the following objectives:

- specify hardware components
- develop control strategies
- analyze the combustion process
- evaluate operation modes

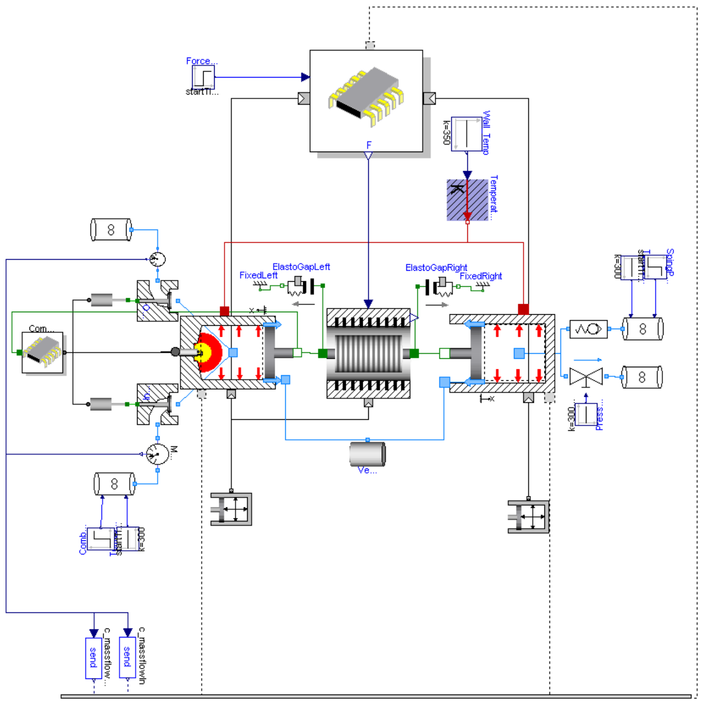


Figure 2: Modelica assembly of free-piston linear alternator

- design dynamic system behavior

The investigation of the operation modes and their dynamic transitions are of special interest since knowledge of system reaction on changes in the parameters is not available. Due to Dymolas dynamic modeling capability parameter influence on stroke, compression ratio and piston motion are a simulation focus.

3 Modelica Architecture

A Modelica library was outlined to provide basic and advanced components for free-piston engine modeling. A major scope is the compatibility with Modelica standard libraries and with future standards of thermodynamic modeling.
The free-piston linear alternator model can be built from library components. Figure 2 shows the simulation setup. The control systems, the thermodynamic models of combustion cylinder and gas spring are shown. The electrical system consists of the linear alternator and the power electronics. On a sub-layer the physical effects are modeled in detail: The mass flows into and out of the cylinders can be observed, the combustion process is modeled, heat transfer effects are investigated and the thermodynamic properties describe the state of the cylinders.
The `FixedShape` from the `Multibody` library is extended to visualize the motion of the piston assembly. The cylinder pressures are visualized by changing the cylinder color. Figure 3 shows the 3D representation of the free-piston linear alternator. The combustion cylinder is shown on the left, changing color during combustion. The linear alternator is shown in the middle next to the gas spring cylinder on the right. The animation is not only helpful for presentation purposes but also enables the developer to analyze the dynamics of the free-piston linear alternator model.

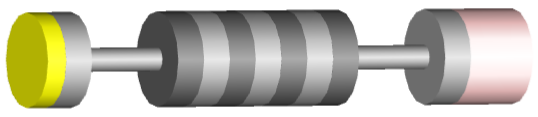


Figure 3: Modelica visualization of free-piston linear alternator

3.1 Interfaces

Modelica provides connector definitions for most physical domains. However, a common thermodynamic connector is not yet available. Thus, pressure and temperature are the two potential variables, mass flow rate as well as heat flow rate are chosen to be the flow variables. Additionally, to satisfy multi-phase flow conditions, the mass fraction for each species is added. From this information all necessary thermodynamic properties can be calculated. A code fragment of the connector definition is shown below:

```
connector CombustionGas
 parameter FKLG.Types.NumberOfSpecies nX;
 SI.MassFraction X[nX] "Mass fraction";
 SI.Pressure P "Gas pressure";
 SI.Temperature T "Gas temperature";
 flow SI.MassFlowRate mdot "Mass flow rate";
 flow SI.HeatFlowRate Hdot "Heat flow rate";
end CombustionGas;
```

For an ideal gas for example, the number of species equals one and the mass fraction is consequently unity.

Complex models tend to exchange a fair amount of information leading to components with several connectors and connections. To reduce the connection complexity a bus system is defined as an assembly of other connectors. Defining a bus currently becomes tedious since every signal has to be added by hand. The following code fragment shows the bus implementation for combustion, gas spring and linear alternator:

```
connector Bus
  import SI = Modelica.SIunits;
  import MoIn = Modelica.Blocks.Interfaces;
// Combustion
MoIn.RealSignal c_pressure (redeclare
 type SignalType = SI.Pressure);
MoIn.RealSignal c_temperature (redeclare
 type SignalType = SI.Temperature);
MoIn.RealSignal c_position(redeclare
 type SignalType = SI.Position);
MoIn.BooleanSignal c_burning;
...
// Gas Spring
MoIn.RealSignal s_pressure(redeclare
 type SignalType = SI.Pressure);
MoIn.RealSignal s_temperature(redeclare
 type SignalType = SI.Temperature);
MoIn.RealSignal s_massflowOut(redeclare
 type SignalType = SI.MassFlowRate);
...
// Linear Alternator
MoIn.RealSignal force(redeclare
 type SignalType = SI.Force);
end Bus;
```

3.2 Base Models

Defining base models for components enables the user to implement further models with the same external connections but different content. Such a base model may contain variable declarations, connectors and common equations. Another benefit from this declaration is the use of selection boxes in higher level models to switch between all models extended from the base model. In that way the user can change the setup of complex models by simply selecting different components from a list. At this time gas properties, fuel models, heat transfer models and combustion models are implemented using the base model approach.

4 Thermodynamical System

The free-piston linear alternator model is divided into the sub-models of combustion cylinder, linear alternator, gas spring and controls. These components are built from basic thermodynamic components, like control volumes, valves and pipes. Detailed sub-models concerning heat transfer and piston blowby are added to the cylinder model. The gas spring model is validated with experimental data. Both, heat transfer and blowby model are successfully obtained using other simulation software.

4.1 Combustion Modeling

Generally, the simulation of combustion is a highly complex process involving several disciplines such as thermodynamics, heat transfer, chemical kinetics, and fluid motion.
Since the compression ratio and the stroke of the free-piston alternator is not constant through out the operation, a major task is to define an appropriate combustion model. To describe the operation modes of the combustion process it can be divided into several combustion processes with differing strokes and compression ratios. In other words the free-piston linear alternator contains several conventional combustion cylinders with the same diameter but varying stroke and compression ratio. The challenge is to find a representative physical process that best describes the combustion behavior with a minimum of input parameters since experimental data is not yet available. Thus, a rough approximation of the combustion process is needed. As a starting point the combustion components presented by Tiller [2] can be used. The cylinder gas is modeled as a single phase ideal gas leading to a straightforward formulation of all connected components such as valves and pipes for the gas exchange. This is not a truly satisfying solution yet and a more detailed combustion model is currently under construction.

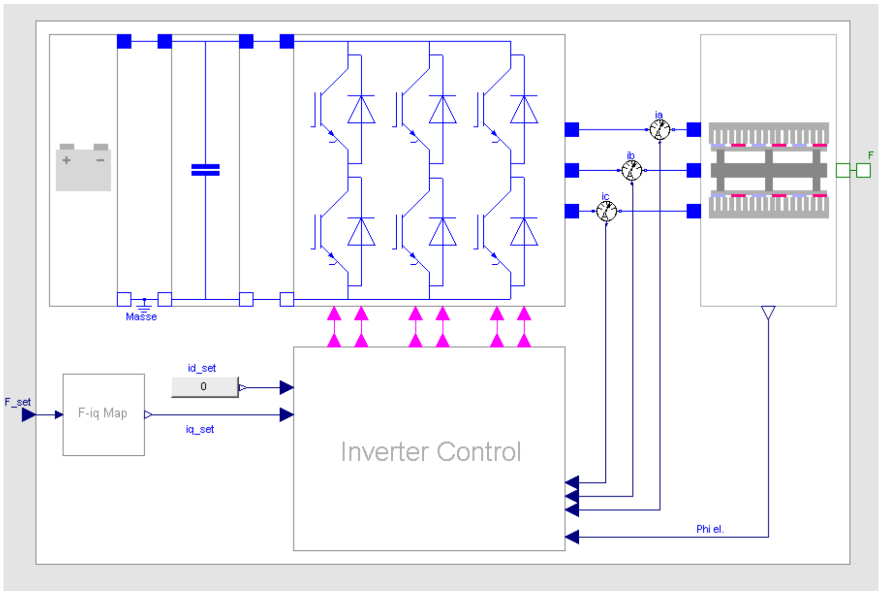


Figure 4: Electrical system of linear alternator

4.2 Medium Models

Three property models for perfect air, ideal air and an air/exhaust gas mixture were implemented as base layer for the gas spring and combustion components using Modelicas `replaceable` notation. The air/exhaust gas properties use the correlations by Zacharias [4]. To reduce computation time the property model was transferred into Modelica.

4.3 Heat Transfer Models

As described in section 3.1 the cylinder wall heat loss models are extended from a base heat transfer model. Two basic approaches for the wall heat loss are available. The first handles the cylinder heat loss for idle running engines based on the approach by Huber [1]. Secondly, for a firing engine, the approach by Woschni [3] is implemented.

4.4 Orifice Flow Models

A general orifice flow model using the isentropic flow formulation found in textbooks is extended for valve modeling.

Commonly, a small gas leakage between piston and cylinder exits the cylinder. This blowby gas flow is based on the orifice flow model. The effective area is implemented as a parameter and is validated using experimental data in case of the gas spring.

5 Mechanical System

The mechanical system is represented by the piston where the equation of motion is solved. Additionally, a spring-damper system mechanically prevents the piston from reaching the cylinder heads or the cylinder pressure from rising above a critical value.

6 Electrical System

The electrical system consists of the battery, the intermediate circuit, the power electronics module and the electromechanical model for the linear alternator.

The control unit demands a specific force from the linear alternator. This signal is mapped into a set-value *iq* for the inverter control. The inverter control generates PWM-signals for the IGBT-B6-Modul using the *dq*-transformation. The IGBT is modeled as a diode and a switch. The free wheeling diode is put in parallel to the IGBT. The diode itself is described with 3 characteristic curves, a straight line for the negative branch, a 3rd grade polynomial equation for the forward characteristics and a straight line for describing the system beyond the normal operation area.

The linear alternator is described with maps for the flux linkage and the inductance matrix for every position. This data is impressed on the equivalent circuit of a permanent magnet machine, composed of a resistor, an inductance and a voltage source for the induced voltage for all 3 phases.

The generated force is calculated after a *dq*-transformation of the real currents with the inverse F-iq map described in the beginning. As a result the linear alternator force is simulated under dynamic conditions including all time constants influencing the over-all system. In addition the Modelica inverter control model will be used in combination with the dSPACE box for controlling the constructed hardware. The power electronics are mainly modeled using the `Modelica.Electrical` and `Modelica.StateGraph` packages.

7 Control System

Disconnecting the piston from the crankshaft requires a new approach to system control since the continuous shaft motion is not available. In fact developing control strategies is the most challenging task in the free-piston linear alternator design process.

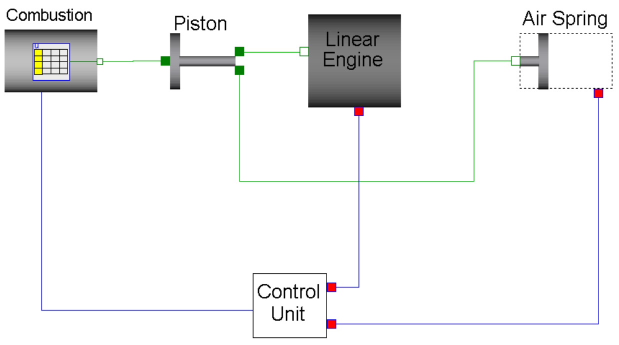


Figure 5: Simulation setup for co-simulation

In conventional engines it is needless to mention that the crankshaft returns the piston to the starting point of a cycle. In a free-piston engine the piston not necessarily returns to the same point. The piston position is strongly dependent on the states in the cylinders and the energy converted by the linear alternator. A piston motion control is developed by adjusting the converted energy of the alternator such that the piston returns to its starting point. To account for all losses occurring during the cycle the cylinder pressures are taken as calculation basis. It should be noted that the linear alternator can be actively used to control the piston motion. Hence, the linear alternator control can accelerate or slow down the piston to either influence the combustion process or prevent the piston from crashing.

In order to level the power output the energy released by the combustion process is converted by the linear alternator partly in the expansion phase and partly in the compression phase. Thus, the gas spring is used as temporary energy storage. This ”force split strategy” also effects the piston motion depending on the amount of energy converted in each phase. The simulation results presented in this paper are based on an equal energy conversion in expansion and compression phase. Extracting the energy in both, compression and expansion phase also reduces the linear alternator size and consequently piston weight. The

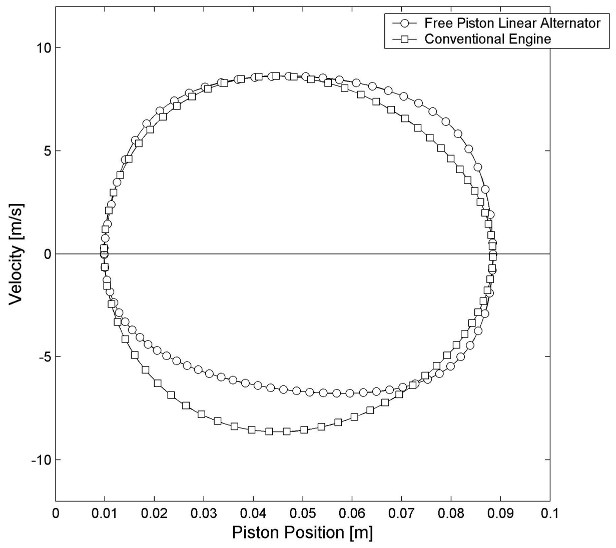


Figure 6: Comparison of piston velocities for crank shaft engine versus a free-piston linear alternator

variable volume of the combustion cylinder demands for flexible valve and ignition timing. An ”electronic camshaft” needs to be implemented to control valve and ignition timing according to the operation mode. For that purpose a virtual camshaft angle is introduced to coordinate the timing issue.

8 Simulation Results

In the remainder of this section two simulation approaches are shown investigating the potentials and challenges of a free-piston engine.

8.1 Step 1: Co-Simulation

In a first step the combustion process is simulated externally and the combustion thermodynamics are loaded into the free-piston linear alternator model (see Figure 5). In an iterative process the result is then used to re-simulate the combustion process until convergence. In that way the combustion process can be simulated in detail with a well validated programm. Due to unknown inlet and exhaust measures only the high-pressure part of the combustion cycle is investigated.

Figure 6 compares the piston velocity of a free-piston linear alternator in respect to a conventional crank shaft engine. As a result of the degrees of freedom of a free-piston alternator the piston velocity is a function of the system states.

The fundamental advantage of the proposed free-piston engine over a conventional engine is emphasized by Figure 7. In the left figure a comparison of the engines at full load conditions is displayed. Both processes show about the same performance. However, in part load conditions, shown in the right figure, the conventional engine keeps its stroke and compression ratio and the maximum pressure is quite low. Due to the variability of the free-piston linear alternator the stroke is lowered and the compression ratio is adjusted such that the cylinder pressure reaches a sufficiently high value. The performance in part load conditions of the free-piston linear alternator is consequently higher. This first approach shows the potential of a free-piston linear alternator in terms of combustion enhancement.

8.2 Step 2: Dynamic Simulation

A transient simulation is performed applying the Modelica model shown in Figure 2 and described in the sections above. Similar to many simulation tasks using DAE-solvers the free-piston linear alternator simulation needs solid start values. A solution to this task is to define a starting sequence where the linear alternator actively follows a fixed path for a few cycles before switching to a general operating mode. In the starting sequence the linear alternator simulates the piston motion of a crankshaft engine. During operation the system depends on the states in the cylinders and the energy converted by the linear alternator as well as the combustion process. The operation mode, e.g. the power output, can be changed by adjusting the system variables during the simulation process. The dynamic change in the system variables directly influence the operation mode. Figure 8 displays the change of operation mode from full load to part load conditions. The piston stroke is reduced and the compression ratio rises while the power output decreases as expected.

9 Conclusions

Examining the concept of a free-piston linear engine two main fields of interest for simulation are detected: Firstly, the development and testing of solid control of the free-piston system before applying it to hardware. On a second level the components involved in the system, namely the gas spring, the linear alternator and the combustion process can be studied in the free-piston context.
The dynamic simulation shows promising results. The system behavior as well as the cylinder conditions can be investigated, even when changing the operation mode. Developing control strategies is found to be a challenging task since solid piston control and an electronic camshaft are needed to ensure principle functioning of the free-piston assembly. Control models have been implemented and tested successfully.
Hardware components, such as valves and injectors, naturally have dead times which effect their reaction time. A predictive control to time the events in advance will be a focus of further development. Additionally, future efforts will be made to extend the free-piston linear alternator model in order to form a solid model to be built into a hybrid electric vehicle.
In order to implement a complex model of the free-piston linear alternator in the different levels of detail, subcomponents and components were modeled using Modelica. The system analysis was performed using Dymola. Both, the language formulation of Modelica and the powerful capabilities of Dymola were found to meet the expectations.

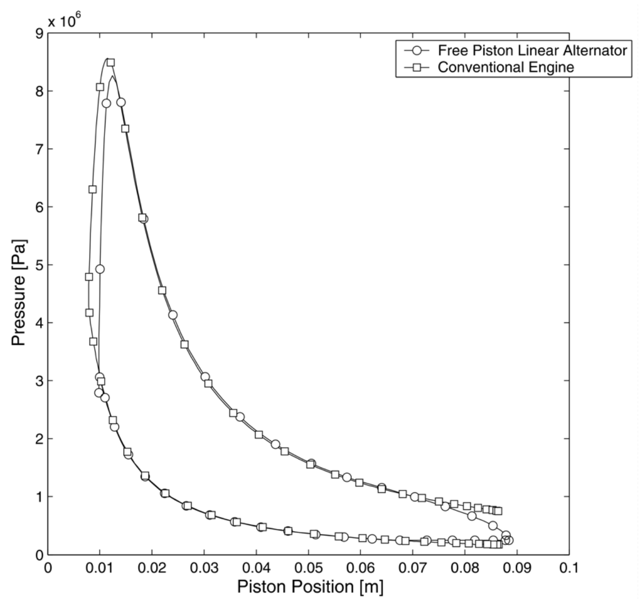
References

[1] Huber, K. *Der Wärmeübergang schnellaufender, direkt einspritzender Dieselmotoren*. Dissertation, Technische Universität München, 1990.

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

[3] Woschni, G. Beitrag zum Problem des Wandwärmeüberganges im Verbrennungsmotor. *MTZ*, 26, 1965.

[4] Zacharias, F. Mollier-I,S-Diagramme für Verbrennungsgase in der Datenverarbeitung. *MTZ*, 31(7), 1970.



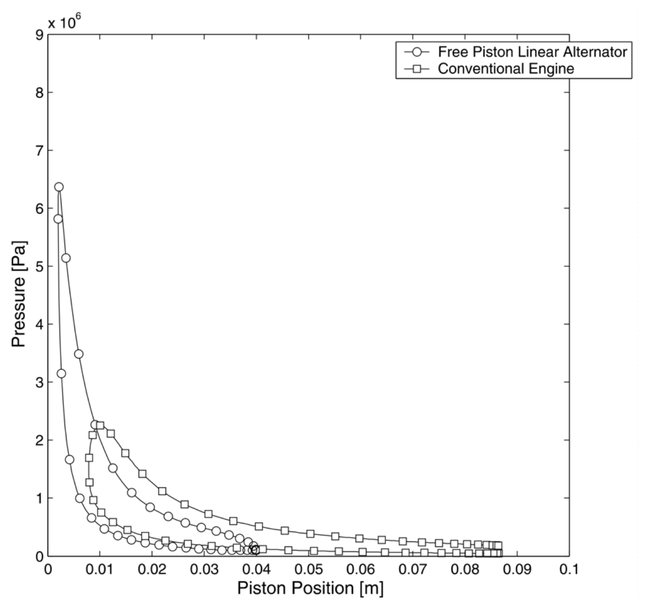
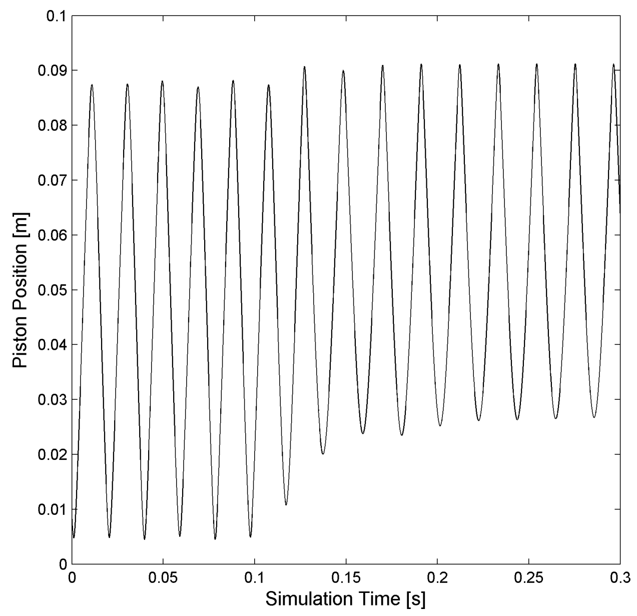


Figure 7: Comparison of combustion pressure at full (left figure) and part (right figure) load conditions for a crank shaft engine versus a free-piston linear alternator



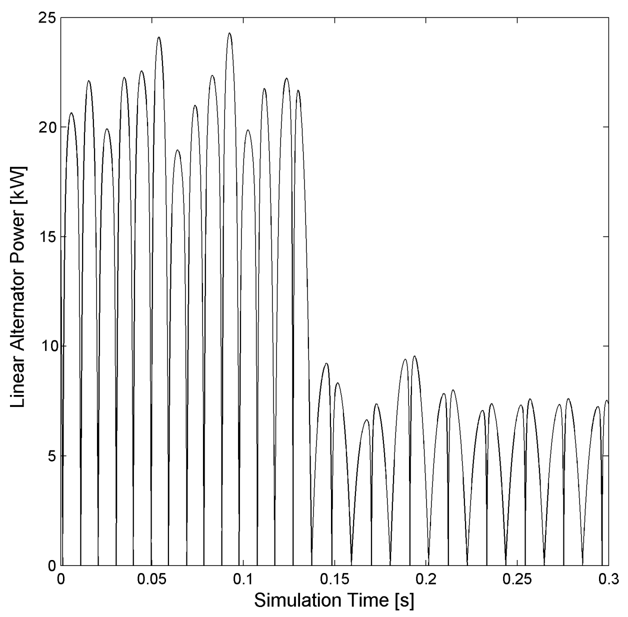


Figure 8: Piston position (left) and power output for dynamic Simulation

Session 5b

Thermodynamic Systems IV

Simulation of transient two-phase flow in parabolic trough collectors using Modelica

Dipl.-Ing. Tobias Hirsch   Dr.-Ing. Markus Eck   Dr.-Ing. Wolf-Dieter Steinmann
German Aerospace Center, Institute of Technical Thermodynamics
Pfaffenwaldring 38-40, 70569 Stuttgart

Abstract

Parabolic trough power plants are a very promising option for the generation of electricity from renewable energy resources. The Modelica library 'DissDyn' is developed to study the transient behaviour of the two-phase flow inside the absorber tubes of such plants. Equations are based on a homogeneous equilibrium model for the pipe flow with axial discretization. The advantages of manually transforming the equations into explicit state space form are shown. The models are validated with analytical solutions and measured data. Using test signals for disturbances in solar irradiation important information on the liquid loads on field separator and drainage system are gained. By adding feed-forward control schemes it is shown that these loads can be reduced significantly.

Keywords: solar thermal power plant; two-phase flow; simulation; renewable energy; absorber tube

1 Introduction

One way of generating electricity from renewable energy resources is the collection of solar radiation to be used in solar thermal power plants. Today's only commercially operated solar power plants in the Mojave Desert in California are based on a parabolic trough collector field. A synthetic oil is heated in the absorber tube and generates steam of 10 MPa/370°C in a heat exchanger unit. The steam is used to run a conventional steam turbine. Current research activities [1] are dealing with the direct solar steam (DISS) generation in the absorber tube itself. Improvements are expected by omitting the heat exchanger unit and allowing for higher steam temperatures thus leading to an increase in turbine efficiency. To avoid material problems caused by an undefined end of the evaporation section the collector loop is divided into two sections, see fig. 1. The evaporation section is run with a water surplus which has to be separated from the steam before the entrance into the superheating section. In the current design realized at a DISS test loop in Almeria (Spain) a maximum of 1.2 kg/s of superheated steam at 3 to 10 MPa and 400°C is obtained from one collector loop. For the first pre-commercial power plant with this technology the dynamic behaviour of the system determines the design of key components like compact field separators and the underlying drainage system. A dynamical simulation tool based on the Modelica language is developed to study the effect of irradiation disturbances and to evaluate different control strategies. For the assembly of the final system and the numerical integration the Dymola program is used.

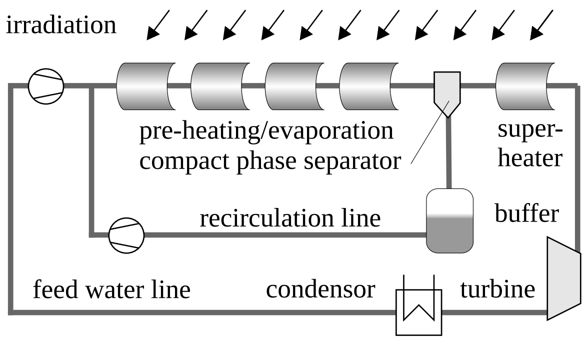


Figure 1: Configuration of a parabolic trough solar power plant with direct steam generation

2 Two-phase flow model equations

The central part of the model library is the fluid dynamical model of the two-phase flow in the absorber pipes. In the following the underlying assumptions and model equations are presented. For complex models it is advantageous and in this case necessary that the selection of state variables is done by the programmer and not by the symbolic transformation tool Dymola. It is shown that the combination of pressure and specific enthalpy is the best choice for the state variables of the fluid elements. The benefits of this manual selection of states and transformation of equations are

- stable and numerically efficient simulation
- independence of system assembly
- reliable initialization procedure
- well defined closure equations.

2.1 Conservation equations

Depending on the location along the collector loop different flow regimes are found in the absorber tube. Starting with single-phase liquid flow in the preheating part the flow changes into two-phase flow as soon as saturation conditions are reached. In the superheating section the flow is again single-phase. Superheating conditions also occur in the evaporator section when the end of the evaporation section drys out due to irradiation transients. Therefore the simulation model must be able to simulate the flow in the three regimes but also the transition between them. With a length of about 1000 m pressure losses in each loop are significant Fluctuations in irradiation cause large changes in mass flow and, as a consequence, in absolute pressure in the field. For this reason it is necessary to use pressure-dependent properties for the fluid. It is assumed that the flow is homogeneous over the pipe cross-section. In the two-phase region equal velocities and temperatures of water and steam phase are assumed (homogeneous equilibrium model). The simulation model is intended to study effects resulting from mass and energy transport which are much slower than the propagation of changes in pressure. Therefore infinite velocity of propagation is assumed for the pressure. This reduces the momentum equation to a stationary momentum balance for frictional pressure losses. The fundamental equations for conservation of mass, energy and momentum for the control volume shown in fig. 2.1 thus yield

$$\frac{\partial \rho}{\partial t} + \frac{\partial}{\partial z}(\rho w) = 0 \quad (1)$$

$$\frac{\partial}{\partial t}(\rho u) + \frac{\partial}{\partial z}(\rho w h) = \frac{\dot{Q}}{V} \quad (2)$$

$$\frac{\partial p}{\partial z} = \Delta p \quad . \quad (3)$$

The system is completed by the energy equation

$$A_W \rho_W c_W \frac{T_W}{\partial t} = \dot{Q}_{ext} - \dot{Q} \quad (4)$$

for the surrounding pipe wall.

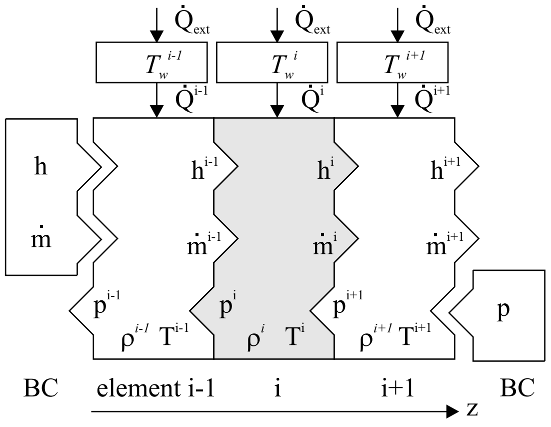


Figure 2: Three control volumes for pipe flow.

2.2 Selection of states

Two variables are sufficient to fully describe the state of the fluid element. The careful choice of the state variables is of essential importance since it determines the structure of the final system of equations. For this work the state variables pressure p and specific enthalpy h are chosen for the following reasons. All fluid properties can be expressed as a polynomial function of p and h. Using temperature T instead of h is not possible since temperature and pressure are directly linked in the two-phase region. The steam fraction $\dot{x}$ can not be used since it is not defined in the single phase regions. Friction losses cause a conversion of mechanical into thermal energy while the sum of both $h = u + p/\rho$ stays the same. By using specific enthalpy as a state, pressure loss terms can easily be defined by just changing the pressure and leaving h constant. The most important advantage of this selection is that spatially discretized systems will result in a set of de-coupled equations while a choice of e.g. mass flows and specific enthalpy leads to a coupled system. Using the fluid property functions

$$\rho = \rho(p, h) \quad (5)$$

$$u = u(p, h) \quad (6)$$

and average fluid velocity

$$w = \frac{\dot{m}}{0.25\,\pi\, d^2\, \rho} \quad (7)$$

the system of equations (1) to (2) can be rewritten explicit in the time derivatives

$$\frac{\partial h}{\partial t} = f_h\left(h,\, p,\, \dot{Q},\, \dot{m}\right) \quad (8)$$

$$\frac{\partial p}{\partial t} = f_p\left(h,\, p,\, \dot{Q},\, \dot{m}\right) \quad . \quad (9)$$

From these equations explicit state space form can be obtained by replacing $\dot{m}$ with an inverse pressure loss relation and $\dot{Q}$ with a correlation for heat transfer.

2.3 Fluid properties

Properties of fluids are defined for three regions, namely single phase water (region 1), two-phase water-steam (2) and single-phase steam (3) in the range 3 MPa $< p <$ 12 MPa and 100 °C $< T <$ 500 °C. Since state variables p and h are pre-selected all properties can be expressed as polynomial function of these variables. In the two-phase region the steam fraction x is defined as

$$x = \frac{h - h'}{h'' - h'} \quad . \tag{10}$$

with the specific enthalpies at saturation implemented in polynomial form

$$h'(p) = a_0 + a_1\, p + a_2\, p^2 + a_3\, p^3 \tag{11}$$

$$h''(p) = A_0 + A_1\, p + A_2\, p^2 \quad . \tag{12}$$

The density in liquid and gas phase is approximated by polynomials of pressure and enthalpy

$$\rho_l(p,h) = b_0(p) + b_1(p)\, h + \ldots + b_n(p)\, h^n \tag{13}$$

$$\rho_g(p,h) = B_0(p) + B_1(p)\, h + \ldots + B_n(p)\, h^n \tag{14}$$

with the coefficients

$$b_i(p) = b_{i,1} + b_{i,2}\, p + \ldots + b_{i,m}\, p^m \tag{15}$$

$$B_i(p) = B_{i,1} + B_{i,2}\, p + \ldots + B_{i,m}\, p^m \quad . \tag{16}$$

Note that for the homogeneous model the steam fraction x based on the control volume is the same as the steam quality $\dot{x}$ based on the mass flows of the two phases. The mixture density in the two-phase region is then given by

$$\rho(p,\, h) = \left(\frac{1}{\rho'} + x\left(\frac{1}{\rho''} - \frac{1}{\rho'}\right)\right)^{-1} \tag{17}$$

$$\rho' = \rho_l(p, h'(p)) \tag{18}$$

$$\rho'' = \rho_g(p, h''(p)) \quad . \tag{19}$$

This approach guarantees a continuous transition between the three regions. The calculation of temperature as function of pressure and enthalpy requires that for a fixed pressure exactly the same temperature is obtained when approaching the saturation line from the liquid region and the gas region. For this reason polynomial approximations are set up relativ to saturation conditions,

$$T_l(p,h) = T' + c_0(p) + c_1(p)\,(h - h') + \ldots \tag{20}$$

$$T_g(p,h) = T' + C_0(p) + C_1(p)\,(h - h'') + \ldots \tag{21}$$

For the calculation of the saturation temperature Antoine's law

$$T' = \frac{T_B}{\left(T_A - \log_{10}\left(\frac{p}{100}\right)\right)} - T_C \tag{22}$$

with the constants T_A= 8.1, T_B=1656.39, T_C=223.2 is used.

To complete the set of fluid property functions polynomial approximations for the dynamic viscosity, specific heat capacity, heat conductivity and surface tension have been derived for both phases. These quantities are needed for the calculation of pressure losses, and heat transfer coefficients. Table 1 gives an overview on the order of the polynomials and the accuracy achieved. High accuracy is desired for saturation enthalpies to reduce the errors in calculating very small steam fractions.

Table 1: Polynomial order of property functions and accuracy in the range 3...11 MPa, 500...3500 kJ/kg

Variable	Order in p	Order in h	Rel. error
h'	3		0.2%
h''	3		0.02%
ρ_l	3	2	2%
ρ_g	2	3	0.5%
η_l	2	4	1%
η_g	1	2	1%
λ_l	2	3	1%
λ_g	2	3	1%
$c_{p,l}$	2	3	1%
$c_{p,g}$	2	4	1%
T_l	1	3	0.5%
T_g	3	3	0.5%
$\sigma\,(T')$	3		1%

2.4 Spatial discretisation

Regarding the whole collector loop, mass flow and specific enthalpy at the inlet and pressure at the outlet are given as boundary conditions. For the spatial discretization of equations (8) and (9) an upwind scheme is applied for mass flow and specific enthalpy. Pressure losses are concentrated downstream of the control volume. Thus the equations yield for control volume i

$$\frac{\partial h^i}{\partial t} = f_h\left(h^i, h^{i-1},\, p^i,\, \dot{Q}^i,\, \dot{m}^i,\, \dot{m}^{i-1}\right) \tag{23}$$

$$\frac{\partial p^i}{\partial t} = f_p\left(h^i, h^{i-1},\, p^i,\, \dot{Q}^i,\, \dot{m}^i,\, \dot{m}^{i-1}\right) \quad . \tag{24}$$

2.5 Pressure loss

The mass flow terms in equations (23) and (24) have to be expressed as function of the state variables. This is realized by the inverse pressure loss relation

$$\dot{m}^i = \dot{m}^i \left(p^i,\, p^{i+1},\, h^i\right) \quad . \tag{25}$$

To avoid implicit equations a pressure loss correlation which can be solved analytically for $\dot{m}$ is used [2].

2.6 Heat transfer

The heat flux $\dot{Q}$ between wall and fluid is defined with the wall temperature T_W and the fluid temperature T in control volume i

$$\dot{Q}^i = \alpha^i \, \pi \, d \, l \left(T^i_\mathrm{W} - T^i\right) \quad . \tag{26}$$

The heat transfer coefficient α can be calculated in terms of the state variables

$$\alpha^i = \alpha^i \left(h^i,\, p^i,\, \dot{m}^i\right) \tag{27}$$

with $\dot{m}^i$ given in equation (25).

Replacing mass flow and heat flow terms in equations (23) and (24) the final simulation equations in explicit state space formulation are obtained

$$\frac{\partial h^i}{\partial t} = f_\mathrm{h}\left(h^i, h^{i-1},\, p^i,\, p^{i-1},\, p^{i+1},\, T^i_\mathrm{W}\right) \tag{28}$$
$$\frac{\partial p^i}{\partial t} = f_\mathrm{p}\left(h^i, h^{i-1},\, p^i,\, p^{i-1},\, p^{i+1},\, T^i_\mathrm{W}\right) \quad . \tag{29}$$

3 The DissDyn library structure

The library contains fluid-dynamic models for two-phase flow in heated pipes as well as models for components like tanks, phase separators, valves and pumps. Models are added to convert the direct normal irradiation into a heat flux on each absorber section.

3.1 Connector definitions

There are four different types of connectors currently defined in the library:

`MassFlow` ≀		
`m_dot`	[kg/s]	mass flow
`h`	[J/kg]	specific enthalpy
`p`	[Pa]	pressure
`AmbData` □		
`t_amb`	[°C]	ambient temp.
`v_wind`	[m/s]	wind speed
`alpha_wind`	[deg]	wind direction
`SolarIrr` ◇		
`altitude`	[deg]	altitude angle
`azimuth`	[deg]	azimuth angle
`I_Dir`	[W/m^2]	direct irradiation
`I_Diff`	[W/m^2]	diffuse irradiation
`ParabolIrr` ○		
`cosphi(n)`	[-]	cos(ϕ)
`alpha_tr(n)`	[deg]	track angle
`I_Dir(n)`	[W/m^2]	direct irradiation

This list is completed by the standard Modelica `Signal` connector. All fluid-dynamic components can be linked using the same connector `MassFlow`. This connector is based on the definitions in the *TechThermo* library developed at the institute [5]. The other three connectors are used to transfer information on solar irradiation and related quantities like incident angles.

3.2 Solar irradiation models

The transformation of direct irradiation coming from the sun into heat flux on the absorber tubes can be subdivided into three steps as illustrated in fig. 3. In the `SolarIrradiation` model altitude and azimuth angles of sun position are calculated based on the day of the year, time of day and geographical latitude and altitude. This component is prepared for the implementation of functions predicting the intensity of direct as well as diffuse solar irradiation throughout the day. At present, the magnitude of direct irradiation is specified by an external signal source.

In the second stage the irradiation finally reaching the individual collector is reduced by cloud coverage or by taking the collector out of focus. In the model `ParabolicField` the position and orientation of all collectors in the field is stored which is used to calculate the optimum track angle and the resulting incident angle for each collector individually. The magnitude of direct irradiation specified via the `SolarIrr` con-

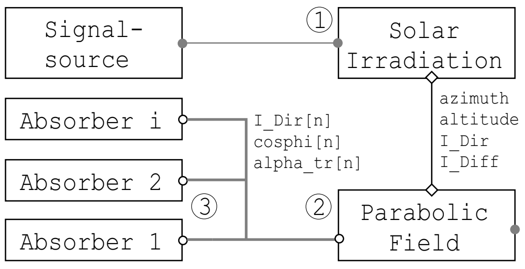


Figure 3: DissDyn-components used to model the path of solar energy from sun to absorber tubes

nector is reduced by a cloud coverage and focus information of the collector. The focus signal of each collector (range 0 to 1) can be read via the Modelica signal connector. Varying cloud coverage is represented using an one-dimensional cloud coverage signal that can be moved with arbitrary speed and direction over the collector field as shown in fig. 4

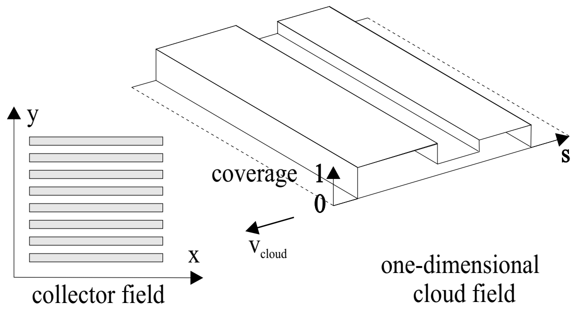


Figure 4: Cloud coverage

The final stage is the transformation of irradiation reaching the collector into the effective heat flux on the absorber tube. This task is implemented in the `absorber` model where efficiency data for the individual collectors are stored.

3.3 Absorber and pipe models

The model equations for two-phase flow in pipes presented in section 2 are used to construct models for simple pipe flow and for the flow in absorber pipes of parabolic trough collectors. The spatial discretization can be defined by the number *n* of axial elements along the pipe section. This is shown schematically in fig 5. For the computation of heat losses the ambient temperature is given via the `AmbData`-connector. The `absorber` models have another connector `ParabolIrr` by which information on actual irradiation, incident angle and theoretical track angle is provided. Since these values are constant along one collector each `absorber` model is intended to

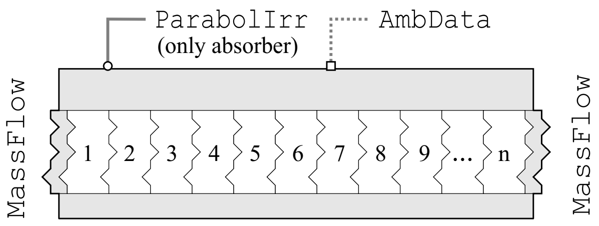


Figure 5: Structure of `absorber` and `pipe` model

represent one collector. A row of collectors is composed of a number of identical collectors and their interconnecting pipes. For convenience reasons a model `collector_Row` is defined which holds a set of `absorber` and `pipe` models connected in series. All relevant geometrical parameters for the `absorber` and `pipe` model can be defined in this top level model. The irradiation and ambient data information is passed to each component as depicted in fig. 6.

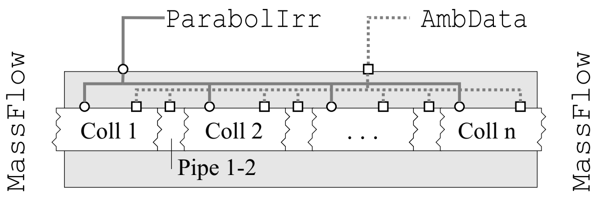


Figure 6: `Collector_Row` model as a collection of `absorber` and `pipe` models

3.4 Fluid system components

The set of fluid models is completed by fluid system components like

- phase separator
- T-junction for flow combination and distribution
- control valve
- pump
- tank .

Except of the phase separator model all of these components are based on stationary conservation of mass and energy.

3.5 Control system components

To implement control functionality Modelica control models and specialized models are used. These models are linked by Modelica `Signal` connectors. Although these models can in principle be constructed from Standard Modelica blocks new models are defined in order to have direct access to all relevant variables within one model.

4 Validation

The validation of the fluid dynamic models is done in two steps. First the simulation results with different spatial discretization are compared with analytical solutions available under special assumptions like constant pressure. Since these assumptions are not valid in the real system a direct comparison with measured data from the DISS test loop is needed in the second step.

4.1 Analytical models

Under some assumptions the set of conservation equations can be solved analytically by Laplace transformations [3], [4]. Using the same simplifications for the simulation the correct implementation and consistence of the simulation model can be checked.

Fig. 7 shows a comparison of transfer function in terms of amplitude and phase response for a 20 m absorber pipe section under two-phase conditions. For this comparison the reaction to a 1% step in irradiation with a spatial resolution of 1 element and 20 elements have been simulated. The corresponding transfer function is derived by Laplace transformation of the change in specific enthalpy at the outlet. While the simulation with just one element has large deviations from the analytical solution a much better agreement is obtained with high resolution of 20 elements indicating the consistence of the simulation model.

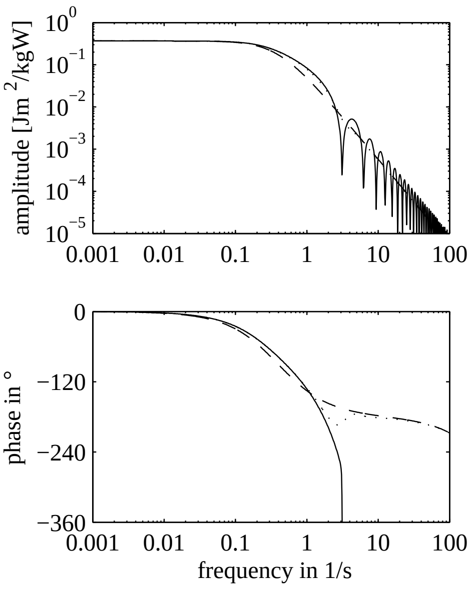


Figure 7: Transfer function of a 20 m two-phase absorber section: calculated with analytical model (-), simulated with discretization 1 element (- -), 10 elements (···)

Comparison in the time domain plottet in fig. 8 shows that for high resolution nearly exact agreement is obtained. This means that the remaining deviations in

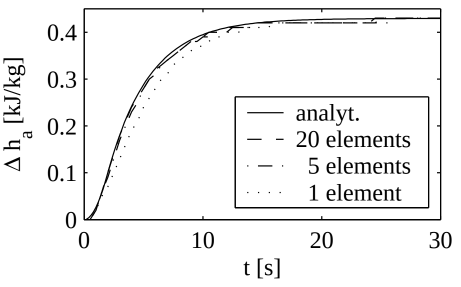


Figure 8: Step response to a 1% change in irradiation for a 20 m two-phase flow absorber

amplitude and phase angle at frequencies of more than 1/s are not serious for the system since the contribution of these frequencies is very small. Although the resolution with 20 elements gives nearly exact agreement the simulation with 5 elements is also very close to the analytical solution.

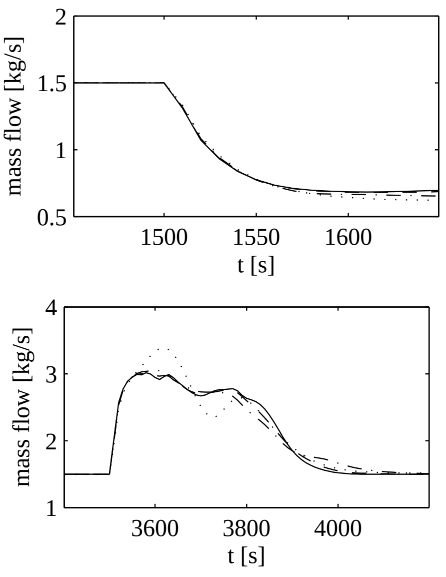


Figure 9: Simulated step response to changes in irradiation of -500 W/m^2 at t=1500 s and +500 W/m^2 at t=3500 s. 100 m evaporator section with resolution of 2.5 m (-), 10 m (- -), 33 m (-.-), 100 m (. .)

In Fig. 9 simulation results are shown for a 100 m evaporator section with large changes in irradiation.

These results are obtained without the simplifying assumption of constant pressure. The curves show that the result with the coarse resolution of 10 m is quite close to the one with fine resolution. Comparisons are preformed for the preheating and superheating section as well giving similar results.

4.2 Comparison with measured data

Especially the assumption of constant pressure along the absorber is not fulfilled if multiple collectors are connected in series. The comparison with the analytical models on the basis of constant pressure is therefore not sufficient to validate the model. For this task a direct comparison with measured data from the DISS test loop is performed for several test days. As an example the reaction of the 500 m collector row (425 m preheater/evaporator, 75 m superheater) to irradiation fluctuations is shown in fig 10 in terms of inlet pressure, steam mass flow and steam temperature.
Though there is a small deviation in the absolute value of inlet pressure the dynamic behaviour is well predicted. Steam production and temperature are in good agreement which is also observed in the other test configurations. Both the check with analytical models and the direct comparison with measured data show that the model assumptions, especially the homogeneous equilibrium model are valid for this application.

5 Transient simulation of a collector loop

The simulation model is used to simulate the reaction of a collector loop to changes in irradiation. Since the dynamical behaviour is dominated by the amount of liquid evaporated, the 800 m boiler section alone is analyzed in the first step with the superheater replaced by an adequate pressure loss term. Constant boundary conditions for feed water flow and enthalpy as well as recirculation flow and enthalpy are imposed at the collector inlet. At the outlet of the field the pressure is fixed at 7 MPa.
Using measured irradiation data as input for the simulation is not useful when looking on general system behaviour. Moreover comparison of results with other researchers requires the same set of input data. This is avoided if a simple test signal is chosen which represents the main characteristics of real irradiation disturbances. In this work three single disturbances in series are used as a test signal, see fig. 11. This signal is defined only by the two parameters interval length Δt and change of intensity ΔI. The time constant of the collector system is closely linked to the residence time in the boiler section. Since the residence time for an 800 m boiler is approximately 700 s a series of short disturbances provokes an overlap of the system reactions. By using the test signal with three ramps the effects of this overlap can be studied. On the other hand, if long intervals are chosen a single isolated disturbance can be analyzed.

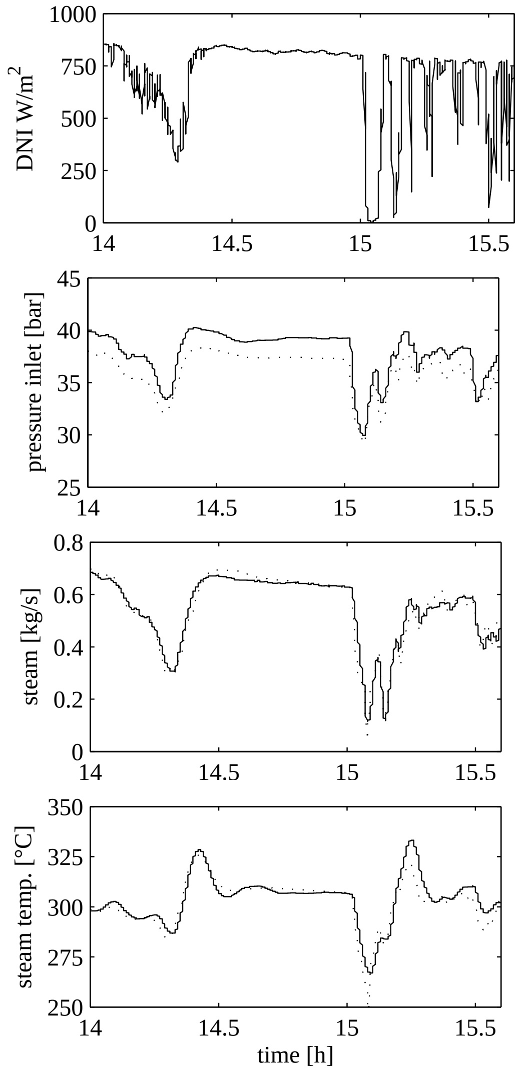


Figure 10: Measured (-) and simulated ($\cdots$) DISS plant operation on June 26, 2001. Spatial Resolution 10 m.

5.1 Simultaneous disturbance on all collectors

Fig. 12 shows the simulation results in terms of steam and water mass flow at the exit of the evaporator section. Irradiation disturbances act simultaneously on

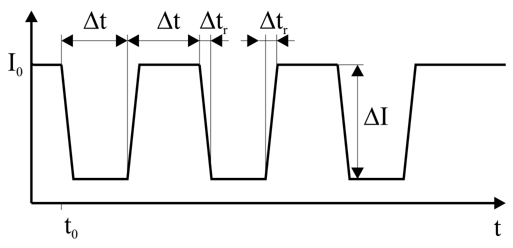


Figure 11: Test signal of three consecutive irradiation disturbances.

all 8 collectors of the boiler. Three studies with three different interval lengths are carried out. The intensity for short interval length is 100%, for long intervals only 70% in order to avoid reverse flow in the pipes. From the steam signal it can be seen that it takes about 700 s after the last disturbance to reach again stationary conditions. With a maximum interval length of 240 s all curves represent a superposition of individual reactions.

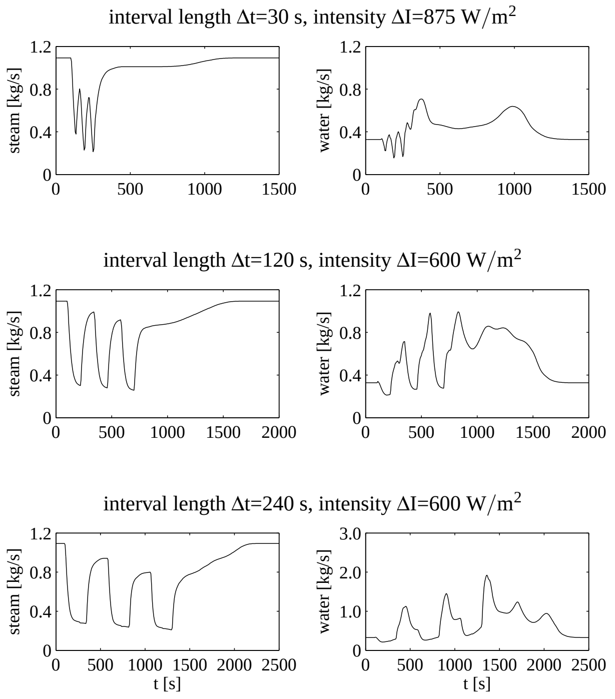


Figure 12: Reaction of the 800 m evaporator section to simultaneous irradiation disturbances on collectors 1-8 according to fig. 11. Plotted are steam and water mass flow at the exit of the evaporator. Feed water 1.2 kg/s, recirculation 0.25 kg/s, irradiation 875 W/m^2, outlet pressure 7 MPa.

The water flow shows large peaks with the maxima located short time after the rising edge of the disturbance. Since constant feed flow is injected at the entrance of the collector water accumulates in the system while irradiation is reduced. With increasing steam production this additional water is blown out and causes temporarily high liquid mass flux. In the Δt=30 s case two peaks appear in the water mass flow signal. The first one immediately after irradiation increases and the second one with a delay of about 700 s just before stationary conditions are reached. While for short intervals (Δt=30 s, 120 s) the second peaks of the three disturbances overlap and form one large peak the single peaks can be identified for longer interval length (Δt=240 s). The maximum in liquid flow reached in this configuration is about 2.5 kg/s which is 8 times the nominal value. This results are very important to define the operating conditions for the layout of the compact field separator and the underlying drainage system.

5.2 Local disturbances

Small clouds can cause a local shading of just a number of collectors. Simulations are performed with assuming local shading of two collectors at a time. The same test signal is used as before. An additional configuration with a recirculation mass flow of 1.0 kg/s instead of 0.25 kg/s is simulated to estimate the impact of recirculation flow on the system dynamics.
Fig. 13 shows the results for a local shading of the preheating section (collectors 1 and 2). From the steam

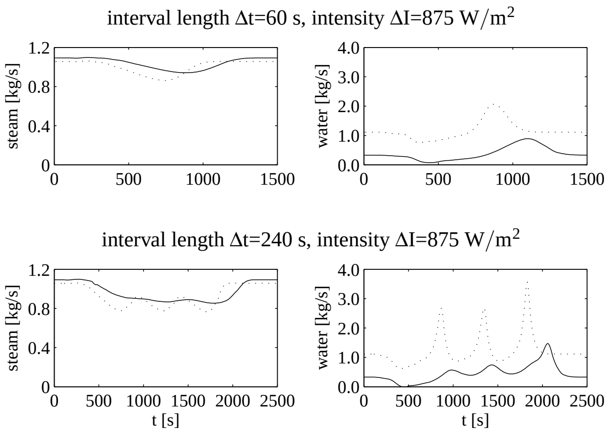


Figure 13: Reaction of 800 m evaporator section to local irradiation disturbance on collectors 1/2 according to fig. 11. Plotted are steam and water mass flow at the exit of the evaporator. Feed water 1.2 kg/s, recirculation 0.25 kg/s (-) / 1.0 kg/s (···), irradiation 875 W/m^2, outlet pressure 7 MPa.

production it can be seen that the system reacts very slow. Since the fluid velocity is very small it takes a long time before the change in specific enthalpy leads to a significant change in steam production. With increased recirculation flow the residence time in the preheating section gets smaller and reactions become faster and more distinct. This effect can be seen clearly in the water flow in the case Δt=240 s. Although the integral value of liquid pushed out of the absorber is the same for both recirculation flows the temporal distribution differs. Since evaporation starts further downstream when collectors 1 and 2 are shaded liquid is stored in the system. This leads to a temporarily reduced liquid mass flow in the rest of the evaporator and, as a consequence, to a short period of dryout at t=500 s. By increasing the recirculation flow the danger of superheating at the end of the evaporator is reduced. On the other hand much higher liquid peaks have to be accepted.

If the local shading is concentrated on collectors 4 and 5, see fig. 14, the reaction becomes faster and the peaks higher. At long intervals there is only weak

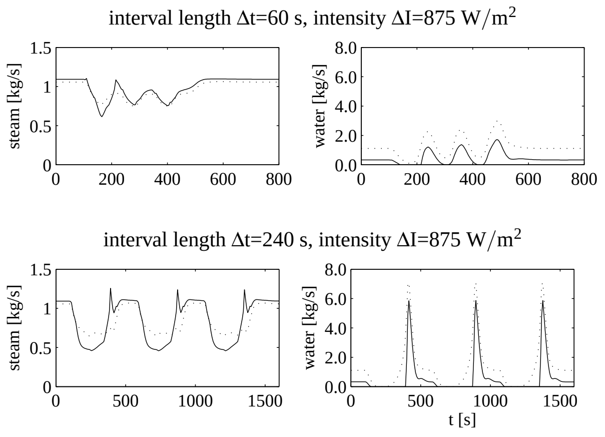


Figure 14: Reaction of 800 m evaporator section to local irradiation disturbance on collectors 4/5 according to fig. 11. Plotted are steam and water mass flow at the exit of the evaporator. Feed water 1.2 kg/s, recirculation 0.25 kg/s (-) / 1.0 kg/s (···), irradiation 875 W/m^2, outlet pressure 7 MPa.

overlapping of the single reactions. Compared with the shading of the pre-heater the reaction time now depends on the residence time in the two-phase region which is much shorter. Since the whole evaporator is nearly at the same temperature a change in steam production requires no heating or cooling of the absorber tube walls. The difference between small and high recirculation is small since recirculation mass flow has only minor impact on the average velocity. Compared to shading of collectors 1/2 the maximum liquid mass flow has nearly doubled and the dryout effect is much more critical. Even increasing the recirculation mass flow can not avoid long periods of dryout.

Lokal Shading at the end of the evaporator in collectors 7/8, see fig. 15 is characterized by very small reaction time. The recirculation mass flow has no effect on the result. Water mass flow reaches its steady-state values very fast with nearly no dynamical overshooting.

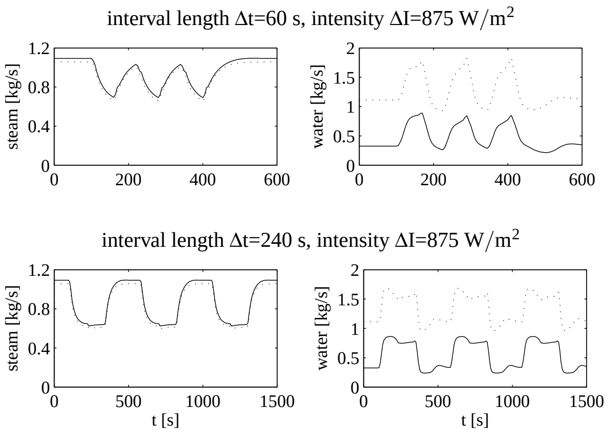


Figure 15: Reaction of 800 m evaporator section to local irradiation disturbance on collectors 7/8 according to fig. 11. Plotted are steam and water mass flow at the exit of evaporator. Feed water 1.2 kg/s, recirculation 0.25 kg/s (-) / 1.0 kg/s (···), irradiation 875 W/m^2, outlet pressure 7 MPa.

6 Combination with control system

A central task of the control system is to inject as much feed water as can be evaporated according to the actual irradiation. A standard feed-back control loop based on a liquid level control in the buffer tank reacts very slow to changes in irradiation. A much faster reaction is possible if measured irradiation is used to calculate the necessary feed flow. Another method is to measure the actual steam production and use this signal for the feed water pump. Due to model errors both approaches are not able to reach a specified set point in the buffer level so in any case an additional level controller is necessary. To analyze the potential of these forward control schemes simulations have been performed. Fig. 16 shows the results for a configuration of an uncontrolled system, a configuration with feed-forward control based on the irradiation measurement and a configuration with feed-forward control based on the steam production.

From the feed water signal the time lag between irradiation based and steam production based approach can be seen. For liquid and gas mass flow at the evaporator outlet this lag has nearly no effect. The steam production gets more continuous with the feed-forward control. There is a significant reduction of maximum liquid peaks and as a consequence in buffer mass compared to the uncontrolled system. The reason is that

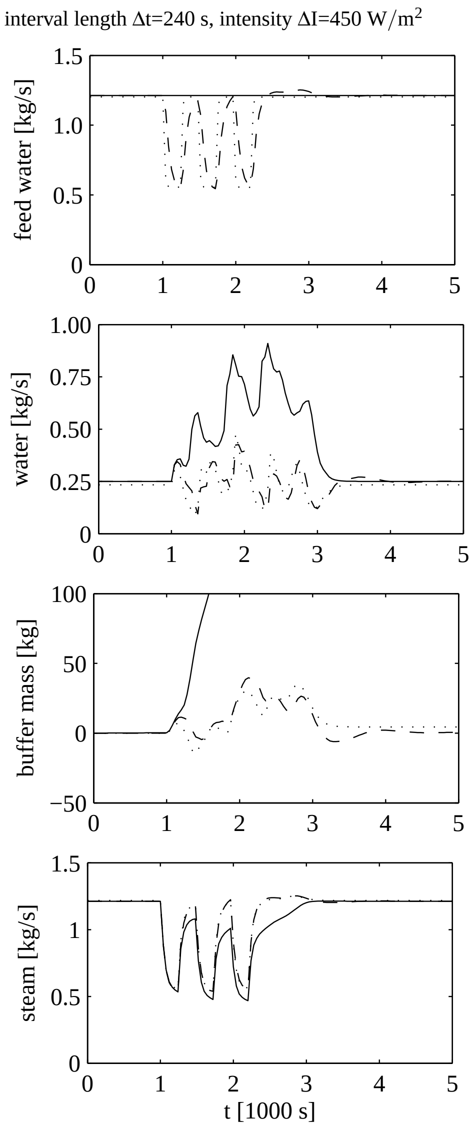


Figure 16: Comparison of uncontrolled (-) system with feed-forward control based on irradiation signal (···) and steam production (-.-). Recirculation 0.25 kg/s, DNI 900 W/m^2.

less water accumulates in the pipes that has to be pushed out when irradiation again increases. In the presented disturbance configuration both feed-forward schemes give similar results. If local shading of the pre-heating section is used the irradiation based approach reacts much faster then steam production changes. This leads to a contradictionary effect for the buffer level. As a preliminary result it can be stated that the feed-forward control has shown high potential in reducing the necessary buffer size without significant negative side-effects.

7 Conclusions

A Modelica library for two-phase flow in parabolic trough collectors is developed and successfully validated against analytical models and experimental data. For the transient simulation of these complex system it is necessary to manually select the state variables and to transform the equations into explicit state space form. Only with this approach it is guaranteed that initialization and numerical integration are reliable without loosing much of the flexibility the Modelica language offers. The library is used at the institute to simulate the reaction of parabolic trough collector loops with direct steam generation to fluctuations in solar irradiation.

References

[1] ECK M., STEINMANN W-D. Direct solar steam generation in parabolic troughs: First results of the DISS project. In: Journal of Solar Energy Engineering (124), pp. 134-139, 2002.

[2] MÜLLER-STEINHAGEN H., HECK K. A simple friction pressure drop correlation for two-phase flow in pipes. In: Chem. Eng. Process.(20), pp. 297-308, 1986.

[3] PROFOS P. Die Regelung von Dampfanlagen. Berlin, Springer, 1962.

[4] ECK M. Die Dynamik der solaren Direktverdampfung und Überhitzung in Parabolrinnenkollektoren. VDI-Fortschrittsberichte, Reihe 6, Nr. 464. Düsseldorf: VDI-Verlag, 2001.

[5] STEINMANN W.D., ZUNFT S. TechThermo - A library for Modelica Applications in Technical Thermodynamics. In: Proceedings of the 2nd International Modelica Conference 2002, Oberpfaffenhofen, Germany, Modelica Association, 18-19 March 2002.

Modelling and Simulation of Central Receiver Solar Thermal Power Plants

L.J. Yebra[†,*] M. Berenguel[*] S. Dormido[‡] M. Romero[†]

[†]CIEMAT-PSA. Ctra. de Senés s/n. Tabernas.
E04200 Almería. Spain. E-mail:{luis.yebra,manuel.romero}@psa.es

[*]Universidad de Almería. Dpto. de Lenguajes y Computación. Ctra. de Sacramento s/n.
La Cañada. E04120 Almería. Spain. E-mail: beren@ual.es

[‡]U.N.E.D. Escuela Técnica Superior de Ingeniería Informática. Dpto. Informática y Automática.
C/ Juan del Rosal, 16. E28040 Madrid. Spain. E-mail: sdormido@dia.uned.es

Abstract

The implementation of advanced control systems to optimize the overall performance of Central Receiver Solar Thermal Power Plants is nowadays a priority research line in CIEMAT-PSA. The development of dynamic models for use in simulation and control of this kind of power plants is presented in this article, focused on the CESA-I solar plant. The developed model is based in the thermohydraulic modelling framework ThermoFluid, and the main components of the system are presented as well as the respective modelling assumptions. A typical operating cycle is simulated and the results are shown and commented.

1 Introduction

This paper presents the current status of the research performed within the framework of modelling and simulation of Central Receiver Solar Thermal Power Plants (CRSTPP). The work is mainly oriented to the development of dynamic models of solar energy plants to be used in the design of automatic control systems aimed at optimizing global performance. The models presented in this article are focused on the solar part, excluding typical components of power plants like turbines or generators.
The system used as test-bed plant is the CESA-I facility, a CRSTPP belonging to the CIEMAT (Centro de Investigaciones Energéticas, Medioambientales y Tecnológicas - *Research Centre for Energy, Environment and Technology*), public organism owned by the Spanish Ministry of Science and Education. This solar plant is located at the Plataforma Solar de Almería (PSA), South-East Spain. A join project between CIEMAT-PSA, the University of Almería (UAL), the National University of Distance Education (UNED) and the University of Seville (US) is being carried out in order to develop models and control systems to automatically control these kind of plants. This test-bed plant can be seen in figure 1.

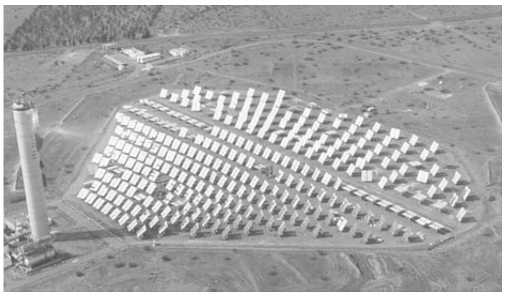


Figure 1: CESA-I solar plant at PSA

The model presented in this paper will be used in the design of hybrid model predictive control and intelligent control schemes to optimize plant performance, even under start-up and shutdown situations and in the presence of highly variable load disturbances due to the daily cycle of solar radiation and passing clouds.

2 Central Receiver Solar Thermal Power Plants

In this section an overview of the basic components and operating procedures for a CRSTPP is introduced. Figure 2 shows an explicative diagram of a general CRSTPP.

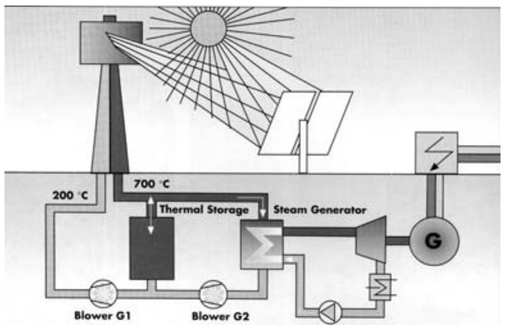


Figure 2: Schematic diagram of a CRSTPP

The operation of this kind of plants is based in the concentration of incoming solar energy using a heliostat field that reflects the incident solar radiation onto a (typically volumetric) receiver (theoretically onto an optical point in the 3-D space). As the sun position changes during the day, each heliostat of the field (composed of 300 heliostat in the case of the CESA-I plant) has to change its position in real time according to the selected aiming point on the receiver, as different aiming points can be selected in order to achieve a uniform temperature distribution on the receiver [7]. The receiver is located at the top of the tower (84 m height in CESA-I) and acts as an energy exchanger, receiving solar energy and transferring it to a thermohydraulic circuit with air medium, see figure 2. The system is also composed by an energy storage tank, an air/water-steam heat exchanger (evaporator), blowers and valves. The combined action of the blowers let to feed either the storage tank or the heat exchanger with hot air. The evaporator is formed by the primary circuit and a secondary one with subcooled inlet water and with superheated steam outlet. A measurement of the overall concentrated input radiation, a controlled water pump and an outlet controlled valve define the main boundary conditions for the system. The final objective of the model is to predict the transient behaviour of the thermodynamics variables associated to the thermohydraulic output power of the evaporator (mainly temperature, pressure, and specific enthalpy), when the external disturbances (concentrated solar radiation, ambient temperature, and wind speed and direction) and controllable inputs (both at the solar field level: number of operating heliostats and aiming point strategy followed, and at the tower level: mass flow rate demanded by the blowers, inlet water flow and outlet pressure valve position) change.

3 Modelling of CRSTPP

In this paper we will concentrate in the thermohydraulic part of the system, skipping the heliostat field and assuming a known input radiation power in the receiver as a consequence of the radiation reflected in the heliostat mirrors and the aiming-point strategy followed [7]. This assumption introduces an error in the estimated irradiation due to the difficulties that exist in getting accurate high concentrated radiation measurements.

Due to the fact that the main phenomena are related to thermofluids, the Modelica language has been used to develop these models including the ThermoFluid library ([12],[6]) as a framework over which create own libraries and final component models. The authors think that this library is an important reference in the framework of object oriented modelling of thermofluid systems with Modelica, and its existence makes a nonsense to develop libraries in the same level of modelling of thermohydralic systems.

The work analyzes each of the components of the thermohydraulic circuits of air and water-steam and explain the modelling assumptions, trying to justify each one as they are oriented to get, by means of the symbolic manipulations that Dymola tool performs, a not high index DAE system for the complete model, in which the number of nonlinear algebraic loops is minimized. For this purpose, all the components are classified, following the modelling methodology derived from the Finite Volume Method (FVM) [10], in Control Volumes (CV in ThermoFluid nomenclature) and Flow Models (FM in ThermoFluid nomenclature). In some cases information about the future control system architecture to be implemented is introduced in the modelling phase. An example of components that are modelled using this kind of information are the blowers in the air circuit, in which a cascade control will help avoid the multivariable nonlinear dependence of the constitutive equations and let consider them like quasi ideal flow rate generators. Due to the existence of components whose internal implementation may vary depending on the modelling hypotheses, the polymorphism and the Modelica language constructs *replaceable/redeclare* have been specially used in some of them, for example in the evaporator.

Figure 3 shows the developed Modelica model of the system.
The following components are shown: blowers, storage tank, solar receiver, evaporator, control valve, sensors, pipes and connections. All of them are directly instantiated and parameterized, or inherited from ThermoFluid classes. It can be seen how the air circuit is composed of a solar receiver, tubes, tank, and evaporator; and the water steam circuit with a water pump, a control valve, tubes, and reservoirs with boundary conditions. In both circuits the mass transfer connections have been drawn with solid and thick lines, while colors are used to describe *hot* fluid (red), *cold* fluid (blue), and hot steam (grey). Input-Output causal connectors appear to access variables of interest for control purposes like:

Output variables. Boolean indicator for saturated/superheated vapor, several pressures and temperatures in both media circuits, stored energy in the tank, mass flow rate in the tank, mean temperature in the tank, etc.

Input variables. Actuators setpoints: blowers 1 and 2, water pump; and boundary conditions: pressure and temperatures of water inlet, estimated inlet concentrated radiation, ambient temperature and pressure, wind velocity.

3.1 ThermoFluid usage

The thermohydraulic interface is formed by connectors from ThermoFluid for single component media and steady-state momentum balance statement. Due to the fact that the dynamics of momentum balance are of no interest for control purposes in the time scales studied at present, the stationary formulation version of ThermoFluid has been used in all the thermohydraulic classes utilized.
The modelling methodology adopted from the beginning for the design of the classes was: *if there exists any class in ThermoFluid that implements the physical phenomenon to model, use it with the corresponding parameters; if not, design the classes using inheritance from the high level partial classes from ThermoFluid; in other cases then use proper ThermoFluid interfaces and base classes and develop the component class with the lacking behavior expressed in differential and algebraic equations from first principles.*
In the next subsections the most important components models will be detailed and the modelling hypotheses will be explained and justified.

3.2 Blowers and Pumps

In this kind of active FM [12], the authors decided to make a simplifying assumption based on the gained experience in control of Parabolic Trough Fields with thermal oil as medium, case of Acurex field of CIEMAT-PSA [4], [5], and water-steam as medium in DISS facility [16], [13]. This assumption is that the water pump and blowers are controlled in a cascade scheme [3] with a local control loop whose dynamics is much faster than the rest of the thermohydraulic system. This assumption has been experimentally validated in blowers and water pump, and helps simplifying these components models until the possibility of state them as quasi-ideal mass flow rate generators. This approximation lets avoiding the consuming time work of fitting the nonlinear multivariate curves of the pumps and blowers. So, the algebraic equation for these components is $\dot{m} = \dot{m}_{ref}$, where $\dot{m}_{ref}$ is the setpoint of the local pump/blower control loop and is assigned in a connector to the model, as can be seen in figure 3.

3.3 Solar Receiver

This component receives energy from the heliostat field, that concentrates solar radiation in different aiming points to avoid large spatial temperature gradients that could damage the component. This aiming point strategy is a research line at present, [7], and is implemented in the heliostat field real time control system. The total energy flow reflected by the heliostats and concentrated in the receiver is nowadays estimated, because the practical difficulties in the measurement of this variable. Therefore, a mean solar concentrated input power is used for modelling purposes. This assumption introduces uncertainty in the model, that makes not to expect from the adjusted and validated model a precision beyond that of the inlet power. Nevertheless, it is expected that this uncertainty will decrease in the near future with the development and implementation of new sensors and by re-calibrating the model with new operating data.
Figure 4 shows the Modelica model composed, in a major number, of ThermoFluid components. This is the model of the system in which the strongest simplifications have been made, due to the internal design of the system. From a system level point of view, the receivers are composed of an arrangement of solid ceramic cups that receive inlet radiation. Due to hydraulic depression caused by external pumping elements (blowers 1 and 2), an air mass flow rate from

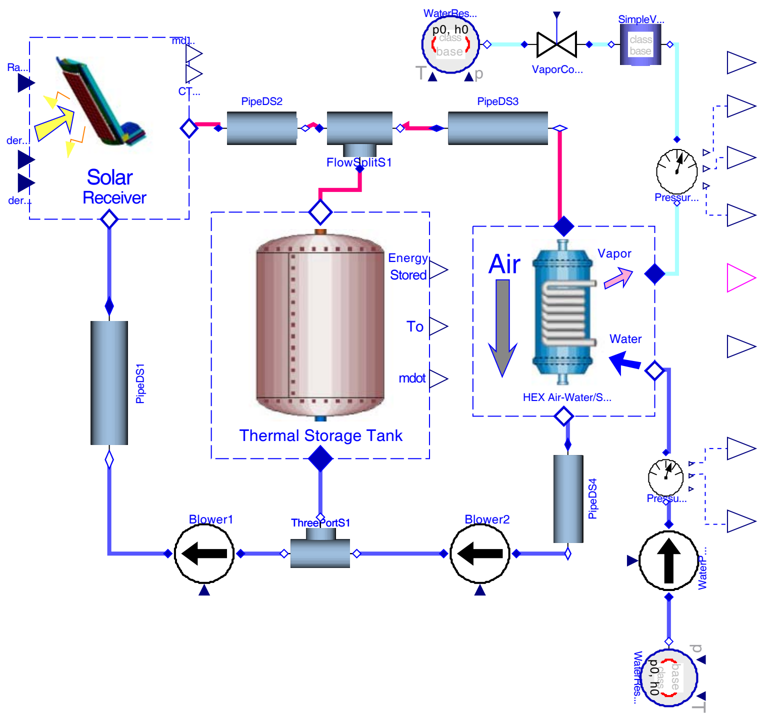


Figure 3: Modelica model of CESA-I CRSTPP

the environment is established and heated when passing through the ceramic cups. The cups are modelled by wall classes, the air mass and energy conservation by concentrated parameters CV's and the medium model for air from medium models. The external boundary conditions are modelled by reservoir componentes representing external pressure and temperature. The conduction-convection and radiation energy flow from receiver to atmosphere is modelled by heat transfer classes with the extensions of include expressions for empirical correlations obtained by experimentation. The convection heat transfer between the CV and the cup array is based in empirical correlations too. The radiation and convection are modelled using the Stefan-Boltzmann Law and Newton Cooling Law [9], in which the radiation conductance in the former case, and the heat transfer coefficient, in the latest, are the variables to identify by means of adjusted correlations. The three input connectors represent: concentrated solar radiation in the receiver (*radin*), derivative of atmospheric pressure and derivative of ambient temperature, respectively. The former is the estimated concentrated solar radiation, and the second and third are used as sources of uncertainties to the model. The derivative of the pressure lets simulate experimented effects that wind gusts could cause in the pressure and temperature close to the receiver. The explicit modelling of disturbances caused by wind are important

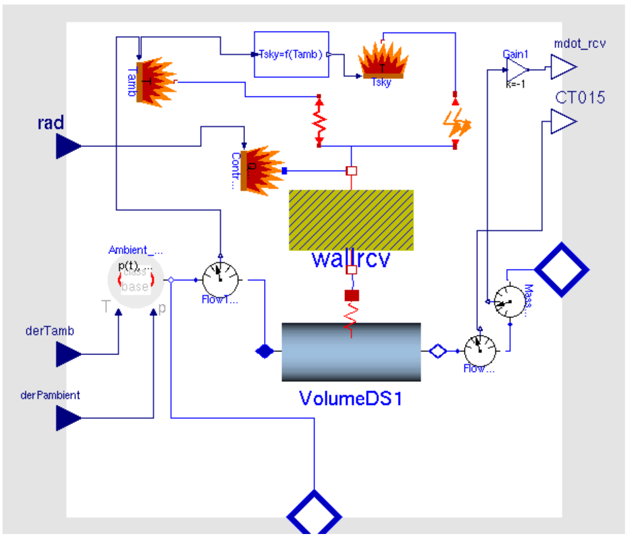


Figure 4: Modelica model of CESA-I receiver

for control purposes of the temperature of the air leaving the receiver to the circuit.

3.4 Storage Tank

The aim of the storage tank is to accumulate energy to let the plant operate when irradiation decreases during a time interval (limited by the tank capacity and layout). The tank can be defined by three states, depending on the mass flow rates of the blowers, see figure

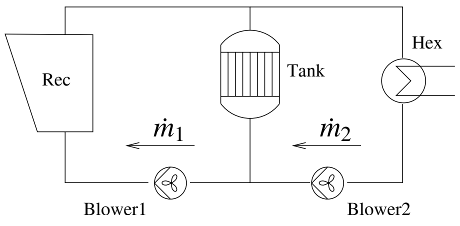


Figure 5: Schematic diagram of the air circuit

5, and assuming enough incoming power from the receiver and negligible energy losses to the environment from the tank:

Loading: $\dot{m}_1 > \dot{m}_2$. The energy stored in the tank is increasing.

Unloading: $\dot{m}_1 < \dot{m}_2$. The energy stored in the tank is decreasing.

Standby: $\dot{m}_1 \approx \dot{m}_2$. The stored energy does not vary noteworthily.

The tank is filled with a solid media that is exposed to thermal contact with air that flows through the tank. Between the air and the solid media there is an energy exchange by convection phenomena that is described by the Newton Cooling Law and the energy conserved in the air flow and the solid media. Due to the spatial distribution of the tank, it is modelled as a parameter distributed system in the direction of the major dimension of the tank. This results in a system of partial differential equations (PDE) formed by:

- Differential formulation of mass and energy conservation through the air volume in contact with the media, in the domain defined along the major spatial dimension of the tank. See [11].

- Differential formulation of energy conservation of the solid media, in the domain defined along the major spatial dimension of the tank. See [11].

To solve the PDE system stated, it is necessary to spatially discretize the equations. ThermoFluid provides partial classes [12] in which the discretization with the Finite Volume Method (FVM) ([10]) is applied. One of these classes is *ThermoFluid.PartialComponents...Volume2PortDS_pT*, which implements this mass, energy and static momentum conservation equations in a volume spatially discretized in n subvolumes. For the solid media, there exists final use classes that implements energy conservation in distributed solids, *ThermoFluid.Components.HeatFlow.Walls*.

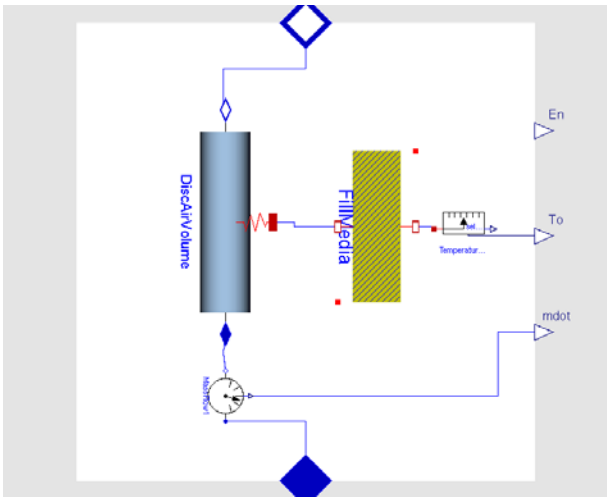


Figure 6: Model of the Storage Tank

To close the system of equations it is mandatory to introduce the heat transfer coefficient between the air flow and the solid media. This coefficient depends of heat transfer correlations using adimensional fluid numbers (Reynold, Prandtl, Pecklet,...), geometry of the contact surface and thermodynamic and transport properties of the fluid (air in this case). Some of the correlation parameters strongly depend on the experimentation and parameter adjusting phase of the modelling work. See [11].
The tank model, designed using ThermoFluid components, is shown in figure 6, where the discretized air volume component (*DiscAirVolume*), the solid media component (*FillMedia*) and the component modelling the heat transfer coefficient[1] can be observed.
The additional variables that are get out through output connectors for control purposes are:

- En : total stored energy in the solid media with respect to a reference level.

- T_0 : spatially averaged temperature of the solid media.

- $\dot{m}$: air mass flow rate through the tank.

3.5 Evaporator

The evaporator in CESA tower is a counterflow air-water/steam hex, in which the water/steam flow is he-

[1]In this version of the model, the heat transfer coefficient is supposed to be constant. Including a dependence of this coefficient using correlations does not pose any additional difficulty, but redesigning a discretized volume for air, polymorphic with the first one, with one variable and one equation for the heat transfer correlation, e.g., Dittus-Boelter [11].

licoidal configured through the air flow. For modelling effects, this components has been considered as a counterflow hex composed of one pipe with air media, one pipe with water-steam media and a wall letting thermal interaction. The simple arrangement can be seen in figure 7.

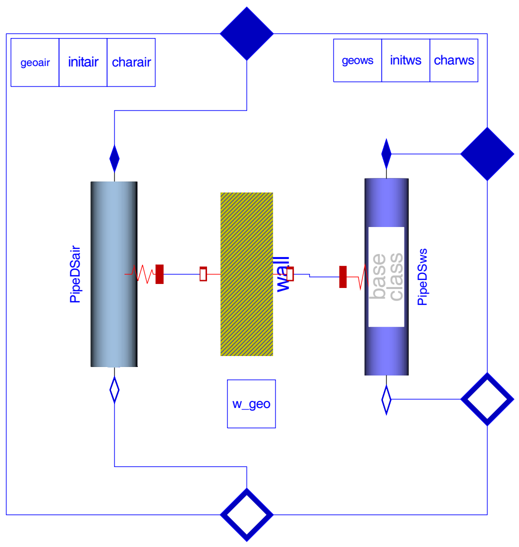


Figure 7: Discretized model of the Evaporator

The length of the water/steam pipe is 1440 m. and under normal operating conditions the inlet water is in subcooled region and the outlet vapor is superheated; thus the dynamical conditions will vary along the pipe depending on the thermodynamic and transport properties of the water/steam. The configuration shown in figure 7 is fully discretized in n CV's in which mass,energy, and momentum balances are applied[2]. The number of CV's, n, is a trade off between accuracy and computing cost, so the final choice is the minimum n that models dominant dynamics for control purposes. Nowadays we are working with values in the interval $[10, 15]$. The wall and the air pipe are discretized with the same discretization level.

In the development of experimental correlations classes for the heat transfer coefficients sliding models and *chattering* have appeared with some frequency around the phase changes of water/steam CV's. This phenomena are manifested with more frequency when CV's pass from subcooled (region 1 in IAPWS-IF97 standard for water/steam properties, [15]) to saturated (region 4 in IAPWS-IF97), due to the discontinuities present in the heat transfer coefficients in the limit boundary between water and walls. To avoid those cases in which *chattering* causes troubles to the simulation, another polymorphic evaporator model has been developed, in which the subcooled and saturated regions of water/steam pipe are replaced by an equivalent Moving Boundary Model (MBM) [8]. Figure 8 shows this mixed discretized and MBM model, where the MBM component has been designed with ThermoFluid interfaces to be connected with the rest of components.

[2]Momentum conservation is stated in staggered CV's with respect to those ones which state mass and energy conservation; [10], [14] and [12].

Although the mixed model lowers the likelihood of finding *chattering* in the integration process, it is theoretically less accurate, and experimentally it is harder to find consistent DAE initial conditions and the validity range of the model is more limited than that of the fully discretized one.

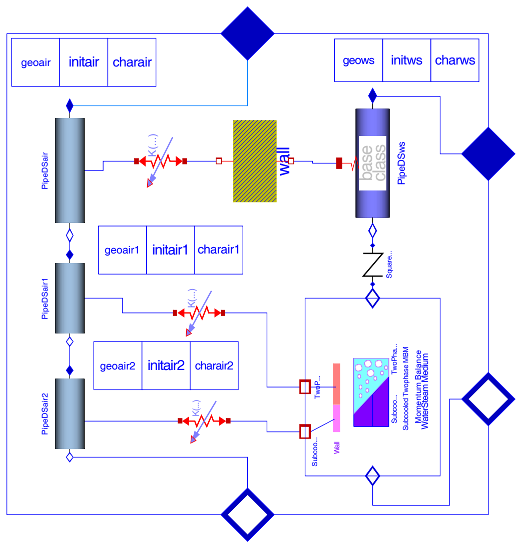


Figure 8: Mixed Moving Boundary and Discretized model of the Evaporator

With the help of *replaceable/redeclare* constructs and the *choices annotations* ([2], [1]), the switching between fully discretized and mixed MBM-discretized models in instantiation time easies the modelling work.

3.6 Simplified model

Some components of the compound model have been introduced to maintain the structure and topology in the model similar to the real system, following the object oriented approach. Some of them, like pipes, actually introduce additional differential equations in the model that could be eliminated due to the fact that the parameters of the real plant make their time constants and delays too low when compared to the rest of components. Eliminating the pipes implies assuming negligible energy losses and fast dynamics in the mass conservation due to the low real volume, which is reasonable. The model could be simplified from figure 3 to the one shown in figure 9.

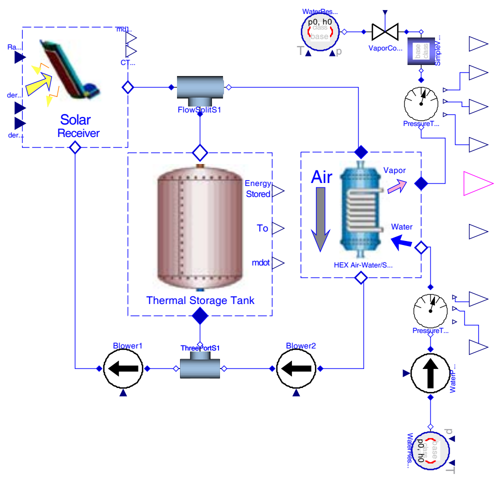


Figure 9: Simplified model of figure 3

4 Simulation

This section shows representative simulation results using the simplified plant model of figure 9, when the system executes the cycle detailed below, that represents a typical operation case:

1. Initial state: the storage tank is unloaded, the air circuit is at ambient conditions and the water circuit is in subcooled region.

2. At $time = 0s$, an input power of 10MW is reflected from the heliostat field in the receiver and the storage tank begins to accumulate energy. No power is delivered to the evaporator.

3. At $time = 2000s$, all the energy from the receiver is delivered to evaporator and the storage tank energy level remains unchanged.

4. At $time = 15000s$, the power from the heliostat field is zero (due to a passing cloud, for example) and all the energy delivered to the evaporator comes from the storage tank.

The results of this simulation are shown in figure 10. The first graph presents the input power radiation (Power_Rad) from the heliostat field. The second the mass flow rates in blowers 1 (mdot_Blower1) and 2 (mdot_Blower2), delivering energy to the storage tank or evaporator in each time interval. The third graph shows the load/unload cycle of the storage tank; it can be seen how the dominating time constants for load/unload are different and that the accumulated energy can be enough to maintain the outlet superheated steam during some time. The fourth graph presents the outlet evaporator water/steam temperature; this augments when the blower2 works until the solar radiation exists, then begins to fall at low rate while stored energy remains in the tank. When the energy coming from the tank is not enough to maintain the superheated steam, it get saturated and then subcooled. The last graph presents the IAPWS-IF97 regions crossed by outlet water/steam travel during this simulation; it can be seen that initially the water is subcooled (region 1), then enters in saturation (region 4), then enters in superheated (region 2); when the solar radiation disappears at $time = 15000s$, the storage tank maintains the superheating state some time until the steam enters in saturation again (region 4), and finally come back to subcooled water (region 1) as during the beginning of the process. At present, no real plant data are allowable for model calibration and validation purposes. Conducted tests are planned to validate each one of the components of the model.

5 Concluding remarks and Ongoing work

This article shows the development of a model of a CRSTPP using the methodology of object oriented development of thermofluid systems. The major part of the components are based in the ThermoFluid framework for thermohydraulic modelling. The CRSTPP components and main operation principles have been described. For the main components, the modelling hypotheses and the composition Modelica diagrams developed with the Dymola tool have been presented. References to the underlying physical phenomena have been made in these composition diagrams, without entering in detail of quantitatively describing them through differential and algebraic equations; instead, the basic bibliography and the ThermoFluid classes that implements them have been referenced. Finally, a simplified model showing a typical operation cycle with a real perturbation introduced by clouds has been simulated.
The ongoing works to develop consists in the adjusting of the main block models parameters based in the experimental results of the real plants, by means of working separately with each shown component with its proper boundary conditions. In this work, the validation of empirical correlations for heat transfer and

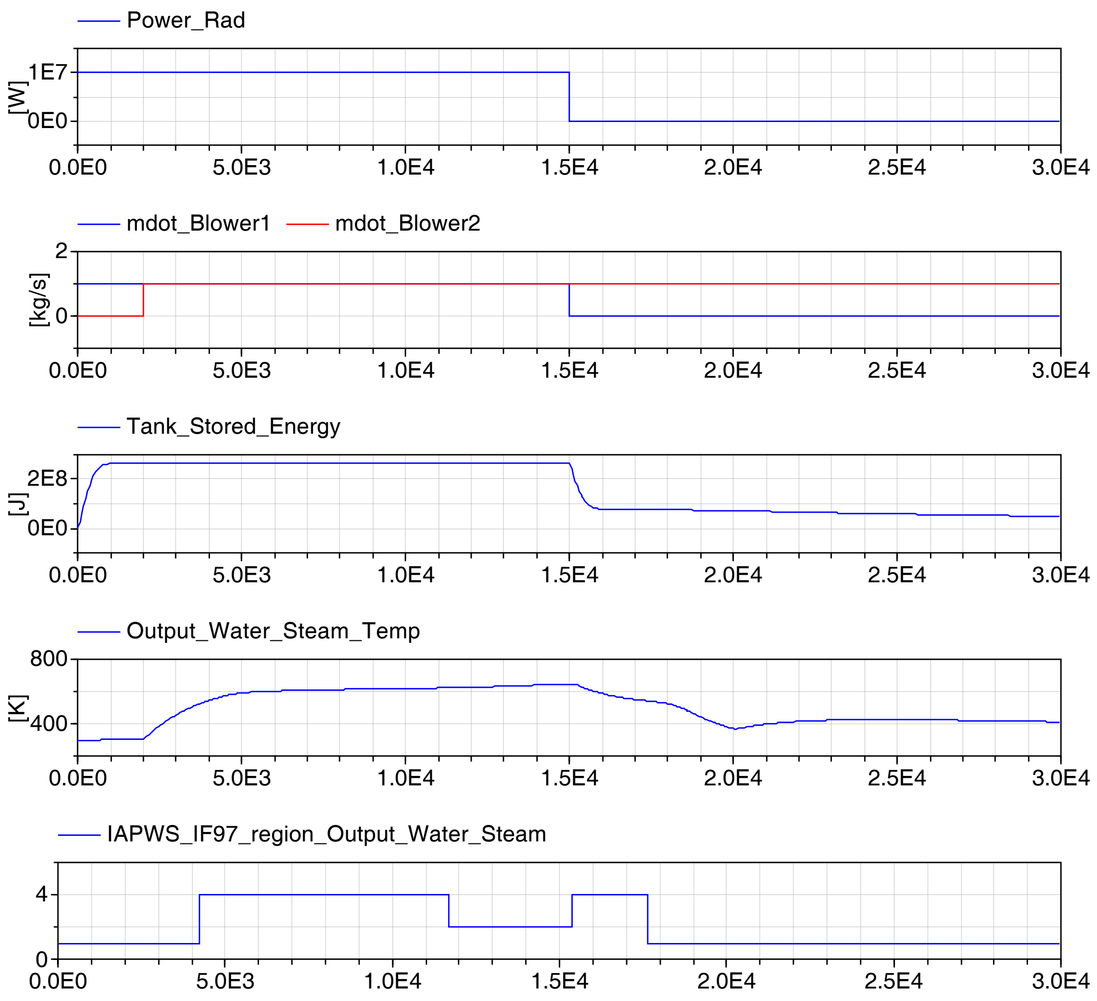


Figure 10: Simulation results.

pressure loss will be an important issue.
The final aim is to develop control and automatic operation systems that help operating in the most autonomous way this kind of plants, in the presence of large disturbances. Automatic start-up and shutdowns of the plants is one of the main objectives in this direction.

Acknowledgements
This work has been financed by CICYT-FEDER funds (projects DPI2001-2380-C02-02, DPI2002-04375-C03, DPI2004-07444-C04-04 and DPI2001-1012) and by the Consejería de Innovación, Ciencia y Empresa de la Junta de Andalucía. This work has been also performed within the scope of the specific collaboration agreement between the Plataforma Solar de Almería and the Automatic Control, Electronics and Robotics (TEP-197) research group of the Universidad de Almería titled ”Development of control systems and tools for thermosolar plants”.
The authors want to acknowledge to J.D. Alvarez for his help with some of the bitmaps of some Modelica models developed. Of course, thanks to Dr. Jonas Eborn, Dr. Hubertus Tummescheit and Lund Technology Institute for the development and make public the ThermoFluid library.

References

[1] Dynasim AB. *Dymola Users Manual. Version 5.3a.* Dynasim AB, Research Park Ideon. SE-223 70 Lund. Sweden.

[2] Modelica Association. Modelica. a unified object-oriented language for physical systems modeling. language specification. version 2.1. Technical report, Modelica Association, January 2004.

[3] K.J. Åström and B. Wittenmark. *Computer-Controlled Systems.* Prentice Hall, 1997.

[4] M. Berenguel. *Contributions to the control of distributed solar collectors (in Spanish).* PhD thesis, Escuela Superior de Ingenieros Industriales de Sevilla. Spain, March 1996.

[5] E.F. Camacho, M. Berenguel, and F.R. Rubio. *Advanced Control of Solar Plants.* Springer, 1997.

[6] J. Eborn. *On Model Libraries for Thermo-hydraulic Applications.* PhD thesis, Department of Automatic Control, Lund Institute of Technology, Sweden, March 2001.

[7] F.J. Garcia-Martín, Berenguel M., Valverde A., and E.F. Camacho. Heuristic knowledge-based heliostat field control for the optimization of the temperature distribution in a volumetric receiver. *Solar Energy*, 66:355–369, 1999.

[8] J.M. Jensen and H. Tummescheit. Moving boundary models for dynamic simulations of two-phase flows. In Martin Otter, editor, *Proceedings of the 2rd International Modelica Conference*, pages 235–244, Oberpfaffenhofen, Germany, March 2002.

[9] Kreith, editor. *The CRC Handbook of Thermal Engineering.* CRC Press, 2000.

[10] S.V. Patankar. *Numerical Heat Transfer and Fluid Flow. Series in Computational and Physical Processes in Mechanics and Thermal Sciences.* Taylor & Francis, 1980.

[11] W.M. Rohsenow, J.P. Hartnett, and Y.I. Cho. *Handbook of Heat Transfer.* McGraw-Hill, 1998.

[12] H. Tummescheit. *Design and Implementation of Object-Oriented Model Libraries using Modelica.* PhD thesis, Department of Automatic Control, Lund Institute of Technology, Sweden, August 2002.

[13] L. Valenzuela, E. Zarza, M. Berenguel, and E.F. Camacho. Direct steam generation in solar boilers. *IEEE Control System Magazine*, 24(2):15–29, 2004.

[14] H.K. Versteeg and W. Malalasekera. *An Introduction to Computational Fluid Dynamics.* Addison Wesley Longman Limited, 1995.

[15] W. Wagner and A. Kruse. *Properties of water and steam.* Springer-Verlag, Berlin, 1998.

[16] E. Zarza. *The Direct Steam Generation with Parabolic Collectors. The DISS project (in Spanish).* PhD thesis, Escuela Superior de Ingenieros Industriales de Sevilla, November 2003.

Object-Oriented Modeling, Simulation and Control of the IRIS Nuclear Power Plant with Modelica

Antonio Cammi[*], Francesco Casella[†], Marco E. Ricotti[*], Francesco Schiavo[†‡]
Politecnico di Milano,
Piazza Leonardo da Vinci 32
20133 Milano
Italy

Abstract

The paper presents the application of the Modelica language to the modeling, simulation, and control of the new IRIS nuclear power plant, under development by an international consortium. The plant model, developed by using components from the ThermoPower library, as well as custom-built nuclear components, is described, as well as the digital control system model, which will eventually become very realistic. Special emphasis is put on the use of inheritance and replaceable objects for the management of a family of model variants over the project life-time. Selected simulation results are included.

1 Introduction

The IRIS project [3] involves 21 organizations from 10 countries and refers to the design of an innovative, light water reactor with a modular, integral primary system configuration. The reactor pressure vessel houses the nuclear fuel, control rods and control rods drive mechanisms, but also all the major reactor coolant system components, including the coolant pumps, the steam generators and the pressurizer (Fig.1).

IRIS is basically a PWR (Pressurised Water Reactor): in the primary loop, liquid water is heated by the nuclear fuel rods in the core, and is then sent by the pumps to the primary side of heat exchanger; the secondary loop actually generates steam which is sent to turbines to produce power.

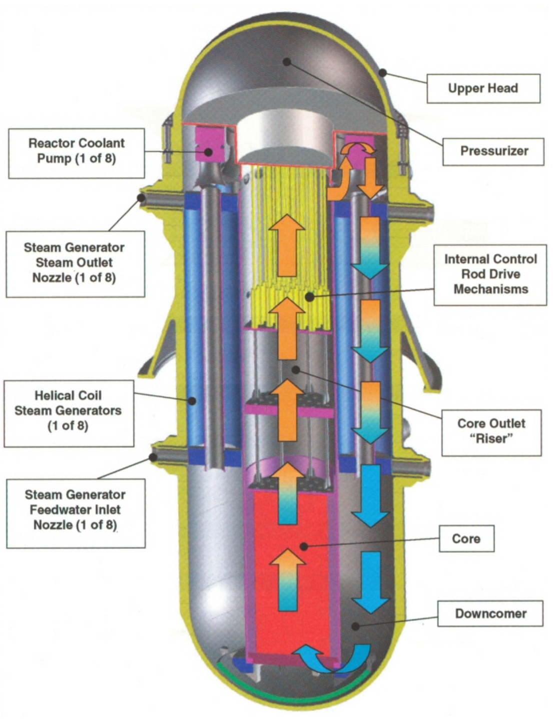

Upper Head
Reactor Coolant Pump (1 of 8)
Pressurizer
Steam Generator Steam Outlet Nozzle (1 of 8)
Internal Control Rod Drive Mechanisms
Helical Coil Steam Generators (1 of 8)
Core Outlet "Riser"
Steam Generator Feedwater Inlet Nozzle (1 of 8)
Core
Downcomer

Figure 1: The IRIS Reactor

Compared to conventional PWR plants, however, IRIS has a set of distinctive features, which directly affect the control system design:

- the integral configuration requires a large water inventory in the primary loop, whose residence time is much greater than usual;
- a helicoidal once-through steam generator is employed on the secondary side, which has a very short residence time, compared to the more widespread U-tube recirculating steam generators;
- sprayers are not available to reduce the pressure in the primary loop during fast transients.

[*]Dipartimento di Ingegneria Nucleare "E. Fermi"CESNEF, {antonio.cammi,marco.ricotti}@polimi.it
[†]Dipartimento di Elettronica e Informazione, {casella,schiavo}@elet.polimi.it
[‡]*Corresponding author*

The control strategy must take these facts into account, and a dynamic simulation tool is essential to ensure that the control objectives can be achieved.

Highly detailed dynamic simulators have been developed for the IRIS reactor [6]. Such simulators, based on the complex computational fluid-dynamics code named *RELAP* [10], are perfectly suited for accident analysis and safety-oriented evaluations of the reactor design features. On the other side, due to the amount of the details involved, they cannot be proficiently used for control-oriented dynamic simulation.

Within this framework, the use of the Modelica language offers a viable solution, allowing the development of dynamic simulators that are detailed enough for control-oriented analysis and yet with limited computational requirements.

To provide the required capabilities for the analysis, specific models for nuclear reactor components have been developed, to be applied for the dynamic simulation of the IRIS integral reactor, albeit keeping general validity for PWR plants. In addition to that, specific digital control blocks have been developed, so that a complete model of the plant and of its digital control system is available.

The paper is organized as follows. An overview of the plant model is presented in Section 2, while in Section 3 the models specifically developed for nuclear components are analyzed in detail. Section 4 contains an overview of the plant digital control system and, in Section 5, the problem of managing a library of plant models with different detail levels is tackled. Section 6 presents some closed-loop simulation results. Finally, Section 7 draws some conclusions and outlines possible future developments.

2 Plant Model

The model of the IRIS plant basically describes the primary circulation loop, i.e. the reactor coolant loop, and the secondary loop, i.e. the once-through evaporators, along with the feedwater and turbine systems. Most of the required models have no specific *nuclear* features, and were thus borrowed from the general-purpose ThermoPower library, designed for the modelling of generic thermo-hydraulic power plants; the library is an open-source project, described with more detail in [4]. The only notable exceptions are the reactor core and the pressurizer, which are described in the next section.

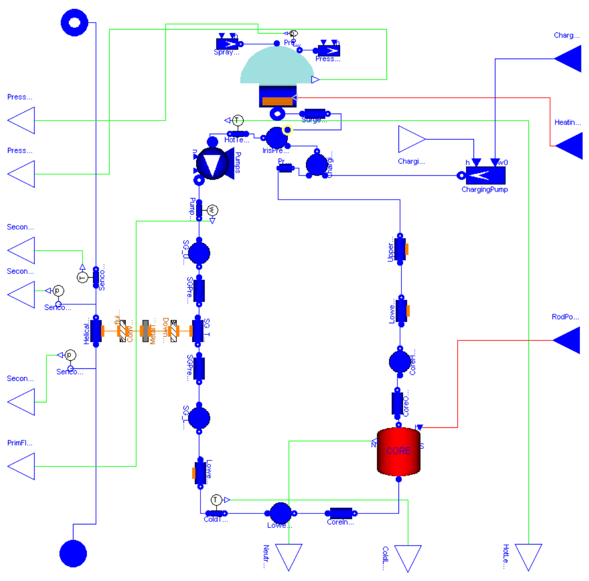


Figure 2: Plant flow diagram

2.1 Primary loop

The primary loop (see Fig. 2) starts with the pressurizer (top of the diagram); the pressurizer is connected by a pressure-loss component to the upper mixing volume, taking into account the mass and energy balances. Starting from the top of the diagram, counterclockwise, the centrifugal pump model can be found, followed by another plenum model. The primary side of the heat exchanger between the primary and secondary loop is then encountered, modelled by three cascaded, finite-volume pipe models; the middle one describes the section where the coolant is actually in contact with the secondary side tube bundle. Proceeding onwards, other two plenum models followed by a pressure loss can be found, leading to the inlet of the core model (see next Section). This in turn is followed by another pressure loss, another plenum, and the two riser sections, modelled by two pipes having different diameter. The loop is closed by a simple model of the chemical and volume control system (CVCS), basically a mixer and an ideal flow source. The fluid in the whole loop is one-phase water, with the exception of the steam filling the upper pressurizer dome.

The heat transfer between the primary and secondary loop is modelled by two heat transfer modules and by the thermal model of the tube metal mass. The primary-side heat transfer coefficient is held constant to its nominal value, since the Reynolds and Prandtl numbers does not vary substantially; on the secondary side, the heat transfer coefficients can be computed according to different laws, e.g. Chen's correlation, or much simpler, empirically tuned curves.

2.2 Secondary loop

The secondary loop is composed of the feedwater system, the helical coil once-through steam generator and the turbine system. The once-through generator is represented by a finite-volume, 10-node model of the two-phase fluid flow, assuming homogeneous flow, i.e. the same velocity for the liquid and vapor phases.
Currently, the feedwater system is represented by an ideal flow source, whose flow rate is determined by the control system, and whose enthalpy is a function of the plant load level, determined from balance-of-plant calculations. The turbine system includes a simplified, linear model of the high- and low-pressure turbines, plus simplified models of the connection to the grid, including an idealized synchronous generator, local loads, and a grid model. The latter ones are included to provide suitable boundary conditions for the (much slower) plant dynamics; therefore, they only model active power flows, neglect the electro-mechanical dynamics, and assume perfect synchronism between the generator and/or the grid.
In the near future, it is planned to replace the feedwater and turbine system models with more realistic counterparts, including steam bleedings and condensate train, to better represent the actual steam generator dynamics under large load variations. On the other hand, the finite-volume fluid evaporator model could be replaced by a simpler version, with moving boundaries between the liquid, 2-phase, and vapor sections, and an averaged description of each section.

3 Nuclear Components

The *Modelica* models for "nuclear" components have been developed to provide solutions which are suitable both for "general" use and specifically for the IRIS nuclear plant modelling. The main components are the *core*, (with separate models for the point kinetic neutronic generation, the fuel thermal dynamics and the moderator, as depicted in Fig. 3) and the *pressurizer*; the main modelling principles are summarized here, for more details see [1, 2].

3.1 Point Kinetics Neutronic

The point kinetic neutronic model describes the dynamics of the neutron generation processes in the core. The model is based on standard point kinetic

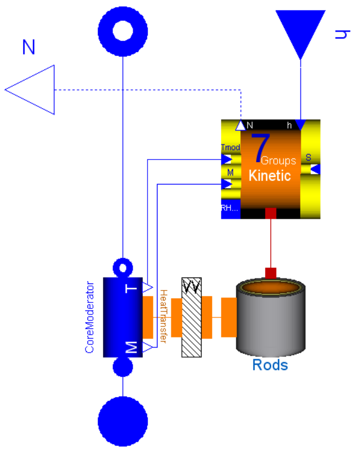


Figure 3: The Core Model Internal Structure

dynamic balance equations, describing the evolution of the neutronic population and of the precursor concentration. Reactivity feedback from coolant density, fuel Doppler effect, and rod insertion are accounted for. The dynamic terms can be switched off, to obtain a simplified static model, neglecting the fast dynamics. The neutronic power generated into the fuel is proportional to the neutronic population n, which responds to the point reactor kinetics balance equations :

$$\begin{aligned} \frac{dn}{dt} &= \frac{\rho-\beta}{\Lambda} n + \sum_{i=1}^{6} \lambda_i c_i \\ \frac{dc_i}{dt} &= \frac{\beta}{\Lambda} n - \lambda_i c_i \quad i = 1, \cdots, 6 \quad , \end{aligned} \tag{1}$$

where c is the precursor concentration leading to a delayed neutron source, ρ is the total reactivity of the core, β is the fraction of delayed neutrons, λ is the decay constant of the precursors and Λ is the characteristic period of the reactor or mean neutron generation time.
Reactivity feedbacks are taken into account as well, by considering linear or non linear feedback coefficients, always negative, for the coolant density effect (α_c), the fuel Doppler effect (α_f), the effect of the boron concentration (α_B) into the primary fluid as a neutronic poison and the level of insertion of the control rod banks into the core (α_{CR}). These relations are

$$\begin{aligned} \rho &= \rho_{CR} + \rho_f + \rho_c + \rho_B\,, \\ \rho_f &= \alpha_f \left(T_{eff} - T_{eff_0}\right)\,, \\ \rho_c &= \alpha_c \left(\frac{1}{v_c} - \frac{1}{v_{c0}}\right)\,, \\ \rho_B &= \alpha_B \left(C - C_0\right)\,, \end{aligned} \tag{2}$$

where T_{eff} and T_{eff_0} are the instantaneous and reference effective fuel temperature, respectively, obtained from the fuel model, v_c and v_{c0} are the instantaneous and reference specific volumes of the coolant, C and C_0 are the instantaneous and reference boric acid concentration in the coolant. The boric acid concentration in the coolant depends mainly on the control rods insertion.

The reference values are those corresponding to the nominal, full power operation of the reactor.

3.2 Fuel model

The fuel model describes the dynamics of the thermal power generated within the core by the nuclear chain reactions. The neutronic generation model and the fuel model are linked by a connection between two *Modelica* standard `HeatPort`, where the connectors variable are the total power generated and the fuel temperature. A *ThermoPower* `DHT` distributed heat transfer connector is used as well, as an interface with the moderator, modelled by a 1-D flow model.

The model is based on the application of the time dependent Fourier equation (in monodimensional cylindrical geometry) to the three fuel zones: pellet, gap and cladding (Fig. 4).

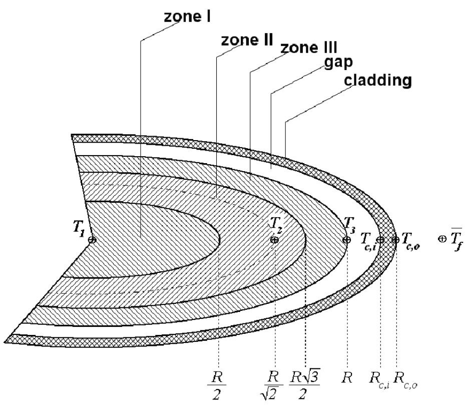


Figure 4: Fuel pellet radial scheme for heat transfer modelling

The main assumption of the model is to consider only the radial heat transfer, thus disregarding both the axial and the circumferential diffusions. Fourier's equation is discretized radially in five zones, and longitudinally in a user-decidable number of segments (N). For the pellet, gap and cladding the corresponding balance equations read:

$$\begin{aligned}
&\rho_p c_{p,p}\frac{\partial T_p}{\partial t} = \frac{1}{r}\frac{\partial}{\partial r}\left(rk_p\frac{\partial T_p}{\partial r}\right) + q''' \ , \\
&\frac{\partial}{\partial r}\left(k_g\frac{\partial T_g}{\partial r}\right) = 0 \ , \\
&\rho_c c_{p,c}\frac{\partial T_c}{\partial t} = \frac{1}{r}\frac{\partial}{\partial r}\left(rk_c\frac{\partial T_c}{\partial r}\right) \ .
\end{aligned} \tag{3}$$

where ρ is the density, c_p is the specific heat, T is the temperature, k is the thermal conductivity, q''' is the volumetric source term, r is the radial dimension and t the time, while the subscripts p, g and c stand for the pellet, the gap, and the cladding, respectively.
The heat transfer model is represented in Fig. 4, with the pellet discretized into three zones of equal volume. Eqs.(3), together with the conditions of heat flux vanishing at the pellet center and the continuity of the temperatures and heat fluxes at the three boundaries pellet-gap-cladding-coolant allow the determination of $T_p(r,t)$, $T_g(r,t)$ and $T_c(r,t)$.
In addition to the above equations, five correlations synthesizing the dependance of $c_{p,p}$, k_p, $c_{p,c}$ and k_c as a function of the fuel temperature and of k_g as a function of both the reactor power and the burn-up have been adopted.
The condition at the cladding-coolant interface is determined by the distributed heat transfer connector variables.
Finally, the effective fuel temperature, used to evaluate the Doppler feedback contribution on neutronics, is defined as follows:

$$T_{eff} = \frac{4}{9}T|_{r=0} + \frac{5}{9}T|_{r=R} \ . \tag{4}$$

3.3 Moderator

The core moderator is modelled by the ThermoPower library `Water.Flow1D`, with a small extension to make the fluid density available to the point kinetics model. The coolant model has the same number of volumes as the fuel. The convective heat transfer between the two components is calculated at each node by

$$\begin{aligned}
&\phi_{mod} = -\phi_c \ , \\
&\phi_{mod} = \gamma(T_c - T_{mod}) \ ,
\end{aligned} \tag{5}$$

where ϕ_{mod} and ϕ_c are, respectively, the moderator and the fuel cladding heat flux, γ is the heat transfer coefficient, and T_{mod} and T_c are the moderator and fuel cladding temperatures. Detailed RELAP simulations have shown that the heat transfer coefficient is approximately constant for all the operating conditions the control system is concerned with.

3.4 Pressurizer Model

The pressurizer model is based on a lumped parameter approach, which is appropriate to the IRIS case. Both water properties in the liquid volume and in the steam volumes are assumed as homogeneous, at equal pressure but not at thermodynamic equilibrium.

The model is based on two groups of dynamic mass and energy balance equations, the first for the liquid phase and the second for the vapor phase inside the tank. Mass and energy transfer between the two phases is provided by bulk condensation and surface condensation of the vapor phase, and by bulk boiling of the liquid phase. Additional energy transfer can take place at the surface if the steam is superheated.

External interfaces are provided for connections to the hydraulic loop by a bottom flange and to a safety circuit by a *safety* flange; also available are a heating power command input and a level signal probe output.

The heating power input is processed by a limiter and a low pass filter block to simulate the delay in heating effect and the limited heaters power. The resulting effective heating power signal drives the production of saturated steam by the heaters at a rate corresponding to the difference between the enthalpy of the liquid holdup and the enthalpy of saturated steam. For simplicity, the corresponding steam flow enters directly the steam holdup, without causing heating of the liquid holdup.

The bottom flange's flow enters directly the liquid volume; its pressure is increased depending on the liquid holdup's level.

The metal wall dynamics is taken into account, assuming uniform temperature. Heat transfer takes place between the metal wall and the two phases and between the wall and the external ambient at fixed temperature.

4 Control

The control design of the IRIS nuclear power plant is a complex task, with objectives that, depending on the plant operating conditions, vary from the management of start-up sequences to the recover from turbine or reactor trips and to the grid power/frequency regulation at full nuclear power.

Classic design concepts, for early nuclear units, relied on separate control systems for each control loop, and limited signal interaction between the loops [9]. This simplified the design of each loop, particularly with analog control systems where each interconnection added hardware expense. On the other side, the current trend is for more integrated systems that can take advantage of coordinating the different control loops [7]. This allows for more effective plant control, but complicates the control system failure analysis. A viable solution for IRIS is the choice of a hierarchical control system, as depicted in Fig. 5.

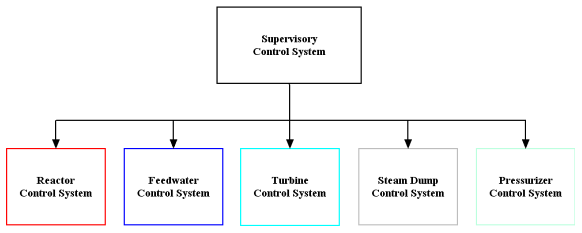

Supervisory Control System
Reactor Control System
Feedwater Control System
Turbine Control System
Steam Dump Control System
Pressurizer Control System

Figure 5: Control system architecture

At the top level is located a supervisory control system with the following functions:

- Establish the plant electric power reference signal. Such reference signal will be used to derive reference and/or feedforward signals for the other major control loops.
- Monitor plant conditions and determine/coordinate the appropriate operating modes for the major control systems.

The control sub-systems have different settings and a varying structure (i.e., different inputs and different controller structure) depending on the specific operating mode of the plant.

All the operating modes to drive the plant during the non-emergency maneuvers have been designed [8]; nevertheless, only the "full-power" control mode (nuclear flux from 20% to 100%) has been fully implemented, simulated and tested yet, so, from here on, the description will cover only such operating mode.

4.1 Supervisory Control

The supervisory control system uses the normalized desired power as an input signal to derive the reference and feedforward signals for the lower-level control systems. On the base of the desired power the temperature and nuclear flux reference for the reactor control are derived, along with a pressure reference for the turbine and steam dump control systems and with the flow rate reference for the feedwater control. The signals to be fed to the lower systems are derived from the desired power reference with linear filtering and through look-up tables based on steady-state plant balances.

4.2 Reactor Control

The aim of the reactor control is to control the coolant temperature, and thus the reactor nuclear power, by driving the control rods stepping system. As a matter of fact, the reference is a temperature signal coming from the upper level, while the measurements include the core coolant average temperature (obtained as the mean between the temperature at the core inlet and the one at the core outlet) and the nuclear neutron flux (obtained through special sensors enclosed within the core shielding).

The temperature error, with suitable dynamic compensation, is used to generate an error signal for determining the speed request for the control rods, along with a power mismatch signal which is used to improve the stability and the velocity of the reactor control system response. The power mismatch signal (i.e. the difference between the reference and the measured neutronic flux) is fed into a rate compensation filter, to eliminate steady-state influence, and then into a non-linear, power-dependant gain, to improve low-power response while avoiding high frequency excitation of the rod stepping system.

The combined error signal enters a rod speed program that features a small dead band to avoid high frequency rod stepping. The speed request thus generated is then serviced by a servo control system embedded within the control rods drive mechanism. This servo is currently described by a high-level behavioral model, which could be eventually replaced by a physical-based model.

4.3 Turbine Admission Valve Control

The turbine system for the IRIS power plant has not been completely designed yet and it is reasonable to assume that the turbine supplier will provide most of the requirements for the turbine control system; however, the design must be compatible with the overall IRIS plant control strategy. The most important constraint is that the IRIS turbine control will have the responsibility for controlling steam pressure by acting on the turbine admission valve (TAV).

The control is based on a PID, its input being the reference pressure signal coming from the supervisory control system and the actual steam pressure measured at the turbine inlet, with suitably low-pass filtering action. The PID output is then summed to the amplified frequency mismatch (i.e., the difference between the actual generator frequency and the desired frequency), with the gain depending on the grid droop setting. The resulting signal is fed to the TAV drive system after being filtered by a non-linear algebraic function, which is an approximate inverse of the TAV characteristic.

4.4 Steam Dump Control

The steam dump control system must control steam pressure when the turbine admission valve control is not doing so, and must provide a backup in all other cases. Experience shows that a simple PID control performs well, particularly if the system uses hydraulic steam dump valves, as it will be in the IRIS case.

The controlled variable is the steam dump valve opening, while the controller inputs are the pressure reference (from the upper level) and the turbine inlet steam pressure (low-pass filtered). Additional steam-dump action is available in case of need: the power reference, filtered through a rate compensator and a suitable gain, is added to the steam dump valve control signal, to provide a faster response in case of sudden changes in the requested power (e.g., when a reactor or a turbine trip occur and the supervisory control system instantaneously lowers the power reference).

4.5 Feedwater Control

The feedwater control system directly controls the feedwater flow in the secondary side by acting on a valve located at the feedwater pumps. The structure is based on two PID controllers in cascade configuration. The inner loop acts to control feedwater flow to the reference value obtained from the supervisory control. In the ideal case with perfect settings in the supervisory controller, this would result in the plant operating at the desired power, at least in steady state. Of course, such an open loop control on power would be sensitive to parameter variations, so the outer loop provides a trim signal to adjust feedwater flow to achieve the desired power by the action of a PID controller with the reference and the actual power as inputs. The feedwater valve control signal is then filtered by a non-linear algebraic function, which is an approximate inverse of the valve characteristic.

4.6 Digital PI controller

Models for digital PI and PID controllers, in ISA form, have been implemented. Here, for the sake of brevity, only the PI development is briefly showed: the PID model development is quite similar.
The model is based on the standard industrial ISA formulation, with the output calculation formula obtained with a Tustin discretization:

$$\begin{aligned} CS(s) =& K_p \left((bSP(s) - PV(s)) + \frac{1}{T_I\, s}(SP(s) - PV(s)) \right) \\ & \Downarrow \left(\text{Tustin: } s = \frac{2}{T_s}\frac{z-1}{z+1} \right) \qquad (6) \\ CS(z) =& SP(z)\frac{a_0 + a_1\, z^{-1}}{1 - z^{-1}} + PV(z)\frac{b_0 + b_1\, z^{-1}}{1 - z^{-1}} \end{aligned}$$

with

$$a_0 = \frac{2\,Kp\,b\,T_I + Kp\,T_s}{2\,T_I} \quad , \quad a_1 = \frac{-2\,Kp\,b\,T_I + Kp\,T_s}{2\,T_I} \quad ,$$
$$b_0 = \frac{-2\,Kp\,b\,T_I - Kp\,T_s}{2\,T_I} \quad , \quad b_1 = \frac{2\,Kp\,b\,T_I - Kp\,T_s}{2\,T_I} \quad .$$

The complete controller model includes also advanced features like *manual* and *tracking* working mode, output saturation, and anti wind-up mechanism.
The resulting block has two boolean inputs (automatic and tracking switch signals), four discrete real inputs (set-point, process value, manual and tracking signals) and a discrete real output (control signal).
The *Modelica* implementation exploits the language features for digital blocks, using discrete variables and with the instructions enclosed within a sampling loop:

```
when {initial(),sampleTrigger} then
...
[PI computations]
...
end when;
```

The anti wind-up mechanism is implemented via an auxiliary variable:

```
CSwind=pre(CS)+a0*SP+a1*pre(SP)+b0*PV+
       b1*pre(PV);
```

where `Cswind` is the auxiliary variable, `CS` the control variable, `SP` the set-point and `PV` the process value.
The actual control value is chosen depending on the controller logic state (automatic, manual or tracking) and on the saturation values, e.g. :

```
if AUTO then
   if CSwind >= CSmax then
       CS = CSmax;
       CSport.signal[1] = CSmax;
   elseif CSwind <= CSmin then
       CS = CSmin;
       CSport.signal[1] = CSmin;
   else
       CS = CSwind;
       CSport.signal[1] = CS;
   end;
else
   ...
```

where the parameters `CSmax` and `CSmin`, are the upper and lower saturation limits for the control action.
With this implementation structure, the controller integral state is automatically updated at every execution cycle so to be coherent with the last output sample.

5 Model Management through the project life-cycle

Object-oriented features such as inheritance and replaceable components are often described as key factors in the development of reusable model libraries. In fact, they can also be extremely useful for the proper management of families of application models throughout an engineering project's lifetime, as it will be explained in this section with reference to the IRIS project.

5.1 Requirements

During the IRIS project lifetime, a considerable number of model variants will have to be built and analyzed; some of them will become obsolete and will have to be discarded, while others should be kept consistently up-to-date. The motivations of the model variants are now briefly discussed.
Depending on the specific simulation to be performed, different accuracy vs. computational load trade-offs are required. Reference simulations should be performed with the maximum level of accuracy and detail, and cross-checked with the results of the reference simulations performed with the certified RELAP code. When performing simulations around a certain operating point, some approximations could then be introduced, which are only valid for that operating region; it should be possible to easily check simplified versions against their more accurate counterparts.
Some of the plant parameters (e.g. the pump characteristics, or some plenum volumes) are not yet definitive, and could change in the future; when one of such parameters is changed, it is essential that all the current model variants are updated consistently.
Once the initial phase of the control system design has been carried out, a systematic simulation campaign

must be performed to check that the operational constraints (i.e., the activation thresholds of the protection system) are never violated in all the predicted operating conditions and transients; thousands of different simulation runs can be required. To carry out this task, the simplest and fastest possible variant of the plant model should be used.

It should be also kept in mind that the plant models will be developed, used, and maintained by different people over a wide time span (several years) and at widely spaced sites (US, Europe). For instance, the model presented in this paper will be presumably frozen for some months, and then possibly resumed when the project will enter the commercial phase. It is therefore essential to avoid building a plethora of distinct models, differing only by some details, which would be extremely difficult to maintain and document consistently.

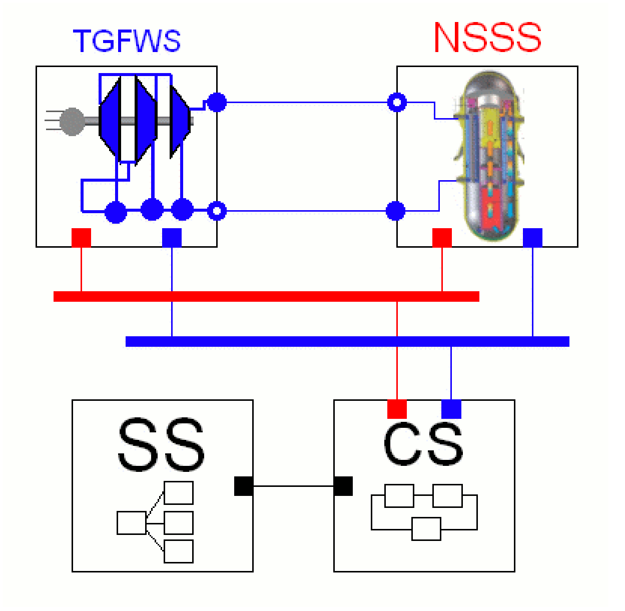


Figure 6: The Base Simulator Model

5.2 Implementation

The top-level structure of the simulator is represented in Fig. 6: the TGFWS block contains the Turbine/Generator/Feedwater system model; the NSSS block contains the Nuclear Steam Supply System model, i.e. the nuclear reactor, with the primary and secondary loops. The two are connected to each other by thermo-hydraulic connectors. The control side is represented by the CS (Control System) block, collecting all the control loops, and the SS (Supervisory system) block, which generates the set points for the CS based on the plant load request. Three bus connectors carry the sensor, actuator, and reference signals. This structure is common to all the possible variants of the model, and thus contained in the `IRISSimulatorBase` partial model. Different versions of the simulator can be instantiated by selecting the actual content of each block; for instance, one could use the simplified TGFWS model described in Section 2, or a more detailed one.

The NSSS model contains a replaceable model (`HelicalCoil`) for the secondary side of the once-through steam generator, which can be implemented by either the finite-volume or the moving boundary model, and by adding through inheritance the desired equations to compute the heat transfer coefficient.

Besides that, it is possible to vary dramatically the degree of detail and the computational load of the model by changing the number of nodes in the core and once-through generator models, as well as by redeclaring the medium models in the primary and secondary loop components. The default medium models are the IF97-based water models taken from `Modelica.Media`, but it is possible to use much faster models, based either on table interpolation or on equation-based simplified medium models. The thermodynamic conditions of the fluid in the primary loop conditions vary in a rather narrow range (140 to 160 bar, 270 to 330 degrees Celsius), so that extremely simplified models can still be acceptable; the fluid conditions in the secondary loop vary in a broader range, from subcooled liquid to superheated steam, albeit in a narrow pressure range around 58 bar, due to the pressure control system action.

Last, but not least, if an incompressible fluid model is adopted for the primary loop, the fast pressure states caused by the small compressibility of the fluid, coupled with the small hydraulic resistances around the circulation loop, are automatically avoided, without any need to change the component models. This is essential to allow the use of the faster explicit integration algorithms (e.g. forward Euler).

The simulation suite is then organized as a small library (Fig. 7), containing the “empty” base models, and the actual models of the different parts, without any unnecessary duplicate of data. Any specific variant of the simulation model can be instantiated from this library by using suitable modifiers. For example, the variant V2 of the simulator, using a simple incompressible water model for the primary loop, 7 nodes in the core model, a finite volume model of the steam generator with 15 nodes using Chen’s correla-

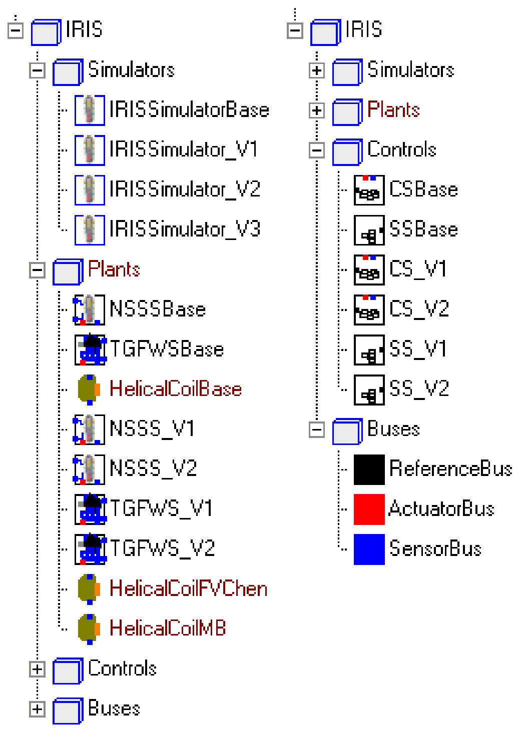


Figure 7: Iris Simulation Suite

tion for the heat transfer coefficient, the variant V1 of the TGFWS, and the variant 2 of CS and SS, is instantiated as follows:

```
model IRISSimulator_V2
 extends IRISSimulatorBase(
    redeclare Plants.NSSS_V1 NSSS(
      redeclare package PrimaryMedium =
        Media.SimpleIncompressibleWater,
      Core(N = 7),
      redeclare Plants.HelicalCoilFVChen
        HelicalCoil(N=15)),
    redeclare Controls.CS_V2 CS,
    redeclare Controls.SS_V2 SS,
    redeclare Plants.TGFWS_V1 TGFWS);
end IRISSimulator_V2;
```

`IRISSimulatorBase` is the empty base model described at the beginning of the section, and its four replaceable components `NSSS`, `TGFWS`, `CS`, `SS` are of type `NSSSBase`, `TGFWSBase`, `CSBase` and `SSBase`, which again only contain the interfaces. The NSSS model in turn contains the replaceable steam generator model `HelicalCoil`.

In this way, it is straightforward to maintain a consistent state for a potentially large family of simulator variants, as well as documenting all of them efficiently.

6 Simulation

The results of a closed-loop simulation, obtained with the tool Dymola ([5]), are now presented. The reference transient is a filtered step variation of the electrical load reference, from 90% to 100% and then back to 90%. Although such a rude transient will never be performed on the actual plant, it is usually employed to assess the overall dynamic response of the control system, in terms of speed of response, damping, overshoot, and so on. The normalized transients of the neutron flux (representative of the generated nuclear power) and of the generated electrical power are shown in Fig. 8, along with the reference power signal. The responses are well-damped and with limited overshoot. The neutron flux transient takes into account the effect of the step-by-step actuation mechanism, as well as of the dead-band included to avoid persistent chattering around a specific operating point. The corresponding normalized transients of some control variables (i.e. TAV opening, feedwater flow rate, and rod insertion) are shown in Fig. 9.

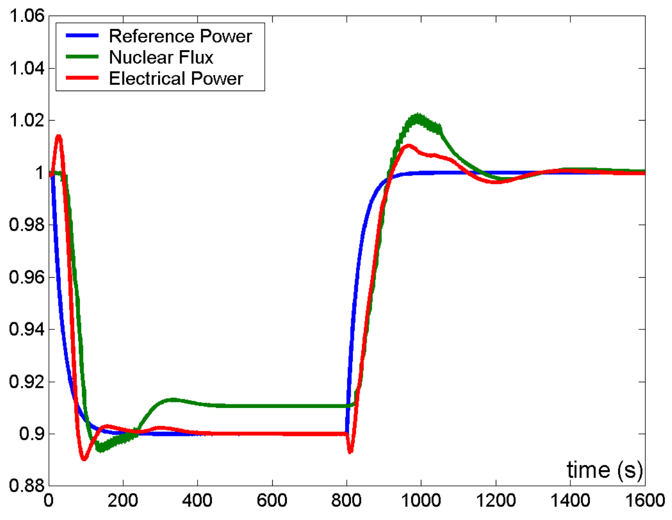


Figure 8: Normalized response to a step load variation: measured variables

7 Conclusions and Future Work

In this paper, the application of Modelica to the study of the control system of the new IRIS nuclear power plant has been presented; this is also the first industrial-scale application of the ThermoPower Modelica library.

The well-behaved nature of the closed-loop transients

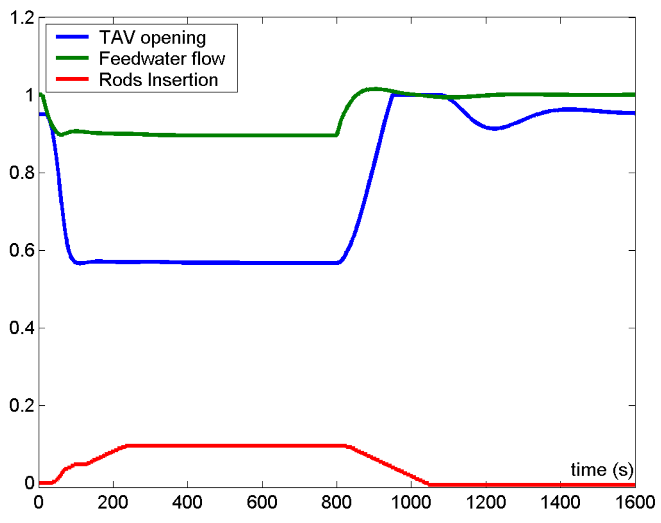


Figure 9: Normalized response to a step load variation: control variables

has confirmed that the new reactor concept will not pose exceedingly difficult problems to the control engineers, compared with already existing PWR plants. On the other hand, the availability of a detailed dynamic model will allow the study of more advanced control concepts, to cope with situations such as. e.g., load/frequency control in small grids, or improved management of blackout transients.
The object-oriented features of the Modelica language (replaceable classes in particular) have been fully exploited to allow the efficient management of all the variants of the plant simulator, which will be needed throughout the project's life-time. The structure of the simulation suite will allow an easier re-use and extension of the models developed so far, when the project will eventually enter the detailed engineering phase, prior to the construction of the first plant.

Acknowledgements

The authors wish to thank Westinghouse Electric Company LLC (IRIS project partner), and particularly Gary D. Storrick, Luca Oriani and Mario D. Carelli, for their contribution and support.

References

[1] A. Cammi, F. Casella, M.E. Ricotti, and F. Schiavo. New modelling strategy for IRIS dynamic response simulation. In 5th *International Conference on Nuclear Option in Countries with Small and Medium Electricity Grids*, Dubrovnik, Croatia, June 3-4, 2004.

[2] A. Cammi, F. Casella, M.E. Ricotti, and F. Schiavo. Object-oriented modelling for integral nuclear reactors dynamic simulation. In *International Conference on Integrated Modeling & Analysis in Applied Control & Automation*, Genoa, Italy, October 3-4, 2004.

[3] M.D. Carelli. IRIS: A global approach to nuclear power renaissance. *Nuclear News*, 46(10):32–42, September 2003.

[4] F. Casella and A. Leva. Modelica open library for power plant simulation: Design and experimental validation. In 3rd *Modelica Conference*, Linköping, Sweden, November 3-4, 2003. `http://sourceforge.net/projects/thermopower/`.

[5] Dymola. *Dynamic Modelling Laboratory, v. 5.3*. Dynasim AB, Lund, Sweden.

[6] D. Grgic, T. Bajs, and L. Oriani. Development of RELAP5 nodalization for IRIS non-LOCA transient analyses. In *American Nuclear Society Topical Meeting in Mathematics & Computations (M&C)*, Gatlinburg, USA, April 6-10, 2003.

[7] IAEA. Modern instrumentation and control for nuclear power plants: A guidebook. Technical Report 387, International Atomic Energy Agency, Vienna, 1999.

[8] F. Schiavo and G. Storrick. IRIS control systems conceptual design. Internal report STD-ES-04-34, Westinghouse Electric Company, September, 2004. *Westinghouse Electric Company Proprietary*.

[9] M. A. Schultz. *Control of Nuclear Reactors and Power Plants*. McGraw Hill, New York, 1961.

[10] A. S. L. Shieh, R. Krishnamurthy, and V. H. Ransom. Stability, accuracy, and convergence of the numerical methods in RELAP5MOD3. *Nuclear Science and Engineering*, 116(10):227–244, 1994.

Session 5c

Tools II

A Portable Debugger for Algorithmic Modelica Code

Adrian Pop, Peter Fritzson
PELAB – Programming Environment Lab, Dept. Computer Science
Linköping University, S-581 83 Linköping, Sweden
{adrpo, petfr}@ida.liu.se

Abstract

In this paper we present the first comprehensive debugger for the algorithmic subset of the Modelica language, which augments previous work in our group on declarative static and dynamic debugging of equations in Modelica. This replaces debugging of algorithmic code using primitive means such as print statements or asserts which is complex, time-consuming and error-prone.

The debugger is portable since it is based on transparent source code instrumentation techniques that are independent of the implementation platform.

The usual debugging functionality found in debuggers for procedural or traditional object-oriented languages is supported: setting and removing breakpoints, single-stepping, inspecting variables, back-trace of stack contents, tracing, etc.

1 Introduction and Related Work

Most language development environments provide some kind of support for debugging and profiling.

Such techniques have also been developed for Modelica at the prototype level with regards to supporting declarative debugging of equation-based models [1, 2]. The presented work complements the existing debugging work with the first (to our knowledge) portable debugger for the algorithmic part of the Modelica language. The debugger is part of the Open Modelica project [3, 4].

There are several techniques for creating debuggers. Most of them are not portable and rely heavily on knowledge of the operating system and the underlying machine [5].

The approach we use in this paper is based on source code instrumentation of the intermediate representation in the compiler, similar with the work described in [6-8]. One can view the Modelica algorithmic code as an event generator and the debugger as a collector of these events that reacts to them accordingly.

The compiler has intimate knowledge of the Modelica code in its internal representation. We augment this representation with debugging nodes (or sites) that actually performs calls to the debugging functions. We have introduced a new compiling phase in the compiler where we walk on the internal representation and augment it with calls to several debugging functions implemented in C.

We have experimented with several ways of augmenting the internal compiler representation with debug nodes in order to search for the best memory consumption vs. speed of the debugger. These augmentation choices deal with the way the variables and code position is passed to the debugger functions.

The paper is structured as follows: This section presented an introduction and related work. Next section presents a debugging session on a short Modelica example, concentrating on the debug functionality. Also the debugger commands are introduced here. Details about the debugger are presented in detail in Section 3. Section 4 presents our conclusion and future work.

2 A debugging Session

This section presents the debugger functionality presenting a debugging session on a short Modelica example. The functionality of the debugger is presented using pictures from the Emacs debugging mode for Modelica (`modelicadebug-mode`).

2.1 The Debugger Commands

The Emacs Modelica debug mode is implemented as a specialization of the Grand Unified Debugger (GUD) interface (`gud-mode`) from Emacs [9]. Because the Modelica debug mode is based on the GUD interface, some of the commands have the same familiar key bindings.

The actual commands sent to the debugger are also presented together with GUD commands preceded by the Modelica debugger prompt: `mdb@>`.

If the debugger commands have several alternatives these are presented using the notation: `alternative1|alternative2|....`

The optional command components are presented using notation: `[optional]`.

In the Emacs interface: `M-x` stands for holding down the `Meta` key (mapped to `Alt` in general) and pressing the key after the dash, here `x`, `C-x` stands for holding down the `Control (Ctrl)` key and pressing `x`, `<RET>` is equivalent with pressing the `Enter` key and `<SPC>` with pressing `Space` key.

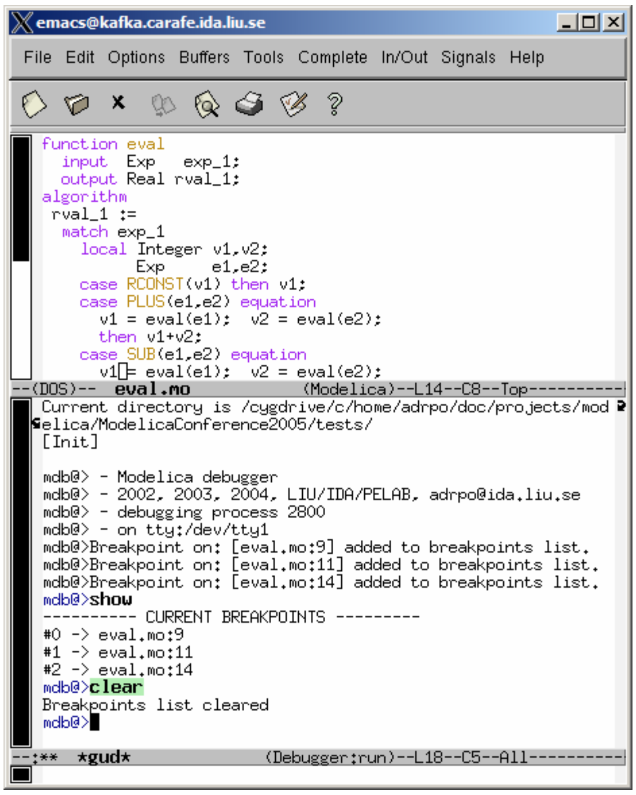
2.1.1 Starting the Modelica Debugging Subprocess

The command for starting the Modelica debugger under Emacs is the following:

```
M-x modelicadebug <RET> executable <RET>
```

2.1.2 Setting/Deleting Breakpoints

A part of a session using this type of commands is shown in Figure 1 below. The presentation of the commands follows.



emacs@kafka.carafe.ida.liu.se
File Edit Options Buffers Tools Complete In/Out Signals Help

```
function eval
  input  Exp   exp_1;
  output Real rval_1;
algorithm
 rval_1 :=
  match exp_1
    local Integer v1,v2;
          Exp     e1,e2;
    case RCONST(v1) then v1;
    case PLUS(e1,e2) equation
      v1 = eval(e1);  v2 = eval(e2);
      then v1+v2;
    case SUB(e1,e2) equation
      v1 = eval(e1);  v2 = eval(e2);
--(DOS)--  eval.mo           (Modelica)--L14--C8--Top---------
Current directory is /cygdrive/c/home/adrpo/doc/projects/mod
elica/ModelicaConference2005/tests/
[Init]

mdb@> - Modelica debugger
mdb@> - 2002, 2003, 2004, LIU/IDA/PELAB, adrpo@ida.liu.se
mdb@> - debugging process 2800
mdb@> - on tty:/dev/tty1
mdb@>Breakpoint on: [eval.mo:9] added to breakpoints list.
mdb@>Breakpoint on: [eval.mo:11] added to breakpoints list.
mdb@>Breakpoint on: [eval.mo:14] added to breakpoints list.
mdb@>show
---------- CURRENT BREAKPOINTS ---------
#0 -> eval.mo:9
#1 -> eval.mo:11
#2 -> eval.mo:14
mdb@>clear
Breakpoints list cleared
mdb@>
--:**  *gud*            (Debugger:run)--L18--C5--All---------
```

Figure 1: Using breakpoints

To set a breakpoint on the line the cursor (point) is at:

```
C-x <SPC>
mdb@> break on file:lineno|string <RET>
```

To delete a breakpoint placed on the current source code line (`gud-remove`):

```
C-c C-d
C-x C-a C-d
mdb@> break off file:lineno|string <RET>
```

Instead of writing `break` one can use alternatives `br|break|breakpoint`.

Alternatively one can delete all breakpoints using:

```
mdb@> cl|clear <RET>
```

Showing all breakpoints:

```
mdb@> sh|show <RET>
```

2.1.3 Stepping and Running

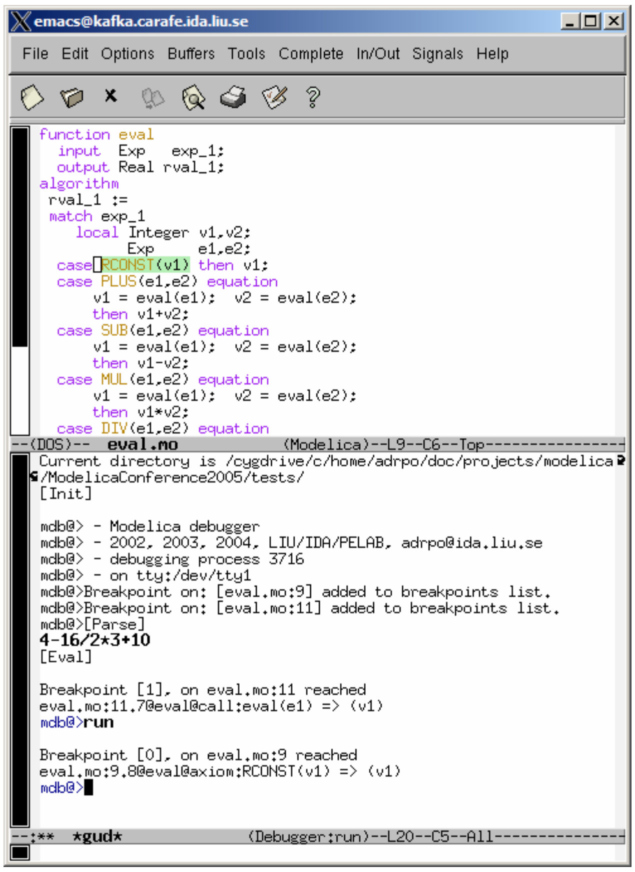
To perform one step (`gud-step`) in the Modelica code:

```
C-c C-s
C-x C-a C-s
mdb@> st|step <RET>
```

To continue after a step or a breakpoint (`gud-cont`) in the Modelica code:

```
C-c C-r
C-x C-a C-r
mdb@> ru|run <RET>
```

Examples of using these commands are presented in Figure 2. The example uses an extended version of Modelica [10] and is briefly described in Section 3.1.



emacs@kafka.carafe.ida.liu.se
File Edit Options Buffers Tools Complete In/Out Signals Help

```
function eval
  input  Exp   exp_1;
  output Real rval_1;
algorithm
 rval_1 :=
 match exp_1
    local Integer v1,v2;
          Exp     e1,e2;
  case RCONST(v1) then v1;
  case PLUS(e1,e2) equation
      v1 = eval(e1);  v2 = eval(e2);
      then v1+v2;
  case SUB(e1,e2) equation
      v1 = eval(e1);  v2 = eval(e2);
      then v1-v2;
  case MUL(e1,e2) equation
      v1 = eval(e1);  v2 = eval(e2);
      then v1*v2;
  case DIV(e1,e2) equation
--(DOS)--  eval.mo         (Modelica)--L9--C6--Top--------------
Current directory is /cygdrive/c/home/adrpo/doc/projects/modelica
/ModelicaConference2005/tests/
[Init]

mdb@> - Modelica debugger
mdb@> - 2002, 2003, 2004, LIU/IDA/PELAB, adrpo@ida.liu.se
mdb@> - debugging process 3716
mdb@> - on tty:/dev/tty1
mdb@>Breakpoint on: [eval.mo:9] added to breakpoints list.
mdb@>Breakpoint on: [eval.mo:11] added to breakpoints list.
mdb@>[Parse]
4-16/2*3+10
[Eval]

Breakpoint [1], on eval.mo:11 reached
eval.mo:11.7@eval@call:eval(e1) => (v1)
mdb@>run

Breakpoint [0], on eval.mo:9 reached
eval.mo:9.8@eval@axiom:RCONST(v1) => (v1)
mdb@>
--:**  *gud*          (Debugger:run)--L20--C5--All--------------
```

Figure 2: Stepping and running

2.1.4 Examining Data

There are no GUD keybindings for these commands but they are inspired from the GNU Project debugger (GDB) [2].

To print the contents/size of a variable one can write:

```
mdb@> pr|print variable_name <RET>
mdb@> sz|sizeof variable_name <RET>
```

at the debugger prompt. The size is displayed in bytes.

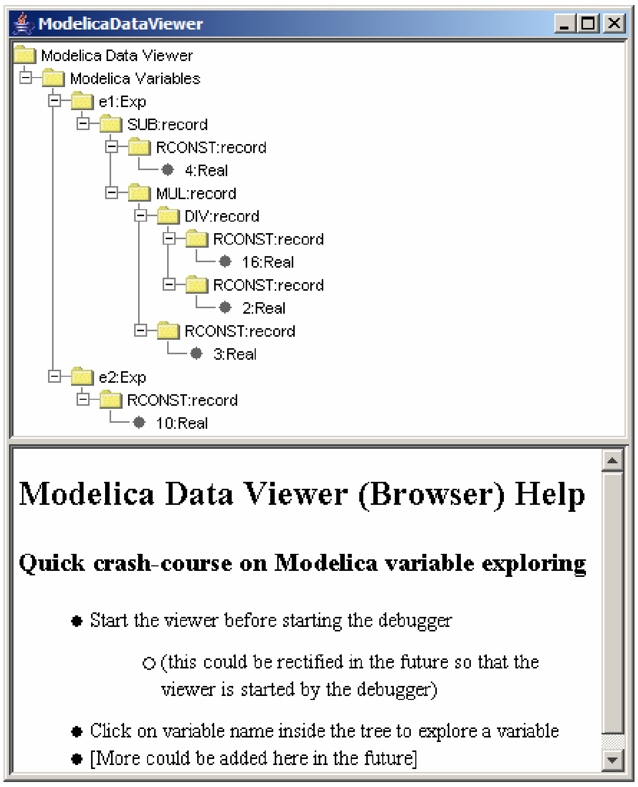
Variable values to be printed can be of a complex type and very large. One can restrict the depth of printing using:

```
mdb@> [set] de|depth integer <RET>
```

Moreover, we have implemented an external viewer written in Java called `ModelicaDataViewer` to browse the contents of such a large variable. To send the contents of a variable to the external viewer for inspection one can use the command:

```
mdb@> bw|browse|gr|graph var_name <RET>
```

at the debugger prompt. The debugger will try to connect to the `ModelicaDataViewer` and send the contents of the variable. The external data browser has to be started a priori. If the debugger cannot connect to the external viewer within a specified timeout a warning message will be displayed. A picture with the external `ModelicaDataViewer` tool is presented in Figure 3 below:



```
Modelica Data Viewer
  Modelica Variables
    e1:Exp
      SUB:record
        RCONST:record
          4:Real
        MUL:record
          DIV:record
            RCONST:record
              16:Real
            RCONST:record
              2:Real
          RCONST:record
            3:Real
    e2:Exp
      RCONST:record
        10:Real
```

Modelica Data Viewer (Browser) Help

Quick crash-course on Modelica variable exploring

- Start the viewer before starting the debugger
  - (this could be rectified in the future so that the viewer is started by the debugger)
- Click on variable name inside the tree to explore a variable
- [More could be added here in the future]

Figure 3: External variable browser

If the variable which one tries to print does not exist in the current scope (not a live variable) a notifying warning message will be displayed.

Automatic printing of variables at every step or breakpoint can be specified by adding a variable to a display list:

```
mdb@> di|display variable_name <RET>
```

To print the entire display list:

```
mdb@> di|display <RET>
```

Removing a display variable from the display list:

```
mdb@> un|undisplay variable_name <RET>
```

Removing all variables from the display list:

```
mdb@> undisplay <RET>
```

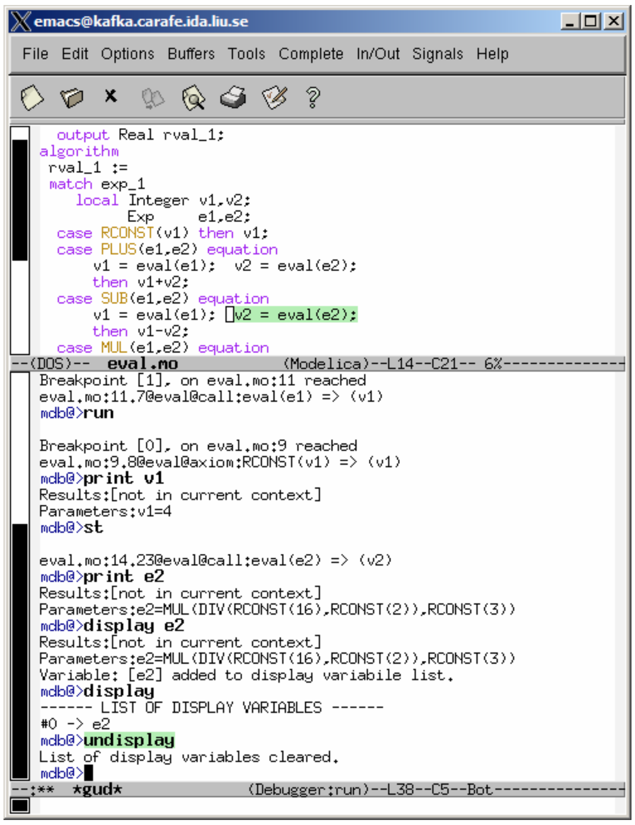
Printing the current live variables:

```
mdb@> li|live|livevars <RET>
```

Instructing the debugger to print or to disable the print of the live variable names at each step/breapoint:

```
mdb@> [set] li|live|livevars [on|off]<RET>
```

Figure 4 shows examples of some of these commands within a debugging session:


emacs@kafka.carafe.ida.liu.se
File Edit Options Buffers Tools Complete In/Out Signals Help

```
  output Real rval_1;
algorithm
 rval_1 :=
 match exp_1
    local Integer v1,v2;
          Exp     e1,e2;
  case RCONST(v1) then v1;
  case PLUS(e1,e2) equation
      v1 = eval(e1);  v2 = eval(e2);
      then v1+v2;
  case SUB(e1,e2) equation
      v1 = eval(e1); v2 = eval(e2);
      then v1-v2;
  case MUL(e1,e2) equation
--(DOS)--  eval.mo          (Modelica)--L14--C21-- 6%-------------
Breakpoint [1], on eval.mo:11 reached
eval.mo:11.7@eval@call:eval(e1) => (v1)
mdb@>run

Breakpoint [0], on eval.mo:9 reached
eval.mo:9.8@eval@axiom:RCONST(v1) => (v1)
mdb@>print v1
Results:[not in current context]
Parameters:v1=4
mdb@>st

eval.mo:14.23@eval@call:eval(e2) => (v2)
mdb@>print e2
Results:[not in current context]
Parameters:e2=MUL(DIV(RCONST(16),RCONST(2)),RCONST(3))
mdb@>display e2
Results:[not in current context]
Parameters:e2=MUL(DIV(RCONST(16),RCONST(2)),RCONST(3))
Variable: [e2] added to display variabile list.
mdb@>display
------ LIST OF DISPLAY VARIABLES ------
#0 -> e2
mdb@>undisplay
List of display variables cleared.
mdb@>
--:**  *gud*          (Debugger:run)--L38--C5--Bot--------------
```

Figure 4: Examining data

2.1.5 Additional commands

The stack contents (backtrace) can be displayed using:

```
mdb@> bt|backtrace <RET>
```

Because the contents of the stack can be quite large, one can print a filtered view of it:

```
mdb@> fbt|fbacktrace filter_string <RET>
```

Also, one can restrict the numbers of entries the debugger is storing using:

```
mdb@> maxbt|maxbacktrace integer <RET>
```

For displaying the status of the Modelica runtime:

```
mdb@> sts|stat|status <RET>
```

The status of the extended Modelica runtime comprises information regarding the garbage collector, allocated memory, stack usage, etc.

The current debugging settings can be displayed using:

```
mdb@> stg|settings <RET>
```

The settings printed are: the maximum remembered backtrace entries, the depth of variable printing, the current breakpoints, the live variables, the list of the display variables and the status of the runtime system.

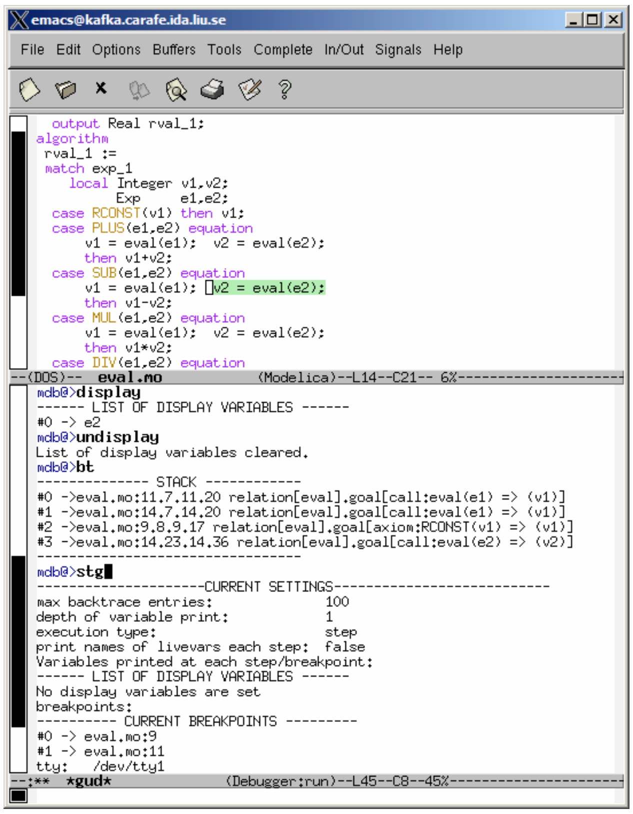
One can invoke the debugging help by issuing:

```
mdb@> he|help <RET>
```

For leaving the debugger one can use the command:

```
mdb@> qu|quit|ex|exit|by|bye <RET>
```

A session using these commands is presented in Figure 5 below:


emacs@kafka.carafe.ida.liu.se
File Edit Options Buffers Tools Complete In/Out Signals Help

```
  output Real rval_1;
algorithm
 rval_1 :=
 match exp_1
    local Integer v1,v2;
          Exp     e1,e2;
  case RCONST(v1) then v1;
  case PLUS(e1,e2) equation
      v1 = eval(e1);  v2 = eval(e2);
      then v1+v2;
  case SUB(e1,e2) equation
      v1 = eval(e1); v2 = eval(e2);
      then v1-v2;
  case MUL(e1,e2) equation
      v1 = eval(e1);  v2 = eval(e2);
      then v1*v2;
  case DIV(e1,e2) equation
--(DOS)--  eval.mo              (Modelica)--L14--C21-- 6%--------------------
mdb@>display
------ LIST OF DISPLAY VARIABLES ------
#0 -> e2
mdb@>undisplay
List of display variables cleared.
mdb@>bt
-------------- STACK ------------
#0 ->eval.mo:11.7.11.20 relation[eval].goal[call:eval(e1) => (v1)]
#1 ->eval.mo:14.7.14.20 relation[eval].goal[call:eval(e1) => (v1)]
#2 ->eval.mo:9.8.9.17 relation[eval].goal[axiom:RCONST(v1) => (v1)]
#3 ->eval.mo:14.23.14.36 relation[eval].goal[call:eval(e2) => (v2)]
---------------------------------
mdb@>stg
--------------------CURRENT SETTINGS---------------------------
max backtrace entries:              100
depth of variable print:            1
execution type:                     step
print names of livevars each step:  false
Variables printed at each step/breakpoint:
------ LIST OF DISPLAY VARIABLES ------
No display variables are set
breakpoints:
---------- CURRENT BREAKPOINTS ---------
#0 -> eval.mo:9
#1 -> eval.mo:11
tty:   /dev/tty1
--:**  *gud*              (Debugger:run)--L45--C8--45%--------------------
```

Figure 5: Additional commands

3 The Debugger Implementation

This section presents the debugging strategy in detail. We first start with two examples on how the debugger instruments the code, and then we enter into the details of the implementation. The examples illustrate Modelica algorithmic code and some of the new extensions of the Modelica language like pattern matching and union type declarations on a simple expression evaluator example.

3.1 Example Applications to Debug

In this section we present two examples of Modelica algorithms.

3.1.1 Bubble Sort function

The first example application we present for debugging is a BubbleSort function:

```
function bubbleSort
 input Real [:] unordElem;
 output Real [size(unordElem, 1)] ordElem;
 protected
   Real tempVal;
   Boolean isOver = false;
 algorithm
   ordElem := unordElem;
   while not isOver loop
     isOver := true;
     for i in 1:size(ordElem, 1)-1 loop
      if ordElem[i] > ordElem[i+1]
      then
         tempVal      := ordElem[i];
         ordElem[i]   := ordElem[i+1];
         ordElem[i+1] := tempVal;
         isOver := false;
      end if;
     end for;
   end while;
end bubbleSort;
```

The instrumented version of this function is presented below:

```
function bubbleSort
 input Real [:] unordElem;
 output Real [size(unordElem, 1)] ordElem;
 protected
   Real tempVal;
   Boolean isOver = false;
 algorithm
 Debug.register_in("unordElem",unordElem);
 Debug.step(...);
 ordElem := unordElem;
 Debug.register_out("ordElem", ordElem);
 Debug.register_in("isOver", isOver);
 Debug.step(...);
 while not isOver loop
     isOver := true;
     Debug.register_out("isOver", isOver);
     Debug.register_in("ordElem",ordElem);
     Debug.step(...);
     for i in 1:size(ordElem, 1)-1 loop
```

```
        Debug.register_out("i", i);
        Debug.register_in("i", i);
        Debug.register_in("ordElem[i]",
                          ordElem[i]);
        Debug.register_in("ordElem[i+1]",
                          ordElem[i+1]);
        Debug.step(...);
        if ordElem[i] > ordElem[i+1]
        then
          Debug.register_in("i", i);
          Debug.register_in(
            "ordElem[i]",
             ordElem[i]);
          Debug.step(...);
          tempVal := ordElem[i];
          Debug.register_out(
            "tempVal",
            tempVal);
          Debug.register_in("i", i);
          Debug.register_in(
                "ordElem[i+1]",
                ordElem[i+1]);
          Debug.step(...);
          ordElem[i] := ordElem[i+1];
          Debug.register_out("i", i);
          Debug.register_out(
                "ordElem[i]",
                ordElem[i]);
          Debug.register_in("i", i);
          Debug.register_in(
                "tempVal",
                tempVal);
          Debug.step(...);
          ordElem[i+1] := tempVal;
          Debug.register_out("i", i);
          Debug.register_out(
                "ordElem[i+1]",
                ordElem[i+1]);
          Debug.step(...);
          isOver := false;
          Debug.register_out("isOver",
                             isOver);
          Debug.step(...);
        end if;
      end for;
      Debug.register_out("i", i);
      Debug.register_out(
                "ordElem",
                ordElem);
      Debug.step(...);
    end while;
    Debug.register_out("isOver", isOver);
    Debug.register_out("ordElem",ordElem);
    Debug.step(...);
 end bubbleSort;
```

As presented above, the debugger instruments all statements using the defined functions from the `Debug` package. A statement is analyzed for input and output variables which are registered with the debugging framework using `register_in` and `register_out` functions. The function `step` verify internally if we have to stop at a breakpoint or continue without stopping and also is responsible for parsing user commands addressed to the debugger. The instrumentation is better than what a programmer/user would do with print or assert statements because it provides better control through stop/inspect functionality. As one can see the code grows quite much when is instrumented, but this does not affect the final version of the code. For debugging purposes the user is rather interested in correctness of the code than in the speed/size of the code.

3.1.2 An expression evaluator

The second application is an expression evaluator implemented in the algorithmic Modelica subset extended with support for recursive tree data structures and a case-expression construct that allows pattern-matching and tree traversal. These language extensions are described in a companion paper [10] and are independent of the implemented debugger described here. For the sake of completeness we make present the extensions briefly in the following.

The declaration of an abstract syntax tree (AST) data type Exp for representing simple expressions:

```
uniontype Exp
 record RCONST Real x1; end RCONST;
 record PLUS   Exp x1; Exp x2; end PLUS;
 record SUB    Exp x1; Exp x2; end SUB;
 record MUL    Exp x1; Exp x2; end MUL;
 record DIV    Exp x1; Exp x2; end DIV;
 record NEG    Exp x1;         end NEG;
end Exp;
```

The union type declaration above is defining record constructors for the nodes of the simple expression representation. Examples of expressions represented in this way can be found in the following table:

Expression	Modelica constructor form
1+2	`PLUS(RCONST(1), RCONST(2))`
1-2/3	`SUB(RCONST(1), DIV(RCONST(2), RCONST(3)))`

Table 1: Representing simple expression trees

To be able to evaluate simple expression trees we need an evaluation function. The evaluation function will apply pattern matching on the constructors of the expression language and then perform the actual evaluation on the components of the constructor.

Below we present the evaluation function `eval` of our simple expression evaluator:

```
function eval
  input  Exp   exp_1;
  output Real rval_1;
algorithm
 rval_1 :=
  match exp_1
    local Real v1,v2;
          Exp  e1,e2;
    case RCONST(v1) then v1;
    case PLUS(e1,e2) equation
      v1 = eval(e1);  v2 = eval(e2);
      then v1+v2;
    case SUB(e1,e2) equation
      v1 = eval(e1);  v2 = eval(e2);
      then v1-v2;
    case MUL(e1,e2) equation
      v1 = eval(e1);  v2 = eval(e2);
      then v1*v2;
    case DIV(e1,e2) equation
      v1 = eval(e1);  v2 = eval(e2);
      then v1/v2;
    case NEG(e1) equation
      v1 = eval(e1);
      then -v1;
    else
      assert("Bad expression!"));
   end match;
 end eval;
```

This function has as input an expression in the form presented in Table 1, second column. The expressions are represented as trees using constructors defined in the union type `Exp`. A model that uses this function is presented below:

```
 model Example
   Exp exp=PLUS(
            SUB(RCONST(4),
                MUL(DIV(RCONST(16),
                          RCONST(2)),
                     RCONST(3))),
            RCONST(10));
   Real result;
 algorithm
   result := eval(exp);
 end Example
```

The first component of the `Example` model defines a simple tree that corresponds to `4-16/2*3+10` expression. We used this simple expression in the examples and figures in Section 2.

The instrumented code of the eval function is presented below. The debugging code is underlined to be more visible:

```
 function eval  // instrumented version
   import Modelica.Debugging;
   input  Exp     exp_1;
   output Real realval_1;
algorithm
 Debug.register_in("exp1", exp_1);
 Debug.step(...);
 realval_1 :=
   match exp_1
     local Real v1,v2;
     Exp e1,e2;
     case RCONST(v1) equation
       Debug.register_out("v1", v1);
       Debug.step(...);
      then v1;
     case PLUSop(e1,e2) equation
       Debug.register_out("e1", e1);
       Debug.register_out("e2", e2);
       Debug.register_in("e1", e1);
       Debug.step(...);
       v1 = eval(e1);
       Debug.register_out("v1", v1);
       Debug.resister_in("e2", e2);
       Debug.step(...);
       v2 = eval(e2);
       Debug.register_out("v2", v1);
       Debug.register_out("v1+v2",v1+v2);
       Debug.step(...);
      then v1+v2;
      ...
    else
      Debug.step(...);
      assert("Bad expression!"));
   end match;
 Debug.register_out("realval_1",realval_1);
 Debug.step();
end eval;
```

As one can see, debugging code is added for each variable. This style of debug code instrumentation can be changed into one where all the debugging calls are collapsed into just one call `Debug.step(...)` with more arguments specifying in or out variables. We have experimented with different debug instrumentation styles in order to choose the best speed vs. memory consumption for the debugger.

The following instrumentation functions are inserted into the generated code:

- The functions: `Debug.register_in("name", var)` and `Debug.register_out("name", var)` register in a data structure the variables which are live at a certain moment during the execution.
- The `Debug.step(...)` function then performs a query of this data structure to show which variables are available in the current context.
- The function `Debug.register_in(...)` registers variables that are used in the next statement or expression.
- The function `Debug.register_out(...)` registers variables that result from the execution of the previous statement or expression.

Note that the debug instrumentation functions are low-level C functions that do not fulfill the Modelica requirement of being mathematical functions.

3.2 Overview

In this section we present the compilation path followed by the compiler when instrumenting the code with debugging calls. The debugger is actually the executable generated by the compiler when instructed to generate debugging calls before and after each relevant Modelica statement or expression.

Figure 6 presents both the normal compilation path performed by the compiler when compiling algorithmic code and also the path followed by the compiler when compiling algorithm sections that include debugging information.

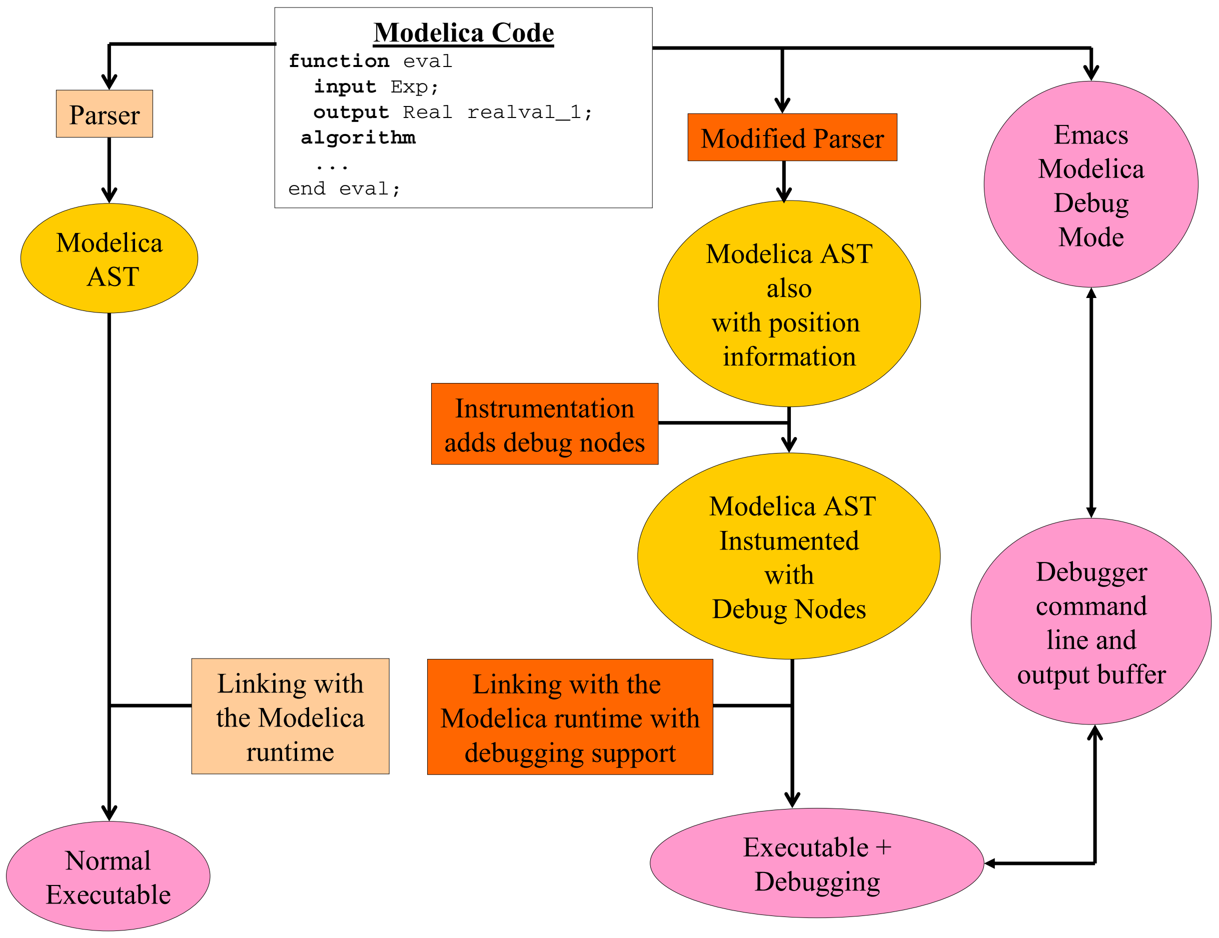

Modelica Code
function eval
input Exp;
output Real realval_1;
algorithm
...
end eval;
Parser
Modelica AST
Modified Parser
Modelica AST also with position information
Emacs Modelica Debug Mode
Instrumentation adds debug nodes
Modelica AST Instumented with Debug Nodes
Debugger command line and output buffer
Linking with the Modelica runtime
Linking with the Modelica runtime with debugging support
Normal Executable
Executable + Debugging

Figure 6: Normal compilation (left) and compilation with debug support (right)

An overview of our debugging strategy is presented on the right side of Figure 6. The path taken by the debugger comprises several more steps in order to instrument the Modelica AST with debug nodes and live variable information. Also, the runtime system is extended with several data structures that support debugging and a parser for the debugger commands.

3.3 Augmenting the Modelica AST with Debug Call Nodes

The modified Modelica parser saves additional position information about each statement or expression. This information is used by the debug instrumentation phase in order to generate calls to the debugger functions with exact information on where the current execution is taking place.

We use a very simple and effective algorithm when instrumenting the ModelicaAST with debug nodes. We sketch a pseudo code of this algorithm below:

```
foreach ModelicaAST expression or
        statement Node
 if not filter(Node)
 then select next Node;
 else
  variables_in = collect_variables(Node);
  variables_out = collect_variables(Node);
  position = collect_position(Node);
  construct new tree with these nodes:
    Debug.register_in(variables_in);
    Debug.step(position);
    Node;
    Debug.register_out(variables_out);
  replace Node with the new tree;
 end else;
end foreach;
```

The compiler can be instructed to generate debugging nodes only when reaching certain nodes that are conform to a filter. Using this facility one can tell the compiler to perform debug instrumentation only on a certain function or a certain statement of the code. In this way the delay in the execution speed introduced by the debugging code can be kept to a minimum.

We have experimented with several ways of creating the added debug nodes:

- Each variable is registered using a debug function call either as `in` or `out` variable, as in the pseudo code presented.
- All `in` variables are collected in a list and passed to a single function call. The same approach is used also for `out` variables. This has an impact on mem-

ory consumption, but uses fewer function calls, so it is faster.

- The `in` and `out` variables are collected in two lists which are passed as arguments to the `step` function directly.

The best speed vs. memory consumption is highly dependent on the algorithmic code. However one can experiment with all these choices and choose the best debug instrumentation way for a specific code.

3.4 Short Presentation of the Debugger Library

The debugger library contains several functions implementing the actual debugger functionality and data structures for bookkeeping of breakpoints, live variables, call stack, types of live variables, etc. The library is implemented in C.

The debugger library has the following available functions:

- `Debug.step(...)` function with fixed parameters `file_name, lineno, columno, function_name, next_statement` is stopping the execution of the algorithmic code if a breakpoint is reached or one step was performed and waits for commands from the user. If the commands are not `step` or `run` it waits for more commands from the user in a cycle. Additional parameters like the live variables can be also passed to the `Debug.step(...)` function depending on the debug instrumentation choice.
- `Debug.register_in(...)` function and `Debug.register_out(...)` function are used only when no live variables are passed to the `Debug.step()` function. These functions are registering variables, either each variable at a time or several variables as a list.
- `Debug.parse_command()` is called by `Debug.step()` either at a step or when a breakpoint is reached.
- `Debug.execute_command()` is called by the `Debug.parse_command()` when the user issues a command. Depending on the command several other functions are executed.
- `Debug.set_breakpoint()` adds the breakpoint into the breakpoint list.
- `Debug.delete_breakpoint()` deletes the breakpoint from the breakpoint list.
- `Debug.clear()` and `Debug.show()` clears or shows all current breakpoints, respectively.
- `Debug.print_variable()` prints the specified variable to the debugger output.
- `Debug.print_livevars_names()` prints the names of the variables available in the current context. The distinction between `in` (parameters) and `out` (results) variables is made when printing variable names.
- `Debug.browse_variable()` connects to the external viewer, and sends on demand the value of a variable. This function stops the debugger until the external viewer is done with the browsing.
- `Debug.set_print_depth()` sets the depth of variable printing.
- `Debug.max_remembered_stack_entries()` will set the maximum number of entries of the logged stack trace.
- `Debug.display_variable()` will add the display variable to a display list to be printed at each step or breakpoint.
- `Debug.undisplay_variable()` performs the inverse action of the `Debug.display_variable()`.
- `Debug.display()` prints the list of variable names present in the display list.
- `Debug.undisplay()` clears the display list.
- `Debug.stack_add_node()` pushes a node name on the stack trace.
- `Debug.stack_remove_node()` pops a node name from the stack trace.
- `Debug.status()` prints status information on the extended Modelica runtime, e.g., garbage collections performed, amount of allocated memory, etc.
- `Debug.settings()` prints the current debugger settings.

More functions are actually present in the debug library (dealing with variable – type mapping, connection to the external viewer, etc). Here we have only presented a limited set which has direct connections with the debugger commands presented in the paper.

4 Conclusions and Future Work

We have presented a portable and highly configurable debugger for extended Modelica algorithmic code. Debugging of large algorithmic Modelica codes is now possible using our debugger.

As future work we consider extension of the current debugging scheme and also tighter integration of the debugger with other Modelica tools.

Integration with declarative equation debugger techniques [1, 2] will be provided in the future, in order to address debugging of the entire Modelica language from a central debugger.

We have also started work to integrate the debugger and the OpenModelica [3] compiler within the Eclipse Development platform [11] which will provide integrated editing, navigation, simulation and debugging for the Modelica language.

Acknowledgements

This research was partially funded by Foundation for Strategic Research (SSF) in the Research in International Software Engineering (RISE) project and by the Swedish National Graduate School in Computer Science (CUGS). We would also like to thank the reviewers for their feedback.

References

1. Bunus, P., *Debugging and Structural Analysis of Declarative Equation-Based Languages*, in *Department of Computer and Information Science*. 2002, Linköping University: Linköping, Licentiate Thesis.

2. Bunus, P., *Debugging Techniques for Equation-Based Languages*, in *Department of Computer and Information Science*. 2004, May, Linköping University: Linköping, PhD Thesis.

3. Fritzson, P., et al. *The Open Source Modelica Project*. in *Proceedings of The 2th International Modelica Conference*, 18-19 March, 2002. Munich, Germany, http://www.ida.liu.se/~pelab/modelica/OpenModelica.html.

4. Fritzson, P., *Principles of Object-Oriented Modeling and Simulation with Modelica*. 2003: Wiley-IEEE Press.

5. GNU, T.F.S.F., *The GNU Project debugger*, Last Accessed: December, 2004, http://www.gnu.org/software/gdb/gdb.html.

6. Tolmach, A.P., *Debugging Standard ML*. 1992, October, Princeton University, PhD. Thesis.

7. Pettersson, M. *Portable Debugging and Profiling*. in *7th International Conference on Compiler Construction*, 1998.

8. Hanson, D.R. and M. Raghavachari, *A Machine-Independent Debugger—Revisited.* Software—Practice and Experience, 1999. **29**(10): p. 849-862.

9. GNU, T.F.S.F., *Emacs, The Grand Unified Debuger (GUD)*, Last Accessed: December, 2004, http://www.gnu.org/software/emacs/manual/html_node/Debuggers.html#Debuggers.

10. Fritzson, P., A. Pop, and P. Aronsson. *Towards Comprehensive Meta-Modeling and Meta-Programming Capabilities in Modelica, (to appear)*. in *4th International Modelica Conference*, 7-8 March, 2005. Hamburg, Germany.

11. *Eclipse Development Platform*, Last Accessed: December, 2004, http://www.eclipse.org/.

ModelicaDB - A Tool for Searching, Analysing, Crossreferencing and Checking of Modelica Libraries

Olof Johansson, Adrian Pop, Peter Fritzson
PELAB – Programming Environment Lab, Dept. Computer Science
Linköping University, S-581 83 Linköping, Sweden
{olojo,adrpo,petfr}@ida.liu.se

Abstract

This paper presents ModelicaDB, a tool that provides several kinds of queries on repositories of Modelica models.

The Modelica language has a growing user community that produce a large and increasing code base of models.

However, the reuse of models within the Modelica community can be greatly hampered in the future if there are no tools to address a number of management issues (i.e. scalable searching, analysing, crossreferencing, checking, etc) of such a large repository of models.

We try to address these issues by providing the Modelica community with a ModelicaDB database for storing models and services for quering this database to perform a wide range of model engineering tasks in a scalable fashion.

In the long-term, this work also aims at providing integration between Modelica tools and advanced product development processes that rely on database technology.

1 Introduction

The long-term goal of this work is to efficiently integrate Modelica modeling simulation environments into the overall product development process for products that require advanced systems engineering [2].

There are many engineering processes that precede modeling and simulation, and which output information that defines structure, configuration, and input parameter data to simulation models of the product.

The following are of particular importance:

- The implementation structure of the product that defines its hierarchical division into different subsystems, their components with parameter values, and component interconnections.
- Run-cases derived from the product requirements, which define critical behaviour that the product must achieve, and for which alternative implementation structures' behaviour is evaluated with simulation.

Given these, the simulation model designer can select valid component models from Modelica libraries for the components in the implementation structure, and verify that the simulation component's parameter values are compatible and transferable from the information in the provided implementation structure.

With valid component models assigned and mappings of their parameters, other tools can assemble a final simulation model setup for execution and post-processing of the simulation results for evaluation analysis in subsequent engineering processes.

One purpose of ModelicaDB is to provide fast access to possibly relevant component models in Modelica Libraries, such that the assignment work can be speeded up with automation tools.

In many cases, a matching model component will not be available and ready for use in the Modelica Libraries, so the task of selecting component models is augmented by writing new ones or assembling valid component models from other components in the Modelica library.

Such work is a creative design task, which is significantly aided if the designer has tools for searching, analysing, crossreferencing, and checking the libraries.

The used libraries have been developed by experienced library developers, and contain valuable design-pattern knowledge of how to properly design and implement models, components and libraries. With fast browsing and navigation tools, the designer can quickly find similar designs to the one that is needed, study how they are used/reused as components in other simulation models, and get a good understanding of how to build a new simulation component.

The continuous development and improvement of Modelica libraries by the Modelica design group and similar efforts within companies, indicates that tools with ModelicaDB functionality would be valuable to the Modelica community [7] for many other purposes than we intend here.

Preliminary statistics from the Modelica 2.1 and Modelica 1.5 libraries definitely show that this kind of tool would be helpful for many engineers for grasping, sharing, reusing, and following the large efforts in simulation model development work that is being simultaneously conducted by many people.

The following sections of the paper provide statistics from the current releases of the Modelica libraries, examples of use-cases for ModelicaDB, an overview of the product development process and the intended role of ModelicaDB, the functionality in the user interface of ModelicaDB, an example of an SQL-query on the database and finally results, experiences, future work and conclusions.

2 Statistics from Modelica Standard Libraries

Preliminary analysis of the Standard Modelica 1.5 and 2.1 libraries give the following statistics:

Modelica Library	V 1_5	V 2_1	V 2_1+
Source files	36	87	144
Imports	93	286	343
Class definitions	910	1447	3141
Components	1628	4636	6915
Equations	1055	2768	3262
Algorithms	99	633	1290
Component_refs	30304	60838	92636
Expression_lists	14736	23715	25354
Real literals	4413	5833	33158
Comments	1720	4755	5649
String Comments	1322	3722	5611
Annotations	1326	3120	5093
String literals	3503	7218	13350
Integer literals	33187	59604	67373
Other	88991	157760	235806
Total elements:	183323	336422	499125

The number columns show the Modelica language element count from different releases of the Modelica standard libraries. Modelica 1_5 was downloaded from the public library page [8]. Modelica 2_1 and 2_1+ were obtained from the Modelica CVS repository 2004-11-15.

V 2_1+ includes the following libraries: Modelica, ModelicaReference, ModelicaTest, Modelica_Fluid, Modelica_Interpolation, Modelica_Media and TeachingMaterial.

The source code directory contents of the libraries was converted to a single xml file for each library release by ModelicaXML, which then were preprocessed for import into ModelicaDB.

The *Imports* row is an indicator of reuse. The Component_refs row gives the count of the uses of a component variables in expressions.

The *Comment* row is a higher level parse node for *String_comments* and *Annotations* .

String literals and *Integer literals* are heavily used within annotations, especially for graphical object annotations in Modelica diagrams.

The above statistics shows that the size of the standard libraries is substantial. Commercial Modelica development tools [1],[3] provide user interfaces with tree views of the package hierarchy, connection diagrams, and string based text searches, for quick navigation in the libraries.

ModelicaDB adds additional search facilities and other types of tree views on the libraries, that can help to speed up the task of creating a new simulation component that efficiently reuses existing component models and design-pattern knowledge.

3 Use-Case Examples for ModelicaDB

The following section briefly describes use-cases that illustrate use of additional types of views on Modelica library structures. The views are computed in ModelicaDB, and presented in a tree- or list- based user interface that enables quick navigation with pointing and clicking.

3.1 Finding Relevant Simulation Components

The following example use cases illustrate subtasks in the process of finding reusable components and code sections for building a new simulation component.

- Finding component models with knowledge of the SI-units their instances will need.
- Finding component models with knowledge of their connectors.
- Finding equations with knowledge of the type of the variables used in the expression.
- Finding algorithms with knowledge of their function call parameters.

3.2 Finding Relevant Component Using a Categorization Tree

Categorization trees are an "add-on" feature to Modelica libraries, implemented with annotations.

Categorization trees allow a user to with a few clicks down the tree find a set of relevant component models. The categorization tree itself is an aid for remembering where to find certain components.

Modelica objects can be annotated with a category, which makes them easier to find with the aid of a category system (also known as classification system, focusing on some aspect). Examples of categories are "Electrical Components", "Motors", "Transistors", "Capacitors" etc.

A category system is organized into a tree, where the root category node contains all Modelica objects that have that type of category or any of its sub categories. Sub nodes in the category tree increase the specialization in the categorization. Leaf nodes in the categorization tree usually justify their existence if there are 5-25 component models under this node.

There are many standardized categorization systems used in industry. Classification trees applied on electrical components are specified by IEC [22], and applied in succeding industrial standards like RosettaNet Technical Dictionary which contains a much larger library of classes [18]. ISO-31 [23] categorizes quantities and units into 13 chapters and is well known from the Modelica SIunits in the standard libraries. There are other examples of large classifications systems for standard terms used in e-business.

Commercial design tools for the design processes immediately before simulation, like process and instrumentation diagrams (P&ID), electrical design and control system design usually contain a categorization system for reusable components in their component library catalogues.

The following use cases examples can be well supported with a classification tree.

- Finding a component model for a certain purpose.
- Finding connectors for a certain purpose.
- Finding equations for a certain purpose.

The category method of finding Modelica components requires a library administrator to manually organize or load a standardized categorization tree , and then annotate the component models with their classifications, see section 3.7.

Once the classification tree structure is decided, pattern maching searches in the Modelica repository can be used to populate the categories. For example, a classification tree for equations that compute a value of a certain SIunit type, can be organized according to ISO-31 and the standard SI-units library, and populated with equations whose left hand side variable matches the corresponding SIunit type category.

3.3 Verifying that a component is trustworthy

Simulation results must be accurate in order to provide correct decision support to the product design process. The following use-cases illustrate ways the engineer can determine this.

- Finding examples of usage
- Computing statistics of reuse in other models
- Computing statistics of use of certain design pattern

The last use-case applies when the engineer has designed a new simulation component, and wants to check to what extent others library developers have used a similar design pattern. Such statistics can indicate if this is a good way to solve the problem, and can direct more detailed searches for gaining further confidence, or ideas of how to improve a design.

3.4 Finding relevant design patterns

There are many ways to solve a type of problem. Some of these may prove to be better than others and tend to re-occcur in many places as design patterns. The characteristics of components that play a certain role in the re-occuring interconnection structure of the design pattern, can be used as search criteria.

3.5 Finding relevant naming conventions and documentation

Following naming conventions is important for efficient communication in a large community. Naming conventions usually vary between different engineering disciplines due to the history of their body of knowledge and decisions made by library authors.

When extending or reusing a library, it is valuable to follow the relevant conventions to ease reuse within the community. The use-cases below illustrate how this could be supported.

- Finding Naming Conventions for Variables and Parameters

Pattern search of sorted listings of variable names for a certain type of SI-unit variable, which may play a certain role in an equation.

- Finding References to Literature and Documentation

Searching documentation strings of pattern matched components for references (e.g. brackets or other text patterns that indicate a reference)

3.6 Checking that the New Component Follows Design Rules

Engineering domains may pose restrictions on simulation models that are not possible to enforce directly in Modelica. The same applies for company specific design rules that accumulate from experience, and quality assurance procedures that reduce the cost for errors and maintenance.

The following use-cases are simple examples of design rule checking.

3.6.1 Checking Naming Conventions for Classes, Components

Pattern search and listing of Modelica object identifiers names in a model that do not follow a certain style, or convention.

3.6.2 Checking Complete Documentation

Pattern search that for instance all components in a class have a comment string etc.

3.6.3 Checking Use of SIunits

Pattern search for variables whose type is not derived from SIunits, and are not an array index or similar.

3.7 Managing Product Specific Library Development

While developing a complex product that requires systems engineering, much can be gained by reducing the number of variants of a certain type of simulation component.

The following use cases show how a library developer can direct the users to the best components for various purposes, and identify targets for refactoring amongst existing components.

3.7.1 Finding Candidate Components for Categorization

Various pattern searches that detect component features that make them interesting for a certain classification, and perhaps exclude already classified components

3.7.2 Finding Duplicates or Variants of the Same Models

Duplicates or variants of the same models can be found by pattern searches that compares component sets of variable types and equation patterns within a class definition. Patterns that detect the same equations, based on variable types, where the variables themselves just have different names.

3.8 Additional Analyses and Metrics

Michael Tiller presented analyses and metrics in [21], which inspire development of additional reporting applications which can be computed with SQL-queries on ModelicaDB.

3.9 Automatic Composition and Configuration of new Models

ModelicaDB augments the work presented in [12] on automatic composition and configuration of new models. Using ModelicaDB, designers can compose new models by blending template like models with configuration information stored in other sources (text or XML files, databases, etc) to create new models which are configured accordingly.

4 The ModelicaDB Context and Architecture

Figure 1 shows the role of ModelicaDB amongst some of its surrounding engineering processes, connected with major workflow arrows. Engineering tools (FMDesign, ModelicaDB, ModelicaXML, Modelica Simulation tool) support some of the processes. Engineering models are stored in files (Simulation program, Modelica libraries) and in databases (FMDesign database, ModelicaDB database).

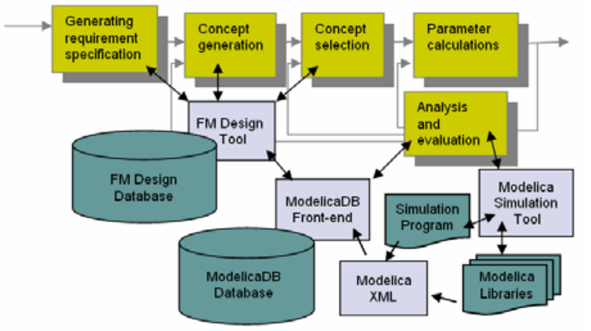


Figure 1. ModelicaDB in its context

The ModelicaDB front-end and database are described in more detail below. We start by briefly describing the role of the other tools in the integrated framework [15].

FMDesign is a tool for designing product concepts with the aid of integrated requirement trees, function-means trees, product concept trees, and implementation trees. The implementation tree specifies the product structure and its interacting components on a level that is detailed enough so its behaviour can be modeled and simulated.

The simulation is deferred to one of the existing Modelica Simulation Tools [1][3][10]. All manual editing of simulation models are performed in one of these tools, and the component models are stored in Modelica Libraries. The simulation program is generated from the configuration information stored in the implementation tree for the product concept.

4.1 ModelicaXML Files

ModelicaXML is a program that converts Modelica source code into XML-files [1]. Recent additions to ModelicaXML is functionality for converting a whole Modelica Library stored in a directory structure into one XML-file. The size of the created files for the standard Modelica libraries version 1.5 and 2.1 is 16 MB and 30 MB respectively.

4.2 ModelicaDB Front-End Tool

This tool parses the ModelicaXML file and builds an object structure in primary memory which can be synchronized or stored into tables in the ModelicaDB database.

The tool also contains a graphical user interface, for fast navigation of the component model level Modelica language constructs.

More detailed constructs like expressions are modelled as parse nodes in the database.

Appendix A shows the class diagram for the UML-model [6] that serves as design specification of ModelicaDB. [20] documents the whole UML-model that was used for generating most of the ModelicaDB specific source code that implements the front end and database. The core specification for designing the ModelicaDB UML model was the ModelicaXML DTD [11]. The reference work used for its documentation was [4] and [9].

The user interface displays the results of queries specific for the use-cases described in section 3 such that found Modelica objects can be quickly inspected, and further navigated, including retrieving and displaying the original Modelica source code from the files. Section 5 gives an overview of the user interface.

4.3 ModelicaDB Database

This is a relational database that is used for processing declarative SQL-queries that do complex searching, compute the analyses, crossreferences, and checks.

The structure of the database is given in the UML class diagram in Appendix A. The database schema can be downloaded from [20] .

The benefit of using an SQL-database instead of navigating parse trees, is that the SQL database optimizer in cooperation with indexes on tables can compute complex queries much faster on a large library, than a traditional procedural or object-oriented program which navigates the parse tree structures.

The performance benefit of a database is first noticed when the number of stored language objects exceed a certain breakpoint.

Writing SQL-queries may be tricky at first, but usually results in little code for a complex task. With a reusable set of SQL-queries for various types of searches and analyses, a new query variant can quickly be written using copy and modify, while verifying that it produces expected results by executing it with paremeters that match a small but well known example model.

5 ModelicaDB Functionality

This section gives a walkthrough of ModelicaDB front-end functionality available in its user interface. An UML class diagram of the user interface design is given in Appendix B. Tree views have a look and feel similar to the windows file explorer, where the folder icons indicate the class restriction or other meta-classes shown in the UML-diagram in Appendix A. When an object shown in a tree view is double-clicked, a form appears which shows the objects attribute values and direct relationships to other objects.

5.1 ModelicaRepository Main Window

This window allows the user to open a Modelica repository file stored in a fast binary format. The user can also login to the database, load the complete repository for caching at the local workstation or synchronize the cached version with the latest changes in the database.

The main window provides a category tree for finding suitable Modelica models, and open a Modelica Model window for these.

Two different types of catalog windows can also be opened from the Modelica repository window. The class catalog window shows categorization trees for Modelica classes, that are organized according to a suitable standard, which allows engineers to quickly find relevant component models for a certain type of product component. Section 3.2 gave a use-case example.

5.2 ModelicaModel Window

The ModelicaModel window provides navigation of all Modelica objects that are recursively owned by a ModelicaModel object, see Appendix A.

In ModelicaDB, a ModelicaModel object is the root of all packages and component models that are assembled within one particular ModelicaXML file. Different versions of the Modelica Standard Libraries, are for example rooted in different ModelicaModel objects in the ModelicaRepository.

The window provides import functionality for ModelicaXML files, and various menu commands for searching, analyzing, crossreferencing, and checking selected Modelica objects.

Below the command menu, the window shows the package hiearchy tree which can be expanded down to class definitions, their components, equations and algorithms.

A separate class browser window can be opened for a selected class, which shows its inheritance hierarchy as a tree. Classes that have no superclasses are shown as parallel root node sorted by the identifier name. Classes that are extended from multiple superclasses, are rooted in the first declared superclass in the tree. The second and remaining extended superclasses are listed together on the level below the class with special object icon, followed by the subclasses that extend the class. Icon superclasses can optionally be filtered away, with a special view setting.

A separate model browser can be opened for a class or component, and shows its part-of structure as a tree. When expanding a component node in the tree, its defining class is shown on the level below, and can be further expanded in a similar way.

The Modelica model window also provides access to various types of two dimensional diagrams, which lay out various structures and interconnections of component models in different views, and are intended as support for seminar discussions on library design and refactoring issues. These diagrams are still in their early design stages, and need some prototype iterations to become useful.

5.3 Report Window for Result Sets

Result sets from searches, analyses, crossreferences and checks, are displayed as interactive report listings in a separate report window with numbered rows. Each row is associated with one object in the Modelica database. If the row is double clicked further details about this object can be inspected in a form. A report row may optionally contain a short text message that further explains the reasons for including its associated object in the report.

Examples of result rows, and their text messages for two use cases is given below.

5.3.1 Finding Component Classes with Knowledge of the SI-units Their Instances Will Need

This is a simple use-case that also can be executed in existing Modelica tools, for instance using the Search facility in Dymola or evaluating a pattern search expression in a MathModelica document cell. This use case can be a benchmark for comparing the time it takes the user to complete the use-case with various user interface implementations.

An instance of this use-case in the ModelicaDB front-end can be as follows:

1) The user has opened the ModelicaModel window on the Modelica 2.1 standard library. In one of the tree-views the users selects the Modelica object that represents Modelica.SIunits.Resistance, which is defined as:

```
type Resistance = Real (
    final quantity="Resistance",
    final unit="Ohm",
    min=0);
```

2) The user issues the *Report* command from the windows top menu, and gets a list of all use cases that can be reported in a dialog box.

3) The user selects "*Find component declarations for predefined types*", which is the short name for this use case.

4) The ModelicaDB front-end processes the query and presents the result rows for the found components in the Report window sorted according to the component variable name. There they can be clicked for further inspection in a form, or set in the focus of one of the available browser window types which better shows a Modelica objects surrounding context.

5.3.2 Computing Statistics of Use of Certain Design Patterns

This is a more complex use-case that illustrates the benefit of storing large Modelica libraries in a relational database.

The use-case instance is checking to what extent other designers have created component models that uses the simulation model of Electromotoric force `Modelica.Electrical.Analog.Basic.EMF` in direct connection with a current sensor component of `Modelica.Electrical.Analog.Sensors.CurrentSensor`.

1) The user has opened the ModelicaModel window on the Modelica 2.1 standard library, and opened the package hierarchy tree down to `Modelica.Electrical.Analog.Basic`. The users selects the EMF class with a first click, and then adds the `CurrentSensor` class to the selection by shift-clicking it in another model browser window showing the `Electrical.Analog` package.

2) The user issues the Report command from the windows top menu, and selects "*Sum connected component uses.*".

3) The ModelicaDB front-end generates an SQL-query with attribute value information from the selected objects as restricting search criteria, sends the query to ModelicaDB and displays the result below in a Report window.

```
0001 ModelicaModel Modelica1_5: count=1
0002 ModelicaModel Modelica2_1: count=1
```

6 SQL-Query Example

The following example show the SQL-queriey for the use case described in 5.3.2.

```
select mmFound.name, count(*)
from class cl1,
     class cl2,
     class clFound,
     modelicamodel mmFound,
     component co1,
     component co2,
     equation eqFound,
     parsenode pn1,
     identifierreference ir1,
     identifierreference ir2
where cl1.identifier = 'EMF'
  and cl1.lowid = co1.classifier_lowid
  and ir1.modelicaobject_lowid = co1.lowid
  and ir1.parsenode_lowid = pn1.lowid
  and pn1.nodeType='equ_connect'
  and ir2.parsenode_lowid = pn1.lowid
  and ir2.lowid != ir1.lowid
  and ir2.modelicaobject_lowid = co2.lowid
  and co2.lowid != co1.lowid
  and co2.classifier_lowid = cl2.lowid
  and cl2.identifier = 'CurrentSensor'
  and pn1.modelicaelement_lowid =
eqFound.lowid
  and eqFound.class_lowid = clFound.lowid
  and clFound.model_lowid = mmFound.lowid
group by mmFound.name
```

The query returns the name of the found ModelicaModel objects, and counts the number of connect equations in the found model, that refers to components that are declared as classes with the name 'EMF' and 'CurrentSensor'.

7 Results and Experience

A first prototype version of ModelicaDB has been implemented that verified the approach. More work is required to cover more advanced features of the Modelica language.

Most of the implementation work was rather straitforward, once the UML models in Appendix A, and underlying detailed specifications [20] were completed. The exception was the currently 408 mapping rules that convert the parsed ModelicaXML elements into connected object structures according to the UML model in the ModelicaDB front end, so they can be stored in the database.

The Modelica grammar and ModelicaXML structures contain many details and requires several passes to resolve all references. This also involves searching the name spaces according to the static and dynamic lookup functions (Chapter 3 in [4]), and resolving identifier references to imported classes in libraries that are not in the current ModelicaXML file.

Other issues that require more work are:

- ModelicaXML-to-ModelicaDB mapping rules, which are currently initially generated from preprocessing of large representative ModelicaXML files, and then manually extended with actions that specify how priority sorted matching patterns of XML-elements are stored into objects in the ModelicaDB front-end. To get better verification of full grammar functionality coverage, the rules should be generated directly from the ModelicaXML DTD, or another formal Modelica grammar specification, but such an approach requires more research.
- How to represent modifications in ModelicaDB, so the users SQL-query pattern searches also hit modified classes, without the need for expensive processing of modification "deltas" in parse node trees.

Some other interesting research results that came out of this work are

- Identification of semantic equivalent functionality between the Modelica language and the industry standard ontology languages UML and Rosettanet technical Dictionary [13]. Thus it is definitely possible to reuse relevant ontologies originating from other modeling languages for exchanging existing product data with Modelica simulation model development tasks. Other more distant future applications can be inferred from [5].

New technical results are:

- Formalized Modelica simulation model interchange format in the form of a DTD, for Modelica 2.1. This DTD contains 88 language elements, and is described in [11] and [9]. The latest version has some small modifications and can be downloaded from the reference URL at [11].
- Extensions of the ModelicaXML tool for packaging directory structures containing Modelica source code libraries into one XML-file.
- UML-model of the Modelica database.
- Implementation of a relational database for searching, analysing, cross-referencing and checking of Modelica libraries [20].

8 Future work

Future work will be determined after members of the Modelica Design group and involved researchers at Linköping University have tested ModelicaDB prototypes and given recommendations for future work.

9 Conclusions

This paper reports work on ModelicaDB – a tool that provides database storage and query of Modelica models. We believe that given proper integration with engineering product development tools, ModelicaDB will be of great value on finding related product models, quick access through categorization, and assisting with a number of other related tasks.

A first prototype of the tool has been implemented. A full database schema has been designed and tested against queries, a Modelica library parser that converts libraries into XML form has been implemented. The main remaining task is completing the set of rules that map ModelicaXML elements to ModelicaDB objects.

Acknowledgements

Swedish Foundation for Strategic Research, ProViking project Systems Engineering and Computational Design (SECD).

References

[1] Dynasim. *Dymola*, http://www.dynasim.se/.

[2] INCOSE. *International Council on System Engineering*, http://www.incose.org.

[3] MathCore. *MathModelica*, http://www.mathcore.se/.

[4] *Modelica: A Unified Object-Oriented Language for Physical Systems Modeling, Language Specification version 2.1*, Modelica Association, 2004

[5] *Semantic Web Community Portal*, http://www.semanticweb.org/.

[6] OMG. *Unified Modeling Language*, http://www.omg.org/uml.

[7] Modelica Community, http://www.modelica.org/

[8] Moldelica Libraries (Ontologies), http://www.modelica.org/library/

[9] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*, Wiley-IEEE Press, 2003, http://www.mathcore.com/drmodelica.

[10] Peter Fritzson, Peter Aronsson, Peter Bunus, Vadim Engelson, Levon Saldamli, Henrik Johansson and Andreas Karstöm. *The Open Source Modelica Project*, in *Proceedings of The 2th International Modelica Conference*,March 18-19, 2002, Munich, Germany.

[11] Adrian Pop, Peter Fritzson. ModelicaXML: A Modelica XML representation with Applications, in International Modelica Conference,3-4 November, 2003, Linköping, Sweden, http://www.ida.liu.se/~adrpo/modelica/

[12] Adrian Pop, Ilie Savga, Uwe Assmann and Peter Fritzson. *Composition of XML dialects: A ModelicaXML case study*, in *Software Composition Workshop 2004, affiliated with ETAPS 2004*,3 April, 2004, Barcelona.

[13] Olof Johansson, Adrian Pop, Peter Fritzson, *A functionality Coverage Analysis of Industrially used Ontology Languages*, in *Model Driven Architecture: Foundations and Applications (MDAFA), 2004*, 10-11 June, 2004, Linköping, Sweden.

[14] Pim Borst, Hans Akkermans and Jan Top, *Engineering ontologies*, in Int. J. Human-Computer Studies, 1997, no. 46, p 365-406

[15] Adrian Pop, Olof Johansson and Peter Fritzson, *An integrated framework for model-driven product design and development using Modelica*, in Proceedings of the 45th Conference on Simulation and Modeling (SIMS), 23-24 September 2004, Copenhagen.

[16] Mogens Myrup Andreasen. *Machine Design Methods Based on a Systematic Approach (Syntesemetoder pa systemgrundlag)*, Lund Technical University, Lund, Sweden, 1980

[17] RosettaNet, http://www.rosettanet.org

[18] RosettaNet, *RosettaNet Technical Dictioanry*, http://www.rosettanet.org/technicaldictionary

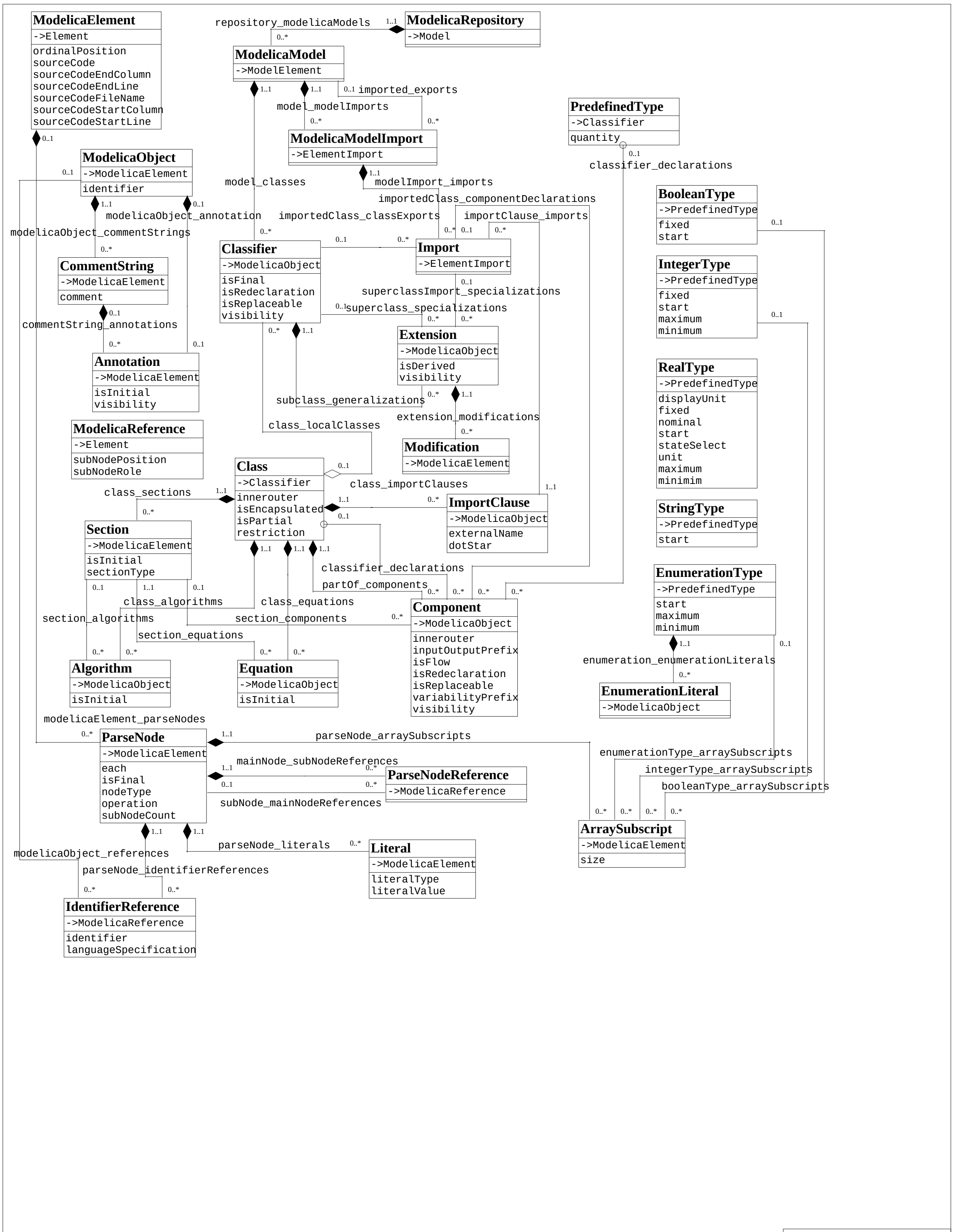
[19] Word Wide Web Consortium (W3C). Web Ontology Language (OWL), http://www.w3.org/TR/2003/CR-owl-features-20030818/

[20] Olof Johansson, ModelicaDB Project, http://www.modelica.org/projects/ModelicaDB/

[21] Michael Tiller, Parsing and Semantic Analysis of Modelica Code for Non-Simulation Applications, in *International Modelica Conference*,3-4 November, 2003, Linköping, Sweden.

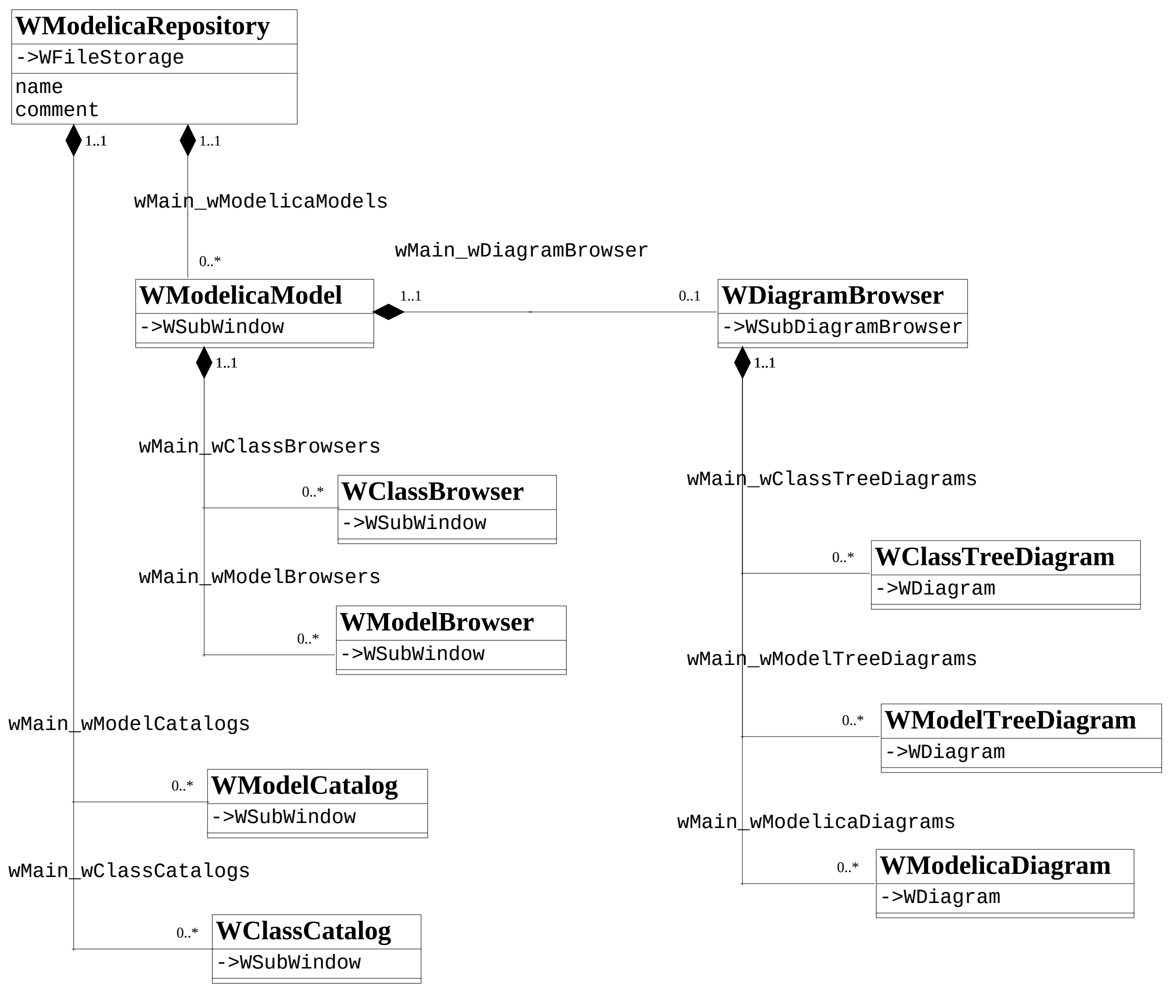
[22] IEC, IEC 61360, http://webstore.iec.ch for a fee

[23] ISO, ISO 31 Quantities and Units, Part 0-13. http://www.iso.org for a fee



Date Printed
2005-01-31 10:00:01

	Domain Model	Filename	Date Added	Date Modified	#Cls	#Rel	#Att
DM.	ModelicaDB-21-10d	ModelicaDB-21-10d.odm	2003-03-04 08:03:07	2005-01-31 09:40:04	67	64	162
	Modelica 2.1 database design for ModelicaXML information exchange. The goal is a repository that can be round-tripped with Modelica source code.						
Dgm	2.0) Modelica High-Level Meta Objects		2004-04-27 14:36:52	2005-01-31 09:40:04	30	37	-
	Complete UML Model of Modelica 2.1.						



Date Printed
2005-01-31 09:59:20

	Domain Model	Filename	Date Added	Date Modified	#Cls	#Rel	#Att
DM.	appmodelicadb-21-10b	appmodelicadb-21-10b.odm	2004-11-05 21:08:37	2005-01-27 09:26:36	18	11	12
	Prototype application window structure for ModelicaDB V2.1x, including Profile support.						
Dgm	2.0) AppModelica		2004-11-05 21:19:14	2004-11-12 07:45:58	10	9	-
	Application windows for the Modelica module.						

Session 6a

Automotive Simulation III

The PowerTrain Library: New Concepts and New Fields of Application

Christian Schweiger[1] Mike Dempsey[2] Martin Otter[1]

[1]German Aerospace Center (DLR), Oberpfaffenhofen, 82234 Weßling, Germany
[2]Claytex Services Limited, Warwickshire, CV35 8XB, United Kingdom

Abstract

Version 2.0 of the PowerTrain library will be released in March 2005. This article presents the new release, which is enriched by optional consideration of 3D effects, a simpler signal bus concept, new components and example models for flexible drivelines, 4wd drivelines and hybrid vehicles. In addition, various new driver models have been added.

1 Introduction

The PowerTrain library [6] is a licensed Modelica package providing components for modelling vehicle powertrains. It is also used for the modelling of gearboxes with speed and torque dependent losses. The available components range from simple, easy to use parts to very sophisticated components. All the components are open and can be extended and modified by the user.

In December 2002, version 1.0 was finished and since then several new developments have been incorporated into the library. In addition, new concepts in the Modelica language have been applied leading to improvements aimed at delivering better interoperability between the different automotive model libraries available. These are described in Section 3. In Section 4, new fields of application are described and new driver models are presented in Section 5.

2 Previous library status

In the past, the library provided standard and planetary gearboxes with speed and torque dependent losses, table-based engine and simple driver models, and components required to model the longitudinal dynamics of vehicles, as well as a range of detailed examples. Version 1.0 of the library contained 45 reusable components and 10 examples, cf. Figure 1. A number of components originally developed for the PowerTrain library have been incorporated into the Modelica standard library (version 1.5) since they are of general interest.

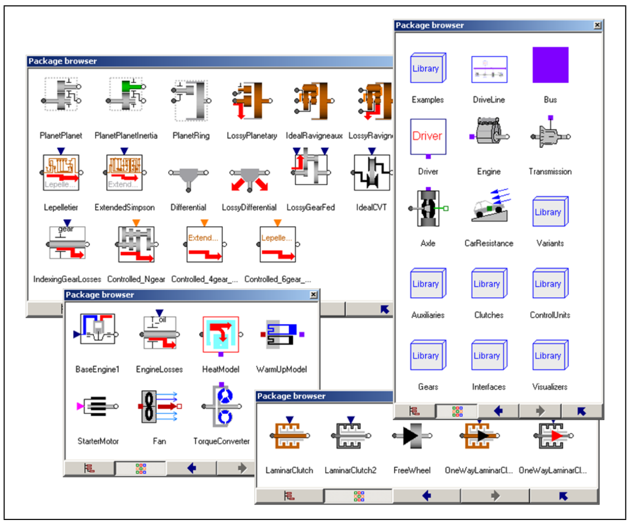
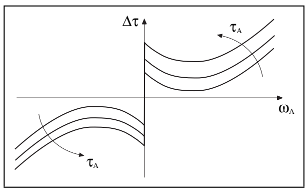


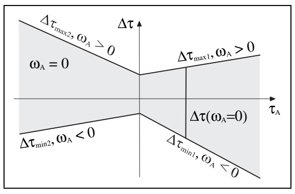
Figure 1: Components of the PowerTrain library and some sublibraries

A very important property of the PowerTrain library is the robust and efficient handling of *speed and torque dependent friction* [8] as illustrated in Figure 2, which occurs when considering gear mesh efficiency (due to gear teeth friction) and bearing friction. This novel type of friction handling was used in many components in version 1.0, especially for planetary gears, Ravigneaux gears, Lepelletier gears, extended Simpson gears and differential gears.

Large system models often become difficult to understand as there can be a large number of signals that need to be passed between the model's top-level components. To overcome this problem, the PowerTrain library used a signal bus as shown in Figure 3. The idea was that all the signals that have to be exchanged by the components are included on the bus. The com-



(a) Dependency on speed ω_A



(b) Dependency on torque τ_A

Figure 2: Speed and torque dependent friction $\Delta\tau$

ponents are then simply connected to the bus and do not have any signal connections to other components. Another common modelling problem is that models with varying levels of detail are required for different tasks. To reduce the number of different models that need to be saved a model architecture was implemented that would allow components contained in the top-level objects to be easily swapped for other compatible models with different levels of detail. The ability to swap the components was realized by making use of the replaceable model features of Modelica. As a consequence, only one model, cf. Figure 3, is necessary to model many different driveline configurations.

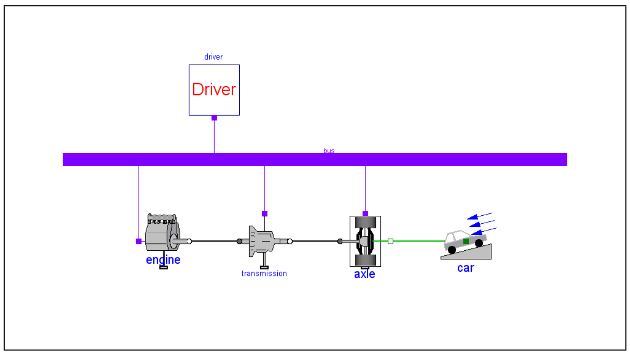


Figure 3: Model Driveline consists of a typical driveline from which all the main variants provided in the library can be selected, e.g., three different types of detailed automatic gearbox models but also user-defined gearboxes. The components need only a single connection to the *signal bus* in order to exchange signals among them.

The gearbox and shaft components can be animated, see Figure 4 for an example, which is useful for plausibility checking and demonstration purposes. Animation can be switched off by a parameter. In this case, the complete animation code is removed from a model in order to get efficient simulation code, e.g., for hardware-in-the-loop simulations.
A number of control systems as shown in Figure 5 were included in the library. These are used to control the engine and transmission models. The control

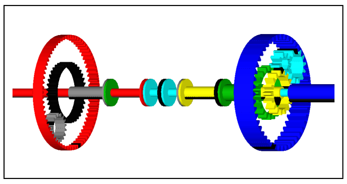


Figure 4: Animation of 6-speed automatic gearbox of Lepelletier type

systems for automatic transmissions are implemented in such a way that they support any number of gears. The driver interacts with these controllers by setting the gearbox mode to be used (P, R, N, D, 1, 2, 3, . . .).

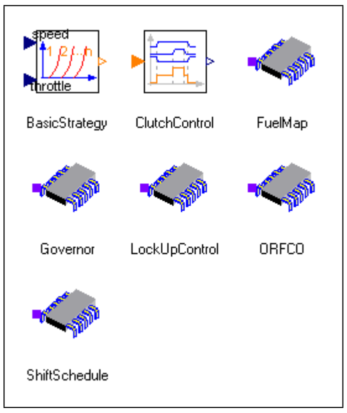


Figure 5: Control units included in the library

Several sophisticated example models are provided which serve as a starting point in developing user-specific models. Especially, examples are provided for power consumption calculation and analysis of shift strategies based on detailed models of 4- and 6-speed automatic gearboxes

3 New concepts and components

Several new concepts and components have been incorporated into version 2.0 of the PowerTrain library. They are described in the following sections.

3.1 Incorporation of 3D effects

In [9], a concept for reproducing the three-dimensional (3D) mechanical effects of one-dimensionally (1D) modelled powertrains has been presented. The idea is to model transmission elements with their mostly 1D rotating behaviour in a convenient way with 1D model components. Due to the simplicity of the 1D equations, this results in very efficient simulation code. When these 1D components are mounted on systems moving in 3D space using the Modelica.Mechanics.MultiBody [7] library a number of important effects, such as support torques and gyroscopic torques, are missing. By including adaptor models and a 3D inertia component it is possible to incorporate these missing effects.

These 3D effects are incorporated in version 2.0 of the PowerTrain library. By default, 3D effects are turned off to get fast simulations, which is especially important for real-time purposes [3, 10]. The 3D effects can be turned on through the use of a parameter. This has been implemented using the new Modelica feature *conditional declarations* that have been introduced in version 2.2 of the Modelica language specification. The idea is illustrated using the example in Listing 1 and the object diagram in Figure 6.

Listing 1: Example demonstrating conditional declarations

```
model DampedInertia
  import Modelica.Mechanics.Rotational;

  extends Rotational.Interfaces.TwoFlanges;

  parameter Boolean damping=true;

  Rotational.Inertia inertia;
  Rotational.Damper damper(d=10) if damping;
  Rotational.Fixed fixed if damping;
equation
  connect(inertia.flange_a, flange_a);
  connect(inertia.flange_b, flange_b);
  connect(damper.flange_a, fixed.flange_b);
  connect(damper.flange_b, flange_b);
end DampedInertia;
```

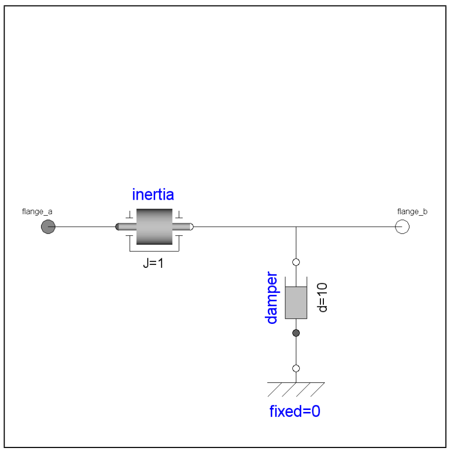


Figure 6: Object diagram of model *DampedInertia*

The declaration of the components *damper* and *fixed* is dependent on the Boolean parameter *damping*. These components are instantiated only if *damping* has the value true. Otherwise these components are not instantiated and the connect statements referring to them are ignored. The advantage in comparison to simply setting the damping coefficient *d* of *damper* to zero is that the equations of the disabled components are removed from the model and from the generated code, leading to more efficient simulations.

This feature is now used to incorporate 3D effects into the PowerTrain library. To include 3D effects into the components, a MultiBody connector is required but for the simple 1D case, it is desirable to remove these connectors. Therefore, the base class shown in Listing 2 was implemented which is inherited by the affected components of the PowerTrain library.

Listing 2: Base class for components with optional 3D effects

```
partial model ThreeD
  import Modelica.Mechanics.MultiBody;
  parameter Boolean enable3D=true;
  MultiBody.Interfaces.Frame_a
                    frame_a if effectiveEnable3D;
protected
  outer MultiBody.World world;
  parameter Boolean effectiveEnable3D=
                     world.enable3D and enable3D;
end ThreeD;
```

This base class allows the 3D effects to be switched on or off in two ways:

- The base class provides a Boolean parameter *enable3D*, which can be used to disable the 3D effects for a particular component.
- It is also possible to enable the 3D effects for all the components within a model by use of a single global setting. This is implemented using the inner-outer concept of Modelica, which is already used in the MultiBody library for providing global settings (e.g. for default animation and gravity field). These definitions are set in the inner component *world* which must be added to the top level of the model. All the components within a model access the global settings through a respective outer component *world*. The component MultiBody.World has been extended by adding a Boolean parameter *enable3D*.

Both parameters have to be true for the 3D behaviour to be modelled. If either of them is false then the 3D MultiBody *frame_a* as well as the additional equations for modelling the 3D effects are removed and only 1D behaviour is modelled. The described base class is

used throughout the PowerTrain library when 3D effects might be included. By default, only 1D behaviour is modelled and all the code for the 3D effects is removed during the code generation phase. By inheriting from this class, it is easy to enable 3D effects and then place the complete powertrain models onto 3D moving parts without neglecting any 3D effects.
Figure 7 shows a model which combines components of the PowerTrain and the VehicleDynamics library [2]. The 3D effects are modelled in the powertrain. First investigations, see Figure 8, of such a model have been performed in [5].

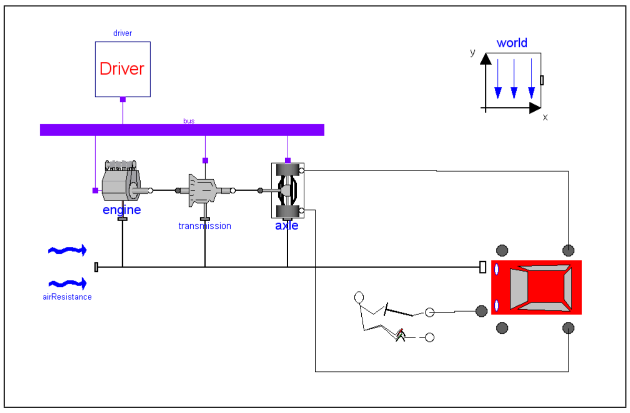


Figure 7: Powertrain model with 3D effects combined with vehicle dynamics model

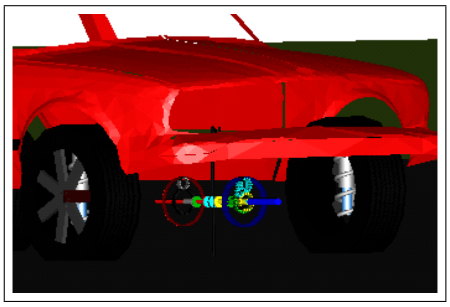


Figure 8: Animation of a joint powertrain and vehicle dynamics model

3.2 Expandable connectors

The signal bus concept introduced in version 1.0 of the PowerTrain library was not easy to extend to user-specific needs. The recommended method for changing the signal bus was to extend from the library and simultaneously replace the bus with a user-defined bus, ending up in a complicated structure.
For the *Vehicle Model Architecture* described in Section 3.4 an improved bus concept — *expandable connectors* — has been developed and included in version 2.2 of the Modelica language specification. In the simplest case, an expandable connector is merely an empty connector, see Listing 3. This connector class can be instantiated in different components, e.g, Source and Integrator in Listing 3, and it is possible to connect to components in the expandable connector, even though they are not defined in the class definition of the bus. The various connect statements are evaluated at compile time and the union of all referenced variables is used to build the actual bus connector. During translation a check is made to ensure that every signal read from the bus is defined exactly once.

Listing 3: Example package demonstrating bus realisation using an *expandable connector*

```
package BusTest
  import Modelica.Blocks;

  expandable connector Bus
  end Bus;

  model Source
    Bus bus;
    Blocks.Sources.Sine sine;
  equation
    connect(sine.y, bus.dq);
  end Source;

  model Integrator
    Bus bus;
    Blocks.Continuous.Integrator integrator;
  equation
    connect(integrator.u, bus.dq);
    connect(integrator.y, bus.q);
  end Integrator;

  model Example
    Bus bus;
    Source source;
    Integrator integrator;
  equation
    connect(bus, source.bus);
    connect(bus, integrator.bus);
  end Example;
end BusTest;
```

A simulation result of model *Example* in Listing 3 is shown in Figure 9. Although the bus connector was defined as empty, the two contained variables can be plotted.

The concept of expandable connectors leads to an enormous gain in flexibility. The bus definition in the PowerTrain library has been changed to that shown in Listing 3. This allows a user to extend the bus in a very convenient way: Just a connection must be drawn between a signal port and the bus, see Figure 10. In addition a variable name on the bus must be provided.

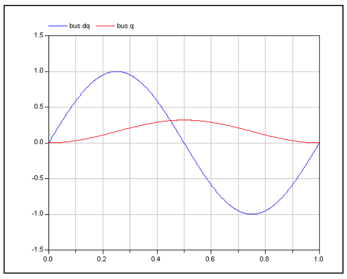


Figure 9: Simulation result of model *BusTest.Example*

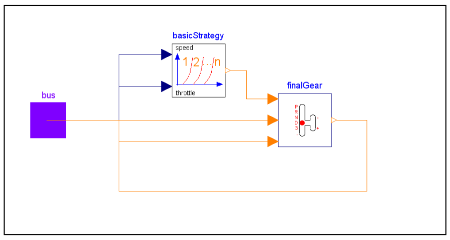


Figure 10: Example model *PowerTrain.ControlUnits.-ShiftSchedule*

3.3 New gears with losses

In the previous version of the PowerTrain library, the two components *Gears.PlanetPlanet* and *Gears.PlanetRing* were provided so that any type of planetary gearbox could be constructed. These elements have been improved so that speed and torque dependent losses are now taken into account. An example is shown in Figure 11 where a planetary gear of the Wolfrom type, with losses, is constructed using the PlanetPlanet and PlanetRing components.

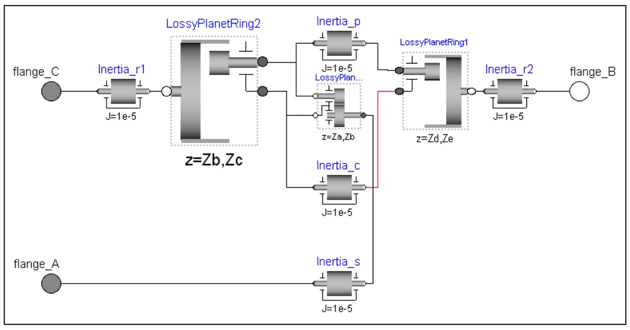


Figure 11: Object diagram of Wolfrom type planetary gearbox with losses implemented using the improved PlanetPlanet and PlanetRing components

The overall gear ratio and efficiency of a planetary gearbox constructed using these basic elements can be calculated using the model *PowerTrain.Examples.WolfromEfficiency* shown in Figure 12, provided that the number of teeth on each of the gearwheels and the efficiencies of each mesh (gear teeth contact) are known.

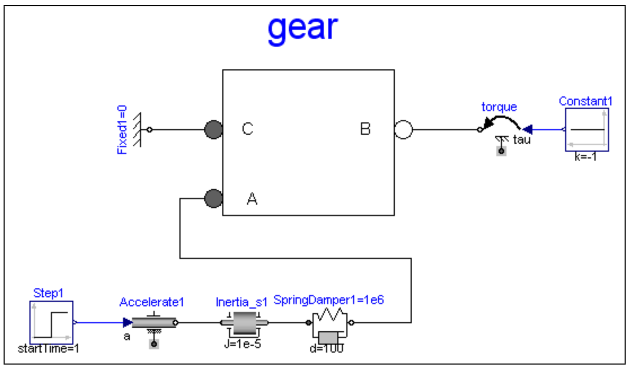


Figure 12: Object diagram of test model to determine gear ratio and gear efficiency between axis A and B of the Wolfrom planetary gearbox

The gear ratio and the gear efficiency between axis A and B are computed as follows:

- At the output (axis B) a unit torque is applied as a load.
- At the input (axis A) a unit acceleration is applied for 1 s. This means that the speed of axis A starts at zero, and rises linearly to 1 rad/s during the first 1 s of the simulation and then remains constant at 1 rad/s. Since the speed is constant, the inertias inside the gear do not have an effect for the power distribution.
- To avoid possible problems with the non-uniqueness of solutions of friction elements when forcing the wheel to rotate according to the desired acceleration, the forced movement of the flange is not directly required. Instead, the acceleration component drives a very stiff spring which in turn drives the gear flange.
- The gear ratio is the ratio of the angular velocities of flange_A and flange_B at the end of the simulation (say at 2 s).
- The gear efficiency is the ratio of the cut-torques of flange_A and flange_B divided by the gear ratio.
- The above two numbers can most easily be determined from the simulation, in Dymola, by clicking in the plot window on *Advanced* and then setting $t = 2$ in the input field *Time*. This displays the values of all variables in the variable browser at $t = 2$.

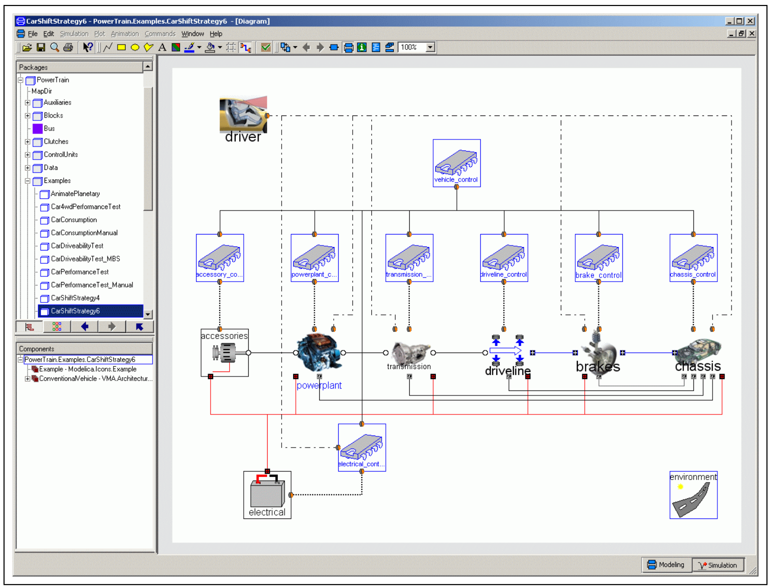


Figure 13: Screenshot of Dymola showing example model on Vehicle Model Architecture basis

It would not be possible to determine these values using a static model where the gear shafts are not rotating. This is because the friction between the teeth would be in the stuck mode and the friction torques are then **computed** implicitly from the requirement that the shaft accelerations are zero. This is correctly described by the Modelica model and therefore does not allow the efficiency to be calculated using a static model.

3.4 Vehicle Model Architecture

Within Ford Motor Company's Powertrain Research Department an architecture for modelling of vehicles has been developed and reimplemented in Modelica. The resulting Modelica package was presented in [11] and is freely redistributable in source code form. In order to promote interoperability with other libraries in the automotive area, this architecture will be used in the PowerTrain library. An example model following this architecture is shown in Figure 13.

4 New fields of application

Version 2.0 of the PowerTrain library has been extended by including a wider range of driver models and new components in a number of new application areas. Example models demonstrating the usage of the new components have also been included in the library and these are described below.

4.1 Flexible driveline models

An area of increasing interest is the modelling of vibrations and oscillatory responses within the whole powertrain. These effects are required when attempting to simulate driveability, shift quality or other similar effects that are likely to introduce oscillatory torques into the powertrain system. The study of these effects has required the development of additional driveline component models that include additional effects such as stiffness, damping and backlash.
The first key component required was the flexible shaft, which introduces the ability to model the twisting of a shaft, such as the propshaft or driveshafts. In it's simplest form the flexible shaft consists of two rotational inertias connected by a linear spring-damper. In this form the shaft can be used to model low frequency effects such as shuffle, which occurs in the 2..10 Hz range.
The flexible shaft can easily be adjusted to model higher frequency effects as it can contain a variable number of spring-dampers and inertia components. This is possible through the use of a parameter n to specify how many spring-damper blocks the flexible shaft model should contain. The effective stiffness and damping of each spring-damper block is adjusted based on the parameter n. The flexible shaft contains $n+1$ inertias and the total inertia of the shaft is evenly distributed across these. The implementation of the flexible shaft is shown in Listing 4.
When developing a model to simulate driveability or shift quality it is important to include the reaction of the powertrain within the vehicle. As the entire pow-

Listing 4: Flexible shaft implementation

```
model FlexibleShaft
  extends Modelica.Mechanics.Rotational.Interfaces.TwoFlanges;

  parameter PowerTrain.Types.TorsionalStiffness c=1 "Stiffness";
  parameter PowerTrain.Types.TorsionalDamping d=0 "Damping";
  parameter Modelica.SIunits.Inertia J=1 "Inertia";
  parameter Integer n(min=1) = 1 "Number of spring-dampers";

  Modelica.Mechanics.Rotational.Inertia inertia[n + 1](each J=J/(n + 1));
  Modelica.Mechanics.Rotational.SpringDamper springDamper[n](each c=c*n, each d=d);
equation
  connect(flange_a, inertia[1].flange_a);
  for i in 1:n loop
    connect(springDamper[i].flange_a, inertia[i].flange_b);
    connect(springDamper[i].flange_b, inertia[i + 1].flange_a);
  end for;
  connect(inertia[n + 1].flange_b, flange_b);
end FlexibleShaft;
```

ertrain is suspended within the vehicle by a number of mounts it can move within the vehicle and have a significant impact on the overall vehicle response. Previously the PowerTrain library did not consider these effects and a number of components have been developed specifically for this task.

Within the PowerTrain library an example has been included showing how to model the reaction of a differential on its mounts for a rear-wheel drive vehicle. In this example the driveline is modelled using the 1D flexible shaft for the propshaft and driveshafts, the differential is modelled using MultiBody components and is connected to the appropriate shafts via the Shaft1D_MBS, see Figure 14.

The Shaft1D_MBS model is used to couple 1D rotating components directly to components in the MultiBody library. This component relates the rotational speed and torque in the 1D connector to the speed and torque on the specified axis in the MultiBody connector.

The differential component is modelled using the MultiBody library and reacts the torques in the driveline onto the differential mount points. The actual differential mounts form part of the chassis subsystem and are discussed below. The differential includes the rotating inertias of the various internal components, backlash referred to the diff input and the mass, inertia and geometry of the complete differential assembly.

The type of mounts typically used to suspend the powertrain within the vehicle are designed to react forces in the x, y and z directions and they leave the powertrain free to rotate. These have been modelled using a series of three ActuatedPrismatic joints that are used to react the forces applied to the mount in three directions. A spherical joint is used at the side of the mount

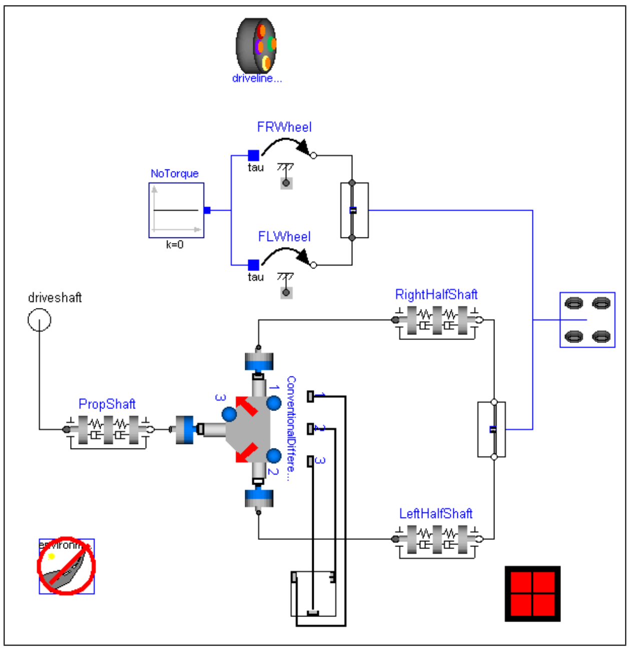


Figure 14: Driveline model that includes the movement of the differential on its mounts

that should be connected to the driveline component being suspended.

A number of tyre slip models have been included in the new version of the PowerTrain library. The available slip models include a simple linear slip model, a Pacejka slip model and the Rill slip model. These have been implemented for longitudinal slip only and consider the vertical load acting on the tyre.

4.2 4wd drivelines

A growing number of vehicles are being developed with all-wheel drive, e.g. sports utility vehicles (SUV) and commercial vehicles. There are a wide-range of

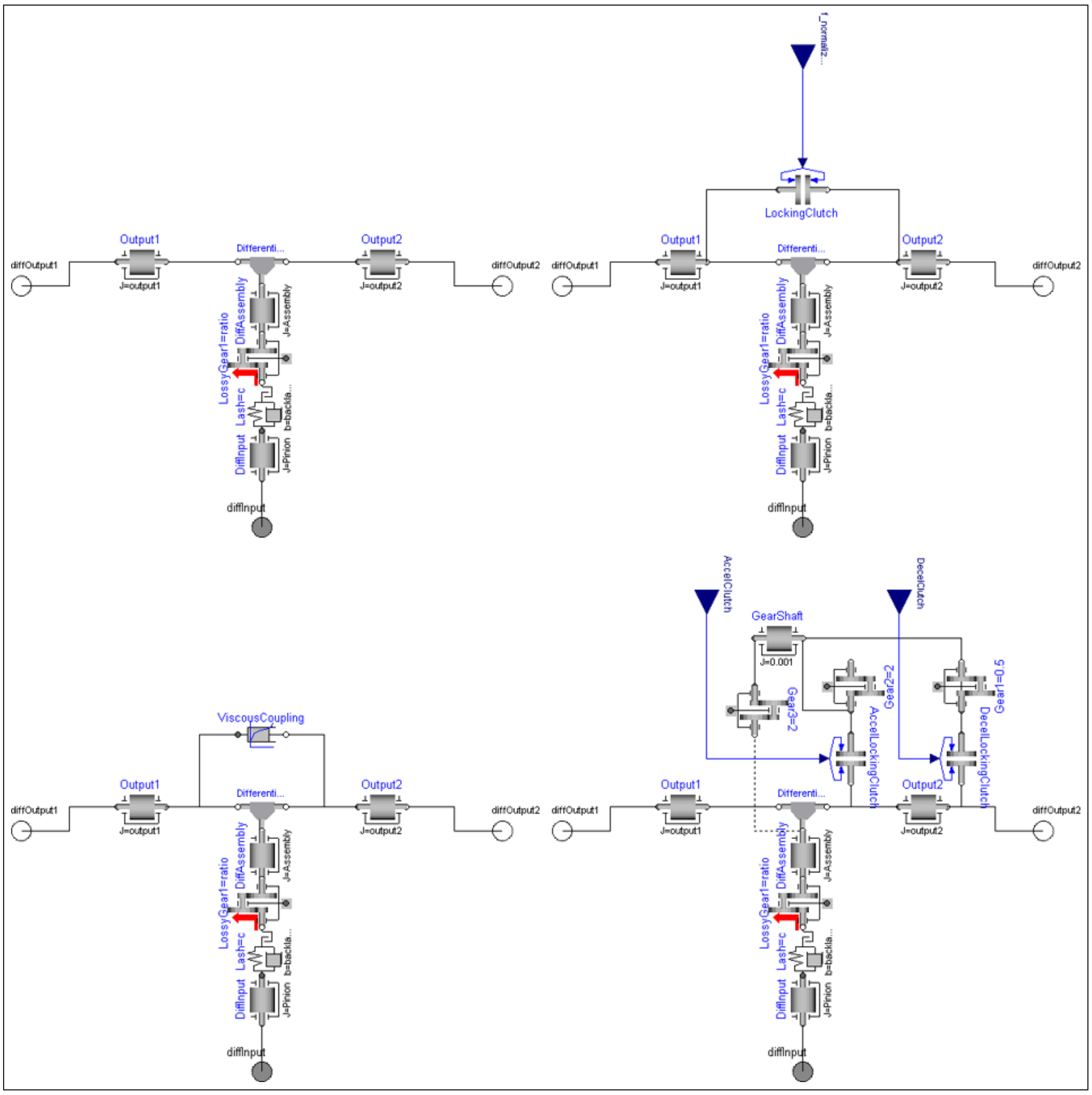


Figure 15: Some of the differential models available in the PowerTrain library. Clockwise from the top-left, conventional differential, simple active differential, torque vectoring differential, viscous differential

possible ways to deliver an all-wheel drive vehicle and components have been included to enable the modelling of the most common types and some of the most advanced. Available models include simple open differentials, viscous differentials, and two types of active differential. The various differential models have all been implemented as 1D rotational systems with only the conventional differential described previously using a MultiBody approach.

All the differential models provided are based around the use of an epicyclic differential unit. The different configurations of active and passive locking mechanisms are then placed around this core epicyclic unit and work in different ways to control the behaviour of the differential. Figure 15 shows four of the differential models available.

For the simple active differential and the torque vectoring differential control systems have also been provided. An example of a four-wheel drive vehicle, that uses three of these simple active differentials, has been added to the library. In this case, the control system has been designed to control each differential separately with the sole objective being to maximise traction. Each differential controller looks at the output shaft speeds from its differential and acts to reduce the difference in speed. It should be possible for some slip to occur between the shafts to allow for cornering and this can be defined through the controller parameters.

In addition to the range of differential models, a power take-off (PTO) style transfer box has been provided. In some four wheel drive applications this type of transfer box is used instead of a centre differential. The key difference between using a differential and a PTO style transfer box is that the ratio between the input and each output shaft is fixed in the PTO style box whereas this ratio can vary when a differential is used.

4.3 Hybrid vehicles

With several automotive manufacturers and suppliers working on hybrid vehicles, there was a necessity to provide corresponding models in order to support concept studies in this area. Models have been included for batteries, motors and the associated controllers to meet this need.

The objective is to deliver models suitable for concept study work so that minimal data is required to develop a working model of a hybrid concept and to test out the functionality. Two hybrid vehicle examples have been included, one based on a vehicle using an Integrated Starter-Generator (ISG) and another based on a series-parallel hybrid style vehicle similar to the *Toyota Prius*. Both examples have been configured to run drive cycle simulations.

The battery model included in the PowerTrain library is based on the Saft capacitance model, which was originally developed in P-Spice [4] and has also been used in the Advisor [1] simulation tool. Figure 16 shows the circuit diagram used for the battery model. Capacitor *Cb* is very large and represents the ability of the battery to store charge chemically, the capacitor *Cc* is small and represents the surface effects of a spiral-wound cell. The three resistances represent the terminal resistance (*Rt*), end resistance (*Re*) and capacitor resistance (*Rc*).

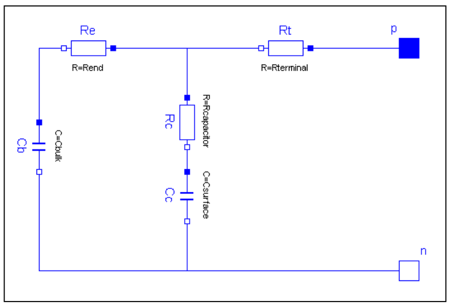


Figure 16: Circuit diagram of the battery model

The power electronics required to transfer energy from the battery to the electric motor have been simplified so that the simulation performance is maintained. An accurate model of the power electronics would require a large number of high frequency effects to be modelled and this would limit the suitability of the library for concept studies.

5 Other enhancements

This section describes the driver models, which have been added to the library, and changes concerning tables.

5.1 New driver models

The range of driver models provided with the PowerTrain library has been expanded to cover a wider range of tests. In addition to the existing cycle driver there are now driver models designed to carry out performance tests and driveability tests. There are also variants for use with both manual and automatic gearboxes.

The cycle driver models are based around the use of a PI controller that actuates either the brake or accelerator pedal to control the vehicle speed so that it follows a defined speed-time profile. A number of drive cycles are included by default such as the NEDC, EPA City and Highway cycles. It is possible to define your own additional drive cycles for use with the driver model. By varying the PI gains, the behaviour of the driver can be altered allowing the driver model to be tuned to match a range of different driving styles. The version of the cycle driver used with manual gearboxes also controls the clutch pedal and gear lever. The shift points are usually defined in the drive cycle to occur at particular points in time and driver starts to change gear at these points.

The driveability driver models are used to perform tip-in and tip-out tests in fixed gears, or fixed gearbox mode in the case of automatic transmissions. The tests start with the driver controlling the vehicle speed to an initial value and then accelerating and decelerating the vehicle between defined speeds using only the throttle. The brakes will not be used to decelerate the vehicle. For manual gearbox vehicles it is normal to define the tip-in and tip-out speeds as engine speeds. Due to the effect of the torque converter, it is more usual to define the tip-in and tip-out speeds using vehicle speed for automatic gearbox equipped vehicles.

The performance driver is used to perform standing start acceleration tests. The version used with automatic gearboxes can perform both an idle start or stall start acceleration test. In both versions the accelerator pedal position for the acceleration test can be defined so it is possible to assess the part-throttle acceleration performance as well as the wide open throttle (WOT) performance. The version used with manual gearboxes will change gear when a defined engine speed is reached. If in-gear acceleration times are required the driveability driver model should be used and the tip-in and tip-out speeds set to be the minimum and maximum speeds for the given gear.

5.2 Replaceable tables

All tables in the library have now been declared as replaceable. This change was made as many customers do not often have data in a form that is compatible with the tables in the Modelica standard library. Instead, they are forced to use their own, proprietary data format and their own table implementations. It was difficult for them to use these in combination with the PowerTrain library in the past.

6 Conclusions and Outlook

Version 2.0 of the PowerTrain library offers several new features, which open many new applications. New Modelica language elements allow a clean implementation of the new features and make it easier for users to adapt the library to their own specific needs.

Acknowledgements

For fruitful discussions, the authors would like to thank Jochen Köhler and his colleagues from ZF Friedrichshafen AG.
In addition, thanks to Michael Tiller from Ford Motor Company for pushing forward the concept of expandable connectors.
This work was in parts supported by *Bayerisches Staatsministerium für Wirtschaft, Infrastruktur, Verkehr und Technologie* under contract AZ300-3245.2-3/01 for the project *Test und Optimierung elektronischer Fahrzeug-Steuergeräte mit Hardware-in-the-Loop-Simulation* in the years 2001–2003.

References

[1] *ADVISOR*. URI `http://www.ctts.nrel.gov/analysis/advisor.html`.

[2] J. Andreasson, *VehicleDynamics library*, in Proceedings of the 3rd International Modelica Conference, Linköping, Sweden, November 2003, Modelica Association and Linköping University, pp. 11–18.

[3] H. Elmqvist, S. E. Mattsson, H. Olsson, J. Andreasson, M. Otter, C. Schweiger, and D. Brück, *Realtime Simulation of Detailed Vehicle and Powertrain Dynamics*, in Electronics Simulation and Optimization (SAE 2004 World Congress), Detroit, USA, March 8–11, 2004, SAE International. Document Number: 2004-01-0768.

[4] V. H. Johnson, A. A. Pesaran, and T. Sack, *Temperature-Dependent Battery Models for High-Power Lithium-Ion Batteries*, in 17th Electric Vehicle Symposium, Montreal, Canada, October 16–18, 2000.

[5] D. Mauermann, *Echtzeitsimulation detaillierter Fahr- und Antriebsstrangdynamik*, diploma thesis, Hochschule für Technik, Wirtschaft und Kultur Leipzig (FH), Fachbereich Elektrotechnik und Informationstechnik, July 2004. Compiled at Deutsches Zentrum für Luft- und Raumfahrt e. V.

[6] M. Otter, M. Dempsey, and C. Schlegel, *Package PowerTrain. A Modelica Library for Modeling and Simulation of Vehicle Power Trains*, in Proceedings of the 1st Modelica Workshop, Lund, Sweden, October 2000, Modelica Association, pp. 23–32.

[7] M. Otter, H. Elmqvist, and S. E. Mattsson, *The New Modelica MultiBody Library*, in Proceedings of the 3rd International Modelica Conference, Linköping, Sweden, November 2003, Modelica Association and Linköping University, pp. 311–330.

[8] C. Pelchen, C. Schweiger, and M. Otter, *Modeling and Simulating the Efficiency of Gearboxes and of Planetary Gearboxes*, in Proceedings of the 2nd International Modelica Conference, Oberpfaffenhofen, Germany, March 2002, Modelica Association and Institute of Robotics and Mechatronics, Deutsches Zentrum für Luft- und Raumfahrt e. V., pp. 257–266.

[9] C. Schweiger and M. Otter, *Modelling 3D Mechanical Effects of 1D Powertrains*, in Proceedings of the 3rd International Modelica Conference, Linköping, Sweden, November 2003, Modelica Association and Linköping University, pp. 149–158.

[10] C. Schweiger, M. Otter, and G. Cimander, *Objektorientierte Modellierung mit Modelica zur Echtzeitsimulation und Optimierung von Antriebssträngen*, in Steuerung und Regelung von Fahrzeugen und Motoren – AUTOREG 2004, VDI/VDE-GMA, ed., no. 1828 in VDI-Berichte, Düsseldorf, Germany, März 2004, VDI-Verlag, pp. 639–650.

[11] M. Tiller, P. Bowles, and M. Dempsey, *Development of a Vehicle Model Architecture in Modelica*, in Proceedings of the 3rd International Modelica Conference, Linköping, Sweden, November 2003, Modelica Association and Linköping University, pp. 75–86.

Development of a Modelica Heavy Vehicle Modeling Library

Per Bengtsson Henrik Jansson Niklas Pettersson Tony Sandberg
Scania CV AB
151 32 Södertälje, SWEDEN

Abstract

Physical modeling for simulation of fuel consumption and other dynamic behavior in heavy vehicles can be useful in many areas from concept design to sales support. Similar models of vehicle subsystems are needed in many applications, it would thus be beneficial to have access to a library of reusable vehicle subsystem and component models to avoid repeated implementation. A solution based on a model architecture and a supporting Modelica library for structured storage of models and components is proposed. The work has been focused on promoting modeling practices enabling reuse, but we have also tried to maintain as much freedom as possible for the modeler.

1 Introduction

There exist a number of different proposals for vehicle modeling architectures in Modelica (for example [5] and [2]. The aim of this project has been to create a complete system with both a hierarchical model structure defining the interfaces between subsystems on several levels, and a model library. The library is used to store sub-system interfaces along with available implementations and required supporting components such as connector definitions. The system is intended to be used for various research and development efforts within Scania CV AB. Since development projects may have very different aims, and be focused on different subsystems, it is unlikely that the library will provide a final model for the task. Hopefully, existing versions of most sub-systems can be used together with new models specific to the current problem. A key consideration in the work has been to build a system which is suitable for use by both experienced and novice modelers. The project is rather applied in its nature, and the article is intended to describe our experiences.

2 Architecture Concerns

2.1 Multi-domain Library

One much hailed property of the Modelica language is its multi-domain modeling capability. Components from model libraries describing different domains can be used together in the same model. However, the majority of available libraries are focused on one domain. In most cases this is a natural partitioning. In this project the common denominator has not been the engineering domain, but rather the system to be described. The purpose of the library is to store component models, defined through the partitioning of the described system into physical sub-systems.

A design goal has been to keep all available models in one central location, easily accessible to everyone. Existing models use an in-house media library to represent air- and coolant flows. This domain specific library is thus also needed by users of the new model library. It was decided to place it within the new library. This issue is further discussed in section 4.2.3

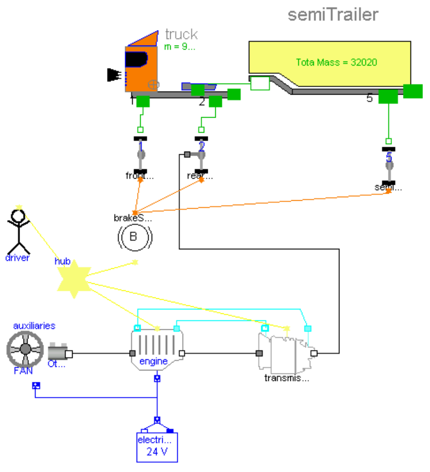


Figure 1: Heavy truck model in Dymola$^{\text{TM}}$.

2.2 Heavy Vehicles

Most existing vehicle model libraries are designed primarily for cars. Heavy vehicles have a number of subsystems which are not present in passenger cars. Particularly the engine/transmission system includes devices like an exhaust brake and possibly a retarder. Further, the cooling system also has a more prominent role than in cars, and coolant is often used both by the engine and the transmission.

2.3 Model Variants

Since this library is designed to be used in many different projects, there is a need to handle different variants of component models. Supporting different model variants, while attempting to preserve compatibility and avoid hidden interdependences has been one of the greatest challenges in this work. In some cases it will be necessary to create an entirely new model to be used instead of one already in the library, but substantial changes in behavior can be achieved without going that far. The library supports three ways of changing model behavior, listed in the order they should be considered.

Some models depend on data found in external parameter files or lookup tables. Theses can easily be changed at run-time without any need to modify or recompile the actual model. When models are added, this approach should be considered for any data that needs to be changed frequently. Recompilation between simulation runs is not only time consuming, it also assumes the presence of a licensed installation of the compiler.

Numerical parameters which are not set through data files can still be influenced at run-time. The simulation reads an initial state file, where values different from the default ones can be specified for real, integer and Boolean model parameters. This solution requires less complex source code than the data file approach, and is advantageous when only a few parameters need to be accessible. This approach would mainly be useful in creating an end-user application where the user should for example be allowed to choose between different tire models in an external GUI.

When a new model structure is needed, and even redeclaration of submodels is not enough, an entirely new model should be created. To make the new model usable in other projects, the existing base classes should be used to define the interfaces. If it is necessary, additional base classes can be created to supply extra connectors. The new model should of course be documented and made as flexible as possible with parameters and replaceable components used appropriately.

We have designed the library with fundamental base classes as blueprints for the physical subsystems and their major components. Only the interfaces required for simple implementations of the models are included. Additional base classes can be used to add more connectors if required by more advanced models. We have opted not to define a completely fixed architecture where all connections are always identical, but rather a supporting framework for developers intending to create reusable models. See also section 3.2.

2.4 Signaling Bus

A key issue in an architecture which contains both physical plant and controller models is the handling of electrical signals. The controllers need to exchange data among themselves and they need to exchange signals with sensors and actuators. For our applications the actual signaling behavior is not that important, an ideal communications model is sufficient. For the communication between a plant and its controller, standard library inports and outports are used. The communication between the controllers was a tougher case. Two implementations of the same controller may not have the same signaling needs, thus it must be possible to change the set of signals sent between control units.

Separate input and output ports for all links between control units in the vehicle would create an undecipherable graphical mess. Some type of signaling bus is needed. Both the standard library bus connectors and the type of bus used in the vehicle modeling architecture proposal by Tiller et. al [5] were evaluated. We did not find enough information about the inter-controller communication in the Tiller paper to implement that system. Our main problem was to find a way of having compatible connectors in all controllers, without modifying the code of every controller when a signal was added to the bus. The Modelica standard library bus does not solve that problem, since it requires all signals to be declared in the connector. Eventually we chose a simpler solution based on a common connector called ”CAN” with a replaceable variable, called ”protocol”, which contains all the signals. The protocol variable can easily be redeclared into a type which contains exactly the signals broadcast on the bus in a particular model. Different implementations of the CAN connector are used for different signal buses in the vehicle.

Most of our control units are implemented through external function calls, thus the drawback of having no convenient graphical way of converting a signal from inport/outport to bus format is minor. See listing 4 for an example of how the electronic control unit models use the CAN bus. The connected control unit model is shown in figure 2.

Listing 1: The CAN connector base class.

```
partial connector CANBase
  "Basic connector for modelling CAN
  comunincation"
  replaceable Protocols.Interfaces
    .ProtocolBase protocol
    "Protocol to be used";
end CANBase;
```

Listing 2: Implementation of the general CAN bus connector.

```
connector CAN_s
  "General control system
    communication bus connector"
  extends CANBase;
  annotation (...);
end CAN_s;
```

Listing 3: A part of the definition of a CAN protocol.

```
record ProtocolStd
  extends Interfaces.ProtocolBase;
  Real EngineSpeed
    "Speed of engine in rpm";
  Real EngineTemp
    "Engine temperature in deg C";
  ...
end ProtocolStd;
```

Listing 4: Sample usage of the CAN bus in the engine control unit.

```
...
/* I/O mapping (sensors/actuators)*/
engineSpeed = inport[ENGINE_SPEED];
outport[FUELING] = fueling;
outport[EXHAUST_BRAKE_ON] =
  CAN.protocol.ReqExhaustBrake;
...
/* Write CAN values */
CAN.protocol.EngineSpeed = engineSpeed;
CAN.protocol.EngineTemp = engineTemp;
CAN.protocol.ActualEngineTorque =
  inport[ACTUAL_ENGINE_TORQUE];
CAN.protocol.ActualExhaustBrakeTorque =
  inport[EXHAUST_BRAKE_TORQUE];
...
```

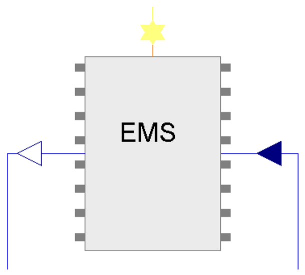
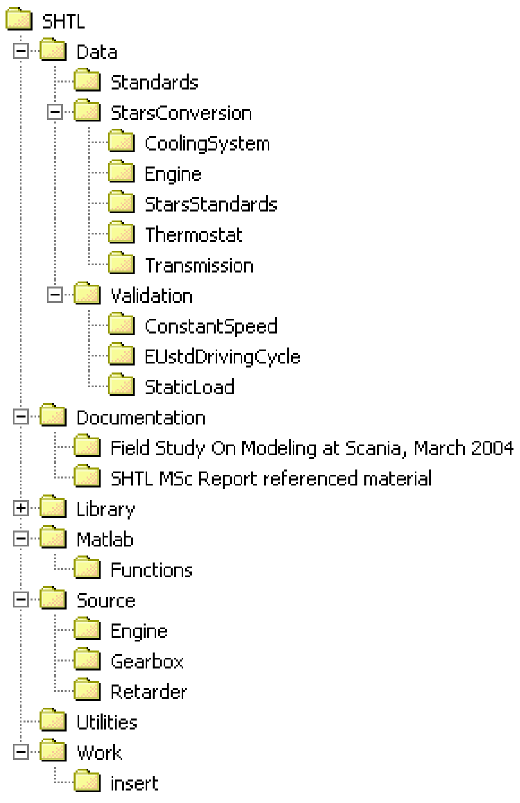


Figure 2: Engine management system electronic control unit.



```
SHTL
├─ Data
│  ├─ Standards
│  ├─ StarsConversion
│  │  ├─ CoolingSystem
│  │  ├─ Engine
│  │  ├─ StarsStandards
│  │  ├─ Thermostat
│  │  └─ Transmission
│  └─ Validation
│     ├─ ConstantSpeed
│     ├─ EUstdDrivingCycle
│     └─ StaticLoad
├─ Documentation
│  ├─ Field Study On Modeling at Scania, March 2004
│  └─ SHTL MSc Report referenced material
├─ Library
├─ Matlab
│  └─ Functions
├─ Source
│  ├─ Engine
│  ├─ Gearbox
│  └─ Retarder
├─ Utilities
└─ Work
   └─ insert
```

Figure 3: Directory structure for non-model files.

3 Usability Concerns

3.1 Non-model Files

The model library itself is relatively easy to distribute to the users. It is sufficient to copy the directory structure containing all the models to an appropriate place in the file system and load the library into Dymola. To get a nice working environment, where all external files are found by the system and the library is included in the Dymola GUI, takes a little more effort. In addition to the library directory tree our completed system consists of a directory structure on a higher level in the file system. This tree contains the model library itself and all external function source code and model data files for the. A work directory for the user is also provided. Data file paths and external source code links (through include annotations in functions) in the model library are given with relative paths, making sure that the files are found if the work directory is used.

If the user starts Dymola with the script provided with the library, an included configuration file is used to make sure that the heavy vehicle library automatically is included in the model browser.

3.2 Top Level Model

It is anticipated that a significant part of the work done with the library will be carried out in project form by people with little or no previous Modelica and/or modeling experience. As a result overly complicated and abstract language constructs have been avoided. The suggested method of putting together a vehicle for a particular task is to select those sub-system models which are most suited and add them to a new model. A ”master” model with most components declared as replaceable would enable new versions to be created with fewer lines of code, but the added abstraction has been deemed to be more difficult to handle than the extra coding. Future enhanced modeling tools may reverse this decision, but today we think that the ability for the novice modeler to fully understand his or her source code is warrants some code duplication.

The master model approach is best suited when it is anticipated that all implementations of a sub-system will be absolutely compatible. While this is a nice assumption we don’t think that it will be valid in our case. The range of intended applications for the vehicle library is so broad that some modifications also to the structure and interfaces certain components will be necessary in many projects. Real-time simulation of components as parts in simulink models is one example where non-standard shortcuts have been very efficient. Strict adherence to the interfaces is certainly optimal from a reusability perspective, but we have not yet found a set of interfaces which have been practical to use in every application. Further work may bring us ever closer to that goal.

3.3 Concurrent development

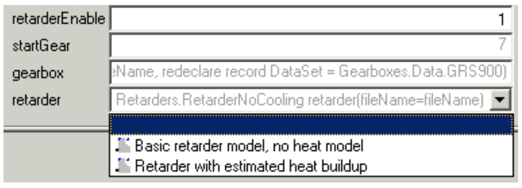
Traditionally one of the main obstacles to reuse of Modelica models created in previous projects has been that new functionality has been spread through many subsystems, rather than contained in one. When developers in two projects have enhanced different subsystems they also have modified many others in the process, making it difficult to incorporate enhanced components from different projects in the same model. Hopefully, the new library will promote solutions where components to a higher degree are created as self contained units.

A version control repository is used to make sure that two different groups do not accidentally make simultaneous conflicting changes. If only a particular sub-tree is checked out with write privileges the developer is encouraged to find solutions within that subsystem. The version control repository also supports named versions of the library to be created, which is useful to make sure that the exact model versions used in a project or application will always be available. The model library code is managed and provided to users in the same way as other source code in the organization. Users across various departments and groups can thus use a code management system which they are already accustomed to.

3.4 Choice Annotations

A number of Modelica entities can have choice annotations, which allow the model user to select appropriate parameter values easily in a modeling GUI. This feature has been used in many places where the parameter is not a physical quantity. For examples file name parameters used to specify data files are declared to be of a certain filetype. Each filetype has an associated list of suggested file names. In figure 4 the dropdown box for retarder model selection is shown.

The signal bus protocol used throughout the vehicle is determined by the type of a replaceable variable. The final setting of the type is propagated to the vehicle level, making it easy to change the type of every bus connector. Available protocols are presented in a



retarderEnable	1
startGear	7
gearbox	eName, redeclare record DataSet = Gearboxes.Data.GRS900)
retarder	Retarders.RetarderNoCooling retarder(fileName=fileName)

Basic retarder model, no heat model
Retarder with estimated heat buildup

Figure 4: The result of a choice annotation to aid in the selection of a retarder model

list, but it is still up to the user to choose one which contains exactly the set of signals that is being broadcast by the currently used set of control units.

4 Model and Library Structure

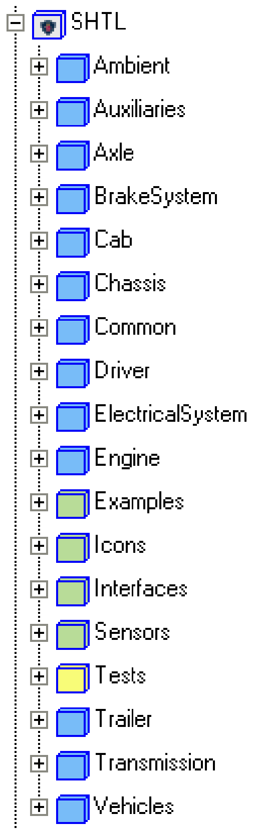


Figure 5: Top level packages in the vehicle library

4.1 Hierarchical levels

The created vehicle model structure defines multiple levels. The vehicle is built from physical components which also have a defined substructure. The supporting library mimics that same substructure. For example control units for subsystems are included in the subsystems modules themselves, to reduce the required number of components and connections in the top level model. In most cases one controller model and one or more interconnected plant models make up a subsystem. The top level subsystems cover the same areas as corresponding groups in the research and development organization. Local development of models by experts in various fields is thus simplified.

4.2 Library Structure

4.2.1 Color Coding

The vehicle library has been created with the same basic structure and package naming conventions as the Modelica standard library. Additionally the various package types have been color coded to make navigation in the package tree easier. Packages containing interfaces, sensors, icons and examples are made green. Data records are kept in red subpackages, and test models in yellow. Ordinary packages are a slightly darker shade of blue than the standard package icon.

4.2.2 Packages

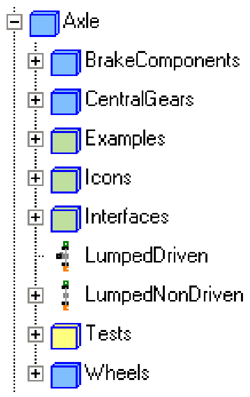


Figure 6: The ”axle” package with subpackages in the library browser.

Each physical subsystem has its own top level package for all its components, interfaces, etc. Additionally there are packages for examples, interfaces, icons, examples and tests on the top level. For an example of a top-level package see figure 6. Complete vehicle models, which can be used as components together with environment models from the ambient package, have a category of their own. To avoid a very deep tree structure these various types have been put at the same level in the hierarchy.

4.2.3 The Common Sub-tree

There are a number of models required for the modeling of a complete vehicle which do not clearly belong in any particular subsystem. Base classes for electronic control units, the signal bus connectors and the media library used for coolant and air modeling are a few examples. These functions are kept under a sub-tree called ”Common”. While it may seem more natural to create a separate library at least for the media components we prioritized keeping the entire model library self contained. The only external dependencies are the Modelica Standard Library and accompanying ModelicaAdditions library.

4.3 Physical Subsystems

The physical subsystems with the interfaces described are used in the current models. As it is impossible to foresee exactly what applications the vehicle library will be used for in the future, unused connectors are not included on speculation. More connectors are likely to be added in the future. To preserve current base classes this can be done through additional base classes as described in section 2.3.

4.3.1 Ambient

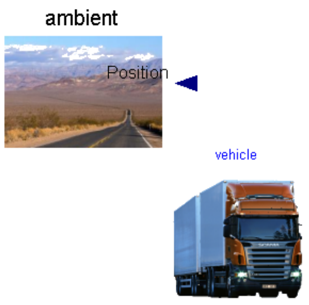


Figure 7: A vehicle model with the ambient component.

The ambient category is used to represents the environment around the vehicle. Models of this type supply data about surrounding temperature, air pressure and other environment constants. These models are also used to keep track of the road parameters such as slope and speed limit. Through the Modelica inner/outer construct one ambient component is accessible to all the other components in a model, which makes this the ideal place for any data that needs to be globally shared. Each model should have exactly one ambient component. The ambient interface has one input connector which is used to communicate the position of the vehicle. A vehicle and ambient combination can be seen in figure 7.

4.3.2 Auxiliaries

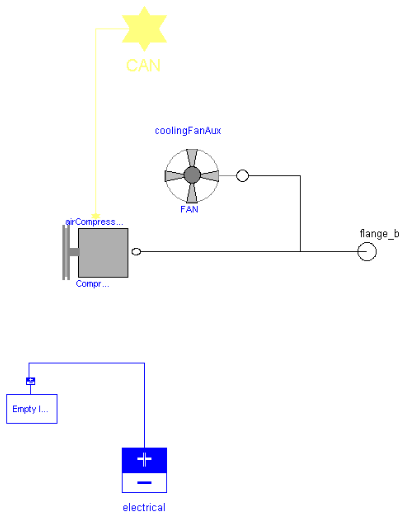


Figure 8: A model representing the auxiliary units.

Auxiliaries are components like the cooling fan, AC compressor, electric generator etc. These are generally mounted somewhere on the front side of the engine, and traditionally obtain their operating power through a mechanical link. It is recommended that any generator model is placed among the auxiliaries, and not in the electrical system. Separate models for each auxiliary unit are placed within this container. Subtrees in the library contain models related to the various units. The auxiliaries are connected to the electrical system and the engine, an implementation can be seen in figure 8.

4.3.3 Axle

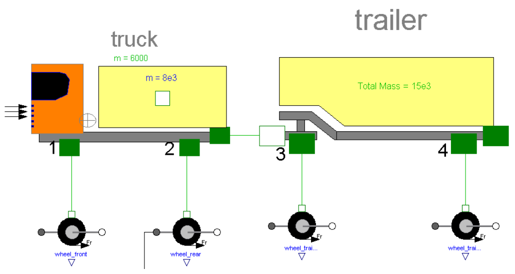


Figure 9: Truck model with full trailer and four axles.

The axle models contain tires, brake actuators and a final gear for driven axles. They are always connected to the chassis and the brake CAN bus. The axles generate a retardation force due to the rolling resistance in the wheels, and possibly due to the wheel brakes. Driven axles have an additional connector which allows the power train to transfer torque to the axle, and propel the vehicle. The number of axles in a vehicle configuration varies; it has to match the number of axle connectors on the chassis and all trailer models used. The vehicle shown in figure 9 thus requires four axles. An air interface could be added to simulate air-powered brake actuators.

4.3.4 Brake System

The brake system is a container for the brake management system and any related plants except for the brake actuator, which are represented at their physical location in the axle modules. The brake system is attached to the vehicle signaling bus and the brake signaling bus. Additional connections are likely in future more developed brake system models.

4.3.5 Chassis

The chassis models represent the frame of the truck. Cargo, axles and trailers attach to the chassis. The base class has the connectors for wheel axles and a draw bar. Derived classes add either a fifth-wheel (where the semi-trailer is attached) for tractor configurations, or a cargo attachment point for rigid trucks.

4.3.6 Driver

The driver model is responsible for overall control of the vehicle. Decisions to accelerate or decelerate depending on the surroundings are made by the driver. Depending on the vehicle, different control signals may be required. A manual transmission requires the driver to select an appropriate gear, while the GMS handles that duty for an automatic or automated manual transmission. In cruise control mode, the EMS controls the fueling, in driver demand mode the throttle is controlled directly by the driver. The driver logic depends on the control units used in other parts of the vehicle. The driver interface is very simple with only one connector, which is used to attach it to the CAN bus.

To control a vehicle from an external model (e.g. in Simulink) input and output ports are needed. In such a case input and output ports could be added to the driver interface. The driver model would then take the commands received from the external input and convert them to appropriate CAN signal values. The resulting action of the truck would be sent back through the output port.

4.3.7 Electrical System

The electrical system is included to enable studies of electrical energy flows. It could for instance be used to study battery operating conditions or effects of using electrically powered accessories instead of mechanically powered ones. The electrical system package is primarily intended for components without a simulated direct mechanical connection to the vehicle. The interface specifies a single special connector, see section 4.4

4.3.8 Engine

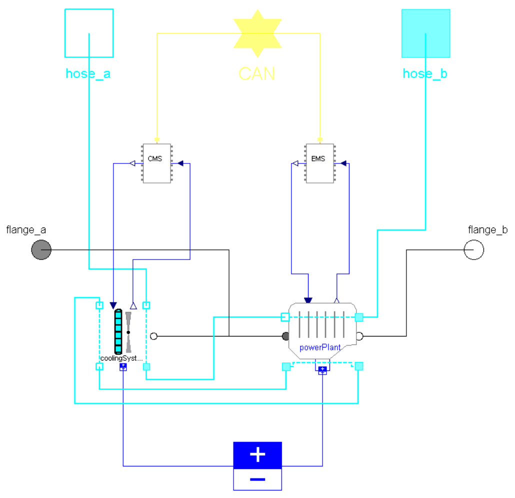


Figure 10: Engine with cooling system.

The engine (figure 10) is one of the larger subsystems, particularly when cooling system behavior is taken into account. The engine model contains submodels for both the power plant itself and any radiators and other cooling system components found in close proximity to it. The power plant model often includes an exhaust brake. The engine connectors required differ depending on the aspects considered. The base interface defines rotational mechanical connections to the auxiliaries and the transmission. The engine is connected to the vehicle signaling bus and the electrical system. An additional base class provides coolant hoses which allow coolant to flow in a circuit through other vehicle systems.

4.3.9 Trailer

The trailer is in many ways rather similar to the chassis. They are both rigid bodies represented as point masses with forces acting on them through translational connectors. The base model is a semi-trailer which can be attached to the back of a tractor. Through the use of a dolly, the semi-trailer gets a second axle and can be used as an independent full-trailer, this configuration is shown in figure 9. A trailer connects to two axles, a towing vehicle and possibly another trailer. A semi-trailer connects to one axle, a tractor or dolly, and can be used to pull a trailer.

4.3.10 Transmission

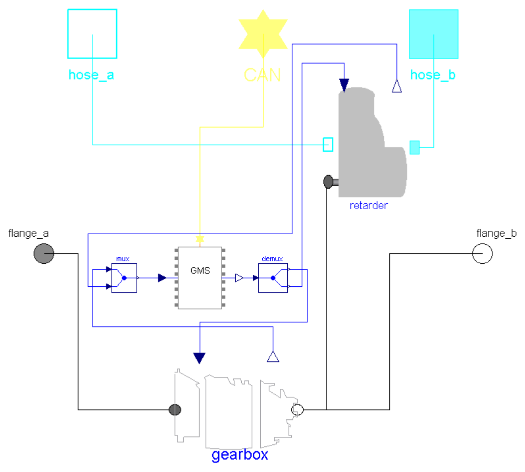


Figure 11: Transmission model with coolant flow.

The transmission includes a gearbox, which can be of any type. There is also a gear management system which has to be compatible with the gearbox used. A retarder, a type of hydraulic brake which acts on the drive shaft, is often included as well. The retarder generates a lot of heat, and a complete cooling system representation needs a connection to it. A transmission model with coolant flow is shown in figure 11. The Transmission has rotational mechanical connections to the engine and any driven axles. It is also connected to the vehicle signaling bus. An optional base class provides coolant hoses.

4.4 Special Connectors

Whenever it has been possible standard library connectors have been used in the models. In some places however we have seen the need to use special connectors which can carry all the information of a certain type between two components with only one connect statement. This has been the case for the signal bus and the electrical connectors. The electrical connector contains two electrical pins. All components connected to the electrical bus are connected in parallel, and adapters are used between the standard electrical pin connectors and our electrical connector. In the future it is envisioned that the electrical system model may contain multiple circuits and voltage levels in the electrical connector, making the advantages of using it higher. The single voltage electrical bus connector is shown in listing 5.

Listing 5: Electrical bus connector with single voltage level.

```
connector ElectricalSingleVoltage
  "Connector class for a single voltage
    electrical system"
  extends Interfaces.ElectricalBase;

  Modelica.Electrical.Analog.Interfaces
    .PositivePin p;
  Modelica.Electrical.Analog.Interfaces
    .negativePin n;
end ElectricalSingleVoltage;
```

The signal bus connector (also described in section 2.4) allows for relatively simple transfer of many (currently around 20) control signals between the various electronic control units in the vehicle. There is a second signal bus, using another identical connector except for the color and protocol, used to connect the brake system to the actuators on the axles. The media library used for the cooling system contains general hose connectors which can carry the simulated media.

5 Sample Vehicle Models

To validate the new architecture two slightly different vehicle models were used in the new framework. Both models share the same overall structure, but one version contains a thermodynamic cooling system representation (this model is seen in figure 1), while the other has no cooling system model at all (figure 12). The two models require different controllers and plant models for the engine and transmission. These are the systems which are affected by the inclusion of coolant flow. All other controllers and plant models are identical in the two versions.

The two sample models illustrate the idea of this vehicle model architecture very well. It isn't possible to generate the two versions from the same ready-

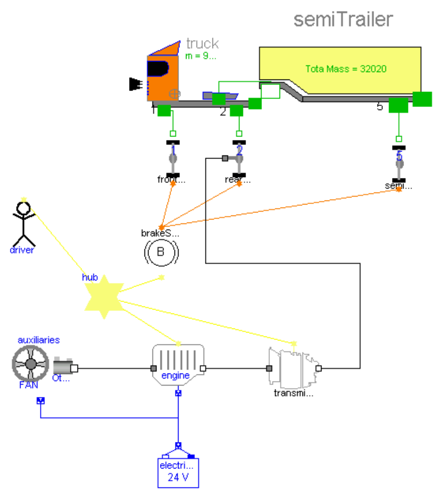


Figure 12: Heavy truck model without any cooling system.

made template only through redeclare statements. On the other hand, the same base classes and connectors are used for all common interconnects. In this way vastly differing projects can still use some of the same components. We hope that this middle ground between a completely rigid architecture and a collection of independently created models with some similarities but many differences will prove useful.

In this particular case it would of course be rather easy to create a common vehicle model, which through replaceable subsystems could be used to generate either model. However, we do not know enough today about what will be required tomorrow to incorporate all possible variations. Thus we have chosen to let the modelers of the next project and the one after that decide how to best handle their particular challenges.

The vehicle model with cooling system representation give rise to a non-linear equation system which needs to be solved iteratively at every time step. This makes simulation very slow, a performance hit of about 100 times compared to the simpler model was observed. The vehicle without a cooling system can be simulated four hundred times faster than real time on a standard laptop (Pentium^{TM}M, 1.6 GHz, 1 Gb ram). The performance difference alone is ample justification to have two vehicle model variants.

6 Limitations

During our work we have faced many compatibility issues which cannot be directly attributed to model design. Phenomena may in themselves lead to varoius solver requirements (such as the cooling system case). Existing PID controllers in external code often have strict sample time needs. Each of these special considerations needs to be explicitly stated in documentation. Documenting every important aspect is a real challenge.

Novice users of Modelica and Dymola often have trouble deciphering the error messages that are output. A nearby expert who can guide through the minefield of rookie mistakes is an invaluable asset for anyone new to the field. Similarly it is anticipated that our library will require some previous knowledge of the tools, despite our intentions to make it simple to use.

7 Conclusion

Our proposed library imposes less rigid structural control than most other vehicle architectures. Generally it does not include connector which are not used. An effort has been made at creating something which works well in the local situation. The system is not primarily intended for exchange of models with external developers. Attention has also been given to a number of practical issues related to working (updating, installing etc.) with the system.

There are many similarities between this work and the vehicle model architecture (VMA) project [6]. With some modifications many of our components could be used in VMA utilising appropriate wrappers. One key difference is the localisation of control units. We have chosen to place them inside the subsystems they control, while the VMA places them at the top level. The future development of the VMA project will be observed with great interest.

8 Future Work

We foresee a continued study of the actual use of the new models to find areas where the architecture can be improved. Inclusion of a wider range of component models is also a likely continuation. More documentation and tutorials to aid users would be beneficial.

The work described was carried out with Dymola 5.1b, but the project has since been upgraded to version 5.3b and Modelica Standard Library 2.1. Certain components have also found use in real-time hardware-in-the loop simulations using Simulink.

The recently proposed relaxations of the connector equivalency requirements in Modelica opens up interesting options for improved handling of the CAN communication.

9 Acknowledgements

The described model and library architecture has been developed based on experiences from the usage of a previous Modelica complete vehicle modeling effort aimed at fuel consumption estimation. This effort is described in a licentiate Thesis by Tony Sandberg [4]. Experiences from the work on modeling auxiliary systems by Niklas Pettersson have also been valuable [3]. Many of the subsystem models in the reference vehicle model have been converted from versions used in those projects. The work on the new library and model has been carried out as a M.Sc. thesis project by Per Bengtsson under the supervision of Niklas Pettersson [1].

References

[1] Bengtsson P. Structuring of Models Intended for Complete Vehicle Simulation. Uppsala, Sweden: Master's thesis, Division of Systems and Control, Dept. of Information Technology, Uppsala University, Dec 2004.

[2] Laine L. and Andreasson J. Modelling of generic hybrid vehicles. *Proceedings of the 3rd International Modelica Conference*, pages 87–93. The Modelica Association, 2003.

[3] Pettersson N. Modelling and Control of Auxiliary Loads in Heavy Vehicles. Stockholm, Sweden: Licentiate thesis, Dept. of Signals, Sensors and Systems Royal Institute of Technology, 2004.

[4] Sandberg T. Heavy Truck Modeling for Fuel Consumption: Simulations and Measurements. Linköping, Sweden: Licentiate thesis, Dept. of Electrical Engineering, Linköping University, 2001.

[5] Tiller M., Bowles P., and Dempsey M. Development of a vehicle model architecture in Modelica. *Proceedings of the 3rd International Modelica Conference*, pages 75–85. The Modelica Association, 2003.

[6] Vehicle Model Architecture, developed at Ford Motor Company, http://www.modelica.org/projects/vma. 2005-01-31.

Evaluation of Motor and Battery Requirements for Hybrid-Electric Powertrains during Cranking

Michael M. Tiller
Research and Advanced Engineering, Ford Motor Company
mtiller@ford.com

Abstract

Hybrid electric vehicles (HEVs) are an emerging technology for improving fuel economy and emissions. However, hybrid powertrains are expensive to manufacture because of the sophisticated electronics required. In particular, the motor and battery requirements must be carefully considered because of the cost and weight of these components.

For this reason, it is important to conduct an upfront analysis to determine the minimum requirements for the motor and battery [1]. Such an analysis ensures that the requirements for the vehicle (acceleration, fuel economy, *etc*) can be met while minimizing the incremental cost to the consumer.

This paper describes the development of engine and transmission models used to perform such an analysis for a research vehicle project. The model must take into account several important effects such as crankshaft position, engine damper design, motor design, control strategies and so on. The multi-domain modeling capabilities of Modelica allow us to formulate a model with which all these important effects can be captured [2].

This paper will show that such a model is not only capable of helping hardware designers evaluate the performance of different electrical components but also allows experimentation with various control strategies for controlling the launch clutch and drive motor.

Keywords: System engineering, hybrid electric, VMA

1 System Engineering Process

Development of a complete vehicle is a daunting task. There are numerous regulations and constraints on the development process. In addition, while the attributes of the vehicle as a whole (performance, fuel economy, emissions, *etc*) may be specified, there is a complex relationship between the design specifications for individual components and the performance of the entire vehicle system.

For this reason, system engineering principles are used to formalize the design process [3]. As part of this process, requirements are identified during the early stages of development. These requirements are then used to define performance targets for each of the vehicle subsystems (and, in turn, their constituent components). The process is often represented by the system engineering "V" shown in Figure 1.

For the application described in this paper, we are concerned with the initial requirements cascading. Based on fuel economy analysis, we know what size motor and battery are required and how much power they need to handle. However, fuel economy is only one attribute to be considered. Because we would like to eliminate the cost and weight associated with a dedicated starter motor, we also need to verify that the motor and battery combination we have chosen will be sufficient for starting the engine.

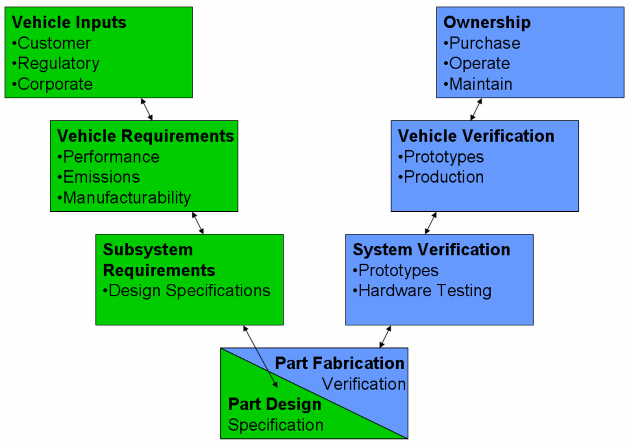


Figure 1: System Engineering Process

While it is possible to rely on "rules of thumb" or knowledge-based engineering solutions to determine requirements for conventional vehicles, it is nearly impossible to apply these to research projects. In such cases, physically-based models of the underlying systems with sufficient levels of detail and fidelity can be created that reasonably approximates the response of a physical incarnation of the design. Because design specification details should result from this process, such models need to be detailed enough to capture the effects of design changes. We term such models “design-oriented” models. In order to capture such effects it is typically necessary to make first-principles based models of the various components and use constitutive relationships based on design parameters (*e.g.* compliance, inertia, mechanical limits, *etc*) to characterize these components.

2 Powertrain Architecture

This section describes some of the relevant details about the powertrain architecture. This analysis was conducted for a research vehicle. Many of the components were relatively new and they had never been used in this particular configuration before. For this reason, models were necessary to analyze the requirements and determine component specifications.

The overall vehicle model used the Modelica Vehicle Model Architecture library [4]. Starting with the conventional vehicle architecture (shown in Figure 2), specific models for the engine and transmission were supplied that captured the physical effects required for the analysis of engine cranking. The remainder of this section will discuss each model in detail.

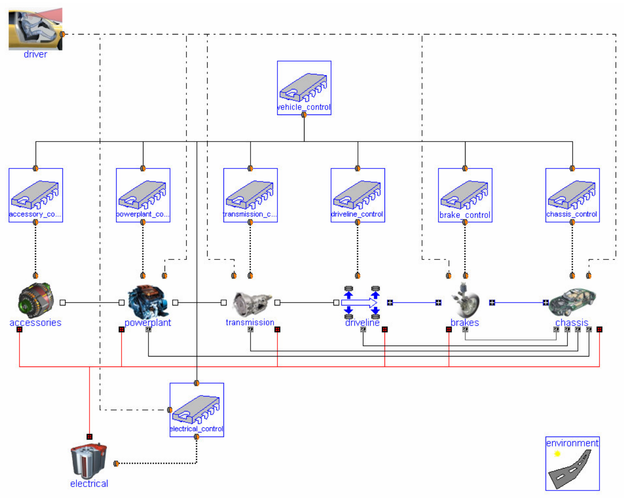


Figure 2: Vehicle Model Architecture

2.1 Engine Model

Modeling engines can be quite complicated because many factors contribute to the dynamics of the engine [5]. Because, in this application, our goal is to reach a critical engine speed in order to begin injecting fuel, we do not need to be concerned with the combustion dynamics of the engine. Instead, we focus on only those dynamics that are present before fueling begins. The engine model used is shown in Figure 3 and includes typical crank-angle based dynamics. For our analysis, two effects are particularly important.

The first effect involves the engine design itself. In particular, the compression ratio of the engine and the valve timing will determine exactly how much “resistance” is felt as we try to crank the engine. The engine configuration we are studying is a V-6 configuration so for every 720 degrees of motion in the crankshaft we will go through 6 compression events. These events normally correspond to the compression of the air-fuel mixture in preparation for combustion and the amount of work that must be done in order to perform such compression is strongly influenced by the compression ratio and valve timing of the engine.

The other effect we consider is friction. Friction is very sensitive to both engine speed and ambient thermal conditions. Friction is extremely hard to quantify because of the various non-linear effects involved (viscosity, thermal expansion, wear) and the fact that it is typically only calculated under steady state conditions for normal operating points. Because our analysis was conducted for an engine that was still in a prototype stage (without complete friction data), we will assume a conservative friction relationship and show how sensitive our results are with respect to this estimate.

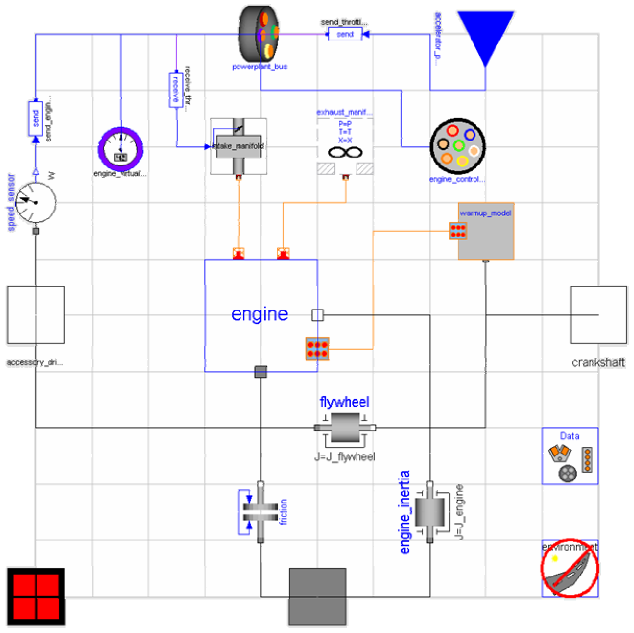


Figure 3: Engine Model

2.2 Damper Model

While the damper model is technically part of the transmission model, its design and behavior can be described independently of the other transmission components. The purpose of the "damper" is to prevent engine torque fluctuations from being propagated into the transmission and driveline. In addition to preventing these vibrations from being "felt" by the driver of the vehicle, the isolation also protects downstream components from experiencing torque reversals between combustion events leading to gear rattle and other NVH phenomena.

The damper design must be concerned with two kinds of dynamics. The first is the normal engine "torque signature" under steady operating conditions. In these cases, the damper should be as efficient as possible in transmitting energy to the transmission (to avoid a fuel economy penalty) but still isolate the fluctuations of the engine so they do not lead to downstream disturbances. The other mode involves damping of large scale disturbances (*i.e.* those that might occur as a result of pressing the accelerator pedal). It is desirable that in such circumstances the damper should "extract" energy so that these disturbances are quickly damped out.

These two, seemingly contradictory, goals are accomplished by a design, shown in Figure 4, that combines a compliant (typically multi-stage) spring in parallel with a hysteretic element surrounded by inertia elements on either side. Because of the backlash deliberately designed into the damper, the hysteresis is only triggered for large deflections (determined by the magnitude of torque carried by the element and the compliance of the spring). As a result when large disturbances are generated by the engine, the hysteresis loop removes the energy, via friction, as heat.

The damper must be tuned so that the natural frequency of the driveline is below the idle speed of the engine to avoid excitation of resonances in the driveline. However, there is also a dynamic aspect to this tuning. Because of the multi-stage design of the spring, large deflections result in the stiffer stage of the spring being involved. This increases the "effective stiffness" of the device and, as a result, raises the effective natural frequency. This leads to an interesting phenomenon. As you approach the natural frequency of the spring (for small deflections) from below, the spring will start to resonate. If this resonance leads to deflections that are large enough, the stiffer stage of the spring will begin to participate and the natural frequency will increase. If the natural frequency increases that means that a greater portion of the engine cranking will occur below the natural frequency and more resonance will occur. If this process is gradual enough, the resulting dynamics can become quite violent. To avoid this, it is desirable to move through the resonance as quickly as possible.

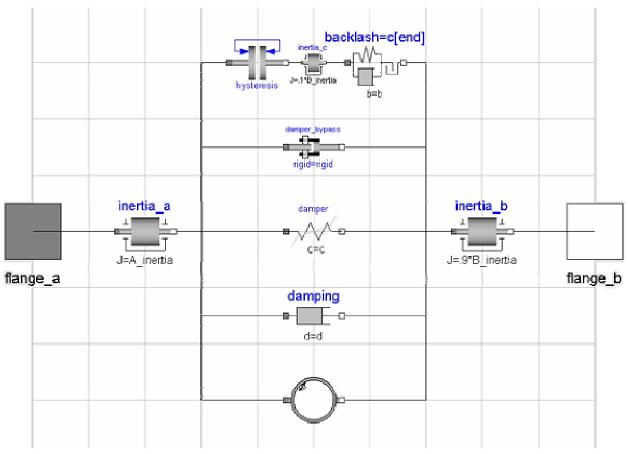


Figure 4: Damper Model

2.3 Transmission Model

Ultimately, the purpose of this analysis is to establish the power and torque requirements to crank the engine in this vehicle. Because the electric motor used for this process is contained in the transmission [6], the transmission plays an unusually key role in the starting process for this vehicle. A schematic of the transmission is shown in Figure 5.

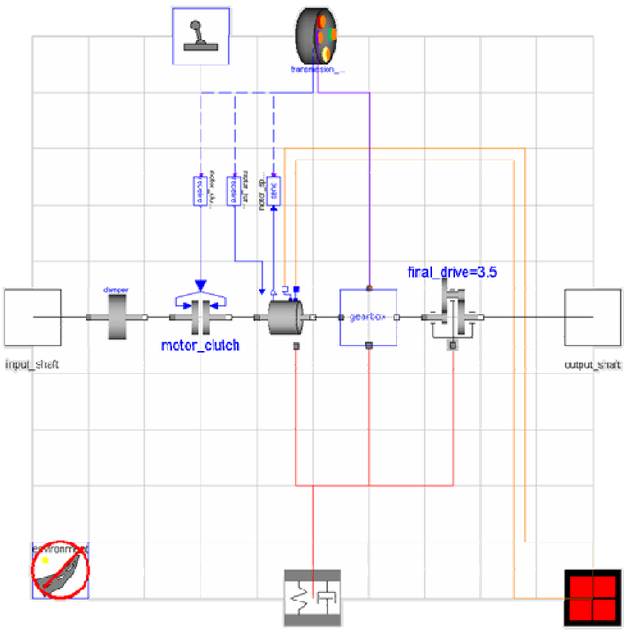


Figure 5: Transmission Model

The design of the gearbox itself is not particular important here because the transmission will not be engaged during our analysis (although it would be in a further assessment of "rolling starts"). What is important is the engagement of the motor clutch (*i.e.* the clutch that connects the motor to the engine). Another important factor is the additional inertia of the motor rotor. Although it would be nice to choose how much inertia to include in the rotor, this is largely determined by packaging constraints and the performance targets of the motor.

It is important to note that all torque used to start the engine must come from the electric motor in the transmission. There is no starting motor on the accessory side of the engine as there is in a conventional powertrain. This means that the motor design must be able to deliver tractive torque (when driving the vehicle) or cranking torque (when starting the engine). Because the peak power requirements are different, these two goals do not necessarily lead to similar designs for the motor.

2.4 Control

Starting the engine involves several discrete phases [7]. These stages are shown at the top of Figure 6. For our analysis we assume that before the engine is started the electric motor is disconnected from the engine (*i.e.* the motor clutch is disengaged) and the engine is completely at rest (phase 1). Before this motor clutch is engaged, the controller uses a PID strategy to bring the motor speed up to a specified value (phases 2 and 3). Once that setpoint has been achieved, the motor clutch is engaged (phase 4). As the clutch is engaged, torque is transmitted to the engine. The critical issue is making sure that the engine "turns over". In practice, this means that sufficient starting torque must be delivered to the engine to overcome the resistance caused by the first compression event in the engine (phase 5). The PID strategy attempts to hold the motor speed at the same setpoint during this process (phases 2-5). Once the engine has reached the desired speed (phase 6 and 7), the motor torque requirement is considerably reduced because it only needs to maintain the desired speed.

The control strategy relies on sensing two different speeds, the motor speed and the engine speed. The motor speed is known to a great degree of accuracy with very little delay. Unfortunately, the same cannot be said of the engine speed. The engine speed sensing relies on a traditional engine speed sensor which is relatively low resolution (as compared to the motor speed), is unreliable at low speeds, and is subject to considerable lag due to its design and implementation.

Using the information about the motor and engine speed, the control strategy can use two actuators, the electric motor and the launch clutch. The control strategy can specify the torque to be generated by the motor and the pressure applied to the launch clutch. Physically, this clutch pressure translates into a "clutch capacity" (*i.e.* how much torque can be transmitted through the clutch). As a result, the clutch is also effectively a torque actuator.

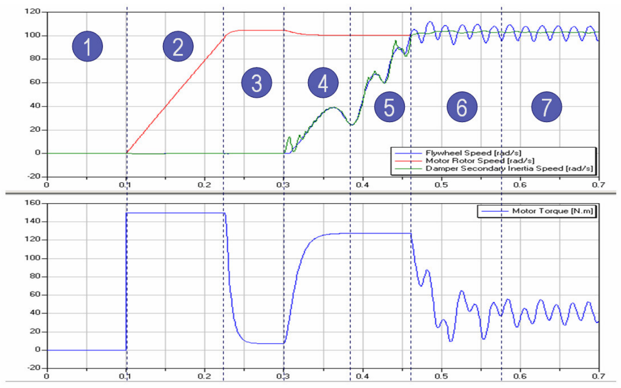


Figure 6: Baseline Analysis Results

3 Validation and Interpretation

While the use of models in system engineering to assist with target cascading and requirements analysis is useful, establishing the validity of the models used in the process is difficult. This is because, by the nature of the process, the system being engineered has not been built yet. Because design-oriented models are built using first principles, they do not rely heavily on empirical data. Instead, design data can be used to directly characterize the model.

In our case, there was existing data showing how similar hardware and control strategies functioned on a research prototype [7]. As a result, our validation focused on making sure that the response from our models matched, qualitatively, the response from actual hardware (albeit different hardware).

Figure 6 shows a typical result. Interesting qualitative features shared with in-vehicle test results:

- Saturation of motor torque during phase 2
- Magnitude of 'parasitic' loses during phase 3
- Motor torque limited to clutch capacity in phase 4 and 5
- "Brake" torque required in phases 6 and 7

A key feature of Figure 6 is the transition from phase 4 to phase 5. The boundary between these phases is defined as the point where the engine crosses the first compression event. Of particular importance during this event is the deceleration of the engine. If the engine speed approaches zero, the engine may stall (*i.e.* the vehicle will fail to start).

4 Analysis

4.1 Baseline results

Figure 6 shows a baseline response for our system. We can see the various phases of the control strategy and the results clearly indicate a successful starting of the engine (*i.e.* the engine achieved the critical speed necessary to begin fueling). We also see no evidence of any serious resonance in the driveline during cranking.

While such results highlight the important features of the experiment, there are other results that are available to us in our design-oriented model that are also useful. For example, these results confirmed that response of our damper did not exceed any of its design constraints (*i.e.* maximum deflections, maximum torque, energy dissipated, *etc*).

Another interesting feature of the simulation is the deflection of and torque transmitted through the engine mounts. While not particularly interesting for target setting and requirements analysis of the electric motor and battery, the model could be used for an additional analysis involving target setting and requirements analysis for the engine mounts.

4.2 Friction Sensitivity

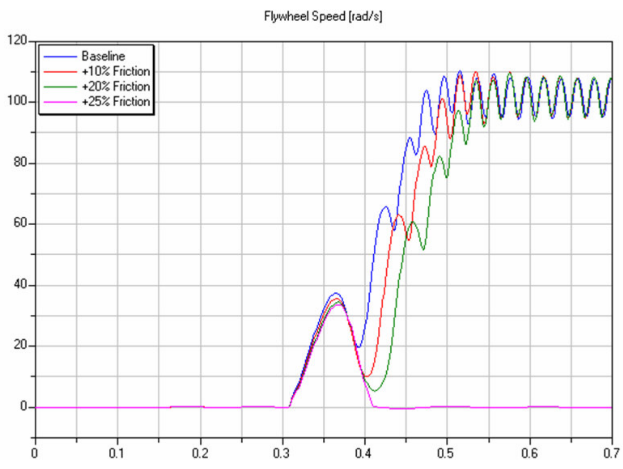


Figure 7: Friction Sensitivity

As mentioned previously, friction is a complex quantity to measure and it changes as a function of engine operating conditions. As such, we would like our analysis to be as robust as possible to our friction estimate. For this reason, it is useful to understand the sensitivity of our baseline response to different amounts of friction. From Figure 7, we see results of several different simulated experiments with different amounts of friction. All conditions are identical between Figure 6 and Figure 7 except the amount of friction. As the amount of friction is increased, the important feature to notice in Figure 7 is the dip in engine speed during the first compression event. Although the baseline case shows a successful start, an increase in friction of only 25% leads to an unsuccessful result. From this we can see that there is significant sensitivity to friction. This analysis can help us establish an upper bound on acceptable friction.

4.3 Crankshaft Position Sensitivity

Another important factor in cranking an engine is the initial position of the crankshaft. Ideally, the engine should be given as much time as possible to build up momentum as it approaches the first compression event. As shown in Figure 8, by taking our baseline case and "backing up" the starting position of the crankshaft we can significantly increase our tolerance to friction (and thereby improve our robustness to friction).

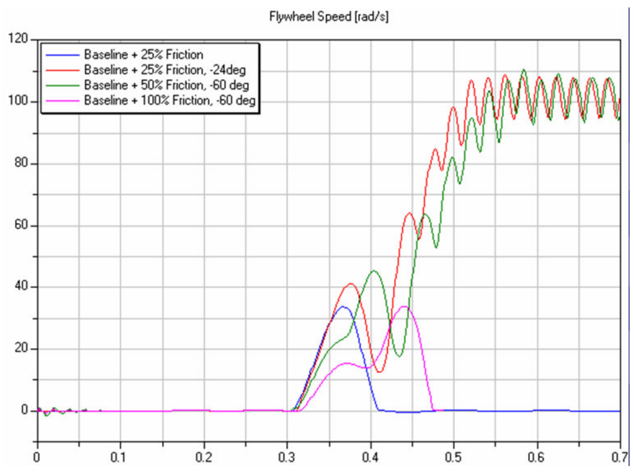


Figure 8: Crankshaft Position Sensitivity

The difficulty in this approach is that we cannot directly control the crankshaft position prior to starting the engine. So this analysis only gives us information about the fact that the results are sensitive to initial crankshaft position and highlights a need to understand the statistical variation in engine shutdown patterns (something we could also use the model to study in detail).

4.4 Performance Limits

Sections 4.2 and 4.3 addressed noise factors in the engine starting process and established sensitivities to help us gage the robustness of the procedure. Now we will turn our attention to the control strategy itself to see what we can achieve with the sensors and actuators we have available.

We will focus on two cases which we will label "Best Case" and "Worst Case". The "Best Case" scenario is important because it shows us how much excess capability we have in our electric motor under the best circumstances. This excess capacity gives us some metric by which we can gage the potential of the system to implement rolling starts (*i.e.* starting the engine while the vehicle is moving). The "Worst Case" scenario helps us to gage the limits of our design by trying to start the engine under very difficult circumstances.

Let us first consider results from the "Worst Case" analysis shown in Figure 9. For this analysis we have specified that the setpoint for motor speed control during phase 3 (see Figure 6) should be twice the speed at which the engine should be started. This "overspeed" gives us additional momentum (built up during phases 2 and 3) that we can use to generate additional torque. Our "Worst Case" corresponds to the green line in Figure 9. What this result shows us is that by using a clutch with a torque capacity of 350 N.m., we can still start the engine in the face of 200% more friction than the baseline case. Note that the additional torque used to crank the engine comes from sacrificing momentum in the motor rotor as exhibited by the deceleration of the motor rotor during motor clutch engagement.

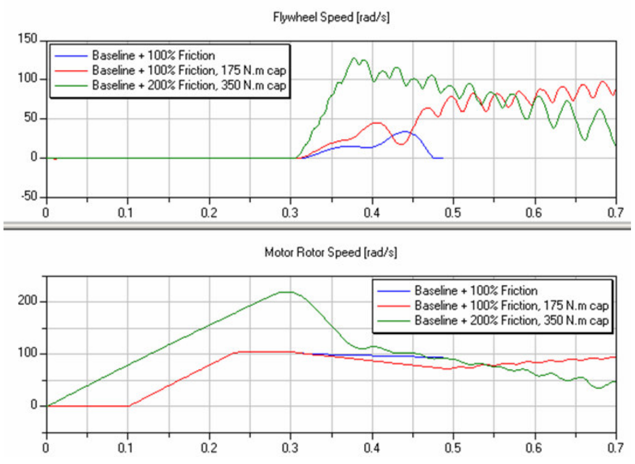


Figure 9: Worst Case Scenario

Looking at Figure 10, we see the results of our "Best Case" scenario. In this case we assume that the amount of friction to be overcome has been reduced by 40% (due, for example, to engine warm up). In such a case we see that we no longer need to use all of our motor torque to start the engine (as demonstrated by the difference in the two traces at the bottom of Figure 10). This is important because it means that we could provide some drive torque to the wheels (through the transmission gearbox) and still have enough torque left over to crank the engine. This analysis gives us some indication of how much excess is available (*i.e.* that could be used to move the vehicle forward)[1].

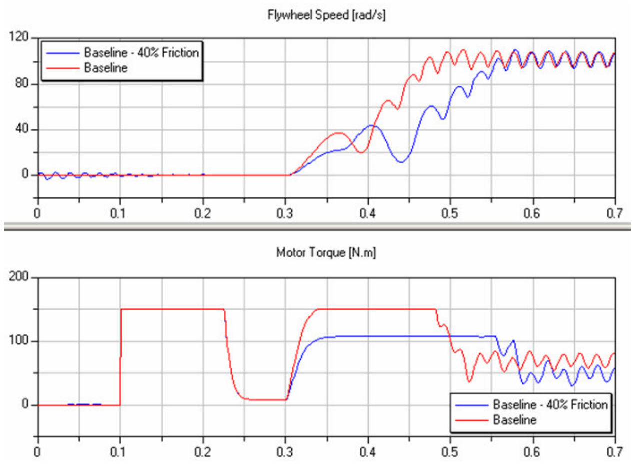


Figure 10: Best Case Scenario

5 Conclusions

The analysis in this paper supports the idea that this particular system is relatively robust with respect to the motor and battery requirements. While the response of the system is sensitive to friction and crankshaft position, the control strategy and the actuators available to it can handle the most extreme cases with enough of a safety margin.

From the analysis presented in this paper, we can see how design-oriented models can be used to guide the development of both hardware and software in the vehicle development process. Although this paper shows how this process was applied to a hybrid electric vehicle, the principle holds not only for other types of vehicles but for many product development activities in general. The key is the ability to quickly develop design-oriented models to help with upfront evaluations. This not only saves time in the development process but can save a considerable amount money by reducing or even eliminating the

[1] Of course, there are significant issues with starting the engine under such circumstances without causing significant (*i.e.* driver perceptible) driveline disturbances. However, this is beyond the scope of this paper (although **not** beyond the scope of this model).

need for prototype hardware that might have to be fabricated to support real-world testing aimed at answering the same questions.

References

[1] "Advanced Hybrid Vehicle Powertrains 2003", *SAE World Congress, 2003*. Detroit, Michigan. SAE-SP-1750.

[2] Tiller M., Tobler, W. E., Kunag, M., "Evaluating Engine Contributions to HEV Driveline Vibrations", *Proceedings of the 2nd International Modelica Conference*, Oberpfaffenhofen, Germany, 2002.

[3] Ford Design Institute, "Systems Engineering Fundamentals", 1999.

[4] Tiller M., Bowles P., and Dempsey M., "Development of a Vehicle Model Architecture in Modelica". *Proceedings of the 3rd International Modelica Conference*, Nov. 3-4, 2003. Linköping, Sweden.

[5] Batteh, J., Tiller M., Newman, C. E., "Simulation of Engine Systems in Modelica". *Proceedings of the 3rd International Modelica Conference*, Nov. 3-4, 2003. Linköping, Sweden.

[6] Ortmann W., Colvin D., Fozo S. R., Encelewski M., Kraska M., "Incorporating an Electric Machine into the Transmission Control of Ford's Modular Hybrid Transmission", SAE2004-01-0069. Society of Automotive Engineers, Inc. Warrendale, PA.

[7] Colvin D., Masterson B., "Challenges of Engine Starts and Drivability in a Parallel Hybrid-Electric System", SAE2004-01-0063. Society of Automotive Engineers, Inc. Warrendale, PA.

Session 6b

Thermodynamic Systems V

Modeling of Desiccant Assisted Air Conditioning Systems

Wilson Casas[*], Katrin Proelss and Gerhard Schmitz
Technische Universität Hamburg-Harburg
Denickestr. 17, 21075 Hamburg, Germany

Abstract

In desiccant air conditioning systems, moist air is dehumidified by means of a desiccant wheel. Water vapour is absorbed by desiccant material as humid air passes through the wheel. Using this technology, considerable energy savings can be obtained compared to conventional air conditioning systems. To evaluate the performance of the desiccant assisted air conditioning process, a model library has been developed. In this paper, control volume models for humid air and desiccant material are implemented and finally the operation of the desiccant wheel is simulated. Comparison of the dynamic and steady state results to open literature, manufacturer data as well as experimental result is used to validate the model.

Keywords: air conditioning; desiccant wheel; sorption; rotating heat exchanger, modelica

1 Introduction

In a desiccant assisted system, moist air is first dehumidified using a desiccant wheel, see figure 1. The wheel consists of a honeycomb structure, which is coated with desiccant materials such as silica gel or lithium chloride. Water vapour is absorbed by the desiccant material as humid air passes through the wheel. The moisture is released when the desiccant is regenerated by heating. For regenerating the desiccant wheel, heat input at relatively low temperatures (e.g. 60-70°C) is required, depending on the desiccant material. Using desiccant technology, the cooling demand can be reduced to 30% of that of a conventional system [2]. The energy demand for air conditioning is thus shifted from electrical to thermal energy, primary energy consumption is reduced as well and waste heat can be used.

The performance of the desiccant wheel depends on

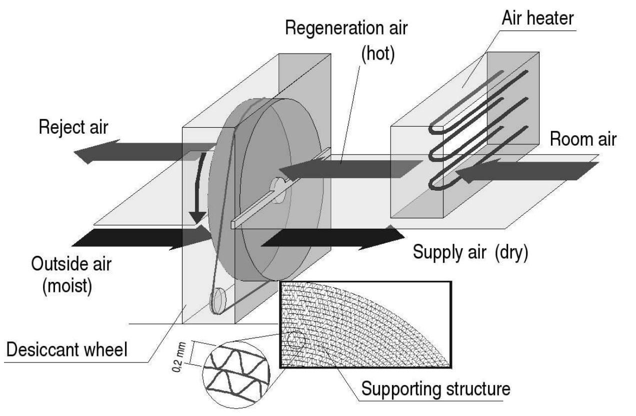


Figure 1: Desiccant wheel for air conditioning systems

several parameters, like ambient air condition (temperature and humidity), regeneration air, volume flow rates and rotation speed. Other wheel specific parameters are geometry structure and sorption properties of the material.

In order to predict heat and moisture transfer in the desiccant wheel, conservation equations for energy and mass need to be postulated. Convective heat and mass transfer are described by lumped coefficients, whereby heat and moisture transfer are coupled by the equilibrium condition of the desiccant material, the so called sorption isotherm. Conservation equations, heat and mass transfer, the sorption isotherm as well as thermodynamic state equations for air and desiccant material result in a complex, non-linear, differential algebraic system of equations (DAE).

Several solutions for such DAEs have been provided in different works. [6] introduces the concept of characteristic potentials, whereby the DAE is rewritten based on these new independant variables. Using heat and mass transfer analogy, a numerical solution for the DAE can be provided. Other authors solved the DAE by linearization and Laplace-Transformation [7], using a finite difference method [14], or finite volume method with a Gauss-Seidel iteration algorithm [13].

[*]email: casas@tu-harburg.de, Tel:+40 4042878 3079

As this short literature review shows, providing a solution for the combined heat and mass transfer problem in the desiccant wheel requires expert knowledge of the applied numerical methods as well as good programming skills. In addition, governing equations have to be rewritten in a very abstract form (for example using dimensionless variables).
A limitation of most of the methods available in the open literature is, that they can not be used for desiccant materials with discontinuities in their sorption isotherms (e.g. lithium chloride), unless the solver algorithm is extended or rewritten [9]. Hence, in case of lithium chloride as desiccant material, numerical solution methods based on heat and mass transfer analogy mentioned above can not be applied.
In this work Modelica/Dymola was used to overcome these limitations and to provide a component oriented modeling approach which offers a high level of flexibility with respect to boundary settings.

2 Physical Model

In air conditioning systems, or thermodynamics systems in general, a working fluid (e.g. air, water) flows through several components (e.g. ventilators, heating or cooling coils). Components can be modeled by algebraic equations or by partial differential equations. In this case, a distributed approach with higher discretization may be necessary. A control volume for humid air is therefore needed.

2.1 Humid air

The model for humid air described here is based on following assumptions:

(1) Humid air is an ideal mixture, Dalton's Law of Partial Pressures is valid.

(2) Air flow is incompressible.

(3) Constant specific heat capacities for dry air, water vapour and liquid water.

(4) Heat conduction in flow direction is neglected.

Balance and state equations will be postulated for a control volume according to figure 2. The control volume model is assumed to be connected to other models, whereby heat and humidity or air can be exchanged.
Specific values (e.g. enthalpy h, internal energy u) used in the equations below are defined with respect

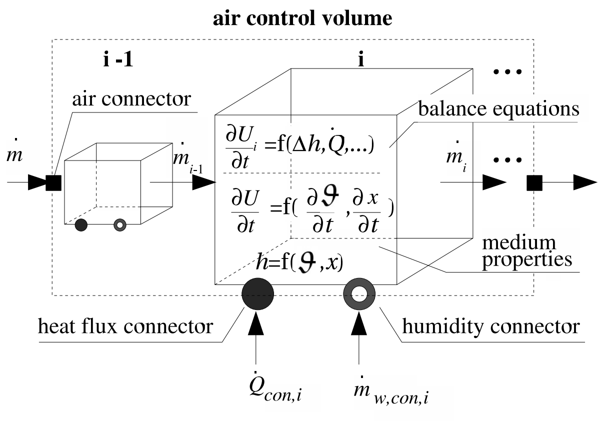


Figure 2: Air control volume

to dry air mass. Water content or absolute humidity of air x is defined as the ratio of water to dry air mass,

$$x_i = \frac{M_{w,i}}{M_i} \tag{1}$$

The energy balance for an i control volume can be written as

$$\begin{aligned}\frac{\partial U_i}{\partial t} &= \dot{Q}_{con,i} + \dot{m}_{i-1} \cdot h_{i-1} - \dot{m}_i \cdot h_i \\ &\quad + \dot{m}_{w,con,i} \cdot h_{w,con,i}\end{aligned} \tag{2}$$

$\dot{Q}_{con,i}$ is the sensible heat flux and $h_{w,con,i}$ denotes the enthalpy of the exchanged moisture. Consequently the term $\dot{m}_{w,con,i} \cdot h_{w,con,i}$ equals the latent heat flux. The dry air mass balance is given by the simple equation

$$\frac{\partial M_i}{\partial t} = \dot{m}_i - \dot{m}_{i-1} \quad . \tag{3}$$

Water balance results in the relationship

$$\frac{\partial M_{w,i}}{\partial t} = \frac{\partial x_i}{\partial t} \cdot M_i = \dot{m}_{w,con,i} + \dot{m}_{i-1} \cdot x_{i-1} - \dot{m}_i \cdot x_i \quad . \tag{4}$$

The dynamic caloric state equation can be obtained from the ideal gas equations,

$$\frac{\partial u_i}{\partial t} = \frac{\partial h_i}{\partial t} - R \cdot \frac{\partial \vartheta_i}{\partial t} \quad . \tag{5}$$

Thereby, enthalpy of humid air is

$$h_i = c_{p,a} \cdot \vartheta_i + x_i \cdot (c_{p,wv} \cdot \vartheta_i + \Delta h_V) \tag{6}$$

and its derivative

$$\begin{aligned}\frac{\partial h_i}{\partial t} &= c_{p,a} \cdot \frac{\partial \vartheta_i}{\partial t} + (c_{p,wv} \cdot \vartheta_i + \Delta h_V) \\ &\quad \cdot \frac{\partial x_i}{\partial t} + x_i \cdot c_{p,wv} \cdot \frac{\partial \vartheta_i}{\partial t} \quad .\end{aligned} \tag{7}$$

From this follows

$$\frac{\partial u_i}{\partial t} = (c_{p,a} - R_a + x_i \cdot (c_{p,wv} - R_w)) \cdot \frac{\partial \vartheta_i}{\partial t} + (c_{p,wv} \cdot \vartheta_i + \Delta h_V) \cdot \frac{\partial x_i}{\partial t} \quad . \tag{8}$$

Dynamic equations for air states may not be needed for every final component model. Hence, it makes sence to implement steady equations

$$\frac{\partial u_i}{\partial t} = 0 \quad \text{and} \quad \frac{\partial x_i}{\partial t} = 0 \tag{9}$$

to be used instead. With an appropriate parameter, the developer may be able to decide if dynamic or steady state equations are to be used.

2.2 Desiccant material and supporting structure

First, a control volume for the supporting structure according to figure 3 is defined. Since the wheel has to be discretized in axial and eventually also in tangential direction, it is advantageous to choose a "pie piece" of the wheel. The control volume will be connected later to the humid air model using the heat flux and moisture connectors. Following assumptions are made:

(1) The homogeneous, uniform wheel consists of supporting material (e.g. cellulose) and desiccant material (mass fraction χ) both with constant specific heat capacities c_r and c_s.

(2) Axial heat conduction is neglected.

(3) Humidity transport and diffusion in the wheel is neglected.

(4) Heat and mass transfer between air and matrix can be described by lumped transfer coefficients.

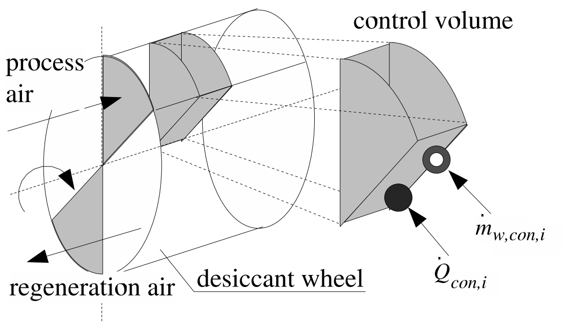


Figure 3: Convective heat and moisture transfer

The mass of the matrix material in a control volume V follows from

$$M_i = V_i \cdot \rho_r \quad , \tag{10}$$

where ρ_r is the density of the supporting structure. The humidity ratio in the desiccant material is defined as

$$q_i = \frac{M_{w,i}}{\chi \cdot M_i} \quad . \tag{11}$$

The change of water in control volume can be calculated from

$$\frac{\partial M_{w,i}}{\partial t} = M_i \cdot \chi \cdot \frac{\partial q_i}{\partial t} = \dot{m}_{w,con,i} \quad , \tag{12}$$

whereby $\dot{m}_{w,con,i}$ equals the sorbed or desorbed water. The energy balance results in

$$\frac{\partial U_i}{\partial t} = \dot{Q}_{con,i} + \dot{m}_{w,con,i} \cdot h_{w,con,i} \quad . \tag{13}$$

As for the air side, $\dot{Q}_{con,i}$ is the sensible heat, $\dot{m}_{w,con,i} \cdot h_{w,con,i}$ the latent heat transferred with the moisture. Enthalpy of sorbed water vapour comprises heat of sorption, which consists of heat of vaporization and binding enthalpy,

$$\begin{aligned} h_{w,con,i} &= \Delta h_{S,i} + c_{p,wv} \cdot \vartheta_{a,i} \\ &= \Delta h_{B,i} + \Delta h_V + c_{p,wv} \cdot \vartheta_{a,i} \quad . \end{aligned} \tag{14}$$

Binding enthalpy $\Delta h_{B,i}$ depends on the desiccant material properties and is supposed to be known from experimental data.
From the enthalpy of wet material

$$h_i = (c_r + c_s \cdot \chi + \chi \cdot q_i \cdot c_w) \cdot \vartheta_i \tag{15}$$

follows the dynamic state equation

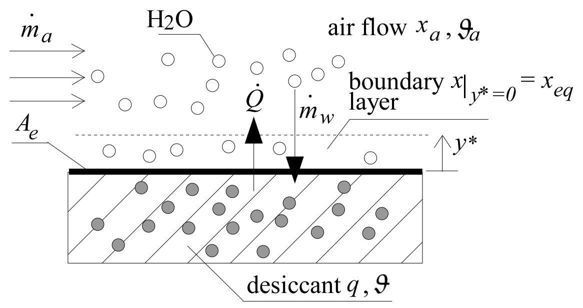
$$\frac{\partial u_i}{\partial t} = (c_r + c_s \cdot \chi + \chi \cdot q_i \cdot c_w) \frac{\partial \vartheta_i}{\partial t} + \chi \cdot q_i \cdot c_w \cdot \vartheta_i \cdot \frac{\partial q_i}{\partial t} \quad . \tag{16}$$

The convective heat transfer between air and desiccant material (see figure 4) can be described using Newton's Law of Cooling,

$$\dot{Q}_{con,i} = \alpha_i \cdot A_{e,i} \cdot (\vartheta_{a,i} - \vartheta_i) \tag{17}$$

$A_{e,i}$ denotes the effective heat transfer area, $\vartheta_{a,i}$ is the fluid temperature and α_i the local heat transfer coefficient. The effective heat transfer area can be written as the ratio of Volume to specific surface

$$A_{e,i} = \frac{V}{\omega} \quad , \tag{18}$$

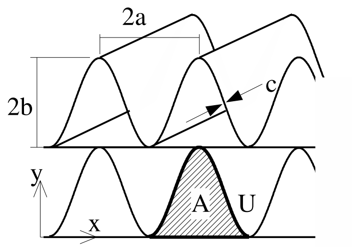


$\dot{m}_a$
H2O
air flow x_a, ϑ_a
boundary layer $x|_{y*=0} = x_{eq}$
$\dot{Q}$
$\dot{m}_w$
A_e
y^*
desiccant q, ϑ

Figure 4: Convective heat and moisture transfer

Table 1: Geometry data for desiccant wheel

$2b$	2,0 mm
$2a$	3,4 mm
c	0,125 mm
U	8,84 mm
A	3,4 mm^2
D_h	1,539 mm
ω	2600 m^2/m^3
ρ_r	129,5 kg/m^3



whereby specific surface can be calculated from rotor geometry, refer to table 1.
The *Nusselt*-Number $Nu = \alpha \cdot D_h/\lambda$ and thus also the heat transfer coefficient for laminar flow in sinusoidal ducts can be obtained from numerical work [8, 12, 1, 11].
In analogy to (17), convective mass transfer could be described by

$$\dot{m}_{w,con,i} = \beta_i \cdot \rho_{a,i} \cdot A_{e,i} \cdot (x_{a,i} - x|_{y^*=0}) \quad . \tag{19}$$

β_i denotes thereby the mass transfer coefficient and $x|_{y^*=0}$ is the moisture content of the boundary layer at the surface of the desiccant material. Hence concentration difference $(x_{a,i} - x|_{y^*=0})$ is the driving force for convective moisture transfer. It has to be assumed, that the boundary layer is in equilibrium with the wall with respect to temperature and moisture content, for instance water content of the boundary layer $x|_{y^*=0}$ equals $x_{eq} := f(q, \vartheta_w)$, namely the water content of air at equilibrium with desiccant of water load q and temperature of matrix ϑ.
Sorption equilibrium is given by the sorption isotherm

$$q_{eq} = q(p_{wv}, \vartheta) \tag{20}$$

describing the equilibrium moisture content of desiccant for a constant temperature depending on vapour partial pressure. The function $q(p_{wv}, \vartheta)$ is usually given as a correlation of measurement data. Another equivalent formulation is

$$p_{eq} = p(q, \vartheta) \tag{21}$$

denoting the equilibrium partial pressure for a known moisture content and temperature.

For most solid desiccants (e.g. silicagel), the sorption isotherm is continuous. For lithium chloride, depending on water content, sorption isotherm slope changes discontinuously due to phase changes of the system (LiCl hydrate formation). As figure 5 shows, for the interesting temperature range of $20 - 90°$C, the LiCl-water system can consist of dilute solution, saturated solution with monohydrates or anhydrous LiCl and monohydrates. Figure 6 shows sorption isotherms from experimental data [5] and the developed correlation used in this work, as well as other correlations [3].

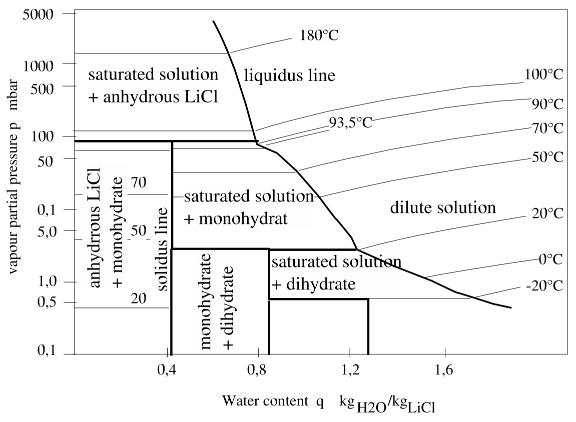


Figure 5: Phase diagram for the system LiCl-water (based on [10])

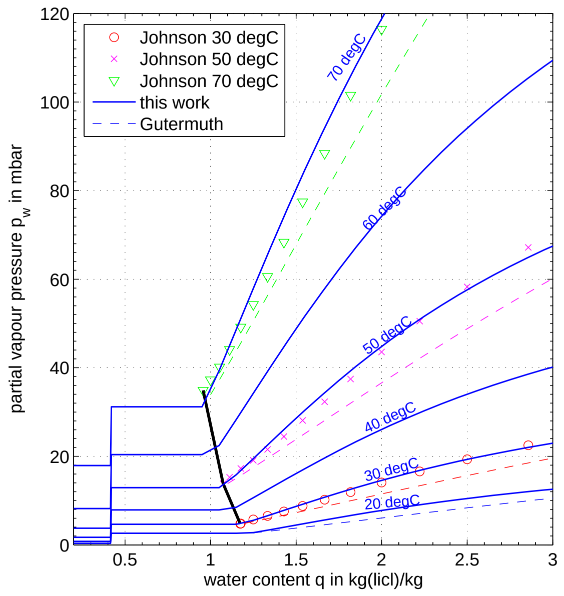


Figure 6: Sorption isotherms for the system LiCl-water

It should be noticed, that for implementation in Modelica, appropiate crossing functions should be used to avoid numerical instabilities during simulation and to reduce computation time.
If the sorption equilibrium equation (21) is known, x_{eq} in (19) follows to

$$x_{eq,i} = \frac{R_a}{R_w} \cdot \frac{p_{eq,i}(q,\vartheta_i)}{(p - p_{eq,i}(q,\vartheta_i))} \tag{22}$$

To determine the convective mass transfer coefficient in (19), the *Lewis* number $Le = a/D_{12}$ may be used to relate the two convective transfer coefficients [4]

$$\frac{\alpha}{\beta} = \frac{\lambda}{D_{12} \cdot Le^n} = \rho \cdot c_p \cdot Le^{(1-n)} \quad . \tag{23}$$

It has to be pointed out, that the relationship given above can only be used, if convective mass transfer at the surface dominates the overall mass transfer resistance. Other moisture transport mechanisms in the material (e.g. pore diffusion) are thus neglected.

3 Modelica implementation

The humid air and desiccant models were implemented in a Modelica library. The base model of the desiccant wheel `RotPair` is built up from two pairs of each one air and desiccant wall model instances, see figure 7. Each air and desiccant wall component is divided into n elements (control volumes). The two model pairs represent two opposites “pie pieces” of the desiccant wheel (see figure 3), so one model pair is in the regeneration and the other in the process air stream. After half a revolution, the boundary conditions (air inlet) changes, since the pie piece goes from process to regeneration side. Therefore, auxiliary models have to be used and outer connectors can not be joined directly to air models. In addition, during regeneration, the first element ($i = 1$) of the desiccant wall model is connected to the last element of the corresponding air model ($i = n$). Boolean signal connectors are used to change the connection order in the desiccant wall model according to the angular position of the pie piece in the wheel.
Higher discretisation in tangential direction (more “pie pieces”) can be used, but computation time increases drastically. In addition to increased number of equations in the system, more state events are generated during simulation each time boundary conditions change. Discretisation in tangential direction makes hence only sence, if inlet conditions change with time rapidly. In that case, outlet conditions have to be computed as a mean value from m pie elements. If only one pie element is simulated, and inlet conditions are constant, air outlet conditions result as mean value for half a revolution and is therefore integrated over time, e.g. for temperature

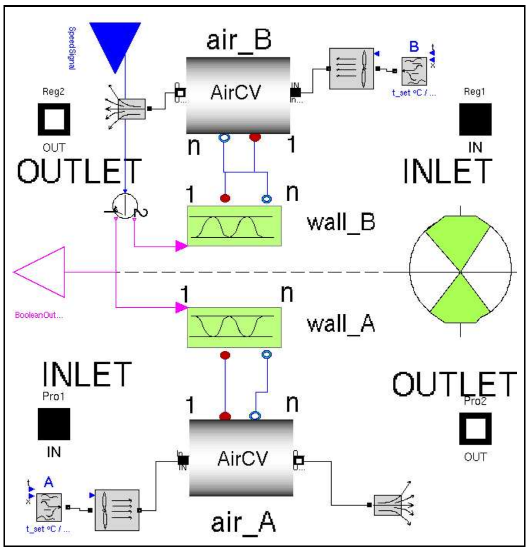


Figure 7: `RotPair` Desiccant wheel base model (diagramm layer)

$$\bar{\vartheta}_{out} = \frac{1}{\Delta t} \int_t^{t+\Delta t} \vartheta \, dt \tag{24}$$

Figure 8 plots the temperature of both pie piece control volumes at the process air outlet side and the calculated time mean value according to equation above. Concerning axial discretisation, best compromise is obtained with $n = 10 \dots 15$ for a wheel width of 250 mm.

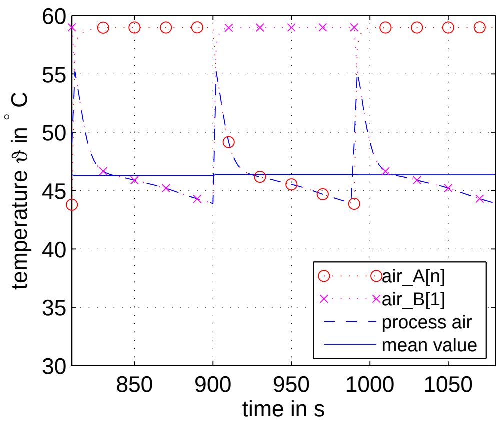


Figure 8: Mean value computation for rotating speed 1/3 rpm ($\Delta t = 90$ s)

Figure 9 shows the structure of the `RotPair` model.

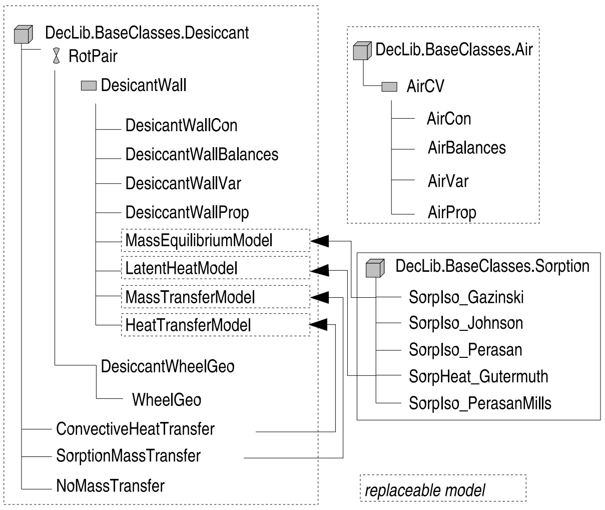


Figure 9: Structure of the model

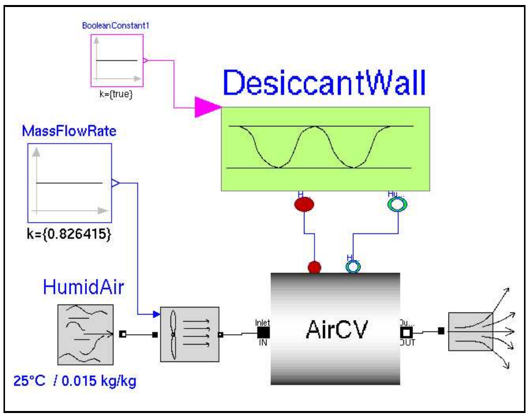


Figure 10: Test model for dynamic simulation

The `DesiccantWall` model consists of several classes containing balance equations and material properties. In order to use the same model with other desiccant materials (e.g. Silicagel) or for simulation of rotary heat exchangers (no mass transfer at all), some classes are replaceable. For instance, the `MassEquilibriumModel` containing the description of the sorption isotherm can easily be exchanged, as well as the latent (sorption) heat model.

4 Simulation results

In this section, first dynamic simulation results will be shown and compared to open literature. Subsequently, results from steady state simulation will be discussed and a total system simulation will be presented.

4.1 Dynamic simulation

The Modelica model described in this paper was used to simulate the dynamic behaviour of a rotary LiCl dehumidifier, see figure 10. The matrix is assumed to be at a regenerated state with uniform temperature $\vartheta_{init} = 75°C$ and water load of $q_{init} = 0.3$ kg/kg. The air flows through the matrix with an inlet temperature $\vartheta_{a,in} = 25°C$ and humidity $x_{a,in} = 15$ g/kg. Initial conditions and other parameters chosen here correspond to those used in [9], so simulation results can be compared.

The temperature and water content profile of the matrix with respect to the axial coordinate are shown in figure 11a,b. Each curve corresponds to one time snapshot. Figure 11c,d shows the temperature and moisture content of outlet air during the process.

At the beginning of the process, the hot matrix is cooled down very fast, since sensible heat exchange due to high temperature differences between matrix and air dominates. It can be seen that outlet temperature of air rises very quickly (inlet 25°C, outlet 73°C). But after a few seconds, the temperature in the matrix remains at 56°C. At this temperature, a phase change in LiCl-water system from hydrate to saturated solution occurs. Equilibrium condition changes and sorption rate decreases, resulting in a rising outlet moisture content of the air. The next phase change from saturated solution to unsaturated solution occurs at 46°C. During this process, outlet air states stay nearly constant. When only unsaturated solution remains, the sorption rate drops and air outlet states approach the inlet conditions rapidly.

Simulation results show good agreement with theoretical simulations in [9]. Differences in temperature and moisture content appear, which could be due to different equations for the description of sorption isotherms between [9] and the present work.

4.2 Steady state simulation

Figure 12 shows the air outlet moisture content of the air for a LiCl rotary dehumidifier (diameter 895 mm, $\dot{V} = 2300$ m^3/h) at constant process air inlet temperature $\vartheta_{a,in} = 32°C$ and rotation speed 1/3 rpm depending on inlet humidity and regeneration temperature. For comparison, manufacturer data is plotted in dashed lines.

As can be seen, outlet humidity decreases at higher regeneration temperatures, but there is a deviation of approximately 0,5 g/kg between simulation and manufacturer data. Besides of that, outlet humidity according to manufacturer appears to decrease linearly with respect to regeneration temperature, whereas sima)

b)

c)

d)

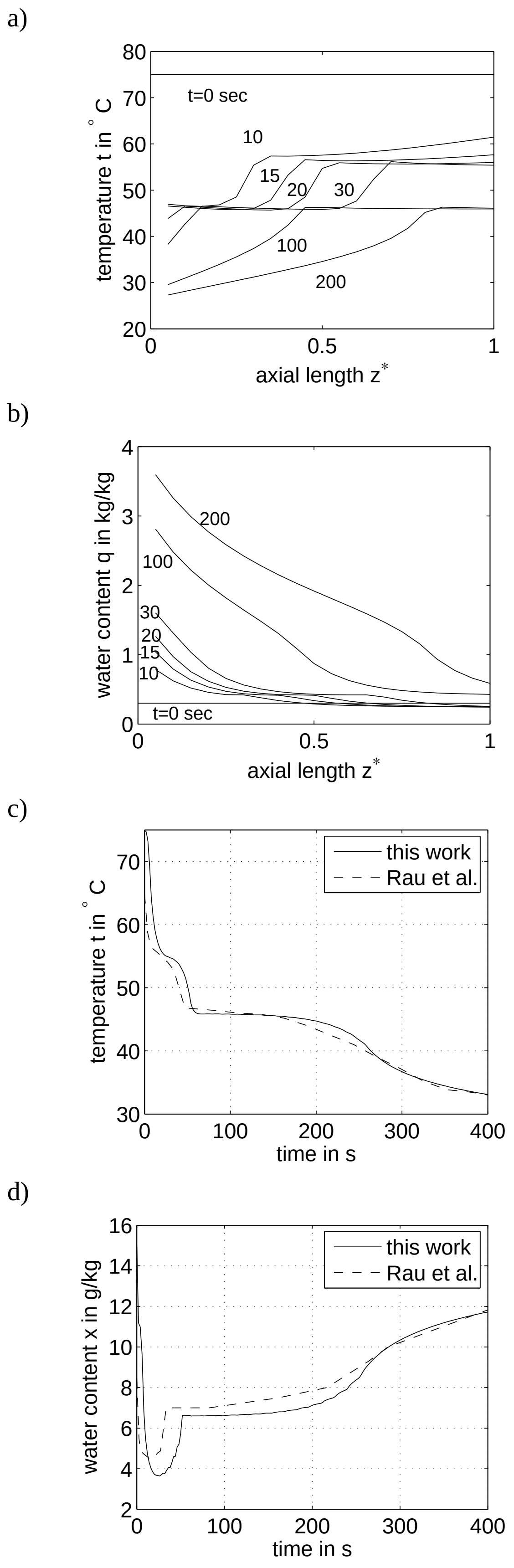


Figure 11: Dynamic simulation results: a) matrix temperature and b) water content, c) air outlet temperature and d) water content

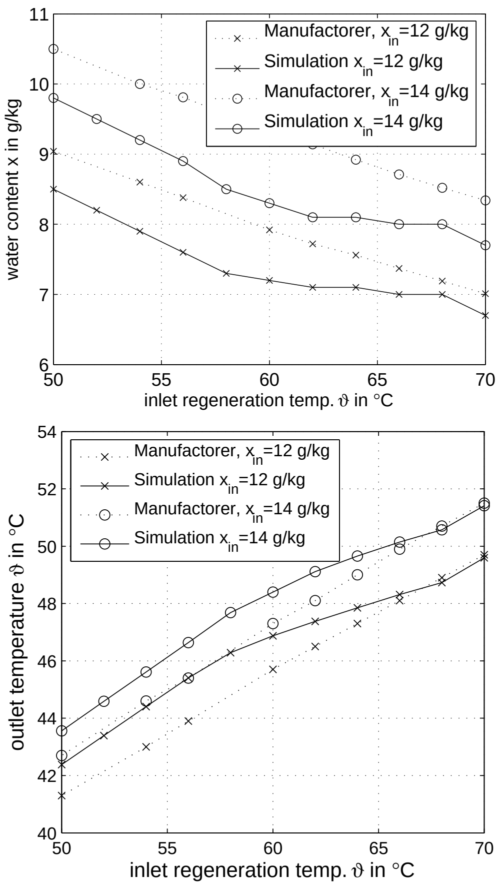


Figure 12: Air outlet water content and temperature for different regeneration temperatures, comparison between simulation and manufacturer data

ulated humidity approaches a limiting value at $\vartheta_{a,reg} = 68°C$ and decreases again for higher temperatures. This behaviour is in full agreement with the sorption isotherm: for temperatures at around 60...65°C LiCl saturated solution forms at the regeneration side. Lower matrix water load does not result in higher sorption potential according to sorption isotherm. Only if water load is reduced under a certain value, anyhdrous LiCl can form and sorption potential is higher. Although there is a limit for outlet humidity – if regeneration temperature is high enough, so that the whole matrix can be dryed ($q \to 0$), increasing regeneration temperature does not affect outlet humidity. This behaviour is in agreement with simulation results for higher temperatures (e.g. 80°C, not shown in figure 12). Manufacturer data seems not to comply with this limit and there is no data available for higher regeneration temperatures.

In terms of outlet temperature, simulation results show good agreement with manufacturer data. Deviation averages to 1°C and is even lower for high regeneration temperatures. Simulated temperature is higher than reference data, which can be explained due to higher dehumidification in the simulation. This leads to higher latent heat flux and therefore to the increased outlet temperature.

Figure 13 shows air outlet humidity for different rotating speeds at constant process and regeneration air inlet conditions. For low speeds, the matrix remains a long time on process air side and dehumidification capacity is exhausted before the drying period ends. At increasing speeds and shorter time periods, air dehumidification is higher and reaches a maximum value (lowest outlet humidity). For higher speeds, the process is dominated by heat transfer, thus dehumidification rate sinks. From figure 13 follows, that optimal rotating speed should be $u \approx 0.3-0.5$ rpm for this inlet conditions.

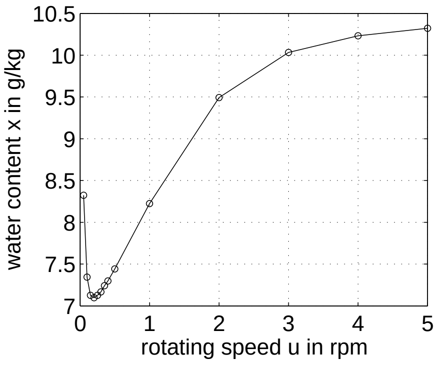


Figure 13: Air outlet water content for different desiccant wheel rotating speeds

As figure 14 shows, process air state changes change from adiabatic dehumidification ($h \approx$ const) to simultaneous mass and heat recovery from regeneration air at higher rotating speeds ($u \geq 5$ rpm). This effect is also known as enthalpy recovery and is used in practice in winter operation mode, to recover heat and humidity at different temperature and humidity levels between process and regeneration air. Hence, in winter, dry outside air can be humidified and heated by enthalpy recovery. Enthalpy recovery can be controlled by changing rotating speed.

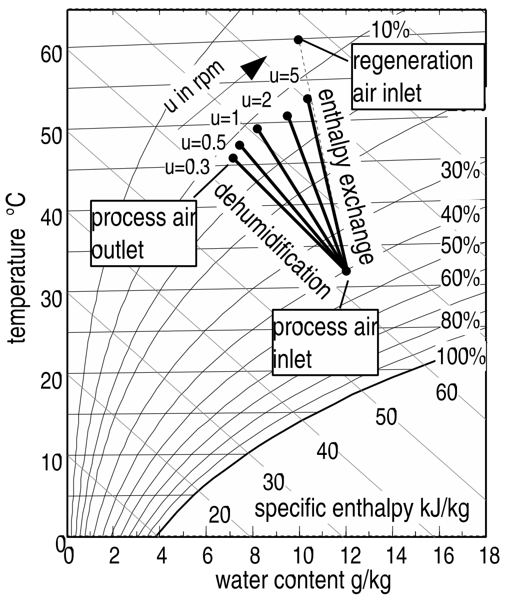


Figure 14: Air state changes for dehumidification and enthalpy exchange

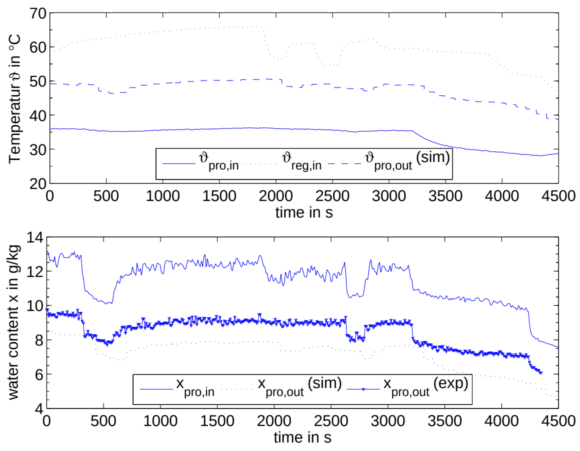


Figure 15: Simulation of LiCl desiccant wheel and comparison to experimental data

4.3 Comparison to experimental data

Within the scope of a research project, experiments have been carried out at TUHH. Measured process and regeneration air states were used as input for the simulation. Figure 15 shows the inlet temperatures and simulated outlet water content and experimental results. Deviation between experiment and simulation ammounts approx. to 1 g/kg.

It has to be taken into account, that experimental results for outlet water content are calculated from relative humidity measurement and temperature. But at higher temperatures, e.g. for outlet process air, the accuracy for humidity measurement (±3% r.H.) results in an absolute error $\Delta x = 0.8$ g/kg. Simulation's deviation from experimental results is thus nearly between accuracy range. Furthermore, air flow measurement used as input for the simulation contains also approx. $5-10\%$ error. In addition, there is actually air carry over effect and air leakages which are not taken into account in the model. For high pressure differences between regeneration and process air, air flows directly from one side to the other making it difficult to deter-

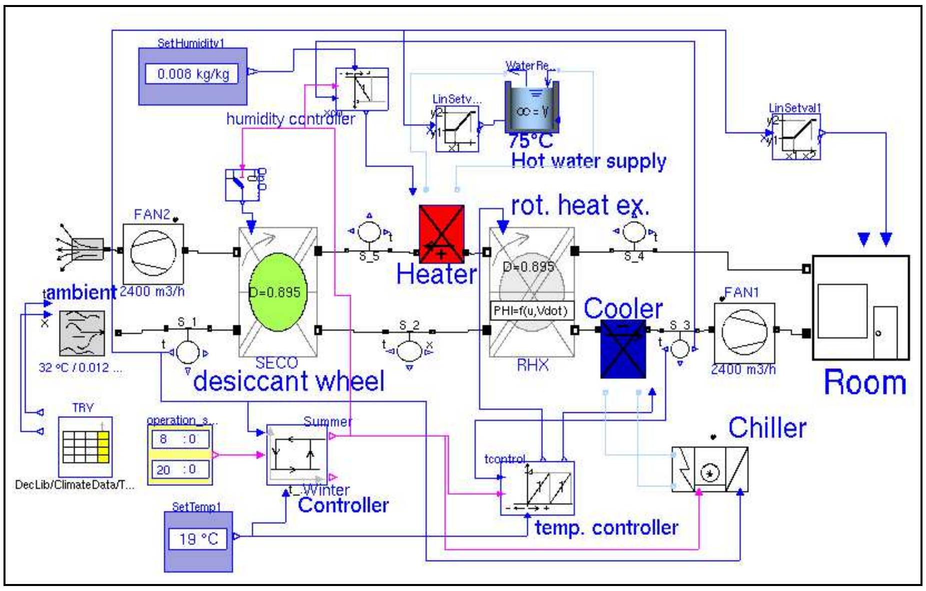


Figure 16: Desiccant assisted air conditioning plant

mine actual air state changes. Time delay in process outlet response for the simulation is an effect of mean value computation (90 s mean value). As stated above in section 3, tangential discretization should be used for changing inlet conditions.

4.4 Desiccant AC plant simulation

The presented Modelica models were used to simulate the performance of a desiccant air conditioning plant, see schematics in figure 16. The plant model includes models for reading climate data, heat exchanger (heater and cooler) and fan models as black box models, as well as models for temperature and humidity control.

Outside air first flows through the desiccant wheel, beeing subsequently pre-cooled using a rotating heat exchanger and finally cooled to supply air temperature. Air leaving the room passes first the rotating heat exchanger and is heated to regeneration air temperature in order to take the moisture out of the wheel. Temperature sequence controller regulates first the rotating speed of the rotating heat exchanger and secondly the cooling capacity of the cooler. Humidity control increases heat input in regeneration air heater if actual humidity higher than set value.

Figure 17 shows results for supply temperature and regeneration air temperature needed for dehumidification during a week in summer operation mode. Supply set value (19°C) can be mantained for the operation hours of the plant. As expected, regeneration air temperature varies with outside air humidity due to higher dehumidification load. For the simulation period, the calculated temperature of regeneration air remains under 60°C. This is advantageous, since low temperature heat sources can be used for the dehumidification process. Supply air humidity can be maintained within desired humidity level (8-9 g/kg). The resulting heating and cooling capacity can be seen in figure 18. The regeneration air heater has a heating capacity of 15 kW, whereas for cooling 7 kW are needed. Integrating capacity results in heating and cooling energy demand for the simulated period, e.g. 843 kWh heating, 434 kWh cooling demand.

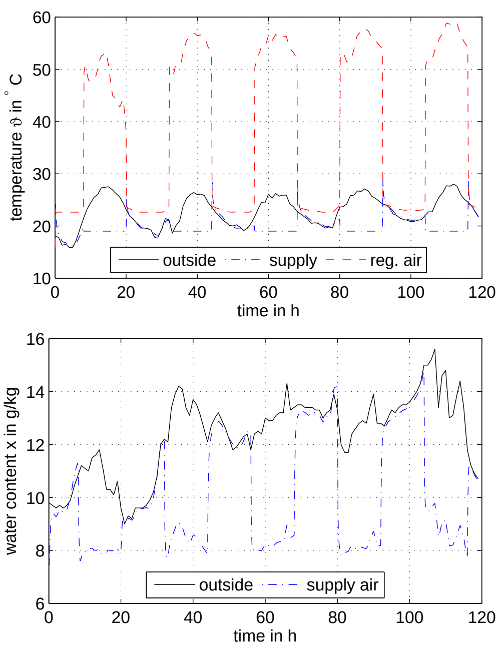


Figure 17: System simulation: temperatures and water content

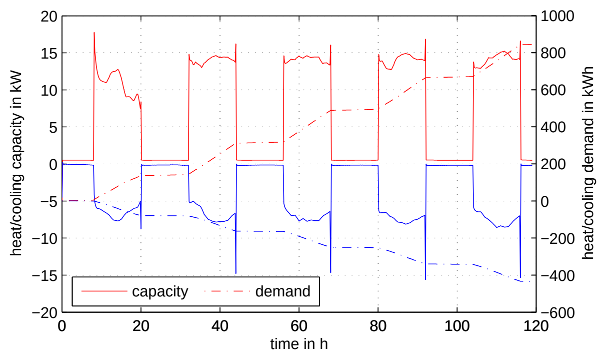


Figure 18: System simulation: heating/cooling capacity and demand

5 Conclusions

In this paper, a Modelica model for the simulation of air dehumidification by means of a desiccant wheel has been presented. The model is based on a finite volume approach for air and desiccant material. Heat and mass transfer are described by lumped convective coefficients. The model was tested for LiCl as desiccant, but can be used for other availabe desiccants, if equilibrium equations (sorption isotherm) are provided.
Transient simulation results are in good agreement with open literature. A comparison to manufacturer data was carried out. It turned out that available manufacturer data is not plausible for higher regeneration temperatures, whereas simulation returns a plausible behaviour according to desicant material properties.
Other components have also been modeled and used to calculate heating and cooling demand of a desiccant assisted air conditioning system.

References

[1] A. Campo, J.C. Morales, and A.E. Larreteguy. Pressure drop and heat transfer associated with flows moving laminarly in straight ducts of irregular, singly connected cross-sections. *Heat and Mass Transfer*, 32:193–197, 1997.

[2] W. Casas and G. Schmitz. Experiences with a gas driven, desiccant assisted air conditioning system with geothermal energy for an office building. *Energy and Buildings*, 37(5):493–501, 2005.

[3] W. Gutermuth. *Untersuchung der gekoppelten Wärme- und Stoffübertragung in Sorptionsregeneratoren*. PhD thesis, TH Darmstadt, 1980.

[4] Frank P. Incropera and David P. DeWitt. *Fundamentals of Heat and Mass Transfer*. John Wiley & Sons, 4 edition, 1996.

[5] E.F. Johnson and M.C. Molstad. Thermodynamic properties of aqueous lithium chloride solutions. *Journal of Physical Chemistry, American Chemical Society*, 55(2):257–281, 1951.

[6] I.L. MacLaine-cross and P.J. Banks. Coupled heat and mass transfer in regenerators – prediction using an analogy with heat transfer. *International Journal of Heat and Mass Transfer*, 15:1225–1242, 1972.

[7] B. Mathiprakasam and Z. Lavan. Performance predictions for adiabatic desiccant dehumidifiers using linear solutions. *Journal Solar Energy Engineering*, 102:73–79, 1980.

[8] J.L. Niu and L.Z. Zhang. Heat transfer and friction coefficients in corrugated ducts confined by sinusoidal arc curves. *Int. Journal of Heat and Mass Transfer*, 45:571–578, 2002.

[9] J.J. Rau, S.A. Klein, and J.W. Mitchell. Characteristics of lithium chloride in rotary heat and mass exchangers. *Int. Journal of Heat and Mass Transfer*, 34(11):2703–2713, 1991.

[10] K.W. Röben and J. Hupe. Zur kontinuierlichen gasentfeuchtung durch absorption und chemisorption. *Chemie-Technik*, 11(7):866–873, 1982.

[11] R.K. Shah and A.L. London. *Laminar Flow Forced Convection in Ducts*. Academic Press, New York, 1978.

[12] D.F. Sherony and C.W. Solbrig. Analytical investigation of heat or mass transfer and friction factors in a corrugated duct heat or mass exchanger. *Int. Journal of Heat and Mass Transfer*, 13:145–159, 1969.

[13] C.J. Simonson and Robert W. Besant. Heat and moisture transfer in desiccant coated rotary energy exhangers: Part i. numerical model. *ASHRAE International Journal of Heating, Ventilating, Air Conditioning and Refrigeration, HVAC& Research*, 3(4):325–350, 1997.

[14] W. Zheng and W.M. Worek. Numerical simulation of combined heat and mass transfer processes in a rotary dehumidifier. *Numerical Heat Transfer, Part A*, 23:221–232, 1993.

Modeling and Simulation of Global Thermal and Fluid Effects in an Aircraft Fuselage

Dipl.-Ing. Bettina Oehler
Air Systems Technology and Processes
Airbus Deutschland GmbH

Abstract

This paper presents a model to determine pressure, temperature and humidity in designated compartments of an aircraft fuselage based on the evaluation of mass and heat flow balances under various environmental and operational boundary conditions. A library has been developed, containing all necessary modules and aircraft templates for modeling different aircraft types. Based on an aircraft template the end-user establishes a customer cabin layout, specifies simulation and operation conditions and evaluates the results. The program code, the library and the application of the template is easily understandable and structured to minimize errors.

This paper briefly describes how the Modelica modeling language was used for the calculation of the fluid properties and the solution of the mass and energy balances. Furthermore the implementation of an aircraft system is shown with a focus on data management. Improvement potential for the use of Dymola/Modelica for this type of application is identified.

Keywords: Thermal and Fluid Effects; Aircraft Fuselage Model; Thermodynamic Library.

1 Introduction

Consideration of the comfort felt by passengers and crew is of great importance in the design phase of air-conditioning system for an aircraft. Currently there are no agreed common standards for thermal comfort in the aircraft cabin, due to insufficient knowledge of the interaction of several parameters such as temperature, air speed and air humidity. These parameters all influence thermal comfort.

In addition, customers increasingly demand variety and flexibility of cabin layouts. Special compartments for example crew rest compartments or special customized installations in the cabin, such as coffee bars, stair-houses and beauty salons, can be installed at various locations in the aircraft cabin.

In order to obtain a better understanding of the thermal and fluid effects in the whole fuselage a highly detailed simulation model has been developed by Airbus, named Overall Fuselage Flow Model (OFFM).

This model will support the work being carried out in the subproject Individual Seat Climatisation (ISC) of the research project Friendly Aircraft Cabin Environment (FACE). For the ISC subproject realistic boundary conditions are needed. The aim of ISC is to optimize the percentage of people that are satisfied with thermal comfort in aircraft cabins.

The simulation model is intended to provide boundary conditions for detailed CFD simulations.

The tool ultimately selected, Dymola/Modelica, best fulfills the tool requirements, such as connecting elements to networks, managing a high number of components without increasing complexity, available flow and heat libraries and accessibility to source code. For the simulation Modelica version 2.1 and Dymola version 5.3a are used.

2 Model Description

The modeled fuselage consists of all compartments inside the pressurized fuselage. The model boundaries related to the mass flows are the total air inlet into the mixer unit and the total air outlet flow of the fuselage. The temperature boundaries are the air temperature at the compartments' air outlets i.e. in the cabin, flight deck, forward and aft cargo compartment and temperatures for all adjacent, unpressurized compartments, for example main and nose landing gear, wing box and the skin temperature.

2.1 Description of Aircraft Air-Conditioning Systems

The environmental control system (ECS) supplies conditioned air to maintain the desired temperatures, ventilation rates and pressurization within the cabin zones and flight deck.

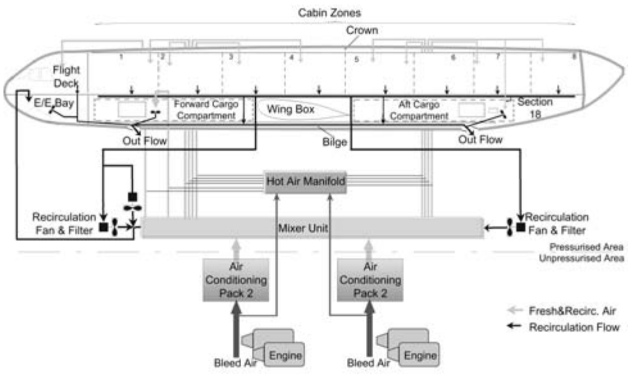


Figure 1: General Design of Environmental Control System

The ECS consists of the air-conditioning packs, the mixer unit, the air distribution/recirculation system and the pressurization system, including the associated fans. These different systems are shown in figure 1 for the A340-500/600.

The pneumatic system provides hot, pressure regulated bleed air from the engines (or Auxiliary Power Unit APU or external sources) to the air-conditioning packs.

Two air-conditioning packs, consisting of heat exchangers, compressor, water extraction and turbine, located in the unpressurized area below the center wing box provide conditioned air to the mixer unit.

In the mixer unit, installed under the cabin floor in front of the center wing box, the conditioned air is mixed with recirculated and filtered cabin air (extracted out of the cabin at floor level through dado panels), which is fed by recirculation fans to the mixer unit.

The air from the mixer unit is distributed to the cabin temperature control zones (for the A340-500/600 eight cabin areas are defined), the flight deck and cargo compartments. The independent temperatures for the different compartments can be achieved by directing hot trim air from the hot air manifold to the distribution ducts. In addition ventilation is provided for the electrical and electronic racks (E/E).

The cabin pressure is regulated automatically by the outflow valve position depending on the required cabin pressure change rate and differential pressure between cabin and ambient.

2.2 Conceptual Model Design

The overall fuselage flow model (OFFM) determines air mass flows, pressures, heat flows, temperatures and humidity of the entire aircraft fuselage for different ground and flight conditions.

For the one dimensional mass and heat flow calculation, the fuselage volume is divided into individual control volumes, such as flight deck, cabin, crown, E/E bay, cargo compartment, etc., see figure 2 and figure 3.

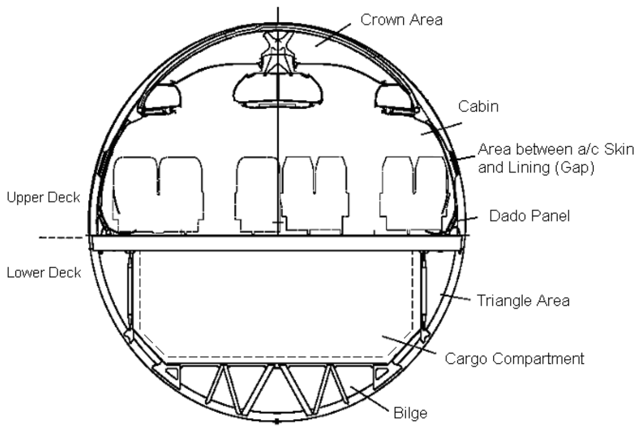


Figure 2: Aircraft Compartments in the Fuselage – Cross Section Overview

The modules include different thermo-fluid components, which cover the physical/mathematical description for the mass and heat flow within the fuselage.

The fuselage airflows are simulated by calculating the condition of the air within, and the pressure loss between these discrete volumes. For the heat flows all relevant heat sources and sinks, the heat exchange with adjacent control volumes and enthalpy flows are considered.

The release of humidity through metabolic processes is considered depending on the passenger load and in relation to the outside air flow per occupant provided to the cabin.

The simulated aircraft model consists of a federation of aircraft modules and overlaying system modules.

The aircraft modules are divided into fuselage standard modules and layout modules. Fuselage standard modules consider all aircraft modules, which are layout-independent. For example the following modules do not differ for one aircraft type with different system and cabin layouts: crown, triangle area, E/E bay, bilge, etc.

The cabin area is subject to various layouts such as different galley/lavatory positions and seat layouts (first, business and economy class). Nevertheless several cabin area modules or parameters are layout-independent such as ceiling air outlets, cabin length, fuselage diameter, doors and the zone classification.

Consequently the specific layout modules are cabin installations such as galley, lavatories, seat rows, staircases and the utilization of the lower deck compartments. The whole or partial volume of the cargo compartments can be occupied e.g. with crew rest compartments.

The system modules contain the air supply and air extraction system, the cabin pressurization system and several individual fans. An example for the system implementation is given in detail for the air supply system in chapter 5.

The interaction between the system and the aircraft standard modules or specific layout modules is considered in two different ways; either by port connections or by a value assignment via variable names. No additional inputs in the system modules by the user are necessary.

With the aircraft standard modules, layout specific modules and the system modules a specific aircraft layout can be built, as shown in figure 3.

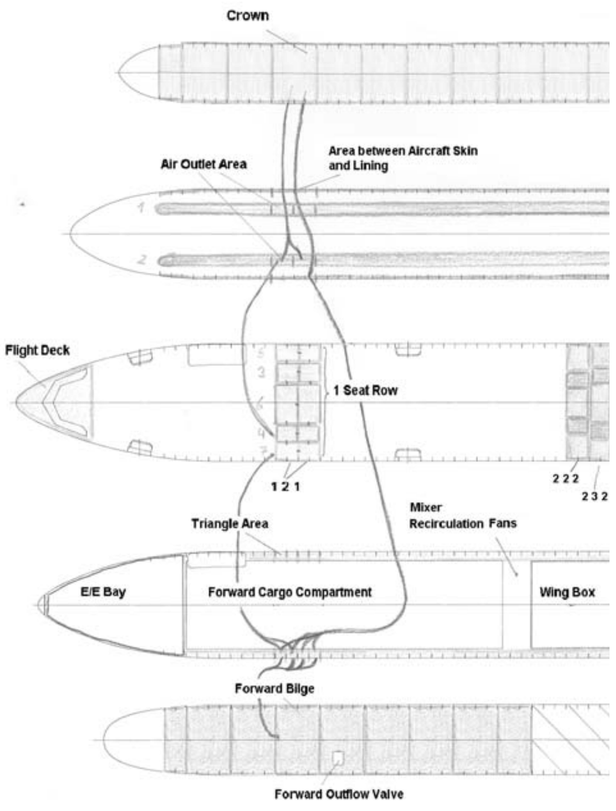


Figure 3: Principle Aircraft Template – Top View

Therefore a template is provided by the model developer containing all fuselage standard modules, system modules and an User Input Parameter Window (parameter dialog). This aircraft template is the basis for the user to create a new layout by implementing the layout specific modules and entering the input values and boundary conditions for the simulation for different cabin, system and environmental conditions. The data management of the model in detail is described in chapter 4.

The simulation model offers the possibility of calculating

- isothermal or non-isothermal conditions,
- with dry or humid air and
- with different flow resistance laws.

For this application only the steady state results are of interest.

The implementation of the above described system and compartment modules results in a large and complex model, so that the model must be modular, adaptable and user-friendly. It is therefore important that the developer and the end-users can easily handle the model.

2.3 Library Structure

A special library is developed to establish a model of the fuselage. It consists of the control volume, the basic components for flow and heat such as flow and heat resistances, constant pressure sources/sinks and the property models for dry and humid air. Derived from these general basic elements a specific library section was created containing the aircraft modules.

One of the most important principles for the development of the library was simplicity, comprehensibility and reusability. In this respect the ThermoFluid [1] library was proofed to be too complicated. Moreover the Modelica_Fluid and Modelica_Media were not available when starting this project.

The OFFM library contains 6 main `packages`. Each `package` contains sub-packages, as described below.

1. General
   - Additional Units,
   - Functions,
   - Parameter,
   - Constant and
   - General Material Properties.
2. Icons - Icon Definitions
3. Interfaces - Flow and Heat Ports
4. Components
   - Control Volume,
   - Mass Flow Source,
   - Pressure Source,
   - Pressure Loss,
   - Heat Source,
   - Const. Temperature and
   - Heat Resistances.

5. Media – Dry Air and Humid Air
6. Aircraft_Models - contains packages for each aircraft type with the following sub-packages:
   - aircraft specific Data (Flow/Heat/Boundary Conditions),
   - layout-independent modules (aircraft standard modules and the corresponding assemblies),
   - aircraft complete (aircraft template) and
   - layout-dependent modules (Cabin, Lower Deck and System).

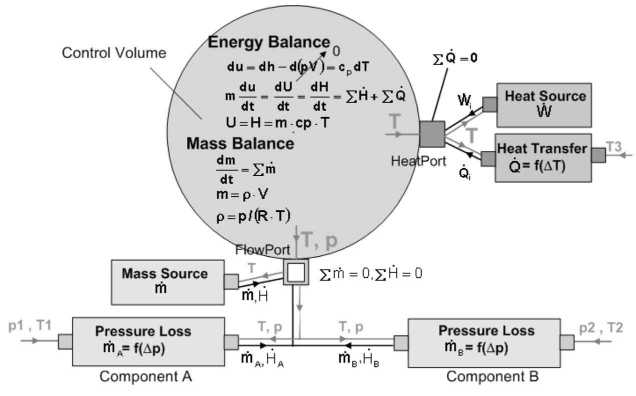
3 Implementation of Physical-Mathematical Description

3.1 Mass and Energy Balances

For the mass balance all internal mass flow sources and mass flows resulting from the control volumes differential pressures must be identified. Each control volume contains one definite flow port at which the sum-to-zero equations for all incoming and outgoing masses are automatically generated, due to the `flow` prefix. In the same flow port the enthalpy flows are determined correspondingly.

Additionally each control volume contains one definite heat port, where all heat flows resulting from internal heat sources and all heat flows transferred across the borders of the control volume are summated.

In the control volume the average value of pressure and temperature is calculated by solving the mass and energy balances, as depicted in figure 4 (the equations in the control volume are completed by humidity here not shown).



Control Volume
Energy Balance
$du = dh - d(pV) = c_p dT$
$m\frac{du}{dt} = \frac{dU}{dt} = \frac{dH}{dt} = \Sigma\dot{H} + \Sigma\dot{Q}$
$U = H = m \cdot cp \cdot T$
Mass Balance
$\frac{dm}{dt} = \Sigma\dot{m}$
$m = \rho \cdot V$
$\rho = p/(R \cdot T)$
$\Sigma\dot{Q} = 0$
Heat Source $\dot{W}$
Heat Transfer $\dot{Q} = f(\Delta T)$
HeatPort
T3
T, p
FlowPort
Mass Source $\dot{m}$
$\Sigma\dot{m} = 0, \Sigma\dot{H} = 0$
p1 , T1
Pressure Loss $\dot{m}_A = f(\Delta p)$
Pressure Loss $\dot{m}_B = f(\Delta p)$
p2 , T2
Component A
Component B

Figure 4: Mass and Energy Balances

The models are developed in such a way that it is possible to run a simulation under isothermal or non-isothermal conditions. The variable `includeEnergyBalance` is `Boolean` and can be set to true or false, depending on whether the energy balance is used or not. The Modelica implementation is as follows:

```
…
if (includeEnergyBalance) then
   der (H)= FlowPort.H_dot + Heat
        Port.Q_dot;
   H = if (includeHumidAir) then
                  humidAir.cp*m*T)
        else (dryAir.cp*m*T);
else
   der(H) = 0;
   T = T_0;
end if;
…
```

3.2 Pressure Loss

The pressure loss between the control volumes is described by the following equation:

$$\Delta p = k_1 \cdot \dot{m} + k_2 \cdot \dot{m} \cdot |\dot{m}|, \tag{3.1}$$

with k_1 = Linear Flow Coefficient,
k_2 = Quadratic Flow Coefficient.

k_1 and k_2 are calculated values based on flight test evaluation.

For the calculation of the mass flow the following equation is implemented:

$$\dot{m} = \text{fdir} \cdot \left(\frac{1}{k_2}\right) \cdot \left[-\frac{k_1}{2} \cdot \sqrt[\text{Thermo RootFct.}]{\left(\frac{k_1}{2}\right)^2 + k_2 \cdot |\Delta p|} \right] \tag{3.2}$$

with `fdir` as an indicator of the flow direction, it can reach the value 1 for design flow and –1 reversal flow,

the `thermo root function` from the free library ThermoFluid [1], with a linear interpolation near 0.

The scale of the parameters is given in the following: The flow resistance values varies between 10^2-10^5 $Pa \cdot s^2/kg^2$ depending on there location, whereas the pressure difference between the control volumes is in a range of 0.1 Pa up to 10 Pa which results to mass flows from 0.001 kg/s up to 0.01 kg/s.

The leakage flow from aircraft fuselage to ambient is calculated by the pressure vessel equation:

$$\dot{m}_{ij} = A_{eff} \cdot \sqrt{\frac{2p_i^2}{R \cdot T_i}} \cdot \sqrt{\frac{\kappa}{\kappa - 1}\left[\left(\frac{p_j}{p_i}\right)^{2/\kappa} - \left(\frac{p_j}{p_i}\right)^{(\kappa+1)/\kappa}\right]} \tag{3.3}$$

3.3 Fluid Properties

In the following section the calculation principle of the fluid properties for dry and humid air is described. The `package` Media contains the following classes separated for dry and humid air:

- Declaration and initialization of the fluid property variables such as specific enthalpy, thermal conductivity, dynamic viscosity, gas constant and specific heat capacity in a `record` and
- a `block` containing inputs and outputs and an equation part. The input of the block for dry air is the temperature and for humid air additionally the pressure and water content. The outputs are the property values. The equation part contains the empirical correlation based on [2] to calculate the fluid properties as a function of inputs.

  The `block` class is used instead of a function in order to avoid differentiation of functions.

In the following paragraphs the interaction of the above referenced `block` and `record` class is described for dry air. Each model, where the fluid properties are needed, contains an instance of the `record` and `block` in the declaration part in the following way:

```
OFFM.Media.Dry_Air.DryAirProp
                     Dry_air(T=T);
OFFM.Media.Dry_Air.DryAirRec dryAir=
                     DryAir.DryAirRec;
```

In the `block` class `DryAir` the values of the fluid property variables are calculated, depending on the temperature and are written to the `record dryAir`.

For humid air the fluid properties are calculated in the same way, but as a mixture of two ideal gases, air and water vapor.

The equations, which must be considered in a descriptive physical-mathematical model, for example in the flow resistance model, are the following:

```
…
OFFM.Media.HumidAir.HumidAirProp
HumidAir(T=T,x=
   if (includeHumidAir) then
   x
   else 0,p=p_upstream);
OFFM.Media.HumidAir.HumidAirRec
   humidair=HumidAir.HumidAirRec;
OFFM.Media.DryAir.DryAirProp
                          DryAir(T=T);
OFFM.Media.DryAir.DryAirRec dryair=
                     DryAir.DryAirRec;
…
```

For the inputs of the `block` class the upstream values are used.

The variable `includeHumidAir` is `Boolean` and can be set to true or false, depending on whether dry or humid air is called for. The fluid properties are called up from the corresponding record by means of an `if`-expression, as shown in the following:

```
cp = if (includeHumidAir) then
        (humidAir.cp) else (dryAir.cp);
```

Additionally for the calculation of humid air the equations for the water content of saturated air and for the pressure of saturated water vapor in air are relevant (Antoine Equation). The pressure of saturated water vapor in air is described by two different equations, one for the vaporization and the other for sublimation. At a temperature of ϑ=0.01°C [3] solid, liquid, and vapor phases of water coexist in equilibrium and a discontinuity occurs between these two curves. To establish a continuous function for the pressure of saturated water vapor in air the curve for vaporization is extended up to a temperature of ϑ=0.0045°C and the curve for sublimation is only valid up to this point.

The following approaches are used for improvement and to avoid numerical problems:

1. The Horner Scheme is applied to the polynomial for the fluid properties calculation. This has the advantage that in the converted representation no exponentials are used, only multiplications and summations.
2. `if`-expression is used in preference to `if`-clause for conditional statements.

4 Data Management

The data is divided into internal parameters and user inputs, as shown in figure 5.

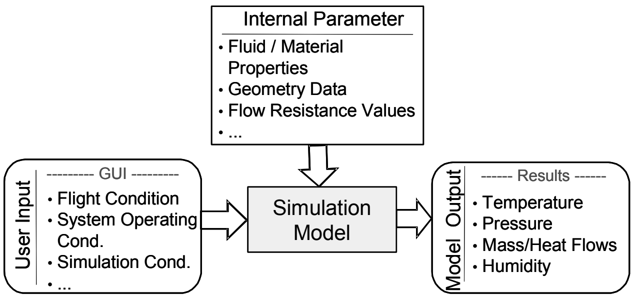


Figure 5: Input and Output Parameters

The first section in this chapter will discuss the data management of the internal parameters and is followed by the user inputs. The model outputs are not further discussed.

Internal Parameters: The internal parameters are classified into parameters, which are valid in general or only for a specific aircraft type. General parameters are e.g. the fluid and material properties and are structured in the library in the `package` General (see section 2.3). Specific aircraft type data are for example geometry data and flow resistance values and are contained in the `package` Aircraft Specific Data. All these data are stored in a `record`.

In addition the aircraft environmental parameters, such as ambient pressure, temperature, humidity or the external heat transfer coefficient, as well as the boundary temperatures, such as the skin temperature or the temperature of the unpressurized compartments, belong to the internal parameters. All of them are functions and mostly depend on the flight condition, for example on Mach number and altitude.

The calculation of the aircraft skin temperature depending on the absorbed solar radiation and hence on the location of the fuselage surface is described here. There are six skin temperatures, three for the upper fuselage (crown, cabin left side and right side) and three for the lower fuselage (bilge, triangle area left side and right side) whereby the calculation distinguishes between ground and flight condition. Depending on the angle β of solar radiation the absorbed radiative heat on the aircraft skin is a function of the polar coordinate φ. The direct sunbeams irradiate one half of the skin surface of the fuselage, the other half is irradiated by diffuse sunbeams. The diffuse radiation can be neglected. In figure 6 the absorbed radiative heat by solar radiation collected per projected surface area is shown. This radiative energy curve is a cosine function, which is at its maximum where it meets the irradiation angle and its minimum at the point where the solar radiation contacts the aircraft skin tangential. In case of a cloudy day solar radiation energy is attenuated by clouds. This is taken into consideration by the factor ω, below altitudes of 22000ft. The average radiative energy $\overline{G}$ for each area can be depicted (see figure 6) and determined by following equations:

$$\dot{Q}_{rad,energy} = \omega \cdot G_0 \cdot A_{proj} \qquad (4.1)$$

$$\text{with} \quad G_0 = f(\text{altitude}), \qquad (4.2)$$

$$A_{proj} = \frac{1}{\Delta\Phi} \int_{\varphi_1}^{\varphi_2} r(\varphi) d\varphi \cdot A_s \,. \qquad (4.3)$$

The surface area, which can be calculated for the aircraft skin, is $A_s = \frac{1}{2} \cdot D \cdot L \cdot \Delta\Phi$, with $\Delta\Phi = \varphi_2 - \varphi_1$. For $r(\varphi)$ two different functions are defined for the shady (1) and sunny (2) side of the fuselage, depending on the polar coordinate and irradiation angle.

(1) For $\varphi > (\beta+90°)$ and $\varphi < (\beta-90°)$ $\quad r(\varphi)= 0$.

(2) For $(\beta-90°) \leq j \leq (\beta+90°)$ $\quad r(\varphi)=\cos(\beta-\varphi)$.

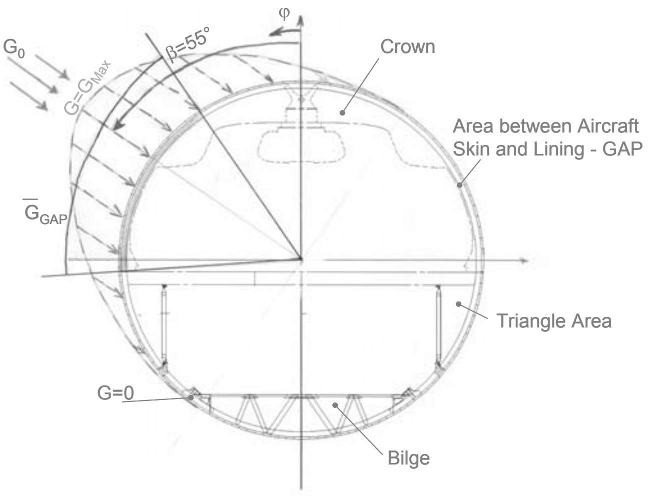
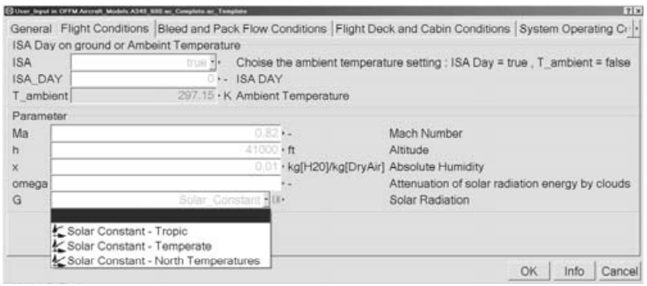


Figure 6: Aircraft Skin Temperature

The implementation in Modelica is difficult, because the integral is not varying with time but with the polar coordinate φ. So, for each range the integral is solved manually depending on the angle of solar radiation.

User Inputs: The parameter input of the user takes place in the parameter dialog, as shown in figure 7. It is contained in the aircraft template. The content of the parameter dialog is stored in a `record`. The statements in Modelica are set by `dialog-annotations` and can be found in [4].


General | Flight Conditions | Bleed and Pack Flow Conditions | Flight Deck and Cabin Conditions | System Operating C

ISA Day on ground or Ambeint Temperature

ISA	true		Choise the ambient temperature setting : ISA Day = true , T_ambient = false
ISA_DAY	0	-	ISA DAY
T_ambient	297.15	K	Ambient Temperature

Parameter

Ma	0.82	-	Mach Number
h	41000	ft	Altitude
x	0.01	kg[H20]/kg[DryAir]	Absolute Humidity
omega		-	Attenuation of solar radiation energy by clouds
G	Solar_Constant		Solar Radiation

Solar Constant - Tropic
Solar Constant - Temperate
Solar Constant - North Temperatures

OK | Info | Cancel

Figure 7: Parameter Dialog for User Inputs

User inputs are all operating conditions such as flight conditions, environmental control conditions and the simulation conditions, where the user has the possibility to choose whether the simulation shall run under isothermal or non-isothermal conditions and with or without the consideration of humidity.

In the parameter dialog the user has different possibilities to make his inputs. For example for the tab Flight Conditions, the user can choose, whether he wants to specify the ISA day or the temperature as an input. Depending on the setting of the parameter ISA, the corresponding input field is enabled.

It is also possible to enter a value directly in a field. To avoid errors, these input values are checked for example against minimum and maximum values.

Another possibility consists in the selection of an input in the context menu from a set of predefined values or replaceable models. For this a pull down menu is available. Figure 7 shows replaceable models for the solar constant, which is a function of altitude and depends on the different climatic zones. The user can choose whether the calculation shall run with a solar constant for tropic, temperate or for north temperatures.

The default values are predefined values for a cruise flight under normal operation conditions, with the maximum number of passengers and non-isothermal simulation conditions.

5 System Implementation

The different aircraft systems as mentioned in chapter 2.1 are implemented in the aircraft template. It is arranged in such a way that only those system modules are visible, where the user is required to give inputs. As an example the air supply system is chosen. Figure 8 depicts the air supply system for the A340-500/600, which consists of the mixer and the cabin air supply calibration module.

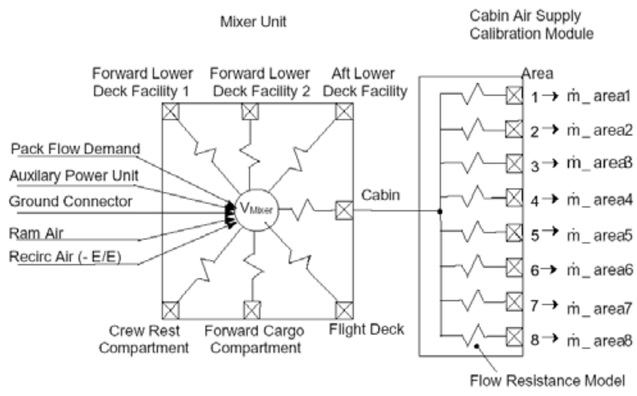


Figure 8: Air Supply System

The recirculation mass flow and the outside supply mass flow [depending on the source - ram air, APU or Engine (Pack Flow)] are input mass flows to determine over the flow resistance values the mass flows into the cabin, flight deck and forward/aft cargo compartments. The flow resistance values can be set in the parameter dialog (tab Environmental Control System).

The air supply system supplies the air to the cabin along the whole length of each zone by four air outlets per two frame bays. On the right and left hand side of the aircraft fuselage one ceiling air outlet and one lateral air outlet position are defined, see figure 9. The ceiling outlets are not affected by any installations in the cabin; however the number and position of the lateral outlets depends on the cabin interior layout.

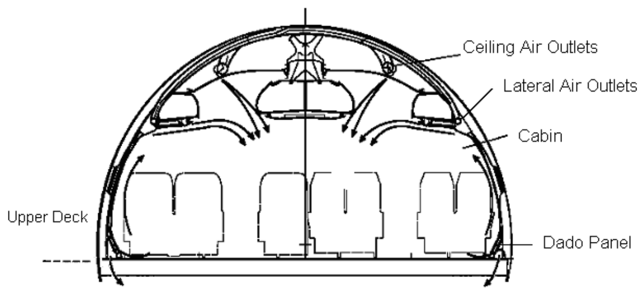


Figure 9: General Ventilation in the Cabin - Cross Section Overview

Depending on a customer layout the simulation model will be established by dragging and dropping the layout specific modules from the library into the aircraft template. The installations in the forward/aft cargo areas must be connected to the mixer unit. If a port of the mixer unit is unconnected e.g. the Lower Deck Facility (LDF) port, this means that the LDF is not installed in the cargo compartment by the user.

If a port of the mixer module is not connected the mass flow in the flow resistance model of the mixer unit shall be considered to be zero. To achieve a zero mass flow the variables at the not connected port of the mixer unit must have the same values as the variables at the port from the mixer control volume.

The modeling of such a condition in Modelica reveals to be difficult, and a complex solution has been chosen as presented hereafter.

The `DeadEndPipe`, see figure 10, represents a `pressure source` with the following Modelica code.

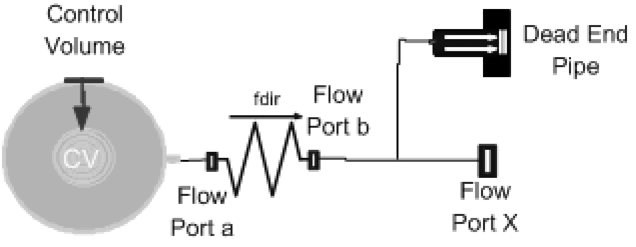


Figure 10: Unconnected Port

```
model DeadEndPipe
   extends OFFM.Icons.Flow.DeadEndPipe;
   import SI = Modelica.SIunits;
   SI.Pressure p;
   SI.Temperature T;
   SI.MassFraction x;
   parameter Boolean asSource=false;
equation
   if asSource then
        a.T = T;    a.p = p;   a.x = x;
   else
        a.H_dot  =0;   a.m_dot  =0;
        a.m_dot_x=0;
   end if;
end DeadEndPipe;
```

The number of connections of the flow port X is checked in the DeadEndPipe model by the following class parameter statement: `DeadEndPipe1 (as Source=cardinality(PortX==1)`. If there is no additional connection at the flow port X, the temperature T, pressure p and the water content x of the DeadEndPipe model are set to the values from the flow port a of the flow resistance model. From there they will be transmitted to the flow port b of the flow resistance model.

The cabin area is established with the specific cabin layout modules, such as galleys, lavatories, seat rows and special crew rest compartments. The individual supply mass flows for each cabin area is assigned by variables and not by a graphical connection. The reasons for that are the high number of connections, that have to be done by the end-user and the calculation of the mass flow supplied to each seat row as part of the total supplied airflow for one area. Each seat row module contains a specific number of air outlets, which varies from layout to layout, so the total number of air outlets in one area is depending on the number of installed seat rows within this area. The interface from the air supply system to the cabin is shown in figure 11.

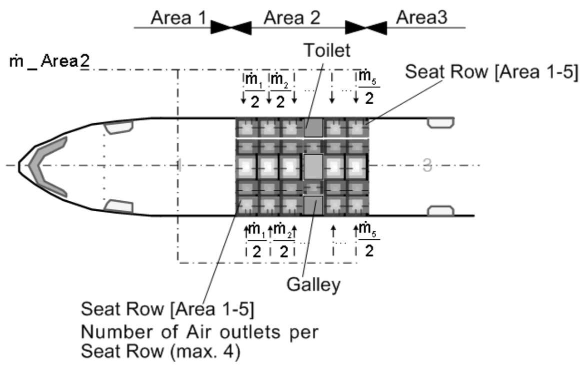


Figure 11: Interface to Cabin (I)

The implementation in Modelica proves to be quite complicated. For calculation of the total supply mass flow to an area, the number of seat rows in this area need to be considered. In a first step the number of air outlets is not considered so that the supply mass flow per seat row is assumed to be constant.

The determined total supply mass flow for each area is written in an array, in the air supply system model, as follows:

```
inner Real m_dot_supply[8]=
{m_dot[area1],
m_dot[area2], …, m_dot[area8]};
```

Then the mass flow per seat row can be calculated as follows:

```
model SeatRow
   outer Real m_dot_supply [8];
   outer Integer no_SeatRow[8];
   parameter Integer area;
   Real m_dot;
equation
   m_dot=m_dot_supply[area]/
                  no_SeatRow[area],
end SeatRow;
The mass flow m_dot must be divided and forwarded to the
two internal mass flow sources in one seat row.
```

The number of seat rows per area is currently an user input parameter in the main parameter dialog. After establishing the layout, the user must manually count the seat rows per area and the resulting number must be set in the parameter dialog. This can easily lead to the error that the entered numbers and the numbers of inserted seat rows are not equal. At the moment there is no possibility known how to count models with the same attribute, e.g. the seat rows in one area.

In order to identify the area in which a seat row is positioned, the seat row module receives the attribute `Area`, e.g. a seat row in area 2: `Area=2`. This parameter must be entered by the user in a parameter dialog, which appears by double-clicking on the component `SeatRow`. An improvement for the user is the graphical position in the diagram layer, defined by a component, which can be used to set parameters. So the parameter `Area` could be set automatically to one, two etc. if the seat row is in a defined position range on the diagram layer.

If in the second step additionally the number of air outlets shall be considered it is necessary to enter the total number of air outlets per area in the parameter dialog and additional the numbers of the individual air outlets in each seat row (double clicking on the component `SeatRow`).

The mass flow per air outlet for all seat rows in one area then can be calculated in the supply system model:

```
m_dot_perAirOutlet=m_dot_supply[area]/
no_SeatRow[area]/no_AirOutlets;
```

The mass flow for each seat row can then be determined by the multiplication of the mass flow per air outlet and the number of air outlets of each seat row.

Up to now it is insufficiently solved, how to forward the `parameter no_AirOutlets` from the Seat Row to the air supply system. The problem is that the parameter must accessed by dot-notation – ComponentName.Parameter. The automatically given component name, by dragging it into the model, is not known, alternatively if the user enters a given component name and the programmer links all, errors may occur when translating the model, if a component name of a seat row is used, where no seat row is installed, for example if a toilet or galley is inserted instead of a seat row. An ideal solution would be if Modelica language provides the possibility to access parameters or variables by a wild card as “class name.*.parameter/variable”.

Currently the found solution consists in additional modules which are implemented into the aircraft template - in this way the name of the new module is known - where the number of air outlets and the calculation of the mass flow for each seat row will be done. For the user this results in an additional expense of establishing connections. The additional modules are shown in figure 12.

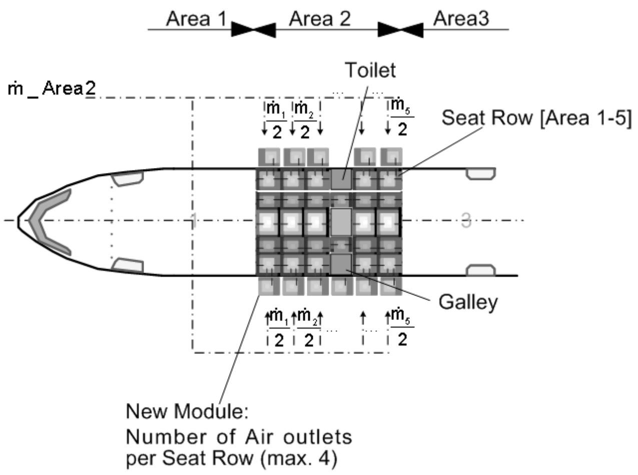


Figure 12: Interface to Cabin (II)

6 Model Application

6.1 Typical Application Procedure

The `package` Aircraft Model is the basis for the user. It contains for each aircraft type a specific library and the corresponding aircraft template, see figure 13. For the user only the layout specific components and the aircraft templates are of interest. For investigation of a new aircraft layout the user can select the template of the desired aircraft type out of the library.

The layout specific components need to be implemented into the aircraft template by the user, corresponding to a specific customer layout. For each dropped layout specific module in the cabin area the user has to set the following parameters: Area, in which the module is located, number of passengers in the module and the number of air outlets. The layout dependent connections to the standard fuselage modules, the connections between the layout specific modules and the connections to the system modules need to be established.

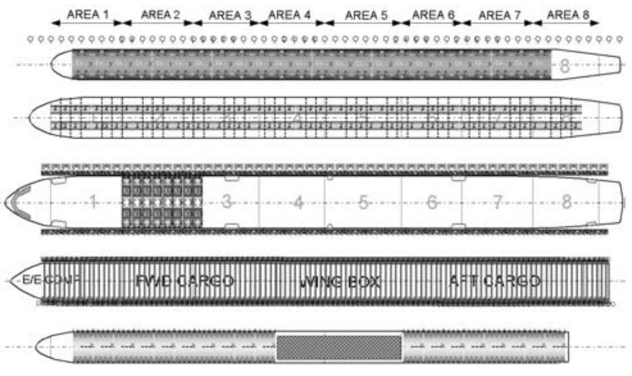


Figure 13: Aircraft Template with Cabin Layout Modules

Afterwards the main parameter dialog has to be filled out and the system configuration has to be entered and checked. Therefore the user has to work very carefully und systematically so that no inputs will be forgotten.

If this procedure has been done the user can check the model and run a simulation. The post processing is done in Matlab.

6.2 Improvement Potential

The capability of the tool Dymola/Modelica to simulate even complex systems is given. In section 4 and 5 some examples for improvement are already given.

In general there are the following suggestions for improvements based on the experience gained by developing this model.

1) Read Only
In this application case a large number of packages and sub-packages in the library shall be protected against changes by the users. They shall not be hidden in order to allow transparency and completeness, so ideally it would be possible to declare parts of the code as 'read only'.

2) Auto Connect
Another point is the number of connections that have to be done by the end-user. For each seat row the user has to make several connections. The possibility of auto connecting would make sense. An idea was to insert a new variable AutoConnect and the demand to specify each seat row with a predefined name. All connection statements are now written in the source code, inside an `if`-clause, which shall be valid if the AutoConnect is set to true. This approach is only possible if all seat rows are inserted, for which the `connect` statements are defined. Even if in one case instead of a seat row, a toilet/galley combination is inserted, and although the AutoConnect is set to false, when translating the model an error message appears. The reason for that is, that the compiler recognizes the `connect` statement in the `if`-clause.

3) Check-Possibility
Finally the need for a check-possibility shall be pointed out. There are several cases, where the user is required to give input values in the parameter dialog e.g. for the calculation of the supplied mass flow the parameter total `no_ofPassengers` is needed. Also when dragging a seat row in the aircraft template, for each seat row the number of passengers to consider the heat dissipation. There is at the moment no possibility to check at the end, if the values of total number of passengers (parameter dialog) is identical to the sum of passenger declared for each seat row by the user.

6.3 Model Numerical Characteristics

Several models of the aircraft fuselage with a different level of discretization have been established, to run simulations using the Dassl-Solver. The least coarse level of discretization without occurring numerical problems was the discretization of the cabin in its eight areas.

The first approach for a more detailed discretization was a cross section of the fuselage over two seat rows. Depending on the simulation time this results to numerical oscillations or it is not possible to reach steady state results. One of the reasons is that the system is stiff, due to the large difference between time constants.

Since the Dassl-Solver seems not to be able to solve such a system, first attempts were made with the RadauIIa-Solver out of Godess (Generic Ordinary Differential Equation Solver System), which is leading to an improvement.

7 Conclusions & Future Work

A short overview of the aircraft systems and the concept of the model and its library are given in this paper.

The emphasis of this paper is to demonstrate the challenges of providing a complex, user-friendly simulation model and potential solutions.

The additional requirement of providing a template to an end-user for his applications causes problems as described in section 6.2. The difficulty appears due to the fact, that the programmer does not know, which kind of library component the end-user needs at which place in the provided template and that the activities of the end-user shall be reduced to a minimum.

The development of the library and the models is an on-going process. It is planned to extend the library to consider effects for advanced modeling, such as the consideration of a human body with environment dependent heat release rather then as a constant heat source or the consideration of radiation from surfaces.

One of the next steps of modeling is to establish the complete aircraft based on a customer layout. Up to now there are several components and partial fuselage models, which have been verified. The whole development of the aircraft simulation model is subject of the validation and verification process.

References

[1] Tummescheit H., Eborn J: ThermoFluid Modelica Library http://www.control.lth.se/hubertus/~ThermoFluid, 2001

[2] Verein Deutscher Ingenieure: VDI Wärmeatlas. VDI-Verlag, Düsseldorf, 7th Edition, 1994

[3] Baehr, H.D.: Thermodynamik. 11th Edition, Springer Verlag, Berlin, 2004

[4] Modelica Language Specification, Version 2.1, http://www.modelica.org

Session 6c

Modelica Language

Modelica Instance Creation

Jakob Mauss
jakob.mauss@dcx.com
DaimlerChrysler Research & Technology
Alt-Moabit 96a, 10559 Berlin, Germany

Abstract

This paper is about instance creation in Modelica. Despite the conceptual simplicity of Modelica's object-oriented framework, instance creation in Modelica requires surprisingly complicated procedures. Hence, it takes considerable effort to develop a Modelica processor for extracting all variables, equations and algorithms from a given Modelica class. This paper is meant to reduce this effort by presenting key representations and algorithms for instance creation. To ease reading and verification, instance creation is developed for a fragment of Modelica, called SimpleModelica. Building on the representations and procedures given here, the implementation of instance creation (flattening) for full Modelica should be straightforward. However, that ultimate procedure is not given here, since it is loaded with technical details, described in the (100 pages) Modelica language specification.

1 SimpleModelica

The syntax of SimpleModelica is defined as follows

```
class_definition : [ encapsulated ] class IDENT class_specifier
class_specifier :  { element ";" } end IDENT |
                   "=" name [ class_modification ]
element : import_clause | extends_clause |
                   class_definition | component_declaration
import_clause :  import ( IDENT "=" name | name ["." "*"] )
extends_clause : extends name [ class_modification ]
component_declaration : name IDENT [ modification ]
modification : class_modification [ "=" expression ] | "=" expression
class_modification : "(" [ argument { "," argument } ] ")"
argument : element_modification | element_redeclaration
element_modification : name [ modification ]
element_redeclaration : redeclare
                   (class_definition | component_declaration)
expression : NUMBER | STRING | true | false | name
name : IDENT [ "." name ]
```

As usual, [x] stands for zero or one, and {x} for zero or more occurences of x, while | denotes alternatives.

SimpleModelica is a proper subset of Modelica. Omissions w.r.t. Modelica are: arrays, most prefixes, equation and algorithm section, class categories, expressions involving functions, comments, and annotations. An example of SimpleModelica is

```
class Ele1000 = Ele(Resistor.r=1000);
class Ele
   class Resistor
      Real r = 1;
   end Resistor;

   class Circuit
      Resistor r1;
      Ele.Resistor r2;
   end Circuit;
end Ele;
```

These class definitions will be used throughout the paper as illustrating example.

2 Representations

The procedure for instance creation operates on data structures as defined by the UML diagram shown in Fig 1. An instance of the shown classes is called *term* here, while an instance of a Modelica class is simply called *instance*.

2.1 Abstract syntax tree

A Modelica parser may use the classes shown in double-framed boxes to create an abstract syntax tree (AST) from a given SimpleModelica class definition. Fig 2 shows the AST that such a parser creates for the class definitions Ele and Ele1000 given above.

In the algorithm for instance creation, the Modelica class tree is represented by the constant `ROOT`, which is a ClassDefinition with no id and no parent that contains all top-level class definitions (typically packages).

Moreover, the constant `GLOBAL` is a ClassDefinition with no id and no parent that contains following built-in ClassDefinitions

(1) primitive types RealType, IntegerType, StringType, BooleanType, e.g.

```
class RealType
end RealType;
class StringType
end StringType;
```

(2) predefined types defined using these primitive types, e.g.

```
class Real
  RealType value;//accessed without dot-notation
  StringType unit;
  RealType min;
  RealType max;
end Real;
```

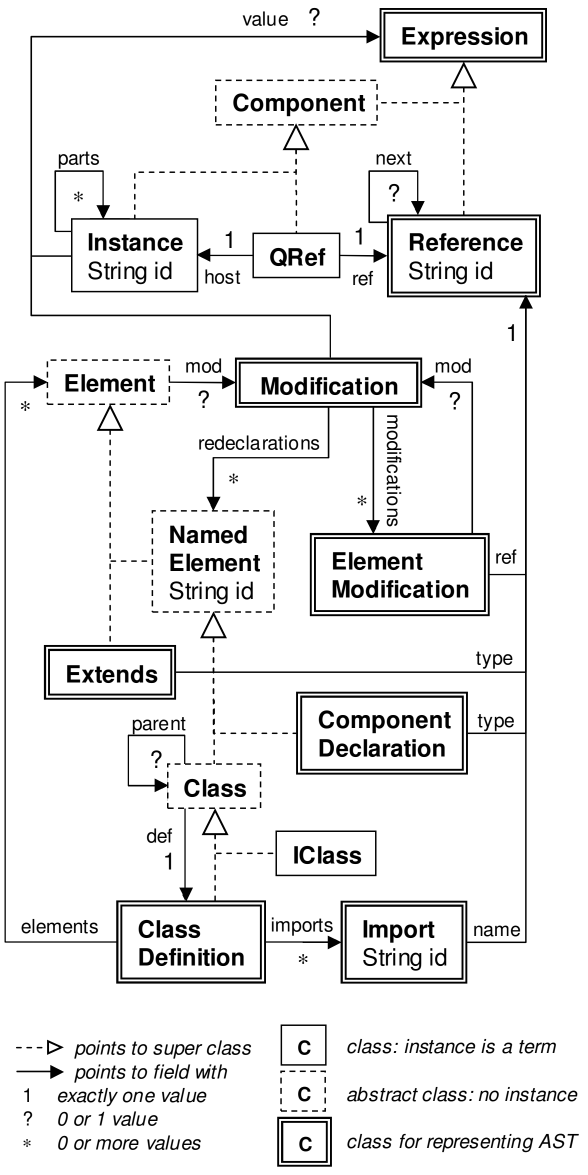


Fig 1: Classes used to implement instance creation

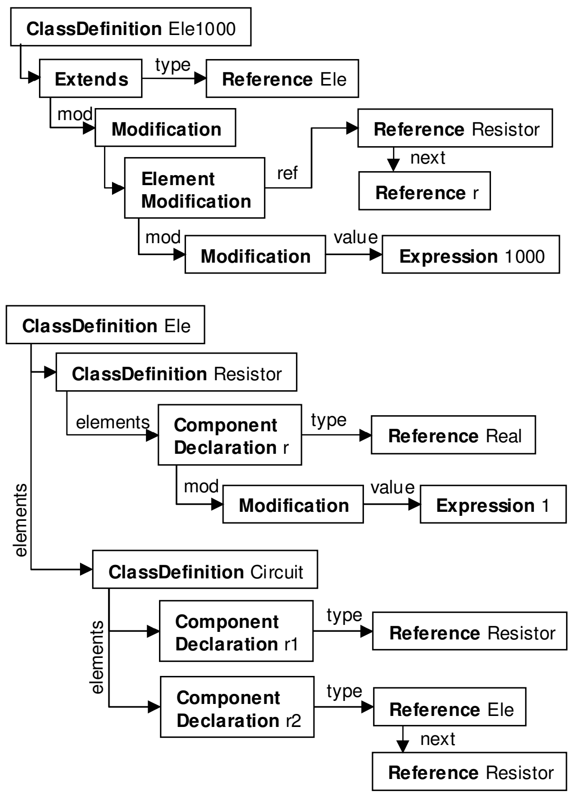


Fig 2: AST for classes Ele1000 and Ele

2.2 Implicit classes: IClass

An *implicit class* is a class that has no explicit definition in the Modelica class tree. In the example above, class Ele1000.Circuit is implicit, because the AST of Ele1000 does not contain a ClassDefinition named Circuit. Class Circuit is only inherited to Ele1000 through its base class Ele. Nevertheless, class Ele1000.Circuit can be instantiated, and the resulting instance differs from the result of instantiating class Ele.Circuit: Ele1000.Circuit.r1.r = 1000 while Ele.Circuit.r1.r = 1.

In other words: Ele1000.Circuit and Ele.Circuit are two different classes. In general, a class A may modify its base classes, which may modify all elements inherited by A from these base classes, including inherited classes.

To deal with implicit classes, a Modelica class is represented here by a tuple <ClassDefinition def, Modification mod, Class parent>, see Class in Fig 1. A ClassDefinition def from the Modelica class tree is complemented with a Modification mod that modifies this definition and a parent class that overrides the definition's parent. Subclasses of Class are ClassDefinition (for which mod is always null, and def refers to the ClassDefinition itself) to represent classes from the Modelica class tree, and IClass to represent implicit classes. The example Modelica classes are hence represented by the following terms

Modelica class	term is a	def, mod, parent
Ele	ClassDef.	Ele, null, ROOT
Ele.Resistor	ClassDef.	Ele.Resistor, null, Ele
Ele.Circuit	ClassDef.	Ele.Circuit, null, Ele
Ele1000	ClassDef.	Ele1000, null, ROOT
Ele1000.Resistor	IClass	Ele.Resistor, (r =1000), Ele1000
Ele1000.Circuit	IClass	Ele.Circuit, null, Ele1000

A ClassDefinition is created by the Modelica parser, while an IClass is created by the procedure for class name lookup during instance creation.

2.3 Qualified references: QRef

Modelica supports a *use before declare* policy for the components of a class. Moreover, there may be cyclic dependencies between components: to instantiate component a, we may need access to component b, and vice versa. This raises the question what kind of object the lookup of a component reference should return during instance creation. The referenced component (an instance) may not yet exist. To cope with this, we decide that lookup of a component may also return a pair <host, ref> where host is an Instance, and ref is a Reference, such that ref.id is the id of an instance in host.parts. This pair is called *qualified reference*, QRef for short. A QRef asserts that the host contains - or will contain - the referenced instance, and it represents this referenced instance. The host may still be under construction at the time the QRef is created, i.e. may not yet actually contain the referenced instance. Using QRefs, we can represent an instance before it actually exists. In contrast to a Reference, the meaning of a QRef does not depend on its context, but only on the host. This is why QRef s are called 'qualified'.

2.4 Qualified and unqualified modifications

A Modification is called *qualified*, if all its References have been replaced by QRefs. A qualified modification does not depend on its context, because all references have been looked up in some scope. In other words, the meaning of an unqualified modification depends on its context, while the meaning of a qualified modification does not.

ComponentDeclaration and Extends have unqualified modifications, while the modification of IClass is qualified or null, and the modification of ClassDefinition is always null. Example

```
class P
  class Ele2000 =Ele(Resistor.r=r2k);
  Real r2k(unit="Ohm")=2000;
end P
```

The extends clause `"=Ele(Resistor.r=r2k)"` of the short class definition Ele2000 is not qualified, i.e. the Modelica parser returns an Extends that contains the Reference "r2k".

The same holds for Modifications occurring in component declarations, such as `(unit="Ohm")=2000.`

In contrast, the Class returned by class lookup of P.Ele2000.Resistor is an IClass with parent P.Ele2000 and with the qualified modification (r = QRef(host = x, ref = "r2k")), where x is an instance of class P. Recall that a short class definition such as P.Ele2000 does not define its own scope, and hence "r2k" has to be looked up in the scope of P.

2.5 Instances

The objective of instance creation is to derive an object - called instance - that contains all inherited and locally declared components of a given Modelica class, with all occurring modifications applied. Fig 3 shows an instance of class Ele1000.Circuit.

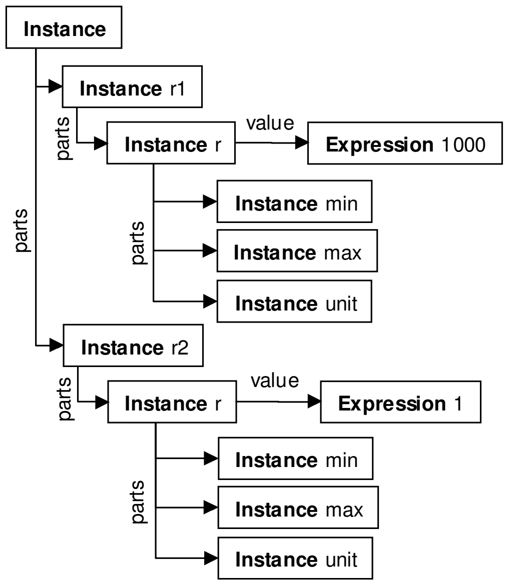


Fig 3: Instance of Modelica class Ele1000.Circuit

3 Algorithm for Instance Creation

Informally, instance creation as implemented below proceeds as follows

- take the term representing the class to be instantiated, e.g. class definition Ele.Circuit shown in Fig 2,
- replace each extends clause found in the class by the parts of an instance of the specified base class,
- replace each component declaration found in the class by an instance of the component type.
- The term resulting from all these replacements is called instance, see e.g. Fig 3.

In this context, a key algorithm is the procedure to lookup (resolve) a reference such as 'Resistor'. Lookup retrieves or computes the referenced component (a named instance) or class.

3.1 Name lookup

The following two procedures show how lookup works in principle, roughly on the level of detail as in the Modelica specification [1]. However, the procedure for instance creation given in section 3.2 uses an extended version, that (1) collects and merges class modifications encountered during lookup to support implicit classes (IClass) and (2) returns a Component (QRef) instead of a ComponentDeclaration, in case the reference refers to a component. Unfortunately, these extensions make the code for lookup less easy to understand. Therefore we also present a simplified version of name lookup here.

lookupName(Class c, Reference ref, Boolean isFirst) → NamedElement

```
1.  x ← lookupIdent(c, ref.id)
2.  if (x = null and isFirst)
3.    x ← import(c.def, ref.id)
4.  end if
5.  if (x ≠ null)
6.    if (ref.next = null) return x
7.    else if (x isa Class)
8.      return lookupName(x, ref.next, false)
9.    else
10.     assert x isa ComponentDeclaration
11.     xc ← the ClassDefinition that contains x
12.     type ← lookupName(xc, x.type, true)
13.     return lookupName(type, ref.next, false)
14.   end if
15. else if (isFirst)
16.     if (c.def is encapsulated or c = ROOT)
17.       return lookupName(GLOBAL, ref, true)
18.     else if (c.parent ≠ null)
19.       return lookupName(c.parent, ref, true)
20.     end if
21. end if
22. error "ref not found"
```

This procedure looks up the given reference in the scope of the given class, and either returns the first named element (class definition or component declaration) found, or signals a "ref not found" error.

The procedure searches the sequence of parents, until an encapsulated class or the unnamed root class of the Modelica class tree is reached. In both cases, search is continued (line 17) in the global scope that contains the predefined elements, such as Real, String, and time. Only the first identifier (as indicated by the isFirst argument) of a name is looked up using the import clauses of the class, see lines 2, 3, 4. E.g. when looking up reference A.B in the scope of class C, then A may be imported by C, but import clauses of A are ignored when looking for B in the scope of A in line 8.

lookupIdent(Class c, String id) → NamedElement

```
1.  if (c.def.elements contains
2.    a NamedElement e with e.id = id)
3.    return e
4.  else
5.    for each Extends ext in c.def.elements
6.      if (ext.type.id = id) return null
7.      end if
8.      base ← lookupName(c, ext.type, true)
9.      e ← lookupIdent(base, id)
10.     if (e ≠ null) return e
11.     end if
12.   end for
13.   return null
14. end if
```

Searches class c and its base classes for a named element with the given id. If id names a local or inherited named element of c, returns that element, returns null otherwise. This search does neither use imports nor parent classes. A tricky part of the algorithm is the test in line 6, which terminates a circular attempt to lookup a base class of c.

3.2 Instance creation for SimpleModelica

This section presents a set of procedures that implement instance creation for SimpleModelica.

instantiate(Reference name) → Instance

```
1.  c ← lookupClass(ROOT, name, null)
2.  ic ← elaborate(c, new Instance())
3.  ic ← replaceQRefs(ic)
4.  ic ← removeDuplicates(ic)
5.  return ic
```

This procedure instantiates the given class and returns the resulting instance. The class is specified by name. This way, also an implicit class (such as Ele1000.Circuit) can be instantiated. In line 1., the class name is looked up in the scope of the unnamed root of the Modelica class tree. In line 2., the entire instance tree is created. However, component references occurring in modifications are replaced by QRefs, not by the referenced instances, see section 2.3. In line 3., ic is the root of the completed instance tree. All referenced instances should have been created by then. Consequently, all QRefs can now be replaced by the referenced instances. In line 4., duplicate instances added through multiple inheritance are removed from the instance.

elaborate(Class c, Instance host) → Instance

```
1.  for each Extends ext in c.def.elements
2.    base ← getClass(c, ext, c.mod, host)
3.    host ← elaborate(base, host)
4.  end for
5.  for each ComponentDeclaration decl
6.  in c.def.elements
7.    if (redeclare(c.mod, decl.id) ≠ null)
8.      decl ← redeclare(c.mod, decl.id)
9.    end if
10.   qmod ← select(c.mod, decl.id)
11.   type ← getClass(c, decl, qmod, host)
12.   comp ← elaborate(type, new Instance())
13.   comp.id ← decl.id
14.   add comp to host.parts
15. end for
16. host.value ← c.mod.value
17. return host
```

This procedure adds for each inherited or locally declared component of the given class c an elaborated instance to the given host. In an elaborated instance, each component reference is represented by a QRef (see section 2.3), but not yet by the referenced instance. During elaboration, modifications are merged in the correct order. Redeclaration of a component is processed in lines 7 - 9.

getClass(Class c, Element e, Modification qm, Instance host) → Class

```
1.  type ← lookupClass(c, e.type, host)
2.  qmod ← qualify(c, e.mod, host)
3.  qmod ← merge(qm, qmod)
4.  return createClass(type, qmod)
```

This auxilliary procedure returns a base class (if e is an Extends) or component type (if e is a ComponentDeclaration) used during elaboration or lookup.

createClass(Class c, Modification qmod) → Class

```
1.  qmod ← merge(qmod, c.mod)
2.  if (qmod = c.mod) return c
3.  else return new IClass(c.def, qmod, c.parent)
4.  end if
```

This auxilliary procedure merges the qualified modification of the given class with the given qualified modification qmod, where elements of qmod override elements of c.mod. Returns either the given class (e.g. if qmod = null), or a new IClass (see 2.2).

lookupClass(Class c, Reference name, Instance host) → Class

```
1.  x ← lookup(c, name, true, host)
2.  if (x isa Class) return x
3.  else error "not a class"
4.  end if
```

Look for the given class name in the scope of class c. The host is either null, or an elaborated instance of c. The given host may be under construction, i.e. not yet completely elaborated. If lookup should require to instantiate c (e.g. to access a component of c that occurs in a modification, see r2k in the example in section 2.4) the host, if given, is used. Otherwise, c is elaborated on demand.

lookup(Class c, Reference ref, Boolean isFirst, Instance host) → Class or Component

```
1.  x ← lookup(c, ref.id, host)
2.  if (x = null and isFirst)
3.    x ← import(c.def, ref.id)
4.  end if
5.  if (x ≠ null)
6.    if (ref.next = null) return x
7.    else if (x isa Class)
8.      return lookup(x, ref.next, false, null)
9.    else
10.     assert x isa QRef
11.     return new QRef(x.host, ref)
12.   end if
13. else if (isFirst)
14.     if (c.def is encapsulated or c = ROOT)
15.       return lookup(GLOBAL, ref, true, null)
16.     else if (c.parent ≠ null)
17.       return lookup(c.parent, ref, true, null)
18.     end if
19. end if
20. error "ref not found"
```

Similar to procedure lookupName, defined in section 3.1. However, if ref names a component, the component is returned (as QRef), not its declaration.

Again, if isFirst is false, then only locally declared or inherited elements are found, i.e. import clauses, the parent of c as well as the global scope are ignored.

The next procedure looks up in class c for a local, inherited or redeclared class or component with the given id. Returns null, if no such class or component is found. This does not use imports or c's parent. If id names a component, the component is returned (represented by a QRef), and not (like procedure lookupIdent defined in section 3.1) the corresponding component declaration.

Lines 5 and 25 handle the case that id names a class that is redeclared by the qualified modification c.mod. If id names a component which is redeclared by c.mod, this redeclaration is either treated during elaboration of c in line 11, or in lines 19-20 in case the component is inherited from a base.

If id names a class inherited to c, then c becomes the new parent of this class (line 24).

lookup(Class c, String id, Instance host) → Class or Component

```
1.  if (c.def.elements contains
2.       a NamedElement e with e.id=id)
3.    if (e isa ClassDefinition)
4.      if (redeclare(c.mod, id) ≠ null)
5.        e ← redeclare(c.mod, id)
6.      end if
7.      return createClass(e, select(c.mod, id))
8.    else
9.      assert e isa ComponentDeclaration
10.     if (host = null)
11.       host ← elaborate(c, new Instance())
12.     end if
13.     return new QRef(host, new Reference(id))
14.   end if
15. else
16.   for each Extends ext in c.def.elements
17.     if (ext.type.id=id) return null
18.     end if
19.     base ← getClass(c, ext, c.mod, host)
20.     e ← lookup(base, id, host)
21.     if (e ≠ null)
22.       if (e isa Class)
23.         if (redeclare(c.mod, id) = null)
24.           e ← new IClass(e.def, e.mod, c)
25.         else e ← redeclare(c.mod, id)
26.         end if
27.         return createClass(e, select(c.mod, id))
28.       else return e
29.       end if
30.     end if
31.   end for
32.   return null
33. end if
```

A tricky part of this procedure is the test in line 17 which terminates a circular attempt to lookup a base class of c.

import(ClassDefinition c, String id) → Class or Component

```
1.  for each Import imp in c.imports
2.    if (imp matches "import A.B.C" and id="C")
3.     return lookup(ROOT, "A.B.C", true, null) else
4.    if (imp matches "import C = A.B" and id="C")
5.     return lookup(ROOT, "A.B", true, null) else
6.    if (imp matches "import A.B.*")
7.      ab ← lookupClass(ROOT, "A.B", null)
8.      e ← lookup(ab, id, null)
9.      if (e ≠ null) return e
10.     end if
11.   end if
12. end for
13. return null
```

Searches the import clauses of the given class definition for a named element with the given id. Returns the first matching class or component, or null if no match was found.

removeDuplicates(Instance host) → Instance

Remove all duplicate instances (instances with same id) from the given instance and return the resulting instance. Duplicates are caused by multiple inheritance. It is an error if two duplicate elements are not equivalent. Example:

```
class A Real x = 1; end A;
class B Real x = 2; end B;
class C
  extends A;
  extends B; // error
end C;
class D
  extends A;
  extends B(x = 1); // ok
end D;
```

Before application of removeDuplicates, instances of C and D contain a duplicate component x. The procedure removes x from D, but signals an error for C, because components x = 1 and x = 2 are not equivalent.

qualify(Class c, Modification mod, Instance host) → Modification

Lookup each Reference contained in the given unqualified modification in the scope of the given class, replace it with the resulting Class or QRef, and return the resulting qualified modification. See 2.4. The given host is either null or, if available, an elaborated instance of class c to be used as argument for name lookup in the scope of class c.

replaceQRefs(Instance host) → Instance

Replace each QRef contained in the given instance by the referenced Instance and return the resulting instance. It is an error if an instance referenced by a QRef is not found in the QRef's host. (For unrestricted Modelica, this method also performs dynamic lookup of the `inner` component, in case a QRef references an `outer` component.)

redeclare(Modification env, String id) → NamedElement

```
1. if (env.redeclarations contains x with x.id = id)
2.   return x
3. else return null
4. end if
```

If the given modification redeclares an element with the given id, return the element. In the AST generated by a parser, the parent of a redeclared class is the class that contains the redeclaration.

Example:

`class A = B(redeclare class C = D);`

In the corresponding AST, the parent of C is A.

select(Modification env, String id) → Modification

```
1.  if (env.modifications contains x with x.ref.id = id)
2.    if (x.ref.next = null)
3.      return x.mod
4.    else
5.      mod ← new Modification()
6.      em ← new
7.        ElementModification(x.ref.next, x.mod))
8.      add em to mod.modifications
9.      return mod
10.   end if
11. else return null
12. end if
```

If the given modification modifies an element with the given id, this procedure returns the corresponding modification. Otherwise returns null.

Examples:

- select((R.r =10), "r") returns null
- select((R.r =10), "R") returns (r =10)
- select((r =10), "r") returns =10
- select((r), "r") returns null

merge(Modification env, Modification mod) → Modification

```
1.  if (env=null) return mod
2.  else if (mod=null) return env
3.  else
4.    result ← copy of env
5.    for each ElementModification em
6.    in mod.modifications
7.      if (select(env, em.ref.id) = null)
8.        add em to result.modifications
9.      end if
10.   end for
11.   for each NamedElement e
12.   in mod.redeclarations
13.     if (redeclare(env, e.id) = null)
14.       add e to result.redeclarations
15.     end if
16.   end for
17.   if (env.value = null)
18.     result.value ← mod.value
19.   end if
20.   return result
21. end if
```

Merge the given modifications, where elements in env beat (override, replace, update) elements in mod, and return the resulting merged modification.

The merge operation is associative, not commutative, and merge(null, m) = merge(m, null) = m for every Modification m. See [1] for a more detailed specification of the merge operation.

Examples:

- merge((x=1,y=2), (x(min=6)=3, z=4)=5) returns the modification (x=1, y=2, z=4)=5
- merge((x), (x=1)) returns (x=1)

3.3 Extension to Arrays

SimpleModelica can be extended to arrays by adding (updating resp.) the follwing syntactic definitions.

```
class_specifier : { element ";" } end IDENT |
                  "=" name [ subscripts ] [ class_modification ]
component_declaration : name [ subscripts ] IDENT [ modification ]
element_modification : [ each ] reference [ modification ]
expression : NUMBER | STRING | true | false | reference |
                  "{" expression { "," expression } "}"
reference : IDENT [ subscripts ] [ "." reference ]
subscripts : "[" (":" | expression) { "," (":" | expression) } "]"
```

Example:

```
class P =Real[2](unit={"x","y"});
class A
  Real[n,n+1] a;
  Real[:] b(each min=1)={2,n,4};
  Integer n = 3;
end A
```

A challenge introduced by arrays is the need to evaluate expressions during instance creation.

- The parameter expressions that specify array size must be evaluated, and return positive integer sizes. E.g. to instantiate component a in class A, expressions n and n+1 must be evaluated.
- The modifier of an array must be split in order to get one single modifier for each array element. E.g., to instantiate b in class A, b's modification is split into three modifiers (min=1)=2, (min=1)=n, and min(=1)=4.

To represent arrays, we use a new class Array, which extends Instance (see Fig 1) and defines the fields elementType, subs, and mod where

- elementType is a Class
- subs is a qualified subscripts expression defining the array size, e.g. [3, 4]
- mod is an optional qualified array modification, e.g. (unit = {"r", "g", "b"}) = {1, 2, 3}.

Expansion of arrays is delayed until the class being instantiated has been elaborated and hence array size expressions can be evaluated. After expansion of an array a, the field a.parts (inherited to Array from Instance) contains the array elements.

A notable feature introduced by arrays are component references that cannot be resolved to a unique component during instance creation.

Example:

```
Real a[:] = {10, 20};
Real b = a[if (time<1) then 1 else 2];
```

The value of b cannot be identified with a unique array element during instance creation.

The following procedures extend instance creation as presented so far to arrays.

instantiate(Reference name) → Instance

```
1. c ← lookupClass(ROOT, name, null)
2. ic ← elaborate(c, new Instance())
3. ic ← expandArrays(ic)
4. ic ← replaceQRefs(ic)
5. ic ← removeDuplicates(ic)
6. return ic
```

The only difference to the procedure given in 3.2 is the inserted line 3, which expands arrays contained in ic by evaluating array size expressions and creating and inserting the corresponding array elements.

elaborate(Class c, Instance host) → Instance

```
1. for each Extends ext in c.def.elements
2.   if (ext.subs ≠ null)
3.     return createArray(c, ext, c.mod, host)
4.   else
5.     base ← getClass(c, ext, c.mod, host)
6.     host ← elaborate(base, host)
7.   end if
8. end for
9. for each ComponentDeclaration decl
10. in c.def.elements
11.   if (redeclare(c.mod, decl.id) ≠ null)
12.     decl ← redeclare(c.mod, decl.id)
13.   end if
14.   qmod ← select(c.mod, decl.id)
15.   if (decl.subs ≠ null)
16.     comp ← createArray(c, decl, qmod, host)
17.   else
18.     type ← getClass(c, decl, qmod, host)
19.     comp ← elaborate(type, new Instance())
20.   end if
21.   comp.id ← decl.id
22.   add comp to host.parts
23. end for
24. host.value ← c.mod.value
25. return host
```

The only difference to the elaboration procedure given in 3.2 are lines 2-5 and 15-17, which treat the case that a short class definition or component declaration contains subscripts.

createArray(Class c, Element e, Modification qm, Instance host) → Array

```
1. elementType ← lookupClass(c, e.type, host)
2. qsubs ← qualify(c, e.subs, host)
3. qmod ← merge(qm, qualify(c, e.mod, host))
4. return new Array(elementType, qsubs, qmod)
```

This auxilliary procedure creates an array with the given fields. The returned array is not yet expanded, but it represents (is equivalent to) an expanded array.

expandArrays(Instance host) → Instance

```
1. if (host isa Array)
2.   expr ← left-most expression in host.subs
3.   next ← host.subs without left-most expression
4.   n ← vectorSize(expr, host.mod)
5.   for i in 1 to n
6.     modi ← split(host.mod, i, n)
7.     if (next = null)
8.       ci ← createClass(host.c, modi)
9.       xi ← elaborate(ci, new Instance())
10.    else
11.      xi ← new Array(ci, next, modi)
12.    end if
13.    add expandArrays(xi) to host.parts
14.  end for
15. else
16.   for each Instance comp in host.parts
17.     replace comp by expandArrays(comp)
18.   end for
19. end if
20. return host
```

This procedure expands all arrays contained in the given host, and returns the resulting expanded instance.

split(Modification qmod, Integer i, Integer n) → Modification

Splits the given qualified modification into n parts and returns the ith part of it. Example:

- split((each unit="V"), 7, 10) returns (unit="V")
- split(={x, x+y, y}, 2, 3) returns = x+y

Note that Modelica arrays have a 1-based index, i.e. the first array index is 1 and not 0.

vectorSize(Expression qexpr, Modification qmod) → Integer

Determine vector size based on the given qualified integer-valued expression and the given qualified vector modification. This requires evaluation of qexpr, as well as evaluation of expressions occuring in qmod. Returns a positive integer, or signals an error. The following examples assume that parameter n evaluates to 3

- vectorSize(2, null) returns 2
- vectorSize(n, null) returns 3
- vectorSize(:, { x }) returns 1 without eval of x
- vectorSize(:, null) signals an error

3.4 Extension to unrestricted Modelica

To extend instance creation to full Modelica, the following remains to be done

- match **outer** references with the corresponding **inner** reference in the instance tree

- add qualified **equations** and **algorithms** to an instance, expand **for** clauses in equation sections and **connect** predicates i.e. generate the corresponding equations
- **validate** semantic constraints, e.g. (1) assertions associated with a **class category**, (2) **type constraints** of the Modelica type system, (3) **constraining clause** for redeclaration, (4) restrict modification to **public** and non-**final** elements of a class

The extension of the above algorithms to most of these features should be straightforward.

4 Application

Modelica's object-oriented approach to modeling opens new ways for systematically validating (models of) engineered systems, e.g. with respect to behavior in the presence of component faults and for alternative input scenarios. We have implemented instance creation as presented here for a large fragment of Modelica. This Java implementation (based on JavaCC, see [5]) is part of a bigger effort to develop a tool for automated simulation which, in a nutshell,

- instantiates a given annotated Modelica model
- extracts for each component of the system the corresponding fault modes (e.g. ok, stuckOpen, stuckClose) as indicated by specific annotations
- extracts information about the intended use of the system (inputs of the model)
- extracts information about specified, i.e. desired behavior of the system
- uses the extracted information to autonomously drive a large number of simulation runs, during which component faults are dynamically inserted and resulting behavior is classified w.r.t. specified behavior. The executable used for simulation is generated by Dymola.

Modelica applications like this one require access to instantiated Modelica classes. Modelica simulators such as Dymola do not currently offer an API to access instances, which currently forces application developers to implement instance creation on their own. This paper may help to reduce the required effort in the future.

5 Related Work

The *Modelica specification* [1] is available for free from www.modelica.org. The specification states that it defines the *static semantics* of Modelica in terms of a procedure for instance creation. Unfortunately, this is done in a quite informal way. No pseudo code is given, and no auxiliary representations (such as Instance, QRef, IClass) are explicitly defined. It would be helpful to complement the specification with a precise procedural definition of instance creation in the future. This paper may be a starting point.

Pelab at Linköping University has developed a *RML specification* [3] for a fragment of Modelica, which can be used to automatically generate [4] a procedure for instance creation. However, for a human reader interested in a procedural view on Modelica, e.g. to understand name lookup or to implement flattening, the RML specification (several thousand lines of declarative rules) is less helpful.

The design of Modelica was influenced by the *Theory of Objects* by Abadi and Cardelli [2]. This book defines various calculi (similar to Lambda-calculus) to model object-oriented languages. The calculi are composed from equational theories, called fragments. Based on these calculi, a procedure for instance creation in Modelica could perhaps be derived on formal grounds as follows

- define an equational theory E of objects
- direct the equations to convert E into a term rewrite system such that a term may be a Modelica class definition, and the irreducible term derived from that by a finite number or reductions is an instance of that class.

This is basically the idea underlying Pelab's RML specification. In this paper, we have chosen a less formal approach.

References

[1] Modelica Language Specification 2.1, www.modelica.org, 2004.

[2] Martin Abadi, Luca Cardelli: A Theory of Objects, Springer 1996.

[3] David Kågedal: A Natural Semantics Specification for the Equation-based Modeling Language Modelica. Master Thesis LiTH-IDA-Ex-98/48, Linköping University, 1998.

[4] David Kågedal and Peter Fritzson: Generating a Modelica compiler from natural semantics specifications. Proceedings of Summer Computer Simulation Conf. SCSC'98, 1998.

[5] JavaCC - Java Compiler Compiler, a Java Parser Generator, https://javacc.dev.java.net/

Towards Comprehensive Meta-Modeling and Meta-Programming Capabilities in Modelica

Peter Fritzson, Adrian Pop, Peter Aronsson
PELAB – Programming Environment Lab, Dept. Computer Science
Linköping University, S-581 83 Linköping, Sweden
{petfr,adrpo,petar}@ida.liu.se

Abstract

The need for integrating system modeling with tool capabilities is becoming increasingly pronounced. For example, a set of simulation experiments may give rise to new data that are used to systematically construct a series of new models, e.g. for further simulation and design optimization. Using models to construct other models is called meta-modeling or meta-programming.

In this paper we present extensions to the Modelica language for comprehensive meta-programming, involving transformations of abstract syntax tree representations of models and programs. The extensions have been implemented and used in several applications, are and currently being integrated into the OpenModelica environment.

1 Introduction

Meta-programming (meta-modeling) is writing programs (models) having other programs (so called object-programs) as inputs or results. A program can for instance take another program as input data, perform computations on the program by traversing its internal structure (the abstract syntax of the program) and return a modified program as output data.

Often, the object program language and the meta-programming language are the same, like for instance in LISP, in Mathematica, or in the Java reflection mechanism. This is also the approach we have taken for Modelica. Thus, a language needs some way of representing the object program as data.

A simple approach is to use text strings as program representation. However, this has the disadvantage that not even simple structural (syntactic) correctness can be guaranteed. Another problem is low performance. Thus, this approach is only suitable for simple and less demanding tasks.

Another solution is to encode the object program using structured data types of the meta-programming language. This basically means that data types for the abstract syntax are defined in the language itself. This approach has the benefit of ensuring correct syntax of object programs. It is used in for instance Java reflection where the class `java.lang.Class` is the data type for a Java class. The class has methods to query a Java class for its methods, members, interfaces, etc.

In a previous paper (Aronsson *et.al.*, 2003) we presented an approach of quoted Modelica code combined with built-in predefined Modelica types to handle certain syntax classes, like for instance `TypeName` for a Modelica type name or `VariableName` for a Modelica variable name. However, this does not give full flexibility and meta-programming power, since the abstract syntax tree representation cannot be fully manipulated in the meta-programming language itself. That work should be seen as a precursor and initial stage for the work presented in this paper.

2 Tree Data Structures

What are then the needs for data structures and operations for full meta-programming capabilities? One of the most common examples of programs that manipulate and produce other programs are compilers, which translate programs in some language into the same or another language.

The most common data type representation for programs in compilers are tree structures, and typical operations are transformations of such trees into trees during the translation process. Lists are a special case of tree data types, but are typically given special support in many symbolic programming languages..

Tree data types have two interesting properties:

- Union type – a tree data type is typically the union of a number of node types, each representing a tree node.
- Recursive type – the children of a tree node may a type which is the tree data type itself.

A small expression tree, of the expression 12+5*13, is depicted in Figure 1. Using the record constructors `PLUS`, `MUL`, `RCONST`, this tree can be constructed by the expression `PLUS(RCONST(12), MUL( RCONST(5), RCONST(13)))`

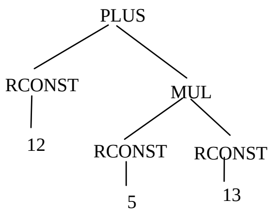


Figure 1. Abstract syntax tree of the expression 12+5*13.

Union types and recursive types are currently missing from the Modelica language, which so far has been a conscious decision in order to avoid heap-allocated objects.

However, with the increased relevance of meta-modeling, the time may now be ripe for a possible extension such as the introduction of the `uniontype` restricted class construct. The example below declares a small expression tree type `Exp` containing 6 different node types represented as ordinary Modelica record types.

```
uniontype Exp
  record RCONST Real x1; end INT;
  record PLUS  Exp x1; Exp x2; end PLUS;
  record SUB   Exp x1; Exp x2; end SUB;
  record MUL   Exp x1; Exp x2; end MUL;
  record DIV   Exp x1; Exp x2; end DIV;
  record NEG   Exp x1;         end NEG;
end Exp;
```

The `uniontype` restricted class construct currently has the following properties:

- Union types can be *recursive*, i.e., reference themselves. The is the case in the above `Exp` example, where `Exp` is referenced inside its member record types.
- Union types are currently restricted to contain only *record* types. This restriction may be removed in the future.
- Record declarations declared within a union type are automatically inherited *into the enclosing* scope of the union type declaration.
- A record type may only belong to *one* union type. This restriction may be removed in the future.

This is a preliminary union type design, which however is very close to (just different syntax) similar constructs in functional languages such as Haskell, Standard ML, OCaml, and RML.

3 Tree Transformation Operations

Regarding operations on trees, most languages supporting tree transformations provide a kind of pattern matching and transformation construct. Therefore we propose the introduction of match-expressions in the Modelica language. A trivial example of match-expression is presented below:

```
  String str;
  Real   x;
algorithm
  x :=
    match str
      case "one"   then 1;
      case "two"   then 2;
      case "three" then 3;
      else   0;
    end match;
```

The string variable `str` is matched against the constant patterns `"one"`, `"two"`, etc., returning the corresponding value from each branch in the match-expression. A default value can be returned from the optional else-branch if no other branch matches.

The general form of the proposed match-expression is as follows:

```
match <expr>  <opt-local-decl>
  case <pat-expr> <opt-local-decl>
    <opt-local-equations>
    then <value-expr>;
  case <pat-expr> <opt-local-decl>
    <opt-local-equations>
    then <value-expr>;
  ...
  else <opt-local-decl>
    <opt-local-equations>
   then <value-expr>;
end match;
```

The **`then`** keyword precedes the value to be returned in each branch.. The local declarations started by the **`local`** keyword, as well as the equations started by the **`equation`** keyword are optional.There should be at least one **`case...then`** branch, but the **`else`**-branch is optional.

The match-expression introduces several new concepts in Modelica:

- Pattern expressions, `<pat-expr>`, which may reference unbound local pattern variables declared within the match-expression.
- Optional local variable declarations, `<opt-local-decl>`. These variables are local and have a scope within the match-expression or within a specific branch of the match-expression if they are declared within such a branch.
- Optional local equations, `<opt-local-equations>`, which are solved locally within the match-expression, and where the unbound unknowns to be solved for have been declared in local variable declarations.

An example of a match-expression within the function `eval` shows its usage in a simple expression tree evaluator. The local variables `v1`,`v2`,`e1`,`e2` have scope throughout the whole match-expression. Pattern variables such as e1 and e2 are belong to pattern expressions that are matched against tree expressions. For example, `PLUS(e1,e2)` is matched against `PLUS(RCONST(12), MUL( RCONST(5), RCONST(13)))` depicted in Figure 1, thereby binding `e1` and `e2` to the children of the `PLUS` node, in this match e1 to `RCONST(12)` and `e2` to `MUL( RCONST(5), RCONST(13))`.

```
function eval
  input  Exp   exp_1;
  output Real rval_1;
algorithm
 rval_1 :=
  match exp_1
    local Integer v1,v2;
          Exp     e1,e2;
    case RCONST(v1) then v1;

    case PLUS(e1,e2) equation
      v1 = eval(e1;  eval(e2) = v2;
      then v1+v2;

    case SUB(e1,e2) equation
      v1 = eval(e1);  v2 = eval(e2;
      then v1-v2;

    case MUL(e1,e2) equation
      v1 = eval(e1);  v2 = eval(e2);
      then v1*v2;

    case DIV(e1,e2) equation
      v1 = eval(e1);  v2 = eval(e2);
      then v1/v2;

    case NEG(e1) equation
      v1 = eval(e1);
      then -v1;
   end match;
end eval;
```

Note that the match-expression just like other expressions can be used in three contexts: inside equations, inside algorithm sections, and inside functions.

As usual in Modelica the equations are not directional, e.g. the two equations `v1 = eval(e1)` and `eval(e1) = v1` are equivalent.

There are some design considerations behind the above match-expression construct that may need some motivation.

- Why do we have *local variable declarations* within the match-expression? The main reason is clear and understandable semantics. In all three usage contexts (equations, algorithm sections, functions) it should be easy to understand for the user and for the compiler which variables are unknowns (i.e., unbound local variables) in pattern expressions or in local equations.

  Other variables that are bound to values might have been declared in some class, or be protected variables in a function. Without the simple rule that local unknowns must be declared locally, it would be hard to discover the difference between variables that are unknowns and still can receive values, and other variables which already have values.

  Another reason for declaring the types of local variables is better documentation of the code – the modeler/programmer is relieved of the burden of doing manual type-inference to understand the code.
- Why *local equations* instead of assignment statements? The match-expression is an expression construct that can be used in the three contexts, including expressions in equations which are declarative. Having non-local assignments inside expressions would make the expressions nondeclarative.
- Why match-expressions and not match-statements? The match-expression is more important since it can be used in all three contexts, and therefore has been designed first. An analogous match-statement without local equations can be designed at a later stage.
- Why the keywords **`match`** ... **`case`** instead of **`switch`** ... **`case`** as in Java? The current choice of keywords is inspired by the languages Modelica, Java, and Mathematica, and is just a matter of taste – it is easy to change to other keywords. However, it is probably good style to indicate the increase power of the match-expression compared to the switch-statement by a different keyword.
- Why the **`then`** keyword before the returned value? We have experimented with various syntax designs, and the code becomes easier to read if there is a keyword before the returned value-expression, especially when it is preceded by local equations. The keyword cannot be **`return`** since that means return from a function. The **`then`** keyword is used in a similar way in Modelica **`if-then-else`** expressions. Note that most functional languages use the `in` keyword instead in this context, which is less intuitive. However, the `in` keyword has more of a set or array element membership meaning in Modelica.

Local equations in match-expressions have the following semantics:

- Only algebraic equations are allowed, no differential equations
- Only locally declared variables (local unknowns) declared by local declarations within the match-expression are solved for.

- Equations are solved one by one in the order they are declared. (This restriction may be removed in the future).
- If an equation or an expression in a case-branch fails, all local variables become unbound, and the next branch is tried. (There is some discussion whether the semantics of trying the next case-branch after a fail should be kept).

3.1 Example of Symbolic Differentiation

To make the following example of symbolic differentiation more realistic, we add a few expression nodes to the `Exp` data type, including a function call node `CALL` whose argument list has the type **`list`**`<Exp>`, see Section 4.1.

```
record IDENT String name; end IDENT;
record CALL  Exp id; list<Exp> args;
  end CALL;
record AND   Exp x1; Exp x2; end AND;
record OR    Exp x1; Exp x2; end OR;
record LESS  Exp x1; Exp x2; end LESS;
record GREATER Exp x1; Exp x2;
  end GREATER;
```

An example function `difft` performs symbolic differentiation of the expression `expr` with respect to the variable `time`, returning a differentiated expression. In the patterns, _ underscore is a reserved word that can be used as a placeholder instead of a pattern variable when the particular value in that place is not needed later as a variable value. The **`as`**-construct: `id` **`as`** `IDENT(_)` in the third of-branch is used to bind the additional identifier `id` to the relevant expression. Both tuples with syntax `(expr1,expr2,....)`, see Section 4.2, and lists are used in the example.

We can recognize the following well-known derivative rules represented in the match-expression code:

- The time-derivative of a constant (`RCONST()`) is zero.
- The time-derivative of the `time` variable is one.
- The time-derivative of a time dependent variable `id` is `der(id)`, but is zero if the variable is not time dependent, i.e., not in the list `tvars`/`timevars`.
- The time-derivative of the sum (`add(e1,e2)`) of two expressions is the sum of the expression derivatives.
- The time-derivative of `sin(x)` is `cos(x)*x'` if `x` is a function of time.
- etc...

We have excluded some operators in the `difft` example because of limitations of space in this paper.

```
function difft "Symbolic differentiation
    of expression with respect to time"
  input  Exp expr;
  input  list <IDENT> timevars;
  output Exp diffexpr;
algorithm
 diffexpr :=
  match (expr, timevars)
    local Exp e1prim,e2prim,tvars;
          Exp e1,e2,id;
 // der of constant
    case(RCONST(_), _) then RCONST(0.0);
 // der of time variable
    case(IDENT("time"), _) then
      RCONST(1.0);
 // der of any variable id
    case difft(id as IDENT(_), tvars) then
      if list_member(id,tvars) then
        CALL(IDENT("der"),list(id))
      else
        RCONST(0.0);
 // (e1+e2)' => e1'+e2'
    case (ADD(e1,e2),tvars) equation
      e1prim = difft(e1,tvars);
      e2prim = difft(e2,tvars);
      then ADD(e1prim,e2prim);
 // (e1-e2)' => e1'-e2'
    case (SUB(e1,e2),tvars) equation
      e1prim = difft(e1,tvars);
      e2prim = difft(e2,tvars);
      then SUB(e1prim,e2prim);
 // (e1*e2)' => e1'*e2 + e1*e2'
    case (MUL(e1,e2),tvars) equation
      e1prim = difft(e1,tvars);
      e2prim = difft(e2,tvars);
      then PLUS(MUL(e1prim,e2),
              MUL(e1,e2prim));
 // (e1/e2)' => (e1'*e2 - e1*e2')/e2*e2
    case (DIV(e1,e2),tvars) equation
      e1prim = difft(e1,tvars);
      e2prim = difft(e2,tvars);
      then DIV(SUB(MUL(e1prim,e2),
                MUL(e1,e2prim)),
             MUL(e2,e2));
 // (-e1)' => -e1'
    case (NEG(e1),tvars)  equation
      e1prim = difft(e1,tvars);
      then NEG(e1prim);
 // sin(e1)' => cos(e1)*e1'
   case CALL(IDENT("sin"),list(e1)),tvars)
      equation e1prim = difft(e1,tvars);
     then MUL(CALL(IDENT("cos"),list(e1)),
            e1prim);
// (e1 and e2)' => e1'and e2'
    case (AND(e1,e2),tvars) equation
      e1prim = difft(e1,tvars);
      e2prim = difft(e2,tvars);
      then AND(e1prim,e2prim);
// (e1 or e2)' => e1' or e2'
    case (OR(e1,e2),tvars) equation
      e1prim = difft(e1,tvars);
      e2prim = difft(e2,tvars);
      then OR(e1prim,e2prim);
// (e1<e2)' => e1'<e2'
    case (LESS(e1,e2),tvars) equation
      e1prim = difft(e1,tvars);
      e2prim = difft(e2,tvars);
      then LESS(e1prim,e2prim);
// (e1>e2)' => e1'>e2'
    case (GREATER(e1,e2),tvars) equation
      e1prim = difft(e1,tvars);
      e2prim = difft(e2,tvars);
      then GREATER(e1prim,e2prim);
// etc...
   end match;

 end difft;
```

4 Lists and Tuples

List and tuple data types are common in many languages used for meta-programming and symbolic programming.

4.1 Lists

The following operations allows creation of lists and addition of new elements in front of lists in a declarative way. Extracting elements is done through pattern-matching in match-expressions shown earlier.

- **`list`** – **`list`**`(el1,el2,el3, ...)` creates a list of elements of identical type. Examples: **`list`**`()`– the empty list, **`list`**`(2,3,4)` – a list of integers.
- **`nil`** – denotes an empty reference to a list or tree.
- cons – the call `cons(element, lst)` adds an element in front of the list `lst` and returns the resulting list. Also available as a new built-in operator `::` (coloncolon), e.g. used as in: `element::lst`.

Types of lists and list variables can be specified as follows:

- **`list`** – **`list`**<type-expr> is also a list type constructor, e.g. :
  `type RealList = list<Real>;`
- Direct declaration of a variable `rlist` that denotes a list of real numbers:
  `list<Real>    rlist;`

4.2 Tuples

Tuples can be viewed as instances of anonymous records. The syntax is a parenthesized list. The same syntax is used in extended Modelica presented here, and is in fact already present in standard Modelica as a receiver of values for functions returning multiple results.

- An example of a tuple literal:
  `(a, b, "cc")`
- A tuple with a single element has a comma in order to have different syntax compared to a parenthesized expression: `(a,)`
- A tuple can be seen as being returned from a function with multiple results in standard Modelica:
  `(a,b,c) := foo(x, 2, 3, 5);`
- Access of field values in tuples is achieved via dot-notation, `tupvalue.fieldnr`, analogous to `recvalue.fieldname` for ordinary record values. For example, accessing the second value in `tup`:
  `tup.2`

The main reason to introduce tuples is for convenience of notation. You can use them directly without explicit declaration. Tuples using this syntax are already present in the major functional programming languages.

A tuple will of course also have a type. When tuple variable types are needed, they can for example be declared using the following notation:

```
type VarBND  = record<Ident, Integer>;
```

or directly in a declaration of a variable `bnd`:

```
record<Ident, Integer>   bnd;
```

The tuple type used in the match-expression of the previous simple `eval` function is **`record`**`<Exp,Exp>`.

5 Positional Type Parameters

Class definitions in Modelica allow type parameters, declared as replaceable local types, e.g.:

```
class C2 = C(redeclare class
                ColoredClass = BlueClass);
```

Using a shorter angle-bracket syntax for positional type parameters similar to what is used in other object-oriented languages such as C++ or Java, this can be expressed as:

```
class C2 = C<BlueClass>;
```

We have used this syntax in several places throughout this paper, including a call to a polymorphic function in Section 7.

6 Expression Evaluator with Environments

The previous small expression evaluator presented in Section 3 could only handle constant expressions. The following example can handle expressions with variables. It demonstrates a different representation of expression trees, with `BINARY` nodes that are parameterized in terms of the operator, and thereby can handle several binary operators in a single of-branch in the match-expression. First we give the type declarations:

```
type Ident   = String;

uniontype Exp
  record RCONST Real x1; end RCONST;
  record IDENT  Ident x1; end IDENT;
  record BINARY Exp x1; BinOp op; Exp x2;
    end BINARY;
  record UNARY  UnOp x1; end UNARY;
  record ASSIGN Ident x1;  Exp x2;
```

```
    end ASSIGN;
end Exp;

uniontype BinOp
  record ADD end ADD;
  record SUB end SUB;
  record MUL end MUL;
  record DIV end DIV;
end BinOp;

uniontype UnOp
 record NEG end NEG;
end UnOp;

uniontype Value
  record REALval Real x1;  end REALval;
end Value;
```

The following `eval` function can handle evaluation of expressions with variable references. It calls the `lookup` function for access of variable references, and `apply_binop` for evaluation of binary operators.

```
type Ident   = String;

function eval
         // Evaluation of expression exp
         // in an environment env
  input  Env   env_1;
  input  Exp   exp_1;
  output Value value_1;
algorithm
 value_1 :=
  match (env_1,exp_1)
    local Real v,v1,v2;
         String   id;
         Env      env;
         Exp      e1,e2;
         Boolean  v3;
         BinOp    relop;
 // Real constant
    case (_,REALval(v)) then REALval(v);

 // variable identifier id
    case (env,IDENT(id)) equation
       v = lookup(env,id);
      then REALval(v);

// If id not declared, give an error
// message and fail through error
    case (env,IDENT(id)) equation
      v = not lookup(env,id);
      print("Error - undef variable: ");
      print(id);  print("\n");
      then fail()

 // expr1 binop expr2
    case (env, BINARY(e1,binop,e2))
      equation
        eval(env,e1) = REALval(v1);
        eval(env,e2) = REALval(v2);
        v3 = apply_binop(env,binop,v1,v2);
      then REALval(v3);

  end match;
end eval;
```

7 Function Parameters

A common and rather useful language feature not yet present in standard Modelica is the ability to pass function parameters. For example, passing the `add1` function to a mapping function that applies it to each element:

```
arr2 := arr_map_int(add1, {2,3,5,8})
```

returns:

```
{2,4,6,9}
```

We propose the following style of declaring a function that accepts a function formal parameter, exemplified through an example. The only syntax extension is to allow the declaration of a function without body, here `Functype`, which allows us to declare the type signature of the function formal parameter `func`.

```
function arr_map_int
  "Map over an array of integers"
   function Functype
     input Integer x1; output Integer x2;
   end FuncType;
  input replaceable function func
                              extends FuncType;
  input  Integer[:] inarr;
  output Integer[size(inarr,1)] outarr;
algorithm
  for i in 1:size(inarr,1) loop
    outarr[i] := func(inarr[i]);
  end for;
end arr_map_int;
```

The next example is polymorphic since the array element type `Type_a` is not fixed. It is a replaceable type, which makes it possible to apply `arr_map` to arrays of any element type. For example, applied to an array of strings, with the `addA` function that adds `"A"` to the end of a string:

```
arr3 :=
  arr_map<String>(addA, {"foo","fie"})
```

returns:

```
{"fooA","fieA"}
```

The definition of the `arr_map` function:

```
function arr_map
"Map over an array of elements of Type_a"
    replaceable type Type_a;
    function Functype
      input Type_a x1; output Type_a x2;
   end Functype;
  input replaceable function func
                          extends FuncType;
  input  Type_a[:]  inarr;
  output Type_a[size(inarr,1)] outarr;
algorithm
  for i in 1:size(inarr,1) loop
    outarr[i] := func(inarr[i]);
  end for;
end arr_map;
```

The semantics of function parameters include the following:

- Functions can be passed as actual arguments at function calls.
- Type checking done on the function formal parameter type signature, not including the actual names of inputs and outputs to the passed function.

8 Exception Handling

The design of exception handling capabilities in Modelica is currently in a preliminary phase. The following constructs are being discussed:

- A **try...catch** statement or expression.
- A **raise**(...) call for raising exceptions.

The statement variant has approximately the following syntax:

```
try
  <statements>
  ...
catch <x> then
  <statements>
  ...
end try;
```

The syntax of the expression variant is as follows:

```
try
  <expression>
catch <x> then
     <expression>
end try;
```

This design is still very preliminary, several issues need to be determined, and no implementation has yet been produced.

9 Conclusions

It has been demonstrated how Modelica can be extended with data structures and operations that are typically needed for comprehensive meta-programming and symbolic transformations. The extensions are declarative and preserve the declarative and equation-based style of Modelica. Recursive data types, lists, and tree pattern matching in match-expressions with local equations can be naturally integrated into the current Modelica 2.1 language. A implementation of most of this functionality has been tested on a number of examples, including those in this paper, and is currently being integrated into the OpenModelica compiler.

We believe that the combination of the modeling power and numeric capabilities of the current Modelica language, combined with symbolic transformation capabilities of the new extensions, will make Modelica into a very powerful meta-modeling and meta-programming language for the future.

10 Acknowledgements

This work has been supported by Swedish Foundation for Strategic Research (SSF), in the RISE project, and by Vinnova in the SWEBPROD project.

References

[1] Peter Aronsson, Peter Fritzson, Levon Saldamli, and Peter Bunus. Incremental declaration handling in Open Source Modelica. In SIMS - 43rd Conference on Simulation and Modeling on September 26-27, Oulu, Finland, 2002.

[2] Peter Aronsson, Peter Fritzson, Levon Saldamli, Peter Bunus and Kaj Nyström. Meta Programming and Function Overloading in OpenModelica. In proceedings of the 3rd International Modelica Conference, Linköping, Sweden, Nov 2003.

[3] Peter Fritzson, et al. The Open Source Modelica Project. In Proceedings of The 2nd International Modelica Conference, 18-19 March, 2002. Munich, Germany http://www.ida.liu.se/~pelab/modelica/ OpenModelica.html.

[4] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica*. 940 pp. Wiley-IEEE Press, 2004.

[5] Paul Hudak. *The Haskell School of Expression*. Cambridge University Press, 2000.

[6] The Modelica Association. The Modelica Language Specification Version 2.1, June 2003. http://www.modelica.org.

[7] Mikael Pettersson. Compiling Natural Semantics. PhD thesis, Linköping Studies in Science and Technology, 1995.

[8] Peter van Roy and Seif Haridi. Concepts, Techniques, and Models of Computer Programming. MIT Press, 2004.

[9] Tim Sheard. Accomplishments and Research Challenges in Meta-Programming. Lecture Notes in Computer Science, 2196:2–.., 2001

MOSILAB: Development of a Modelica based generic simulation tool supporting model structural dynamics

Christoph Nytsch-Geusen[1]
Thilo Ernst[1] André Nordwig[1]
Peter Schneider[2] Peter Schwarz[2]
Matthias Vetter[3] Christof Wittwer[3]
Andreas Holm[4] Thierry Nouidui[4]
Jürgen Leopold[5] Gerhard Schmidt[5]
Ulrich Doll[6] Alexander Mattes[6]

[1]Fraunhofer Institute for Computer Architecture and Software Technology
Kekuléstr. 7, D-12489 Berlin, christoph.nytsch@first.fhg.de

Fraunhofer IIS/EAS[2], Fraunhofer ISE[3], Fraunhofer IBP[4], Fraunhofer IWU[5], Fraunhofer IPK[6]

Abstract

The current GENSIM project, which is being conducted by a consortium of six Fraunhofer Institutes, is developing the generic simulation tool MOSILAB for the analysis of mixed time-continuous / time-discrete (hybrid) models of heterogeneous technical systems. One major innovation here in terms of simulation technology is the mapping of state-dependent changes in the model structure (model structural dynamics). This enables, for example, simulation experiments to be conducted with models of variable modelling depth. The modelling description language in the project MOSILA is based on Modelica, which was extended syntactically in terms of an adequate description of the model structural dynamics. The simulation tool is composed of a kernel and an integrated development environment and will be available in spring 2005 as a first prototypical implementation. The usability of the simulation tool is tested and evaluated in the GENSIM project by means of three use cases in the application areas fuel cell systems, hygrothermal building analysis and cutting tool systems.

Keywords: MOSILAB; Generic simulation tool; Model structural dynamics; object-oriented

1 Introduction

A heterogeneous technical system shows in dependency of its state a different physical behaviour. For example the physical behaviour of a starting plane in the different phases of rolling, taking off and flying can be described with different sets of physical effects, like the air and roll friction on the earth and the aerodynamic laws in the sky. An adequate simulation model for such a technical system needs also a high level of flexibility and adaptation in its model structure and in its equation system.

The innovative goal of the GENSIM project is to develop a new generic simulation tool for hybrid models, which supports model structural dynamics. Model structural dynamics in this context means, the model structure (the number and types of equations) can change during the simulation experiment in dependency of events, which are triggered from the state of the model self or its environment.

The object- and equation-oriented simulation language Modelica (http://www.modelica.org) offers in principal a good language concept for modelling technical systems with structural dynamics. For this reason Modelica was chosen as the language basis for the GENSIM simulation tool. Because the actual specification of Modelica is limited to fixed model structures during the simulation experiment, some syntactical extensions were made in GENSIM to obtain the possibility for describing model structural dynamics in a compact form.

2 Modelica Language Extension

The modelling description language MOSILA (**Mo**delling and **Si**mulation **La**nguage), which is specified and used in the GENSIM project, is based on Modelica. From the view of the modeller MOSILA is mainly an extension of Modelica.

Thereby existing models and also the disposable Modelica standard library can be reused within the GENSIM simulation tool directly or with a small effort of adaptation.

However the means of expressions of Modelica, particular for the description of variable model structures, are not powerful enough yet for using special simulation technologies, e.g. variable modelling depth. Therefore some syntactical extensions are added in MOSILA. These extensions where influenced by the UML^{H} [1], an adaptation of the UML [2] for the context of hybrid systems.

2.1 Dynamical object structures

Dynamical object structures were introduced to represent variable models during the simulation experiment. Thus, it becomes possible to extend the static model tree with dynamical objects during discrete phases of an experiment, which them self can spawn complex model trees. Since objects represent state attributes and behaviour in form of equations, the underlying equation system can be changed in size and quality when a structural change takes place. After such changes a new equation system will be derived for the following continuous phase.

2.2 Object-oriented Statecharts

To ease the description of structural changes, an adequate syntax for the control of discrete model switches were realised on the base of object oriented statecharts. Figure 1 shows a statechart controlling the mode switches of a landing device:

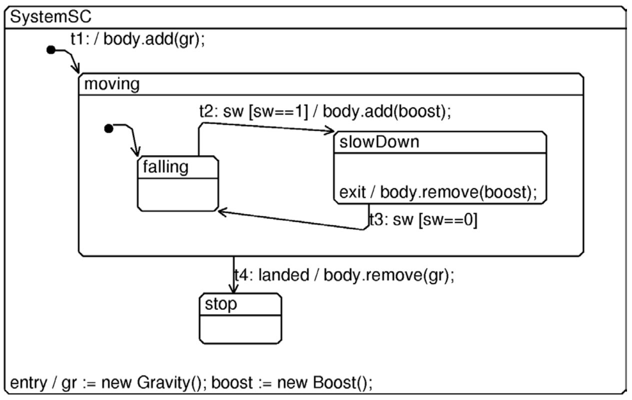

SystemSC
t1: / body.add(gr);
moving
t2: sw [sw==1] / body.add(boost);
falling
slowDown
exit / body.remove(boost);
t3: sw [sw==0]
t4: landed / body.remove(gr);
stop
entry / gr := new Gravity(); boost := new Boost();

Figure 1: Statechart of a landing device

At the beginning the device enters the mode “moving” and within its sub-mode “falling”. If the booster is enabled (depending on the decend speed), than it enters the “slowDown” mode until the booster is disabled. When the device reaches the ground the “moving” mode is left and “stop” is entered.

To simplify the modelling process direct support for these statechart descriptions was introduced in MOSILA as a special section of a model class. This extension is based on the Modelica type system: Each state is introduced by a state declaration. Within such declarations sub-states and transitions between these sub-states can be specified. Depending on the base type of a state two kinds of state compositions can be declared: Within an active XOR state only one direct sub-state is active at each time instant. Within an active AND state all its direct sub-states are active. Therefore, parallel and sequential processes can be comfortably modelled. Actions which have to take place during switching transitions are defined within the associated transition definition. Figure 2 shows the MOSILA implementation of the above introduced statechart:

```
model System
...
statechart
 state SystemSC extends State;
  state Moving extends State;
   state SlowDown extends State;
    exit action
     body.remove(boost);
    end exit;
   end SlowDown;

   State falling, start(isInitial=true);
   SlowDown slowDown;

   transition start -> falling end transition;

   transition t2 : falling -> slowDown
    event sw guard sw==1 action
    body.add(boost);
   end transition;

   transition t3 : slowDown -> falling
    event sw guard sw==0
   end transition;
  end Moving;

  State stop,start(isInitial=true);
  Moving moving;

  transition t1 : start -> moving action
   body.add(gr);
  end transition;

  transition t4 : moving -> stop
   event landed action
   body.remove(gr);
  end transition;

 end SystemSC;
end System;
```

Figure 2: Implementation of a statechart for the control system of the landing device

The state space for action statements, e.g. assignment, is given by the surrounding type definitions, and thus the statechart acts on the attributes of the associated class.

2.3 Dynamical behaviour

Further, the extended language concept offers an infrastructure, which enables the extension of (basic) model by effects in form of behavioural objects. For this purpose the action language is extended by the operations “add()” and “remove()”, which connect/disconnect the given argument (behaviour object) to the target object (base object). To extend underlying balance equations, a new connector type “sum” is introduced. The semantics of this connector type is like zero sum (“flow”), but with a negatively signed base attribute. Thus, balance equations can be extended by terms, which are encapsulated by objects. The following implementation (Figure 3) shows the environment of the landing device:

```
connector FPort
 sum Real F=0;
 Real m=0;
end FPort;
partial model BodyInterface
 FPort p;
end BodyInterface;
model Body extends BodyInterface(p.m=100);
 Real a=0, v=0, s=100;
equation
 der(v) = a; der(s) = v; a = p.F / p.m;
end Body;
model Gravity
 extends BodyInterface;
 parameter Real g=9.81;
equation
 p.F = - p.m * g;
end Gravity;

model Boost
 extends BodyInterface;
 discrete Boolean empty=false;
 Real m;
equation
 p.m = m;
 empty = (not m>20); // = if m>20 then false else
true;
 der(m) = if empty then 0 else -10;
 p.F = if empty then 0 else 1200;
end Boost;
model System
 Body body;
 Gravity gr;
 Boost boost;
 event discrete Integer sw=0;
 event discrete Boolean landed=false;

equation
 sw = if body.v < -5 then 1 else if body.v >= 0
then 0 else pre(sw);

 landed = ( body.s <= 0 ); // = if body.s <=0
then true else false;
...
end System;
```

Figure 3: Implementation of the dynamical behaviour of the landing device

The base model “Body” and the effects “Gravity” and “Boost” have the same interface “BodyInterface”, which introduces the variables to connect/disconnect during add/remove operations. The attribute “F” has “sum” quality since it is used to model a (dynamically changed) balance equation. The events “sw” and “landed”, which drive the above introduced statechart, are modelled within the top level class “System” as special discrete variables.

3 The MOSILAB Simulator

3.1 MOSILAB Architecture

The GENSIM simulation tool MOSILAB (**Mo**deling and **Si**mulation **La**boratory) includes the simulation kernel (consisting of a model compiler, a runtime system and a numerical solver framework) and an IDE (Interactive Development Environment), the interface to the user of the simulation system. It supports him both in the modelling process with the help of graphical UML and text editors and during the simulation experiment.

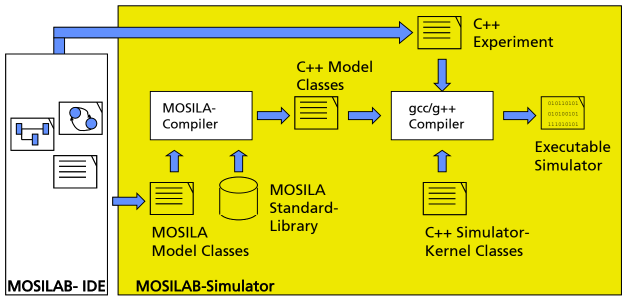


Figure 4: Data flow within MOSILAB

Figure 4 shows the data flow within the MOSILAB tools: Beside experiment definitions, the models also developed within the IDE are stored as MOSILA model classes. Together with the MOSILA standard library, these MOSILA models are compiled to C++ classes by the MOSILA compiler. Using the GNU gcc/g++ compiler, the executable simulator is built from these C++ representations and the simulator kernel classes.

3.2 Numerical solver framework

The numerical solver framework of MOSILAB features general functions such as the construction of the numerical model based on the modelling description, the main simulation control loop and is able to integrate different numerical algorithms.

The simulator kernel library contains some basic algorithms for solving nonlinear implicit differential-algebraic equations (DAE): the EULER backward formula and trapezoidal rule as simple methods from standard textbooks, and IDA, a powerful public-domain DAE solver for sequential and parallel computers [3]. IDA is a successor of DASSL, a well established DAE solver [4, 5]. IDA provides a routine that computes consistent initial conditions from a user's initial guess for a class of problems - a very important feature for simulating systems with model structural dynamics. The integration method in IDA is a variable-order, variable-coefficient BDF method (backward differentiation formula). The nonlinear systems are solved with some form of NEWTON iteration.

But the library is an open one: according to the requirements of a dedicated MOSILAB implementation or of special problem requirements, additional solvers may be implemented, e.g. data-flow or event-oriented methods, or simplified solvers for linear state-space equations in explicit form.

Such a tailored algorithm, which is implemented in the GENSIM project, is the Plug-Flow method [6, 9]: A so called plug flow model uses finite mass elements $\Delta m = m' \cdot \Delta t$ and finite energy elements $\Delta Q = Q' \cdot \Delta t$ with a fixed time step, allowing the mass flow through closed and opened networks to be traced. This guarantees a fast calculation of the object chain. The plug is initiated in a “pump” or “ventilator”-object, then shifted through the branched network and is returned to the origin (pump or ventilator). This mechanism allows simple modelling of flow delay effects and mass balance at the second call of the pump/ventilator in one time step. Due to the decentralised solver method the state equations of each object are calculated with the updated mass flow of the previous object.

3.3 MOSILAB Configurations

MOSILAB can be configured in to act in three modes:

a) The generated simulator is represented by a single, monolithic C/C++ application. This option has the smallest memory footprint and only few dependencies on the underlying platform, so it is most useful e.g. for embedded applications. However, the functionality w.r.t. dynamic parameterization at run-time is limited.

b) The simulator is represented by a shared object file which can be dynamically linked to a main program which controls the simulation. MOSILAB uses the Python language and interpreter (http://www.python.org) as its central mechanism for experiment control. The simulator is loaded as an “extension” into the interpreter, and “experiment scripts”, written in Python, access the simulator API via a Python-level interface.

c) The simulator acts as a service. In this mode, the simulator is linked with appropriate libraries to publish its API via standard TCP/IP-based protocols such as SOAP [5] in a web or grid services framework (e.g. the upcoming release 4 of the Globus Toolkit [6]). In this mode, the simulator can easily be controlled in protocol-based, platform-independent manner, and it is easy to deploy multiple (and potentially large numbers of) “simulator service instances” in a coordinated way in a heterogeneous network or Grid, for instance to solve an optimization problem. Python-based experiment control support is available in this mode as well – a (Python) client library is used to talk to the simulator’s API over the network in this case. The simulator maintains a run-time representation of the model object hierarchy (as defined in the source and evolving according to the structural variability of the model). This run-time model can be inquired via introspection features of the simulator API, so (using the synchronisation features offered by this API, too) experiment scripts are able to follow the structural changes over the entire course of a simulation run. This way, if special reactions to model structure changes are needed, which cannot be formulated in the model itself due to their complexity, such reactions can easily be implemented in the experiment script.

3.4 Simulator Coupling

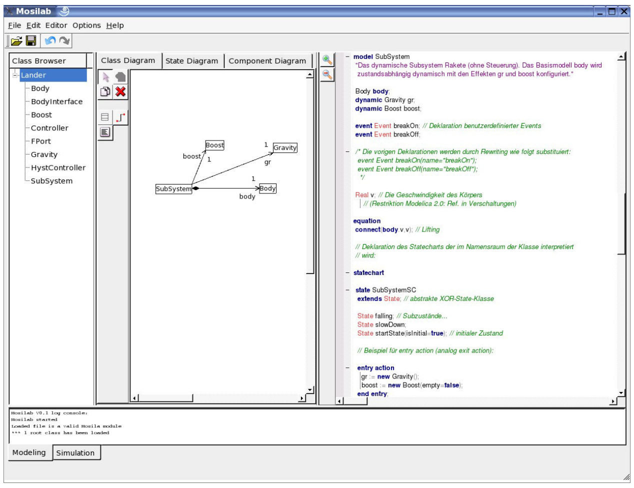
Besides the service-oriented approach to coarse-grained coupling of simulation components described in item c) above, MOSILAB also supports simulator coupling on a fine-granular level. In addition to implementing the standard external function interface defined in Modelica, special interface support is being developed to support coupling with the widely used simulator MATLAB/Simulink and certain specialized simulators relevant to the pilot applications (e.g. CFD and FEM tools). These developments, too, build on the flexible simulator API offered by MOSILAB’S simulation kernel.

4 The MOSILAB Development Environment

The MOSILAB Development Environment (MOSILAB-IDE) supports the user during the modelling process and the simulation experiment.

In the modelling mode the user can choose between three graphical UML^H-editors (class diagrams, collaboration diagrams and statecharts) and a text editor. While the graphical views give the user an intuitive overview about the structure and the logic of a complex model, the text editor offers the user features like syntax highlighting for implementing the MOSILA/Modelica models.

In the experiment mode of the MOSILAB-IDE the user can define the root model for the simulation experiment, can parameterize model variables and can choose and configure a suitable numerical solver. Furthermore he can define a subset of model variables, which should be observed during the simulation experiment. The observed variables are the basis for different types of post-processing. Figure 5 shows a screenshot of the prototypical implementation of the MOSILAB-IDE.



```
model SubSystem
 "Das dynamische Subsystem Rakete (ohne Steuerung). Das Basismodell body wird
  zustandsabhängig dynamisch mit den Effekten gr und boost konfiguriert."

 Body body;
 dynamic Gravity gr;
 dynamic Boost boost;

 event Event breakOn; // Deklaration benutzerdefinierter Events
 event Event breakOff;

 /* Die vorigen Deklarationen werden durch Rewriting wie folgt substituiert:
  event Event breakOn(name="breakOn");
  event Event breakOff(name="breakOff");
   */

 Real v; // Die Geschwindigkeit des Körpers
   // (Restriktion Modelica 2.0: Ref. in Verschaltungen)

equation
 connect(body.v,v); // Lifting

 // Deklaration des Statecharts der im Namensraum der Klasse interpretiert
 // wird:

statechart

 state SubSystemSC
  extends State; // abstrakte XOR-State-Klasse

  State falling; // Subzustände...
  State slowDown;
  State startState(isInitial=true); // initialer Zustand

  // Beispiel für entry action (analog exit action):

  entry action
   gr := new Gravity();
   boost := new Boost(empty=false);
  end entry;
```

```
Mosilab V0.1 log console:
Mosilab started
Loaded file is a valid Mosila module
*** 1 root class has been loaded
```

Figure 5: MOSILAB-IDE in the modelling mode

5 Applications

In the GENSIM project model libraries for the three technical application areas fuel cell systems, hygrothermal building analysis and cutting tool systems are developed. On their basis different use cases should be analysed. In each use case the methodological possibilities of the model structural dynamics will be evaluated: For example cutting tool system models are developed, which can activate different physical sub models for tools und working pieces in dependency of the system state during the simulation experiment (e.g. contact between the tool and the working piece exists or not).

5.1 Fuel cell systems

The future structure of power grids will consist of a huge fraction of decentralised power generators. Especially the low voltage grid will be penetrated by small and medium photovoltaic systems, medium combined heat and power units based on natural gas or bio fuels as well as residential fuel cell cogeneration power systems. Dynamic simulation of the entire low voltage grid offers the possibility of analysis and optimisation of the grid in terms of dimensioning and system management.

If cogeneration systems (e.g. residential fuel cell systems) are regarded, thermal aspects have to be considered. The ecologic and economic evaluation of these innovative energy supply systems needs efficient models due to seasonal effects and the necessity of simulation runs in the range of one year [9, 10].

Model structural dynamics allows the investigation of a huge number of grid connected residential fuel cell cogeneration systems in combination with other decentralised energy systems such as photovoltaic, small wind turbines, bio mass systems, etc. in a very efficient way. In this approach the model depth is defined by the operating point and the operating behaviour respectively.

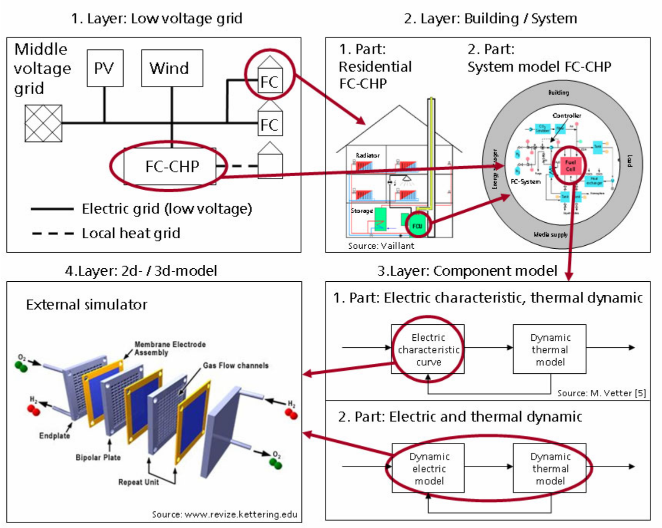


Figure 6: Definition of the single layers of the residential fuel cell system representing the model depth. The current model depth will be switched in dependence of boundary conditions and control actions.

As showcase the model structural dynamics of a fuel cell system is described in the following section. In a first approach four layers of abstraction are defined, shown in figure 6. The current layer will be switched, if discrete events (e.g. caused by a state variable crossing a threshold) require the change of the model depth. In case of a stationary operating point of the fuel cell system, it is sufficient to represent this device with a simple characteristic curve. As soon as boundary conditions (e.g. cooling temperature) are changing or the operating point is changed by a controller, more detailed models are needed to reproduce the transient behaviour in an accurate way.

Furthermore, critical system states can require detailed models, with which even single cells can be investigated. As an example flooding effects at low operating temperatures or at high load currents shall be mentioned. If a system simulation tool is able to reproduce even such effects, efficient control strategies can be developed to reduce or even avoid system failure. In this project the coupling with external FEM tools are planned to fulfil these demands.

5.2 Hygrothermal building analysis

In the area of Building Physics hygrothermal models of building envelopes to compute coupled transport processes of heat and moisture for one- or multidimensional cases are widely used. In those models however the boundary conditions of heat and moisture have to be user-defined before starting the simulation.

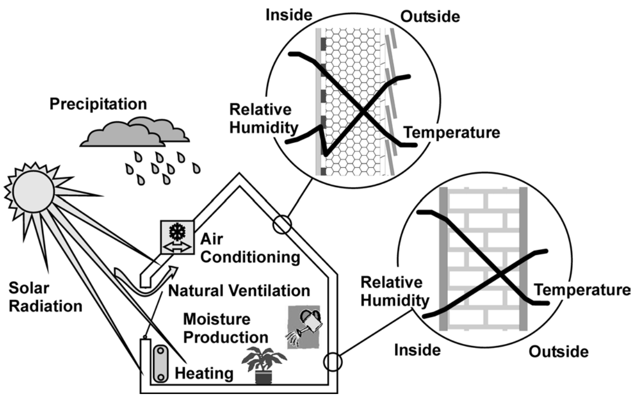


Figure 7: Coupled hygrothermal physical effects in the building envelope

A model that would take into account a multizonal building or even only a single room and the building envelope in detail – thus rendering the definition of the inner boundary conditions for the envelope unnecessary – is still to be defined. Such a model would allow analysing cases with strong reciprocal effects between the climate in the room and the behaviour of the building components (see figure 7). For example the influence of moisture buffering and non buffering surfaces of the components in combination with different ventilation strategies can be investigated to consider the efficiency of thermal drying and ventilation strategies to keep the indoor climate (especially the humidity) in a favourable range.

For this reason Fraunhofer IBP and FIRST started the development of such a new hygrothermal building model within the GENSIM project. Fraunhofer IBP can make use of its extensive experiences with the development and experimental validation of the simulation tool WUFI [11] for the detailed simulation of hygrothermal behaviour of building components. Fraunhofer FIRST has long-year modelling experience in the area of thermal building simulation with the generic and object-oriented simulation environment SMILE [12].

The goal in GENSIM is the development of a Modelica /MOSILA model library, which will contain models of one- and of two-dimensional coupled heat and moisture transport within wall constructions, a thermal/optical window model, a hygrothermal air volume model, a thermal/optical room model, an environment model for the climatic boundary conditions as well as an inhabitant model. From these models it is possible to set up configurations of rooms or whole buildings in a very flexible way by using the object-oriented modelling method. For example figure 8 shows UML^{H}-class diagram for an outside thermal wall model, which is a part of the model library building for hygrothermal building simulation.

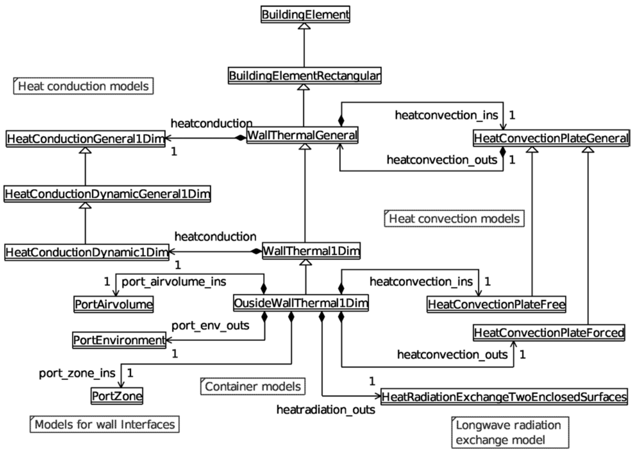


Figure 8: UML^{H}-Class diagram for an outside thermal wall model

The efficiency of the model structural dynamics should be evaluated for the coupled transport processes of heat and moisture in wall constructions: For example, if the gradient of temperature or moisture becomes greater then a limit-value, the level of discretisation of the wall model will be set on a higher value or the other way around.

During the project the hygrothermal building model will be validated on test houses. Two rooms which are identical both geometrically and in respect of solar gain and outdoor climate, however differ extremely in the sorption behaviour of their wall surfaces are used to validate the building model (see figure 9).

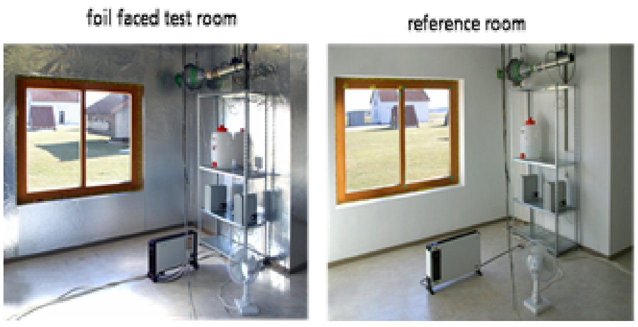


Figure 9: Test rooms for validating the hygrothermal building model

This is to be done by measurements of the energy and moisture balance in both rooms during cycles of heating and cooling as well as humidification and dehumidification.

5.3 Cutting tool systems

The development of high performance cutting processes requires, along with suitable machine tools and clamping devices also specially balanced and designed cutting tools. Safety and precision are the essential criteria at the judgment of tools for the high speed processing. Numerical simulations offer the possibility to evaluate different variants already in the outline process without existing samples. A proved method is the analysis of the tool behaviour under operating conditions with the finite element analysis (FEA).

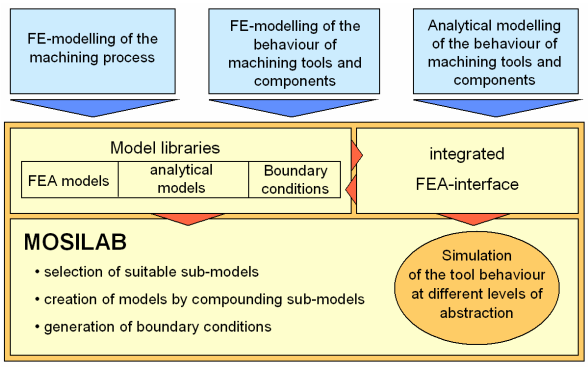

FE-modelling of the machining process
FE-modelling of the behaviour of machining tools and components
Analytical modelling of the behaviour of machining tools and components
Model libraries
FEA models
analytical models
Boundary conditions
integrated FEA-interface
MOSILAB
• selection of suitable sub-models
• creation of models by compounding sub-models
• generation of boundary conditions
Simulation of the tool behaviour at different levels of abstraction

Figure 10: Cutting tool modelling in the framework of MOSILAB

The complexity of requested models depends on workpiece and the shaped elements and also from the type of the tool and its loading. Complex cutting tools consist of several components. Due to relative motions of the components under high centrifugal force load, cutting forces and clamping stresses, the tool models are highly non-linear and heterogeneous.

Different stages of the loading by clamping, centrifugal and cutting forces cause structural dynamic model behaviour and require corresponding changes of model parameters or even switching between different types of models (Figure 10). In the GENSIM project analytical models of complex cutting tools will be developed to be integrated in MOSILAB. FEA will deliver the parameters of these models. Respecting the complex structural dynamics different sets of sub-models are required to compile adequate cutting tool models. In addition to this, a special interface for FEA and MOSILAB data transfer will be developed for these types of cutting tools, which can not be simulated by a homogenous analytical model.

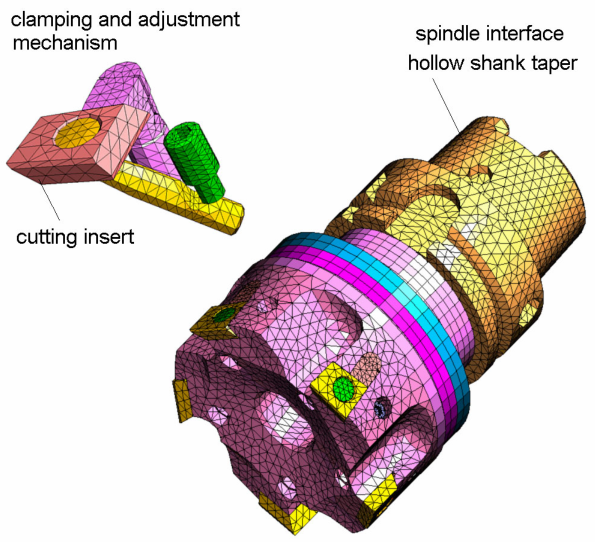

clamping and adjustment mechanism
spindle interface hollow shank taper
cutting insert

Figure 11: Example of a complex cutting tool

The simulation of cutting tool behaviour in the GENSIM project covers the complex tool – starting from the cutting edge up to the spindle interface (see figure 11). The behaviour of the cutting tool in use is determined by the statically and dynamical components of the cutting force. These loads are essential boundary conditions to investigate the structural dynamics of the cutting tool itself.

To get these loads, the cutting process itself is simulated using another FE-model. Here a small section of the cutting process is represented in which the chip formation happens. As seen in figure 12 the outer edge of the cutting tool and the upper layer of the workpiece are modelled.

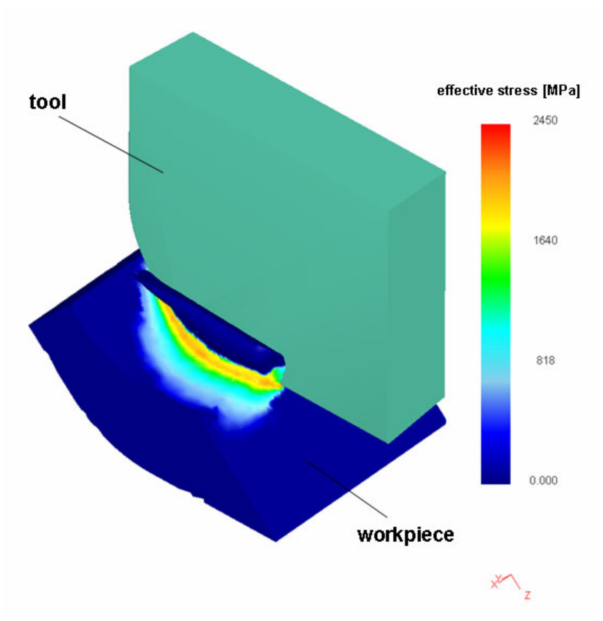


Figure 12: Simulation of the machining process

Through the simulation continuous or segmented chip formation can be described and the distribution of the resulting cutting forces can be calculated connected to the simulation time. Here from a three-dimensional load vector is determined, which acts on the cutting edge of the tool.

Within MOSILAB the simulation models should be combined to a simulation tool which describes the model structural dynamic of the interaction between cutting process and tool holder. For that a library with different combinations of tool holder, cutting tool material and workpiece material will be provided by MOSILAB. If in the cutting process the cutting force reaches limits, which indicate a meaningful influence on the tool holder (vibrations for example) the simulation depth can be changed [13]. This enables analysing changes in behaviour of the tool holder caused by microscope events happening in the region of chip formation. Also the results of the tool holder simulation are useful to include changes in the boundary conditions of cutting process caused by displacement of the tool holder itself.

6 Conclusions

The important results of the GENSIM project are the Modelica language extension for an adequate description of model structural dynamics, the new generic simulation tool MOSILAB and three model libraries of different technical applications:

a) **Modelica language extension MOSILA**: The new modelling description language MOSILA, which is mainly an extension of Modelica, is able to describe simulation models with a time depending model structure during the simulation experiment. This was realised by using dynamical object structures together with object-oriented statecharts for the language specification. An example for this new simulation technology in MOSILA is an adaptive simulation model containing a set of physical sub-models, from these some are activated or not in dependency of the state of the model self.

b) **Simulation tool MOSILAB**: The new developed generic simulation tool for hybrid systems, which includes a model compiler, a runtime system and a numerical solver framework, is able to translate MOSILA or Modelica models to an executable simulation program. Hereby the user is supported by the MOSILAB development environment, which offers possibilities for the graphical and textual modelling process, for simulation experiments and for post-processing. On the one hand the scalable software architecture of MOSILAB can generate small simulators as monolithic C/C++ applications. On the other hand simulator configurations with more flexibility for the simulation experiment are possible by loading MOSILAB and the compiled model libraries as an extension in a python interpreter, while the simulation experiment is formulated in the script language Python. Current research activities in the GENSIM project will show that MOSILAB can also act as a service in web or grid frameworks and can by coupled with other simulation tools like MATLAB/Simulink.

c) **Model Libraries**: Three model libraries for the technical applications fuel cell systems, hygrothermal building analysis and cutting tool systems are developed and validated in GENSIM. In each application area the efficiency of the model structural dynamics are analysed. In relation to its time dynamics, the analysed systems in GENSIM overlap the millisecond- to second-scale (cutting tool systems), the second to hour-scale (fuel cell systems) and the hour to year-scale (hygrothermal building analysis). For these reasons these application areas together are also a suitable test bed for the numerical basis of the simulation tool MOSILAB.

References

[1] Nordwig, A.: Integration von Sichten für die objektorientierte Modellierung hybrider Systeme, Verlag dissertation.de, ISBN 3-89825-692-8, 2003.

[2] Rational: Unified Modeling Language, Version 1.3, 1999.

[3] Hindmarsh, A. C. et al.: "SUNDIALS: Suite of Nonlinear and Differential/Algebraic Equation Solvers," ACM Transactions on Mathematical Software, 2005. Also available as LLNL technical report UCRL-JP-200037, http://www.llnl.gov/CASC/sundials/

[4] Petzold, L.R.: A description of DASSL: A differential/algebraic system solver, in IMACS Trans. Scientific Computing Vol. 1 (1993), pp. 65-68.

[5] Gear, C.W.; Petzold, L.R.: ODE methods for the solution of differential/algebraic systems, SIAM Journal on Numerical Analysis, 21 (1984) 4, 716 – 728

[6] Wittwer, C.: ColSim – Simulation von Regelungssystemen in aktiven solarthermischen Anlagen. Dissertation Universität Karlsruhe (TH), 1999. www.ubka.uni-karlsruhe.de.

[7] http://www.w3.org/TR/soap

[8] http://www.globus.org

[9] Vetter, M.: Modellbildung und Regelstrategien für erdgasbetriebene Brennstoffzellen-Blockheizkraftwerke, Dissertation Universität Karlsruhe (TH), erscheint Anfang 2005.

[10] Muche, L.; Schneider, P..; Vetter, M.; Wittwer, C.: Modellierung und Simulation der Energieversorgung von Gebäuden mittels Brennstoffzellensystem. Proc. 5. GMM/ITG/GI-Workshop "Multi-Nature Systems", 18. Februar 2005, Dresden, 2005.

[11] Künzel, H.M.: Simultaneous Heat and Moisture Transport in Building Components. - One- and two-dimensional calculation using simple parameters. IRB Verlag, 1995.

[12] Nytsch-Geusen, C.; Bartsch, G.: An Object Oriented Multizone Thermal Building Model based on the Simulation Environment SMILE. Proceedings of Building Simulation 2001, International Building Performance Simulation Association, Rio de Janeiro, 2001.

[13] Nytsch-Geusen, C.; Doll, U.; Leopold, J.: Anwendung generischer Simulationstools zum Werkzeugdesign, Chemnitzer Produktionstechnisches Kolloquium, 2004.

Session 7a

Electrical Systems

Standard Package Modelica.Electrical.Digital

Christoph Clauß$^{1)}$**, Ulrich Donath**$^{1)}$**, André Schneider**$^{1)}$**, Enrico Weber**$^{2)}$
1) Fraunhofer Institute for Integrated Circuits, Branch Lab Design Automation
Zeunerstraße 38, D-01069 Dresden, Germany
2) University of Applied Sciences, Technikumplatz 17, D-09648 Mittweida
{clauss, donath, schneider}@eas.iis.fraunhofer.de

Abstract

According to the IEEE 1164 standard the Modelica.Electrical.Digital library was developed which uses nine-valued logical signals. The first stage of extension contains basic gate devices, sources, delay devices, and convertes. The main principles of implementation are demonstrated as well as some examples which show some possibilities of usage. Using converters, the electrical digital components are capable of interacting with the components of other Modelica libraries.

1 Introduction

The Modelica language [1], [2] already allows the formulation of logic behaviour using both the predefined Boolean variable type (true, false) and Boolean operators (or, and, not). For many applications these possibilities are sufficient. However, the description of complex digital electronic behaviour requires a very extension of the simple Boolean logic. The reason is that some of the properties of electronic circuits have to be transmitted to the logic approach, e.g. the existence of an *unknown* signal state, of different signal strengths etc..

Considering the VHDL language, the IEEE 1164 standard [3], [4], [5], [6], [7] is generally accepted and widely used for the description of digital electronic devices. It is based on nine-valued logical signals and defines the behaviour of simple and more general digital devices including time-dependencies. Due to the importance of this standard the digital electronic library should be developed in accordance with it.

In this paper an overview is given on the devices available. Details of the implementation are presented as well as some questions of the usage in combination with other libraries. Many examples give an impression of the actual state of the library.

2 Overview

The nine digital signal values are 'U' (uninitialized), 'X' (forcing unknown), '0' (forcing 0), '1' (forcing 1), 'Z' (high impedance), 'W' (weak unknown), 'L' (weak 0), 'H' (weak 1), '-' (don't care).

The library is devided into:

- delay models (transport, inertial, sensitive inertial)
- basic gates without delay (Not, And, Nand, Or, Nor, Xor, Xnor)
- basic gates including intertial delay (InvGate, AndGate, NandGate, OrGate, NorGate, XorGate, Xnorgate, BufGate)
- sources (Set, Step, Table, Pulse, Clock)
- converters (for connections with Boolean, and with Real, and for the restriction of the digital logic values to 'X01' or to 'X01Z' or to 'UX01')
- auxiliary subpackages of interface definitions and tables
- examples

The model definition can be seen in the library. Some of the models are explained in detail within the next paragraph. The icons of some models can be seen in **Fig. 1**. Most of the icons correspond to the European standard [8].

The digital library will be developed in at least two steps. The first step contains the devices mentioned above. Components like flip-flops, transfer gates, memories (RAM, ROM), and multiplexers are still missing. The behavioural models of these components will be added within the second step of library development. At the present stage such devices must be composed using the available gates. Examples of such compositions can be found in the example subpackage.

3 Details of Implementation

The basic idea was to offer a library of digital logic devices which can be placed and connected by the user to model a digital logic scheme. Otherwise, Modelica also allows to create models in a netlist like way by instan-

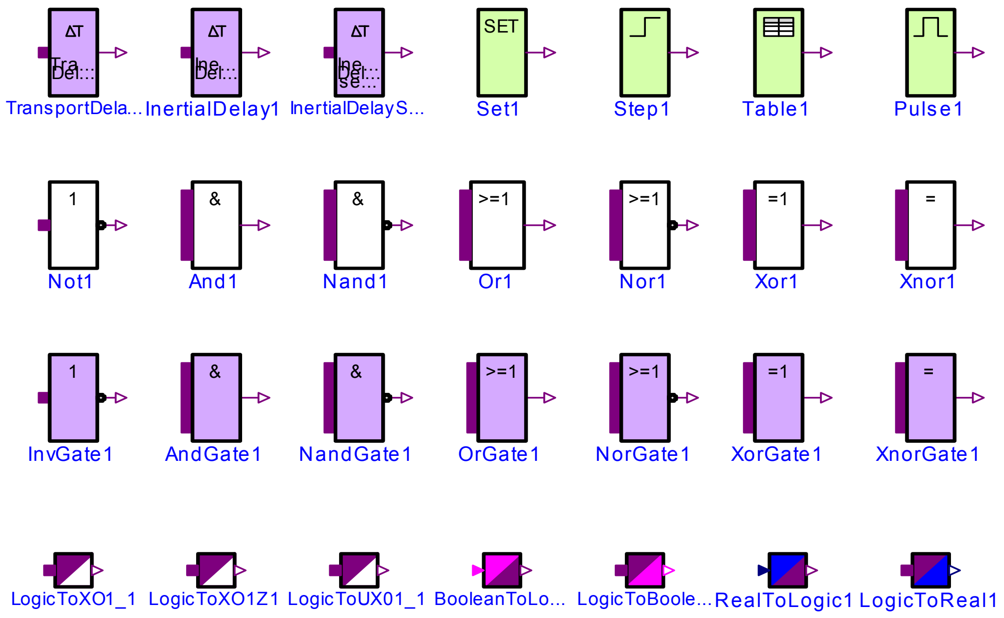


Figure 1: Components of the Modelica package *Modelica.Electrical.Digital*.

tiating and connecting devices on a text level. In both cases, a network of digital devices can be described on its connections where digital logic signals are transmitted. In this paragraph the behavioural modelling of some devices is shown exemplarily.

Since the number of logic values is limited signals do not change continuously but at discrete event times. Furthermore, nothing has to be differentiated. To calculate the output of digital devices an intensive usage of the algorithm section is necessary in the models. The simulator's task is not to solve a DAE but a system of algebraic equations at discrete event time, whose dimension is normally high and which contains lots of conditional clauses.

3.1 Signals and Connectors

The nine logic values are coded using an integer logic type:

```
type Logic = Integer

record LogicValue
  constant Integer min=1;
  constant Integer max=9;
  constant Logic 'U'=1 "Uninitialized";
  constant Logic 'X'=2 "Forcing Unknown";
  constant Logic '0'=3 "Forcing 0";
  constant Logic '1'=4 "Forcing 1";
  constant Logic 'Z'=5 "High Impedance";
  constant Logic 'W'=6 "Weak    Unknown";
  constant Logic 'L'=7 "Weak    0";
  constant Logic 'H'=8 "Weak    1";
  constant Logic '-'=9 "Don't care";
end LogicValue;
```

The sequence coded in this record corresponds to the IEEE 1164 sequence. This way simplifies the adaptation of logic value tables from the standard. Later on this record definition could be replaced by an enumeration type definition.

At the connections (ports) of the devices logic values are transmitted. Therefore, connectors are defined which only need a logic value signal. Since in most cases the signal flow direction is well defined, input and output connectors are specified:

```
connector DigitalSignal=Logic
"Digital port (both input/output
                possible)";
connector DigitalInput=input DigitalSignal;
connector DigitalOutput=
                output DigitalSignal;
```

The signals at the connectors are scalar ones. If vectors of signals are needed vectors of connectors have to be defined. This idea is taken over from the Modelica.Blocks library. The usage of both scalar and vector connectors can be seen at the following partial model for multiple input - single output devices which is used for modeling of Basics and Gates:

```
  partial block MISO
    import D = Modelica.Electrical.Digital;
    parameter Integer n(final min=2) = 2
                          "Number of inputs";
    D.Interfaces.DigitalInput x[n];
    D.Interfaces.DigitalOutput y;
  end MISO;
```

3.2 Basics

In the Basics subpackage the simple logic operations Not, And, Nand, Or, Nor, Xor, and Xnor are modeled.

The Not model is a single-input-single-output model. The logic input value, which is an integer between 1 and 9, specifies the row in the NotTable in which the output value can be found that negates the input value. The Modelica text of the Not device is:

```
  model Not
    import D = Modelica.Electrical.Digital;
    import L = D.Interfaces.LogicValue;
    extends D.Interfaces.SISO;
  protected
    D.Interfaces.Logic
                   auxiliary(start=L.'0');
  equation
    auxiliary = D.Tables.NotTable[x];
    y = pre(auxiliary);
  end Not;
```

The NotTable is defined in the IEEE 1164 standard:

input:	U	X	0	1	Z	W	L	H	-
output:	U	X	1	0	X	X	1	0	X

Regarding that the input code is used as index in the NotTable array, this is described in Modelica:

```
  constant D.Interfaces.Logic
  NotTable[L.max]=
   {L.'U',L.'X',L.'1',L.'0',L.'X',
    L.'X',L.'1',L.'0',L.'X'};
```

In the model the result is not put to the output directly but the pre-operator is applied to an intermediate variable. This is necessary to avoid algebraic loops which can appear in some cases. Therefore, the pre-operator is generally used.

As an example with multiple inputs the And model is explained. The source code is without any annotations:

```
model And
  import D = Modelica.Electrical.Digital;
  import L = D.Interfaces.LogicValue;
  extends D.Interfaces.MISO;
protected
  D.Interfaces.Logic
             auxiliary[n](each start=L.'U');
equation
  auxiliary[1] = x[1];
  for i in 1:n - 1 loop
    auxiliary[i + 1] =
    D.Tables.AndTable[auxiliary[i],x[i + 1]];
  end for;
  y = pre(auxiliary[n]);
end And;
```

The And model inherits the MISO partial model (c.f. 3.1). Within a loop to the first two input signals the *and* operation is applied. To the result and the next input signal the *and* operation is applied again, until all inputs are combined. Like in the Not model the pre-operator is used . The and-operator is realised using the AndTable. The code numbers of the input signals define the position (both row and line number) in the matrix where the result can be found. Written in an abbreviated form the AndTable is:

input1		U	X	0	1	Z	W	L	H	-
i	U	U	U	0	U	U	U	0	U	U
n	X	U	X	0	X	X	X	0	X	X
p	0	0	0	0	0	0	0	0	0	0
u	1	U	X	0	1	X	X	0	1	X
t	Z	U	X	0	X	X	X	0	X	X
2	W	U	X	0	X	X	X	0	X	X
	L	0	0	0	0	0	0	0	0	0
	H	U	X	0	1	X	X	0	1	X
	-	U	X	0	X	X	X	0	X	X

In the models Nand, Nor, and Xnor the NotTable is applied to the result of the And-, Or-, and Xor-tables respectively.

3.3 Delays

In the library there are three delay models. The *transport delay* model is an application of the Modelica delay operator. The input signal is delayed by delayTime exactly as it is. The output of the model can be specified for the time interval between zero and delayTime. The algorithm section of the TransportDelay model is:

```
algorithm
  x_delayed := integer(delay(x, delayTime));
  y := if delayTime > 0 then
         if time >= delayTime then x_delayed
         else y0
       else x;
```

Another type of delay models is the *inertial delay.* In the InertialDelay model the input value must keep constant for the delayTime interval before it is passed on the output. The Modelica code of the inertial delay is:

```
block InertialDelay
  import D = Modelica.Electrical.Digital;
  import I = D.Interfaces;
  import L = D.Interfaces.LogicValue;
  extends DI.SISO;
  parameter Modelica.SIunits.Time
            delayTime=0 ;
  parameter DI.Logic y0=L.'U';
protected
  DI.Logic y_auxiliary(start=y0, fixed=true);
  DI.Logic x_old(start=y0, fixed=true);
  discrete Modelica.SIunits.Time
  t_next(start=delayTime, fixed=true);
algorithm
  when delayTime > 0 and change(x) then
    x_old := x;
    t_next := time + delayTime;
  elsewhen time >= t_next then
    y_auxiliary := x;
  end when;
  y := if delayTime > 0 then y_auxiliary
       else x;
end InertialDelay;
```

If the input signal x changes its value, the variable t_next is set to that time at which the output should change, that means at time + delayTime. If the time reaches t_next without another input change then the input change becomes active at the output. Otherwise if x changes before t_next, t_next is increased due to the new input change. In **Fig. 2** an example of an inertial delay with delayTime=1s is shown. Input changes smaller than 1s are neglected.

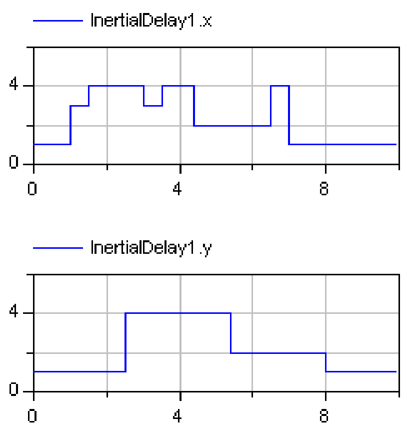


Figure 2: Inertial delay example

A generalization of the inertial delay is the sensitive intertial delay *InertialDelaySensitive*. For rising and falling edges different delay times can be specified. With a delay table it is decided whether a signal changing is regarded as rising (1) or falling (-1) or indifferent (0). Indifferent changes are not delayed. The delay table used in this library is:

	after	U	X	0	1	Z	W	L	H	-
b	U	0	0	-1	1	0	0	-1	1	0
e	X	0	0	-1	1	0	0	-1	1	0
f	0	1	1	0	1	1	1	0	1	1
o	1	-1	-1	-1	0	-1	-1	-1	0	-1
r	Z	0	0	-1	1	0	0	-1	1	0
e	W	0	0	-1	1	0	0	-1	1	0
	L	1	1	0	1	1	1	0	1	1
	H	-1	-1	-1	0	-1	-1	-1	0	-1
	-	0	0	-1	1	0	0	-1	1	0

3.4 Gates

In the Gates subpackage there are collected the InvGate, AndGate, NandGate, OrGate, NorGate, XorGate, XnorGate, and the BufGate. Each of the gates is graphically composed by a basic logic model whose output is delayed by a sensitive inertial delay. The InvGate consists of a Not model with delayed output. As a special case the BufGate consists only of a sensitive inertial delay. For the sake of completeness the BufGate should belong to that subpackage. In **Fig. 3** the composition of Gates is demonstrated considering the AndGate as an example. The strange connecor at the left hand site is an interim solution of painting vectors of connectors.

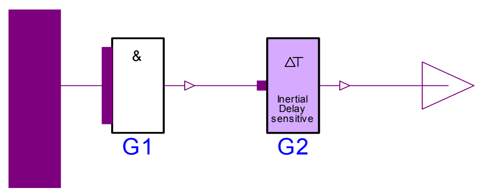


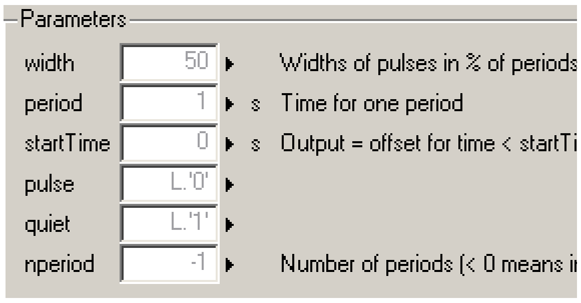
Figure 3: AndGate

3.5 Sources

The sources Set, Step, Table, Pulse, and Clock are not borrowed from the standard but written as nice-to-have sources. The *Set* source simply sets a logic value. *Step* steps one-time from one value to a second value at a given time. The *Table* source follows a user specified value-time-table. The essential part of the Modelica code of the Table model is (after checking the acceptance of parameters):

```
algorithm
  y := y0;
  for i in 1:n loop
    if time >= t[i] then
      y := x[i];
    end if;
  end for;
end Table;
```

With the *Pulse* source arbitrary pulsing between two values can be created. The Pulse parameters are shown in **Fig. 4**.



Parameters

width	50		Widths of pulses in % of periods
period	1	s	Time for one period
startTime	0	s	Output = offset for time < startTi
pulse	L.'0'		
quiet	L.'1'		
nperiod	-1		Number of periods (< 0 means i

Figure 4: Pulse Source Parameters

The *Clock* source is a simplified Pulse source without counting the number of periods which pulses between '0' and '1'. The code for generating the pulsing behaviour of Clock is:

```
  Modelica.SIunits.Time t_i
    (final start=startTime)
    "Start time of current period";
  Modelica.SIunits.Time
     t_width=period*width/100;
algorithm
  when sample(startTime, period) then
    t_i := time;
  end when;
  y := if time < startTime or
         time >= t_i + t_width
       then L.'0' else L.'1';
end Clock;
```

3.6 Converters

The IEEE 1164 like converters *LogicToUX01*, *LogicToX01Z*, and *LogicToX01* map the nine-valued digital logic to the limited sets of values {'U', 'X', '0', '1'}, {'X', '0', '1', 'Z'}, or {'X', '0', '1'} respectively. The mapping is done with conversion tables. E.g. the conversion table for LogicToX01 is:

```
input:   U  X  0  1  Z  W  L  H  -
output:  X  X  0  1  X  X  0  1  X
```

The following converters are not from the IEEE 1164 standard.

The *BooleanToLogic* converter maps the Boolean input to Logic according to the following table (t - true, f - false):

```
input:   t  f
output:  1  0
```

The LogicTo*Boolean* converter maps the Logic input to the Boolean output according to the following table (t - true, f - false):

```
input:   U  X  0  1  Z  W  L  H  -
output:  f  f  f  t  f  f  f  t  f
```

Further conversions are possible between Real and Logic values. In the *LogicToReal* converter the Real output jumps to a real number wich can be defined by the user for each of the nine logic values. The default values are:

```
input:    U  X  0  1  Z  W  L  H  -
output:  .5 .5  0  1 .5 .5  0  1 .5
```

The RealToLogic converter has two limits: an upper limit, and a lower limit. If the input x is x > upper limit, an upper_value is chosen, If x < lower limit, a lower value is chosen, otherwise the middle_value is chosen. The limits and the values are parameters of the converter. In Figure **Fig. 5** a sine curve is converted to logic using the default parameters (lower_limit=0, upper_limit=1, lower_value='0', upper_value='1', middle_value ='X').

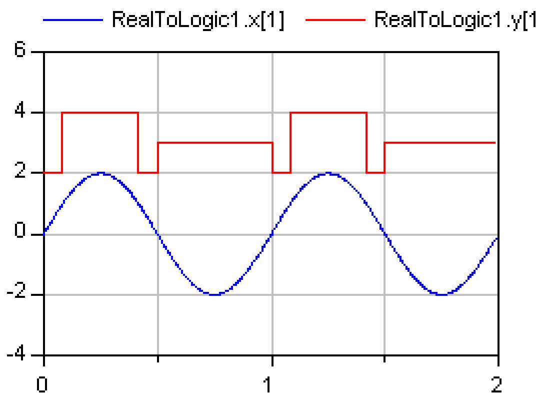


Figure 5: Default Real to Logic Conversion

4 Usage

The components of the electrical digital library can be combined to form more complex models. This is possible on the text level, or in a graphical way.
Since complex devices like flipflops, multiplexers, memories, ... are still missing, such components have to be composed using the set of basic gates. In the example package some of these components are available.
Furthermore, the user can modify the models by changing the description, adding pins, introducing parameters, fixing parameters...
The signal strengh according to the IEEE 1164 strength table is not modeled yet, since no resolution function is implemented. This will be added in a later version of that library.

Sometimes it is possible that algebraic loops occur which can be not solved by the simulator. In such cases often the inclusion of additional delay components helps.
The whole variety of the possibilities of the library usage is not presented. Some aspects of the library usage are demonstrated in the examples.

5 Examples

The examples are part of a validation suite, some of them are furthermore part of the library example subpackage. They show some of the possibilities of the library. Since the library is developed recently further tests e.g. with 'large' logic designs are necessary. Test examples were developed using wellknown textbooks [9], [10], [11]. All examples presented here were simulated using the simulator Dymola5.3a [12].

5.1 Logic Equivalence

This simple example tests the logic equivalence $AB \vee \bar{A}C \vee BC = AB \vee \bar{A}C$ by modeling both sides $X = AB \vee \bar{A}C \vee BC$ and $Y = AB \vee \bar{A}C$ of the logic equation with basic components.

The following Modelica text shows the circuit description without graphical instructions. The instantiation of library components can be seen as well as the usage of parameters. Once instantiated the devices are connected in the equation part.

```
model LogicEquivalence
  import DD=Modelica.Electrical.Digital;
  DD.Basic.And And1, And2, And3;
  DD.Basic.Or Or1, Or2(n=3);
  DD.Basic.Not Not;
  DD.Sources.Table TabB
   (x={3,3,4,4,3,3,3,3,3,4,4,4,4,3,3},
   t={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14});
  DD.Sources.Table TabA
   (x={3,3,4,4,4,4,4,3,3},
   t={0,1,2,3,4,5,6,7,8},);
  DD.Sources.Table TabC
   (x={3,3,4,3,3,4,3,3,3,3,4,3,3,3,4,3,3},
   t={0,1,2,3,4,5,6,7,8,9,10,11,
      12,13,14,15,16});
  DD.Interfaces.Logic X, Y;
equation
  connect(TabA.y, And1.x[2]);
  connect(TabA.y, Not.x);
  connect(TabB.y, And1.x[1]);
  connect(TabB.y, And3.x[2]);
  connect(TabC.y, And2.x[1]);
  connect(TabC.y, And3.x[1]);
  connect(And1.y, Or1.x[2]);
  connect(And1.y, Or2.x[3]);
  connect(And2.y, Or2.x[2]);
  connect(And2.y, Or1.x[1]);
  connect(AndB3.y, Or2.x[1]);
  connect(Not.y, And2.x[2]);
  X = Or2.y;  Y = Or1.y;
end LogicEquivalence;
```

More instructive is the graphical representation like **Fig. 6** which is normally used to model digital circuits:

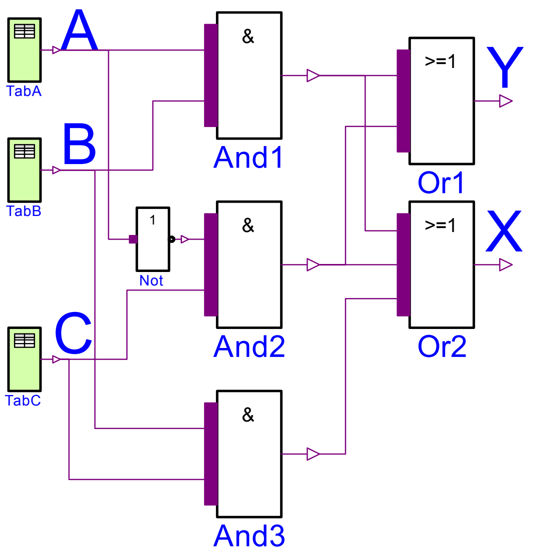


Figure 6: Logic Equivalence Circuit

The simulation result are the outputs X and Y of both Or components which are equivalent. Furthermore the input values of TabA, TabB, and TabC are shown which correspond to A, B, and C:

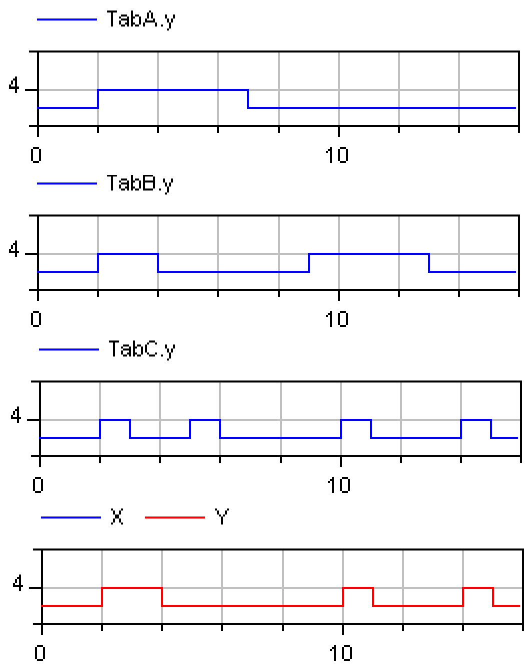


Figure 7: Simulation results of the equivalence circuit

5.2 Half-Adder

A half-adder can be found in the Examples.Utility package. It is composed according to **Fig. 8** using gates with a delay of 0.5s.

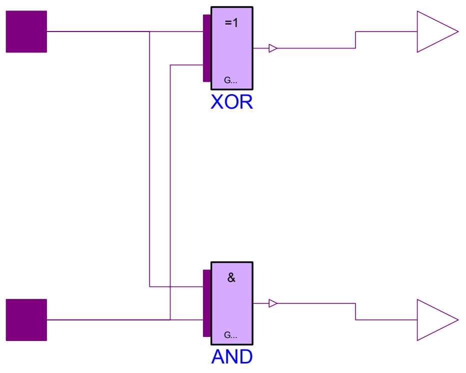


Figure 8: Half-Adder

Starting with 'Unknown' at the signal inputs a and b, and testing all combinations with '0' and '1' the behaviour is as expected, c.f. **Fig. 9**.

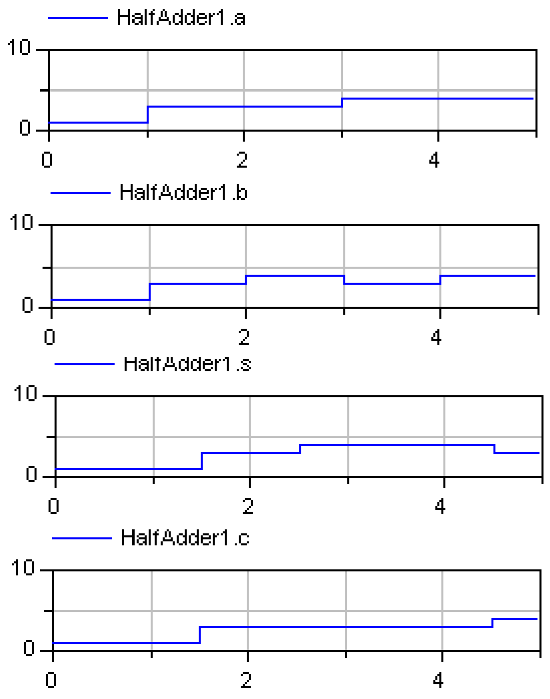


Figure 9: Half-Adder, Results

5.3 JK-Flip-Flop

A JK-Flip-Flop (with inputs j, k, and clock) is composed according to **Fig. 10**. It uses a static RS-Flip-Flop which is shown in **Fig. 11**. Both components are in the Examples.Utitlies package of the Digital library.

The results in **Fig. 12** show the behaviour of the JK-Flip-Flop: If J is '0' (coded by 3), the output q follows K, if both inputs are '1' (coded by 4), the output is clocked, if J is '1' and K is '0' the output becomes '0'.

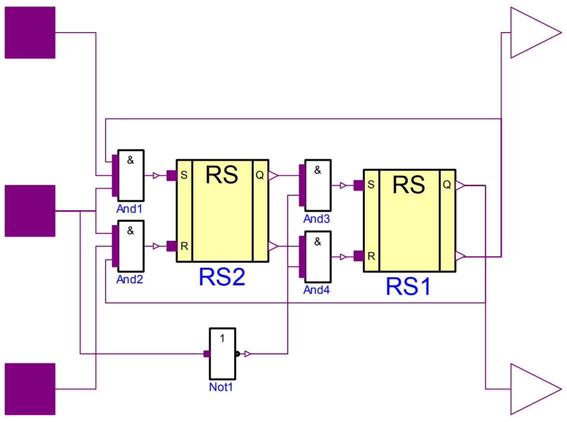


Figure 10: JK-Flip-Flop

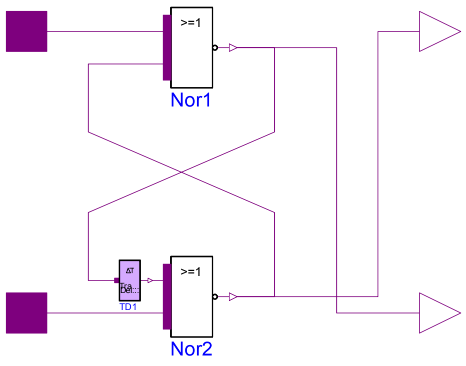


Figure 11: RS-Flip-Flop

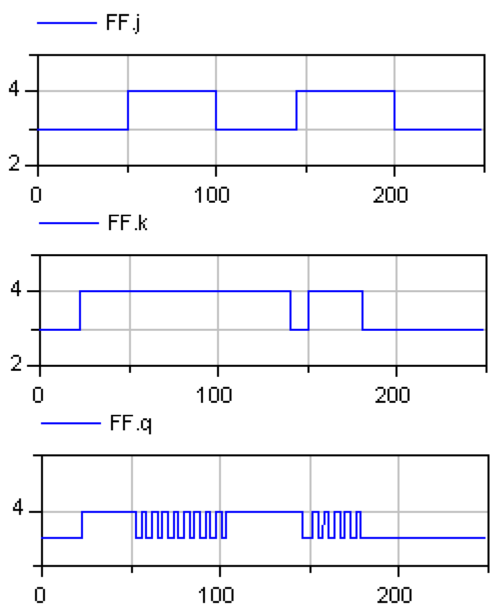


Figure 12: Results of the JK-Flip-Flop

The JK-Flip-Flop described in **Fig. 10** can be combined to a counter. Depending on the number of Flip-Flops the number of digits can be chosen. Figure **Fig. 13** shows the schematic of a three-bit-counter.

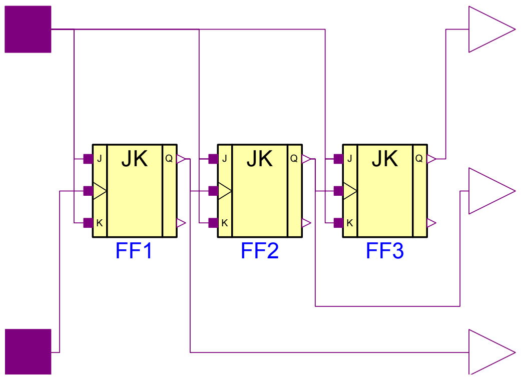


Figure 13: Three-Bit-Counter

Within the schematic of **Fig. 14** the three-bit-counter output is taken as input of a full-adder. The full-adder sums up the three outputs of the counter. The result can be found in **Fig. 15**.

5.4 Adder with Counter

Two half-adders described in 5.2 and an or gate can be combined to a full-adder which is able to add two digits including a carry bit from a preceding full-adder. In **Fig. 12** the schematic of the full-adder is shown.

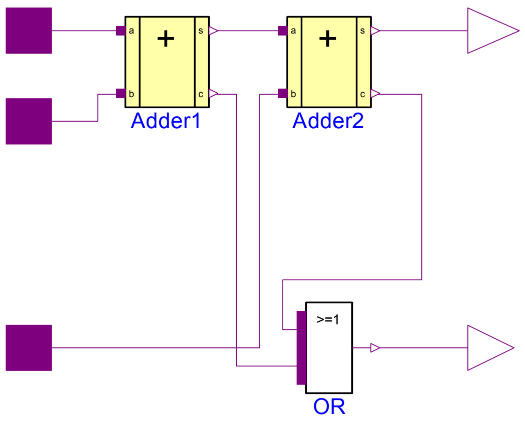


Figure 12: Full-Adder Schematic

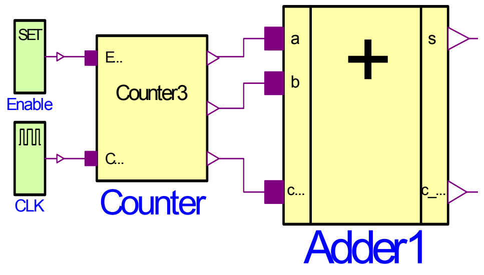


Figure 14: Counter with Adder

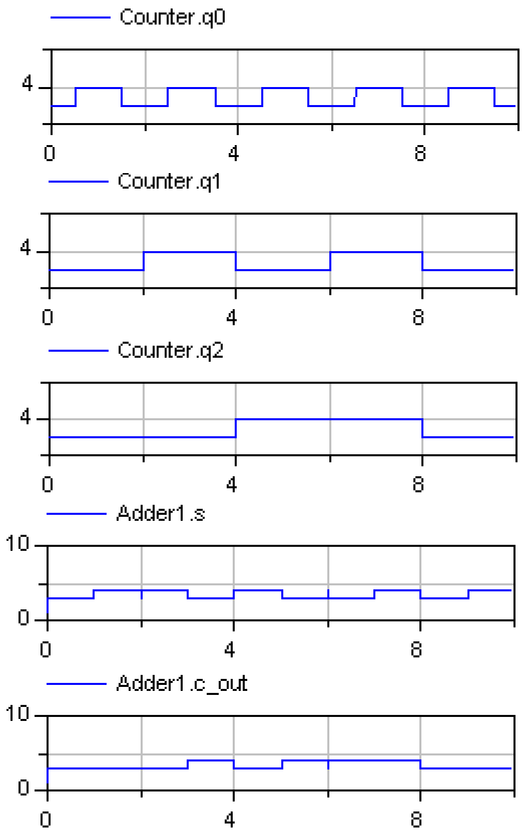


Figure 15: Results of the Counter with Adder

6 Summary

The digital electric library presented is part of the Modelica standard library. In this paper the devices and their principles of implementation are explained. Some examples show the usage of this library.

Although tested during development a wide usage is desirable to get extensive experiences. Especially large designes are needed as well as mixed applications with other physical domains.

Once the first version of the digital library is accepted it will be extended by behavioural models of flip-flops, latches, transfer gates, tristate devices, multiplexers, and memories. A discussion on principles will be expected concerning the introduction of a resolution function at general nodes.

This work was supported by the Sonderforschungsbereich 358, Teilprojekt D4 of the Deutsche Forschungsgemeinschaft. We are grateful to Dr. Peter Schwarz, and Dr. Martin Otter for encouraging this work, and to our students Teresa Schlegel, Liane Jacobi, and Hagen Staemmler for implementing and testing.

7 References

[1] Elmqvist, H. et al.: Modelica - A Unified Object-Oriented Language for Physical Systems Modeling. Version 2.1, January 2004. http://www.Modelica.org

[2] Otter, M.; Elmqvist, H.; Mattsson, S.E.: Objektorientierte Modellierung physikalischer Systeme, Teil 8. at Automatisierungstechnik 47(1999)9

[3] IEEE Standard Multivalue Logic System for VHDL Model Interoperability. http://www.ieee.org

[4] IEEE 1076-1993: IEEE Standard VHDL Language Reference Manual (ANSI). 288 p. ISBN 1-55937-376-8. IEEE Ref. SH16840-NYF.

[5] IEEE 1164-1993: IEEE Standard Multivalue Logic System for VHDL Model Interoperability (Std_logic_1164). 24 p. ISBN 1-55937-299-0. IEEE Ref. SH16097-NYF.

[6] Lipsett, R.; Schaefer, C.; Ussery, C.: VHDL: Hardware Description and Design. Boston: Kluwer, 1989, 299 p. ISBN 079239030X.

[7] Navabi, Z: VHDL: Analysis and Modeling of Digital Systems. New York: McGraw-Hill, 1993, 375 p. ISBN 0070464723.

[8] Normen über graphische Symbole für die Elektrotechnik, Schaltzeichen. DIN-Taschenbuch 514, Beuth Berlin, Wien, Zürich, 1994

[9] Ashenden, P. J.: The Designer's Guide to VHDL. San Francisco: Morgan Kaufmann, 1995, 688 p. ISBN 1-55860-270-4.

[10] Horowitz, P.; Hill, W.: The Art of Electronics. Cambridge University Press, 1989, ISBN 0-521-37095-7

[11] Tietze, U.; Schenk, C.: Halbleiter-Schaltungstechnik. Springer-Verlag Berlin, Heidelberg, New York, 1980, ISBN 3-540-09848-8

[12] Dymola: http://www.Dynasim.se

Modelica libraries for dc machines, three phase and polyphase machines

C. Kral, A. Haumer
Arsenal Research, Faradaygasse 3, 1030 Vienna, Austria

Abstract

This paper presents two libraries, the basic *Machines* library which is released with Modelica standard library 2.1 as well as an *ExtendedMachines* library, both for modelling electric machines. The basic library provides the basic machines types such as dc machines, three phase induction machines, three phase permanent magnet synchronous machines and three phase reluctance machines. The three phase machine models are implemented on basis of space phasor theory. By contrast, the extended machines library models the winding topology of polyphase induction and permanent magnet synchronous machines. Such machine models can be used to simulate machines with arbitrary phase number such as large six phase induction machines or machines with winding asymmetries or even winding faults. The used winding models of the *ExtendedMachines* library are flexible enough to consider even higher field harmonics in future impementations.

1 General

Each of the presented machine models considers linear inductors. This means that neither saturation effects nor skin effects are considered yet. Iron losses such as eddy current and hysteresis losses are not taken into account. All resistive parameters are assumed to be constant, therefore machines are modeled without thermal behavior. Friction and ventilation losses are not considered. For the basic *Machines* library symmetrical three phase windings are assumed for induction machines, synchronous machines and reluctance machines. The extended library supports an arbitrary number of stator and rotor phases including possible asymmetries.
Any leakage inductances and ohmic resistors of the windings are modelled as discrete elements. These elements are connected between machine terminals (*MultiPhase* plugs or regular pins) and the air gap model.
All quantities accessible at the electrical and mechanical connectors are shown in physical units, not as p.u. values. This allows the coexistence of different machines in a complex system model.

2 Basic Machines Library

The basic elements of the *Machines* libraries are the air gap models. These elements model the magnetic main flux in the air gap including induced voltages and the electromagnetic torque generation. This torque is directed to a mechanical `flange` connector representing the shaft as well as to a second mechanical `support` connector, representing the housing respectively the machine legs or flange where the reaction torque is noticeable. The airgap's `flange` is connected to the rotor's inertia, which is connected to the machine's `flange` (i.e. shaft). The airgap's `support` now is implicitly fixed but allows to add a `support` connector for the whole machine in the next release. Adding a `support` connector for the whole machine, besides the rotor's inertia also the stator's inertia will be needed as a parameter.
The Basic *Machines* library is structured as follows:

- `BasicMachines.AsynchronousInduction-Machines` containing machine models
- `BasicMachines.SynchronousInduction-Machines` containing machine models
- `BasicMachines.DCMachines` containing machine models
- `BasicMachines.Components` contains elements like air gaps, squirrel cage and permanent magnet

- `Sensors` provides voltage and current RMS sensors (using space phasor functions), electrical and mechanical power sensor, as well as a sensor calculating the phase angle between the rotor and the rotating field. Voltage and current RMS sensors actually calculate the $\frac{1}{\sqrt{2}}$-fold of the magnitude of the current and voltage space phasor, respectively. The outputs of these sensors equal the RMS values of the currents and voltages for sinusoidal and stationary operation (only).

- `SpacePhasors.Components` provides the basic transformation element between three phase instantaneous voltages and currents and their space phasor representation, including zero sequence system

- `SpacePhasors.Blocks` implements space phasor transformation blocks to support the design of controllers

- `SpacePhasors.Functions` provides the same functionality as blocks but implemented as functions for use in initial equations

- `Interfaces` defines the space phasor connector as described later as well as some partial machine models, defining the common mechanical connectors

- `Examples` demonstrate the usage of the machine models as well as some utilities (used for the examples)

All machine models ensure correct initialization and efficient integration by using appropriate `stateSelect`-modifiers.

2.1 DC Machines

For the dc machines an air gap model is used which evaluates the (perpendicular) two axis equations of the armature and field excitation circuit. The air gap model has two electric pins, for the armature and the field excitation circuit each. The mechanical connectors `flange` and `support` are connected with the rotor's inertia and the implicitly fixed housing, respectively. The equations of the air gap model considers

- the induced voltage of the armature (electromotive force and emf $v_{a.i}$, respectively) due to magnetic field and rotor movement,

- the voltage drop of the excitation winding due to the time derivative of the magnetic main flux, and

- the electromagnetic torque which is directed to `flange` and `support`.

The armature voltage equation reads:

$$v_a = R_a i_a + L_a \frac{\mathrm{d}i_a}{\mathrm{d}t} + v_{a.i} \tag{1}$$

where $R_a i_a$ and $L_a \frac{\mathrm{d}i_a}{\mathrm{d}t}$ are modelled as discrete elements and $v_{a.i}$ is calculated by the air gap model. The excitation voltage equation reads:

$$v_e = R_e i_e + v_{e.i} \tag{2}$$

where $R_e i_e$ is modelled as discrete element and the induced (inner) voltage $v_{e.i}$ is calculated by the air gap model. Excitation voltage equation (2) is used for electrically excited dc machines, only.
The air gap model uses the following voltage equations:

$$v_{a,i} = \texttt{TurnsRatio} \cdot \psi_e \omega \tag{3}$$

$$v_{e,i} = \frac{\mathrm{d}\psi_e}{\mathrm{d}t} \tag{4}$$

as well as the flux linkage equation:

$$\psi_e = L_e i_e \tag{5}$$

and the torque equation:

$$\tau = \texttt{TurnsRatio} \cdot \psi_e i_a \tag{6}$$

Parameter `TurnsRatio` (between armature and excitation winding) is calculated internally from rating plate values. For an electrically excited dc machine the defining equation is:

$$\begin{aligned} v_{a,\mathrm{nom}} - R_a i_{a,\mathrm{nom}} =& \texttt{TurnsRatio} \\ & \cdot (L_e i_{e,\mathrm{nom}})\, \omega_{\mathrm{nom}} \end{aligned} \tag{7}$$

Therefore the electrical parameters of the dc machine models are the armature resistance R_a and field excitation resistance R_e (except permanent magnet machine), armature inductance L_a, field inductance L_e (except permanent magnet machine) as well as nominal values from the rating plate: $v_{a,\mathrm{nom}}$, $i_{a,\mathrm{nom}}$, ω_{nom} and $i_{e,\mathrm{nom}}$ (only for electrically excited dc machines).
The *Machines* library provides the following types of dc machines:

- `DC_ElectricalExcited`: model of an electrically excited dc machine which can either be connected as separate excited or shunt excited machine.
- `DC_SeriesExcited`: model of an electrically excited dc machine with series excitation; the only to difference to `DC_ElectricalExcited` is a modified assignment of parameters due to different rating plate data.
- `DC_PermanentExcited`: the magnetic field is provided by a permanent magnet which is modelled internally as a constant current source in the field excitation circuit.

2.2 Induction Machines

All three phase machines of the basic *Machines* library rely on space phasor theory [1]. For each system of three instantaneous voltages v_1, v_2 and v_3 a space phasor (indicated by underline) can be defined:

$$\underline{v} = \frac{2}{3}(v_1 + e^{\mathrm{j}2\pi/3}v_2 + e^{-\mathrm{j}2\pi/3}v_3) \tag{8}$$

The factor $\frac{2}{3}$ in this equation is an arbitrary factor which is due to normalization reasons. Similar to (8) a space phasor of three instantaneous currents can be defined. The current space phasor can be interpreted as a vector representing the fundamental harmonic of the magnetomotive force (mmf) of the three phase currents. The direction of the phasor represents the phase angle, the length of the phasor represents the peak value of the fundamental mmf.

Therefore a space phasor connector is defined in the library. Both the voltage and the current space phasors are combined in the `SpacePhasor` connector. Real and imaginary part of voltage and current are stored in two elements of an array, respectively.

Space phasor theory is not restricted to any waveforms of voltages and currents in the time domain. Due to a given winding topology mainly the fundamental space harmonics of the electromagnetic quantities such as flux density and the mmf are physically present [2]. The harmonics of the electromagnetic quantities refer to the spatial domain along the tangential direction in the air gap. The restriction of space phasor theory is, that it only takes the fundamentals of these quantities into account. Higher harmonic components cannot be considered.

The mathematically correct formalism of space phasor theory requires a third transformed quantity, because the three original quantities (index 1,2 and 3) have to be linearly transformed into three transformed quantities. The third quantity is the zero sequence component:

$$v_0 = \frac{1}{3}(v_1 + v_2 + v_3) \tag{9}$$

Any zero sequence quantity does not contribute to the fundamental of the main field due to the symmetry of the windings.

From (8) and (9) we determine the equations for back transformation:

$$v_1 = v_0 + \mathrm{Re}\left(\underline{v}\right) \tag{10}$$

$$v_2 = v_0 + \mathrm{Re}\left(e^{-\mathrm{j}2\pi/3}\underline{v}\right) \tag{11}$$

$$v_3 = v_0 + \mathrm{Re}\left(e^{\mathrm{j}2\pi/3}\underline{v}\right) \tag{12}$$

Transformations (8) and (9) and (10)–(12) are modelled in element `SpacePhasors.Components.SpacePhasor`.

To describe the machine's behavior, first we need an equation for the stator voltages of the three phases ($i \in [1, 2, 3]$):

$$v_{Si} = R_S i_{Si} + \frac{\mathrm{d}\psi_{Si}}{\mathrm{d}t} \tag{13}$$

Applying transformation (8) – neglecting the zero sequence system at the moment – on (13), we obtain the stator voltage space phasor equation in a stator fixed reference (coordinate) system (indicated by index (S)):

$$\underline{v}_{S(S)} = R_S \underline{i}_{S(S)} + \frac{\mathrm{d}\underline{\psi}_{S(S)}}{\mathrm{d}t} \tag{14}$$

It is possible to split the stator flux linkage $\underline{\psi}_{S(S)}$ into main flux $\underline{\psi}_{m(S)}$ and leakage flux:

$$\underline{\psi}_{S\sigma(S)} = L_{s\sigma}\underline{i}_{S(S)} \tag{15}$$

This leads to:

$$\underline{v}_{S(S)} = R_S \underline{i}_{S(S)} + L_{S\sigma}\frac{\mathrm{d}\underline{i}_{S(S)}}{\mathrm{d}t} + \frac{\mathrm{d}\underline{\psi}_{m(S)}}{\mathrm{d}t} \tag{16}$$

Similarly we obtain the rotor voltage equation using space phasors in a fixed rotor reference (coordinate) system (indicated by index (R)):

$$\underline{v}_{R(R)} = R_R \underline{i}_{R(R)} + L_{R\sigma}\frac{\mathrm{d}\underline{i}_{R(R)}}{\mathrm{d}t} + \frac{\mathrm{d}\underline{\psi}_{m(R)}}{\mathrm{d}t} \tag{17}$$

The main flux linkage depends on the sum of stator and rotor phase current, formulated in a common reference (coordinate) system:

$$\underline{\psi}_m = L_m \left(\underline{i}_S + \underline{i}_R\right) \tag{18}$$

At last we need an equation for the electromagnetic torque; this may be done by setting up power balance:

$$\begin{aligned} p_{el} &= v_{S1}i_{S1} + v_{S2}i_{S2} + v_{S3}i_{S3} \\ &\quad + v_{R1}i_{R1} + v_{R2}i_{R2} + v_{R3}i_{R3} \end{aligned} \tag{19}$$

After some transformations we obtain:

$$p_m = m_{el}\omega_m \tag{20}$$

$$\tau_{el} = \frac{3}{2} p \,\mathrm{Im}\left(\underline{i}_S \underline{\psi}_m^*\right) \tag{21}$$

In the latter equation p represents the number of pole pairs. This equation is valid in any arbitrary reference frame. To avoid transformation of (16) to the rotor fixed reference frame, or, alternatively to transform (17) to the stator fixed reference frame – obeying the product rule of differentiation – the ohmic resistors R_S and R_R as well as the leakage inductances $L_{S\sigma}$ and $L_{R\sigma}$ are modelled as discrete three phase elements outside the air gap. The next element of the modelled voltage equation transforms the three phase system into space phasors which are connected with the air gap model (fig. 1).
The air gap model using the stator fixed reference frame `BasicMachines.Components.AirgapS` transforms the rotor current space phasor $\underline{i}_{R(R)}$ to the stator fixed reference frame and calculates main flux linkage $\underline{\psi}_{m(S)}$ from (18). Then it is possible to calculate $\frac{\mathrm{d}\underline{\psi}_{m(S)}}{\mathrm{d}t}$ and – after the transformation of $\underline{\psi}_{m(S)}$ to the rotor fixed reference frame – the term $\frac{\mathrm{d}\underline{\psi}_{m(R)}}{\mathrm{d}t}$.
The air gap model using the rotor fixed reference frame `BasicMachines.Components.AirgapR` transforms the stator current space phasor $\underline{i}_{S(S)}$ to the rotor fixed reference frame and calculates main flux linkage $\underline{\psi}_{m(R)}$ from (18). Then it is possible to calculate $\frac{\mathrm{d}\underline{\psi}_{m(R)}}{\mathrm{d}t}$ and – after transformation of $\underline{\psi}_{m(R)}$ to the stator fixed reference frame – the term $\frac{\mathrm{d}\underline{\psi}_{m(S)}}{\mathrm{d}t}$.
Additionally, both air gap models calculate the electromagnetic torque from (21).

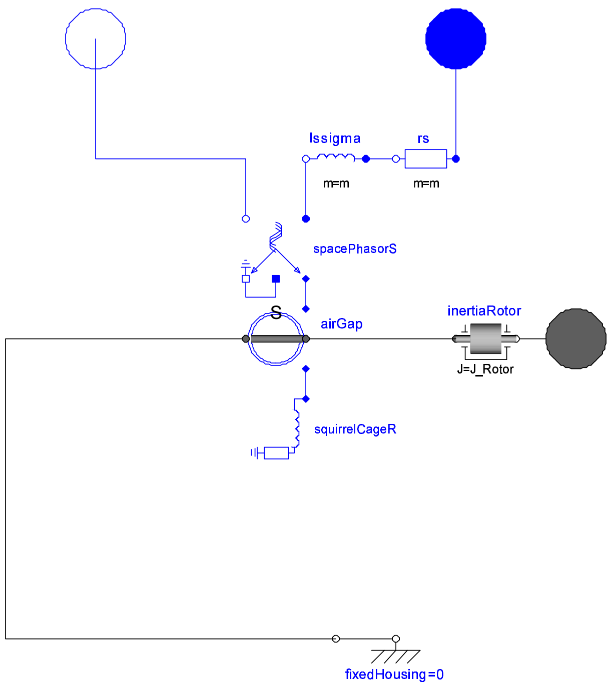


Figure 1: Asynchronous induction machine with squirrel cage; basic *Machines* library

The angular displacement between the rotor and the stator reference frame is:

$$\gamma = \int \omega_m dt \tag{22}$$

The transformation of a space phasor from a rotor fixed coordinate system to a stator fixed reference frame is performed by:

$$\underline{\psi}_{m(S)} = \underline{\psi}_{m(R)} \mathrm{e}^{\mathrm{j}\gamma} \tag{23}$$

The inverse transformation therefore is:

$$\underline{\psi}_{m(R)} = \underline{\psi}_{m(S)} \mathrm{e}^{-\mathrm{j}\gamma} \tag{24}$$

Although the zero system quantities do not contribute to the fundamental of the main field, they may give rise to additional leakage flux linkage components, though. Therefore, zero sequence connector `zero` is also provided in the `SpacePhasor` transformation model. If the zero sequence component connector is grounded, which is the regular case, no additional leakage flux linkages are taken into account [3].
Standard machine parameters are stator and rotor resistance R_s and R_r, stator and rotor leakage inductance $L_{s\sigma}$ and $L_{r\sigma}$, the main inductance L_m as well as the number of pole pairs p and rotor

inertia J. The following models of asynchronous induction machines are provided in the *Machines* library:

- `AIM_SquirrelCage` (asynchronous induction machine with squirrel cage): The squirrel cage is modelled as equivalent two axis winding model in the rotor circuit; the rotor is not accessible.

- `AIM_SlipRing` (asynchronous induction machine with wound rotor winding and slip rings): The rotor is equipped with a symmetrical rotor winding; this winding topology requires an additional parameter `TurnsRatio` which represents the effective ratio of stator to rotor turns of the respective three phase windings.

2.3 Permanent Magnet Synchronous Machine

The permanent magnet synchronous machine has the same stator winding topology as an asynchronous induction machine. Since the developed models are equipped with a damper winding, rotor winding topology is comparable with a squirrel cage induction machine. Synchronous machines without damper cage have to have control in order to work stable; they are therefore not provided in the current basic *Machines* library.
Saliencies of the rotor are considered through different main field inductances in the d- and q-axis (direct and quadrature axis) L_{md} and L_{mq}. We have to use the rotor fixed reference frame to consider these saliencies correctly.
Currently there are only two models provided:

- `SM_PermanentMagnetDamperCage`: The permanent magnet is modelled by means of a superimposed constant current source in the direct axis.

- `SM_ReluctanceRotorDamperCage` (synchronous induction machine with reluctance rotor): The rotor has a squirrel cage; electromagnetic torque in synchronism is generated due to saliencies of the rotor only.

3 Extended Machines Library

The *ExtendedMachines* library models are not restricted to three phases and symmetrical windings.

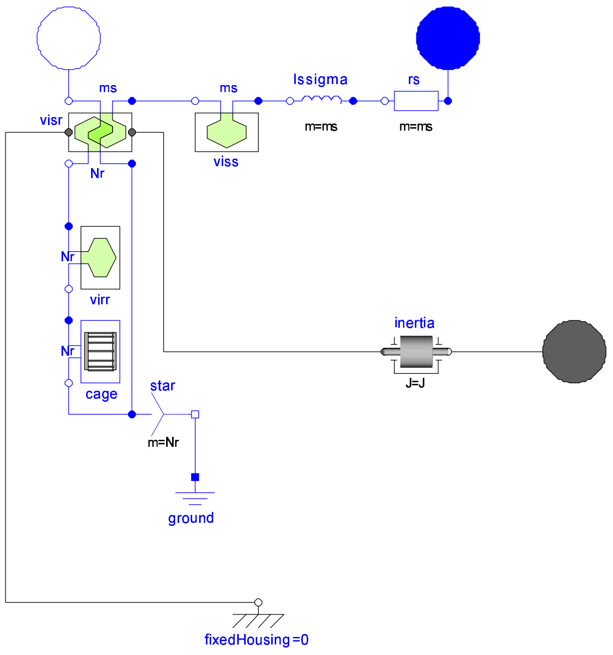


Figure 2: Extended model of a squirrel cage induction machine

The focus of this library is polyphase asynchronous and synchronous induction machines. The provided types of machines are squirrel cage and slip ring asynchronous induction machines as well as permanent magnet synchronous machines. For each of these models two implementations are provided. The *symmetrical* implementation assumes symmetrical windings in the stator and rotor of the polyphase induction machines. The *winding topology* implementation models the topology of each stator and rotor winding including possible asymmetries and winding faults. For these reasons, space phasor theory is not applicable to the *ExtendedMachines* library models any more.

3.1 Voltage Equations

It is assumed that neither the number of phases of the stator m_s nor the number of phases of the rotor m_r is restricted to three.

The voltage equations of the machines are modelled graphically. The used models which represent the partial voltage drops are based on equations, though. Stator voltage equation for each of the $i \in [1, 2, ..., m_s]$ stator phases is:

$$v_{Si} = R_{Si} i_{Si} + L_{S\sigma i} \frac{\mathrm{d} i_{Si}}{\mathrm{d}t} + \sum_{j=1}^{m_S} L_{Si,Sj} \frac{\mathrm{d} i_{Sj}}{\mathrm{d}t} + \frac{\mathrm{d}}{\mathrm{d}t} \sum_{j=1}^{m_R} L_{Si,Rj} i_{Rj} \tag{25}$$

In this equation $L_{Si,Sj}$ represents the mutual inductance between the stator phases i and j. For $i = j$ this term equals the self inductance of stator phase i. The voltage drops across the inductances $L_{Si,Sj}$ are modeled through `VoltageEquations.SelfInductance`. The syllable `Self` was chosen due to the fact the linkage refers to only one machine side (either stator or rotor), although mutual linkages are taken into account. The instance `viss` of such model is implemented in the graphical model of the squirrel cage induction machine, which shown in fig. 2.

The mutual inductance $L_{Si,Rj}$ between a stator phase i and a rotor phase j is dependent of the rotor angle (22). The voltage drops across this inductances with respect to the stator and rotor side (linkage) are processed in the instant `visr` (fig. 2) of `VoltageEquation.MutualInductance`. This model computes torque and is connected with the mechanical shaft (flange) and the support which is currently fixed. Once the basic *Machines* library provides a support connector of the machine, the *ExtendedMachines* library will provide such a connector, too.

Rotor voltage equation has basically the same structure as (25):

$$v_{Ri} = R_{Ri} i_{Ri} + L_{R\sigma i} \frac{\mathrm{d} i_{Ri}}{\mathrm{d}t} + \sum_{j=1}^{m_R} L_{Ri,Rj} \frac{\mathrm{d} i_{Rj}}{\mathrm{d}t} + \frac{\mathrm{d}}{\mathrm{d}t} \sum_{j=1}^{m_S} L_{Ri,Sj} i_{Sj} \tag{26}$$

This equations have to be applied to each phase of the rotor winding. The voltage drop across mutual inductances is processed in instant `visr`, the voltage drops across the mutual and self inductances $L_{Ri,Rj}$ of the rotor side are processed in `virr`, which is also an instant of `VoltageEquations.SelfInductance`.

The air gap model of the basic *Machines* library is equivalent to the models `viss`, `visr` and `virr` in fig. 2.

For a slip ring induction machine the voltage drop across the rotor resistances R_{Ri} and the rotor leakage inductances $L_{R\sigma i}$ can be modeled in accordance with the stator model. For the squirrel cage asynchronous induction machine of fig. 2 these voltage drops are replaced by a sophisticated model of the cage with $m_r = N_r$ rotor bars. This model takes the connection of the bars and end rings of the real squirrel cage into account.

The effort of the *ExtendedMachines* models is the determination of parameters. In accordance to [4] each inductance of the stator side is designed in the fashion of:

$$L_{Si,Sj} = L w_{Si} w_{Sj} \xi_{pSi} \xi_{pSj} \operatorname{Re}(\xi_{zSi} \xi_{zSj}^*) \tag{27}$$

In this equation L represents a base inductance, w_{Si} is the number of turns of phase i, ξ_{pSi} is the pitch factor of phase i and ξ_{zSi} is a complex winding factor which considers the topology of phase winding i with respect the stator reference frame. The inductances of the rotor side are based on equivalent equations. The mutual inductances of the stator and rotor inductances have to take the rotor angle (22) into account:

$$L_{Si,Rj} = L w_{Si} w_{Rj} \xi_{pSi} \xi_{pRj} \operatorname{Re}(\xi_{zSi} \xi_{zRj}^* \mathrm{e}^{-\mathrm{j}\gamma}) \tag{28}$$

The only difference between this equations and (27) is the rotational phasor $\mathrm{e}^{-\mathrm{j}\gamma}$. The inductances of (28) depend on the rotor angle (22) and are therefore time dependent. Rotor skewing is not considered. In (28) the rotor or stator skewing is not considered.

So far only the first order space harmonics waves (fundamental) are taken into account. The machine models are going to be extended for higher space harmonics; this will be implemented in the near future.

3.2 Symmetrical Winding Models

The pitch factors of the symmetrical winding models are set to one without having any restrictions. The symmetrical topology is considered by

$$\xi_{zSi} = \mathrm{e}^{-\mathrm{j}2\pi i/m_S}, \tag{29}$$

the product of the number of turns and the base inductance is derived from the magnetizing inductance L_m which is a (symmetrical) machine parameter.

The implemented models with respect to a symmetric winding are:

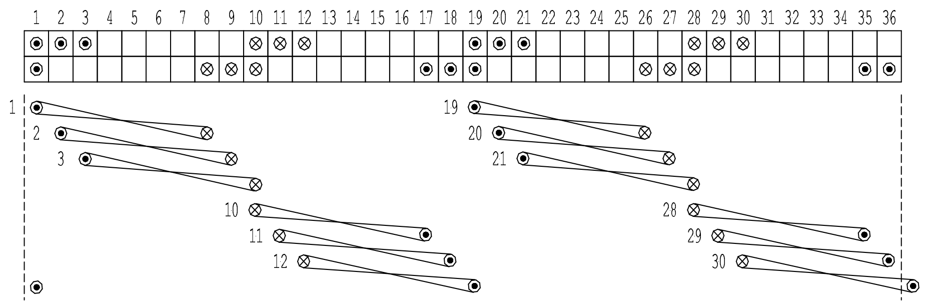


Figure 3: Topology of 7/9-pitch symmetrical three phase stator winding, phase 1

- `AIM_SlipRing`: Same parameters as basic *Machines* library; arbitrary numbers of stator and rotor phases are supported; symmetric winding

- `AIM_SquirrelCage`: Same parameters as basic *Machines* library; arbitrary number of stator phases and rotor bars is provided; symmetric stator winding; rotor may be modelled asymmetrically by overwriting rotor parameters

- `SM_PermanentMagnetDamperCage`: Same parameters as `SM_Permanent-MagnetDamperCage` of the basic *Machines* library; arbitrary number of stator phases is supported; stator inductances are not constant due to the saliency of the rotor, which is considered; the rotor is equipped with a damper cage with an arbitrary number of bars; damper cage may be modelled asymmetrically by overwriting rotor parameters

- `SM_PermanentMagnet`: Same parameters as `SM_PermanentMagnetDamperCage`, without parameters of the damper cage, though

3.3 Winding Topology Models

The winding topology models can handle polyphase machines and even stator asymmetries and stator faults. For now, the only assumption with respect to winding topologies is, that the coil pitch of each winding has to be equal. Further developments will consider coils with unequal coil pitches, too. However, the coil pitches of stator and rotor windings do not have to be equal, though.

Structure wise there is no difference between the symmetrical implementation the topology implementation. Just the handling and pre-calculation of machine parameters is different. An example of the topology of one phase winding (phase 1) of a symmetrical three phase machine is shown in fig. 3. The location of each coil of the phases in terms of multiples of stator slots is indicated by matrix $y_S[i,j]$. The first index i indicates the phase, the second index j represents the of the number of the coil. The first row (phase 1) of this matrix with respect to the depicted winding in fig. 3 is:

$$\begin{aligned} y_S[1,:] = [&1,2,3,19,20,21,10,\\ &11,12,28,29,30] \end{aligned}$$

Since the orientations of the coils are not equal, matrix e is defined, which defines the orientation of each coil. In our example the first row (phase 1) of this matrix is:

$$\begin{aligned} e[1,:] = [&+1,+1,+1,+1,+1,+1,\\ &-1,-1,-1,-1,-1,-1] \end{aligned}$$

The orientation $e[i,j]$ of a certain coil corresponds with the location $y_s[i,j]$ and the respective number of turns (each coil has 12 turns):

$$\begin{aligned} w_{cS}[1,:] = [&12,12,12,12,12,12,\\ &12,12,12,12,12,12] \end{aligned}$$

If the magnetic field caused by any coil is oriented up, the respective element in e equals +1, otherwise it equals −1. Parameter $y_{cs} = 7$ means that the coil pitch is seven slots. The total number of stator slots N_s is a model parameter, too.

For phase i the winding factors are:

$$\xi_{cSi} = \sin(p\frac{y_{cS} m_s}{N_s}) \tag{30}$$

$$\xi_{zSi} = \frac{1}{w_{Si}} \sum_j e[i,j] \mathrm{e}^{-\mathrm{j}p y_S[i,j] \frac{2\pi}{N_s}} \tag{31}$$

The number of turns of $i \in [1, 2, ..., m_s]$ is:

$$w_{Si} = \sum_j w_{cS}[i,j] \tag{32}$$

Depending on the machine type, winding factors and the numbers of turns for the polyphase stator and rotor winding are derived in the initial sections of the machine models. These parameters are passed to the graphically modelled voltage equations as shown in fig. 2.
Currently the following machines with winding topology are modelled:

- `AIM_SlipRingWinding`: Same parameters as `AIM_SlipRing`; additionally, winding topology of stator and rotor winding has to be provided through data vectors which determine the exact orientation and arrangement of each coil.

- `AIM_SquirrelCageWinding`: Same parameters as `AIM_SquirrelCage`; additionally winding topology of stator has to be provided through data vectors which determine the exact orientation and arrangement of each coil; rotor topology has to be provided by the number of bars and the ratio of bar to end ring resistance as well as the ratio of bar to end ring leakage reactance; a symmetrical cage is predefined, certain parameters can be overwritten in the parameter window, though.

- `SM_PermanentMagnetWinding`: Same parameters as `SM_PermanentMagnet` of the basic *Machines* library; arbitrary numbers of stator phases are supported; stator inductances are not constant due to the saliency of the rotor, which is considered; additionally winding topology of stator has to be provided through data vectors which determine the exact orientation and arrangement of each coil; the machine has no damper cage.

- `SM_PermanentMagnetDamperCageWinding`: Same parameters as `SM_PermanentMagnetWinding`; arbitrary numbers of stator and rotor phases are supported; additionally winding topology of stator has to be provided through data vectors which determine the exact orientation and arrangement of each coil; the machine has a damper cage; a symmetrical cage is predefined, certain parameters can be overwritten in the parameter window, though, in order to simulate an asymmetrical damper cage.

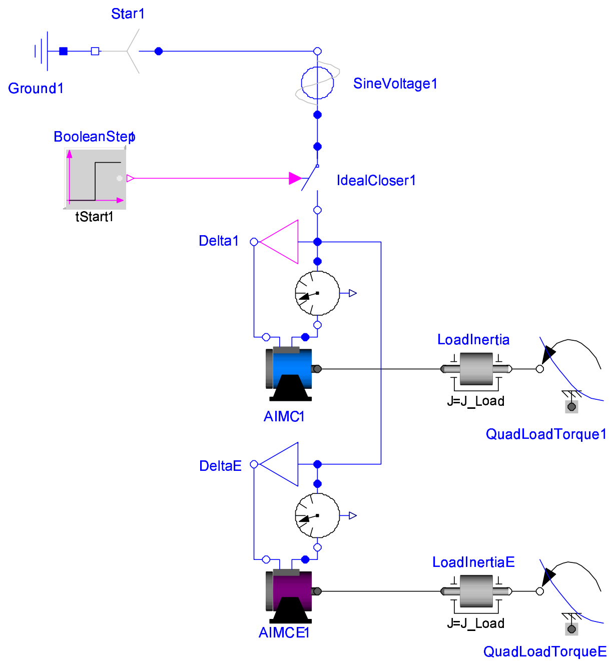


Figure 4: Direct start on line of squirrel cage induction machines; AIMC1 from basic *Machines* library, AIMCE1 from *ExtendedMachines* library

4 Example

To show the usage of the libraries, we will demonstrate simulation of starting an asynchronous induction machine with squirrel cage direct on line. Figure 4 presents the model using the basic *Machines* library as well as the *ExtendedMachines* library:
Each `AIM_SquirrelCage`-model connected in delta is electrically connected through one three phase switch to a three phase sinusoidal voltage source from the `MultiPhase` library. Mechanically each machine is connected to an individual load, consisting of an inertia and a load torque which is quadratic dependent on speed. Standard machine parameters are used. Load inertia is the same

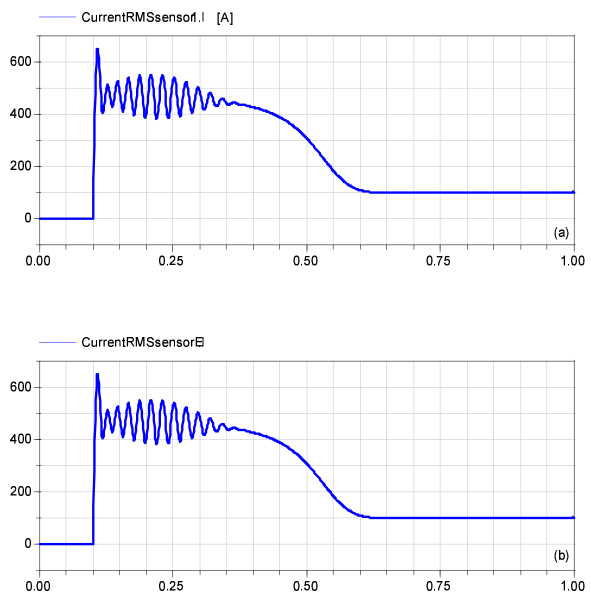


Figure 5: $\frac{1}{\sqrt{2}}$-fold of the magnitude of the current space phasor of the modelled squirrel cage machines (a) *Machines* library, (b) *ExtendedMachines* library

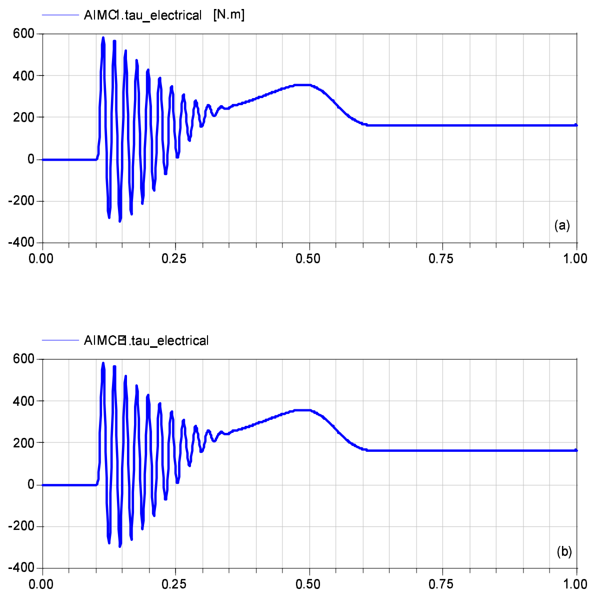


Figure 6: Electrical torque of the modelled squirrel cage machines (a) *Machines* library, (b) *ExtendedMachines* library

as the machine's inertia, and nominal load torque and speed are the same as the machine's rated values. After 0.1 seconds the switch is closed, high starting currents build up the magnetic field and each motor is able to develop starting torque, accelerating the inertias. The extended machines models was simulated with symmetrical stator windings and 28 rotor bars. The resulting currents of each machine are shown in fig. 5, the electrical torques are shown in fig. 6. These results demonstrate that both machine models have the same operational behavior.

5 Conclusions

The structure and basic equations of the basic *Machines* library using space phasors as well as the *ExtendedMachines* were presented. The *ExtendedMachines* library deals also with polyphase machines with arbitrary numbers of stator and rotor phases as well as asymmetrical or even faulty windings. The simulation results of both models were compared and match qualitatively and quantitatively.
Further developments will focus on:

- Implementation of a mechanical support connector representing the housing
- Additional machine types like electrically excited synchronous machines
- Modelling of iron, friction and stray losses
- Modelling of saturation effects of main and leakage inductances
- Modelling of skin effect in deep rotor bars
- Coupling of the electromagnetic models with thermal models [5]
- Modelling of higher field harmonics in space

References

[1] H. Kleinrath, *Stromrichtergespeiste Drehfeldmaschinen.* Wien: Springer Verlag, 1980.

[2] C. Kral, "Derivation of the space phasor equations and the required parameters of a squirrel cage induction machine with faulty rotor

bars," *Conference Proceedings of the International Symposium on Diagnostics for Electrical Machines, Power Electronics and Drives, SDEMPED*, pp. 395–400, 1999.

[3] H. Späth, *Elektrische Maschinen.* Berlin: Springer Verlag, 1973.

[4] C. Kral, *Model and Behavior of an Induction Machine with a Faulty Rotor Bar, Including Detection by Means of the Vienna Monitoring Method (in German).* PhD thesis, Vienna University of Technology, 1999.

[5] C. Kral, A. Haumer, and M. Plainer, "Simulation of a thermal model of a surface cooled squirrel cage induction machine by means of the SimpleFlow-library," *Modelica Conference*, 2005.

A Magnetic Library for Modelica

Thomas Bödrich
Dresden University of Technology, Institute of Electromechanical and Electronic Design
01062 Dresden, Germany

Thomas.Boedrich@mailbox.tu-dresden.de

Thomas Roschke
Saia-Burgess Dresden GmbH
Wilhelm-Liebknecht-Strasse 6,
01257 Dresden, Germany

Thomas.Roschke@saia-burgess.com

Abstract

A Modelica library for modeling of electro-magneto-mechanical actuators and drives is presented in this paper. The model components in this library are suited for actuator design itself as well as for dynamic simulation of complete drive systems. For modeling of the magnetic subsystem of actuators and drives, the concept of magnetic flux tubes is used in this library, leading to a network model of the actuators magnetic subsystem. Both the method of magnetic flux tubes and the developed library will be presented below. As an example, modeling of an industrial solenoid actuator will be explained.

Keywords: Magnetic library; Magnetic flux tube; Magnetic network; Solenoid Actuator

1 Introduction

Electromagnetic motors and actuators convert electrical energy via a magnetic field into mechanical work (or vice versa for generators). Different physical effects are utilized for this energy conversion, depending on the structure of the electromagnetic device. It is necessary to arrange the interactions in the electro-magneto-mechanical energy conversion process carefully during actuator design in order to achieve an optimal solution. Also, not only the actuator itself but also the environment of the system to be developed must be considered during design, too. For example, the efficiency of electromagnetic actuators and drives can be significantly increased by means of electronic circuits for excitation and control. The complexity of the above design task requires utilization of computer-based modeling and simulation techniques for both:

- design of the distribution of the magnetic field inside the actuator, and
- dynamic simulation of the complete drive system.

Latter task can be accomplished e. g. with the multi-domain model description language *Modelica* and accompanying simulation environments, where the system to be simulated is described with a set of differential and algebraic equations (DAE).

The problem for the design of magnetic actuators is, however, that the distribution of the magnetic field is described by partial differential equations (*Maxwell*'s equations). Calculation and optimization of such field distributions and resulting integral quantities such as magnetic forces is necessary during actuator design. Finite Element Analysis (FEA) is a valuable tool for this task. However, the high computational effort of FEA restricts its use in most cases to stationary field calculations. Dynamic simulation of a complete system incorporating a detailed model of the actuators magnetic field and detailed models of the feeding electronics and the mechanical load to be moved is not possible with reasonable effort using FEA.

To overcome the difficulties described above, the method of magnetic flux tubes can be used for actuator and system design [1]-[4]. This method allows a simplified description of the magnetic field inside a magnetic device so that the field distribution can be described with a set of algebraic equations rather than with the precise partial differential equations. With this approach, the field distribution in a magnetic actuator can be simulated together with neighboring subsystems in one DAE system with little computational effort. The accuracy is reasonable for coarse design of actuators and for system simulation.

2 Magnetic Flux Tubes

A magnetic flux tube is a defined volume inside a magnetic field with homogenous distribution of the magnetic field strength **H** and the magnetic flux density **B** within this region (Fig. 1).

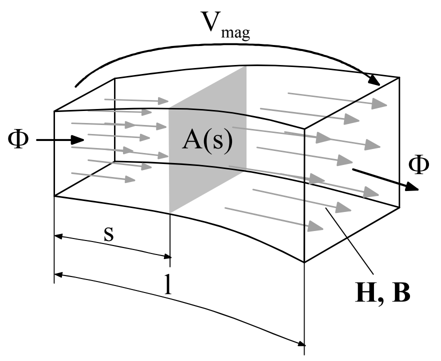


Fig. 1 Magnetic flux tube

Presuming that the principal distribution of the magnetic field inside a magnetic device is known, the magnetic field in the complete device can be divided into a network of flux tubes. For each flux tube element, a magnetic potential difference V_{mag} is defined as an across variable:

$$V_{mag} = \int_s \mathbf{H} ds = Hl \qquad (1).$$

The magnetic flux Φ through each cross sectional area A of a flux tube is calculated as follows:

$$\Phi = \int_A \mathbf{B} dA \qquad (2)$$

With the above across variable V_{mag} and the flow variable Φ, a magnetic reluctance R_{mag} can be defined analogue to resistive network elements in other physical domains:

$$R_{mag} = \frac{V_{mag}}{\Phi} = \frac{\int_s \mathbf{H} ds}{\int_A \mathbf{B} dA} \qquad (3).$$

Homogenous distribution of **B** and **H** through each cross section inside a flux tube is one of the assumptions of the flux tube approach. Hence, equation (2) simplifies to

$$\Phi = B(s)A(s) \qquad (4).$$

Maxwell's constitutive equation

$$\mathbf{B} = \mu_o \mu_r \mathbf{H} \qquad (5)$$

describes the material properties (μ_0 - permeability of vacuum, μ_r - relative permeability, see section 3.2). With the equations (3)-(5), the general formula for a magnetic reluctance of any shape can be given:

$$R_{mag} = \frac{V_{mag}}{\Phi} = \int_s \frac{ds}{\mu_0 \mu_r(s) A(s)} \qquad (6).$$

For a prismatic or a cylindrical volume of length l and cross sectional area A with the magnetic flux entering and leaving the region through its end planes, equation (6) simplifies to

$$R_{mag} = \frac{l}{\mu_0 \mu_r(B) A} \qquad (7).$$

As for the above example, equations for the magnetic reluctance of other common geometries can be found.

Similar lumped elements with the same magnetic flow and across variables can be defined for sources of a magnetic potential difference V_{mag} (see section 3.3) or for sources of a magnetic flux Φ, if needed. For a magnetic network consisting of at least one source and one reluctance element, the field distribution in a magnetic device can be calculated with little computational effort according to *Kirchhoff*'s laws.

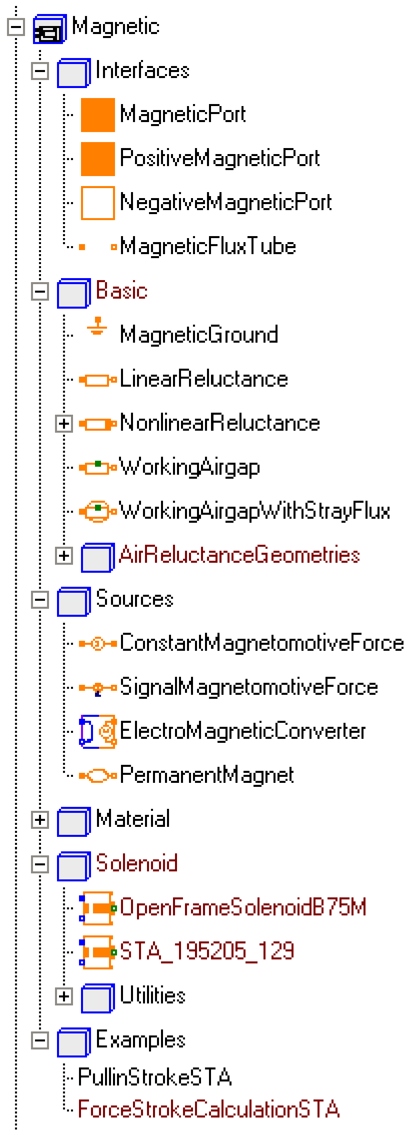
3 Structure of the Magnetic Library

Based on the concept of magnetic flux tubes, a Modelica library for modeling of magnetic components and devices has been developed. The structure of this library is shown in Fig. 2. Its sublibraries and model components are described only in short in this section; a more detailed description of selected model components can be found in section 4, where usage of the library elements is explained with an example.

3.1 Interfaces Sublibrary

A domain specific magnetic connector was defined:

```
connector MagneticPort
   Modelica.SIunits.MagneticPotentialDifference V_mag
      "Magnetic potential at the port";
   flow Modelica.SIunits.MagneticFlux Phi
      "Magnetic flux flowing into the port";
end MagneticPort;
```



```
Magnetic
  Interfaces
    MagneticPort
    PositiveMagneticPort
    NegativeMagneticPort
    MagneticFluxTube
  Basic
    MagneticGround
    LinearReluctance
    NonlinearReluctance
    WorkingAirgap
    WorkingAirgapWithStrayFlux
    AirReluctanceGeometries
  Sources
    ConstantMagnetomotiveForce
    SignalMagnetomotiveForce
    ElectroMagneticConverter
    PermanentMagnet
  Material
  Solenoid
    OpenFrameSolenoidB75M
    STA_195205_129
    Utilities
  Examples
    PullinStrokeSTA
    ForceStrokeCalculationSTA
```

Fig. 2 Structure and sublibraries of the Modelica Magnetic library

Based on the connector definition above, a base model for all reluctance elements and selected source elements is used throughout the library:

```
partial model MagneticFluxTube
      "Component with two magnetic ports p and n and
       magnetic flux Phi from p to n"
   PositiveMagneticPort p "Positive magnetic port";
   NegativeMagneticPort n "Negative magnetic port";
   Modelica.SIunits.MagneticPotentialDifference V_mag
      "Magnetic potential difference between the two ports";
   Modelica.SIunits.MagneticFlux Phi
      "Magnetic flux flowing from pin p to pin n";
equation
   V_mag = p.V_mag - n.V_mag;
   0 = p.Phi + n.Phi;
   Phi = p.Phi;
end MagneticFluxTube;
```

3.2 Basic Sublibrary

Most magnetic devices contain ferromagnetic components that carry the magnetic flux imposed by one or more coils or permanent magnets to a working air gap where the desired magnetic force (or torque for rotating machines) is generated. This is because of the high relative permeability of ferromagnetic material compared to that of vacuum or air ($\mu_r = 1$). However, the relative permeability for each point of a ferromagnetic material is not constant but depends on the actual magnetic field strength H respectively the actual magnetic flux density B of this point [2], [4]. The so called commutation curve $B(H)$ of a steel commonly used in magnetic devices is shown in Fig. 3 as an example. According to equation (5) this results in the characteristic shape $\mu_r(H)$ of Fig. 3. Proper modeling of this nonlinear relationship is crucial for the accuracy of flux calculations and hence for the resulting magnetic forces. In engineering practice, the relationship $\mu_r(B)$ is often used instead of $\mu_r(H)$.

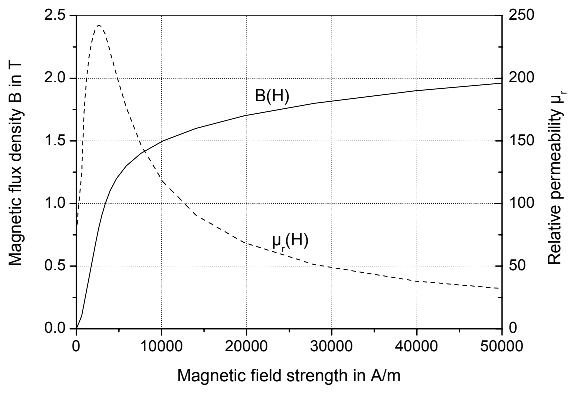


Fig. 3 Magnetic flux density B vs. magnetic field strength H and corresponding relative permeability μ_r for steel 1.0715 (9SMn28)

For the nonlinear reluctance elements in the Basic sublibrary, the relationship $\mu_r(B)$ is approximated with a function [4]. The Material sublibrary contains the coefficients of this function for different ferromagnetic materials. It is planned to implement additional possibilities to describe the relationship $\mu_r(B)$, e. g. spline interpolation.

3.3 Sources Sublibrary

The source elements *ConstantMagnetomotiveForce* and *SignalMagnetomotiveForce* both are ideal sources of a magnetic potential difference V_{mag}. They are intended for use in stationary flux calculations where no coupling between the feeding electrical and the magnetic subsystem is needed and where dynamic effects of this coupling need not to be considered.

The source element *ElectroMagneticConverter* couples the electrical subsystem of a electromagnetic drive system with the magnetic subsystem, i. e. with the network of magnetic flux tubes. Two equations are needed to describe this coupling [4]: *Faraday*'s law

$$u_{ind} = -w\frac{d\Phi}{dt} \qquad (8)$$

and

$$V_{mag} = i \cdot w \qquad (9).$$

Equation (8) describes the voltage u_{ind} induced in a coil due to a change of the flux linkage $w\Phi$ inside this coil with respect to time (w – number of coil windings). In most electromagnetic devices, the coil flux Φ is a nonlinear function of both the coil current i (due to saturation effects in ferromagnetic components) and the position x respectively φ of the device's moving component: $\Phi = f(i, x)$ for translational actuators and $\Phi = f(i, \varphi)$ for rotating electrical machines. The dependencies of the coil flux Φ on both current and position are intrinsically accounted for in actuator models according to the flux tube approach as will be obvious from the example in section 4.

Equation (9) describes the magnetic potential difference V_{mag} fed into a network of flux tubes due to the coil current. This equation is derived from *Ampere*'s law.

The Sources sublibrary is completed by a model of a permanent magnet. It is made up of a series connection of an ideal source of a magnetic potential difference V_{mag} and the linear ($\mu_r = const.$) reluctance of the permanent magnet [2], [3].

4 A Solenoid Actuator as an Example

Solenoid actuators offer a very robust and simple structure, a good force to mass ratio with respect to dynamic behavior and stroke and a low price. For that reasons they are widely used as drive element in a huge diversity of applications, e.g. in locking mechanisms throughout automation and automotive engineering, in fluidic valves or in relays and switchgear.

The principal structure of an electromagnetic actuator is shown in Fig. 4. Depicted is an industrial DC solenoid for applications throughout automation. The cross-sectional view above shows a solenoid of the Saia-Burgess STA series that will be used as modeling example [5]. In contrast to the depicted actuator with a conical pole shape, the modeled solenoid STA 195205-129 has a plane pole face as shown in the schematic view below.

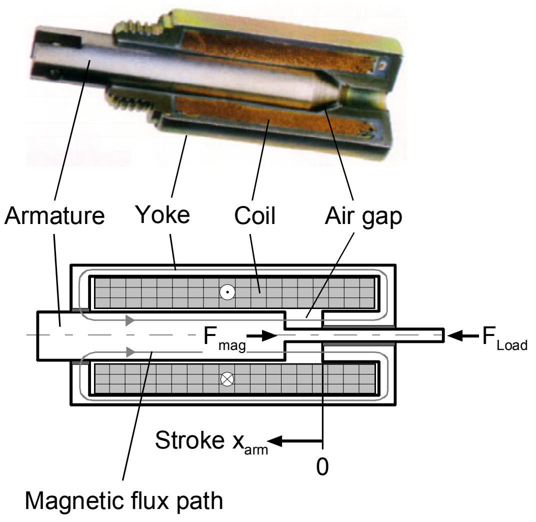


Fig. 4 Cross-sectional view and principal structure of a typical solenoid actuator [5]

The working principle of solenoid actuators is based on reluctance forces [2], [4]: The magnetic flux Φ generated by a current in the coil goes through a working air gap where a magnetic force is developed due to the gradient in relative permeability μ_r on the boundaries between the ferromagnetic parts with $\mu_r >> 1$ and the air with $\mu_r = 1$.

The force-stroke characteristics of electromagnetic actuators can be widely influenced and shaped according to applications needs by variation of the geometry of pole and armature. Typically however for most electromagnetic actuators is a highly nonlinear force-stroke characteristic $F_{mag}(x_{arm})$ with a minimum force at the armature rest position (maximum air gap length) and a strong increase in magnetic force towards minimum air gap length [2].

4.1 Model of the Solenoid

The graphical representation of the Modelica model of the actuator is shown in Fig. 5. It consists of the electrical subsystem (coil) on the left side with the electro-magnetic converter, the magnetic network based on the method of flux tubes in the middle, and the mechanical subsystem that models the armature dynamics on the right side. At the right flange connector, additional models from the Translational library of the Modelica Standard library can be attached, e.g. a mass to be moved, a return spring or process forces.

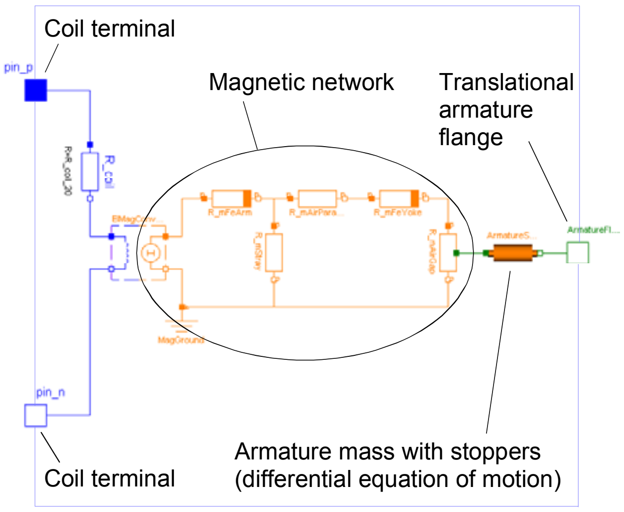


Fig. 5 Graphical representation of the Modelica model of the solenoid

The electrical subsystem with the current *i*, the voltage across the coil terminals v_{coil}, the magnetic flux Φ enclosed by each of the w windings, and the winding resistance R_{coil} is described with the following equation (see also equation (8)):

$$v_{coil} = iR_{coil} + w\frac{d\Phi}{dt} \quad (10).$$

For voltage controlled operation of the solenoid, v_{coil} is the voltage of the driving voltage source.

The magnetic network of Fig. 5 is intentionally kept simple in this example. In Fig. 6 the network is shown over a field plot of the actuators magnetic field obtained from FEA.

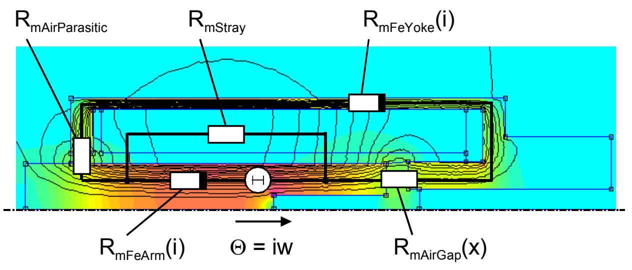


Fig. 6 Magnetic network over a FEA field plot of the sample actuator (line of symmetry at the bottom)

The magnetomotive force Θ is fed into the network by the electro-magnetic converter. It is an ideal source of a magnetic potential difference V_{mag} according to equation (9). The two ferromagnetic reluctance elements $R_{mFeArm}(i)$ and $R_{mFeYoke}(i)$ represent the ferromagnetic components of the actuator. They are calculated from the actuators main dimensions with equation (7). Due to the nonlinear $B(H)$-relationship of ferromagnetic materials, they depend on the solenoid current *i*. The steel 1.0715 with the $B(H)$-characteristic of Fig. 3 is used in both reluctance elements. An approximated function $\mu_r(B)$ derived from that material data is used for calculation of both reluctance values with equation (7).

Three reluctance elements through air are present in this magnetic network model: R_{mStray} is a simple but yet effective description of the stray flux of the solenoid. Despite the simple structure of the actuators magnetic network it should not be omitted in the model. This is because of the large ratio of total actuator length to outer diameter of the sample actuator. Typical for solenoids with such a geometry is that part of the magnetic field lines close without going through the working air gap $R_{mAirGap}$.

The reluctance of the working air gap is calculated with equation (7), where air gap length *l* is identical with the armature position x_{arm}. The magnetic or reluctance force F_{mag} that is generated at the boundaries between the ferromagnetic armature respectively the pole and the air is calculated with *Maxwell*'s formula [2], [4]:

$$F_{mag} = \frac{\Phi^2}{2\mu_0 A} \quad (11),$$

where *A* is the cross sectional area of the air gap respectively the area of the pole face. The developed force F_{mag} is fed as driving force for the armature mass into the mechanical subsystem of the actuator model, where acceleration, velocity and armature position x_{arm} are calculated from the differential equation of motion.

The magnetic network is completed with the reluctance $R_{mAirParasitic}$. This element describes the parasitic air gap in the non-ferromagnetic slide guiding for the armature.

4.2 Force-Stroke Characteristics

An important criterion that characterises an actuator is its force-stroke characteristics. For the solenoid model, this characteristic $F_{mag}(x_{arm})$ was calculated with a quasi-static enforced movement of the armature (Fig. 7). This is in accordance to real force-stroke measurements on actuators.

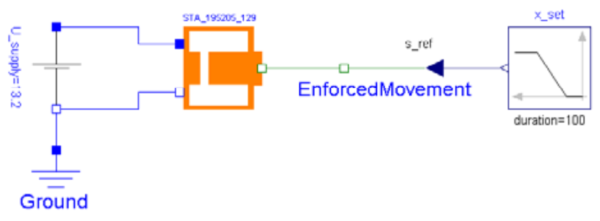


Fig. 7 Quasi-static enforced armature movement for calculation of the force-stroke characteristics

Simulated and measured curves for the sample actuator are compared in Fig. 8. Simulation was done with the magnetic network model of Fig. 5. Measured data was taken from the catalogue of the manufacturer. For comparison, the force-stroke characteristic obtained from FEA at different armature positions is included in this diagram, too.

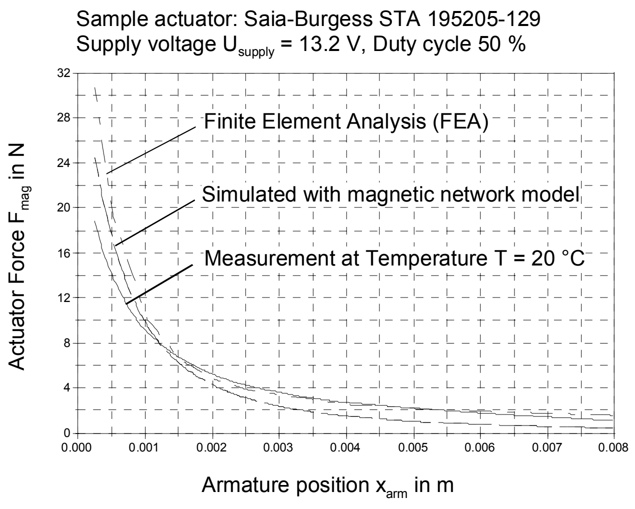


Fig. 8 Comparison of measured and simulated force-stroke characteristics of the sample actuator

Reasons for the differences between the three curves are:

- limited accuracy of the force-stroke measurement especially at small air gaps due to deflection of the load cell and the mounting rack,
- simple structure of the magnetic flux tube model in this example (e. g. no additional stray reluctance around the working air gap),
- neglect of additional parasitic air gaps especially in the FEA model, e. g. between yoke lid and yoke,
- differences between the simulated and the real material characteristics $\mu_r(B)$ for the ferromagnetic yoke and the armature.

Despite the differences between measurement and simulation with the magnetic network model, the flux tube approach is well suited for coarse design of magnetic devices and estimation of force-stroke characteristics respectively torque-angle characteristics for rotational devices prior to further design steps.

4.3 Simulation of a Pull-in Movement

The main advantage of a modeling approach based on magnetic flux tubes compared to a design approach using FEA is the little computational effort of flux tube models that allows for extensive dynamic simulations during both actuator and system design. To illustrate the capabilities of magnetic network models for dynamic simulation, a pull-in stroke of the armature was simulated with the network model of the sample solenoid of Fig. 5. In Fig. 9, simulated coil current i and armature position x_{arm} are shown together with measured data after a voltage step from 0 to 13.2 *VDC* at time $t = 0$ was applied. The armature was in horizontal position, no additional mechanical load was attached to it.

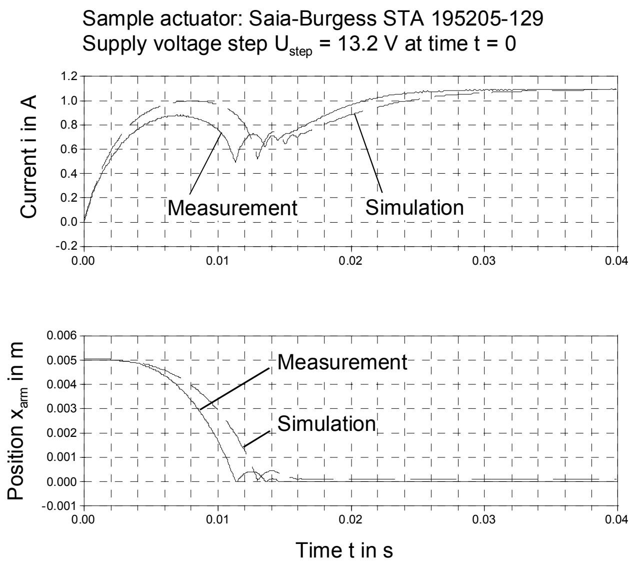


Fig. 9 Comparison of simulated and measured solenoid current i and armature position x_{arm} after a voltage step at time $t = 0$

The characteristic current drop during the pull-in stroke of the armature results from the increase of the solenoids inductance as the armature moves into the solenoid. This is due to the decrease of the air gap reluctance and the increase of the total magnetic flux during the movement.

Differences between the measured and the simulated curves can be explained with the simple structure of the network model. Higher accuracy of the solenoid model can be achieved with a more detailed magnetic network, if needed.

5 Conclusions

Basic magnetic network elements in Modelica were already presented in [6]. The model elements described in this paper are intended for modeling of rotating electrical machines. The magnetic elements (e.g. linear magnetic resistance, electro-magnetic converter) are described in terms of integral machine parameters. Differently from the approach in [6], the model components of the newly developed Magnetic library are defined in terms of their geometric dimensions and their material properties. They are hence suited for modeling of the geometry of magnetic components during actuator design, but for system simulation of the actuator together with its neighboring subsystems, too.

A dynamic model of a general purpose solenoid was presented to illustrate the concept of magnetic flux tubes as well as usage of the library. Modeling of the dynamic properties of solenoids during actuator design is of vital importance in many sectors. For example, electromagnetic injection valves in automotive applications must actuate in as little as 1 *ms*, and proportional solenoids that drive valves in fluidic applications operate at frequencies above 100 *Hz*.

Future work on the Magnetic library will focus on the following issues:

- refinement of implementation details, e. g. the description of material properties with tables and spline interpolation,
- estimation of eddy current losses for flux tube elements with electrical conductivity,
- implementation of additional models for magnetic devices, e.g. for solenoids with different pole shapes, electrodynamic drives or small rotating motors.

Preferably, the latter models shall be scalable, so that device manufacturers can easily build models for each product within a product family with different sizes. On the other hand, system engineers can use these models for simulation of complex mechatronic systems, e.g. in automation industry or in automotive applications.

References

[1] Roters, H.: Electromagnetic Devices. New York: John Wiley & Sons 1941

[2] Kallenbach, E.; Eick, R.; Quendt, P.; Ströhla, T.; Feindt, K.; Kallenbach, M.: Elektromagnete: Grundlagen, Berechnung, Entwurf und Anwendung. 2. Aufl. Wiesbaden: B.G. Teubner 2003

[3] Hendershot, J.R. Jr.; Miller, T.J.E.: Design of Brushless Permanent-Magnet Motors. Magna Physics Publishing and Oxford University Press 1994

[4] Roschke, Th.: Entwurf geregelter elektromagnetischer Antriebe für Luftschütze. Düsseldorf: VDI-Verlag 2000

[5] Saia-Burgess Solenoid Catalogue. Internet: www.saia-burgess.com

[6] Beuschel, M.: A Uniform Approach for Modeling Electrical Machines. Proceedings of the Modelica 2000 Workshop, Lund, Sweden, Oct. 23-24, 2000, pp. 101-108

Session 7b

Real-Time and Reactive Systems

StateGraph – A Modelica Library for Hierarchical State Machines

Martin Otter[1], Karl-Erik Årzén[2], and Isolde Dressler[2]
[1]DLR Institute of Robotics and Mechatronics, Oberpfaffenhofen, Germany, Martin.Otter@dlr.de
[2]Lund Institute of Technology, Lund, Sweden, {karlerik, Isolde.Dressler}@control.lth.se

Abstract

The new library Modelica.StateGraph is a free Modelica package providing components to model discrete event and reactive systems in a convenient way. It has a similar modeling power as Statecharts, but avoids some deficiencies of Statecharts by using elements of JGrafchart and by using Modelica as an "action" language. An overview of the StateGraph library is given, the available components and an application example. The implementation of the library in Modelica is sketched, especially the needed extension to Modelica that will be available in release 2.2 of the Modelica language.

1 Introduction

This section shortly discusses discrete event formalisms and the relationship to the StateGraph library.

Grafcet [3], or the industrial alias Sequential Function Charts (SFC), is a state-transition based computational model that has been widely accepted in the industrial automation industry for representing sequential control logic. It is defined in the standards IEC 848 and IEC 61131-3. States are represented by steps to which actions can be associated, and the steps are interconnected by transitions with associated Boolean conditions or event expressions. The activity in a Grafcet diagram flows downwards from the top of the diagram. It supports alternative branches, parallel branches, and repetition. Hierarchies are supported in the form of macro steps.

Although Grafcet has the same formal power of expression as an ordinary state machine, it is cumbersome to use for representing larger state-machine oriented models. For these applications the Statecharts formalism is better suited [6]. Statecharts use a syntax that is similar to ordinary state machines and supports hierarchical states through the concept of superstates, a considerably more powerful concept than the macro steps of Grafcet.

Grafchart is the name of a graphical language aimed at supervisory control applications developed at Lund University [1]. It combines the function chart formalism of Grafcet with the hierarchical states of Statecharts. It also supports parameterized function chart procedures. Through this the best concepts from both Grafcet and Statecharts are combined. JGrafchart is the name of a Java implementation of Grafchart [2]. It is a combined graphical editor and run-time system, and can be viewed as a soft-PLC. It is also possible to use JGrafchart only as a graphical editor generating executable code. In [4] code generation from JGrafchart to Modelica is presented. Code generation has also been provided to C and Java.

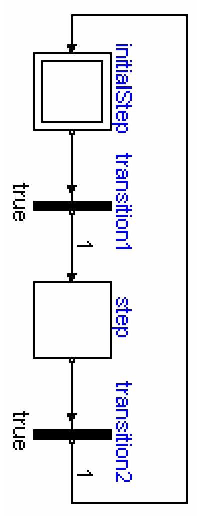
The StateGraph library is based on a subset of JGrafchart. Besides minor modifications to arrive at a suitable Modelica implementation, the essential difference is to use Modelica as an "action" language. The "single assignment rule" of Modelica makes it completely different to the action languages used in the formalisms from above. It will be shown that this has significant advantages.

2 Users View

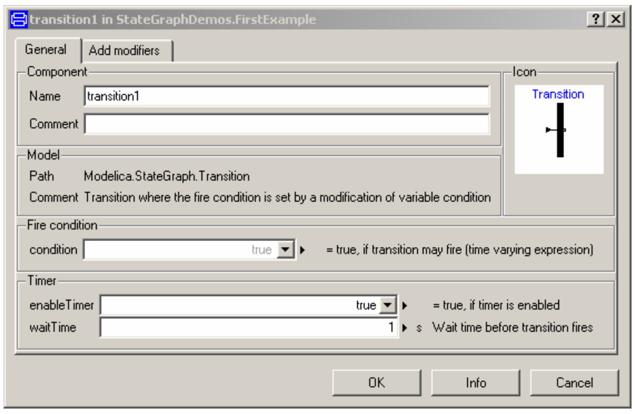
In this section the components of the StateGraph library are introduced by examples to show how it can be used in applications.

2.1 Steps and Transitions

The basic elements of StateGraphs are **steps** and **transitions** as shown in the next figure. **Steps** represent the possible states a StateGraph can have. If a step is active the Boolean variable **active** of the step is **true**. If it is deactivated, **active** = **false**. At the initial time, all ordinary steps are deactivated. The **InitialStep** objects are steps that are activated at the initial time. They are characterized by a double box (see next figure at the left).



Transitions are used to change the state of a StateGraph. When the step connected to the input of a transition is active, the step connected to the output of this transition is deactivated and when the transition condition becomes true, then the transition fires. This means that the step connected to the input to the transition is deactivated and the step connected to the output of the transition is activated. The transition **condition** is defined via the parameter menu of the transition object. Clicking on object "transition1" in the above figure, results in the following menu:


transition1 in StateGraphDemos.FirstExample

General | Add modifiers

Component
Name: transition1
Comment:

Icon: Transition

Model
Path: Modelica.StateGraph.Transition
Comment: Transition where the fire condition is set by a modification of variable condition

Fire condition
condition: true = true, if transition may fire (time varying expression)

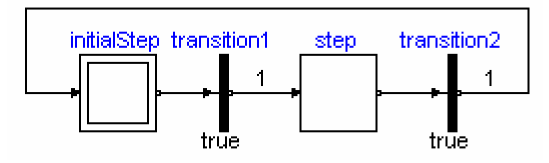
Timer
enableTimer: true = true, if timer is enabled
waitTime: 1 s Wait time before transition fires

OK | Info | Cancel

In the input field "**condition**", any type of time varying Boolean expression can be given (in Modelica notation, this is a modification of the time varying variable **condition**). Whenever this condition is true, the transition can fire. Additionally, it is possible to activate a timer, via **enableTimer** (see menu above) and provide a **waitTime**. In this case the firing of the transition is delayed according to the defined waitTime. The transition only fires if the condition remains true during the waitTime. The transition condition and the waitTime are displayed in the transition icon.

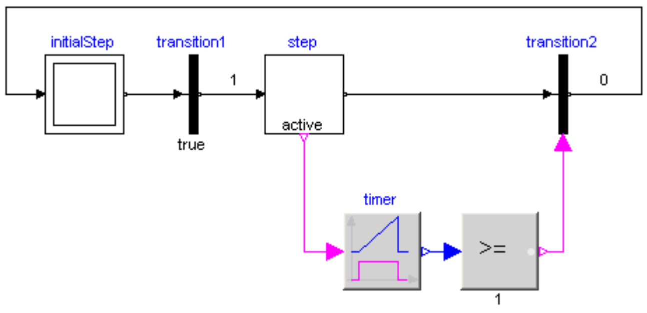
In the above example, the simulation starts at **initialStep**. After 1 second, **transition1** fires and **step1** becomes active. After another second **transition2** fires and **initialStep** becomes again active. After a further second **step1** becomes active, and so on.

In Grafchart, Grafcet and SFC the network of steps and transitions is drawn from top to bottom. In StateGraphs, no particular direction is defined, since Modelica models do not depend on the placement of components and connection lines. Usually, it is more practical to define the network from left to right, since it is easier to read the labels on the icons. The example from above has then the following layout:

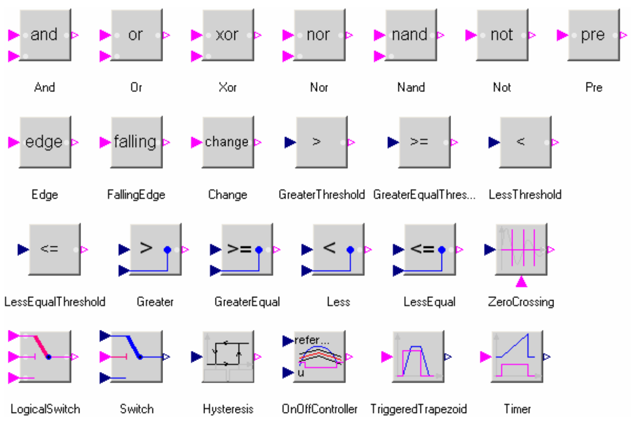


2.2 Conditions and Actions

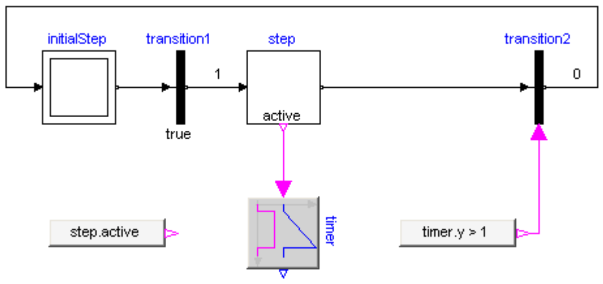
With the block **TransitionWithSignal**, the firing condition can be provided as Boolean input signal, instead as entry in the menu of the transition with block Transition, see example in the next figure:



Component "step" is an instance of "StepWithSignal" that is a usual step where the active flag is available as Boolean output signal. To this output, component "Timer" from library "Modelica.Blocks.-Logical" is connected. It measures the time from the time instant where the Boolean input (i.e., the active flag of the step) became true up to the current time instant. The timer is connected to a comparison component. The output is true, once the timer reaches 1 second. This signal is used as condition input of the transition. As a result, "transition2" fires, once step "step" has been active for 1 second. Of course, any other Modelica block with a Boolean output signal can be connected to the condition input as well, especially blocks of the Modelica.Blocks.-Logical library, see next figure. The Logical library will be extended in the future. It is also easy for a user to define his own, specialized logical blocks.



Instead of using logical blocks, via the Modelica.Blocks.Sources.SetBoolean component any type of logical expression can be defined in textual form, as shown in the next figure:



With the block "**SetBoolean**", a time varying expression can be provided as modification to the output signal **y** (see block with icon text "timer.y > 1" in the figure above). The output signal can be in turn connected to the condition input of a TransitionWithSignal block.

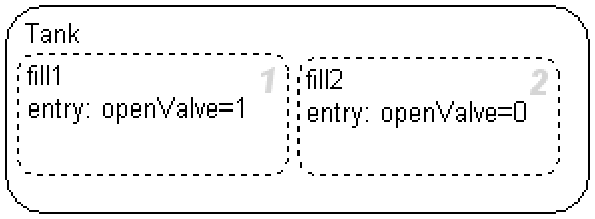
The "**SetBoolean**" block can also be used to compute a Boolean signal depending on the active step. In the figure above, the output of the block with the icon text "step.active" is true, when "step" is active, otherwise it is false (note, the icon text of "SetBoolean" displays the modification of the output signal "y"). This signal can then be used to compute desired **actions** in the physical systems model. For example, if a **valve** shall be open, when the StateGraph is in "step1" or in "step4", a "SetBoolean" block may be connected to the valve model using the following condition:

```
      step1.active or step2.active
```

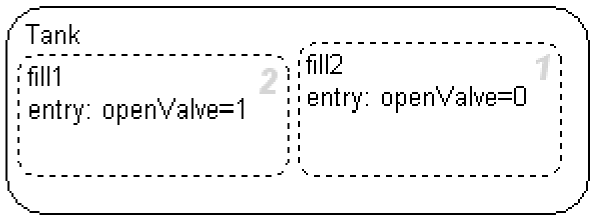
Via the Modelica operators **edge**(..) and **change**(..), conditions depending on rising and falling edges of Boolean expressions can be used when needed.

In Grafchart, Grafcet, SFC and Statecharts, **actions** are formulated **within a step**. Such actions are distinguished as **entry**, **normal**, **exit** and **abort** actions. For example, a valve might be opened by an entry action of a step and might be closed by an exit action of the same step. In StateGraphs this is **not possible** due to Modelicas "single assignment rule" that requires that every variable is defined by exactly one equation. Instead, the approach explained above is used. For example, via the "SetBoolean" component, the valve variable is set to true when the StateGraph is in particular steps.

This feature of a StateGraph is very useful, since it allows a Modelica translator to **guarantee** that a given StateGraph has always **deterministic** behaviour without conflicts. In the other methodologies this is much more cumbersome. As an example, in the next figure a critical situation in Stateflow is shown (Mathworks Stateflow is similar to a StateGraph but has, e.g., a slightly different visual appearance, and is integrated in Mathworks Simulink):



The two substates “fill1” and “fill2” are executed in parallel. In both states the variable “openValve” is set as entry action. The question is whether openValve will have value 0 or 1 after execution of the steps. Stateflow changes this non-deterministic behaviour to a formally deterministic one by defining an execution sequence of the states that depends on their graphical position. The light number on the right of the states shows in which order the states are executed. In the figure above this means that “openValve=0” after leaving the two states. If the second state “fill2” is changed a little bit graphically



“openValve=1” after “fill1” and “fill2” have been executed. This is a dangerous situation because (a) slight changes in the placement of states might change the simulation result and (b) if the parallel execution of actions depends on the evaluation order, errors are very difficult to detect.

Note, similar problems occur in other StateGraph variants, SFC, Grafcet and Graphcharts: Variables are changed according to an evaluation sequence of the simulator. It seems not possible to provide an easy-to-grasp rule about evaluation order of actions that are executed in parallel. Therefore, either the simulator just uses an internal evaluation order, or non-obvious rules are present as in Stateflow that do not solve the underlying problem.

In a StateGraph, such a situation is detected by the translator resulting in an error, since there are two equations to compute one variable. The user is forced to reformulate the network by explicitly defining priorities. For example, if “fill1” and “fill2” are steps that are executed in parallel, there might be a “SetBoolean” block that defines:

```
  openValve =
      if fill1.active then 1 else
      if fill2.active then 0 else 2
```

Therefore step fill1 has a higher priority as step fill2.

In a Stategraph or Graphchart it is difficult to modularize a sub chart if the used actions reference variables in an outer scope: Assume, for example, that a state machine "control" has the following hierarchy:

```
control.superstate1.step1
```

Within "step1" a Statechart would, e.g., access variable "control.openValve", say as "entry action: openValve = true". This typical usage has the drawback that it is difficult to use the hierarchical state "superstate1" as component in another context, because "step1" references a particular name outside of this component.

In a StateGraph, there would be typically a "SetBoolean" component in the "control" component stating:

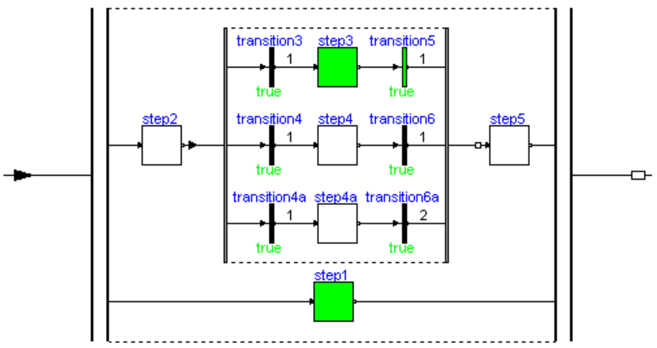
```
openValve = superstate1.step1.active;
```

As a result, the "superstate1" component can be used in another context, because it does not depend on the environment where it is used.

The disadvantage of the StateGraph approach is that the user might not be able to formulate the network directly as desired. For example, in order to fill a tank usually several actions are necessary, e.g., to close one valve and to open another one. In a SFC all actions to "fill a tank" would be defined as actions to a "fill_a_tank" step and this might be more convenient for the user. For example, copying or deleting a "fill_a_tank" step would require only a change at one place in a SFC whereas it would require changes at several places in a StateGraph.

2.3 Parallel and Alternative Execution

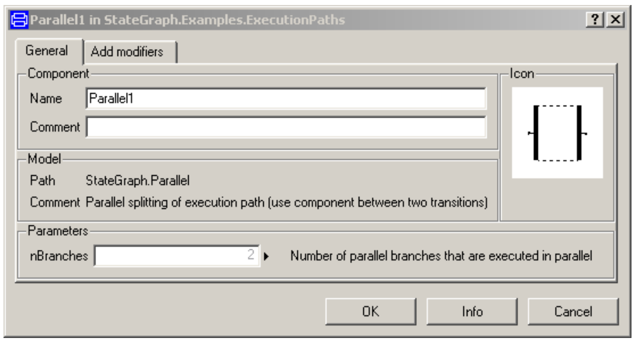
Parallel activities can be defined by component StateGraph.**Parallel** and alternative activities can be defined by component StateGraph.**Alternative**. An example for both components is given in the next figure. Here, the branch from "step2" to "step5" is executed in parallel to "step1". A transition



connected to the output of a parallel branch component can only fire if the final steps in all parallel branches are active simultaneously. The figure above is a screen-shot from the animation of the StateGraph: Whenever a step is active or a transition can fire, the corresponding component is marked in **green** color.

The three branches within "step2" to "step5" are executed alternatively, depending which transition condition of "transition3", "transition4", "transition4a" fires first. Since all three transitions fire after 1 second, they are all candidates for the active branch. If two or more transitions would fire at the same time instant, a priority selection is made: The alternative and parallel components have a vector of connectors. Every branch has to be connected to exactly one entry of the connector vector. The entry with the lowest number has the highest priority.

Parallel, Alternative and Step components have vectors of connectors. The dimensions of these vectors are set in the corresponding parameter menu. E.g. in a "Parallel" component:


Parallel1 in StateGraph.Examples.ExecutionPaths

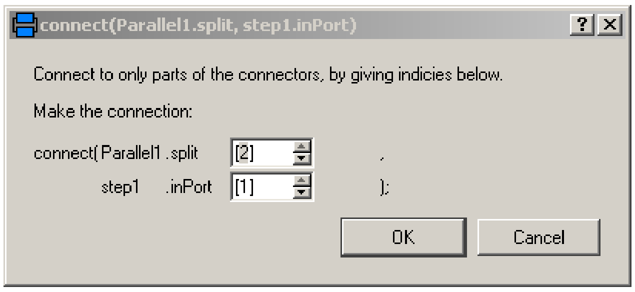
General | Add modifiers

Component
Name: Parallel1
Comment:

Model
Path: StateGraph.Parallel
Comment: Parallel splitting of execution path (use component between two transitions)

Parameters
nBranches: 2 — Number of parallel branches that are executed in parallel

Currently in the Modelica tool Dymola the following menu pops up when a branch is connected to a vector of components in order to define the vector index to


connect(Parallel1.split, step1.inPort)

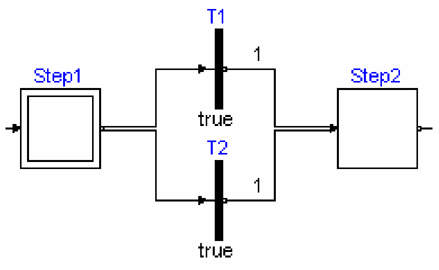
Connect to only parts of the connectors, by giving indicies below.

Make the connection:

connect(Parallel1 .split [2] ,
step1 .inPort [1]);

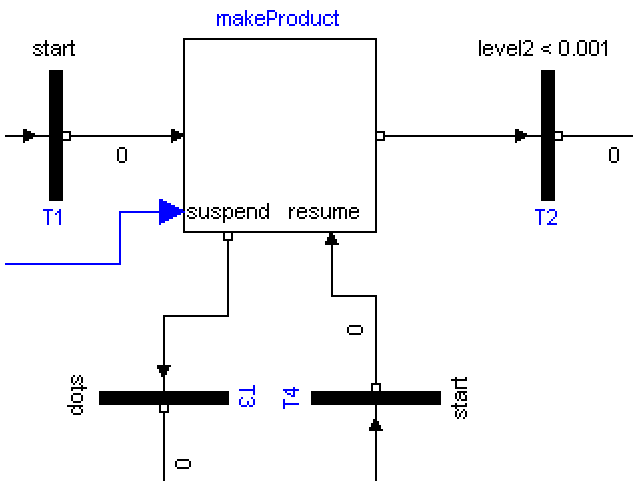
which the component shall be connected. There are discussions to improve the Modelica language to handle such situations more conveniently.

Note, alternative branches can also be defined without the "Alternative" component by just connecting several transitions to the outputs of the same step as shown in the next figure:



2.4 Composite Steps

A StateGraph can be hierarchically structured by using a component that inherits from StateGraph.**PartialCompositeStep**. An example is given in the next figure:



The CompositeStep component contains a local StateGraph that is entered by one or more input transitions and that is left by one or more output transitions. Also, other needed signals may enter a CompositeStep. The CompositeStep has similiar properties as a "usual" step: The CompositeStep is **active** once at least one step within the CompositeStep is active. Variable **active** defines the state of the CompositeStep.

Additionally, a CompositeStep has a **suspend** port. Whenever the transition connected to this port fires, the CompositeStep is left at once. When leaving the CompositeStep via the suspend port, the internal state of the CompositeStep is saved, i.e., the active flags of all steps within the CompositeStep. The CompositeStep might be entered via its **resume** port. In this case the internal state from the suspend transition is reconstructed and the CompositeStep continues the execution that it had before the suspend transition fired (this corresponds to the history ports of Statecharts or JGrafcharts).

A CompositeStep may contain other CompositeSteps. At every level, a CompositeStep and all of its content can be left via its suspend ports (actually, there is a vector of suspend connectors, i.e., a CompositeStep might be aborted due to different transitions).

The CompositeStep can be used in the same way as a superstate in Statecharts. In a superstate it is possible to enter the state in different ways ending up in different internal states. This can be modeled in a StateGraph or a Graphchart by having multiple input transitions, each leading to a different internal step. In a superstate it is possible to exit a superstate in different ways depending on which internal state that is active. This is modeled in a StateGraph or Graphchart by associating different output transitions to the different internal steps. In a superstate it is, finally, also possible to exit the state independently from which internal state that is active. This is achieved with the suspend port here. The conditions connected to the transitions attached to the suspend port can also be conditioned by the status of the internal steps of the CompositeStep. In this way it is possible to suspend the step if a certain condition holds and unless a certain internal step is active. The history arcs in Statecharts correspond to the resume port. Superstates with parallel subparts, so called XOR superstates, can be modeled using parallel constructs inside the CompositeStep.

In addition to using CompositeSteps for modeling hierarchical states they can also be used to simply aggregate a part of a larger StateGraph. This can be useful to improve the structure

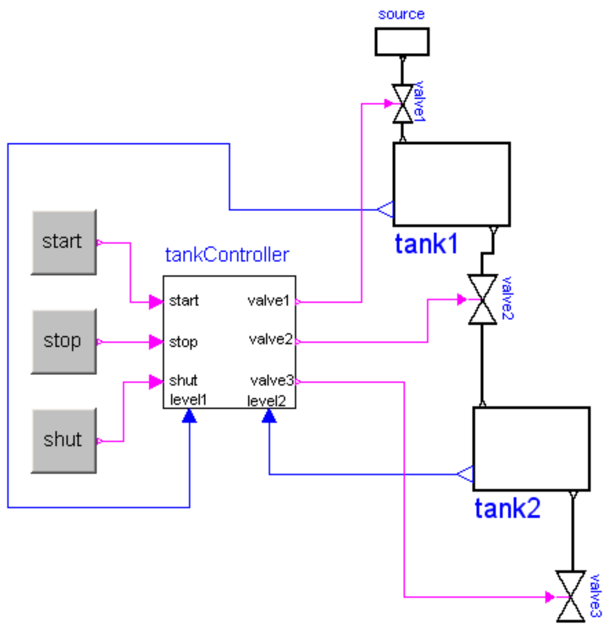
2.5 Execution Model

The execution model of a StateGraph follows from its Modelica implementation: Given the states of all steps, i.e., whether a step is active or not active, the equations of all steps, transitions, transition conditions, actions etc. are sorted resulting in an execution sequence to compute essentially the new values of the steps. If conflicts occur, e.g., if there are more equations as variables, of if there are algebraic loops between Boolean variables, an error occurs. Once all equations have been processed, the **active** variables of all steps are updated to the newly calculated values. Afterwards, the equations are again evaluated. The iteration stops, once no step changes its state anymore, i.e., once no transition fires anymore. Then, simulation continuous until a new event is triggered, i.e., when a relation changes its value.

With the Modelica "sampled(..)" operator, a StateGraph might also be executed within a discrete controller that is called at regular time instants. In a future version of the StateGraph library, this might be more directly supported.

3 Example of a Tank Controller

In this section a more realistic, still simple, application example is given, to demonstrate various features of the StateGraph library. This example shows the control of a two tank system from [4]. In the following figure the top level of the model is shown.



This model is available as Modelica.StateGraph.-Examples.ControlledTanks. In the right part of the figure, two tanks are shown. At the top part, a large fluid source is present from which fluid can be filled in **tank1** when **valve1** is open. Tank1 can be emptied via **valve2** that is located in the bottom of tank2 and fills a second **tank2** which in turn is emptied via **valve3**. The actual levels of the tanks are measured and are provided as signals **level1** and **level2** to the **tankController**.

The **tankController** is controlled by three buttons, **start**, **stop** and **shut** (for shutdown) that are mutually exclusive. This means that whenever one button is pressed (i.e., its state is **true**) then the other two buttons are not pressed (i.e., their states are **false**). The buttons could be implemented as dynamic elements that react when clicking on them. In the example, they are implemented with logical tables, i.e., block Modelica.StateGraph.Temporary.RadioButton, in order that the result of the simulation is reproducible.

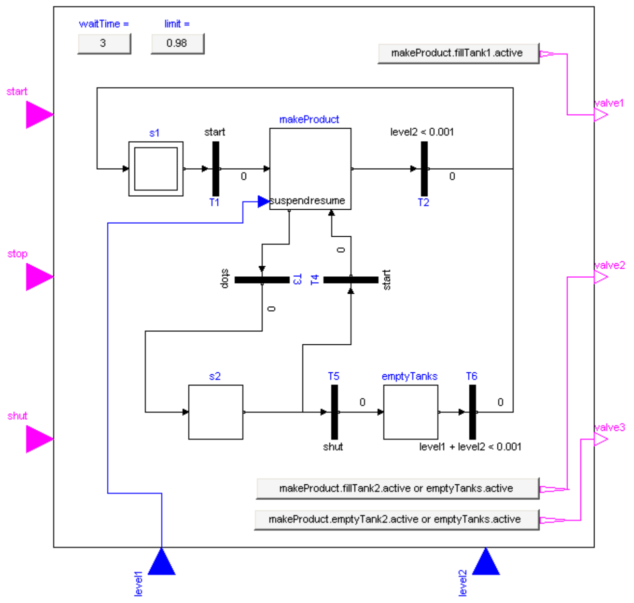
When button **start** is pressed, the "normal" operation to fill and to empty the two tanks is processed:

1. Valve 1 is opened and tank 1 is filled.
2. When tank 1 reaches its fill level limit, valve 1 is closed.
3. After a waiting time, valve 2 is opened and the fluid flows from tank 1 into tank 2.
4. When tank 1 is empty, valve 2 is closed.
5. After a waiting time, valve 3 is opened and the fluid flows out of tank 2
6. When tank 2 is empty, valve 3 is closed

The above "normal" process can be influenced by the following buttons:

- Button **start** starts the above process. When this button is pressed after a "stop" or "shut" operation, the process operation continues.
- Button **stop** stops the above process by closing all valves. Then, the controller waits for further input (either "start" or "shut" operation).
- Button **shut** is used to shutdown the process, by emptying at once both tanks. When this is achieved, the process goes back to its start configuration. Clicking on "start", restarts the process.

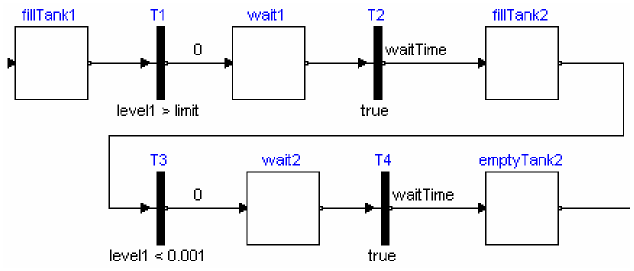
The implementation of the **tankController** is shown in the next figure. When the "**start**" button is pressed, the stateGraph is within the CompositeStep "**makeProduct**". During normal operation this CompositeStep is only left, once tank2 is empty. Afterwards, the CompositeStep is at once re-entered. When the "**stop**" button is pressed, the "makeProduct" CompositeStep is at once terminated via the "**suspend**" port and the stateGraph waits in step "**s2**" for further commands. When the "**start**" button is pressed, the CompositeStep is re-entered via its **resume** port and the "normal" operation continues at the state where it was aborted by the suspend transition. If the "**shut**" button is pressed, the stateGraph waits in the "**emptyTanks**" step, until both tanks are empty and then waits at the initial step "**s1**" for further input.



The opening and closing of valves is **not** directly defined in the StateGraph. Instead via the "**set-ValveX**" components, the Boolean state of the valves are determined. For example, the output y of "setValve2" is computed as:

```
y = makeProduct.fillTank2.active
    or emptyTanks.active
```

i.e., valve2 is open, when step "makeProduct.fillTank2 or when step "emptyTanks" is active. Otherwise, valve2 is closed. The main part of the composite step “makeProduct” is shown in the next figure. Step “fillTank1” is left, once the highest level



for the tank is reached (level1 > limit). The StateGraph remains in step “wait1” during the defined “waitTime”. Afterwards, step “fillTank2” remains active until tank1 is empty (level1 < 0.001). After a waiting phase, the “emptyTank2” step is entered.

4 Implementation

In this section the implementation of the most important parts of the library is sketched.

4.1 Steps and Transitions

Steps and transitions are implemented according to the method described in [7][5] to define Petri nets with an equation based language.

A transition has one inPort and one outPort connector and is basically defined by the following equations (if no timer is present):

```
fire = condition and
          inPort.available and not
          outPort.occupied;
inPort.reset = fire;
outPort.set  = fire;
```

Note, that the inPort connector of a transition consists of the Boolean variables “available” and “reset” and the outPort connector consists of the Boolean variables “occupied” and “set”. The above equation states that “fire = true”, if (1) the firing condition is true, (2) the inPort step is active and (3) the outPort step is not active. The “fire” value is reported to the two steps to which the transition is connected.

A step has a vector of input and a vector of output connectors. It is basically defined as:

```
   active = pre(newActive);
newActive = anyTrue(inPort.set) or active
            and not anyTrue(outPort.reset)
```

The function “anyTrue(..)” returns **true**, if any element of the input vector is **true**. The step becomes active in the next iteration when one of the transitions connected to the inPort connectors fires (set = true if a transition fires). The step remains active if it was active and no transition connected to the outPort connectors fires (reset = true, if a transition fires).

A step reports its active flag to the transition connected to its first outPort by the equation:

```
outPort[1].available = active;
```

In order to make sure that only one of the transitions connected to the outPorts can fire, the active flag is hidden to the second outPort transition if the first transition decides to fire and sends a reset condition:

```
outPort[2].available =
       outPort[1].available and not
       outPort[1].reset;
```

The general case can be written in Modelica as

```
for i in 1:size(outPort,1) loop
  outPort[i].available =
     if i == 1 then active else
       outPort[i-1].available and not
       outPort[i-1].reset;
end for;
```

A step needs to signal to its inPort transitions whether it is possible to activate it or whether it is about to become active via transitions with higher priorities. This is described as

```
for i in 1:size(inPort,1) loop
  inPort[i].occupied =
    if i == 1 then active else
      inPort[i-1].occupied or
      inPort[i-1].set;
end for;
```

The inPort and outPort connectors contain appropriate “input” and “output” prefixes of the connector variables, in order that steps can only be connected to transitions and vice versa. Furthermore, the annotation “Hide = **true**” is set on all connector variables, in order that these variables do not show up in the plot browser, because these are internal variables that are of no interest for the user of the StateGraph library.

In a parameter menu of a component usually only variables are displayed that are declared as parameters. In the parameter menu of a transition, additionally the time varying variable “condition” is displayed as shown in section 2.1. This is implemented by adding the annotation “Dialog” to the variable declaration:

```
Boolean condition annotation(Dialog);
```

Usually, the “Dialog” annotation has additional subentries, such as “group” or “tab”. However, if no subentries are present, this annotation just means to include the variable in the parameter menu.

In a JGraphchart there is a timer associated with every step by providing the time difference between the actual time and the time when the step became active via variable “t”. In a StateGraph no time variable is associated with a step, but an optional timer is provided in a transition and via the connector "active" of a step a timer from the Logical library can be attached to the step. This provides similar functionality as for a JGraphchart. One reason for this change was to improve the efficiency. For example, in a transition the following code fragment to define a timer is present:

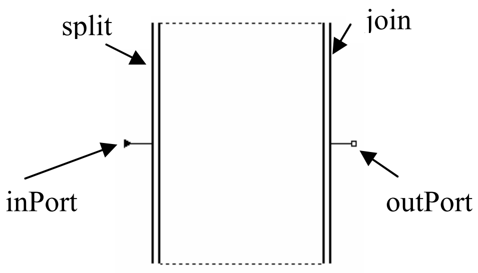
```
if enableTimer then
  when enableFire then
    t_start = time;
  end when;
  t_dummy = time - t_start;
  t = if enableFire then t_dummy else 0;
  fire = enableFire and
         time >= t_start + waitTime
else
  ...
end if;
```

A Modelica translator triggers an event when time reaches "t_start + waitTime". Since "t_start" is a variable that is set in the same scope in a when clause and "waitTime" is a parameter, a Modelica translator can easily trigger a **time event**.

The situation is different, if the when clause "**when** enableFire **then** t_start = time; **end when**" is present within a step and the relation "time >= t_start + waitTime" is present in another component, e.g., in a "condition" of a transition. A Modelica translator will then usually trigger a **state event** because in the scope of the relation it is not known that "t_start" can change its value only at event instants.

4.2 Parallel and Alternative Execution

The parallel component has the following icon



and consists of 4 connectors. The "inPort" and outPort" connectors allow only a connection to transitions. The "split" and "join" connectors are vectors of connectors that are drawn in a quite "lenghty" format to resemble the usual visual layout of parallel execution in SFC. They allow only a connection to steps. After dragging this icon in a model, it is usually enlarged until the desired elements can be placed between the "split" and the "join" connectors.

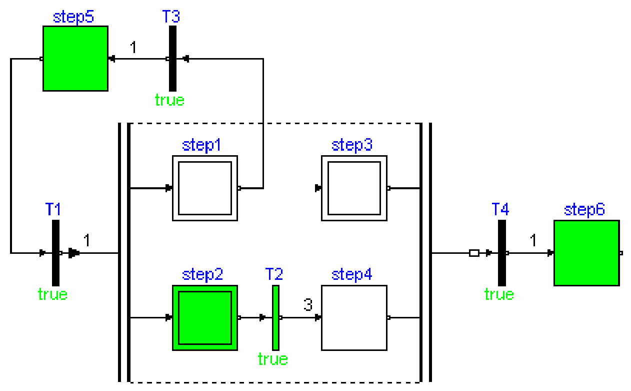
Besides appropriate "assert" statements to guarantee the desired connection structure, the Parallel component consists of the following equations only:

```
n = size(split,1);
split.set  = fill(inPort.set, n);
join.reset = fill(outPort.reset,n);
inPort.occupied  =anyTrue(split.occupied);
outPort.available=allTrue(join.available);
```

The second and third equation report the "set" and "reset" flags of the inPort and outPort connectors to the "split" and "join" connectors. The two last equations perform the synchronization of the parallel branches: Via function "anyTrue(..)" it is defined that the input transition can only fire if none of the steps connected directly to the "split" connector array is active. Via function "allTrue(..)" it is defined that the output transition can only fire if all steps connected directly to the "join" connector array are active.

The implementation of the "Alternative" component is performed in a similar way.

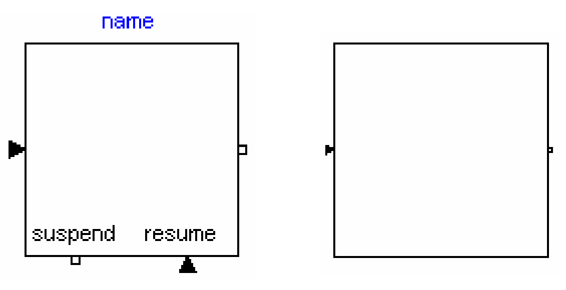
Both the Parallel and the Alternative component have the (slight) disadvantage that they can be misused. For example, in a Parallel Component it is possible to connect from a step in the parallel branches to a transition that is connected to a step outside of the Parallel component, see the example in the next figure:



It would be desirable to prevent such types of networks in a StateGraph. However, it seems not possible to formulate a corresponding restriction with the Modelica language. There are currently Modelica scripting functions under development that allow to traverse a Modelica model and extract information about the model. It might be that such functionality will allow to detect such undesirable networks. These types of function charts are also known as *unsafe* or *unreachable*. In commercial SFC editors it is common that the editor makes it impossible to enter these types of charts, rather than including these global constraints in the language itself.

4.3 Composite Steps

A composite step is a model that extends from PartialCompositeStep. The icon and diagram layer of this superclass is shown in the next figure:



icon layer *diagram layer*

There is one default "inPort" and "outPort" connector on the left and right side. More connectors to enter and leave a composite step may be added. In the icon layer a vector of "suspend" and a vector of "resume" connectors is present. These connectors are not visible in the diagram layer and therefore it is in the graphical editor not possible to connect a component in a composite step to them. The "suspend" and "resume" connector instances are not visible in the diagram layer of a composite step, because the underlying connector classes have an empty diagram layer.

A composite step is active, if at least one step in the composite step is active, and a composite step is deactivated, and also all steps in the composite step, if a transition fires that is connected to one of the "suspend" connectors. This means a communication channel between a composite step and all steps within a composite step is necessary. This is implemented by having a connector

```
connector CompositeStepStatePort
  Boolean suspend;
  Boolean resume;
  flow Real activeSteps;
end CompositeStepStatePort;
```

and use an inner definition of this connector in PartialCompositeStep:

```
inner CompositeStepStatePort root;
   ...
activeSteps = -integer(root.activeSteps);
root.suspend = anyTrue(suspend.reset);
root.resume  = anyTrue(resume.set);
newActive = activeSteps > 0 and not
            anyTrue(suspend.reset) or
            anyTrue(resume.set);
active    = pre(newActive);
```

Via flow variable "activeSteps in the inner root connector, the number of active steps is reported from the steps to the composite step. The composite step is active if this number is greater than zero and no transition at the suspend connector fires ("anyTrue(suspend.reset)") or a transition at one of the "resume" connectors fires. The information about the "suspend" and "resume" connector settings are reported to the steps inside the composite step again via the inner root connector.

In a step, a corresponding "outer" declaration of connector "root" is present and the code of section 4.1 of a step is slightly changed to:

```
protected
  outer CompositeStepStatePort root;
  CompositeStepStatePort localRoot;
equation
  connect(localRoot, root);

  localRoot.activeSteps =
                 if active then 1 else 0;
  active    = pre(newActive);
  newActive =
     if localRoot.resume then oldActive
     else (anyTrue(inPort.set) or
         active and not
         anyTrue(outPort.reset))
        and not root.suspend;

  when localRoot.suspend then
    oldActive = active;
  end when;
```

Via outer flow variable activeSteps, the active setting is reported to the composite step. Additionally, a memory is introduced via variable "oldActive" to remember the current value of the "active" flag when the composite step is terminated via its "suspend" port ("**when** localRoot.suspend **then** ..."). The assignment to "newActive" is slightly changed to include the transitions via the "suspend" and "resume" connectors in the composite step.

A composite step may contain not only steps but other composite steps. The implementation above does not handle this case. In fact, with the Modelica language version 2.1 it is not possible to provide a proper implementation. Therefore, an extension was needed that is defined in the coming version 2.2 of the Modelica language (it is already supported in Dymola):

In a composite step a construct of the following form would be needed:

```
// wrong Modelica code
inner CompositeStepStatePort root;
outer CompositeStepStatePort root;
```

where the "inner root" connector is used in all steps inside the current composite step and the "outer root" connector refers to the composite step outside of the current scope in order to have a communication channel to the outside scope. However, this is wrong Modelica code because there are two declarations with the same name. Note, the names must be the same, because in a step a communication channel to the “nearest” composite step is needed and the name used in the “outer” declaration of a step must be identical to the name used in the “inner” declaration of a composite step.

In the Modelica language version 2.2 the following extension was introduced:

```
// Modelica 2.2 code
inner outer CompositeStepStatePort root(..)
```

to define actually a new “inner” variable “root” and at the same time reference an “outer” variable “root”. **References** to “root” inside the current scope, references the “**outer**” variable. **Modifications** to “root” are not allowed for “outer” variables and therefore apply to the “**inner**” variable. In other words, inside a composite step the “outer root” is accessed by variable “root” and settings for the “inner root” have to be performed via a modification in the declaration of “root”.

The previous code fragments must be slightly modified to include the new “inner outer” declaration, and to handle the case of composite steps that are inside and/or outside the current one.

5 Summary

The free Modelica.StateGraph library offers new features to conveniently define discrete event and reactive systems in Modelica models. Since Modelica is used as an action language, a Modelica translator can guarantee that a StateGraph has deterministic behaviour. StateGraph models can be combined with components of any other Modelica library and can therefore be very easily used to control a continuous plant.

StateGraph is based on Grafchart, which contains several features that not, so far, have been implemented in StateGraph. Some of these features, such as function chart procedures, assume support for dispatching at run-time, which does not match well with the philosophy of Modelica. Other features such as lists could very well be included in StateGraph.

It is also planned to improve the graphical handling of StateGraphs in the future and to add more functionality especially also to the Modelica.Blocks.-Logical library that is often used in a StateGraph. Improvement suggestions and contributions are welcome.

References

[1] Årzen K.-E. (1996): Grafchart: **A Graphical Language for Sequential Supervisory Control Applications**. IFAC'96, Preprints 13th World Congress of IFAC, San Francisco.

[2] Årzen K.-E., Olsson R., and Akesson J. (2002): **Grafchart for Procedural Operator Support Tasks**. Proceedings of the 15th IFAC World Congress, Barcelona, Spain.

[3] David R., and Alla H. (1992): **Petri Nets and Grafcet: Tools for modeling discrete event systems**. Prentice Hall.

[4] Dressler I. (2004): **Code Generation from JGrafchart to Modelica**. Master Thesis, ISRN LUTFD2/TFRT-5726-SE, Department of Automatic Control, Lund Institute of Technology, Sweden.

[5] Elmqvist H., Mattsson S.E., and Otter M. (2001): **Object-Oriented and Hybrid Modeling in Modelica**. Journal Européen des systèmes automatisés, vol. 35, no. 1, pp. 1-22.

[6] Harel D. (1987): **Statecharts: A Visual Formalism for Complex Systems**. Science of Computer Programming, Vol. 8.

[7] Mosterman P.J., Otter M., and Elmqvist H. (1998): **Modeling Petri Nets as Local Constraint Equations for Hybrid Systems Using Modelica**. In Proceedings of SCS Summer Simulation Conference, pp. 314-319, Reno, Nevada, July 1998.

Using Modelica and Control Systems for Real-time Simulations in the Pulp & Paper industry

Johan Bäckman Mattias Edvall
Metso Paper Sundsvall AB
851 94 Sundsvall, SWEDEN
johan.backman@metso.com mattias.edvall@metso.com

Abstract

Metso Paper regularly uses process simulations when designing and delivering a new pulp mill. The simulations have two main purposes. Operators are trained in running the new process before it exists and complex control logic can be validated efficiently before start-up. The simulations are built using Dymola/Modelica, executed in real-time and connected to the plant's control system.

This paper will discuss the general technical solution for using Dymola/Modelica in combination with a control system for real-time simulations. The paper will also provide an overview of the different applications that Metso Paper has implemented. Expectations of future development of Modelica and connected software will be discussed from an industry perspective.

Keywords: dynamic simulation; control system; pulp and paper; operator training; logic verification

1 Introduction

Different kinds of simulations are used within Metso Paper for designing and developing the pulp and paper processes. Static simulations for balancing and dimensioning the process with respect to flows, steam, chemicals etc. The second type is advanced and computational demanding FEM-type of simulations for more detailed simulations and analyses of different optional machine designs. The third type of simulations is the real-time dynamic process simulator for operator training and control system verification. This paper will focus on the real-time dynamic process simulations. A training session using simulators is shown in Figure 1.

Figure 1. Simulator training in progress

When delivering a new machine, process area or a complete mill Metso Paper has offered dynamic process simulations since beginning of the 1990's. The main purpose of these simulations is to train operators and other mill personnel in running the new equipment in an efficient and optimal way. The operators get used to the new operator displays, interlocking logic and most importantly new process dynamics. Difficult and rare process conditions can be introduced in the simulations. By regularly exposing the mill personnel to these difficult situations in the simulator environment, expensive and unwanted stops in the real production can be avoided.

Earlier these types of simulations were made in a single PC where the process simulations as well as a mimic of the real operator displays and the plant's control system was configured. Even if this system worked fine technically speaking and served its purpose as an educational tool, it had some drawbacks. The two major problems with this solution were:

- It was a very expensive solution due to the many engineering hours needed in order to convert the real operator displays and control logic into the simulator world. When new revisions of displays and control logic were made, new revisions must be made in the simulator as well. On top of this, it was very difficult to reuse models and configurations between projects.

- It was difficult to keep the simulator up to date once delivered to a customer since every change in the real control system and operator displays must be followed by a corresponding change in the simulator system. Typically, the changes in the real process were done by the mill personnel while changes in the simulator demanded involvement from other competencies. In practice the simulator and real world configurations slowly drifted apart.

Starting in beginning of year 2001 a new architecture for dynamic process simulations within Metso Paper were developed using Modelica and Dymola. The main difference between the old and new architecture is that in the new architecture the real control system and real operator displays are used instead of including a mimic version of the two into the simulator. The simulation models are also designed in a way that modular building blocks of typical process equipment and areas easily can be reused from project to project. The advantages of the new architecture are substantial

- Since no mimic is done of the operator displays and control logic the engineering hours has reduced dramatically. The modular design of the new simulation models has also contributed to lower the engineering hours since much can be reused from project to project. The operators get to use the real displays and real control system when doing simulator training, and no changes needs to be made in the simulator when changes are made to the displays and control logic.
- It is much easier to keep the simulator system up to date since the real control system is used. Changes made to the displays and control logic can be transferred into the simulator system directly by the mill personnel.

Another big advantage with the new architecture is that the simulator system can be used to validate the upper level control system and mill control system before start-up. This has traditionally been a very time and resource-consuming task and even then, the quality of the validation has been difficult to verify due to the complexity of the control logic. Errors in the control system, as well as in the dimensioning of process equipment, are corrected easily and effectively early in the projects. Considerably reductions in test time have been noticed since starting using simulators for test purposes. As the simulations get more accurate it is also possible in some areas to pre-tune PID controllers using the simulator. Starting the real mill with verified control logic, trained operators and pre-tuned PID controllers are a great benefit for Metso Paper customers.

2 The Simulator system

2.1 Architecture

The new simulator architecture is shown in Figure 2.

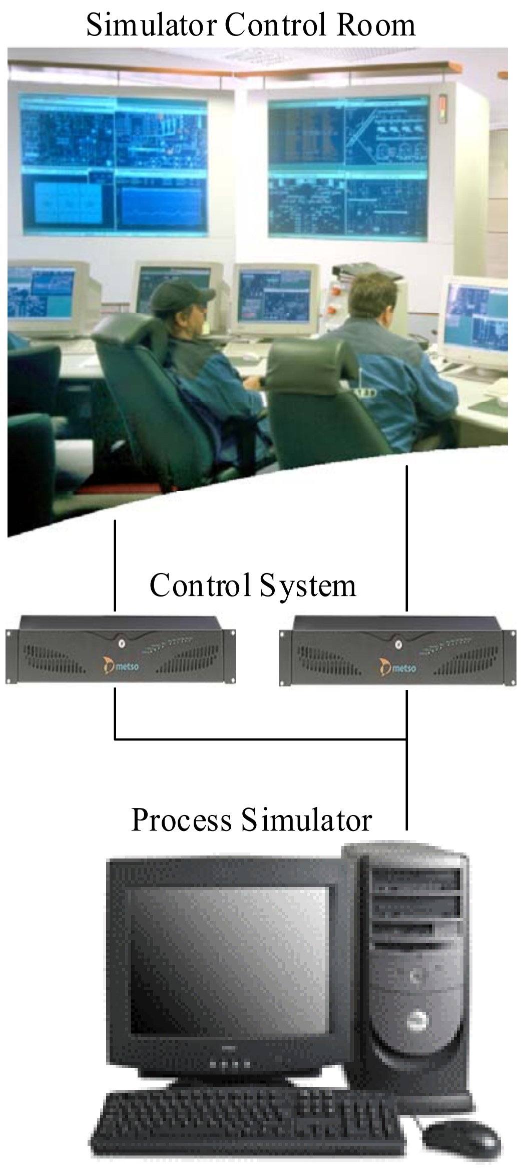


Figure 2. The new simulator architecture

A Metso Paper simulator system typically consists of a control system that is the same control system used to control the real process (example: metsoDNA, ABB Industrial IT, Siemens PCS7, Emerson DeltaV, etc). Another computer executes specific process simulation models in real-time. The operators are using the same operator displays as they would in the real control room. The system can be extended to include several computers that simultaneously are running different simulations towards the same control system. Real control system hardware can be used to execute the control logic and operator displays. However, a modern control system usually offers a software version (example Metso's VirtualDNA) of the control system, which can be more convenient to use. Included in the control system are all control logic, graphical displays, PID controllers, alarm limits, interlocking diagrams, trend displays etc. Everything that an operator will have once running the real process is already there since it is the same system being used.

2.2 Models

Using Modelica and Dymola a specific model library has been developed including models that make it possible to build complete simulations of Metso Paper machines and processes. Examples are SuperBatch^{TM} cooking, Washing and Screening, Refining, Pulp Bleaching and Pulp Drying area. In Figure 3 a small part of the washing and delignification process can be seen.

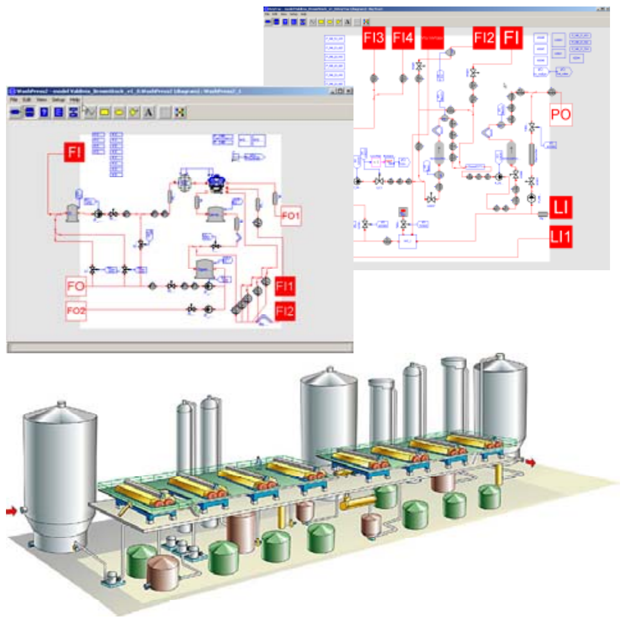


Figure 3. A part of pulp making process

The library includes models of Metso Paper specific machinery such as SuperBatch^{TM} digesters, DeltaCombi^{TM} screens, Refiners, TwinRoll^{TM} presses, OxyTrac^{TM} delignification towers and Pulp Drying machines, but also more general equipment such as valves, pumps, pipes, heat exchangers, tanks, etc. Together the models are used to build up simulations of complete pulp mills. Creating a large net of equations defining the flows, pressures, temperatures, chemical consumption, pulp consistency and other process values, all values will react depending upon the actions taken by the operator using the control system. A complete simulation includes all process machinery and instrumentation, starts with chips enter the digester area, and finish when dried and bleached pulp exits the drying machine.

2.3 Communication

The simulation models are compiled with Dymola forming a Windows application that is used as a real-time DDE Server. Modern control systems offer an OPC Server in order to open communication with any OPC Client. The Metso Paper simulator system includes special software to transfer the signals from the DDE Server in the run-time simulation to the OPC Server in the control system. Since practically all modern control systems have opened up to OPC technology the communication link can be used without changes no matter what control system the customer selects. The only configuration that needs to be done is a cross-reference list between the signal names in the simulator and the corresponding signal names in the control system. Typically are process values like flows, pressures, temperatures, pH, consistencies etc, sent from the simulations to the control system. The control system writes values like valve openings, starting orders for pumps and motors etc, back to the simulation.

Modelica models for I/O communication have been developed. They are using the input/output qualifiers to accept values from, and to give values to, the control system via the communication link.

2.4 Teacher interface

The extended functionality of the communication link includes an interface for a teacher. From this software a teacher can operate the dymosim application, start, pause and stop the simulation. It is also possible to start the simulations with different initial positions, empty tanks, almost full tanks etc. making it possible to train how to get out of difficult process situations without losing production.

Different kind of scenarios and disturbances can be applied to the simulations to investigate how operators solve and detect common process problems. Problems like drifting process values, malfunction in valves, blinding of screens, channeling in reactors, web break in the dryer section as well as other criti cal situations can be trained. Figure 4 shows screen displays from the teacher interface.

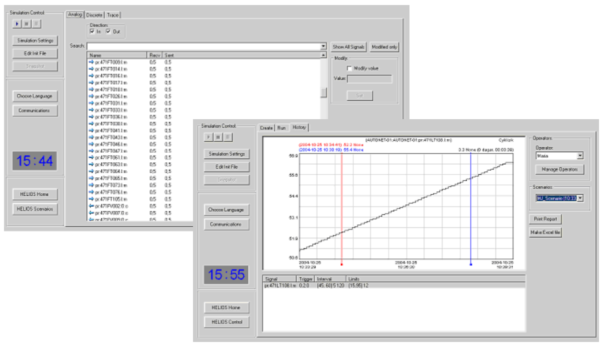


Figure 4. Teacher interface

A large database is used to collect data from the training sessions and each operator can evaluate their actions and compare the results with earlier sessions.

2.5 Performance

Due to the complexity and scope of the process to be simulated the simulation model files and I/O communication lists becomes large. For example when simulating a SuperBatch^{TM} digester area the I/O list consists of about 1800 signals, the executable simulation file is close to 40 Mb. Figure 5 shows messages from a translation of a digester area model.

```
 STATISTICS

Original Model
 Number of components: 20084
 Variables: 137615
   Constants: 1574 (0 scalars)
   Parameters: 35394 (69071 scalars)
   Unknowns: 100647 (100658 scalars)
   Differentiated variables: 9146 scalars
 Equations: 52124
   Nontrivial : 39885

Translated Model
 Constants: 53502 scalars
 Free parameters: 17018 scalars
 Parameter depending: 19966 scalars
 Inputs: 0
 Outputs: 0
 Continuous time states: 9146 scalars
 Time-varying variables: 42482 scalars
 Alias variables: 24590 scalars
 Number of mixed real/discrete systems of equations: 18
 Sizes of linear systems of equations: {2, 2, 2}
 Sizes after manipulation of the linear systems: {0, 0, 0}
 Sizes of nonlinear systems of equations: {5, 732, 35, 35, 66, 38,
7, 14, 35, 7, 35, 18, 66, 18, 35, 7, 14, 35, 7, 14, 7, 7, 14, 35, 7, 35,
7, 7, 5, 1, 7, 23, 6}
 Sizes after manipulation of the nonlinear systems: {1, 89, 4, 4,
9, 2, 1, 3, 4, 1, 4, 2, 9, 2, 4, 1, 3, 4, 1, 4, 1, 1, 3, 4, 1, 4, 1, 1, 1, 1,
1, 3, 1}
 Number of numerical Jacobians: 0

Finished
```

Figure 5. Translation of a digester area model

The digester area is about 25% of the total pulping process. It is possible to run these kinds of simulations on a PC. However, the speed of the internal memory bus is vital to simulation performance. Due to the slow nature of pulping processes, it is common that the simulations are run for several days without interruptions.

For training purposes it is very important that the system is robust and that the accuracy is high enough for the operators to trust the simulations. However, it is important to keep in mind that the purpose is not to simulate each process component as accurately as possible. The purpose is to produce a simulation of a large area including a large amount of equipment that will give an operator the right look and feel for the dynamics in the mill.

3 Practical experiences and future development

Today the simulator system has been connected to the following control systems: metsoDNA, ABB Industrial IT, Emerson DeltaV and Siemens PCS7. There are of course advantages and disadvantages with all of these systems when it comes to engineering efficiency and costs, but from our point of view; there are no technical differences. The same communication link has been used without modification, and the signal transferring between an OPC Server included in the control system and the real-time simulations has shown a similar performance with all mentioned control systems. Two complete simulator systems have been delivered so far. Figure 6 shows operator training in Chile. During 2005, three more deliveries are expected.

Figure 6. Simulator training in Chile

During the internal development work and practical training sessions with Metso Paper customers around the world some development ideas have emerged. The ideas, if emasculated, would heavily increase the value and competitiveness of using Modelica and Dymola in combination with control systems for real-time simulations. Some of the most important areas to strengthen in order to meet industry demands in the future are as we see it:

- Replacing the Dymosim DDE Server with a modern OPC Server will open the simulation environment to all modern control systems and increase the communication performance significantly. Many OPC Clients are available on the market, some even free of charge. Viewing and changing simulator values during run-time simulation would be facilitated.
- A possibility to see and change simulation values from the modeling user interface during run-time simulation would be very time saving and valuable during development of large simulation models.
- Increased debugging functionality. When a model for some reason crashes after several days of simulation, it is very hard to find the reason quickly. The large simulation models and long simulation time makes it difficult to store values for debugging due to the huge amount of data collected.

4 Conclusions

Dynamic process simulation is a powerful tool when educating new operators in running a pulp mill. Successful real-time simulators using Dymola/Modelica with control system in the loop have been used for control system validation and for operator training in Metso Paper processes. The strong modular focus in Dymola/Modelica have been helpful in reusing simulation models and thereby shortened the engineering hours in new projects.

Some proposals of improvements to Dymola/Modelica has been made in order to strengthen the focus, increase competitiveness and general awareness of using Dymola/Modelica in industry applications.

Session 7c

Modelica Interfaces

Usage of *Modelica* for transmission simulation in ZF

Jochen Köhler Alexander Banerjee
ZF Friedrichshafen AG
Graf-von-Soden-Platz 1, D-88046 Friedrichshafen, Germany

Abstract

At ZF transmission models are an essential component within the development process. To guarantee an efficient use of modeling know how as well as a persistent use of models in different simulation environments, Dymola has been declared as a standard at ZF and a central model component library ZFlib has been developed accessible to all business units. This paper gives an overview of some features of the library, such as easy parameterization of models independent of the environment in use and the export of complex models into environments with simple integration algorithms. Furthermore a short description of the automatic testing of the library which guarantees a software development process at a high level of quality will be given.

Keywords: Model Export, Parameterization, Automatic Testing

1 Introduction

For ZF as an automotive vendor for transmission components and systems, modeling and simulation has a long tradition. A long time ago the modeling of transmission systems was mainly done by experts of the business units which often used different tools and different approaches. In the meanwhile there has been an exponentially increasing demand for models and model components which should be accessible to and usable even by non-experts. Due to this fact ZF started a centralization process with the intention to standardize the modeling approach of transmission systems and merge the modeling know-how of components and systems in one library accessible to all business units. A further requirement has been to guarantee a persistent use of models in different simulation environment (e.g. *SIMULINK* or *dSpace*), which actually demands for models given as C-Code (Figure 1). Beside this, the tool under consideration

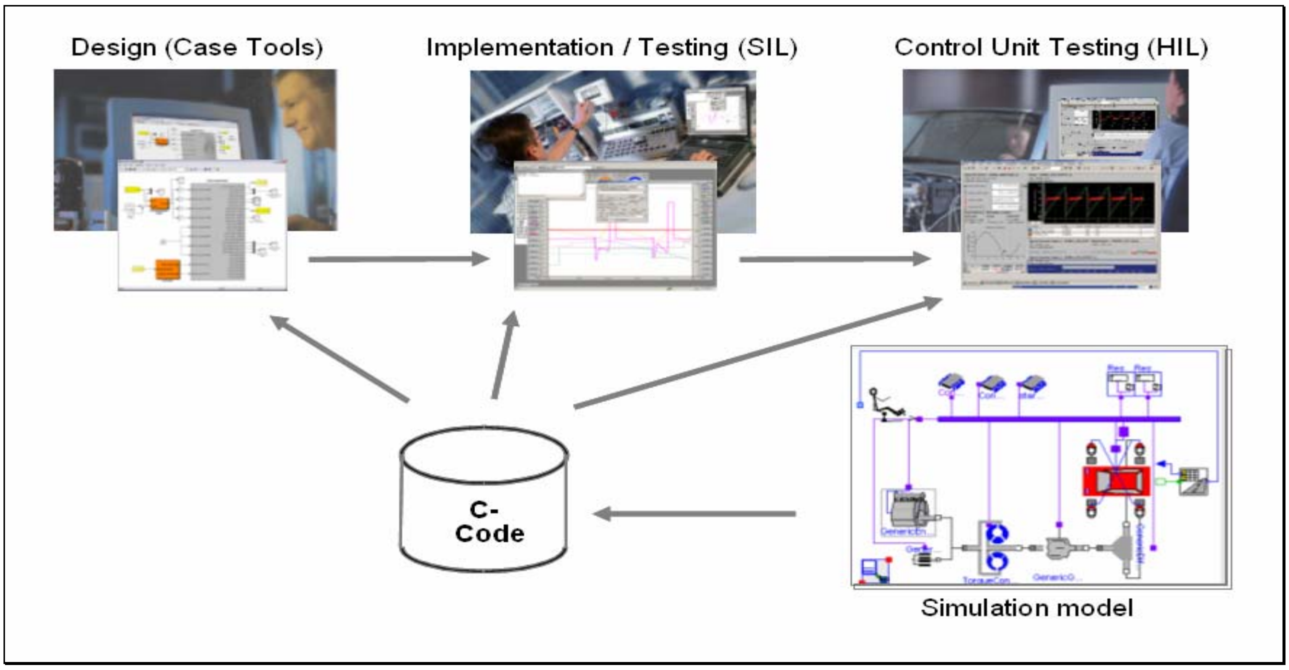


Figure 1: Exporting models to various environments

should offer the user an interactive graphic interface for modeling. Since Dymola offers such an interface, allows the continuous extension of libraries by using the Modelica language and enables the user to export models as C-Code, ZF came to the decision to define Dymola as a standard for transmission system modeling.

This was the starting point for the development of a *Modelica* library called `ZFlib` which is an extension of the commercial library `Powertrain`. This library has a lot of models of ZF specific components as well as some add-ons for the export to other simulation tools.

The focus of this article is a short explanation of the ZFlib and how models can be parameterized independently from the environment in use (chapter 2), how the export of complex models into environments which only allow the use of discrete integration algorithms (chapter 3) has been realized and how an automatic testing of modules can be performed whenever a modification within the library has occurred (chapter 4).

2 Usage of `ZFlib`

2.1 `ZFlib` as code generator

The main issue in using the `ZFlib` is to set up models that can be used within different environments – not only with *Modelica* Tools like *Dymola* or Math-*Modelica*. *Dymola* is used to translate the models developed with *Modelica* to C code.

The generated C code in form of the `dsblock` function can be integrated into the demanded environment. This is done already by Dynasim for *Simulink* with a special interface-module called DymolaBlock.

Nevertheless we did some modifications of this module to meet our special demands. Other interfaces for further environments (e.g. ASCET and some ZF programs) were developed to wrap the different calls (e.g. initialization, update …) from these environments to the appropriate functionality of the `dsblock` function.

One big drawback with the generated code of a standard *Modelica* model is that we can't parameterize the model in *one*, easy way in different environments. Another aspect is, that we want to use a standardized way of parameterization in ZF for these models. The basic idea was to separate models and parameters from each other. The parameter values should be stored in ASCII-files according to a standardized format. At initialization the files are read by the model for parameterization. Therefore some extensions had to be done to read the files and parameterize the modules in an easy way.

2.2 Performing parameterization in different environments

The format of these ASCII-files has a very primitive syntax to define scalar or vectors as well as characteristics with variable dimensionality. One parameter is defined simply by a triple: Identifier, Unit and Value (see Figure 2).

```
J1   [kgm^2]  0.1
; scalar parameter
InU  [-]      0   1   2
OutY [-]      0   1   2
; two vectorial parameters
Test_Table2D[
[-]   U1 [-]  0   1    2
2     Y  [-] -2  -1    0
1     Y  [-] -1   0    1
0     Y  [-]  0   1    2
Test_Table2D]
; Two-Dimensional-Table
```

Figure 2 Example of an ASCII-file

A large database of parameter data exists already in ZF that shall be used also for the *Modelica* models. We implemented a *Modelica* package to read these files and made the data available to other modules.

The functionality consists of two parts:

First, we implemented some basic *Modelica* functions that serve as an interface to a set of C-functions, which do the data-management.

Second, the C-functions handle the reading of the needed files, collect all parameter data specified there and make them available within *Modelica*. For tables, also the interpolation is done in C-Code.

To use this functionality, users have to do the following things:

1. Specify the files to be read:

By including the `Load` block, the user specifies all the files to be read in a list. At initialization all parameters are scanned and stored in database.

2. Use scalar or vectorial parameters

With the ZFlib-functions `GP(String identifier)` and `GV(String Identifier)`, the model finds the data of interest through the defined data identifier within the database. If the asked identifier is not found in the database, the simulation aborts.

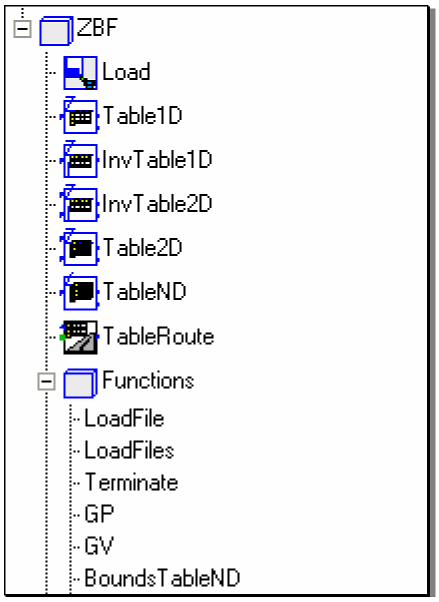

ZBF
Load
Table1D
InvTable1D
InvTable2D
Table2D
TableND
TableRoute
Functions
LoadFile
LoadFiles
Terminate
GP
GV
BoundsTableND

Figure 3 ASCII file package in the `ZFlib`

3. Use tables

For using tables we have implemented three different Table blocks for one-, two and N-dimensional Tables. They also get access to the tables within the database through some interface functions. With the function (`BoundsTableND(String Id)`) the boundaries of the table input vectors can be queried. In addition to the interpolation, the gradient (one dimensional) or normal vector (n-dimensional) can be calculated on demand. This is very useful for some forward control algorithms.

Another useful feature is the possibility to convert the ASCII data to SI units. This is needed because very often data are saved in units like [rpm] or [mm] and *Modelica* handles only SI units.

The implementation of these features was done in C++. The sources or binaries can be included in various environments and therefore, this kind of parameterization can be used everywhere, where code can be included.

Using this functionality allows the user of the model to change the parameters easily by editing the used ASCII files. The changed parameters will be used at the next simulation run without a new translation of the model.

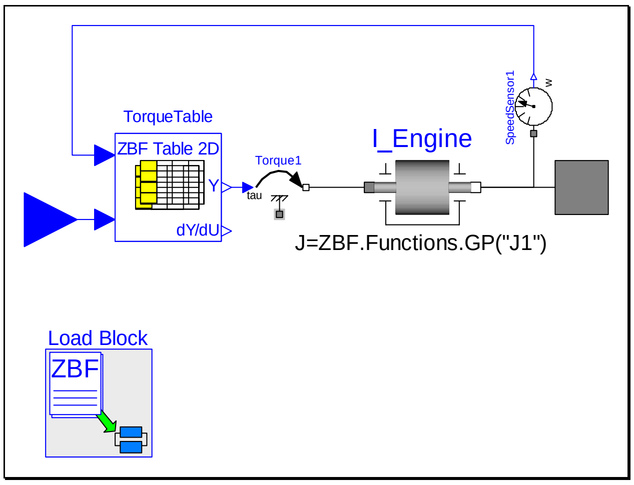


Figure 4 Example of a model using ASCII file package

2.3 Extending from the `Powertrain` library

The `ZFlib` was extended from the commercial library `Powertrain.` The maximum benefit can be achieved by reusing the included elements as often as possibly. Nevertheless, we want to use our own tables within the models in the `Powertrain` library. Due to this, modifications have to be done for a few models within the `Powertrain`: The *Modelica* keyword `replaceable` in front of all declarations of tables had to insert.

Another modification concerns the usage of the bus. The concept of the `Powertrain` bus has a lot of advantages: The implementation of complex models is easily done and quite transparent. It's very easy to change big parts of a complex model without intensive modifications. We decided to extend our own bus called `ZFBus` from the Bus in the `Powertrain`. To be able to use the control units in the `Powertrain` together with this extended bus it was also necessary to add the `replaceable` keyword to all bus declarations with control unit implementations. It would be more convenient, if it would be possible in *Modelica* to link connectors that extend from one root. Another approach to make combinational usage of different libraries easier could be a hierarchical structure of one global bus.

With this second modification we can use all models in the `Powertrain` and our library seamlessly. Certainly it would be a great help to add these modifications to the original `Powertrain` without any disadvantage for other users.

3 Using *Modelica* models with discrete algorithms in *Simulink*

Simulink is used more and more to develop control algorithms for transmissions in ZF. The simulation of systems with a digital controller requires the use of a discrete, fixed step integrator. On the other hand, there is a very strong demand to use complex and often numerically stiff models of transmissions to test these controls in simulation. Hence the use of numerical integrators with varying step size is necessary.

These transmission models are modeled in *Dymola* using the `ZFlib`. Because of the arguments mentioned above, the original DymolaBlock for *Simulink* can't be used unmodified in conjunction with the fixed step integrator from *Simulink*.

3.1 Making the Dymolablock "discrete"

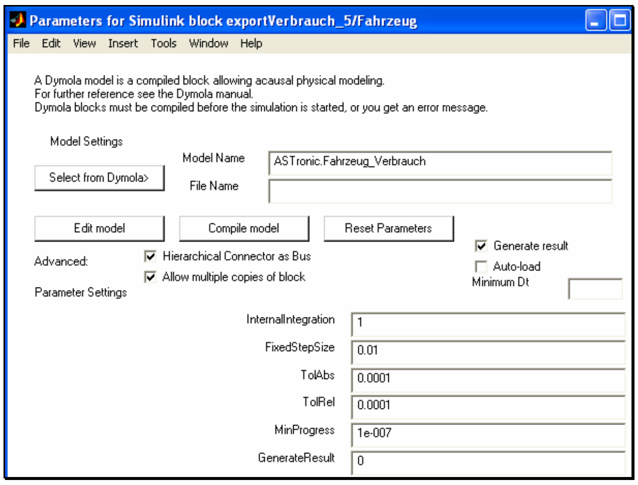
So, we decided to embed a continuous variable step integrator within the S-function of the DymolaBlock. We used the same DASKRT integrator as in *Dymola* [1]. To make the modified DymolaBlock look like a discrete Block, the S-function just returns that no continuous states are needed. Then it's easily possible to combine this model with discrete controllers in Simulink.

The DASKRT is used exclusively for the exported Modelica model inside the S-function. Every time the outputs are demanded from Simulink, the S-function calls the internal integrator to integrate up to the new time. During this interval several events may happen, that have to be handled. Additionally we have to make sure, that the DASKRT stops correctly at the given time point. Otherwise it may happens, that the integrator takes a step size larger than the sample interval of the *Simulink* model. This would lead to outputs which refer to a later point in time with respect to the actual integration time.

3.2 Controlling the behavior of the DymolaBlock

A simple model was implemented in *Modelica* as a parent to be able to control the behavior of the modified DymolaBlock. The declared parameters can be read by the DymolaBlock. The user can enable or disable the internal integration by setting the parameter InternalIntegration to 1 or 0. So it's possible to use the model also in continuous environments or with the simple fixed step integrators of Simulink.

The FixedStepSize specifies the sample time of the block. Furthermore you can specify the tolerances of the integration algorithm. MinProgress is an abstract criterion to abort the simulation if the simulation progress is too low.


Parameters for Simulink block exportVerbrauch_5/Fahrzeug

A Dymola model is a compiled block allowing acausal physical modeling.
For further reference see the Dymola manual.
Dymola blocks must be compiled before the simulation is started, or you get an error message.

Model Settings

Model Name: ASTronic.Fahrzeug_Verbrauch
File Name:

Advanced:
[✓] Hierarchical Connector as Bus
[✓] Allow multiple copies of block
[✓] Generate result
[ ] Auto-load
Minimum Dt

Parameter Settings

InternalIntegration	1
FixedStepSize	0.01
TolAbs	0.0001
TolRel	0.0001
MinProgress	1e-007
GenerateResult	0

Figure 5 The user can change the DymolaBlock behaviour easily by changing the mentioned parameters

With GenerateResult the user can enable the logging to the dsres.mat file if needed. This is more convenient than the check box in the original DymolaBlock because you can change it without compiling.

The interface between *Simulink* and the *Modelica* model consists of two connectors that contain all input or output signals. It's possible to use the same or similar connectors for different models because they contain the same signals as the "real software connectors" in a vehicle. Another advantage is that existing algorithms can be connected easily to these models.

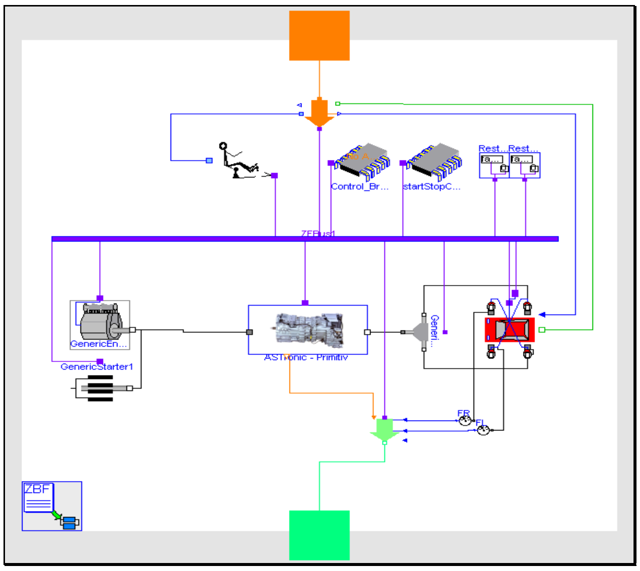


Figure 6 Example for a model that can be exported to *Simulink* with one input connector (top) and one output connector (bottom).

As one can see in Figure 6, the interface between *Simulink* and *Modelica* consists exclusively of directed input or output signals. This is caused by the analogy to a connector for a control unit but it's also a design decision to have no coupling on a "physical layer". Due to this fact, there is no problematic coupling of two integrators as it can be with "normal co-simulation".

4 Automated Testing of `ZFlib` components

One important aspect in implementing basic elements for a library which will be used heavily in simulation, is to be sure, that everything works correctly. Based on experience, modifications on library elements that worked well before can cause strange errors in a complex model. To avoid this, we introduced basic "single element tests". For (almost) every ZFlib element at least one test exists. Such a test shall stimulate an element in a way that a specific behavior can be checked.

For more complex elements, it is necessary to set up several tests with different parameters or with different structures to cover all possible "areas". After the tests have been set up, the developer has to judge, if the simulation result is correct. This result and the corresponding parameters are saved as a reference for later comparison. Of course, this work has to be done very careful!

This approach leads to several benefits. The original developer had to make sure that the new module works as required. The second benefit is the possible use of the tests as an example on how to use the tested element. As mentioned, the third benefit is the possibility to test this element after modifications against the old references.

Also, if a developer finds a bug in an element later on, he can set up new tests that check, if the bug still exists. Testing with these new tests, the developer can be sure, that the same error can't pop up again after another modification.

These tests are useful only, if they are run on regular basis. They should be performed and automatically compared with the reference results of all elements. Therefore we implemented some Matlab scripts, which scan the *Modelica* code for existing tests, run them automatically and compare the results with the reference files.

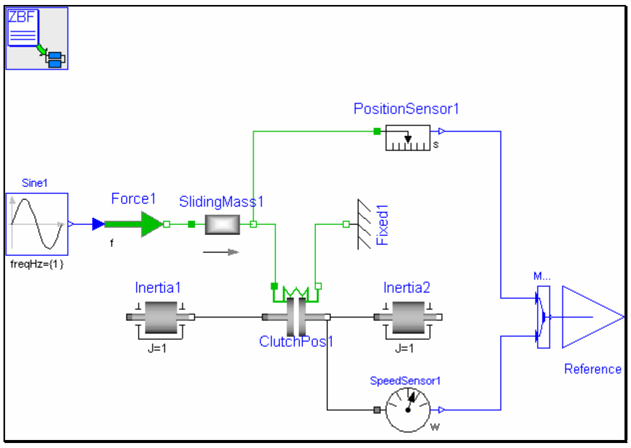


Figure 7 Test for a clutch with an opening/closing spring.

4.1 Collecting tests

To add a test to the automatic procedure it must fulfill two conditions: It has to be inside a package called Tests and it has to be extended from the basic model `ZFlib.Tools.TestModel`. This model defines some tolerance parameters and an `OutPort` reference to the signals which are used for comparison.

```
// preparation
clear
openModel("../dymola/zflib/package.mo")
checkModel("ZFlib")
cd ../dymola
// newModel M_Limiter
translateModel(
"ZFlib.Blocks.NonLinear.Tests.M_Limiter")
// newResultTest
RunScript(
"../TestReferences/M_Limiter_Ref.mos")
simulate
$copy  dsres.mat ..\temp\M_Limiter.mat
// endTest
// newModel M_RateLimiter
translateModel(
"ZFlib.Blocks.NonLinear.Tests.M_RateLimiter")
// newResultTest
RunScript(
"../TestReferences/M_RateLimiter_Ref.mos")
simulate
$copy  dsres.mat ..\temp\M_RateLimiter.mat
// endTest
```

Figure 8 *Modelica* Script built by the test collector

For each of these models, the collecting function searches for reference parameter files (exported from dsin.txt). These were saved earlier with the result reference files. For every parameter file that can be found, the model is simulated and tested against the corresponding reference result. The collecting of the tests leads to a *Modelica* script file that can be run by *Dymola*.

4.2 Performing tests

The test script is parsed by another *Matlab* script that calls the *Modelica* instructions sequentially and tests the results. Every line in this script is a test! First of all the whole package is syntactically checked. After this, each test model is translated. If this was successful, the simulation runs can be performed.

```
// preparation

// newModel M_StarterPrimitive
translateModel(
"ZFlib.Engines.Tests.M_StarterPrimitive")
// .. Caused an Error
// endTest
// newModel M_EngineTableWithBrake
translateModel(
"ZFlib.Engines.Tests.M_EngineTableWithBrake")
// newResultTest
RunScript(
"../TestReferences/M_EngineTableWithBrake_Ref
.mos")
// .. Caused an Error
// endTest
// newModel M_Clutch
translateModel(
"ZFlib.Mechanics.Rotational.Tests.M_Clutch")
// newResultTest
RunScript(
"..\TestReferences/M_Clutch_Ref.mos")
simulate
$copy  dsres.mat ..\temp\M_Clutch.mat
// endTest
// COMPARE DIFF M_Clutch   Correlation Coef-
ficient <0.99s.mat ..\temp\M_KK_3.mat
// endTest
```

Figure 9 Result script with failed commands and error description

The testing environment can test against the result file and against the log file (dslog.txt). In the result file, only signals that end in the `Outport` called Reference are tested (see Figure 7). Every command that was performed successful is appended to a text file. In the same way every command that causes any error is stored in another file. Nevertheless, the whole test script is executed. After the run has been completed, the user can take a look at the test result log files and decide on further actions.

The result log files can be run in the same way as the first test script. The log file provides some explanations of the errors, which supports the user in doing the corrections.

5 Conclusions

The `ZFlib` library has been used successfully in several projects working with various simulation environments. The ASCII file parameterization is well accepted. With the modified `PowerTrain` library we can use a lot of components and save time for implementing ZF-own models.

The enhanced “discrete” *Dymola*-Block works fine and quite fast (also in combination with ZBF parameterization).

The automated testing needs some extra effort. But on the other side, errors can be found quicker and other users get a better understanding of the tested element.

6 References

[1] L. R. Petzold. A description of DASSL: A differential-algebraic system solver. In R. S. Stepleman, editor, Scientific Computing, pages 65–68, Amsterdam, 1983, North-Holland.

Implementation of a Generic Data Retrieval API for Modelica

Dr. Michael M. Tiller
Ford Motor Company, Research and Advanced Engineering
Dearborn, MI USA

Abstract

For model developers, the Modelica modeling language is a valuable tool for describing the behavior of dynamic systems. However, developing models and performing analyses as part of a large scale engineering operation involves much more than creating behavioral descriptions [1],[2]. In order to integrate modeling and simulation into a typical product development process it is necessary to extract data (*e.g.* product information, part geometry, controller calibrations) from external sources.

This paper will describe an application programmer interface (API) for data retrieval that has been developed using the standard external function interface in Modelica. The API is composed of generic functions that can be implemented to extract data from a variety of external data sources. Such an API can be used to access data for material properties, part geometries, data tables, *etc*.

While the interface definitions are generic, our implementation of the generic API was specifically developed to retrieve data stored in XML [3] and utilizes the `libxml2` library [4] to retrieve and parse XML files containing product information. Furthermore, the API queries are performed using XPath expressions [5].

Currently, there is no standard API to allow Modelica models to retrieve information from external data sources. Hopefully this paper can demonstrate the power of such capabilities and prompt further discussion on formalizing a standard API with similar functionality.

Keywords: XML, XPath, HDF, MATLAB, Java

1 Introduction

Data is an integral part of modeling. Because Modelica is so often used for physical, first-principles modeling, there is typically a need to provide design data for numerous individual components. Such data is often available "somewhere" (we will use the term *external data source* as a generic term for sources of such data) but it must be collected to populate the Modelica model.

Because there is no standard way in Modelica to access such external data sources, this data is typically either entered by hand for each component or aggregated and organized into Modelica record definitions. We will refer to data managed in this way as a *Modelica representation* of the data.

While Modelica representations can be used, there are numerous drawbacks when trying to integrate the resulting models into large scale engineering and analysis processes. For example, such data often already exists in an external data source and copying it into a Modelica representation is both tedious, redundant and error prone. It also makes models that depend on the Modelica representation of the data difficult to update as new data becomes available. The best approach is one that retrieves the data as needed from the centralized external data sources. For example, if product information is stored in a relational database somewhere within a company, the ideal situation would be that the information could be automatically extracted directly from that database.

Another problem with Modelica representations of the data is cataloging large collections of component data. Representing such data in Modelica means, in practice, that large datasets are loaded into the modeling environment when only a very small percentage of that data is used. For example, we have data for a large number of production engines. The space required to store the data for each engine is considerable. We currently store all this information in a hierarchy of Modelica records. Storing the data in this way means slower loading times and higher memory consumption even though any given analysis only requires the data for one particular engine. Another issue with cataloging the data is querying the data set to see what information is available. While most data management systems include formalized query systems, there is no functional equivalent in Modelica to query languages such as SQL, XPath or XQuery [6].

Often times, different characterization data is needed for the same component depending on the desired level of fidelity. Representation in Modelica often results in a variety of record definitions associated with a given physical component (*i.e.* one for each level of fidelity). Typically these record definitions include large amounts of redundant data between them. However, because of the semantics associated with records[1] in Modelica, it is difficult to eliminate such redundancy.

Use of data expressed directly in Modelica typically results in that data being "hard-wired" into the resulting simulation. Although the data can be changed it is typically a manual process and impractical for large data sets. Ideally it should be possible to load the data on demand from a data source in the event that such a data source has been changed or updated.

Another concern is storage of the data. A centralized data source is often accessed over the network. As such, the data is only stored in one place. This not only conserves space but also provides a definitive source for the data. If data is represented in Modelica there is the risk that variations will develop across multiple copies of the data. Loading data on demand over a network provides a more dynamic system for data management.

Some applications require very large data sets to be available but only use relatively small chunks at any given time [7]. In such a case, a system that is able to load data into memory for use by a model on an "as needed basis" can save a considerable amount of space (*e.g.* in results files). By avoiding the need to represent the entire dataset in Modelica and the compilation process (*e.g.* symbolic analysis) avoids the need to read in and analyze such data.

2 Interface

For these reasons, we have developed an API that allows us to retrieve data from external sources. This is not a "database API" because it does not include the complete set of operations typically associated with database interactions (*e.g.* changing data, committing transactions, *etc*). Instead, the focus for this package is on retrieval only. The interface is generic so it could be mapped to a wide variety of external data sources (including, but not limited to, databases).

In this section we will go through the API in detail to explain the basic functionality before moving on to a discussion of our implementation of the interface and some examples of its use.

The data retrieval API is implemented within a package called `DataRetrieval`. The package contains several class definitions that extend from the ExternalObject class used for handling opaque references to external (*e.g.* C language) data. In addition, it contains several functions that operate on these locally defined data types.

2.1 Opening and Closing a Data Source

In order to access a data source it is first necessary to open it for queries by instantiating an object to represent the data source. This object can then be used in subsequent query operations. To open a database, a `Source` object must be created, *e.g.*

```
import DataRetrieval.*;
parameter Source s=Source(
                          format="…",
                          url="…",
                          context="…");
```

where `format` identifies the format of the data source (*e.g.* "XML"), `url` is a string encoded using the *uniform resource locator* (URL) syntax [8] and `context` is used, in a data source specific way, to limit the scope of subsequent queries.

The ExternalObject interface also provides for a destructor although that is not called directly so the details are not included.

2.2 Query Expressions

Once a data source is available (in the form of a `Source` object), queries can be made against it. Because Modelica is a strongly typed language and it is currently not allowed to overload functions, query functions are defined for specific data types (*i.e.* String, Real, Integer and Boolean) and for specific dimensionalities (*e.g.* scalars, vectors, matrices, *etc*). But, each query function relies on a common query expression syntax.

The precise semantics of the query expressions do not necessarily have to be defined for each data source. The generic aspect of the API does not interact in anyway with the semantics of these expressions. As we will discuss shortly, our implementation uses XPath expressions for such queries.

[1] Specifically, the strict requirement that assignment is only possible between identical record types.

2.3 Retrieving Scalars

For each built-in type, there is a function to extract scalars of that type from the data source. For example, the function `getReal` is invoked as follows:

```
import DataRetrieval.*;
parameter Real x = getReal(
                    source=SourceObj,
                    name="…");
```

where `source` is a `Source` object, `name` is a query expression to identify the scalar. The function returns a scalar of the type indicated by the name of the function. In a completely analogous fashion, similar "get" functions are defined for each other built-in type.

The assumption is that the underlying, data source specific layer will interpret the query expression according to the semantics for that source and will perform the necessary casting of the data into the appropriate built-in type.

2.4 Retrieving Arrays

Unlike scalars, extracting data from arrays cannot be done in a single step for several reasons. First, we do not know, *a priori*, the size of an array to be extracted from an external source. So if the entire array is to be loaded, we must first establish the size of the data and then use that information to declare the Modelica variable to hold the data. Furthermore, it is often the case that we may only wish to extract a portion of an array and in some cases which particular portion may change during the simulation (*e.g.* table lookup).

For these reasons, a special type of `ExternalObject` must be created to represent arrays. Creating an instance of such a type loads the entire array into *memory* along with information on the size of each array dimension. For each built-in type there is a type definition in the `DataRetrieval` package formed by concatenating the built-in type name with either `Vector` or `Matrix`. For example, a matrix of `Reals` would be defined by `RealMatrix`. Creating instances of these special objects is accomplished by calling the constructor, e.g.

```
import DataRetrieval.*;
parameter RealMatrix props =
  RealMatrix(source=engine,
    rows="…", cols="…");
```

where `source` specifies the data source, `rows` is an expression to identify all rows to be loaded and `cols` is an expression (typically evaluated within the context of each row) to identify individual columns.

Once these special objects have been created, data can be extracted from them by using the 'extract' function. Because it is not possible in Modelica to define methods associated with these objects, functions have been defined to perform such extractions. The name of the function is constructed by prepending `extract` to the name of the special type from which the data will be extracted. For example,

```
import DataRetrieval.*;
parameter Real y[…,…] =
     extractRealMatrix(mat=MatrixObj,
                       i=5:17, j=1:3);
```

where `mat` is a special array object (whose type is consistent with the name of the extraction function, *e.g.* `RealMatrix` in this case), `i` represents the row indices of the elements to be extracted and `j` represents the column indices of the elements to be extracted.

While it is quite simple to write special functions to provide the size of the data associated with these array objects, it is not particularly useful for reasons that will be discussed later in Section 6.

2.5 Retrieving Records

Given the ability to extract scalars and arrays from data sources, we can do more sophisticated things like populating records with information from data sources. Ideally, it would be possible to automate the process of loading records with information from external data sources. However, such functionality would require some degree of integration with the underlying tool or some kind of reflective capabilities in the Modelica language itself. Nevertheless, it is certainly possible (using the functionality already described) to load records from data sources using specially written functions (see Section 4 for an example).

As previously eluded to, data sources are not likely to match Modelica records definitions exactly. Typically such external sources will include information that is not modeling related. Of the information that is modeling related, only subsets may be useful depending of the model's particular level of fidelity. For this reason, the fact that information in a data source does not have to match the Modelica record definition exactly (either in naming or in structure) is a useful advantage. It means that a single data source can be used for modeling and non-modeling related information and that information for different levels of model fidelity can be grouped together in the same data source.

2.6 Querying Available Choices

Our data retrieval API is built around the idea of query expressions. In most of the previous sections it is assumed that the query expression is written to match exactly one piece of data. However, allowing query expressions to match multiple pieces of data can be quite useful because it would provide tools with the ability to identify all data that is potentially compatible for a specific data type. For example, when loading records that characterize electric motors it is useful to query a data source for compatible data and use it in the same way that the `choices` family of annotations are used.

The current version of our API does not provide such functionality for two reasons. First, such functionality would require tool support. The other reason is that such functionality would require certain concepts (*e.g.* ordering, filtering, *etc*) not current expressed in the data retrieval API. In Section 6 we will discuss how such capabilities could be implemented with some degree of tool support and a slightly more sophisticated querying scheme.

3 Implementation

Up to this point, the discussion has been completely generic with only a few fragments of actual code and only vague discussions on query expressions. In the next two sections we will describe an implementation of the API and get into specific detail about how it can be used.

Our implementation was developed specifically to extract data from XML documents. Such documents may exist on web servers or they may be stored in local files. XML is fast becoming an important technology in all aspects of computing because of its ability to structure information in an operating system, programming language and application neutral way. In addition to existing high quality implementations [9], there are several advances on the horizon that will support handling of large collections of binary data [10], [11] (*e.g.* simulation results).

We treat each XML document as an object-oriented database (OODB). An OODB is useful for storing heterogeneous collections of objects. In our experience, engineering data (part dimensions, test data, *etc*) fits quite well into OODBs.

But storing the data is only one aspect that we need to worry about. The other aspect is querying our data source to extract data. For this, our implementation uses XPath, a standard for "addressing parts of an XML document". XPath provides a standardized way of identifying what data in an XML document we wish to extract. A similar emerging standard is XQuery [6] which may prove to be a superior (and mostly backwardly compatible) technology once it is formally standardized.

Consider the sample engineering database shown in Figure 1. We will use this trivial database to demonstrate the capabilities of the XPath standard. Although the data and structure of the database are quite simple, all these examples could be applied to much larger databases without alteration.

```
<?xml version="1.0"?>
<engineering_data>
  <product name="ZY300">
    <base_motor type="C12"/>
  </product>
  <part>
    <motor name="C12">
      <rotorJ>0.011</rotorJ>
    </motor>
  </part>
</engineering_data>
```

Figure 1: Sample Engineering Database

Let's begin with a simple example. Imagine we wanted to extract the name of the base motor used in the ZY300 product. Using the query expression:

```
  //product[@name='ZY300']
   /base_motor
   /attribute::type
```

to query the database shown in Figure 1 would return '`C12`'. The '`//`' at the start of the request means "at any level in the document hierarchy". The '`product`' string following this is interpreted as the name of the element type that is being requested. Anything contained in '`[]`'s represents a predicate. Elements for which the predicate is false will be filtered. In this predicate '`@name`' represents the attribute '`name`'. So the first line locates the ZY300 product in the database. Each subsequent '`/`' in the expression is used to indicate traversal one level deeper into the hierarchy. If a name is prefixed by 'attribute::' that indicates that the query is for an attribute rather than an element. So the complete expression can be interpreted as "Search the hierarchy for product elements whose name attribute is ZY300 and for each of these find the base_motor element immediate below it and return the value of the type attribute for that element".

Imagine we wish to extract the value contained between the rotorJ tags in Figure 1. We can extract that data with the following XPath expression:

```
   //motor[@name='C12']
    /rotorJ/text()
```

This query is quite similar to the previous query except it uses the 'text()' function to return the textual content within the rotorJ element.

But now let's look at a more challenging example. In the second example, we assumed that we knew the model name for our motor, C12, *a priori*. Imagine we want to extract the rotational inertia of the rotor but we don't know the motor name. Instead, what we know is that it is the base motor used in the ZY300 product? In this case, we can combine the two queries we made previously into:

```
 //motor[@name=
   //product[@name='ZY300']
    /base_motor
    /attribute::type]
  /rotorJ/text()
```

With this example we have nested our requests for the type of base motor used inside a predicate used to search for the motor. By using the query for the ZY300 base motor type in the predicate involving the motor name, we were able to identify rotor inertia based on its relationship to the ZY300 product rather than by name.

These are a few examples of the kinds of queries that are possible with XPath expressions. This is by no means a complete introduction to XPath. Instead the goal of these examples was to provide sample expressions so that expressions in subsequent examples can be interpreted.

It is important to note that parsing XML, converting it into a traversable data structure and implementing an XPath query engine are not trivial tasks. Fortunately, there are multiple implementations of these standards that can be used as off-the-shelf software components. For our work, we chose to use the libxml2 [4] library that was developed for use with the Gnome desktop environment. The libxml2 library includes complete, robust implementations of many XML related standards including DOM [12], SAX, XPointer [13] and XPath [5].

4 Examples

4.1 Retrieving Parameter Values

One of the most common uses of the data retrieval API is to supply parameter values in a model. In this section we will show how the data retrieval API can be combined with a sample data set (shown in Figure 2) and specific query expressions (using the XPath notation described in Section 3) to accomplish this task.

```
<?xml version="1.0"?>
<engine_data>
  <engine name="Beta">
    <real name="bore">88.2</real>
    <stroke>84.0</stroke>
    <val name="conrod" units="mm"
         value="125.0"/>
  </engine>
  <engine name="Gamma">
    <real name="bore">87.2</real>
    <stroke>85.0</stroke>
    <val name="conrod" units="mm"
         value="123.7"/>
  </engine>
</engine_data>
```

Figure 2: Sample Engine Data

An important thing to note about the engine data shown in Figure 2 is that each parameter (bore, stroke and conrod) are represented using different XML constructs. The engine bore is represented as the text inside a generic real element, the stroke appears as the text inside a special element type of its own and the connecting rod length is given by an attribute associated with another generic element type, val, but with a specific string, conrod, given for its name attribute.

So the challenge in this example is to show how XPath syntax is expressive enough to allow us to address each piece of data even though the contexts are quite different. Figure 3 shows the various XPath expressions that can be used to extract the necessary data from the XML file.

```
model TestReals
  import DataRetrieval.*;
  parameter Source engine =
    Source(format="XML",
           url="engines.xml");
  parameter Real bore =
    getReal(source=engine,
      name="//engine[@name='Beta']
             /real[@name='bore']
             /text()");
  parameter Real stroke =
    getReal(source=engine,
      name="//engine[@name='Beta']
             /stroke/text()");
  parameter Real conrod =
    getReal(source=engine,
      name="//engine[@name='Beta']
             /val[@name='conrod']
             /attribute::value");
end TestReals;
```

Figure 3: Parameter Extraction

4.2 Populating Records

The example shown in Figure 3 includes several complex XPath expressions. Because it can be difficult to formulate such expressions and because entering them manually or copying and pasting them several times can be error prone and/or difficult to maintain, it is desirable to try and encapsulate these expressions somehow. One way to accomplish this in Modelica is to create a special record type for the data and then create a function that can populate such a record automatically. For example, consider the following Modelica record definition:

```
  record EngineData
    import Modelica.SIunits.*;
    parameter Diameter bore;
    parameter Length stroke;
    parameter Length conrod;
  end EngineData;
```

To populate such a record with data from the file shown in Figure 2, we could write a function that constructed such a record from the name of the engine and the location of the data. Figure 4 shows what such a function might look like.

```
function Load
  import DataRetrieval.*;
  input String engine;
  input String url;
  output EngineData data;
protected
  String context=
   "//engine[@name='"+engine+"']";
  Source source=
    Source(format="XML", url=url,
           context=context);
algorithm
  data.bore :=
    getReal(source=source,
      name="real[@name='bore']
            /text()");
  data.stroke :=
    getReal(source=source,
      name="stroke/text()");
  data.conrod :=
    getReal(source=source,
      name="val[@name='conrod']
            /attribute::value");
end Load;
```

Figure 4: Populating a record

The function shown in Figure 4 also highlights another feature of the data retrieval API. When the `source` object is created, the optional `context` argument is used to define the context in which all subsequent XPath expressions should be evaluated. What this means in practice is that it is assumed that any queries associated with the `source` object apply only to the specific engine for which data is being retrieved. In this way, the query expressions for each invocation of `getReal` can leave off the engine selection prefix expressions resulting in shorter path expressions.

Once defined, the `Load` function shown in Figure 4 can then be used to provide values for a record without any need to include calls to the data retrieval API or any XPath expressions, e.g.

```
  parameter EngineData engine =
    Load(engine="Gamma",
    url="engines.xml");
```

4.3 Loading Arrays

While loading a complete array into Modelica for use in a model is an obvious example of how the data retrieval API might be used, there are also other reasons why you might want to load only a partial array. Consider the case of cubic interpolation. Imagine we have interpolation data that is stored in an array as follows:

$$\begin{bmatrix} x_1 & f(x_1) & f'(x_1) \\ \cdots & \cdots & \cdots \\ x_n & f(x_n) & f'(x_n) \end{bmatrix}$$

Now, if we need to construct the cubic polynomial approximation for any value x, we only need to know the values for the function and its derivative associated with x_i and x_{i+1} (where $x_i<=x<=x_{i+1}$). The important point is that we do not need to load the entire matrix into a Modelica variable. Instead, we could simply extract the values that we need at any given time and construct the approximations in a piecewise form. So given the following data file:

```
<?xml version="1.0"?>
<data>
  <function name="z">
    <point x="0" f="0" df="0"/>
    <point x="1" f="0" df="1"/>
    <point x="2" f="1" df="0"/>
    <point x="3" f="0" df="-1"/>
    <point x="4" f="0" df="0"/>
  </function>
</data>
```

We can use the following Modelica code to evaluate the function “z” described in the data file:

```
  parameter RealMatrix data =
    RealMatrix(source=f,
      rows="//function[@name='z']
             /point",
      cols="attribute::x
            |attribute::f
            |attribute::df");
  Interpolate2 y(x=time,
                 data=data);
```

This code defines the contents of the matrix using the XPath expressions and then passes it to a model which only extracts the function and derivative values for the two closest points at any given time. Now, formulating a cubic polynomial approximation for a simple 1D function does not necessarily require such powerful functionality. However, if we wanted to construct a 3D approximation for a relatively large data set [7] using a complex cubic interpolation scheme this API could help us minimize memory consumption while still exposing the underlying mathematical structure.

4.4 Generating “choices”

As mentioned previously, an XPath expression might match several different entities in an XML document. For example, if we wanted to extract the names of all engines present in Figure 2, we could express this with the XPath expression:

```
//engine/attribute::name
```

The results of such a query could then be used in subsequent queries to select from elements in an XML document. As mentioned previously, this capability would require some degree of tool support.

5 Discussion

5.1 Alternative Source

While the data retrieval API is generic, the implementation discussed in this paper assumes that the data will be represented natively in XML and the query expressions will follow the XPath specification. But there are several other formats that are frequently used to store data and for which a retrieval API might be useful. Examples of these would include HDF [14] and the MATLAB “.mat” file formats [15].

The only significant impact of changing the format of the underlying data source is on the query expressions. There are two ways to approach query expressions in such cases. First, for each format a (potentially) unique query expression syntax could be used. This would allow, for example, SQL to be used if the underlying data source was a relational database. The drawback of this approach is that it would be impossible to write general functions (*e.g.* the Load function for loading engine data shown in Figure 4) for an arbitrary data source. Instead, a function would have to be defined for each potential data source format.

On the other hand, if each data source used the XPath approach for querying, then a consistent syntax would be available across the various platforms. The advantage of this is that users would only need to be familiar with XPath and no other query expression format. The difficulty is that XPath applies to

XML, not to other formats. One way to bridge the gap would be to define a mapping from each format to XML. For example, consider the following MATLAB code which writes several matrices to a file:

```
>> A = [1, 2, 3; 4, 5, 6];
>> B = [6,7; 8,9; 10,11];
>> save 'AB.mat' -V4 A B
```

The contents of the file `'AB.mat'` are stored in the MATLAB specific format. But for the purposes of formulating queries we could create a mapping that defines a translation to XML that would result in an XML document that looks like this:

```
<?xml version="1.0">
<MATLAB>
  <matrix name="A">
    <row><col>1.0</col>…</row>
    <row><col>4.0</col>…</row>
  </matrix>
  <matrix name="B">…</matrix>
</MATLAB>
```

In this way, it would then be possible to load data from MATLAB using the data retrieval API with code like

```
parameter Source f =
  Source(format="MAT4",
         url="AB.mat");
parameter RealMatrix data =
  RealMatrix(source=f,
    rows="//matrix [@name='A']
           /row ", cols="col");
```

Note that the data itself would not necessarily have to be translated into XML. Instead, a special XPath interpreter could be developed for each format that understood the "mapping" involved.

5.2 Data Management

The goal of this API is not just to provide a package for opening and querying data sources. In addition, the design goals are also meant to address nagging problems with handling data in Modelica. With this new API we can avoid loading large amounts of data either as constants or definitions in packages (*e.g.* Modelica.Media idea gas data) and we can avoid (through the selective extraction functions) loading entire data sets into Modelica variables when only a subset are needed at any given time during a simulation.

In addition, data stored in Modelica typically ends up being compiled into simulations. In a sense, the data is then frozen inside the analysis. Any change in the data requires the model to be recompiled or have its input files modified in some way. By relying on external data sources, the "fresh" data can be loaded on demand.

5.3 Modelica Deficiencies

While the external function interface in Modelica provided enough functionality to implement the API and create functioning examples, there are still a few areas where Modelica could be improved.

First, the API structure would benefit greatly from support for methods that can be invoked on user defined classes. Without methods, special functions much be written and type information about arguments and return types must somehow be aggregated to form unique function names. In addition, features to support better abstraction and polymorphism support would allow specialized `Source` objects to be developed (e.g. `XMLSource`, `HDFSource`) but remain compatible with all existing functions that required `Source` objects as arguments. As things stand currently, the definition for the `Source` class must be familiar with all potential formats (hence the `format` argument) but with the ability to subclass, new formats could be supported without the need to change or update the existing `Source` definitions.

Another issue with Modelica is units. While the language allows unit information to be associated with variables and data sources may include unit information, the current API specification does not exploit any of this information. Built-in unit conversion capabilities in Modelica might make it possible to handle units without having to implement any manual unit conversions.

Finally, the XML related tools used in this implementation were available in C and could be integrated nicely through the Modelica external function interface. However, more and more of these capabilities are appearing in Java. As things currently stand, it is not possible to leverage Java code through the external function interface although it would be nearly trivial to do so. By including an instance of a Java Virtual Machine in Modelica tools and/or generated code, it would be possible to easily load Java classes into memory and invoke functions (and perhaps methods) defined in Java. Simply defining how arguments are passed to and from Java code would enable leveraging tremendous amounts of existing Java code.

5.4 DTDs and Namespaces

Two features of XML not discussed in this paper are Document Type Definitions (DTDs) and namespaces. This section includes some discussion about these topics and how they relate to our data retrieval API.

DTDs define a specific schema associated with an XML document. We could have forced all XML data to be used with the data retrieval API to follow a specified DTD. This would have made retrieval considerably simpler because we could have anticipated, to a greater degree, the structure of the data we were trying to retrieve. However, it is quite impractical to expect that external sources of data will always conform to a specified DTD. It is possible to translate such data from its native format into a form that conforms to a specific DTD but this would likely involve more work than our approach and would still involve XPath or something similar. The strength of our approach is the ability to use it in conjunction with arbitrarily structured data.

Namespaces could also be useful in annotating existing datasets with new element types that are explicitly tagged to be specially included for our purposes. In such a scenario, special tags could be defined within a namespace and then added to existing XML documents. These specialized elements should, in theory, be ignored by other applications since they belong to a namespace that the application is unfamiliar with. This would add a level of complexity to the implementation and the need to specially annotate external data sources but without any real benefit. For this reason, we did not utilize namespaces.

6 Benefits of Standardization

While we have created this implementation for our own purposes based on identified needs in our organization, it is quite likely that many Modelica users would benefit from a standard data retrieval API like the one described in this paper. In this section, we highlight some of the benefits a standard API would have over the “user space” implementation we have created.

First, query expressions could be used to generate lists of “choices” much like the existing `choices` annotation. Such functionality would have to be available (*i.e.* compiled into) Modelica tools in order to link such information to the graphical user interface. The current external function interface is, at least in the case of Dymola, limited to user simulations and such external functions are not available to the Dymola process itself.

Another advantage of a standard data retrieval API is that it could be used within the standard Modelica libraries to manage data. For example, the Modelica.Media library contains a tremendous amount of data associated with ideal gases. This data could be stored outside the Modelica environment and loaded selectively on an as-needed basis.

As mentioned previously, our API is implemented through the Modelica external function interface and, as such, is not available to the Dymola GUI. This makes model checking and model compilation impossible for cases where variables in Modelica are dimensioned based on calls to the API (*i.e.* to determine the full size of an external matrix). By standardizing the API, it would be possible to use external data to dimension variables used in Modelica.

7 Conclusions

In order to integrate Modelica models with existing engineering and analysis processes, retrieval of data from external data sources for use in models is essential. This paper outlines one way such integration can be accomplished. Use of XPath expressions is a powerful component of our implementation and, through formalized mappings as described in Section 5.1, this approach to querying can be extended to other non-XML based data sources as well. Our implementation focuses only on XML documents as data sources and represents only a proof-of-concept implementation (*e.g.* no caching is performed in our implementation).

Standardization of this API opens up many possibilities for integration of database information into graphical model development environments. It could also automate the tedious and error prone process of writing special functions (like the one shown in Figure 4) to populate records used to characterize models.

The topic of a formalized API for retrieving external data sources has come up occasionally in Modelica design meetings. Hopefully this implementation can serve as a starting point for further discussions, proposals and eventually standard functionality available to all Modelica users.

References

[1] Tiller M., Bowles P., and Dempsey M., "Development of a Vehicle Model Architecture in Modelica". *Proceedings of the 3rd International Modelica Conference*, Nov. 3-4, 2003. Linköping, Sweden.

[2] Tiller M, "Parsing and Semantic Analysis of Modelica Code for Non-Simulation Applications". *Proceedings of the 3rd International Modelica Conference*, Nov. 3-4, 2003. Linköping, Sweden.

[3] "Extensible Markup Language (XML)", http://www.w3.org/XML/

[4] "The XML C parser and toolkit of Gnome", http://www.xmlsoft.org/

[5] "XML Path Language (XPath) Version 1.0", http://www.w3.org/TR/xpath

[6] "XQuery 1.0, An XML Query Language", http://www.w3.org/TR/xquery/ (working draft)

[7] Newman C. E., Batteh J. J., and Tiller M., "Spark-Ignited-Engine Cycle Simulation in Modelica", *Proceedings of the 2nd International Modelica Conference*, March 18th-19th, 2002. Oberpfaffenhofen, Germany.

[8] Berners-Lee T., Fielding R., and Masinter L., "Uniform Resource Identifiers (URI): Generic Syntax", RFC 2396. http://www.ietf.org/rfc/rfc2396.txt

[9] "Berkeley DB XML 2.0", Sleepycat Software, http://www.sleepycat.com/products/xml.shtml

[10] "XBIS XML Information Set Encoding", http://xbis.sourceforge.net/

[11] "XML-binary Optimized Packaging", http://www.w3.org/TR/xop10/

[12] "Document Object Model (DOM)", http://www.w3.org/DOM/

[13] "XML Pointer Language (XPointer)", http://www.w3.org/TR/xptr/

[14] "HDF 4.1r3 User's Guide", http://hdf.ncsa.uiuc.edu/UG41r3_html/

[15] "MAT-File Format, Version 7", http://www.mathworks.com/access/helpdesk/help/pdf_doc/matlab/matfile_format.pdf

External Interface to Modelica in Dymola

Hans Olsson
Dynasim AB
Research Park Ideon
S-223 70 Lund, Sweden

Introduction

Dymola [1] provides an integrated environment for modeling, simulation and scripting based on the Modelica language. However, Modelica is not always the best choice for GUI-design, database access, or canned presentations of libraries (in the last case the usual choices are PowerPoint® presentations, pdf-documents, or animation files).

The solution in these cases is to leverage other tools, such that they can interface *to* Dymola's Modelica functionality and vice-versa for call-backs/links *from* Dymola to external tools. These solutions are already implemented for the forthcoming version Dymola 6, except for a few specific items described as future plans (these might be implemented in time for the release of Dymola 6). Some solutions are also available in previous versions of Dymola as described in the manual [1].

1 Model callbacks in Dymola

One goal of the external interface is to allow a model developer to customize commands for the model by calling external tools. This functionality is provided in two separate parts:

- External tools can be interfaced to any Modelica function.
- Models can be customized to have commands calling Modelica scripts or functions in Dymola.

As a concrete example a model developer can add model-specific commands to Dymola's Commands-menu. Users of the model can then call a command from the Commands-menu, which for example executes a Modelica script calling external Modelica functions implemented in C.

1.1 Variants of commands

The commands can be generic (independent of the selected model, e.g. to select a specific demo model or check that the model fulfils some guidelines), or specific to the currently selected class (e.g. post-process the simulation result).

The command can be called explicitly by the user (from the Commands-menu in Dymola), or implicitly to extend existing functionality (the enable-field in parameter-dialogs is one example where a user can enable the input field based on a predicate call-back). In the future, callbacks will be generated at specific stages of translation, e.g. for users to gather additional statistics of the use of specific models.

Obviously we could for a specific example provide the functionality inside Dymola, but by providing an API and callbacks we allow the customer to extend Dymola. Thus the API to Modelica structure which is presented later in the paper is intended to also be useful for e.g. gathering statistics about the components used in the translated model. Some of these functions need access to browser information (such as the current class) as will be discussed below.

1.2 Calling functions directly

A specific case of running a command is to call a function related to the model e.g. to run an optimization of the model.

The advantage with directly using function calls is that there is no need for any model-specific script files (making it easier to e.g. copy the Modelica model) and that a function call is part of the Modelica language and thus syntactically correct.

Furthermore it is possible to optionally prompt the user to modify the arguments of the function call before the function is called, e.g. to specify the operating point to optimize for. This uses the normal (and extensible) function call dialog.

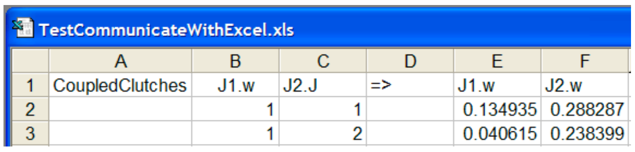
2 Communication protocols

The interprocess communication is between two running programs one of which is Dymola. The transport mechanism can be seen as separate from the structure of the messages. Currently Dymola can send and receive DDE-messages. For the future extension of exchanging XML-data in SOAP-encoding SOAP-HTTP is a suitable alternative [2], and is portable to non-Windows platforms (for which the demand is increasing).

2.1 DDE-interface

For DDE-execute, the return value does not allow meaningful result values and thus specially formatted DDE-Request(s) is used for returning data to other programs with a special case for Matlab (below).

Thus Windows programs can call DDE-routines to e.g. perform a parameter sweep from Excel (by programming in Visual Basic for Application). In this case the update of the excel spread-sheet is done by running a macro and there are no links in the Excel-document, which only contains the start-values (J1.w), parameter-values (J2.J), and final values for two variables (J1.w and J2.w):


TestCommunicateWithExcel.xls

	A	B	C	D	E	F
1	CoupledClutches	J1.w	J2.J	=>	J1.w	J2.w
2		1	1		0.134935	0.288287
3		1	2		0.040615	0.238399

Figure 1 Parameter-sweep from Excel

The macro opens a DDE-channel to Dymola, sends the command simulateExtendedModel as a DDE-request to simulate and get the final values of variables. To access the spread-sheet the Excel-routine `Sheets( "Sheet1").Cells(r, c).Value` is used to get and set the values.

It is also possible to use DDE to communicate directly with a running simulation, Dymosim, (provided the compiler option 'Visual C++ with DDE' is selected). This is described in [1] and also allows automatic update of variables after changes.

These protocols are extensible which allows calls between two programs running on different computers, i.e. remote procedure calls. Although remote procedure calls are beneficial, for security reasons remote procedure calls must be explicitly enabled (as is necessary for remote DDE).

Limitations of DDE

Unfortunately DDE has some restrictions (in addition to being platform-specific), in particular on the maximum length of the messages, and no general high-level API for communicating structured data.

For communication with a running simulation (DDE-communication between Dymola and Dymosim) we have found it necessary to use special formats to achieve the high bandwidth needed for e.g. online animation of Modelica models, while respecting the limitations of the protocol.

We do not anticipate similar bandwidth needs for the communication with Dymola, since the natural way of communicating a vector of values is to send it as *one* DDE-message (which automatically solves most of the performance problems).

The DDE-interface in the caller is preferably written as one generic routine (as we have done in Matlab) to make it easy to later extend it e.g. with handling of messages exceeding the maximum length, and optimized alternatives to the CF_TEXT-format.

2.2 Direct interfacing

The above handles the complex case of interprocess communication between two running programs, but sometimes a simpler mechanism suffices.

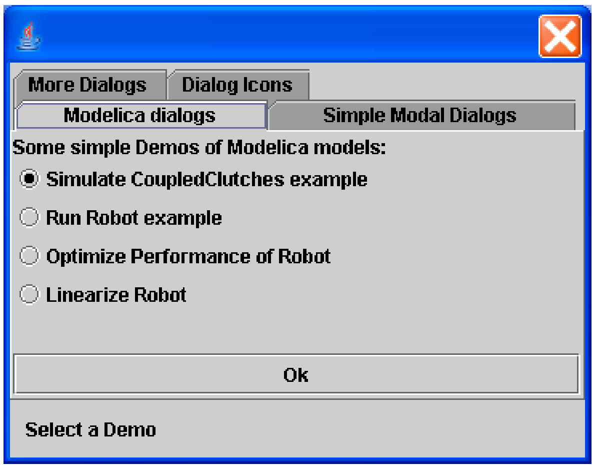
2.2.1 Call of external functions

The Modelica language offers the possibility to directly interface to C and FORTRAN-functions such that calls of Modelica functions declared as external C/Fortran are mapped into calls of the corresponding C or Fortran functions.

There is no restriction on the use of external functions in Modelica and to allow easy use of them in the interactive environment Dymola performs demand-compilation of external functions. Thus a user can call external functions in the same way as non-external functions.

This C interface provides an extensible mechanism that also handles other languages that can give routines C linkage, such as C++ and languages that provide an interface for calls from C, such as Java. Since the JNI interface to Java allows dynamic loading of Java-libraries this could be done internally in Dymola making it possible to directly call a Java function from Dymola to e.g. show a modal dialog and get the user response back without using any external programs.

Below we demonstrate running a Java function showing a modal dialog box, where the call of the Java-functions has been included as an external function call in Modelica (with suitable arguments), and then compiled by Dymola (the JNI-implementation require that calls in translated C-code use the Visual C++ compiler).



Some simple Demos of Modelica models:

- (●) Simulate CoupledClutches example
- ( ) Run Robot example
- ( ) Optimize Performance of Robot
- ( ) Linearize Robot

Select a Demo

Figure 2 Calling a modal dialog in Java from a Modelica function

Extending Modelica's external interface to directly include Java in addition to C and Fortran 77 is straightforward and the specification was deliberately written to allow such extensions. By using single-quoted names it is possible to directly support hierarchical external function name (i.e. containing a dot) as in Java.

2.2.2 Linking to libraries

It is also possible to link with specific C-libraries (including Dynamic Link Libraries, DLLs). Due to the limitations of the C-compilers used, any libraries must be provided in a binary format compatible with the C-compiler used to compile the Modelica code in Dymola (Visual C++ 6/7, egcs, or Watcom). Provided the external code is portable and available in source-form that is in general possible. Additionally Modelica models are often downloaded and run on realtime platforms, which require different libraries (or that the C-code is provided in source-form and linked together with the model).

Another potential problem on Windows is that some API calls require that the caller is a Windows-program.

This is e.g. necessary to use the DirectX interface from Microsoft. An application of DirectX is to allow users to control a car-model from VehicleDynamics [6] by a steering wheel. In those cases a Windows program must be generated (in Dymola this currently requires that you select the compiler Visual C++ with DDE) and special routines obtain the window handles.

2.2.3 Calling programs

Modelica.Utilities.System [3] enables functions written in Modelica to call external programs. Command line arguments to Dymola enable external tools to for instance run simulations in Dymola.

The Commands-menu, Dymola's Execute-function, and links in the documentation layer also allows opening other files than Modelica scripts using the file associations in Microsoft Windows. This is useful for canned presentations, and selecting a menu entry will automatically open the file in the corresponding tool (e.g. pdf-documents in Acrobat Reader®, animation files in the media player, html-files in the browser).

3 Data-structure encoding

To communicate Modelica data-structures in Dymola to other tools the data-structures must be mapped into other data-structures. Following the C and FORTRAN-interface this is defined in a generic encoding for each interface, i.e. there is no need to specify a mapping for each data-structure. If a special mapping is desired for a specific case that can then be done either in a Modelica function or in the other tool.

The basic idea of the interface is to return a string that when evaluated returns the value, e.g. a numeric value is returned as itself, i.e. 3/2 is returned as the string '1.5' (without the quotes).

For more advanced data-structures, arrays and records, it is necessary to define how the resulting string is evaluated. The two variants that are implemented are Modelica data-types constructors and Matlab.

3.1 Mapping to Modelica

The Modelica-mapping is identical to the output format used in Dymola's command-window and makes use of record and array constructors for complex data-structures. Consider the examples (input in bold and the response-string is given after the '='):

```
Matrices.inv([1,2;3,4])
=
[(-2.0), 1.0;
1.5, (-0.5)]
```

```
GetClassAttributes("Modelica")
 = Dymola.AST.ClassAttributes(
        fullName = "Modelica",
       isPartial = false,
     isProtected = false,
      restricted = "package",
        isInner = false,
        isOuter = false,
  isEncapsulated = false,
    isShortClass = false,
   isReplaceable = false,
    isRedeclared = false
)
```

The mapping is sufficiently straightforward that we will not go into details of it, and if the result is pasted into Dymola's command input and evaluated it returns the same result once more.

To use this functionality the application programmer has to set up a DDE-channel to Dymola, and send a Modelica-function call as string in a DDE-request. Dymola's DDE-server will respond with a string containing the result as a Modelica data-structure.

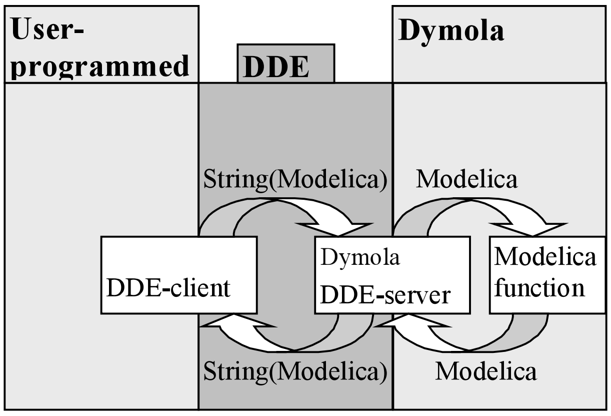


Figure 3 Using Dymola's interface to Modelica functions from other programs.

For this to be possible, all data-structures in Modelica must have an output format that when evaluated gives the data-structure back. This seems straightforward, but the problem is empty arrays, since Modelica as a strongly typed language does not allow {} for empty arrays.

For primitive types a work-around is to use the fill-operator. For an empty array of records this requires using the record-constructor, but in many cases the record constructor does not have defaults for all elements and thus cannot be called without specifying arguments. A future extension of Modelica would be to allow calls of the record constructor with no arguments in this specific case.

3.1.1 Grammar for Modelica subset

We have defined a subset of Modelica for representing any structured data values, that is primitive types, arrays, and records.

The advantage of this format is that the mapping is self-explanatory, complete for Modelica data-types, and to be able to parse the Modelica-format it is only necessary to implement a parser for a subset of expressions from the Modelica grammar:

```
expression:
  primary
| "-" primary

primary :
  UNSIGNED_NUMBER
| STRING
| false
| true
| component_reference  function_call
| "[" expr_list { ";" expr_list } "]"
| "{" expr_list "}"

component_reference :
  IDENT [ "." component_reference ]

expr_list :
  expression { "," expression }

function_call :
  "(" [ function_args ] ")"

function_args :
  expression [ "," function_args  ]
| named_args

named_args:
  named_argument [ "," named_args ]

named_argument: IDENT "=" expression
```

Some of the names in the grammar have been shortened to keep the grammar elements on one line.

The reason to keep component_reference and function_call is to use record constructors to build record data-structures (using named arguments). Function call without named arguments (the line in italics) is only needed for the above-mentioned use of fill to construct empty arrays.

3.2 Mapping for Matlab

Dymola can automatically map data-structures *to* Matlab data-structures. They are first encoded in a string, that is then automatically evaluated by the Matlab-interface to the corresponding data-structure.

Dymola/Modelica	Matlab
Real	double
Integer	double
String	string
Enumeration(planned)	string
Boolean	double
Array	matrix or cell array
Record member	struct member

Figure 4 Mapping to Matlab

Array results are returned as matrices, except array of records and array of strings that are returned as cell arrays.

This provides a complete interface from any data-structure (i.e. return-value) defined in Modelica to a corresponding data-structure in Matlab. This includes the Modelica class and component structure as will be defined later.

The interface for sending requests from Matlab to Dymola cannot provide a similar feature based on the data structures in *Matlab*. The reasons are that Matlab does not distinguish between scalars, vectors and matrices (i.e. ndims in Matlab is always >=2), and that Modelica lacks a counterpart to Matlab's struct, i.e. an untyped record constructor.

However, as will be discussed in a following section an API to the class structure is available and the caller routine in Matlab (dymolaCall) has been extended with code to perform this mapping based on the declaration of the *called* Modelica function. Thus the Matlab-programmer only has to call dymolaCall with the name of the Modelica function and arguments as Matlab data-structures (arrays and structs). Dymola and dymolaCall and internally handle this and the result is a Matlab data-structure.

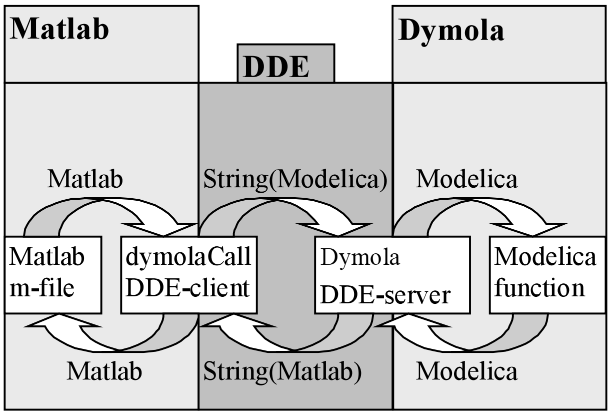


Figure 5 Interface between Matlab and Dymola

This does not include some of the advanced features of Modelica, e.g. the mapping does not automatically handle vectorized arguments to functions or allow you to use named arguments from Matlab.

We have not yet found any performance issues with this interface, but the m-file could be improved to locally cache the Modelica class structure in order to avoid sending the same query several times to Dymola (as will happen with e.g. arrays).

3.2.1 Examples

The following examples are only intended to demonstrate the possibilities and that strings, arrays of doubles, and records (containing strings and booleans) are returned (running from Matlab). The first examples demonstrate sending the entire command as one string:

```
>> dymolaCall('"Hello"+" world"')
ans =
Hello world

>> dymolaCall(…
'Modelica.Math.Matrices.inv([1,2;3,4])')
ans =
  -2.0000   1.0000
   1.5000  -0.5000
```

As indicated above the interface also allows call with function arguments as Matlab data-types (the second argument can optionally be used to specify an existing DDE-channel):

```
>> dymolaCall('Modelica.Math.Matrices.inv',[],..
[1,2;3,4])
ans =
  -2.0000   1.0000
   1.5000  -0.5000

dymolacall('GetClassAttributes',[], 'Modelica')
ans =
      fullName: 'Modelica'
     isPartial: 0
   isProtected: 0
    restricted: 'package'
      ...
```

```
>>dymolaCall(…
'Modelica.Math.Matrices.leastSquares',[],…
[1;0;1],[1;2;3])
ans =
    2
```

We will later return to how this uses the API to the Modelica class structure to construct the call.

3.3 Mapping for XML

For the abstract syntax tree one mapping to XML [4] is defined in [5]. A future mapping of data-structures to XML could use a subset of this by viewing them as a function call/expression in this structure (i.e. similar to the subset for the mapping to Modelica).

This can be viewed as too generic and another possibility is to automatically construct a specific document type declaration defining the grammar for the specific Modelica data-structure(s), i.e. one for each record class used, and placing this first in the XML-file [4]. This might still be combined with an external pre-defined data-type declaration for the built-in Modelica types, i.e. Boolean, Integer, Real, and String.

4 API to Modelica class structure

The first problem with defining an API to the Modelica class and component structure is that it is not possible to define a data-structure for the entire class structure in Modelica (at least not as implemented in Dymola), because the class structure is inherently recursive. However, even if it *were* possible to replicate the entire class structure as a set of nested records it would not provide an efficient interface to the class structure for simple queries or modifications.

Therefore we have instead defined access routines that allow tree walking to be built in Modelica (in the future there will also be corresponding routines for modifying and adding classes and components).

4.1 Basic design of API

In order to provide a useful interface to the classes and components three sets of routines were provided as follows in package Dymola.AST. Originally each set was only comprised of two functions and one record, and the intention is to further extend this (e.g. with routines for modifying the elements).

The three sets of routines are for classes, extends-clauses and components. In each set there is a routine for obtaining the elements (as an array), a record defining the "attributes" (protected, inner, full class name, …) and a routine for getting the attributes for a specific element.

These interfaces assume that one can use the name of elements in the queries, which is possible in the cases above (technically excluding the obscure case of repeated identical extends-statements which is legal Modelica, but without any reasonable use). Note that Dymola enforces this semantic restriction in Modelica already during parsing of classes, and thus it is safe to base the API-routines on this assumption.

The requirements also include access to the import-statements in the class. For import-clauses it is hard to define which name to use as a key (when considering both the qualified and the unqualified import-statements, thus a combined routine has been added that returns an array of records defining the import-statements.

This was found to provide such an increase in ease of use that similar routines were added for the other cases. These were trivial to implement based on the existing routines, and we give a full example below (excluding its documentation):

```
function ComponentsInClassAttributes
   "Get components of a class"
   input String className;
   output ComponentAttributes res[:]=
    GetComponentAttributes(className,
       ComponentsInClass(className));
algorithm
end ComponentsInClassAttributes;
```

Here the names of the components is constructed by ComponentsInClass and this is then used in a vectorized call (as defined in Modelica [3]) of GetComponentAttributes to get the attributes of all components.

Thus functions exists for all elements of table given on the next page (where "elements in class" has a class/ package as input and get attributes also exist in a form that returns an array containing the attributes of all elements).

	Record of attributes	Elements in class	Get attributes
Classes	ClassAttributes	ClassesInPackage	GetClassAttributes
Extends	ExtendsAttributes	ExtendsInClass	GetExtendsAttributes
Components	ComponentAttributes	ComponentsInClass	GetComponentAttributes
Import	ImportAttributes	ImportsInClassAttributes	

Figure 6 Overview of API to class structure. The row headings are the element types and the column headings the different functions (and records).

To make it possible to traverse all classes it is also possible to list all top-level classes (optionally limited to the ones defined in a specific Modelica file).

4.1.1 Example

These functions can be used in Modelica to find all restricted classes and provide e.g. the following list of accessible classes (excluding protected and partial ones):

	Modelica 1.6	Modelica 2.1
Model	222	429
Block	71	147
Function	41	199
Type	485	513
Package	50	130

Figure 7 Statistics for Modelica Standard Library

The growth of the standard library is in part due to the fact that ModelicaAdditions libraries were completed and after (in some cases major) revisions included in the Modelica Standard Library.

An alternative to returning all elements as one array of records would be to provide an iterator, or a callback-routine for enumerating the elements (and access routines instead of record elements). This is a traditional style in several environments (iterators in C++, enumeration callbacks in Windows API) since it avoids allocating large arrays. However, it requires additional state (in the iterator or enumerator call), which is contrary to the limitations on functions in Modelica, and therefore also increases the risk of errors in application code.

4.2 API to semantics not only to syntax

The API above defines basic routines that can be used directly. They also provide the basis for writing functions intended to answer higher-level questions, e.g. to search in a hiearchy for all components declared of a certain class.

Programming such queries require that the API answers questions related to the semantics of the declarations instead of questions based on their syntax (i.e. Dymola must not only parse the Modelica classes to answer the question, but also implement e.g. the semantics of look-up in Modelica).

To clarify this consider the declaration of T2 in the coupled clutches demo:

```
parameter SI.Time T2;
```

To obtain information about this declaration we can use the following:

```
Dymola.AST.GetComponentAttributes(
"Modelica.Mechanics.Rotational"+
".Examples.CoupledClutches","T2")
```

which gives the result:

```
Dymola.AST.ComponentAttributes(
  name = "T2",
  fullTypeName="Modelica.SIunits.Time",
  isProtected = false,
  sizes = {},
  variability = "parameter",
  isInput = false,
  isOutput = false,
  isInner = false,
  isOuter = false,
  isReplaceable = false,
  isRedeclared = false,
  isGraphical = false
)
```

By returning the full name of the type ("Modelica.SIunits.Time") and not the type-name part of the declaration ("SI.Time") it is straightforward to program this kind of queries and this also made it easier to program the calling interface from Matlab.

Obviously advanced users would like to also have access to the exact declaration (including modifiers and annotations) and that is planned for the future.

Basing the API on the semantics is also important for future improvement of providing routines to modify the classes through the API, where declaring a new component might require the addition of import-statements. When a user drags and drops a class to the diagram layer to declare a new component Dymola automatically adds import-statements if necessary. The API can internally re-use this functionality.

Similarly copying (or moving) a class from one Modelica package to another might require changes to its declarations which is done automatically by Dymola's GUI and hidden from the user.

4.3 Simplification in Modelica 2.2

One previous problem with using these functions was that the sizes of non-inputs to functions had to be known from the call according to the Modelica 2.1 standard [3]. That requires complex work-arounds and/or additional functions.

The restriction has now been lifted in Dymola (and accepted for Modelica 2.2) allowing a variable declared with size : (and without any binding assignment) to be re-sized (if necessary) when assigned in the function (note that this includes not only variables declared directly in the function but also their record elements). The change is backward compatible since such variables previously were semantically incorrect.

The change is not limited to working with the Modelica-structure, but is also useful in Modelica for unrelated uses, e.g. a routine that returns the positive eigenvalues. It was also needed to implement the API functions themselves, in particular the size-field for array of a component.

Those wanting an additional rationale can examine the case below where the same function as given in two versions, one written before the feature was implemented and another version rewritten to use it:

4.3.1 Example after simplification

As an example consider a function for finding the attribute of all classes defined in package (including the contents of packages – after the package):

```
function attributeModelsInPackage
  import Dymola.AST.*;
  input String s;
  output ClassAttributes attr[:];
protected
  String localClasses[:]=
    ClassesInPackage(s);
 ClassAttributes attributes;
algorithm
  for i in 1:size(localClasses,1) loop
    attributes:= GetClassAttributes(
       s + "." + localClasses[i]);
    attr:=cat(1,attr,{attributes});
    if attributes.restricted
        == "package" then
      attr := cat(1, attr,
        attributeModelsInPackage(
           attributes.fullName));
    end if;
  end for;
end attributeModelsInPackage;
```

4.3.2 Example before simplification

Before this feature of automatic resizing of arrays was available it was necessary to write two routines, one to determine the length of the array and one to actually return the array.

We consider this for the simpler case of only returning the full names of the classes, first we have to count the size of the output:

```
function countModelsInPackage
  import Dymola.AST.*;
  input String s;
  output Integer count= 0;
protected
 ClassAttributes attributes;
algorithm
  for i in ClassesInPackage(s) loop
    attributes:=GetClassAttributes(
        s + "." + i);
    count:=count+1;
    if attributes.restricted
         =="package" then
      count := count +
        countModelsInPackage(
           attributes.fullName);
    end if;
  end for;
end countModelsInPackage;
```

Note that there is no declared array for the result of ClassesInPackage – instead it is directly iterated over removing the need for any local variable (and the problem of its size) .

```
function attributesModelsInPackage
  import Dymola.AST.*;
  input String s;
  output String
    attr[countModelsInPackage(s)];
protected
 ClassAttributes attributes;
 Integer index=0;
 Integer len;
algorithm
  for i in ClassesInPackage(s) loop
    attributes:=
      GetClassAttributes(s + "." + i);
    index:=index+1;
    attr[index]:=attributes.fullName;
    if attributes.restricted
         =="package" then
      len :=countModelsInPackage(
        attributes.fullName);
      attr[index+1:index+len] :=
       attributesModelsInPackage(
         attributes.fullName);
      index:=index+len;
    end if;
  end for;
end attributesModelsInPackage;
```

Apart from practical problem of writing such complex functions an additional problem is that there is a need to maintain multiple functions. If requirements change (e.g. only return public classes) it is necessary to update two functions.

4.4 Revisited example from Matlab

When we previously considered the following call from Matlab

```
>>dymolaCall(…
'Modelica.Math.Matrices.leastSquares',[],…
[1;0;1],[1;2;3])
```

we indicated that Dymola's API was used to construct this argument. The calls of Dymola API functions are:

```
Dymola.AST.GetComponentAttributes(
"Modelica.Math.Matrices.leastSquares",
"A")
Dymola.AST.GetClassAttributes("Real")
Dymola.AST.GetClassAttributes("Real")
Dymola.AST.GetClassAttributes("Real")
Dymola.AST.GetComponentAttributes(
"Modelica.Math.Matrices.leastSquares",
"b")
Dymola.AST.GetClassAttributes("Real")
Dymola.AST.GetClassAttributes("Real")
Dymola.AST.GetClassAttributes("Real")
```

Finally the result is the following function call:

```
Modelica.Math.Matrices.leastSquares(
[1;0;1],{1,2,3})
```

The first call, GetClassAttributes, determines that this is a function call and not the call of a record constructor. The next call, ComponentsInClass, is used to determine the components of the function. For each argument the next input component is found by looking at the component attributes (this check is not performed for record constructors). The type of input component is then accessed, GetClassAttributes("Real"), to find that it is a primitive numeric type (since booleans must be treated specially).

The significant part is that in Matlab there are two matrices/columns vectors and based on the Modelica function the first one is sent as matrix to Dymola ([1;0;1]) and the second one as a vector ({1,2,3}). Without the API-calls it would not have been possible to determine that these should be treated differently.

5 Conclusions

This paper shows that Dymola 6's Modelica implementation provides an extendable external interface to use other tools and also be useful from other tools. In addition it shows that an interface to the Modelica class structure is useful in itself and also can be used when implementing the external interface.

References

[1] Dynasim (2005): **Dymola User's Manual**, Dynasim, www.dynasim.com,

[2] XML Protocol Working Group (2000-): **SOAP** http://www.w3.org/2000/xp/Group

[3] Modelica Association (2004): **Modelica Language Specification Version 2.1**, www.modelica.org

[4] World Wide Web Consortium (1996-): **Extensible Markup Language (XML)**, http://www.w3.org/XML/

[5] Pop A., P. Fritzson P. (2003): **ModelicaXML: A Modelica XML Representation with Applications**, Proceedings of the 3rd International Modelica Conference, Linköping Sweden.

[6] Andreasson J. (2003): **VehicleDynamics library**, Proceedings of the 3rd International Modelica Conference, Linköping Sweden.