



2002

**Proceedings of the
2nd International Modelica Conference**

March 18-19, 2002

Deutsches Zentrum für Luft- und Raumfahrt e.V.
Oberpfaffenhofen, Germany

Martin Otter (editor)

organized by
The Modelica Association and
Institut für Robotik und Mechatronik, Deutsches Zentrum für Luft- und Raumfahrt e.V.

All papers of this conference can be downloaded from
<http://www.Modelica.org/Conference2002/papers.shtml>

Proceedings of Modelica'2002

Deutsches Zentrum für Luft- und Raumfahrt e.V.,
Oberpfaffenhofen, Germany, March 2002

Editor:

Martin Otter

Published by:

The Modelica Association (<http://www.Modelica.org>) and
Institut für Robotik und Mechatronik, Deutsches Zentrum für Luft- und Raumfahrt e.V. (<http://www.robotic.dlr.de>)

Printed by:

Digital- & Offsetdruck Gerdfried Wolfertstetter, Carl-Benz-Str. 14, 82205 Gilching

Acknowledgement:

The organization of this conference was in parts supported by the European Commission under contract IST-199-11979 with DLR for the project entitled "Real-time Simulation for Design of Multi-physics Systems".

Preface

The Modelica modeling language has been designed to allow convenient and efficient modeling and simulation of complex, multi-domain physical systems described by differential, algebraic and discrete equations, aiming at full system simulation. Since January 2002, version 2.0 of the Modelica language definition is available together with many free Modelica libraries, as well as commercial Modelica simulation environments. Convenient interfaces exist for Matlab and Simulink (with Dymola) and to Mathematica (with MathModelica). The language, libraries and tools are used by a growing number of people in industry and academia for advanced applications, such as detailed fuel cell simulations, power systems, full vehicle dynamics models, hardware-in-the-loop simulations, embedded control systems with nonlinear Modelica models.

In October 2000, the first Modelica workshop took place in Lund, Sweden. Due to the great success, with more than 80 participants, this event is repeated in March 2002, to bring together people interested in Modelica, Modelica language designers, Modelica tool vendors and Modelica library developers. This gives the conference participants the opportunity to be informed about the latest developments, to influence the future development of Modelica and its libraries and to get in touch with people solving similar modeling problems.

This volume contains papers that are presented at the Second International Modelica Conference at DLR in Oberpfaffenhofen, Germany, March 18-19, 2002. A number of high quality papers were received. The program committee had a difficult task of planning the conference since not all submissions could be accommodated for the limited time of two days. Thirty-five papers were selected for presentations, and four papers were selected for poster presentations.

More information about the Modelica language, the Modelica association, this and future events can be found at the web page <http://www.modelica.org>. Especially, all papers from these proceedings are stored at this site after the conference.

The Modelica'2002 conference was arranged by the Modelica Association in cooperation with the Institut für Robotik und Mechatronik, Deutsches Zentrum für Luft- und Raumfahrt e.V., Oberpfaffenhofen, Germany.

Oberpfaffenhofen, March 5, 2002.

Martin Otter

Program Committee

- Martin Otter, Institut für Robotik und Mechatronik, Deutsches Zentrum für Luft- und Raumfahrt, Oberpfaffenhofen, Germany (chairman of the program committee).
- Hilding Elmqvist, Dynasim AB, Lund, Sweden.
- Peter Fritzson, PELAB, Programming Environment Laboratory, Department of Computer and Information Science, Linköping University, Linköping, Sweden.

Local Organizers

Martin Otter, Astrid Jaschinski, Christian Schweiger, Erika Woeller, Johann Bals,
Institut für Robotik und Mechatronik, Deutsches Zentrum für Luft- und Raumfahrt,
Oberpfaffenhofen, Germany.

Contents

Tutorials	5
M. Otter, H. Olsson: <i>New Features in Modelica 2.0</i>	7
Mattsson S.E., Elmqvist H., Otter M., and Olsson H.: <i>Initialization of Hybrid Differential-Algebraic Equations in Modelica 2.0</i>	9
Applications and Tools	17
Tiller M., Tobler W.E., Ming Kuang: <i>Evaluating Engine Contributions to HEV Driveline Vibrations</i>	19
Clauß C., Beater P.: <i>Multidomain Systems: Electronic, Hydraulic, and Mechanical Subsystems of an Universal Testing Machine modeled with Modelica</i>	25
Tummescheit H., Eborn J.: <i>Chemical Reaction Modeling with ThermoFluid/MF and MultiFlash</i>	31
Fritzson P., Gunnarsson J., Jirstrand M.: <i>MathModelica - An Extensible Modeling and Simulation Environment with Integrated Graphics and Literate Programming</i>	41
Brück D., Elmqvist H., Mattsson S.E., Olsson H., : <i>Dymola for Multi-Engineering Modeling and Simulation</i>	55
Automotive Powertrains and Hardware-in-the-Loop Simulation	57
Elmqvist H., Mattsson S.E., Olsson H.: <i>New Methods for Hardware-in-the-Loop Simulation of Stiff Models.</i>	59
Soejima S., Matsuba T.: <i>Application of mixed mode integration and new implicit inline integration at Toyota</i>	65
Schlegel C., Bross M., Beater P.: <i>HIL-Simulation of the Hydraulics and Mechanics of an Automatic Gearbox</i>	67
Puchalsky C., Megli T., Tiller M., Trask N. and Wang Y., Curtis E.: <i>Modelica Applications for Camless Engine Valvetrain Development</i>	77
Electrical Systems	87
Hongesombut K., Mitani Y., Tsuji K.: <i>An Incorporated Use of Genetic Algorithm and a Modelica Library for Simultaneous Tuning of Power System Stabilizers</i>	89
Urquía A., Dormido S.: <i>DC, AC Small-Signal and Transient Analysis of Level 1 N-Channel MOSFET with Modelica</i>	99
Ferretti G., Magnani G., Rocco P., Bonometti L., Maraglino M.: <i>Simulating permanent magnet brushless motors in DYMOLA</i>	109
Kalaschnikow S.N.: <i>PQLib - A Modelica Library for Power Quality analysis in Networks</i>	117
Automotive Powertrains	123
Treffinger P., Goedecke M.: <i>Development of Fuel Cell Powered Drive Trains With Modelica</i>	125
Newman C.E., Batteh J.J., Tiller M.: <i>Spark-Ignited-Engine Cycle Simulation in Modelica</i>	133
Thermodynamic Systems I	143
Pfafferott T., Schmitz G.: <i>Modeling and Simulation of Refrigeration Systems with the Natural Refrigerant Carbon Dioxid</i>	145
Felgner F., Agustina S., Cladera Bohigas R., Merz R., Litz L.: <i>Simulation of Thermal Building Behaviour in Modelica</i>	147

Poster session	155
Bunus P., Fritzson P.: <i>Methods for Structural Analysis and Debugging of Modelica Models</i>	157
Torrey D.A., Selamogullari U.S.: <i>A Behavioral Model for DC-DC Converter using Modelica</i>	167
Torrey D.A., Selamogullari U.S.: <i>Modelica Implementation of Field-oriented Controlled 3-phase Induction Machine Drive</i>	173
Wischhusen S., Schmitz G.: <i>Numerical Simulation of Complex Cooling and Heating Systems</i>	183
Discrete Event Modeling	193
Freiseisen W., Keber R., Medetz W., Pau P., Stelzmueller D.: <i>Using Modelica for Testing Embedded Systems</i>	195
Remelhe M.A.P.: <i>Combining Discrete Event Models and Modelica - General Thoughts and a Special Modeling Environment</i>	203
Färnqvist D., Strandemar K., Johansson K.H., Hespanha J.P.: <i>Hybrid Modeling of Communication Networks Using Modelica</i>	209
Thermodynamic Systems II	215
Steinmann W.D., Zunft S.: <i>TechThermo - A Library for Modelica Applications in Technical Thermodynamics</i>	217
Fabricius S.M.O., Badreddin E.: <i>Modelica Library for Hybrid Simulation of Mass Flow in Process Plants ..</i>	225
Jensen J.M., Tummescheit H.: <i>Moving Boundary Models for Dynamic Simulations of Two-Phase Flows ..</i>	235
Mechatronic Applications	245
Hellgren J.: <i>Modelling of Hybrid Electric Vehicles in Modelica for Virtual Prototyping</i>	247
Pelchen C., Schweiger C., Otter M.: <i>Modeling and Simulating the Efficiency of Gearboxes and of Planetary Gearboxes</i>	257
Andreasson J., Jarlmark J.: <i>Modularised Tyre Modelling in Modelica</i>	267
Moormann D., Looye G.: <i>The Modelica Flight Dynamics Library</i>	275
Aberger M., Otter M.: <i>Modeling Friction in Modelica with the Lund-Grenoble Friction Model</i>	285
Special Methods and Tools	295
Fritzson P., Aronsson P., Bunus P., Engelson V., Saldamli L., Johansson H., Karström A.: <i>The Open Source Modelica Project</i>	297
Saldamli L., Fritzson P., Bachmann B.: <i>Extending Modelica for Partial Differential Equations</i>	307
Franke R.: <i>Formulation of dynamic optimization problems using Modelica</i>	315
Sjöberg J., Fyhr F., Grönstedt T.: <i>Estimating parameters in physical models using MathModelica</i>	325
Aronsson P., Fritzson P.: <i>Multiprocessor Scheduling of Simulation Code From Modelica Models</i>	331

Session 1

Tutorials

New Features in Modelica 2.0

Martin Otter¹ and Hans Olsson²

¹DLR, Oberpfaffenhofen, Germany, Martin.Otter@dlr.de

²Dynasim AB, Lund, Sweden, Hans.Olsson@dynasim.se

Abstract

The second major release of Modelica was finished and formally approved at the last Modelica design meeting, January 2002, Lund, Sweden. In this paper, the new features of Modelica 2.0 are described.

1. Introduction

The freely available, object-oriented modeling language Modelica is developed continuously since 1996. Modelica is designed to allow effective, component-oriented modeling of complex engineering systems described by differential, algebraic and discrete equations, e.g., systems containing mechanical, electrical, electronic, hydraulic, thermal, control, electric power or process-oriented subcomponents. A large number of free and commercial libraries of fundamental models are available as well as commercial Modelica simulation environments. More information is provided at <http://www.Modelica.org/>.

In 1997, the first major version of Modelica was released, followed by four minor revisions released once a year. The second major release of Modelica was completed and formally approved at the last Modelica design meeting, January 2002, Lund, Sweden. The most important design goal was to enhance the development and use of application libraries, incorporating the experience and feedback of library developers, while keeping backward compatibility. A number of language enhancements have been added, significantly facilitating library development and use. In this paper, the new features of Modelica 2.0 are described. The following members of the Modelica Association have contributed to the development of Modelica 2:

P. Aronsson, Linköping University, Sweden.
B. Bachmann, University of Bielefeld, Germany.
P. Beater, University of Paderborn, Germany
D. Brück, Dynasim, Lund, Sweden
P. Bunus, Linköping University, Sweden
H. Elmqvist, Dynasim, Lund, Sweden
V. Engelson, Linköping University, Sweden
P. Fritzson, Linköping University, Sweden
R. Franke, ABB Corporate Research, Ladenburg

P. Grozman, Equa, Stockholm, Sweden
J. Gunnarsson, MathCore, Linköping
M. Jirstrand, MathCore, Linköping
S. E. Mattsson, Dynasim, Lund, Sweden
H. Olsson, Dynasim, Lund, Sweden
M. Otter, DLR, Oberpfaffenhofen, Germany
L. Saldamli, Linköping University, Sweden
M. Tiller, Ford Motor Company, Dearborn, MI, U.S.A.
H. Tummescheit, Lund Institute of Technology, Sweden
H.-J. Wiesmann, ABB Corp. Res., Baden, Switzerland

2. Component Arrays

One part of the redesign of Modelica 2 was based on the experience with the Modelica.Blocks library in Modelica 1. The redesign supports *generic formulation* of blocks applicable to both scalar and vector connectors, *connection* of (automatically) *vectorized blocks*, and *simpler input/output connectors*. This allows significant simplifications of the input/output block library of Modelica, e.g., since only scalar versions of blocks that naturally vectorize have to be provided. Furthermore, new library components can be incorporated more easily. In addition, it is possible to use functions and functional blocks allowing, e.g., the *sin-function* to be inserted in a block diagram.

Since the first release, it was possible in Modelica to define homogenous component arrays, i.e., arrays where the array elements are instances of any desired class, for example:

```
Resistor R[10];
```

is an array of 10 resistors including both the resistor parameters and the resistor equations. In Modelica 2, features have been added for component arrays to widen their applicability.

Component Array Modifications

Assume a component is defined as

```
model FixedFrame
  parameter Real r[3] = {0,0,0};
  parameter Real alpha = 0;
  parameter Real beta = 0;
  parameter Real gamma = 0;
  ...
endmodel
```

```
end FixedFrame;
```

which describes a coordinate system with respect to another one as fixed translation with vector **r** and fixed rotation around angles α, β, γ along x-, y-, and z-axis respectively. A part may have several frames and also other properties and can be defined as

```
model Part
  parameter Integer n=0;
  FixedFrame frames[n];
  ...
end Part;
```

There are different possibilities to define a part which has several frames:

```
Part p(n=2, frames[1] (r={1,0,0},
                      alpha = 1),
      frames[2].alpha = -1);
```

Here, every element of the frame vector is explicitly modified. Another alternative is

```
Part p(n=4, frames(beta={1,2,3,4},
                  r = fill(1,4,3)));
```

where the *same* parameter of all frames are modified. For example, frames[:].beta is treated as a vector of 4 elements and therefore a vector of 4 elements has to be provided as modification. On the other hand, frames[:].r is treated as a (4,3) matrix.

```
Part p(n=10, frames(each r={1,0,0}));
```

defines 10 frames, using the same vector **r** for all frames. In a similar way also nested component arrays are handled:

```
Part p[10] (each n=3,
            each frames(each beta=1));
```

Here, 10 parts are declared, where every part has 3 identical frames with beta=1. In this application it is not very useful to define so many identical frames. However, in lumped models deduced from the discretisation of partial differential equations, often many elements of a component array have the same value. Example:

```
parameter Integer n;
parameter Real L=1 "length";
parameter Real r=1 "resistance
                    per meter";
protected
```

```
parameter Real Re=r*L/n;
Resistor R[n+1] (R =
  vector ([Re/2;
          fill (Re,n-1);
          Re/2]));
```

All elements of the resistance vector **R** are the same, with the exception of the first and last one which each take half of the value of an element Resistance **Re**.

Block Vectorization

Connectors of signals of the Modelica.Blocks library are in Modelica 1.4 defined as

```
connector InPort
  parameter Integer n=1;
  input Real signal[n];
end InPort;
```

That is, the connector consists of a vector of Reals which are used as input signals. Such a connector is utilized in an input/output block as:

```
block FirstOrder
  parameter Real T=1 "time const.";
  InPort inPort;
  OutPort outPort;
  ...
end FirstOrder;
```

Accessing the input signal of such a block is inconvenient:

```
FirstOrder b;
...
b.inPort.signal[1] // input signal
```

In Modelica 2 it is possible to define a connector as an extension from the base types, i.e., the following definition is possible:

```
connector InPortNew = input Real;
```

Also annotations for the graphical layout of icon and diagram layer of such a connector can be defined. Therefore, this connector may be dragged from a library window in a model window to construct a new model graphically. In a model, this connector is used as:

```
block FirstOrderNew
  parameter Real T=1 "time const.";
  InPortNew u;
  OutPortNew y;
  ...
end FirstOrderNew;
```

Accessing the input signal of this block is now much simpler:

```
FirstOrderNew b;
...
b.u // input signal
```

In the 1.4 version of the Modelica.Blocks library, most blocks are manually vectorized, e.g., to define an instance which has 10 input and 10 output signals and 10 first order blocks for every signal path. This complicates the class definitions in Modelica.Blocks considerably, and in all cases, except Sources.KinemanticPTP, a vectorized block behaves as a vector of scalar blocks. With the extensions described above, this is much simpler. For example, a scalar Sine block may be defined as:

```
block Sine
  import Modelica.Math.*;
  import Modelica.Constants.*;
  parameter Real Amplitude = 1;
  parameter Real frequency = 1;
  parameter Real phase = 0;
  InPortNew u;
  OutPortNew y;
equation
  y = Amplitude*
    sin(2*pi*frequency*time+phase);
end FirstOrderNew;
```

This looks like a text-book example of a sine source. Using 3 Sine sources is now performed by component arrays:

```
Sine s[3] (each frequency=50,
           phase = {0,2,-2});
```

Note, that it is easy to define that all sine-sources shall have the same frequency, but different phases roughly corresponding to 3 electrical phases. A state space model may be defined as:

```
block StateSpace
  final parameter Integer nx =
    size(A,1);
  final parameter Integer nu =
    size(B,2);
  final parameter Integer ny =
    size(C,1);
  parameter Real A[:,nx];
  parameter Real B[nx,:];
  parameter Real C[:,nx];
  parameter Real D[ny,nu];
  InPortNew u[nu];
  OutPortNew y[ny];
```

```
Real      x[nx]
equation
  der(x) = A*x + B*u;
  y = C*x + D*u;
end StateSpace;
```

Connecting the 3 sin-sources as input to an instance of StateSpace which has three inputs can be performed in the following way:

```
Sine s[3] (each frequency=50,
           phase = {0,2,-2});
StateSpace b(B=[0,0,1;...],...);
equation
  connect(s.y, b.u);
```

This is a connection of `s[:,y]` with `b.u[:,]`. This is allowed due to a new connection rule, provided the dimension sizes match, which is the case here.

3. Enumeration Types

Modelica 2 introduces enumerations to construct new base types which consist of countable sets of elements. Example:

```
type TextStyle = enumeration(
  Bold, Italic, UnderLine);
```

This declaration defines a new type TextStyle. An instance of this type may have only the values TextStyle.Bold, TextStyle.Italic or TextStyle.UnderLine. Such a type can be used in the following way:

```
TextStyle t1 = TextStyle.Bold;
TextStyle t2 = t1;
```

Currently, the only operations defined for enumeration types are the equal ("=") and the assignment (":=") operations. Furthermore, the relational operators <, <=, >, >=, ==, <> can be applied. The result depends on the order of the element in the enumeration declaration. For example TextStyle.Bold < TextStyle.Italic. It is planned to provide more operations in future Modelica releases, e.g., to access array indices by enumerations or inquire the next or previous enumeration element.

Enumerations are useful for defining properties and options in an understandable and safe way. Since enumerations are internally mapped to an Integer type, processing them is safer and much more efficient than if properties or options would be defined as Strings. Compared to using Integer constants it is clearer, requires less typing, and is

safer since each enumeration is a separate type. In Modelica 2, several enumeration types are predefined, such as `StateSelect` (see next section) and enumerations in graphical annotations.

4. State Selection Control

The continuous part of a Modelica model is mapped to a DAE, a differential-algebraic equation system, of the form

$$0 = f(dx/dt, x, y, t)$$

where $x(t)$ are variables appearing differentiated and $y(t)$ are pure algebraic variables. Conceptually, this DAE is transformed in to state space form

$$\begin{aligned} dx_s/dt &= f_1(x_s, t) \\ x_n &= f_2(x_s, t) \\ y &= f_3(x_s, t) \end{aligned}$$

where $x_s(t)$ are a subset of x which are independent from each other and $x_n(t)$ are the other variables of x . Variables $x_s(t)$ are called **states** of the model. A numerical integration method essentially discretizes x_s over time, whereas all other variables are calculated as the solution of an algebraic system of equations at the actual time instant. The selection of x_s is not unique. Different choices may lead to drastically different numerical behaviour. A dynamic automatic selection of x_s by a tool is always possible, [4]. However, experience shows that user insight may lead to better choices or avoid the need for dynamic selection. On the other hand automatic selection is an efficient and reliable method, and users should not be forced to manually perform a complete manual state selection merely because dynamic state selection might be inefficient for some models. For this reason, in Modelica 2 it is possible to guide the state selection via the new attribute **stateSelect** of Real variables. The attribute has values from the enumeration **StateSelect** defined as:

```
type StateSelect = enumeration(
    never, avoid, default,
    prefer, always);
```

For "**never**", a variable will never be selected as a state, whereas for "**always**" the variable shall always be used as a state. For "**default**", which is the default for all Real variables, the states are automatically selected among the variables which appear differentiated. If "**prefer**" is used, the variable need not to be differentiated and is

preferably used as state over those having the default value. Finally, for "**avoid**", the variable is only selected as a state, if it appears differentiated and if no other selection of variables with "default", "prefer", or "always" value is possible. A state preference definition may be given in the following way:

```
Real w(stateSelect =
    StateSelect.prefer);
```

Examples for appropriate state selection (from [2]):

Accuracy:

In rotating machinery systems used for power transmission (but not for positioning drive systems) and in power systems, angular positions of shafts are increasing with time, but relative positions between shafts are rather constant, at least in normal operation. Say that two rotating inertias are connected by a spring such that the relative distance between them are 0.1 rad and that their angular speed is 1000 rad/s. If the positions are calculated with a relative accuracy of 0.001, after 10 seconds there is hardly any accuracy in calculating the distance by taking the difference. The difference behaves irregularly and gives an irregular torque if simulations take too long. It is very difficult for a tool to find this out without actually doing simulation runs. Therefore, it is useful to define **StateSelect.prefer** for all relative variables in force elements (e.g., spring, damper, clutch). This will be performed in the next version of the Modelica.Mechanics.Rotational library.

Avoiding function inversion:

In thermodynamic problems property functions are utilized. These functions usually assume two variables to be inputs (for example pressure and enthalpy) and calculate other properties (such as temperature, density). Thus, if such variables are selected as state variables it is "simply" calling property functions to calculate other needed variables. Otherwise, it is necessary to solve non-linear equation systems to calculate the input variables of the property functions. Therefore, a good choice is to use **StateSelect.prefer** on all input variables of property functions, or use **StateSelect.avoid** on output variables from property functions.

Less nonlinear equations:

For three-phase power systems several choices of states are possible, especially selecting states from the stator side or from the rotor side. The first

choice leads to a **non-linear** DAE, whereas the second one leads (under certain assumptions) to a **linear** DAE. In a periodic steady state, the first choice results in a **periodic** solution of the states whereas in the second choice the states are identical to **zero**. As a result, selecting states from the rotor side (= Park transformation) leads to a more efficient and more reliable numerical solution and therefore these variables should have the attribute value **StateSelect.prefer**.

Avoid dynamic state selection:

For 3-dimensional mechanical systems having closed kinematic loops, an automatic static selection of states is not possible. Instead, the states have to be dynamically selected and changed during simulation in order to keep the (time-varying) Jacobian of the system non-singular. In many cases a suitable set of state variables is known, e.g., the relative position and velocity variables of the joints driving the mechanism. If these variables have attribute value **StateSelect.always** the simulation is more efficient which is especially important for real-time simulations.

Sensors:

A sensor may measure the speed "v" of a translational connector. Since the speed is not part of the connector, but the position "s" is, an equation of the form "**der**(s) = v" is present in the sensor, i.e., "s" appears differentiated and can be potentially used as a state. However, in most case the selection of "s" as a state is not appropriate, since introduction of a variable for just plotting should not influence the state selection. Therefore, an attribute value of **StateSelect.avoid** should be preferably used for differentiated variables in sensor objects (here: "s").

The general advice is that selection of states ought to be done automatically. This is also possible and unproblematic in most models. Only if there are good reasons, as pointed out above at hand of several examples, the modeler may give hints for state selection. Note, that in a library the values **StateSelect.never** or **StateSelect.always** should not be used, because a library has usually not enough information to rigidly force a state selection.

5. Improved Initialization

Modelica 2.0 introduces a mathematically rigid specification of the initialization of Modelica

models, i.e., of hybrid differential algebraic equations. The new language constructs permit flexible specification of initial conditions as well as the correct solution of difficult, non-standard initialization problems occurring in industrial applications, for example:

- Stationary initialization around a constant reference velocity of an aircraft.
- Stationary initialization around periodic solutions, needed in power systems or in detailed engine models.
- Stationary initialization of continuous systems controlled by sampled data systems (the states of the discrete controllers are computed in such a way that the overall system is in a steady state when simulation starts).
- Initialization of discontinuous or variable structure systems, e.g., systems containing friction or backlash.

Since this is a large topic by itself, only a short overview is given here. Details are presented in the companion paper [3].

Before any operation, in particular simulation, is carried out with a Modelica model, initialization takes place to assign **consistent** values for all variables present in the model, including derivatives, **der**(...), and **pre**-variables, **pre**(...). The initialization uses all equations and algorithms that are utilized during the simulation.

In the most simplest case, when only continuous equations are present without algebraic dependencies of states (= no higher DAE index), a Modelica model is mapped to the following differential-algebraic equation system (DAE):

$$\mathbf{0} = \mathbf{f}(\mathbf{dx}/dt, \mathbf{x}, \mathbf{y}, t)$$

where $\mathbf{x}(t)$ are variables appearing differentiated, $\mathbf{y}(t)$ are algebraic variables and $\dim(\mathbf{f}) = \dim(\mathbf{x}) + \dim(\mathbf{y})$. These equations have to be fulfilled at all time instants, especially also at the initial time t_0 . During simulation, an integrator calls the model providing basically \mathbf{x} as input. Therefore, the model equations are solved under the assumption that \mathbf{x} is known. During initialization, \mathbf{x} is, however, unknown. As a result, there are only $\dim(\mathbf{x}) + \dim(\mathbf{y})$ equations for $2 \cdot \dim(\mathbf{x}) + \dim(\mathbf{y})$ unknowns during initialization. In the most general case this means that the modeler has to provide additionally $\dim(\mathbf{x})$ equations $\mathbf{g}(\cdot)$ at the initial time resulting in the following system of equations

$$\mathbf{0} = \begin{bmatrix} \mathbf{f}(\dot{\mathbf{x}}(t_0), \mathbf{x}(t_0), \mathbf{y}(t_0), t_0) \\ \mathbf{g}(\dot{\mathbf{x}}(t_0), \mathbf{x}(t_0), \mathbf{y}(t_0), t_0) \end{bmatrix}$$

which has to be solved for the unknowns $\mathbf{dx}/dt(t_0)$, $\mathbf{x}(t_0)$, $\mathbf{y}(t_0)$. In general this means that the standard algorithms, such as BLT transformation, should be applied to this system, in order to compute the solution reliably and efficiently. From a user's point of view this procedure means that $\dim(\mathbf{x})$ equations have to be additionally provided for the initial time, e.g., $\mathbf{x}(t_0) = \mathbf{x}_0$ or $\mathbf{dx}/dt(t_0) = \mathbf{0}$. In Modelica 2 these initial equations can either be defined in the new **initial equation** / **initial algorithm** sections or as start value with attribute `fixed = true`. For example two initial equations $x1(t_0) = 1$ and $dx2/dt(t_0) = 0$ may be defined as:

```
Real x1(start=1, fixed=true);
Real x2 //default: fixed=false
initial equation
  der(x2) = 0;
equation
  der(x1) = -x1 + x2;
  der(x2) = -x2;
```

If there are constraints between states, the number of initial equations to be provided is less than $\dim(\mathbf{x})$. It may be difficult for a user of a large model to figure out how many initial equations have to be added. Therefore, it is essential that a Modelica environment has appropriate support. For example, Dymola performs index reduction and selects state variables for the simulation model [1], [3], [4]. Thus, it establishes how many states there are and how many initial conditions have to be additionally provided. If there are too many initial equations, Dymola outputs an error message indicating a set of initial equations or fixed start values from which initial equations must be removed or start values inactivated by setting `fixed = false`. If initial conditions are missing, Dymola makes automatic default selection of initial conditions. The approach is to select continuous time states with inactive start values and make their start values active by turning their `fixed` attribute to `true` to get a structurally well posed initialization problem. A message informing about the result of such a selection can be obtained.

6. Function Applications

In Modelica 1.4, a function application can have *either* positional *or* named *input* arguments. In Modelica 2, a function application may have optional *positional input arguments* followed by

zero, one or more *named input arguments*. Arguments not explicitly present get the default value supplied in the function declaration. This feature is useful to make the same function fit for beginners and expert users. For example, a function **RealToString** may be defined as follows to convert a Real number into a String representation:

```
function RealToString
  input Real number;
  input Real precision = 6;
  input Real minLength = 0;
  output String string;
algorithm
  ...
end RealToString;
```

Argument "number" is the number to be converted, "precision" is the number of significant digits in the String representation and "minLength" is the minimum length of the String in which the number is stored right justified. Since positional, named and default arguments are allowed, the following function applications are equivalent:

```
RealToString(2.0);
RealToString(2.0, 6, 0);
RealToString(2.0, 6);
RealToString(2.0, precision=6);
RealToString(2.0, minLength=0);
```

Note, that the following call leads to an error

```
RealToString(2.0, 6, precision=4);
```

since argument 2 is defined twice. This function may be used to conveniently build up a message string, such as

```
Variable "mass" (= -10.4562) shall
be non-negative.
```

via the function call

```
assert(v>=0,"Variable \"mass\" (= "
+ RealToString(v) + " shall be "
+ "non-negative.\n")
```

As before, only positional output arguments of a function application are possible. However, output arguments shall be omitted, if the corresponding variables has attribute `enable=false` in the function declaration. This makes it possible to avoid dummy output arguments in the function application which are not used in the calling function. For example, a function to compute eigenvalues and optionally right and left eigenvectors may be defined in Modelica as:

```

function eigen
  parameter Integer n = size(A,1);
  input Real A[:,n];
  input Boolean getREV = false;
  input Boolean getLEV = false;
  output Real eigenValues[n,2];
  output Real REV[n,n] (enable=getREV);
  output Real LEV[n,n] (enable=getLEV);
algorithm
  // compute eigenvalues
  if getREV then
    // compute right eigenvectors
  end if;
  if getLEV then
    // compute left eigenvectors
  end if;
end eigen;

```

This function may be called to calculate only the eigenvalues of a matrix or to just determine whether a matrix has only stable eigenvalues:

```

ev = eigen(A);
b = isStable(eigen(A)); //

```

to calculate eigenvalues and right eigenvectors:

```

(ev, REV) = eigen(A, getREV=true);

```

to calculate additionally also the left eigenvectors:

```

(ev, REV, LEV) = eigen(A, getREV=true,
                        getLEV=true);

```

7. Record Constructor

In Modelica 2, the missing constructor for the record data type is introduced. It is defined as a function with the same name and the same scope as the corresponding record containing all modifiable components of the record as input arguments and a record instance as output argument. Since a record constructor is just a function, it can be used at all places, where a function call is allowed. For example, with the following record declaration

```

record Complex "Complex number"
  Real re "real part";
  Real im "imaginary part";
end Complex;

```

a Complex data type is defined and implicitly its constructor function

```

function Complex
  input Real re "real part";
  input Real im "imaginary part";
  output Complex out (re=re, im=im);
end Complex;

```

Additionally, functions are needed, operating on this data type, such as:

```

function add "Add Comp. numbers"
  input Complex u, v;
  output Complex w (re=u.re + v.re,
                    im=u.im + v.im);
end add;

```

The record constructor allows, e.g., to avoid the usage of unnecessary auxiliary variables:

```

Complex c1, c2;
equation
  c2 = add(c1, Complex(sin(time),
                       cos(time)));

```

Note, that the second argument of the function application uses the record constructor to construct a temporary instance of type Complex.

Record constructors are very useful in situations where previously replaceable records have been needed (which are much less convenient to utilize). For example, a data sheet library of motors shall be constructed. The motor model consists essentially of two parts, one part containing all the data defining a particular motor as a record, e.g.,

```

record MotorData
  parameter Real inertia;
  parameter Real nominalTorque;
  parameter Real maxTorque;
  parameter Real maxSpeed;
  ...
end MotorData;

```

and the motor model utilizing the motor data

```

model Motor
  MotorData data;
  // connector definitions
equation
  ...
end Motor;

```

When using a motor, specific values of the motor data could be given in the usual way:

```

model Robot1
  Motor m1 (data (inertia      = 0.001,
                  nominalTorque = 10,
                  maxTorque     = 20,
                  maxSpeed      = 3600));
  ...
end Robot1;

```

When using the same motor type several times, it is better to define the motor data just ones, i.e., build up a **data sheet library** by **modifications** of the default values of the basic MotorData record:

```
package Motors
  record M103 = MotorData(
    inertia      = 0.001,
    nominalTorque = 10,
    maxTorque    = 20,
    maxSpeed     = 3600);

  record M104 = MotorData(
    inertia      = 0.0015,
    nominalTorque = 15,
    maxTorque    = 22,
    maxSpeed     = 3600);

  ...
end Motors;
```

Whenever one of the motors of package Motors is needed, it can be accessed by using the corresponding **record constructor**:

```
model Robot2
  Motor m1(data = Motors.M103());
  Motor m2(data = Motors.M104(
    inertia=0.0012));
  ...
end Robot2;
```

It is still possible to override parameters in such a definition, see declaration of m2, by calling the record constructor function with appropriate positional or (preferably) named arguments.

8. Iterators

Modelica 2 introduces several enhancements to support more powerful expressions, especially in declarations, in order to avoid inconvenient local function definitions:

Deduction of Ranges

In all iterators, e.g., in for-loop, the expression to define the range of the iteration need not to be given if the *iterator variables appear as array indices*. In such cases the iteration range is deduced from the dimension sizes of the corresponding arrays. Example:

```
for i loop
  A[i] = B[i]^2;
end for;
```

A nested for loop

```
for i in 1:size(A,1) loop
  for j in 1:size(A,2) loop
    A[i,j] = B[i,j]^2;
```

```
    end for;
  end for;
```

may be abbreviated as

```
for i in 1:size(A,1),
  j in 1:size(A,2) loop
  A[i,j] = B[i,j]^2;
end for;
```

or even shorter by automatic deduction of ranges

```
for i,j loop
  A[i,j] = B[i,j]^2;
end for;
```

Reduction Operators

An expression

```
function(expression for iterators);
```

is a **reduction-expression**. Currently, only the function names **sum**, **product**, **min**, and **max** can be used. The result is constructed by evaluating "expression" for each value of the iterator variable and computing the sum, product, minimum, or maximum of the computed elements. Examples:

```
sum(i for i in 1:10);
```

is the same as

$$\sum_{i=1}^{10} i = 1+2+\dots+10=55$$

A Modelica translator may transform this operation into:

```
algorithm
  result := 0;
  for i in 1:10 loop
    result := result + i;
  end for;
```

The sum of elements could also be defined as

```
sum(1:10);
```

using the built-in operator **sum()**. However, when summing up complex expressions or non-scalar expressions the reduction-expression can be made more readable than finding the appropriate vectorized expressions. As an example consider summing the squares instead:

```
sum(i^2 for i in 1:10);
```

The sum of squared elements could also be defined as

```
sum(diagonal(1:10)^2);
```

but even though it is slightly shorter it is not as readable.

Other examples are:

```
product(a[i,1]*s + a[i,2] for i);
```

is the same as

$$\prod_{i=1}^n (a_{i1}s + a_{i2}) = (a_{11}s + a_{12}) \cdot (a_{21}s + a_{22}) \cdot \dots$$

As usual, a vector of values may be given as an iterator:

```
sum(i^2 for i in {1,3,7,6})
```

Gives $\sum_{i \in \{1, 3, 7, 6\}} i^2 = 1+9+49+36=95$

```
max(i^2 for i in {3,7,6})
```

results in 49

Iterator Array Construction

In a similar way as a reduction operator, the construction

```
{expression for iterators};
```

with n iterators generates an array with n dimensions. The array is constructed by evaluating the expression for every iterator value and collecting the results to a corresponding array. Examples:

```
{i^2 for i in 1:5}
```

results in the vector

```
{1, 4, 9, 16, 25}
```

An (n,m) array having the same value v for all elements may be constructed as

```
{v for i in 1:n, j in 1:m}
```

which is the same as "fill(v,n,m)". The special matrix

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 4 \end{bmatrix}$$

may be created with

```
{if i==j then i else 0  
for i in 1:n, j in 1:n}
```

9. External Utility Functions

Modelica 1.4 has already a convenient and simple to use interface for external C and FORTRAN procedures which allows to pass nearly all data types of Modelica. The only exceptions have been String types which could not be returned. In Modelica 2, the following utility functions can be called in external C functions:

```
void ModelicaMessage  
    (const char* string)  
void ModelicaFormatMessage  
    (const char* string, ...)  
void ModelicaError  
    (const char* string)  
void ModelicaFormatError  
    (const char* string, ...)  
char* ModelicaAllocateString  
    (size_t len)  
char* ModelicaAllocateStringWithErrorReturn  
    (size_t len)
```

ModelicaMessage and **ModelicaFormatMessage** output a string to the message window of the Modelica environment. The latter with the same format control as the C-function printf. In both cases linefeeds need to be explicitly defined in the string by "\n". Similarly, **ModelicaError** and **ModelicaFormatError** output an error to the error window of the Modelica environment. Contrary to the first two functions, these functions never return to the calling function, but handle the error similarly to an assert in the Modelica code. Example for usage:

```
ModelicaFormatError(  
    "\"%s\" cannot be copied to \"%s\""  
    ":\n%s", oldFile, newFile,  
    strerror(errno));
```

Here, an error message is printed if a file cannot be copied. The error message of the operating system containing the source of the error is included at the end of the message by a call to the C function strerror(...).

ModelicaAllocateString allocates memory for a Modelica string which is used as return argument of an external Modelica function. If an error occurs, this function does not return. The Modelica environment is responsible to free this memory when appropriate. In a similar way **ModelicaAllocateStringWithErrorReturn** allocates string memory, but returns in case of error. This allows the external function to close files and free other open resources in case of error. Due to these two functions, Modelica Strings can now also be returned from external Modelica functions. For example, with the following external Modelica interface

```
function blanks
  input Integer n(min=0);
  output String blankString;
  external "C"
    blankString = blanks(n);
end blanks;
```

a string containing n blanks shall be returned. An implementation of this function in C could be accomplished in the following way:

```
#include "ModelicaUtilities.h"

const char* blanks(int n) {
  /* Create string with n blanks */
  char *c = ModelicaAllocateString(n);
  int i;
  for(i=0; i<n; ++i)
    c[i]=' ';
  c[n]='\0';
  return c;
};
```

Note, that it is not necessary to check in the C-function that the input argument "n" is not negative, because this is already defined in the Modelica interface and therefore the Modelica environment is responsible to check this property. Furthermore, it needs not to be checked whether memory could be allocated, because **ModelicaAllocateString** will not return in such a case but will raise an exception in the Modelica run-time environment and will jump to a place where execution can continue, e.g., after terminating the simulation.

Note that the newly introduced enumeration types can also be used as input and output arguments in external functions. They are mapped to int in C and INTEGER in FORTRAN. The first value in an enumeration type is hereby mapped to 1, the second to 2, etc.

10. External Objects

Formally, external functions in Modelica 1.4 need to be functions in the mathematical sense, i.e., they do not have a memory and therefore return exactly the same result if the function is called with the same input arguments. In Modelica 2.0, additionally **external objects** are supported in C, i.e., several functions may operate on a C data structure which is passed between function calls and represents an "object memory". Example:

A table data structure may be defined in such a way, that the table data is read in a user defined format from file. Furthermore, the table is interpolated in a user defined manner in the Modelica model utilizing the last used table interval for efficiently finding the current interval, i.e., an internal memory is needed. This requires the following Modelica definition:

```
class MyTable
  extends ExternalObject;

  function constructor
    input String fileName;
    input String tableName;
    output MyTable table;
    external "C" table =
      initMyTable(fileName, tableName);
  end constructor;

  function destructor
    input MyTable table;
    external "C" closeMyTable(table);
  end destructor;
end MyTable;
```

That is, a Modelica class has to be defined as a direct subclass of the new predefined class **"ExternalObject"**. This class shall contain exactly two function definitions, called "constructor" and "destructor" (and no other elements). The constructor function is called once before the first use of the object. For each completely constructed object (here: instance of MyTable), the destructor is called once, after the last use of the object, even if an error occurs. These two functions are always called implicitly and it is not allowed to call them explicitly. The MyTable Modelica class can be used in a Modelica model in the following way:

```
model test
  MyTable table1=MyTable(
    "testTables.txt", "table1");
  MyTable table2=table1; //copy of table1
  input Real u1, u2;
  output Real y1, y2;
equation
  y1 = interpolateMyTable(table1, u1);
```

```

    y2 = interpolateMyTable(table2, u2);
end test;

```

In the declaration of "MyTable" either the MyTable constructor is called using the class-name as a function name, or a copy of another object of the same type is constructed (see table2). The objects may then be used in other external Modelica functions. Here, a special external interpolation function is used:

```

function interpolateMyTable
  input MyTable table;
  input Real u;
  output Real y;
  external "C" y =
    interpolateMyTable(table, u);
end interpolateMyTable;

```

The three external functions defined above may be implemented in C in the following way:

```

typedef struct {
  double* array;
  int nrow;
  int ncol;
  int type; /* interpolation type */
  int lastIndex; /* for search */
} MyTable;

void* initMyTable(char* fileName,
                  char* tableName) {
  MyTable* table = malloc(sizeof(MyTable));
  if ( table == NULL ) ModelicaError(
    "Not enough memory");
  // read table from file and store
  // all data in *table
  return (void*) table;
};

void closeMyTable(void* object) {
  MyTable* table = (MyTable*) object;
  if ( object == NULL ) return;
  free(table->array);
  free(table);
}

double interpolateMyTable(void* object,
                          double u) {
  MyTable* table = (MyTable*) object;
  double y;
  // Interpolate using "*table" data
  return y;
};

```

The external object interface allows, for example, convenient implementations of

- user-defined table data structures,
- access to property databases,
- sparse matrix handling with specially defined data structures to store sparse matrices,
- hardware interfaces, since the constructor and destructor are called exactly once, even in case

of error, so that the resources of the hardware are initialized and freed correctly in all situations (once the hardware is initialized, i.e., the Modelica object constructed, it is guaranteed that the destructor is called exactly once for this object when the object, i.e., the hardware, is no longer needed or when an error occurs).

11. Graphical Appearance

The graphical appearance of Modelica object diagrams has been defined informally up to Modelica version 1.4 in the respective tutorial. In Modelica 2, the graphical appearance is formally defined in the Modelica specification with several improvements, especially based on the new enumeration features. In this section the most important properties are sketched. Note, that all graphical information is defined with the **annotation(...)** language element and annotations are defined to have no effect on the result of a simulation. Therefore, annotations can be ignored when generating simulation code.

A graphical representation of a class consists of two abstraction layers, **icon layer** and **diagram layer**. The icon representation visualizes the component by hiding hierarchical details. The hierarchical decomposition is described in the diagram layer showing icons of sub-components.

Icon and diagram layer are described by different coordinate systems which means that the shape and size of the two layers are independent from each other. This is different to previous versions of Modelica where only one coordinate system is defined for both layers. As a result, in Modelica 2 it is easier to arrive at nice looking drawings, because connectors may have different sizes in the icon and diagram layer and because a resizing of the diagram or the icon layer does not influence the size of the corresponding other layer. All size information, e.g., the size of icons and diagrams, the thickness of a line or the size of a font, is defined with the predefined type DrawingUnit:

```

type DrawingUnit =
  Real(final unit="mm");

```

The interpretation of "unit" in "mm" is with respect to printer output in natural size (not zoomed). Therefore, a rectangle with width=20 DrawingUnits, height = 10 DrawingUnits and line thickness of 0.5 DrawingUnits will be output as a

rectangle with 20 mm width, 10 mm height and 0.5 mm line thickness on a printer. The representation on screen is not formally defined. It is typically a direct mapping of "mm" to "pixels", e.g., 1 mm in "natural size" is typically mapped to 4 pixels. On high resolution screens, this mapping may be different.

The properties of graphical objects are mostly defined with enumerations, e.g.,

```
type LinePattern = enumeration(
    None, Solid, Dash, Dot,
    DashDot, DashDotDot);
```

Colors are defined as RGB values

```
type Color=Integer[3] (min=0,max=255)
```

There is a set of predefined graphical primitives - Line, Polygon, Rectangle, Ellipse, Text, Bitmap - which may have graphical properties such as lineColor, fillColor, linePattern, fillPattern, borderPattern, lineThickness. For Text primitives, the font name and the font size can be defined. All graphical primitives are placed by defining the placement of the corresponding object coordinate system together with additional attributes to scale, rotate, flip the object.

Note: a Modelica tool is free to define and use other graphical attributes, in addition to those defined in the Modelica specification. The only requirement is that any tool must be able to save files with all attributes intact, including those that are not used. To ensure this, annotations shall be represented with constructs according to the Modelica grammar.

12. Outlook

We have this far described the status of Modelica 2.0. Some minor extensions have not been mentioned, such as the "smooth" operator and the "elseif" clause of if-expressions. In the near future we can also expect the Modelica 2.0 libraries, and in particular the blocks library, redesigned as described above. In addition a ModelicaFunctions library with matrix operations (linear algebra) will be made available and the new rules for variable number of input and output arguments will make it possible to provide one function easily usable both by experts and novices.

The ModelicaFunctions can also be used interactively, as well as other functions and we

expect more use of Modelica scripts and potentially a formal definition of such scripts, and API-functions to access model properties from scripts. Other free libraries are also under development, e.g., for 1-dim. heat transfer and for 3-dim. vehicle dynamics.

From the language point of view some areas where improvements are needed is already clear, e.g., enumerations (as described above), impulses (eliminating the need for the reinit-operator [5]), heterogeneous arrays and PDEs (automatic discretization). More advanced use of the language and construction of large libraries and models will probably help in discovering areas where the specification can be made clearer and where further enhancement of the language is needed to better support the growing number of users of Modelica.

Bibliography

- [1] Dymola, *Dynasim AB*, Lund, Sweden, version 5.0, <http://www.dynasim.se>.
- [2] Mattsson S.E., Elmqvist H., and Olsson H.: Means to Control the Selection of States in Modelica (white paper of Dynasim), Nov. 2001.
- [3] Mattsson S.E., Elmqvist H., Otter M., and Olsson H.: Initialization of Hybrid Differential-Algebraic Equations. Modelica 2002, Oberpfaffenhofen, pp. 9-15, March 18.-19., 2002.
- [4] Mattsson S.E., Olsson H. and Elmqvist H: Dynamic Selection of States in Dymola. Modelica'2000, Oct. 2000.
- [5] Mattsson S.E., Olsson H. and Elmqvist H: Varying structure Hybrid DAE (white paper of Dynasim), Jun. 2001.

Initialization of Hybrid Differential-Algebraic Equations in Modelica 2

Sven Erik Mattsson[†], Hilding Elmqvist[†], Martin Otter[‡] and Hans Olsson[†]

[†]Dynasim AB, Lund, Sweden, {svenerik, elmqvist, hans}@dynasim.se

[‡]DLR, Oberpfaffenhofen, Germany, martin.otter@dlr.de

Abstract

Modelica 2 provides new powerful language constructs for specifying initial conditions. Before any operation is carried out with a Modelica model, such as simulation or linearization, initialization takes place to assign consistent values for all variables, derivatives and pre-variables present in the model. To obtain consistent values, the initialization uses all equations and algorithms that are utilised during the simulation. Additional constraints necessary to determine the initial values of all variables can be provided as start values for any variables as well as additional constraint equations in initial equation sections. A novel feature is the possibility to have a sampled controller initialized in steady state. This tutorial paper describes and explains the new language constructs and illustrates how they in combination with Modelica's other language elements allow very flexible and powerful initialization conditions to be defined.

1. Introduction

A dynamic model describes how the states evolve with time. The states are the memory of the model, for example in mechanical systems positions and velocities. When starting a simulation, the states need to be initialized.

For an ordinary differential equation, ODE, in state space form, $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, t)$, the state variables, \mathbf{x} , are free to be given initial values. However, more flexibility in specifying initial conditions than setting state variables is needed. In many cases we would like to start at steady state implying that the user specifies $\dot{\mathbf{x}} = 0$ as initial condition to get the initial values of \mathbf{x} calculated automatically by solving $\mathbf{f}(\mathbf{x}, t) = 0$. Besides the states, a model has also other variables and in many cases it is natural to specify initial conditions in terms of these variables.

In January 2002, Modelica 2 was released [3]. The new language constructs permit flexible specification of initial conditions as well as the correct solution of difficult, non-standard initialization problems occurring in industrial applications. Modelica 2 provides a mathematically rigid specification of the initialization of hybrid differential algebraic equations.

Dymola [1,2] supports the new language constructs of Modelica 2. Earlier Dymola versions had pure numeric support for initialization. Experiences from industrial applications including closed kinematics loops and thermodynamic problems showed that this was not sufficient. The numerical solvers were often not able to solve the large and non-linear problems. Now Dymola also manipulates symbolically the initialization problem and generates analytic Jacobians for nonlinear subproblems. Experience indicates that this approach is more robust and reliable. Moreover, the special analysis of the initialization problem allows Dymola to give diagnosis and user guidance when the initialization problem turns out not to be well posed.

This paper describes the language constructs to specify initial conditions and examples for the usage are given.

2. Basics

Before any operation is carried out with a Modelica model, especially simulation, initialization takes place to assign consistent values for all variables present in the model. During this phase, also the derivatives, **der**(...), and the **pre**-variables, **pre**(...), are interpreted as unknown algebraic variables. The initialization uses all equations and algorithms that are utilized during the simulation.

Additional constraints necessary to determine the initial values of all variables can be provided in two ways:

1. Start values for variables
2. Initial equations and initial algorithms

For clarity, we will first focus on the initialization of continuous time problems because there are some differences in the interpretation of the start values of continuous time variables and discrete variables. Also there are special rules for the usage of when clauses during initialization. All this makes it simpler to start discussing pure continuous time problems and after that discuss discrete and hybrid problems.

3. Continuous time problems

Initial equations and algorithms

Variables being subtypes of Real have an attribute *start* allowing specification of a start value for the variable

```
Real v(start = 2.0);
parameter Real x0 = 0.5;
Real x(start = x0);
```

The value for start shall be a parameter expression.

There is also another Boolean attribute *fixed* to indicate whether the value of start is a *guess* value (*fixed* = **false**) to be used in possible iterations to solve nonlinear algebraic loops or whether the variable is required to have this value at start (*fixed* = **true**). For constants and parameters, the attribute *fixed* is by default **true**, otherwise *fixed* is by default **false**.

For a continuous time variable, the construct

```
Real x(start = x0, fixed = true);
```

implies the additional initialization equation

```
x = x0;
```

Thus, the problem

```
parameter Real a = -1, b = 1;
parameter Real x0 = 0.5;
Real x(start = x0, fixed = true);
equation
  der(x) = a*x + b;
```

has the following solution at initialization

```
a      := -1;
b      := 1;
x0     := 0.5;
x      := x0;      // = 0.5
der(x) := a*x + b; // = 0.5
```

Initial equations and algorithms

A model may have the new sections **initial equation** and **initial algorithm** with additional equations and assignments that are used solely in the initialization phase. The equations and assignments in these initial sections are viewed as pure algebraic constraints between the initial values of variables and possibly their derivatives. It is not allowed to use when clauses in the initial sections.

Steady state

To specify that a variable *x* shall start in steady state, we can write

```
initial equation
  der(x) = 0;
```

A more advanced example is

```
parameter Real x0;
parameter Boolean steadyState;
parameter Boolean fixed;
Real x;
initial equation
  if steadyState then
    der(x) = 0;
  else if fixed then
    x = x0;
  end if;
```

If the parameter *steadyState* is **true**, then *x* will be initialized at steady state, because the model specifies the initialization equation

```
initial equation
  der(x) = 0;
```

If the parameter *steadyState* is **false**, but *fixed* is **true** then there is an initialization equation

```
initial equation
  x = x0;
```

If both *steadyState* and *fixed* are **false**, then there is no initial equation.

The approach as outlined above, allows *x0* to be any expression. When *x0* is a parameter expression, the specification above can also be given shorter as

```
parameter Real x0;
parameter Boolean fixed;
parameter Boolean steadyState;
Real x(start = x0,
        fixed = fixed and
                not steadyState);
initial equation
  if steadyState then
    der(x) = 0;
  end if;
```

Mixed Conditions

Due to the flexibility in defining initialization equations in Modelica 2, it is possible to formulate more general initial conditions: For example, an aircraft needs a certain minimum velocity in order that it can fly. Since this velocity is a state, a useful initialization scheme is to provide an initial velocity, i. e., an initial value for a *state*, and to set all other *state derivatives* to zero. This means, that a mixture of initial states and initial state derivatives is defined.

How many initial conditions?

How many initial conditions are needed for a continuous time problem?

For an ordinary differential equation, ODE, in state space form, $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, t)$, *exactly* $\dim(\mathbf{x})$ additional conditions are needed, in order to arrive at $2 \cdot \dim(\mathbf{x})$ equations for the $2 \cdot \dim(\mathbf{x})$ unknowns $\mathbf{x}(t_0)$ and $\dot{\mathbf{x}}(t_0)$.

The situation is more complex for a system of differential algebraic equations, DAE,

$$\mathbf{0} = \mathbf{g}(\dot{\mathbf{x}}, \mathbf{x}, \mathbf{y}, t)$$

where $\mathbf{x}(t)$ are variables appearing differentiated, $\mathbf{y}(t)$ are algebraic variables and $\dim(\mathbf{g}) = \dim(\mathbf{x}) + \dim(\mathbf{y})$. Here it can only be stated that *at most* $\dim(\mathbf{x})$ additional conditions $\mathbf{h}(\cdot)$ are needed in order to arrive at $2 \cdot \dim(\mathbf{x}) + \dim(\mathbf{y})$ equations for the same number of unknowns, $\dot{\mathbf{x}}(t_0)$, $\mathbf{x}(t_0)$, $\mathbf{y}(t_0)$:

$$\mathbf{0} = \begin{bmatrix} \mathbf{g}(\dot{\mathbf{x}}(t_0), \mathbf{x}(t_0), \mathbf{y}(t_0), t_0) \\ \mathbf{h}(\dot{\mathbf{x}}(t_0), \mathbf{x}(t_0), \mathbf{y}(t_0), t_0) \end{bmatrix}$$

The reason is that the DAE problem may be a higher index DAE problem, implying that the number of continuous time states is less than $\dim(\mathbf{x})$.

It may be difficult for a user of a large model to figure out how many initial conditions have to be added, especially if the system has higher index. At translation Dymola performs an index reduction and selects state variables. Thus, Dymola establishes how many states there are. If there are too many initial conditions, Dymola outputs an error message indicating a set of initial equations or fixed start values from which initial equations must be removed or start values inactivated by setting `fixed = false`.

If initial conditions are missing, Dymola makes automatic default selection of initial conditions. The approach is to select continuous time states with inactive start values and make their start values active by turning their fixed attribute to `true` to get a structurally well posed initialization problem. A message informing about the result of such a selection can be obtained.

Interactive setting of start values

The initial value dialogue of the Dymola main window has been redesigned. Previously, it included all continuous time states. Now it includes the continuous time variables having active start values i.e., `fixed=true` and the start value being a literal. Setting parameters may of course influence an active start value bound to a parameter expression.

When setting variables from scripts Dymola generates a warning if setting the variable has no effect what so ever, e.g. if it is a structural parameter.

Non-linear algebraic loops

A non-linear algebraic problem may have several solutions. During simulation a numerical DAE solver tends to give the smoothest solution. A DAE solver is assumed to start at a consistent point and its task is to calculate a new point along the trajectory. By taking a sufficiently small step and assuming the existence of a Jacobian that is non-singular there is a local well-defined solution.

The initialization task is much harder and precautions must be taken to assure that the correct solution is obtained. The means to guide the solver include min and max values as well as start values for the unknowns.

As a simple example, consider a planar pendulum with fixed length L .

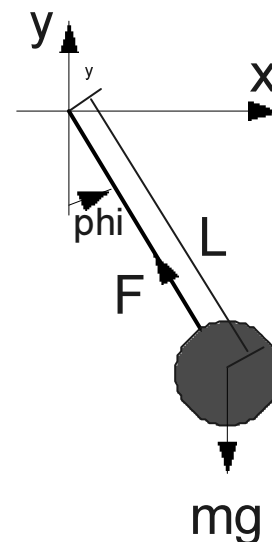


Figure 1: A planar pendulum.

The position of the pendulum can be given in polar coordinates. Introduce an angle, ϕ , that is zero, when the pendulum is hanging downward in its rest position. The model can be given as

```
parameter Real g = 9.81;
parameter Real m = 1;
parameter Real L = 1;
Real phi, w;
equation
  der(phi) = w;
  m*der(w) = -(m*g/L)*sin(phi);
```

Assume now that we want to specify the initial condition in Cartesian coordinates defined as

```
x = L*sin(phi);
y = -L*cos(phi);
```

If we define

```
Real y(start = 0; fixed = true);
```

the pendulum will start in a horizontal position. However, there are two horizontal positions, namely

```
x = -L and x = L
```

To indicate preference for a positive value for x , we can define

```
Real x(start = L);
```

It means that we provide a guess value for numerical solvers to start from. They will hopefully find the positive solution for x , because, it is closer to L than the negative solution.

For the angle ϕ there are many values giving the desired position, because adding or subtracting 2π gives the same Cartesian position. Also, here the start value can be used to indicate the desired solution. However, critical it is to get a special solution depends of course on what ϕ will be used for in the model and the aim of the simulation. If no start value is given zero is used.

4. Parameter values

Parameters are typically given values in a model through definition equation or set interactively before a simulation. Modelica 2 also allows parameter values to be given implicitly in terms of the initial values of all variables.

Recall the planar pendulum and assume that we would like to specify the initial position as

```
Real x(start = 0.3; fixed = true);
Real y(start = 0.4; fixed = true);
```

This means that we in fact also specify the length of the pendulum to be 0.5. To specify that the parameter L shall be calculated from the initial conditions, we define it as

```
parameter Real L(fixed = false);
```

Recall that the attribute `fixed` is by default **true** for constants and parameters, otherwise `fixed` is by default **false**.

The semantics of parameters in Modelica is a variable that is constant during simulation. The possibility to let the parameter value to depend on the initial values of

time dependent (continuous-time or discrete) variables does not violate this semantics.

This feature has many useful applications. It allows powerful reparametrizations of models. As an example, consider the model of an ideal resistor. It has one parameter, R , being the resistance. Assume that we would like to have use it as a resistive load with a given power dissipation at a steady state operating point. It is just to extend from the resistor model given in the Modelica Standard Library and

1. Add a parameter $P0$ to specify the power dissipation.
2. Set `fixed=false` for parameter R .
3. Add an initial equation section with $v \cdot i = P0$.

In power systems, it is common practice to specify initial conditions in steady state and use different kind of load models including resistive load and specify their steady state operating conditions in terms of active and reactive power dissipation.

In some cases parameters may be provided outside of a Modelica model and the actual values may be read from file or parameter values may be inquired from a database system during initialization:

```
parameter Real A(fixed=false);
parameter Real w(fixed=false);
Real x;
initial equation
(A,w) = readSineData("init.txt");
equation
der(x) = -A*sin(w*x);
```

5. Discrete and hybrid problems

The language constructs for specifying initial conditions for discrete variables are as for the continuous time variables: start values and initial equations and algorithms.

Variables being subtypes of Real, Integer, Boolean and String have an attribute `start` allowing specification of a start value for the variable.

For discrete variables declarations

```
Boolean b(start = false, fixed = true);
Integer i(start = 1, fixed = true);
```

imply the additional initialization equations

```
pre(b) = false;
pre(i) = 1;
```

This means that a discrete variable v itself does not get an initial value ($= v(t_0 + \epsilon)$), but the **pre**-value of v ($= v(t_0 - \epsilon)$) does.

When clauses at initialization

For the initialization problem there are special semantic rules for **when** clauses appearing in the model. During simulation a **when** clause is only active when its condition becomes **true**. During initialization the equations of a **when** clause are only active during initialization, if the **initial()** operator explicitly enables it.

```
when {initial(), condition1, ...} then
  v = ...
end when;
```

Otherwise a **when** clause is in the initialization problem replaced by $v = \text{pre}(v)$ for all its left hand side variables, because this is also the used equation during simulation, when the **when**-clause is not active.

Non-unique initialization

In certain situations an initialization problem may have an infinite number of solutions, even if the number of equations and unknown variables are the same during initialization. Examples are controlled systems with friction, or systems with backlash or dead-zones. Assume for example backlash is present. Then, all valid positions in this element are solutions of steady state initialization, although this position should be computed from initialization. It seems best to not rely on some heuristics of the initializer to pick one of the infinite number of solutions. Instead, the continuous time equations may be modified during initialization in order to arrive at a unique solution. Example:

```
y = if initial() then
  // smooth characteristics
else
  // standard characteristics
```

Well-posed initialization

At translation Dymola analyses the initialization problem to check if it is well posed by splitting the problem into four equation types with respect to the basic scalar types Real, Integer, Boolean and String and decides whether each of them are well-posed.

As described for the pure continuous-time problem, Dymola outputs error diagnosis in case of over specified problems. In case of under specified problems Dymola makes automatic default selection of initial conditions.

How many initial conditions?

Basically, this is very simple: Every discrete variable v needs an initial condition, because $v(t_0 - \epsilon)$ is otherwise not defined. Example:

```
parameter Real t1 = 1;
discrete Real u(start=0, fixed=true);
Real x(start=0, fixed=true);
equation
  when time > t1 then
    u = ...
  end when;
der(x) = -x + u;
```

During initialization and before the **when**-clause becomes active the first time, u has not yet been assigned a value by the **when**-clause although it is used in the continuous part of the model. Therefore, it would be an error, if **pre**(u) would not have been defined via the start value in the u declaration.

On the other hand, if u is used solely inside this **when**-clause and **pre**(u) is not utilized in the model, an initial value for u may be provided but *does not influence* the simulation, because the first access of u computes u in the **when**-clause and afterwards u is utilized in other equations inside the **when**-clause, i. e., the initial value is never used.

Since it may be tedious for a modeller to provide initial values for all discrete variables, Modelica 2 only requires to specify initial values of discrete variables which influence the simulation result. Otherwise, a default value may be used.

6. Example: Initialization of discrete controllers

Below four variants to initialize a simple plant controlled by a discrete PI controller are discussed.

Variant 1: Initial values are given explicitly

```
parameter Real k=10, T=1;
  // PI controller parameters.
parameter Real Ts = 0.01 "Sample time";
input Real xref "reference input";

Real x (fixed=true, start=2);
discrete Real xd(fixed=true, start=0);
discrete Real u (fixed=true, start=0);

equation
  // Plant model
  der(x) = -x + u;

  // Discrete PI controller
  when sample(0, Ts) then
    xd = pre(xd) + Ts/T*(xref - x);
    u = k*(xd + xref - x);
  end when;
```

The model specifies all the initial values for the states explicitly. The when clause is not enabled at initialization but it is replaced by

```
xd      := pre(xd)
u       := pre(u)
```

The initialization problem is thus

```
x      := x.start  // = 2
pre(xd) := xd.start // = 0
pre(u)  := u.start  // = 0
xd      := pre(xd)  // = 0
u       := pre(u)   // = 0
der(x)  := -x + u   // = -2
```

Variant 2: Initial values are given explicitly and the controller equations are used during initialization. It is as Variant 1, but the **when** clause is enabled

```
// Same declaration as variant 1
equation
der(x) = -x + u;

when {initial(), sample(0,Ts)} then
  xd = pre(xd) + Ts/T*(xref - x);
  u  = k*(xd + xref - x);
end when;
```

It means that the **when** clause appears as

```
xd = pre(xd) + Ts/T*(xref - x);
u  = k*(xd + xref - x);
```

in the initialization problem, which becomes

```
x      := x.start  // = 2
pre(xd) := xd.start // = 0
pre(u)  := u.start  // = 0
xd      := pre(xd) + Ts/T*(xref - x);
u       := k*(xd + xref - x);
der(x)  := -x + u;
```

Variant 3: As Variant 2 but initial conditions defined by initial equations

```
discrete Real xd;
discrete Real u;
```

```
// Remaining declarations as in variant 1
equation
der(x) = -x + u;
when {initial(), sample(0, TS)} then
  xd = pre(xd) + Ts/T*(xref - x);
  u  = k*(xd + xref - x);
end when;
```

```
initial equation
pre(xd) = 0;
pre(u)  = 0;
```

leads to the following equations during initialization

```
x      := x.start  // = 2
pre(xd) := 0
pre(u)  := 0
```

```
xd      := pre(xd) + Ts/T*(xref - x)
u       := k*(xd + xref - x)
der(x)  := -x + u;
```

Variant 4: Steady state initialization

Assume that the system is to start in steady state. For continuous time state, x , it means that its derivative shall be zero; $\text{der}(x) = 0$; While for the discrete state, xd , it means $\text{pre}(xd) = xd$; and the when clause shall be active during initialization

```
Real      x (start=2);
discrete Real xd;
discrete Real u;
```

```
// Remaining declarations as in Variant 1
equation
// Plant model
der(x) = -x + u;
```

```
// Discrete PID controller
when {initial(), sample(0, Ts)} then
  xd = pre(xd) + Ts/T*(x - xref);
  u  = k*(xd + x - xref);
end when;
```

```
initial equation
der(x)  = 0;
pre(xd) = xd;
```

The initialization problem becomes

```
der(x) := 0
```

```
// Linear system of equations in the
// unknowns: xd, pre(xd), u, x
pre(xd) = xd
xd      = pre(xd) + Ts/T*(x - xref)
u       = k*(xd + xref - x)
der(x)  = -x + u;
```

Solving the system of equations leads to

```
der(x) := 0
x      := xref
u       := xref
xd      := xref/k
pre(xd) := xd
```

7. Conclusions

This paper has described and illustrated how the new language constructs of Modelica 2 in combination with Modelica's other language elements allow very flexible and powerful initialization conditions to be defined.

Dymola supports Modelica's new way of specifying initial conditions. To support reliable and robust initialization, Dymola manipulates symbolically the initialization problem and generates analytic Jacobians for nonlinear subproblems. Moreover, the special analysis of the initialization problem allows Dymola to

give diagnosis and user guidance when the initialization problem turns out not to be well posed.

Acknowledgements

This work was in parts supported by the European Commission under contract IST-199-11979 with Dynasim AB under the Information Societies Technology as the project entitled "Real-time simulation for design of multi-physics systems".

8. References

- [1] D. Brück, H. Elmqvist, S.E. Mattsson, H. Olsson: *Dymola for Multi-Engineering Modeling and Simulation*, Proceedings of Modelica 2002. Modelica homepage: <http://www.Modelica.org>,
- [2] Dymola. *Dynamic Modeling Laboratory*, Dynasim AB, Lund, Sweden, <http://www.Dynasim.se>
- [3] Modelica. *A unified object-oriented language for physical systems modelling - Language Specification*. Version 2.0, Modelica homepage: <http://www.Modelica.org>,

Session 2

Applications and Tools

Evaluating Engine Contributions to HEV Driveline Vibrations

Michael Tiller, William E. Tobler and Ming Kuang

Ford Motor Company

Abstract

In order to comply with increasing consumer and regulatory demand for improved fuel economy and lower emissions, Ford Motor Company is developing a Hybrid Electric Vehicle (HEV) version of the Escape sport utility vehicle for production in 2003. Since HEVs typically have several different operating modes (e.g. electric launch, active neutral, regenerative braking), an important concern is the fact that each of these modes and the transitions between them lead to minimal driver perceived vibrations. In order to understand how the design and control of an HEV influences what is "felt" by the driver, we need to build models that accurately reproduce the dynamic response of the powertrain. In this way, the response for a given mechanical configuration and/or controller design can be evaluated.

A model targeted at prediction of driver perceived vibration was developed and validated against experimental data. However, one unexpected result of this work was to demonstrate that we could take the dynamic model used to reproduce the behavior described previously and, by using some advanced Modelica features, derive a second model that predicts the system efficiency of the transmission without having to create an entirely new model for that purpose. The system efficiency model was also validated against experimental data and showed very good agreement. The result is that rather than spending time creating and maintaining two different models (one for dynamic response and one for system efficiency) we were able to build one on the foundation of the other. Furthermore, we determined it was possible to generate a single model that could describe both types (i.e. dynamic and steady-state) of responses by merely changing the values of a few model parameters.

Introduction

The idea of using computer-aided methods to evaluate powertrain and vehicle NVH (i.e. noise, vibration and harshness) is not new [1,2]. Furthermore, the use of Modelica to model automotive systems is increasing [3,4,5,6,7]. However, the contribution that the internal combustion engine makes as a "forcing function" to a powertrain system is not typically examined in detail since the steady state operation of the engine is well understood and sufficient for most applications. For HEVs though, the engine starts and stops frequently, both with the vehicle in motion and at rest, and this

makes a significant contribution to vibration perceived by the driver.

In order to understand the effects of powertrain design and control on driver perceived vibrations, a detailed thermodynamic model of an Atkinson cycle internal combustion engine was developed and integrated with a detailed model of a hybrid electric transmission. The computational model was then validated against experimental data and showed very close agreement. With this validated dynamic model of the powertrain, we analyzed the effect that changes in the mechanical design of the powertrain had on the natural frequencies of the vehicle, examined the effect that engine control parameters (e.g. spark timing and valve timing) had on powertrain response and created realistic Simulink plant models which could be used to test different control strategies.

Physical Models

The focus of paper is on modeling the physical response of the powertrain. Issues about control system design or strategy are larger issues beyond the scope of this paper. Nevertheless, a good physical model of the powertrain can provide useful insights for both the hardware and control system designers.

In particular, we are interested in predicting the sensitivity of the powertrain response with respect to component design parameters and actuator commands. To preserve the effects of design parameters, it is generally necessary to provide design-oriented models built from first-principles based component models rather than models derived from empirical relationships or experimental data.

Our discussion of modeling efforts will start with some general modeling issues and then present details of the engine, transmission and vehicle models used in this work.

Control Signals

Both the engine and transmission subsystems contain components that require control signal inputs (e.g. spark timing and motor torque). One interesting problem that arises when actuator and sensor models are included is the need to communicate these control signals into and out of the physical model hierarchy. The difficulty is in managing the propagation of these signals especially in the context of replaceable components.

For example, we may develop an HEV model that includes an engine with certain control signals (*e.g.* spark timing and injector timing). At some later point, we may wish to create a variation of that model by simply extending the original model and replacing the engine with different engine model. The difficulty comes when the new engine happens to have a different set of control inputs (*e.g.* cam phasing). In order to propagate these new signals, this could require the model developer to add a whole new set of connectors up and down the model hierarchy.

To avoid this situation, we use something in our Modelica models that we call the *SignalBus* idiom. In this approach, all the signals associated with each subsystem are grouped onto a "master" bus (*e.g.* `eng_control_bus`) at the top-level of the model. The *SignalBus* idiom is useful because the component models only need to be aware of *the specific signals they require* and not all signals on the master bus.

```
connector SignalBus
  annotation(...);
end SignalBus;

model FuelInjector
  outer ControlBus eng_control_bus;
  // ...
protected
  connector ControlBus
    extends SignalBus;
    Ford.Types.Degree inj_start;
    Ford.Types.Degree inj_stop;
  end ControlBus;
end FuelInjector;

model FullVehicle
  inner EngineMaster eng_control_bus;
  Engine eng "has fuel injectors";
protected
  connector EngineMaster
    extends SignalBus;
    Ford.Types.Degree inj_start;
    Ford.Types.Degree inj_stop;
    Ford.Types.Degree spark_adv;
  end EngineMaster;
end FullVehicle;
```

Figure 1: Example of SignalBus Idiom

To implement the *SignalBus* idiom, we define an empty connector with a specific graphical annotation. Although not strictly necessary, it makes the bus connectors very easy to identify in diagrams. Next, inside each component requiring control signals (*e.g.* a fuel injector), we declare a specific bus type for that component (preferably in a *protected* section to clearly indicate that this definition is for internal use). The bus definition should include only the signals required by the component. This bus can then be

instantiated with the *outer* qualifier. The name of the instance should be that of the master bus where the signals ultimately reside. At the top-level, the master bus type must contain (at least) the union of all subsystem component buses and an inner instance must be declared. An example of the definitions and declarations required is shown in Figure 1.

The *SignalBus* idiom has the following advantages over signals. First, it avoids the necessity to place connectors at each level in the hierarchy. This is important because every change in control signals can potentially change the set of connectors and connections, a situation that becomes difficult to maintain. In addition, because the *outer* bus only has to be a subtype of the matching inner bus, the component models are only required to declare the signals they are interested in. This avoids dealing with complex combinatorial possibilities that result when all signals are included in a single connector definition. One disadvantage with *SignalBus* definitions is that responsibility for assigning the control signals is not clearly specified by the definition. Instead, this requires some discipline and understanding of the idiom.

Trying to decide on the best logical grouping for the signals could be an involved task. Fortunately, there are developing internal corporate standards or identifying and grouping control signals and the *SignalBus* idiom fits nicely into these emerging standards.

Reaction Torques

One limitation of the current rotational mechanics library in the Modelica Standard Library is the fact that it neglects reaction torques on rotational components. For example, consider the *IdealGear* model definition shown in Figure 2. The problem with the *IdealGear* model is that it contains the equation:

$$R\tau_a + \tau_b = 0$$

which, in general, results in the torques not summing to zero for this component. Since the torques represent the flow of angular momentum, angular momentum is not conserved.

```
within Modelica.Mechanics.Rotational;
model IdealGear "without inertia"
  parameter Real ratio "Gear ratio";
  Interfaces.Flange_a flange_a;
  Interfaces.Flange_b flange_b;
equation
  flange_a.phi=ratio*flange_b.phi;
  0=ratio*flange_a.tau+flange_b.tau;
end IdealGear;
```

Figure 2: Standard IdealGear Model

While for many applications the models in the Modelica Standard Library are sufficient, it is necessary to include an additional flange in applications where the entire

geartrain assembly has the potential to rotate. For our application, we are interested in the motion of the transmission housing and engine block and as a result, we must include a special flange on many of our component models (*e.g.* electric motors, spur gears, crank-slider mechanisms) to account for the reaction torque which ultimately causes vibrations in the powertrain casing. In most cases, it is necessary to formulate the reaction torques by considering conservation of momentum and conservation of energy.

Engine Modeling

One of the key features of the models developed for this application is the ability to predict the torque generated by the engine during startup and shutdown. In order to predict this torque, it is necessary to model some of the detailed thermodynamic processes of the engine (*e.g.* breathing, compression, combustion). Fortunately, we had already developed, prior to this application, a library of thermodynamic components for the purpose of studying engine behavior [6,8,9].

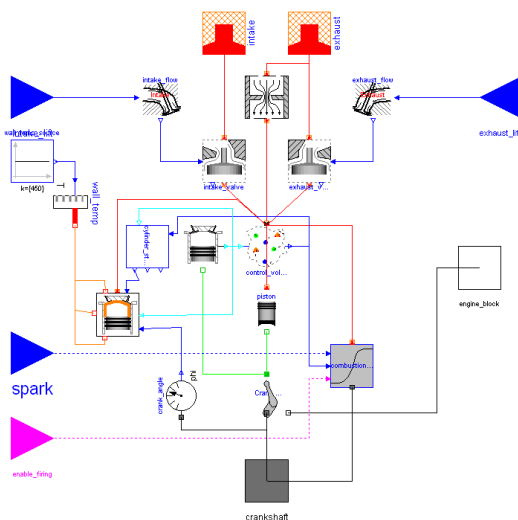


Figure 3: Combustion Chamber Processes

The engine model uses the same geometry and valve timing as the intended production engine to ensure that the predicted torque fluctuations have the same characteristics as the actual engine. To accomplish this, the engine model was developed such that it could reproduce effects due to throttle position, spark timing, cam phasing, valve lift profiles, engine geometry and injection timing during both startup and shutdown. This involves modeling the behavior of manifold filling and emptying, variable valve timing mechanisms, combustion and the application of cranking torque. Several of these behaviors are represented in the combustion chamber schematic shown in Figure 3.

Transmission Modeling

Using rotational components that account for the necessary reaction torques, construction of transmission models is straightforward. Unlike our previous transmission models which included hydraulic subsystems [4], the only complex behavior in the hybrid transmission is related to frictional elements and these can all be captured using the components in the Modelica Standard Library. In other words, no complex models had to be developed in order to build a reasonable model of the transmission.

To accurately predict the behavior of the hybrid transmission, several effects must be considered. First, a non-linear spring is connected to the input shaft of the transmission to isolate the transmission from the high-frequency torque fluctuations produced by the engine. In addition, the differential on the output side of the transmission includes a single backlash used to represent the backlash distributed throughout the transmission.

Vehicle Modeling

The vehicle response model is quite simple and neglects effects due to tire and suspension compliance. Currently, we treat the vehicle as a single mass connected by a kinematic tire model. The only real detail of the vehicle model is in the modeling of the front halfshafts (our current model handles only the front wheel drive configuration of the powertrain) which are modeled as non-symmetric compliances connecting the transmission to the wheels. In the future, we plan on refining our model to include suspension, tire and driveline details so that we are able to predict driver seat accelerations due to powertrain vibrations. It has been shown previously [4] that such large and complex models can be expressed in Modelica and simulated using Dymola.

Analyses

Dynamic Response

The dynamic response of the powertrain is due to the various inertias (*e.g.* gears, shafts, flywheel) and compliances (*e.g.* halfshafts and engine mounts) distributed throughout the system. The compliances are all modeled as linear with the exception of the isolation element on the input shaft of the transmission which is modeled as a piecewise linear spring. In addition to the inertias and the compliances there are several non-linear elements. While the transmission includes several frictional elements, they are not involved during start stop operation. Finally, as previously mentioned, all backlash in the transmission is lumped at the differential on the output shaft.

The dynamic response of the transmission can be modeled in Dymola and as we shall see later in the 'Validation' section, the results show close agreement

with experimental results. Because of the simplicity of our current vehicle model, the results we focus on are the halfshaft torque trajectories. However, interpreting the time domain results by inspection of the trajectories is not a very good way of establishing the "quality" of the startup or shutdown operation. Instead, we use a signal processing algorithm which reduces the time domain trajectories down to a scalar value. Using this information, we can then generate plots of the startup and shutdown quality as a function of spark timing, cam phasing, *etc.* While we are not in a position to discuss the results of such analyses, they have proved quite useful in identifying what factors contribute to powertrain vibrations.

Steady-State Response

One of the unexpected results of this work was to demonstrate that additional types of analyses could be performed using the model initially developed for studying dynamic powertrain response. Once we had established our ability to predict dynamic response of the powertrain, we were asked whether we could apply our model to understanding some experimental efficiency data taken on a powertrain dynamometer. The experimental results had shown what appeared to be anomalous data points during the testing and the question was whether the model could explain these anomalies.

To study the problem, we went back to our dynamic response model and made all the geartrains in the transmission replaceable. We then created a new transmission model for studying steady-state efficiency issues by extending our dynamic model and redeclaring all the geartrains so that steady-state efficiency data for each geartrain could be provided to the model. In other words, we took our original dynamic model and redeclared all the gears to include more detailed gear models necessary for studying steady-state efficiency.

In addition to redeclaring some of the components, several additional modifications were required. However, none of these modifications required changes to the original model but could instead be accomplished via the modification semantics in Modelica and by the addition of some new components. The first modification was to add some slight parasitic losses for some of the frictional elements. These losses could be introduced through modifications to the parameters in the original model. The other big change for the steady-state response was to eliminate the compliances since they only play a role in the dynamic response of the powertrain.

Thankfully, eliminating the compliances did not require crude methods like making the stiffness of the elements extremely large. This would not have eliminated the dynamics but would have just shifted the natural frequencies until they were extremely high. Instead, we employed a technique which is quite easy in Modelica.

We created the `RigidBypass` model shown in Figure 4. Placing an instance of this model in parallel with all our compliances allowed us, just by changing the value of the `rigid` parameter, to eliminate completely all compliance in the model.

Another important difference between the dynamic and steady-state response models is how they were used. The dynamic response model was used in the context of a vehicle simulation where the vehicle moves in response to the output torque of the transmission. On the other hand, the steady-state response model was used to reproduce the results of experiments conducted on a powertrain dynamometer where the speeds of various elements were fixed. To analyze our model we had to place the transmission on a virtual powertrain dynamometer. Once again, the experimental and model results showed good agreement.

```

model RigidBypass
  import Modelica.Mechanics.Rotational;
  parameter Boolean rigid;
  Rotational.Interfaces.Flange_a a;
  Rotational.Interfaces.Flange_b b;
equation
  if rigid then
    a.tau + b.tau = 0;
    a.phi = b.phi;
  else
    a.tau = 0;
    b.tau = 0;
  end if;
end RigidBypass;

```

Figure 4: RigidBypass Model

Frequency response

Another type of analysis that we could do quite easily with these models was to study the frequency response of the powertrain. With this capability, we could then study the effect that different design and control changes had on the poles of the system.

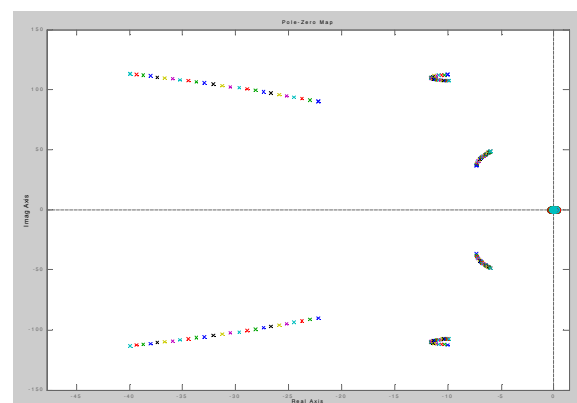


Figure 5: Design Dependence of Poles

To perform this analysis we used the "Linearize" and scripting functionality in Dymola [10] to generate a

linear time invariant system of equations for several different sets of design parameters. To do this properly, we needed to find a state where the backlash was taken up and the engine isolation spring was in the appropriate behavioral regime. We could then use the "Itviewer" functionality in MATLAB [11] to visualize the poles and zeros and to study how the poles moved in response to changes in hardware or controller design. Figure 5 shows one example of how the poles are visualized.

Validation

Dynamic Response

To validate the dynamic response of the powertrain, we used experimental data collected from vehicle tests conducted on our test track. The experiments involved starting the engine and looking at the resulting engine speed and halfshaft torque trajectories. The tests themselves were conducted with a closed loop controller. For our validation, we extracted the actuator signals used in the test and applied them in an open loop fashion to our model. The vehicle testing consisted of 27 different experiments involving 9 different controller strategies.

Figure 6 shows a comparison between the engine speed measured during the testing (dotted green line) and what the model predicts (blue line) based on the same actuator commands. The effects of the first few compression strokes can be seen as distinct bumps in the engine speed profiles.

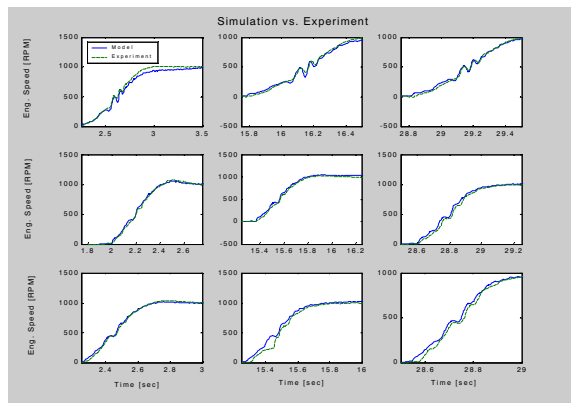


Figure 6: Validation of Dynamic Engine Response

Similarly, Figure 7 shows a comparison (during the same experiments) of the halfshaft torque predicted by the model compared to the halfshaft torque measured in the experiment. The halfshaft torque results are sensitive to the initial crankshaft position and the initial gap in the backlash.

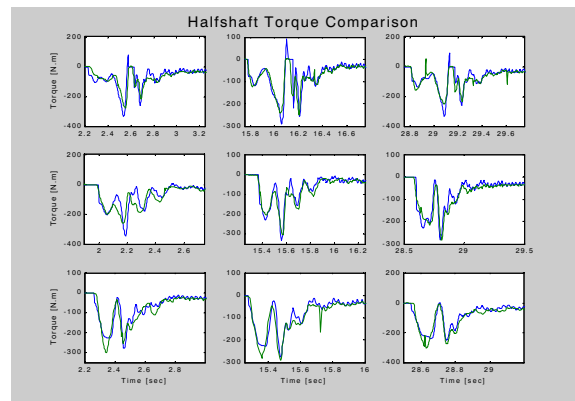


Figure 7: Validation of Dynamic Driveline Response

Steady-State Response

The steady-state response of the transmission was also validated by comparison to experimental data. However, the steady-state response is based on powertrain dynamometer data. Again, we saw good agreement between our model and the experimental data taken over a range of different operating conditions. Figure 8 shows a comparison between the experimental data (black bars) compared with the model predicted efficiency (red stars). The important thing to note in this data is how well the model predicts the conspicuously low efficiency present in some of the tests.

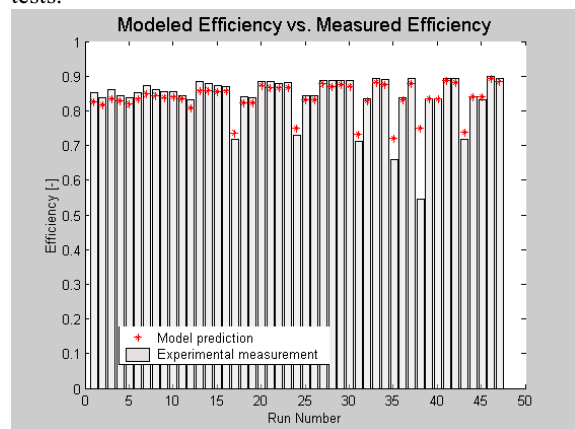


Figure 8: Validation of Steady-State Efficiency

Conclusions

There are several important points to be made about this modeling project. The models described in this paper were constructed from data about the individual components that appear in the model. Whenever such data was available, we used it. The only exception was a slight modification to the crankshaft inertia to demonstrate better agreement in engine speed trajectories. For component data that is not easily obtained or measured (*e.g.* damping ratios), we started by using "rule of thumb" numbers (which showed reasonable agreement) and then we made some small

adjustments, within reasonable limits, to calibrate those parameters so that we could achieve the agreement shown in the validation figures.

Another important point to make is the flexibility and reusability that is inherent in Modelica models. This is evidenced by our ability to do component level and powertrain level validation studies, the flexibility of using the models in different contexts (i.e. with different causalities) and the ability to reuse the dynamic models to reproduce steady-state response characteristics. This flexibility combined with the efficient code generation and solution methods in Dymola ensured that the model development and analysis process was able to provide accurate answers in a timely manner.

References

1. M. C. Tsangarides, W. E. Tobler and C. R. Heermann, "Interactive Computer Simulation of Drivetrain Dynamics", SAE 850978, *Surface Vehicle Noise and Vibration Conference Proceedings*.
2. M. C. Tsangarides and W. E. Tobler, "Dynamic Behavior of a Torque Converter with Centrifugal Bypass Clutch", SAE 850461, *Surface Vehicle Noise and Vibration Conference Proceedings*.
3. M. Otter, M. Dempsey and C. Schlegel, "Package PowerTrain. A Modelica library for modeling and simulation of vehicle power trains", *Modelica Workshop 2000 Proceedings*, pp. 23-32.
4. M. Tiller, P. Bowles, H. Elmqvist, D. Brück, S. E. Mattson, A. Möller, H. Olsson and M. Otter, "Detailed Vehicle Powertrain Modeling in Modelica", *Modelica Workshop 2000 Proceedings*, pp. 169-178.
5. M. Tiller, "Introduction to Physical Modeling with Modelica", Kluwer Academic Publishers, ISBN 0-7923-7367-7
6. C. Newman, J. Batteh and M. Tiller, "Spark-Ignited Engine Cycle Simulation in Modelica", *Second International Modelica Conference Proceedings*.
7. C. Puchalsky, T. Megli, M. Tiller, E. Curtis, N. Trask, Y. Wang, "Modelica Applications for Camless Valvetrain Development", *Second International Modelica Conference Proceedings*.
8. H. Tummescheit, M. Tiller, "Modellierung von Ottomotoren", *Objektorientierte Modellierung Physikalischer, Teil 17*.
9. M. Tiller, H. Tummescheit, C. Davis, N. Trigui, "Powertrain Modeling with Modelica", *ASME 2000 Congress Proceedings*.
10. H. Elmqvist, D. Brück, S. E. Mattsson, H. Olsson and M. Otter, "Dymola User's Guide, version 4.2a", *Dynasim AB, Sweden*
11. "MATLAB Release 12" (documentation), The MathWorks, Inc., Natick, MA.

Multidomain Systems: Electronic, Hydraulic, and Mechanical Subsystems of an Universal Testing Machine Modeled with Modelica

Christoph Clauß

Christoph.Clauss@eas.iis.fhg.de

Fraunhofer-Institut für Integrierte Schaltungen, Außenstelle Entwurfsautomatisierung
Zeunerstraße 38, 01069 Dresden, Germany

Peter Beater

Beater@mailso.uni-paderborn.de

Universität-GH Paderborn, Abt. Soest
Lübecker Ring 2, 59494 Soest, Germany

Abstract

The Simulation of hydraulic or electronic systems has been state of the art for a long time. For both of these domains there exist highly specialized simulation programs which can be regarded as a kind of industrial standards. Often problems arise if different domains of technology occur within one system and very detailed models are needed.

As an example a universal testing machine is presented which consists of hydraulic, mechanical, and electronic component systems. Each component is modeled fully detailed using the **Modelica** language [1]. Without coupling of simulators the whole simulation model can be investigated by **one** tool.

1 Introduction

The engineer of today is used to powerful simulation tools. Within the last forty years these tools mutated from simple solvers of differential equations to computer-aided design software for technical systems. Tools like HSPICE in electronics, ADAMS in mechanics, or HOPSAN in Hydraulics are highly specified to meet the needs of the discipline. These tools “know” the domain-internal peculiarities. Often the models and the simulation algorithms are closely related. Therefore, these tools are very advantageous in simulation, modeling, and postprocessing.

Often problems arise if technical systems cover more than one established discipline, e.g. in microsystems engineering. The two fundamental ways out are coupling of simulators, and compact modeling for one simulator.

From the very beginning the Modelica language is designed for covering several technical disciplines [2], [3], [4]. Complex systems can be modeled with **one** language to get **one** model. The further processing within the Dymola simulator results in **one** mathematical model, typically a differential algebraic equation, which is solved by **one** simulation core. The challenge of the Modelica approach is to show that its efficiency is not much less than the efficiency of domain specific tools. To offer evidence of this is surely a long process. In this paper the multidomain example of a universal testing machine is presented. It demonstrates that the unified multidiscipline simulation tool Modelica/Dymola meets the challenge quite well.

At first the physical device is presented with emphasizing the hydraulic and electronic parts. The Modelica model is shortly described, and simulation results are discussed. It is shown that numerical problems could be solved, and the performance can be accepted.

2 The Universal Testing Machine

Fig. 1 shows the universal testing machine. It is a simple mechanical construction of a one-sided working Plunger cylinder and a hydraulic unit on the left side in the picture. The hydraulic unit consists of a small AC motor, a variable displacement pump, and a pressure limiting valve.

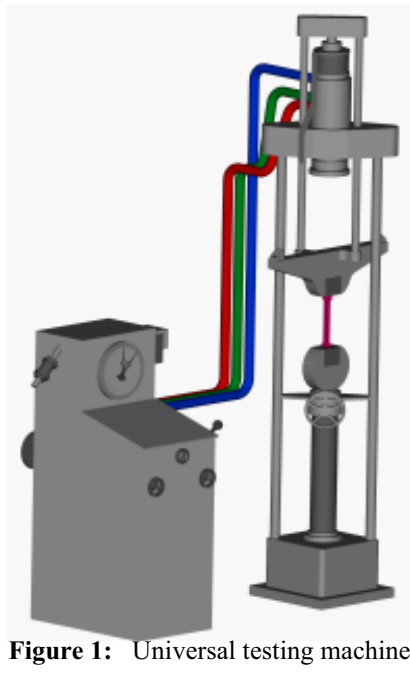


Figure 1: Universal testing machine

This kind of machines is used for tensile tests of a rod to determine e.g. the tensile strength, which is a material property. The resulting **quasi-static** stress-strain diagram describes how the material reacts under a continuously increasing load. Often the load is necessary to be regarded not as static but as periodic. In these cases the testing method has to be modified to get pulsating forces. A simple modification is like this: Within the hydraulic circuit an electro-hydraulic proportional valve of high quality is included as a by-pass to the cylinder. This valve is controlled using a sine-wave generator as reference input and a PI-controller. The machine is described in more detail in [5].

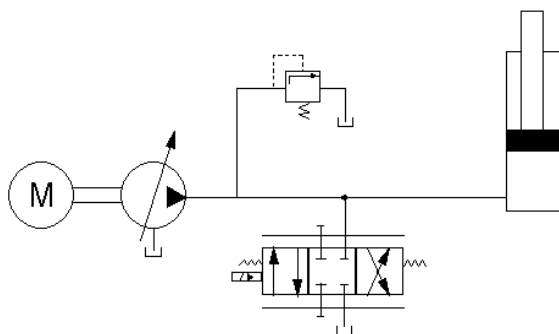


Figure 2: Hydraulic circuit of the testing machine

The task of the simulation is the investigation of the modifications before they are applied. E.g. the characteristic parameters of the valve and the electronic controller have to be determined.

3 The Hydraulic and Mechanical Parts

After preliminary work using the analogue computer in the fifties the simulation of hydraulic systems became important in the eighties. Graphical user interfaces were added in the nineties [6]. Using Modelica and its libraries it is easy to model hydraulic or mechanical systems [7]. The user needs not absolutely know the details of component modeling. If nevertheless details are essential the source code of the models is available.

Using HyLib models the hydraulic circuit according to **fig. 2** could be modeled. Since the pump is driven via a V belt transmission parts of the standard Modelica mechanics library are used to build the model according to **fig. 3**. A further mechanical component is the model of the specimen which is a linear spring.

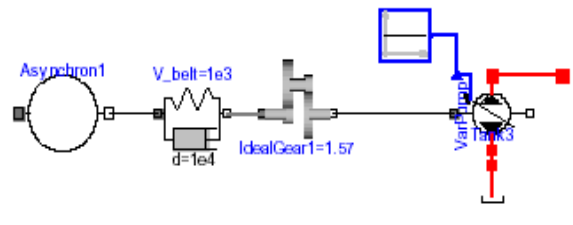


Figure 3: Model of oil source

To enable dynamic testing an electro-hydraulic valve is used as a by-pass to the cylinder. In more detail the hydraulic and mechanical parts are described in [8], [5].

4 The Electronic Part

Since 1975 SPICE [9] is available for the simulation of electronic and especially for microelectronic circuits. Later on, powerful circuit simulators with graphical and textual input possibilities were designed on SPICE. For electronic devices very comprehensive models are available which sometimes are based on semiconductor technology parameters.

In the electrical analog Modelica library [10] the most often used electrical components are collected which are easy to understand and of a wide interest. Although the SPICE semiconductor devices are still missing it is possible to model rather complicated electrical circuits.

The electronic part of the testing machine is a PID-controlling device [11], which amplifies (proportional), integrates, and differentiates the input signal. The circuit scheme can be seen in **fig. 4**.

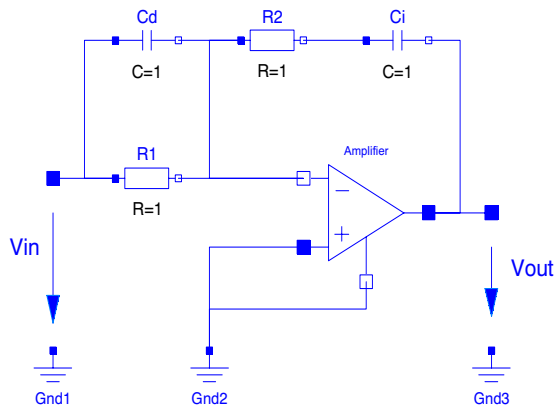


Figure 4: PID circuit

By choosing the resistances and capacitances according to

$$P = \frac{R_2}{R_1} + \frac{C_D}{C_I}, I = \frac{1}{C_I R_1}, D = C_D R_2$$

the controlling parameters P, I, and D can be adjusted.

$$V_{out} = P V_{in} + I \int V_{in} dt + D \frac{dV_{in}}{dt}$$

The operational amplifier was modeled on different abstract levels. On the transistor level the well-known $\mu A741$ [12] was used which is modeled using bipolar transistors (14 NPN, 7 PNP) of the Modelica standard library.

The numbers of the values of currents in the electronic part are orders of magnitude smaller than the numbers of values in the hydraulic part. Small capacitances in the transistors cause very short transient responses. Therefore the mathematical model becomes stiff, which is a challenge for the simulation system.

The bipolar transistors are modelled in the most simple way according to the Ebers-Moll-approach [13], [14]. The circuit structure (**fig. 5**) shows the components

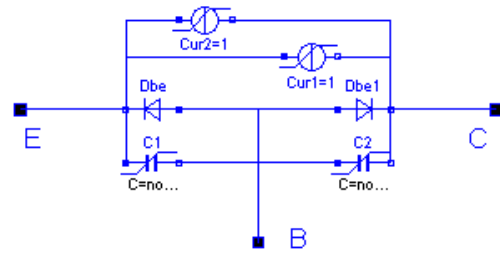


Figure 5: Ebers-Moll transport model

which are nonlinear ones. Since the currents of the nonlinear sources depend on the diode currents the transistors are modelled using a behavioural description instead of a structural one. Both the diodes and the capacitors use exponential growing functions. Because of numerical reasons these functions are linearized, if their results grow extremely.

The characteristic of an NPN transistor is shown in **fig. 6**. The collector current is growing exponentially if the base-emitter-voltage exceeds a certain value. In detail the characteristic depends on 16 parameters which are explained in the Modelica Standard Library.

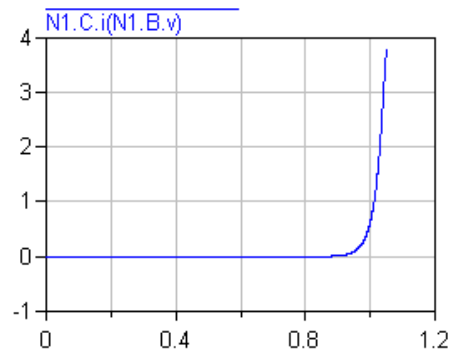


Figure 6: NPN characteristic

5 The Modelica Model

The simulation model of the controlled universal testing machine is shown in **fig. 7**. The mechanical and electronic models are from the Modelica Standard Library [1], the hydraulic models from the HyLib [7].

Unfortunately, the $\mu A741$ operates in a very small voltage range. Otherwise it runs into saturation. To avoid saturation effects, both the input signal and the output signal of the controller are transformed using the Gain model of the Modelica standard blocks library. The

Gain model simply multiplies the signal by a constant factor. The input signal is multiplied by $4.0\text{e-}7$, the output signal by 0.1.

The electronic library uses the pin definition:

```
connector Pin
  SIunits.Voltage v;
  flow SIunits.Current i;
end Pin;
```

For the block library the port definition is (the OutPort definition is quite similar):

```
connector InPort
  parameter Integer n=1;
  replaceable type SignalType=Real;
  input SignalType signal[n];
end InPort;
```

When electronics is coupled with block library elements these connector definitions hit each other. Since the voltage carries the information which is relevant for the signal processing the voltage is mapped on the signal value. This is simply done using the elements SignalVoltage, which converts an InPort signal value into an electrical voltage, and the VoltageSensor, which does it vice versa.

6 Results

With Dymola version 4.1a [15] the model of the universal testing machine was composed graphically, analyzed, translated into executable code, and simulated.

The simulations started at the quiescent state (all voltages are zero, the hydraulic pressures are equal to the environment pressure) at time zero and finished after

10 seconds in the steady state. Several simulations with parameter variations were necessary. As a result the nominal valve value and parameters of the controller could be chosen. Both the maximum excitation frequency and the maximum force reachable could be calculated. Measurements which were done afterwards at the real machine confirmed this choice of parameters. In the following pictures the behaviour of some variables is shown.

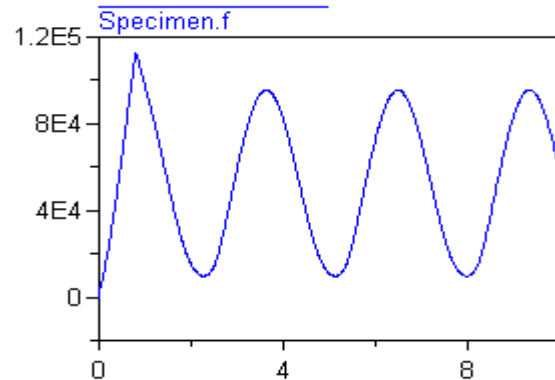


Figure 8: Force acting on the specimen

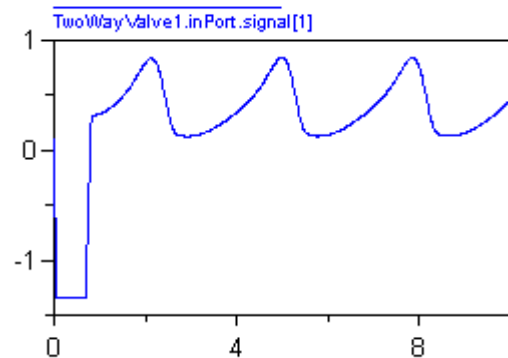


Figure 9: Valve input signal

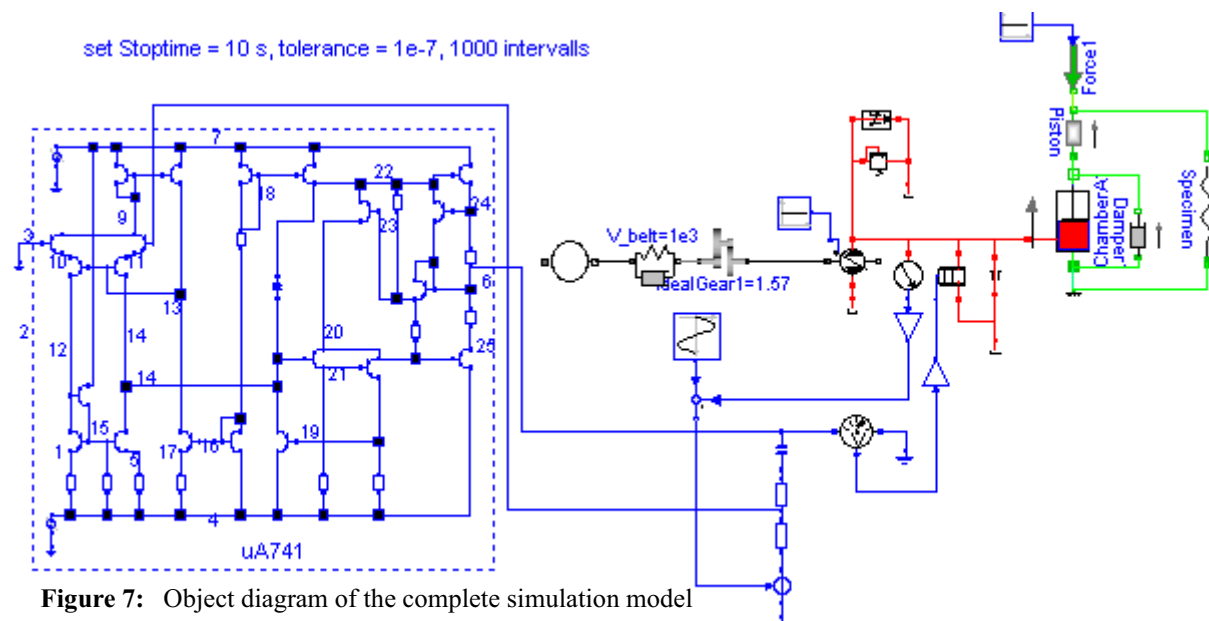


Figure 7: Object diagram of the complete simulation model

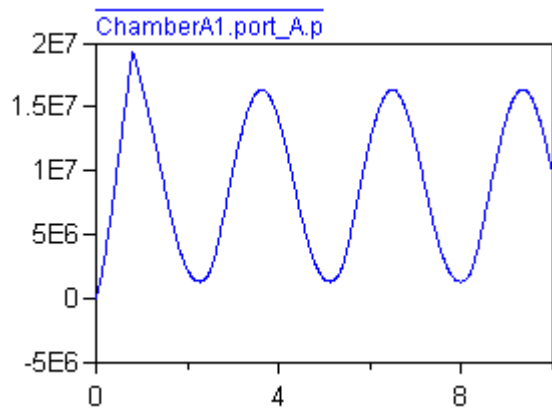


Figure 10: Pressure in the chamber

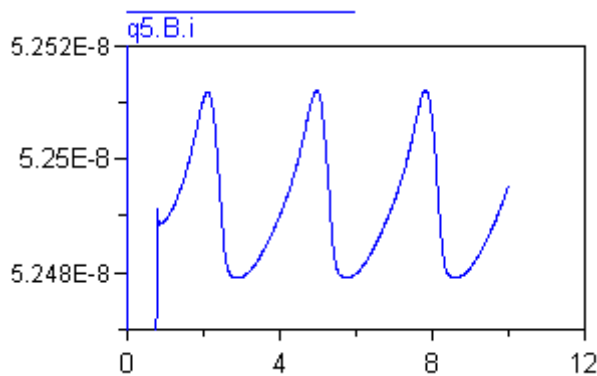


Figure 11: Base current into transistor q5

At first the Dymola tool establishes the total differential algebraic system. A symbolic calculation step reduces the number of variables/equations before the integration starts.

In the following considerations the model without electronics but with a PI-controller of the block library is used for comparisons. It will be called block model, whereas the detailed model described above will be called detailed model.

The following table compares the number of variables/equations before and after the symbolic reduction.

	Number of variables/equations	
	before reduction	after reduction
detailed model	1031	487
block model	309	137

Characteristical are the very different ranges of the variables. This is illustrated by the above shown pictures **fig. 8** to **fig. 11**.

The eigenvalues of the linearized system differ exceptionally: the smallest is about $-4.7361e+11$, the largest about $-1.9441e-5$. Therefore, the system is extremely stiff.

The CPU time needed depends on the tolerance of the numerical solver. If the tolerance is $1.e-7$ and 1000 output intervals are specified then on a Pentium III (533 MHz) it takes the translation and linking 23 s, and the simulation 232 s. Most of the simulation time is used for leaving the quiescent state. If the stop time is 20 s the CPU time needed is only 4 s higher.

Important for an effective simulation is the optimal choice of the tolerance of the numerical solver. In the following table the statistic is compared at different tolerances for a stop time of 10 seconds and 1000 output intervals, regarding the number of successful steps, the number of F-evaluations, and the number of step events:

Tolerance	Number of		
	succ. steps	F-evaluations	state events
1.0e-5	-	-	-
5.0e-6	5561	253025	104
1.0e-6	6160	215790	106
1.0e-7	9821	266447	516
2.0e-8	16774	390555	145
1.2e-8	26368	938770	1509

If the tolerance is $1.e-5$ the simulation time progress is very small. This table shows that the performance slows down if small tolerances are used. But it also slows down if tolerances are too large. Therefore, an optimal tolerance exists which is at about $1.e-6$. In contrast with this behaviour at the block model the computational work for the block model does not increase if the tolerance becomes larger.

Consequently, the CPU times depend on the tolerance chosen. If the optimal tolerance $1.e-6$ is used the CPU time of the total model is as high as the CPU time of the block model at the same tolerance. With other tolerances the CPU time of the total model is of course higher.

These results show that in multidomain examples also the difficulties of each domain come together and react together. This point of view will have to be investigated more thoroughly.

7 Conclusion

A rather complicated multidomain example could be modeled and simulated in an easy way **without** simulator coupling. Within reasonable computing times several problems of design specifications could be solved. More than thousands of variables can be handled. Both extremely stiffness and very different ranges of variables are possible.

To encourage more detailed and more easy modeling the following improvements are suggested:

- Further physical components with multidomain aspects should be offered in the Modelica standard library
- For electronic devices the support of SPICE netlists and SPICE models is necessary

To get more insight in the multidomain simulation with regard to both modeling and numerical aspects much more complex examples are desirable.

8 Acknowledgements

The Soest part of this work was supported by the Ministerium für Schule, Wissenschaft und Forschung of Nordrhein-Westfalen, Germany, under grant Interdisziplinäres Praktikum in Ingenieurstudiengängen.

The FhG part of this research was funded by the Deutsche Forschungsgemeinschaft (DFG) within the SFB 358 Automated System Design.

9 References

- [1] Modelica Language Specification 1.4. www.Modelica.org
- [2] Elmqvist, H.: A Structured Model Language for Large Continuous Systems. PhD-Thesis Lund Institute of Technology, Lund, Sweden, 1978
- [3] Larsson, M.: ObjectStab - A Modelica Library for Power System Stability Studies. Lund, Modelica Workshop 2000, 13-22
- [4] Tummescheidt, H.: Development of a Modelica Base Library for Modeling of Thermo-hydraulic Systems, Lund, Modelica Workshop 2000, 41-52
- [5] Beater, P.: Modeling and Digital Simulation of Hydraulic Systems in Design and Engineering Education using Modelica and HyLib. Lund, Modelica Workshop 2000, 33-40
- [6] Beater, P.: Über 4 Jahrzehnte Simulation in der Hydraulik. O+P Ölhydraulik und Pneumatik 43 (1999)2, 107-111
- [7] HyLib. Library of Hydraulic Components. www.Hylib.com
- [8] Beater, P.: Entwurf hydraulischer Maschinen - Modellbildung, Stabilitätsanalyse und Simulation hydrostatischer Antriebe und Steuerungen. Berlin, Heidelberg, New York, Springer-Verlag, 1999
- [9] Johnson, B.; Quarles, T.; Newton, A.R.; Pederson, D.O.; Sangiovanni-Vincentelli, A.: SPICE3 Version 3e, User's Manual., Univ. of California, Berkeley, Ca., 94720, 1991
- [10] Clauß, Chr.; Leitner, Th.; Schneider, A.; Schwarz, P.: Modelling of electronic circuits with Modelica. Lund, Modelica Workshop 2000, 3-11
- [11] Tietze, U.; Schenk, Ch.: Halbleiter-Schaltungselektronik. Berlin, Heidelberg, New York, Springer-Verlag, 1980
- [12] Herpy, M.: Analoge integrierte Schaltungen. Akadémiai Kiadó. Budapest 1976
- [13] Ebers, J.J.; Moll, J.L.: Proc. IRE 42 (1954) 1761-1772
- [14] Horneber, E.-H.: Simulation elektrischer Schaltungen auf dem Rechner. Berlin, Heidelberg, New York, Tokyo, Springer-Verlag 1985
- [15] Dymola: www.Dynasim.se

Chemical Reaction Modeling with ThermoFluid/MF and MultiFlash

Hubertus Tummescheit[†] and Jonas Eborn[‡]

[†]Department of Automatic Control
Lund University, Sweden
hubertus@control.lth.se

[‡]United Technologies Research Center
East Hartford, Connecticut, USA
EbornJP@utrc.utc.com

Abstract

The free Modelica library THERMOFLUID (see [2] and [11]) was developed for simulation of thermo-hydraulic applications, both for single-species applications like the water-steam cycle in a thermal power plant and for multi-species applications with gas mixtures. It has demonstrated its flexibility for modeling thermodynamic and process applications in a variety of industrial and academic projects, see [10], [3] and [7]. This article describes how support for chemical reactions and membrane diffusion has been added to THERMOFLUID, thus expanding the area of possible applications to include reacting flows, chemical batch reactors, catalytic converters, etc. Another crucial part of the modeling work has to be spent on getting physical property data of sufficient accuracy and with acceptable computational complexity for engineering purposes into the model. This has been addressed in the development of a commercial interface to the industry-standard physical property package MultiFlash. The new Modelica library THERMOFLUID/MF provides the modeler with two toolboxes. Firstly, a low-level Modelica function interface to MultiFlash. MultiFlash consists of a core of physical property calculation routines and a basic database of the most common chemical components and a number of add-on property databases. The interface gives access to multi-component, multi-phase property calculations including gas, several liquid and condensed phases, wax formations and hydrates. Secondly, a high-level Modelica model library which is fully integrated with the THERMOFLUID library and implements robust and efficient dynamical models for the most common process engineering equipment. In addition, reliable crossing functions for detecting phase boundaries in multi-phase, multi-component mixtures

have been implemented for the first time in a high-level modeling language. The crossing functions make it possible to simulate processes correctly even at off-design operating points and under start-up conditions. A flash volume may in such cases be filled with only liquid or only gas. Crossing functions for phase transitions ensure high performance simulation even in these cases.

1 Flexible handling of chemical reactions

In standard chemical textbooks, reactions are treated as source terms in component concentration balances:

$$\frac{dc_i}{dt} = c_i^{in} - c_i^{out} + r_i \quad (1)$$

where r_i are the component reaction rates, given by a kinetic expression. In a more general way, we can include the reaction terms in the component mass balance and total energy balance

$$\frac{dM_i}{dt} = \dot{m}_i^{in} - \dot{m}_i^{out} + rZ_i \cdot MW_i \quad (2)$$

$$\frac{dU}{dt} = \dot{q}^{in} - \dot{q}^{out} + \sum_{i=1}^{nc} rZ_i \cdot H_i^f \quad (3)$$

where rZ are reaction rates in moles/s, \dot{q} is convective heat flow, \dot{m} mass flow and H^f is component enthalpy of formation.

1.1 ThermoFluid balance equations

In THERMOFLUID, the general balance equations are implemented in the package `BaseClasses.Balances`. The basic balance equations should not be modified by the average user and thus

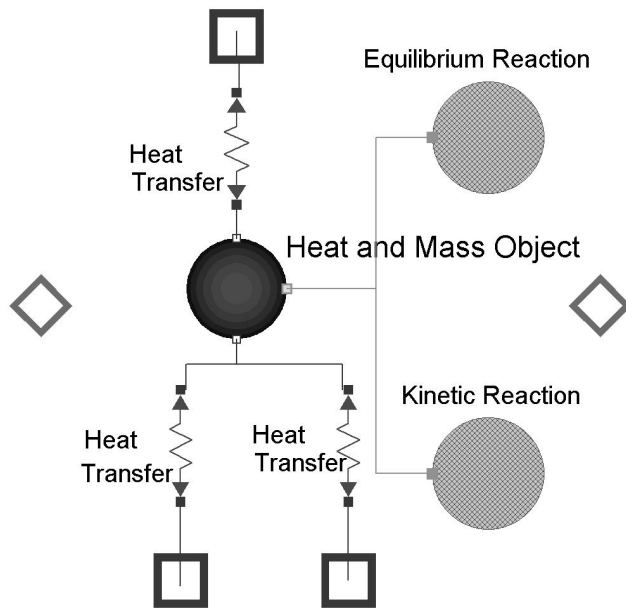


Figure 1: Schematic of the HeatAndMassObject with heat interaction and reaction objects (no diffusion connector present).

need to be general enough to handle all cases from an isolated gas volume to a reactor with added mass- and heat transfer laws. The model structure has to provide the option to add any kind of heat- and mass transfer interaction with the control volume later, as an add-on component. The basic balance equation of a control volume with two connectors is implemented as

$$\begin{aligned} dM_x &= a.mdot_x + b.mdot_x + rM; \\ dU &= a.q_{conv} + b.q_{conv} + Q_s; \end{aligned}$$

This means that rM and Q_s , corresponding to the source terms in (3) should be unspecified in the general base class, and then specified at a later stage when the balance class is reused in the model of a specific component. When there are no reactions or heat interaction with the volume there is no need for any source terms. In this case the model should provide a default value of zero production.

1.2 The HeatAndMassObject, a gateway to the balances

The contradiction of leaving the option open to specify production terms but not having to add a default value of zero can be handled with open flow connectors. In Modelica, all quantities which are flows are marked with the **flow**-prefix. Flow variables obey the zero-sum rule (Kirchhoffs' current law) and have in unconnected connectors a zero default value. Since these connectors should be internal to the volume,

they need to be attached to an object inside the volume model. This is the HeatAndMassObject, see Figure 1, which acts as a gateway between the balance equations and possible heat- and mass transfer objects. External connectors can also be connected to the HeatAndMassObject.

Interfaces

The HeatAndMassObject interact with other objects through a number of different connectors. The currently implemented connectors include the HeatFlow connector for pure heat interaction (conduction/radiation), the ChemFlow connector for chemical reactions, both kinetic and equilibrium, and a connector for membrane diffusion.

```
connector HeatFlow
  parameter Integer      n;
  Temperature [n]        T;
  flow Power [n]          q;
end HeatFlow;

connector ChemFlow
  parameter Integer      n, nc;
  parameter String       MediumType;
  Temperature [n]        T;
  Pressure [n]            p;
  Concentration [n,nc]   conc;
  flow MolarFlowRate [n,nc] rZ;
  flow Power [n]          q;
end ChemFlow
```

The diffusion connector is similar to the ChemFlow connector but has mass flow rate instead of molar flow rate since this is standard for diffusion.

The flow semantics of Modelica for the molar flow rZ and the heat flow q make sure that all contributions to the mass- and energy balances are correctly taken into account, no matter whether there are zero, one or many connections to the HeatAndMassObject, see Figure 1. Inside the HeatAndMassObject the contributions from the different connectors are summed up and transferred to the balance equations in the volume.

1.3 Objects for encapsulating reactions

To be able to drag and drop reaction models into a volume model (a reactor), they are encapsulated in reaction objects. As shown in the code example below, the Basic reaction inherits interfaces and basic parameters from the reaction BaseObject.


```

package Reactions

partial model BaseObject
  parameter Integer n, nc, nr;
  MolarReactionRate[n,nr] reacRate;
  Enthalpy[nc] compHf;
  parameter StoichiometricNumber[nr,nc]
    stoich=zeros(nr,nc);
equation
  for i in 1:n loop
    r.rZ[i,:] =
      transpose(stoich)*reacRate[i,:];
  end for;
end BaseObject;

model Basic "Simple Arrhenius reaction"
  extends BaseObject;
  parameter Rate[nr] A0;
  parameter Real[nr] b;
  parameter MolarInternalEnergy[nr] Ea;
  Concentration[n,nc] conc;
equation
  for i in 1:n loop
    reacRate[i,:] = ...;
    r.q[i] = -compHf*r.rZ[i,:];
  end for;
end Basic;

end Reactions;

```

The reaction rates are calculated from standard Arrhenius expressions using the concentrations and the parameters. To use the reaction component, like in the kinetic reaction in Figure 1, the user simply needs to specify the parameters. The stoichiometry matrix is constructed as shown in Table 1 and the heat of formation parameters are added from the medium model. To construct models of other types of reactions the reaction `BaseObject` can be reused. The customized reaction model needs to give expressions for the reaction rates, either by adding equations or by calling a rate function. In this way packages of reactions can be

	{O ₂ H ₂ H ₂ O O H OH Ar}
H + O ₂ → OH + O	$\begin{bmatrix} -1 & 0 & 0 & 1 & -1 & 1 & 0 \end{bmatrix}$
OH + O → H + O ₂	$\begin{bmatrix} 1 & 0 & 0 & -1 & 1 & -1 & 0 \end{bmatrix}$
O + H ₂ → OH + H	$\begin{bmatrix} 0 & -1 & 0 & -1 & 1 & 1 & 0 \end{bmatrix}$
H ₂ O + H → H ₂ + OH	$\begin{bmatrix} 0 & 1 & -1 & 0 & -1 & 1 & 0 \end{bmatrix}$
H ₂ + OH → H ₂ O + H	$\begin{bmatrix} 0 & -1 & 1 & 0 & 1 & -1 & 0 \end{bmatrix}$
H ₂ O + O → 2 OH	$\begin{bmatrix} 0 & 0 & -1 & -1 & 0 & 2 & 0 \end{bmatrix}$
2 H + Ar → H ₂ + Ar	$\begin{bmatrix} 0 & 1 & 0 & 0 & -2 & 0 & 0 \end{bmatrix}$
2 O + Ar → O ₂ + Ar	$\begin{bmatrix} 1 & 0 & 0 & -2 & 0 & 0 & 0 \end{bmatrix}$

Table 1: Reactions included in the H₂ O₂ reaction system and the corresponding stoichiometric matrix.

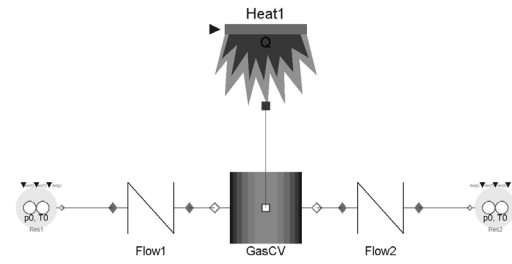


Figure 2: Schematic of example system with H₂-O₂ reaction.

built and reactions can graphically be added to standard reactor models.

1.4 Example, combustion of hydrogen

As an example, we consider the combustion of hydrogen and oxygen into water. In a simple setting, see Figure 2, the system consists of a reservoir supplying the reactants, a reactor volume and a sink for the product flow. A heat source is added to provide the heat necessary to ignite the mixture.

The complete set of sub-reactions for this process involves a large number (> 40) of very fast reactions, see [12]. Here we only consider the 8 main reactions, involving the components { O₂, H₂, H₂O, O, H, OH, Ar }. Argon is included as an inert gas. The included reactions are listed in Table 1. The corresponding stoichiometry matrix and reaction rate parameters have been coded into a `Basic` reaction object inside the `GasCV` reaction vessel.

The result plots show clearly that the reactions are extremely fast once they started. They saturate when all H₂ is burned up and the flow through the volume reaches steady state. The mass flows in Figure 3 show a violent explosion when the mixture ignites. After the

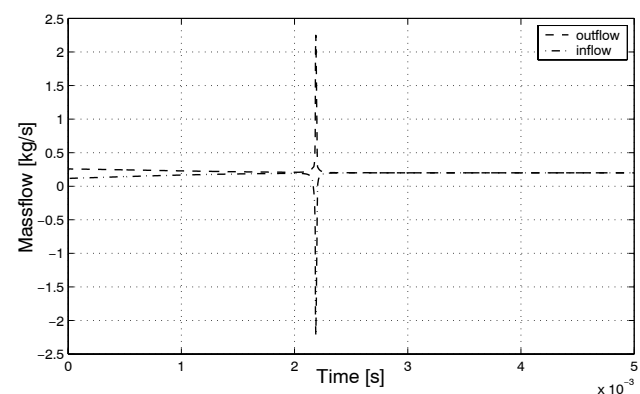


Figure 3: Mass flows into and out of the control volume during the ignition phase

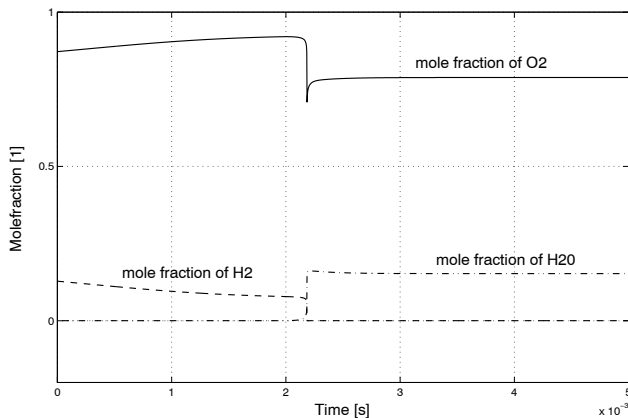


Figure 4: Molar fractions of the principal reactants and products

initial ignition, a steady inflow of premixed gases leads to a steady combustion with plenty of surplus oxygen. The speed of the reactions makes the system very stiff. The whole simulation shown in Figures 4-3 spans only a few milliseconds.

2 The MultiFlash interface

MultiFlash is the generic name for a physical property software from Infochem Ltd. It is a comprehensive system that calculates the thermophysical properties of pure substances and mixtures and carries out phase and chemical equilibrium calculations for fluid and solid phases. MultiFlash consists of several software modules: databases with the raw property data, access software to the databases and different modules for pure component property calculation, mixture models for thermodynamic properties and transport properties, handling of binary interaction parameters and phase and chemical equilibrium calculations. A number of process simulators, e. g., gPROMS from PSE, uses an interface to MultiFlash for the calculation of physical properties. For use in a dynamic simulation program typically only a small fraction of the MultiFlash functions are needed. The current interface is kept as simple as possible, with all necessary interaction with the property database encapsulated into one medium property object. The interface has been tested with both Dymola by Dynasim AB [4] and MathModelica by MathCore AB [6].

2.1 The low-level interface

The low-level interface between Modelica models and the MultiFlash modules, which are accessible via a

Win32 Dynamic Link Library (dll) under Windows, consists of the standard Modelica foreign function interface for the C-language. This means that all calls to MultiFlash routines are provided exactly as documented in the MultiFlash programmers guide, including identical variable names. There are two minor, but necessary exceptions. Modelica does not allow to overwrite inputs with outputs in the calling of functions. This is common practice in Fortran numerical routines and in MultiFlash this is exclusively used to provide estimates of the solutions in the input variables. In the Modelica interface, the estimated solutions are provided as *additional* input arguments to the function and the original MultiFlash variables are kept as outputs. The second exception is the handling of error message strings. The handling of errors and warnings is done in the C wrapper functions. Diagnostic messages are written to the simulation log. A flag with the number of errors is returned in the Modelica function call for error trapping purposes.

2.2 Computational efficiency

Simulation time is an important issue and the interface library uses all available methods to make function calls computationally efficient. A simple rule is to get as many physical properties as possible from one call to MultiFlash. All essential medium properties needed for the default dynamic model are available in one property record which is calculated with one single function call to MultiFlash. The dynamic state model and other ThermoFluid models need everything in this standard property record, so it is computationally not efficient to slim down this function call. Boolean flags to the MultiFlash routines are used to ensure that only the medium properties of interest are calculated. High level functions that return only a single property have not been implemented in order to close the door on unnecessarily slow models. However, all low-level MultiFlash functions are available and thus single function calls to obtain properties can be used if it is desired.

Providing good estimates of the solution makes a big difference in the solution time for any nonlinear system of equations, especially for phase equilibrium calculations. It is obvious that for continuous, dynamic simulation the result from the last time step usually provides such an estimate. Internal caching of the last solution in the same control volume is therefore implemented in the THERMOFLUID/MF MultiFlash interface.

2.3 Using MultiFlash with ThermoFluid/MF

Setting up a model that uses MultiFlash properties is currently done through the MultiFlash windows user interface. The user selects the wanted components by querying the available MultiFlash databases. For complex systems, the MultiFlash stream facility is used to define different component streams to be used in different parts of the system. The global stream which has to be defined first contains the union of the components in all streams. If necessary, the thermodynamic models can be changed from the default suggested by MultiFlash through the graphical user interface. All standard equations-of-state (EOS) models, e. g., Redlich-Kwong-Soave or Peng-Robinson, can be used with a selection of mixing rules. Binary interaction parameters can be entered if needed. Transport property models are selected independently of the EOS models. The problem setup is saved in an mfl-file which is then read and parsed during initialization. The file name is the main parameter to the medium models in the THERMOFLUID/MF library. Problem setup files are read from the current directory or from a repository, where all problem setup definitions can be managed in a centralized way.

2.4 Dynamic state equations

The most efficient method of combining the dynamic states and the physical property calculation is to choose the dynamic states of the model such that they are inputs to the physical property calculation routines. That avoids the solution of non-linear equation systems during simulation. Otherwise, inputs to the property functions have to be computed from outputs of that functions through a non-linear equation system. This happens when the outputs are dynamic states or time-invariant parameters, like the volume in a closed vessel. If the property functions are computationally expensive relative to the rest of the model, the saving in computation time by using a model which is explicit in the states is significant. When this can not be achieved, as is the case with the MultiFlash routines, it is still preferable to get non-linear equation systems of the lowest possible dimension. Due to the MF calling structure with pressure p , temperature T and N (mole amounts) as inputs and total volume V as output a special state model has been defined. It is incorporated in the free THERMOFLUID library and can be used interchangeably with the simple ideal gas models in the free THERMOFLUID library and the commercial MultiFlash property models. The state model uses

temperature T and mole amounts N as dynamic states, while p can be regarded as an algebraic variable that contains state information. For the standard case of a constant volume control volume, the pressure is solved for iteratively to ensure that the total volume is kept constant, $V_{fixed} = V(p)$. This is the only non-linear equation system in the model for single phase calculations. The dynamic state model is derived from the standard text-book form of an energy and mass balances. In block matrix notation, the inner energy and mole amount balance can be recast into temperature and moles as dynamic states as follows (boldface for vectors and matrices, sizes follow from dimension of N , the number of components in the mixture.):

$$\begin{pmatrix} \mathbf{N}_t \\ U_t \\ V_t \end{pmatrix} = \begin{pmatrix} \mathbf{I} & \mathbf{0} & \mathbf{0} \\ \frac{dU}{dN}|_{T,V} & \frac{dU}{dT}|_{N,V} & \frac{dU}{dV}|_{N,T} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \mathbf{N}_t \\ T_t \\ V_t \end{pmatrix} \quad (4)$$

The subscript t is used for the time derivative, N stands for mole amounts, U for total inner energy. The inverse of the jacobian is used to make this model explicit in the mole vector, the temperature and the volume as dynamic states:

$$\mathbf{J}^{-1} = \frac{1}{\frac{dU}{dT}|_{N,V}} \begin{pmatrix} \mathbf{I} & \mathbf{0} & \mathbf{0} \\ -\frac{dU}{dN}|_{T,V} & -\frac{dU}{dT}|_{N,V} & -\frac{dU}{dV}|_{N,T} \\ 0 & 0 & 1 \end{pmatrix} \quad (5)$$

The structure of the jacobian inverse reveals that only the equation for the inner energy is transformed into one for the temperature. The mole balance equations remain unchanged from (4).

The partial derivatives occurring in the transformed dynamic and initial equations can be calculated from derivatives that are returned by MultiFlash by setting the appropriate flags. However, the derivatives must be transformed using thermodynamic determinants since MultiFlash returns derivatives at constant pressure and the THERMOFLUID balances are derived at constant volume. As an example we pick the derivative of total enthalpy w.r. t. temperature (all derivatives are at constant composition):

$$\frac{\partial H}{\partial T}|_V = \frac{\partial H}{\partial T}|_p - \frac{\partial H}{\partial V}|_T = \frac{\partial H}{\partial T}|_p - \left(\frac{\partial H / \partial p|_T}{\partial V / \partial p|_T} \right) \quad (6)$$

All derivatives on the right hand side of the equation are returned by standard MultiFlash property calls.

2.5 Initialization

In the Modelica 2.0 specification and Dymola version 4.2 the possibility to separate the equations for steady state initialization from the dynamic states was introduced. Due to that separation, a control volume can now easily be initialized at steady state pressure, even when the pressure is not a dynamic state. In complex flow sheets, the calculation of an initial steady state is usually numerically much more challenging than the subsequent dynamic simulation. A helpful way around that problem is to use a suitable “pseudo steady state” instead which avoids harsh initial transients. One possibility to do so is to use steady state initialization only for the states with relatively fast eigenvalues. Setting the pressure gradient to zero, but supplying initial estimates for temperature and composition is one such suitable choice of a “pseudo steady state”. The fast modes of the system (pressure and, if a dynamic momentum balance is used, mass flows) are initialized in steady state, while the much slower modes of temperature and composition are set by a non-steady state initial guess. The pressure gradient becomes:

$$\frac{dp}{dt} = \frac{dp}{dT} \bigg|_N \frac{dT}{dt} + \sum_{i=1}^n \frac{dp}{dN_i} \bigg|_T \frac{dN_i}{dt} = 0. \quad (7)$$

This equation together with given initial composition and temperature is much easier to solve than a full steady state, especially for large networks, but the initial transients due to errors in the initial guesses are orders of magnitude smaller than the ones obtained from non steady state pressures. The new initialization method has been implemented for all state models in the THERMOFLUID and THERMOFLUID/MF libraries and has improved the handling of model initialization considerably. Before implementation of the improved initialization, computation time for small problems was dominated by the time to simulate past the initial transients. With that obstacle removed, typical simulation times for small systems are an order of magnitude faster than before.

This initialization is the default setup when the THERMOFLUID/MF high-level models are used. The initial state is defined by given temperature and mass fractions and an initial pressure estimate. The initialization then solves for the mole amount states.

2.6 Debugging

In order to improve feedback and error messages for debugging, an identifier for each control volume is allocated during the initialization of the model. The

identifier is passed to the wrapper functions calling the MultiFlash property routines. Using the unique control volume identifier, it is possible to connect error- and warning messages from the MultiFlash routines to the location in the flow sheet where the error occurred, e. g., if a temperature rises above the range of validity of the property function. All error and warning messages from MultiFlash are written to the Dymola simulation log. Information about the version, the configuration, the number and composition of streams etc. is also included in the log.

2.7 ThermoFluid/MF high level models

Modeling of process engineering problems can not be cast into fixed, unchangeable model library components as for example multibody systems. Instead flexibility is needed to have basic building blocks taking care of the standard parts of any dynamic model. These basic models need to be easy to adapt to a specific problem. A large part of the physical property calculations is identical for all modeling problems. The THERMOFLUID/MF library provides such basic models and building blocks for control volume models based on MultiFlash properties. Extensions are simple to add by using elements of the THERMOFLUID or THERMOFLUID/MF libraries. Some examples of lumped and distributed models demonstrate how to build components and larger systems from the building blocks in the library. A mixture which is typical for fuel cell reformer systems is used to demonstrate how the minimal physical property model is used and also how to add transport properties. Transport properties are not included in the THERMOFLUID library except for water, but MultiFlash includes several models for viscosity, thermal conductivity and surface tension for pure components and mixtures.

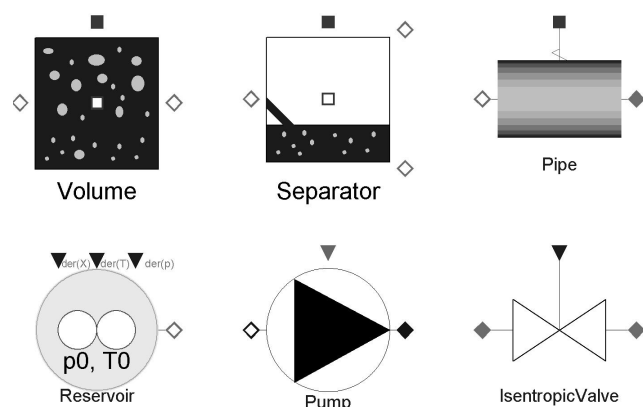


Figure 5: Example models from the library.

The number of high level models in the THERMOFLUID/MF library is fairly small, because most standard models can be used from the THERMOFLUID library. The only base models that are different are control volume models. For flash models new flash control volumes are introduced. They determine which phase is flowing in or out at a connector from the position of the flow connector and the liquid level in the volume. Tray models for distillation columns will be added later.

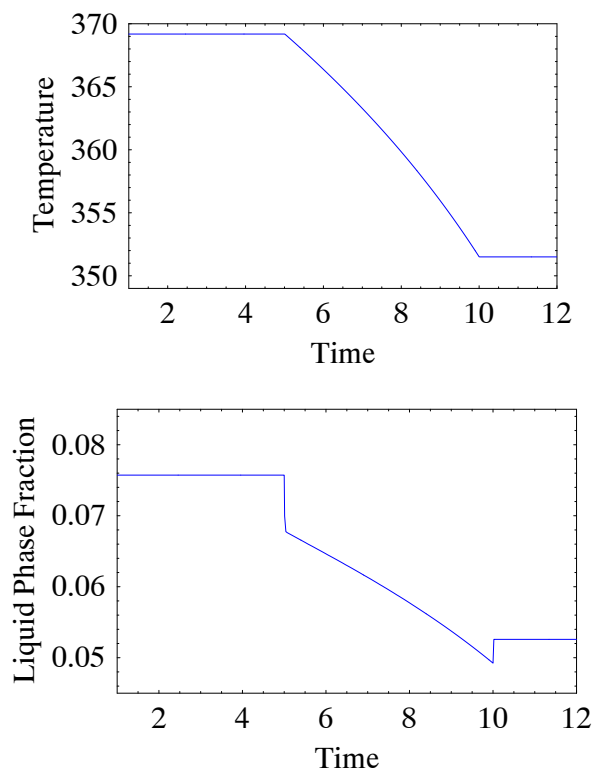


Figure 6: Change of temperature and liquid phase fraction in a water-ethanol mix during a pressure transient.

The simulation result from the depressurization of a flash volume filled with a water-ethanol mixture in thermodynamic equilibrium of the two phases is shown in Figure 6. A ramp from 1 bar to 0.5 bars is imposed on the volume which has a feed flow of constant composition. The jump in the liquid phase fraction at the start and end of the transient is due to the changing in- and outflow phase fractions. Using MultiFlash properties with the THERMOFLUID/MF library is very simple and requires only few steps of setup:

- Define the components, phases and models to be used in the MultiFlash user interface and save the result in a model setup file.

- Define a THERMOFLUID-compatible property model, following the examples in the THERMOFLUID/MF library.
- Use that property model in a suitable control volume model from the THERMOFLUID/MF library.

3 Crossing Functions for multi-component multi-phase mixtures

In Modelica, crossing functions are usually automatically generated from all equations that contain statements which indicate that a function $f(x)$ is discontinuous at a certain point x_0 . For example, in the following equation:

```
phase = if h < hliq or h > hvap or p > pcrit then 1 else 2;
```

three crossing functions are introduced to monitor the states of the boolean conditions. This is necessary because numerical integration routines assume continuity of their right-hand side functions. This assumption is violated in most if-clauses. This can not be automated for external functions that are discontinuous at a point x_0 . Thus crossing functions have to be provided by the user in order to make the simulator detect the discontinuity. These crossing functions have to be consistent with the actual discontinuities, otherwise they will not work. In the context of phase equilibrium calculations for multi-component fluid mixtures this means that a unique function of composition, pressure and temperature (N, p, T) must be returned from the phase equilibrium calculations which has a sign change at the point where a new phase is formed or one phase ceases to exist. At a phase boundary thermodynamic variables have discontinuous first derivatives or are discontinuous by itself, like the heat capacity at constant volume c_v . For a mixture with n components the crossing function is a function $\mathfrak{R}^{(n+1)} \mapsto \mathfrak{R}$. It calculates a measure for the distance to the phase boundary surface which is in \mathfrak{R}^n .

3.1 Deviation index

Collaboration with Infochem Ltd. [5] brought forth an implementation of such a function to increase efficiency and reliability of phase equilibrium calculations in dynamic simulations. It is available in the latest release of MultiFlash, version 3.1. This is the first time that a multi-phase property package has been equipped with this feature, which is indispensable for being able to reliably simulate the formation or disappearance of phases in a control volume with high

quality integrators with event detection and error control. Infochem calls the new function the *deviation index*. The calculation of the deviation index is numerically much more efficient than other possibilities to determine the number of phases at a given (N, p, T) during dynamic simulation. Geometrically, the deviation index can be interpreted as a normalized length of the normal vector from the current point in the space spanned by composition, pressure and temperature to the n -dimensional tangent hyperplane to the phase separation surface. The tangent plane is also known as *Gibbs' tangent plane*. It has been used for stability analysis in phase equilibrium calculations before, see [8], [9] and [1]. The new feature is to assign a value to the distance from the hyperplane which allows a solver using interpolation to exactly locate the point in time when the simulation trajectory in the N, p, T -space will pass through the hyperplane. At equilibrium, the Gibbs energy of the system is at a minimum. This condition may be expressed as the equality of fugacities for each component in all phases or equivalently ([1]) as

$$\ln(K_{ij}) + \ln(F_{ij}) - \ln(F_{ir_i}) = 0 \quad i = 1..n_c; \quad j = 1..n_p \quad (8)$$

where n_c is the number of components, n_p is the number of phases, K_{ij} is the K-value for component i in phase j , F_{ij} is the fugacity coefficient for component i in phase j and r_i is the reference phase for component i . The K-values are defined as

$$K_{ij} = \frac{y_{ij}}{y_{ir_i}} \quad (9)$$

where y_{ij} is the mole fraction of component i in phase j . In the vicinity of the phase split surface, the left hand side of (8) gives the value of the desired crossing function, the deviation index.

Furthermore, the function needs to reliably calculate the properties used in equation (8) of a phase which is unstable at the current (N, p, T) . Considering for simplicity single component mixtures and the calculation of thermodynamic properties for a phase which is unstable inside the 2-phase dome (i.e., superheated liquid or subcooled vapour), it becomes clear that the numerical computation is only possible to the limit of the so called *spinoidal lines*. For simple cubic EOS the spinoidal lines are defined by the connection lines of the maxima and minima of the theoretical isothermes inside the two-phase dome. An implementation thus has to guard against erroneous results far from the phase boundary.

4 Conclusions

The inclusion of reaction calculations and the interface to the physical property database MultiFlash into the THERMOFLUID library opens new possibilities of modeling process systems and combustion processes which up to now have been blocked by the large initial investment in modeling work to set up the physical property calculation.

The general reaction, diffusion and heat transfer object provides a clean and unified way of encapsulating sub-models for heat and mass transfer. Base classes never need to be changed no matter how many connections to the control volume exist. Standard reactions can be stored in component libraries and used with any reactor model that has a compatible medium property model. Membrane diffusion uses the same mechanism to couple into the standard dynamical equations.

The THERMOFLUID/MF library provides two sets of models: low level models which are one-to-one wrappers to the MultiFlash physical property routines and high level base models for multi-component liquid-gas two phase models. Care has been taken to make the time consuming VLE-calculations as efficient as possible and at the same time numerically robust.

Crossing functions for multi-phase, multi-component mixtures have been implemented in collaboration with Infochem Ltd. They allow a numerically robust detection of the formation of new phases in a multi phase mixture.

References

- [1] J. F. Counsell, R. A. S. Moorwood, and R. Szczepanski. Calculating Multiphase Equilibria. In *Proceedings of the Conference on Vapour-Liquid Equilibria*, Aston, 1990.
- [2] Jonas Eborn. *On Model Libraries for Thermo-hydraulic Applications*. PhD thesis, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, March 2001.
- [3] Rüdiger Franke. Formulation of dynamic optimization problems using modelica. In Martin Otter, Hilding Elmqvist, and Peter Fritzson, editors, *Proceedings of the International Modelica Conference 2002*. Modelica Association and DLR, March 2002.
- [4] <http://www.dynasim.se>.
- [5] <http://www.infochemuk.com>.

- [6] <http://www.mathcore.com>.
- [7] Jakob Munch Jensen and Hubertus Tummescheit. Moving Boundary Models for Dynamic Simulations of Two-Phase Flows. In Martin Otter, Hilding Elmqvist, and Peter Fritzson, editors, *Proceedings of the International Modelica Conference 2002*. Modelica Association and DLR, March 2002.
- [8] M. L. Michelsen. The isothermal flash problem. I. Stability analysis. *Fluid Phase Equilibria*, 9:1–19, 1982.
- [9] M. L. Michelsen. The isothermal flash problem. II. Phase-split calculation. *Fluid Phase Equilibria*, 9:21–40, 1982.
- [10] Torge Pfafferot and G. Schmitz. Numerische Simulation von CO₂-Kühlprozessen mit Modelica. In *DKV-Tagungsbericht 2001*, volume IV 28. Jahrgang. DKV, Stuttgart, 2001.
- [11] Hubertus Tummescheit, Jonas Eborn, and Falko Wagner. Development of a Modelica base library for modeling of thermo-hydraulic systems. In *Modelica 2000 Workshop Proceedings*, pages 41–51, Lund, October 2000. Modelica Association.
- [12] Stephen R. Turns. *An Introduction to Combustion*. McGraw Hill International Editions, 1993.

MathModelica

An Extensible Modeling and Simulation Environment with Integrated Graphics and Literate Programming

(Abridged Version*)

Peter Fritzson¹, Johan Gunnarsson², Mats Jirstrand²

1) PELAB, Programming Environment Laboratory, Department of Computer and Information
Science, Linköping University, SE-581 83, Linköping, Sweden
petfr@ida.liu.se

2) MathCore AB, Wallenbergs gata 4, SE-583 35 Linköping, Sweden
{johan,mats}@mathcore.se

Abstract

MathModelica is an integrated interactive development environment for advanced system modeling and simulation. The environment integrates Modelica-based modeling and simulation with graphic design, advanced scripting facilities, integration of program code, test cases, graphics, documentation, mathematical type setting, and symbolic formula manipulation provided via Mathematica. The user interface consists of a graphical Model Editor and Notebooks. The Model Editor is a graphical user interface in which models can be assembled using components from a number of standard libraries representing different physical domains or disciplines, such as electrical, mechanics, block-diagram and multi-body systems. Notebooks are interactive documents that combine technical computations with text, graphics, tables, code, and other elements. The accessible MathModelica internal form allows the user to extend the system with new functionality, as well as performing queries on the model representation and write scripts for automatic model generation. Furthermore, extensibility of syntax and semantics provides additional flexibility in adapting to unforeseen user needs.

1 Background

Traditionally, simulation and accompanying activities [Fritzson-92a] have been expressed using heterogeneous media and tools, with a mixture of manual and computer-supported activities:

- A simulation model is traditionally designed on paper using traditional mathematical notation.
- Simulation programs are written in a low-level programming language and stored on text files.
- Input and output data, if stored at all, are saved in proprietary formats needed for particular applications and numerical libraries.
- Documentation is written on paper or in separate files that are not integrated with the program files.
- The graphical results are printed on paper or saved using proprietary formats.

When the result of the research and experiments, such as a scientific paper, is written, the user normally gathers together input data, algorithms, output data and its

visualizations as well as notes and descriptions. One of the major problems in simulation development environments is that gathering and maintaining correct versions of all these components from various files and formats is difficult and error-prone.

Our vision of a solution to this set of problems is to provide integrated computer-supported modeling and simulation environments that enable the user to work effectively and flexibly with simulations. Users would then be able to prepare and run simulations as well as investigate simulation results. Several auxiliary activities accompany simulation experiments: requirements are specified, models are designed, documentation is associated with appropriate places in the models, input and output data as well as possible constraints on such data are documented and stored together with the simulation model. The user should be able to reproduce experimental results. Therefore input data and parts of output data as well as the experimenter's notes should be stored for future analysis.

1.1 Integrated Interactive Programming Environments

An integrated interactive modeling and simulation environment is a special case of programming environments with applications in modeling and simulation. Thus, it should fulfill the requirements both from general integrated environments and from the application area of modeling and simulation mentioned in the previous section.

The main idea of an integrated programming environment in general is that a number of programming support functions should be available within the same tool in a well-integrated way. These means that the functions should operate on the same data and program representations, exchange information when necessary, resulting in an environment that is both powerful and easy to use. An environment is interactive and incremental if it gives quick feedback, e.g. without recomputing everything from scratch, and maintains a dialogue with the user, including preserving the state of previous interactions with the user. Interactive environments are typically both more productive and more fun to use.

There are many things that one wants a programming environment to do for the programmer, particularly if it is interactive. What functionality should be included? Comprehensive software development environments are

* The complete version of the paper can be found at <http://www.mathcore.com> and <http://www.ida.liu.se/~pelab/modelica/>

expected to provide support for the major development phases, such as:

- requirements analysis,
- design,
- implementation,
- maintenance.

A programming environment can be somewhat more restrictive and need not necessarily support early phases such as requirements analysis, but it is an advantage if such facilities are also included. The main point is to provide as much computer support as possible for different aspects of software development, to free the developer from mundane tasks so that more time and effort can be spent on the essential issues. The following is a partial list of integrated programming environment facilities, some of which are already mentioned in [Sandewall-78], that should be provided for the programmer:

- Administration and configuration management of program modules and classes, and different versions of these.
- Administration and maintenance of test examples and their correct results.
- Administration and maintenance of formal or informal documentation of program parts, and automatic generation of documentation from programs.
- Support for a given programming methodology, e.g. top-down or bottom-up. For example, if a top-down approach should be encouraged, it is natural for the interactive environment to maintain successive composition steps and mutual references between those.
- Support for the interactive session. For example, previous interactions should be saved in an appropriate way so that the user can refer to previous commands or results, go back and edit those, and possibly re-execute.
- Enhanced editing support, performed by an editor that knows about the syntactic structure of the language. It is an advantage if the system allows editing of the program in different views. For example, editing of the overall system structure can be done in the graphical view, whereas editing of detailed properties can be done in the textual view.
- Cross-referencing and query facilities, to help the user understand interdependences between parts of large systems.
- Flexibility and extensibility, e.g. mechanisms to extend the syntax and semantics of the programming language representation and the functionality built into the environment.
- Accessible internal representation of programs. This is often a prerequisite to the extensibility requirement. An accessible internal representation means that there is a well-defined representation of programs that are represented in data structures of the programming language itself, so that user-written programs may inspect the structure and generate new programs. This property is also known as the principle of program-data equivalence.

1.2 Vision of Integrated Interactive Environment for Modeling and Simulation.

Our vision for the MathModelica integrated interactive environment is to fulfill essentially all the requirements for general integrated interactive environments combined with the specific needs for modeling and simulation environments, e.g.:

- Specification of requirements, expressed as documentation and/or mathematics;
- Design of the mathematical model;
- symbolic transformations of the mathematical model;
- A uniform general language for model design, mathematics, and transformations;
- Automatic generation of efficient simulation code;
- Execution of simulations;
- Evaluation and documentation of numerical experiments;
- Graphical presentation.

The design and vision of MathModelica is to a large extent based on our earlier experience in research and development of integrated incremental programming environments, e.g. the DICE system [Fritzson-83] and the ObjectMath environment [Fritzson-92b,Fritzson-95], and many years of intensive use of advanced integrated interactive environments such as the InterLisp system [Sandewall-78], [Teitelman-69,Teitelman-74], and *Mathematica* [Wolfram-88,Wolfram-97]. The InterLisp system was actually one of the first really powerful integrated environments, and still beats most current programming environments in terms of powerful facilities available to the programmer. It was also the first environment that used graphical window systems in an effective way [Teitelman77], e.g. before the Smalltalk environment [Goldberg 89] and the Macintosh window system appeared.

Mathematica is a more recently developed integrated interactive programming environment with many similarities to InterLisp, containing comprehensive programming and documentation facilities, accessible intermediate representation with program-data equivalence, graphics, and support for mathematics and computer algebra. Mathematica is more developed than InterLisp in several areas, e.g. syntax, documentation, and pattern-matching, but less developed in programming support facilities.

1.3 Mathematica and Modelica

It turns out that the Mathematica is an integrated programming environment that fulfils many of our requirements. However, it lacks object-oriented modeling and structuring facilities as well as generation of efficient simulation code needed for effective modeling and simulation of large systems. These modeling and simulation facilities are provided by the object-oriented modeling language Modelica [MA-02a,MA-02b], [Tiller-01], [Elmqvist-99], [Fritzson-98].

Our solution to the problem of a comprehensive modeling and simulation environment is to combine Mathematica and Modelica into an integrated interactive environment called MathModelica. This environment provides an internal representation of Modelica that builds on and extends the standard Mathematica representation, which makes it a well integrated with the rest of the Mathematica system.

The realization of the general goal of a uniform general language for model design, mathematics, and symbolic transformations is based on an integration of the two languages Mathematica and Modelica. Mathematica provides representation of mathematics and facilities for programming symbolic transformations, whereas Modelica provides

language elements and structuring facilities for object-oriented component based modeling, including a strong type system for efficient code and engineering safety. However, this language integration is not yet realized to its full potential in the current release of MathModelica, even though the current level of integration provides many impressive capabilities.

The current MathModelica system builds on experience from the design of the ObjectMath [Fritzson-92b, Fritzson-95] modeling language and environment, early prototypes [Fritzson-98b], [Jirstrand-99], as well as on results from object-oriented modeling languages and systems such as Dymola [Elmqvist-78, Elmqvist-96] and Omola [Mattsson-93], [Andersson-94], which together with ObjectMath and a few other object-oriented modeling languages, have provided the basis for the design of Modelica.

ObjectMath was originally designed as an object-oriented extension of Mathematica augmented with efficient code generation and a graphic class browser. The ObjectMath effort was initiated 1989 and concluded in the fall of 1996 when the Modelica Design Group was started, later renamed to Modelica Association. At that time, instead of developing a fifth version of ObjectMath, we decided to join forces with the originators of a number of other object-oriented mathematical modeling languages in creating the Modelica language, with the ambition of eventually making it an international standard. In many ways the MathModelica product can be seen as a logical successor to the ObjectMath research prototype.

2 The MathModelica Integrated Interactive Environment.

The MathModelica system consists of three major subsystems that are used during different phases of the modeling and simulation process, as depicted in

Figure 1 below:

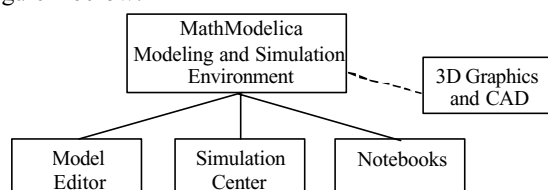


Figure 1. The MathModelica system architecture.

These subsystems are the following:

- The graphic *Model Editor* used for design of models from library components.
- The interactive *Notebook* facility, for literate programming, documentation, running simulations, scripting, graphics, and symbolic mathematics with Mathematica.
- The Simulation center, for specifying parameters, running simulations and plotting curves.

Additionally, MathModelica is loosely coupled to two optional subsystems for 3D graphics visualization and automatic translation of CAD models to Modelica. [Bunus-00], [Engelson-99], [Engelson-00]. In order to provide the best possible facilities available on the market for the user, MathModelica integrates and extends several professional software products that are included in the three subsystems. For example, the model editor is a customization and extension of the diagram and visualization tool Visio

[Visio] from Microsoft, the simulation center includes simulation algorithms from Dynasim [Elmqvist-96], and the Notebook facility includes the technical computing system Mathematica [Wolfram-97] from Wolfram Research.

A key aspect of MathModelica is that the modeling and simulation is done within an environment that also provides a variety of technical computations. This can be utilized both in a preprocessing stage in the development of models for subsystems as well as for postprocessing of simulation results such as signal processing and further analysis of simulated data.

2.1 Graphic Model Editor.

The MathModelica *Model Editor* is a graphical user interface for model diagram construction by "drag-and-drop" of model classes from the Modelica Standard Library or from user defined component libraries, visually represented as graphic icons in the editor. A screen shot of the *Model Editor* is shown in Figure 2. In the left part of the window three library packages have been opened, visually represented as overlapping windows containing graphic icons. The user can drag models from these windows (called stencils in Visio terminology) and drop them on the drawing area in the middle of the tool.

The *Model Editor* is an extension of the Microsoft Visio software for diagram design and schematics. This means that the user has access not only to a well developed and user friendly graph drawing application, but also to a vast array of professional design features to make graphical representations of developed models visually attractive. Since Modelica classes often represent physical objects it is of great value to have a sufficiently rich graphical description of these classes.

The Model Editor can be viewed as a user interface for graphical programming in Modelica. Its basic functionality consists of selection of components from libraries, connection of components in model diagrams, and entering parameter values for different components.

For large and complex models it is important to be able to intuitively navigate quickly through component hierarchies. The *Model Editor* supports such navigation in several ways. A model diagram can be browsed and zoomed. The Model Editor is well integrated with *Notebooks*. A model diagram stored in a notebook is a tree-structured graphical representation of the Modelica code of the model, which can be converted into textual form by a command.

2.2 Simulation Center.

The simulation center is a subsystem for running simulations, setting initial values and model parameters, plot results, etc. These facilities are accessible via a graphic user interface accessible through the simulation window, e.g. see Figure 3 below. However, remember that it is also possible to run simulations from the textual user interface available in the notebooks. The simulation window consists of five areas or subwindows with different functionality:

- The uppermost part of the simulation window is a control panel for starting and running simulations. It contains two fields for setting start and stop time for simulation, followed by Build, Run Simulation, Plot, and Stop buttons.
- The left subwindow in the middle section shows a tree-structure view of the model selected and compiled for simulation, including all its submodels and variables. Here, variables can be selected for plotting.

- The center subwindow is used for diagrams of plotted variables.

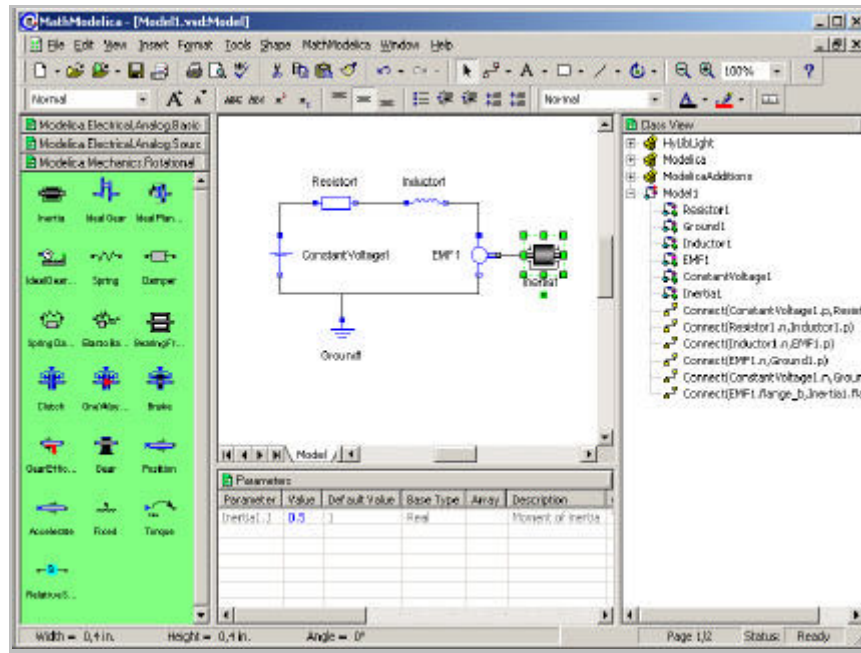


Figure 2. The Graphic Model Editor showing an electrical motor with the Inertia parameter J modified.

- The right subwindow in the middle section contains the legend for the plotted diagram, i.e. the names of the plotted variables.
- The subwindow at the bottom is divided into three sections: Parameters, Variables, and Messages, of which only one at a time is visible. The Parameters section, shown in Figure 3, allows changing parameter values, whereas the Variables section allows modifying initial (start) values, and the Message section to view possible messages from the simulation process.

If a model parameter or initial value has been changed, it is possible to rerun the simulation without rebuilding the executable code if no parameter influencing the equation structure has been changed. Such parameters are sometimes called structural parameters.

2.3 Interactive Notebooks with Literate Programming.

In addition to purely graphical programming of models using the *Model Editor* MathModelica also provides a text based programming environment for building textual models using Modelica. This is done using *Notebooks*, which are documents that may contain technical computations, text, and graphics. Hence, these documents are suitable to be used both as simulation scripting tools, model documentation and storage, model analysis and control system design, etc. In fact, this article is written as such a notebook and in the live version the examples can be run interactively. A sample notebook is shown in Figure 4.

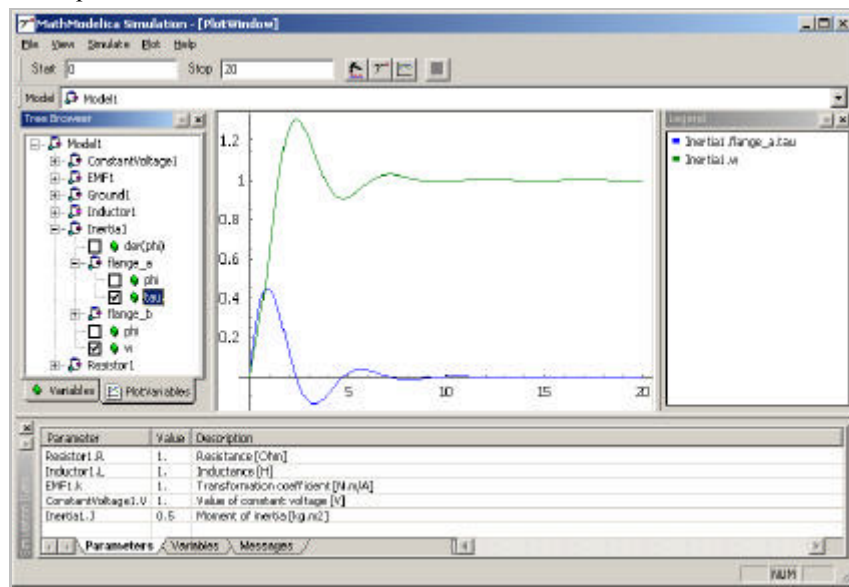


Figure 3. The Simulate window with plots of the signals `Inertia.flange_a.tau` and `Inertia.w`.

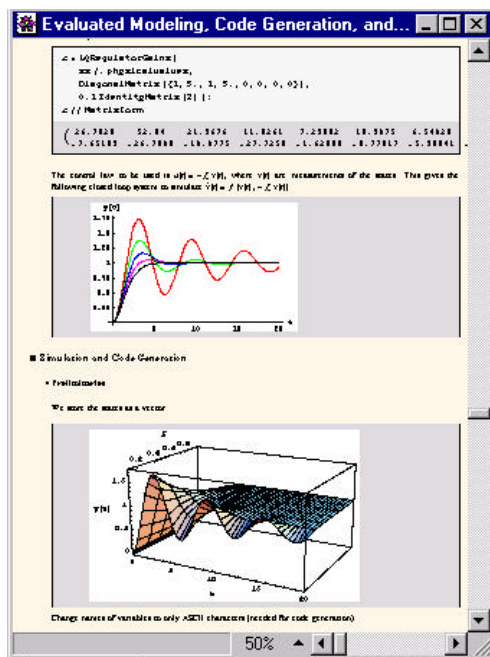


Figure 4. Examples of MathModelica notebooks..

The MathModelica *Notebook* facility is actually an interactive WYSIWYG (What-You-See-Is-What-You-Get) realization of Literate Programming, a form of programming where programs are integrated with documentation in the same document, originally proposed in [Knuth-84]. A noninteractive prototype implementations of Literate Programming in combination with the document processing system LaTeX has been realized [Knuth-94]. However, MathModelica is one of very few interactive WYSIWYG systems so far realized for Literate Programming, and to our knowledge the only one yet for Literate Programming in Modeling.

Integrating Mathematica with MathModelica does not only give access to the Notebook interface but also to thousands of available functions and many application packages, as well as the ability of communicating with other programs and import and export of different data formats. These capabilities make MathModelica more of a complete workbench for the innovative engineer than just a modeling and simulation tool. Once a model has been developed there is often a need for further analysis such as linearization, sensitivity analysis, transfer functions computations, control system design, parametric studies, Monte Carlo simulations, etc.

In fact, the combination of the ability of making user defined libraries of reusable components in Modelica and the *Notebook* concept of living technical documents provides an integrated approach to model and documentation management for the evolution of models of large systems

2.3.1 Tree Structured Hierarchical Document Representation.

Traditional documents, e.g. books and reports, essentially always have a hierarchical structure. They are divided into sections, subsections, paragraphs, etc. Both the document itself and its sections usually have headings as labels for easier navigation. This kind of structure is also reflected in MathModelica notebooks. Every notebook corresponds to one document (one file) and contains a tree structure of cells. A cell can have different kinds of contents, and can even

contain other cells. The notebook hierarchy of cells thus reflects the hierarchy of sections and subsections in a traditional document.



Figure 5. The package Mypackage in a notebook

In the MathModelica system, Modelica packages including documentation and test cases are primarily stored as notebooks, e.g. as in Figure 4. Those cells that contain Modelica model classes intended to be used from other models, e.g. library components or certain application models, should be marked as exports cells. This means that when the notebook is saved, such cells are automatically exported into a Modelica package file in the standard Modelica textual representation (.mo file) that can be processed by any Modelica compiler and imported into other models. For example, when saving the notebook MyPackage.nb of Figure 5, a file MyPackage.mo would be created with the following contents:

```
package MyPackage
  model class3
  ...
end class3;
  model class2 ...
  model class1 ...
  package MySubPackage
    model class1
    ...
  end class1;
end MySubPackage;
end MyPackage;
```

2.3.2 Program Cells, Documentation Cells, and Graphic Cells.

A notebook cell can include other cells and/or arbitrary text or graphics. In particular a cell can include a code fragment or a graph with computational results.

The contents of cells can for example be one of the following forms:

- Model classes and parts of models, i.e. formal descriptions that can be used for verification, compilation and execution of simulation models.
- *Mathematical* formulas in the traditional mathematical two dimensional syntax.
- Text/documentation, e.g. used as comments to executable formal model specifications.

- Dialogue forms for specification and modification of input data.
- Result tables. The results can be automatically represented in (live) tables, which can even be automatically updated after recomputation.
- Graphical result representation, e.g. with 2D vector and raster graphics as well as 3D vector and surface graphics.
- 2D structure graphs, that for example are used for various model structure visualizations such as connection diagrams and data structure diagrams.

A number of examples of these different forms of cells are available throughout this paper.

2.3.3 Mathematics with 2D-syntax, Greek letters, and Equations

MathModelica uses the syntactic facilities of Mathematica to allow writing formulas in the standard mathematical notation well-known, e.g. from textbooks in mathematics and physics. Certain parts of the Mathematica language syntax are however a bit unusual compared to many common programming languages. The reason for this design choice is to make it possible to use traditional mathematical syntax. The following three syntactic features are unusual:

- Implied multiplication is allowed, i.e. a space between two expressions, e.g. x and $f(x)$, means multiplication just as in mathematics. A multiplication operator $*$ can be used if desired, but is optional.
- Square brackets are used around the arguments at function calls. Round parentheses are only used for grouping of expressions. The exception is Traditional Form, see below.
- Support for two-dimensional mathematical syntactic notation such as integrals, division bars, square roots, matrices, etc.

The reason for the unusual choice of square brackets around function arguments is that the implied multiplication makes the interpretation of round parenthesis ambiguous. For example, $f(x+1)$ can be interpreted either as a function call to f with the argument $x+1$, or f multiplied by $(x+1)$. The integral in the cell below contains examples of both implied multiplication and two-dimensional integral syntax. The cell style is called MathModelica input form (called standard form in Mathematica) and is used for mathematics and Modelica code in Mathematica syntax:

$$\int \frac{x f[x]}{1 + x^2 + x^3} dx$$

There is also a purely textual input form using a linear sequence of characters. This is for example used for entering Modelica models in the standard Modelica syntax, and is currently the only cell format in MathModelica that can interpret standard Modelica syntax. However, all mathematics can also be represented in this syntax. The above example in this textual format appears as follows:

```
Integrate[(x*f[x])/(1 + x^2 + x^3), x]
```

Finally, there is also a cell format called traditionalform which is very close to traditional mathematical syntax, avoiding the square brackets. The above-mentioned syntactic ambiguities can be avoided if the formula is first entered using one of the above input forms, and then converted to traditional form.

$$\int \frac{x f(x)}{x^3 + x^2 + 1} dx$$

The MathModelica environment allows easy conversion between these forms using keyboard or menu commands. Below we show a small example of a Modelica model class SimpleDAE represented in the Mathematica style syntax of Modelica that allows greek characters and two dimensional syntax. The apostrophe (') is used for the derivatives just as in traditional mathematics, corresponding to the Modelica `der()` operator.

```
Model[SimpleDAE,
Real β1;
Real x2;
Equation[
  
$$\frac{\beta_1'}{1 + (\beta_1')^2} + \frac{\sin[x_2']}{1 + (\beta_1')^2} + \beta_1 x_2 + \beta_1 = 1;$$

  
$$\sin[\beta_1'] - \frac{x_2'}{1 + (\beta_1')^2} - 2 \beta_1 x_2 + \beta_1 = 0;$$

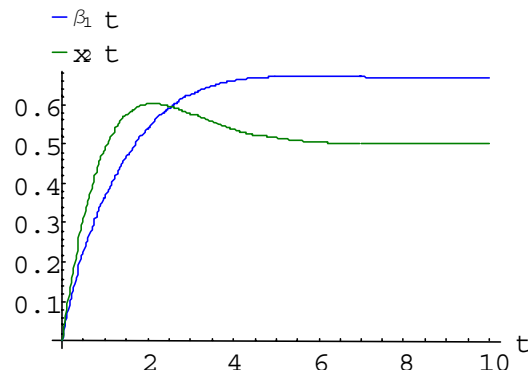
]]
```

We simulate the model for ten seconds by giving a `Simulate` command:

```
Simulate[SimpleDAE, {t, 0, 10}];
```

We use the command `PlotSimulation` for plotting the solutions for the two state variables, which of course both are functions of time, here denoted by t in Mathematica syntax:

```
PlotSimulation[{β1[t], x2[t]}, {t, 0, 10}];
```



2.4 Environment and Language Extensibility

Programming environments need to be flexible to adapt to changing user needs. Without flexibility, a programming tool will become too hard to use for practical needs, and stopped to be used. Adaptability and flexibility is especially important for integrated environments, since they need to interact with a number of external tools and data formats, contain many different functions, and usually need to add new ones.

There are two major ways to extend a programming environment

- Extension of functionality, e.g. through user-defined commands, user-extensible menus, and a scripting languages for programmability.
- Extension of language and notation, e.g. by facilities to add new syntactic constructs and new notation, or extend the meaning of existing ones.

Mathematica has been designed from the start to be an inherently extensible environment, which is what is used in MathModelica. Almost anything can be redefined, extended, or added.

2.4.1 Scripting for Extension of Functionality

An interactive scripting language is a common way of providing extensibility of flexibility in functionality. The MathModelica environment primarily uses the Mathematica language and its interpreter as a scripting language, as can be seen from a number of examples in this paper. Another possibility would be to use the Modelica language itself as a scripting language, e.g. by providing an interpreter for the algorithmic and expression parts of the language. This can easily be realized in MathModelica since the intermediate form has been designed to be compatible with Mathematica, and we already have Modelica input cells: just use Modelica input cells also for commands, which are sent to the Mathematica interpreter instead of the simulator.

2.4.2 Extensible Syntax and Semantics

As was already apparent in the section on mathematical syntax, MathModelica provides a Mathematica-like input syntax for Modelica in addition to the usual Modelica syntax. One reason is to give support for mathematical notation, as explained previously. Another reason is to provide user extensible syntax.

This is easy since syntactic constructs in Mathematica apart from the operators use a simple prefix syntax: a keyword followed by square brackets surrounding the contents of the construct, i.e. the same syntax as for function calls. If there is a need to add a new construct no changes are needed in the parser, and no reserved words need to be added. Just define a Mathematica function to do the desired symbolic or numeric processing.

The other major class of syntactic constructs are operators. There are special facilities in Mathematica to add new operators by defining their priority, operator syntax, and internal representation. It is also possible to extend the meaning of existing operators like +, *, -, etc.

2.4.3 Mathematica vs Modelica syntax.

In order to show the difference between the standard Modelica textual syntax and the extensible Mathematica-like syntax, we first show a simple model in a Modelica-style input cell:

```
model secondordersystem
  Real x(start=0);
  Real xdot(start=0);
  parameter Real a=1;
equation
  xdot=der(x);
  der(xdot)+a*der(x)+x=1;
end secondordersystem;
```

The same model in the Mathematica-like Modelica syntax appears below. Note the use of the simple prefix syntax: a keyword followed by square brackets surrounding the contents of the construct. All reserved words, predefined functions, and types in MathModelica start with an upper-case letter just as in Mathematica. Equation equality is represented by the == operators since = is the assignment operator in Mathematica. The derivative operator is the mathematical apostrophe (') notation rather than der(). The

semicolon (;) is a sequencing operator to group more than one declaration, statement, or expression together.

```
Model[secondordersystem,
  Real x[{Start == 0}];
  Real xdot[{Start == 0}];
  Parameter Real a == 1;
Equation[
  xdot == x';
  xdot' + a*x' + x == 1
]
]
```

3 Application Examples

This section gives a number of application examples of the use of the Mathmodelica environment. The intent is to demonstrate the power of integration and interactivity - the interplay between the object-oriented modeling and simulation capabilities of Modelica integrated with the powerful scripting facilities of Mathematica within MathModelica. This includes the representation of simulation results as 1D and 2D interpolating functions of time being combined with arithmetic operations and functions in expressions, advanced plotting facilities, and computational capabilities such as design optimization, fourier analysis, and solution of time-dependent PDEs. For the PDEs see the long version of the paper.

3.1 Advanced Plotting and Interpolating Functions

This section illustrates the flexible usage of simulation results represented as interpolating functions, both for further computations that may include simulation results in expressions, and for both simple and advanced plotting. The simple bouncing ball model below from [MA-02a] is used in the simulation and plotting examples.

3.1.1 Interpolating Function Representation of Simulation Results

The following simulation of the above BouncingBall model is done for a short time period using very few points:

```
res1=Simulate[BouncingBall,{t,0,0.5},
  NumberOfIntervals->10]
```

```
<SimulationData: BouncingBall: 2002-2-26
10:48:10 : {0., 0.5} : 15 data points : 1
events : 7 variables>
{c, g, height, radius, velocity, height'
velocity'}
```

The results returned by Simulate are represented by an access descriptor or handle. Some of the contents of such descriptor is shown as the result of the above call to Simulate. At this stage the simulation data is stored on disk and referenced by res1 which acts as a handle to the simulation data. When one of the variables from the last simulation is referenced, e.g. height, radius, etc., the data for that variable is loaded into the system in an load-by-need manner, and represented as an InterPolatingFunction.

3.1.2 PlotSimulation

First we simulate the bouncing ball for eight seconds and store the results in the variable `res1` for subsequent use in the plotting examples.

```
res1=Simulate[BouncingBall,{t,0,8}];
```

The command `PlotSimulation` is used for simple standard plots. If nothing else is specified, i.e. by the optional `SimulationResult` parameter, the command refers to the results from the last simulation.

Plotting several arbitrary functions can be done using a list of function expressions instead of a single expression:

```
PlotSimulation[{height[t] +  $\sqrt{3}$ ,  
Abs[velocity[t]]}, {t, 0, 8}];
```

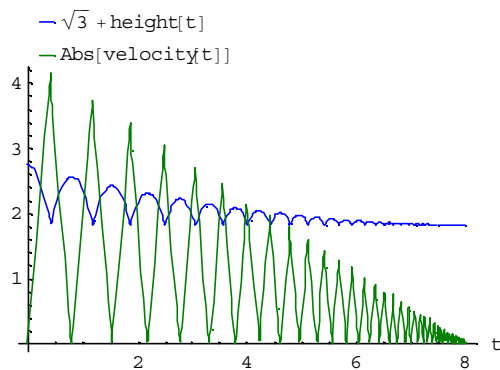


Figure 6. Plotting arbitrary functions in the same diagram.

3.1.3 ParametricPlotSimulation

Parametric plots can be done using `ParametricPlotSimulation`.

```
ParametricPlotSimulation[  
{height[t], velocity[t]},  
{t, 0, 8}];
```

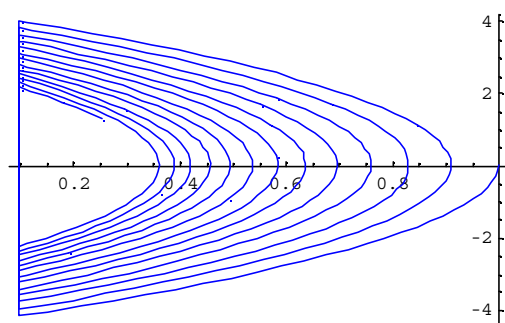


Figure 7. A parametric plot.

3.1.4 ParametricPlotSimulation3D

In this example we are going to use the Rossler attractor to show the `ParametricPlotSimulation3D` command. The Rossler attractor is named after Otto Rossler from his work in chemical kinetics. The system is described by three coupled non-linear differential equations:

$$\begin{aligned}\frac{dx}{dt} &= -y - x \\ \frac{dy}{dt} &= x + \alpha y \\ \frac{dz}{dt} &= \beta + (x - ?)z\end{aligned}$$

Here α, β and $?$ are constants. The attractor never forms limit circles nor does it ever reach a steady state. The model is shown in Mathematica syntax, enabling the use of greek characters:

```
Model[Rossler, "Rossler attractor",  
Parameter Real  $\alpha$  == 0.2;  
Parameter Real  $\beta$  == 0.2;  
Parameter Real  $\gamma$  == 8;  
Real x[{Start == 1}];  
Real y[{Start == 3}];  
Real z[{Start == 0}];  
Equation[  
x' == -y - z;  
y' == x +  $\alpha$  y;  
z' ==  $\beta$  + x z -  $\gamma$  z  
]  
]
```

The model is simulated using different initial values. Changing these can considerably influence the appearance of the attractor.

```
Simulate[Rossler, {t, 0, 40},  
InitialValues -> {x == 2, y == 2.5, z == 0},  
NumberOfIntervals -> 1000];
```

The Rossler attractor is easy to plot using `ParametricPlotSimulation3D`:

```
ParametricPlotSimulation3D[  
{x[t], y[t], z[t]},  
{t, 0, 40},  
AxesLabel -> {X, Y, Z}];
```

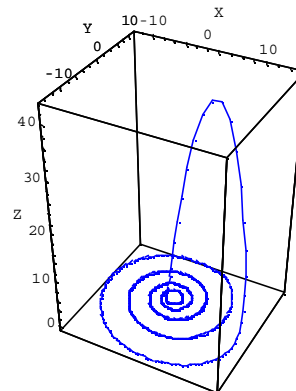


Figure 8. 3-D parametric plot of curve with many data points from the Rossler attractor simulation.

3.2 Design Optimization

This is an example of how the powerful scripting language of MathModelica can be utilized to solve non-trivial optimization problems that contain dynamic simulations.

First we will define a Modelica model of a linear actuator with spring damped stopping and then a first order system. Using MathModelica scripting we will then find a damping for the translational spring-damper such that the step response is as "close" as possible to the step response from a first order system.

Consider the following model of a linear actuator with a spring damped connection to an anchoring point:

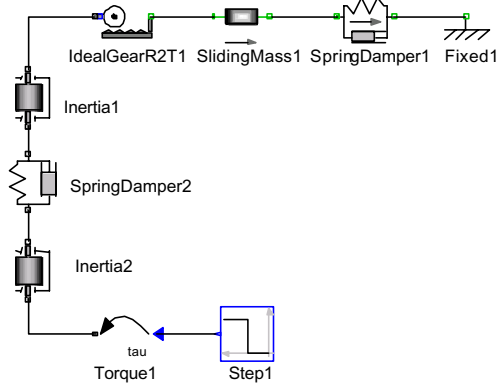


Figure 9. A LinearActuator model containing a spring damped connection to an anchoring point.

Assume that we have some freedom in choosing the damping in the translational spring-damper. A number of simulation runs show what kind of behavior we have for different values of the dampingparameter d . The Mathematica `Table[]` function is used in `Simulate[]` to collect the results into an array `res`. This array then contains the results from simulations of `LinearActuator` with a damping of 2 to 14 with a step size of 2, i.e. seven simulations are performed.

```
res = Table[Simulate[LinearActuator,
  {t, 0, 4},
  ParameterValues →
    {SpringDamper1.d == s}],
  {s, 2, 15, 2}];
PlotSimulation[SlidingMass1.s[t],
  {t, 0, 4},
  SimulationResult → res,
  Legend → False];
```

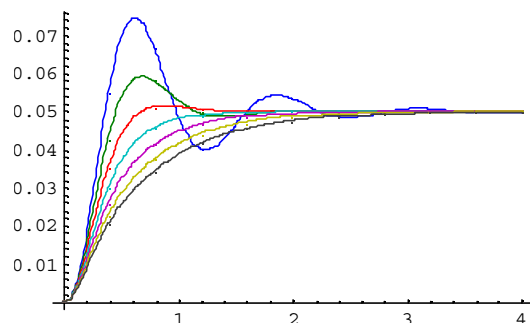


Figure 10. Plots of step responses from seven simulations of the linear actuator with different damping coefficients.

Now assume that we would like to choose the damping d so that the resulting system behaves as closely as possible to a certain first order system response.,

We simulate for different values of d and interpolate the result

```
f_pre = Interpolation[res2];
Plot[f_pre[a], {a, 2, 10}];
```

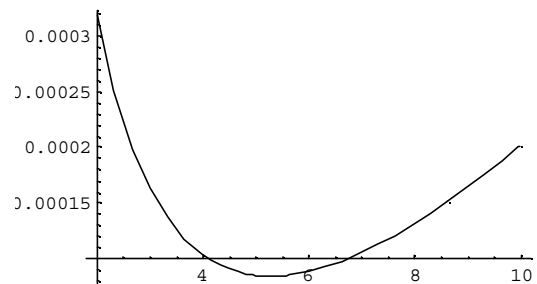


Figure 11. Plot of the error function for finding a minimum deviation from the desired step response.

The minimizing value of a can be computed using `FindMinimum`:

```
FindMinimum[f_pre[s], {s, 4}]
{0.0000832564, {s → 5.28642}}
```

3.3 Fourier Analysis of Simulation Data

Consider a weak axis excited by a torque pulse train. The axis is modeled by three segments joined by two torsion springs. The following diagram is imported from the MathModelica *Model Editor* where the model was defined.

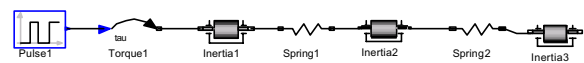


Figure 12. A WeakAxis model excited by a torque pulse train.

We simulate the model during 200 seconds:

```
Simulate[WeakAxis, {t, 0, 200}];
```

The plot of the angular velocity of the rightmost axis segment appears as follows:

```
PlotSimulation[{Inertia3.w[t],
  Torque1.τ[t]}, {t, 0, 200}];
```

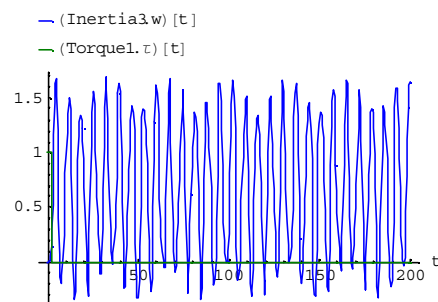


Figure 13. Plot of the angular velocity of the rightmost axis segment of the WeakAxis model.

Now, let us sample the interpolated function `Inertia3.w` using a sample frequency of 4Hz, and put the result into an array using the Mathematica `Table` array constructor:

```
data1 = Table[Inertia3.w[t],
  {t, 0, 200, .25}];
```

Compute the absolute values of the discrete Fourier transform of data1 with the mean removed:

```
fdatal = Abs[Fourier[data1 -
  MeanValue[data1]]];
```

Plot the 80 first points of the data.

```
ListPlot[fdatal[[Range[80]]],
  PlotStyle -> {Red, PointSize[0.015]}];
```

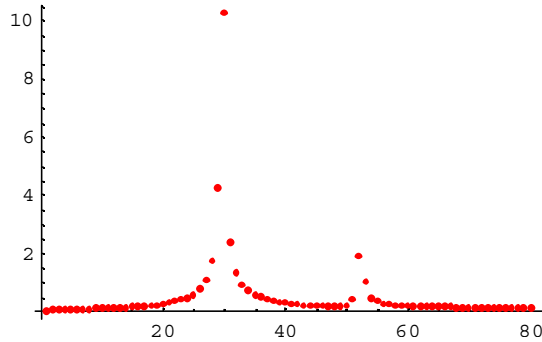


Figure 14. Plot of the data points of the Fourier transformed angular velocity.

It can be shown that the frequencies of the eigenmodes of the system is given by the imaginary parts of the eigenvalues of the following matrix (c_1 and c_2 are the spring constants)

$$\frac{1}{2\pi} \text{Eigenvalues} \left[\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ -c_1 & 0 & -c_1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ -c_1 & 0 & -c_1 - c_2 & 0 & -c_2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & -c_2 & 0 & -c_2 & 0 \end{pmatrix} \right] / .$$

$\{c_1 \rightarrow 0.7, c_2 \rightarrow 1\}$ // Chop

```
{0.256077 i, -0.256077 i,
 0.143343 i, -0.143343 i, 0, 0}
```

These values, 0.256077, 0.143344, fit very well with the peaks in the above diagram.

4 Using the Symbolic Internal Representation

In order to satisfy the requirement of a well integrated environment and language, the new MathModelica internal representation was designed with a Mathematica compatible version of the syntax. Note that the Mathematica version of the syntax has the same internal abstract syntax tree representation and the same semantics as Modelica, but different concrete syntax. Which syntax to use, the standard Modelica textual syntax, or the Mathematica-style syntax for Modelica is however largely a matter of taste.

The fact that the Modelica abstract syntax tree representation is compatible with the Mathematica standard representation means that a number of symbolic operations such as simplifying model equations, performing Laplace transformations, and performing queries on code as well as automatically constructing new code is available to the user. The capability of automatically generating new code is especially useful in the area of model diagnosis, where there is often a need for generating a number of erroneous models for diagnosis based on corresponding fault scenarios.

4.1 Mathematica Compatible Internal Form

An inherent property of Mathematica is that models or code is normally not written as free formatted text. Instead, Mathematica expressions (also called terms) are used, internally represented as abstract syntax trees. These can be conveniently written in a tree-like prefix form, or entered using standard mathematical notation. Every term is a number, an identifier, or a form such as:

$$\text{head}[term_1, \dots, term_n]$$

For example, an expression: $a+b$ is represented as `Plus[a,b]` in prefix form, also called `FullForm` syntax. A while loop is represented as the term `While[test,body]`.

In order to satisfy the requirement of a well integrated environment, we designed the new MathModelica internal representation with a Mathematica compatible version of the syntax. Note that MathModelica has the same abstract syntax trees and the same semantics as Modelica, but different concrete syntax. This means that essentially the same language constructs are written differently, as illustrated below.

The Mathematica language syntax uses some special operators, see below, and arbitrary arithmetic expressions composed from terms.

$term_1; \dots; term_n$ //sequencing operator

$\{term_1; \dots; term_n\}$ //array/list constructor

$term_1 term_2$ //Implied multiplication by space
instead of *

$term_1 == term_2$ //Equation equality

Internally the MathModelica system uses the `MathModelicaFullForm` format. This format is the abstract syntax of the MathModelica language where all the elements of the language have been defined to be easy to extract and compare for the functions operating on the MathModelica language representation, as well as achieving a high degree of compatibility with both Modelica and Mathematica.

The following is a simple constant declaration:

```
model Arr
  constant Real
    unitarr[2,2] = {{1,0},{0,1}}
    "2D Identity";
end Arr;
```

This definition is stored internally in the `MathModelicaFullForm` format which can be retrieved by calling the function `GetDefinition` which returns the internal abstract syntax tree representation of the model:

```
ff2 = GetDefinition[Arr,
  Format -> MathModelicaFullForm]
```

The tree is wrapped into the node `Hold[]` to prevent symbolic evaluation of the model representation while we are manipulating it. All nodes are shown in prefix form excepts the array/list nodes shown as `{...}` instead of the prefix form `List[...]` for arrays.

```

Hold[SetType[Arr,
  TYPE[Model[Declaration
    [TYPE[Real, {2, 2}, {Constant}, {}],
    VariableComponent[unitarr,
    ValueBinding[{{1, 0}, {0, 1}},
    {}, {}, Null]
  ];
  "2D Identity"
], {}, {}, {}
], {}, Null, Null
]
]

```

A declaration of a variable such as `unitarr` is represented by the `Declaration` node in the abstract syntax. This node has two arguments: the type and the variable instance. The type is represented by the `TYPE` node which stores the name, array dimension, type attributes (`Constant`) and type modifications (which is empty in this case). The instance argument contains a `VariableComponent` including the name of the variable, the initialization (`ValueBinding`), at the end the comment string that is associated with the variable.

There are several goals behind the design of the `MathModelicaFullForm` format, which are fulfilled in the current system:

- *Abstract syntax.* The format systematically sorts out the different constructs in the language making the navigation of types and code easier.
- Preserving the syntactic structure of both Modelica and Mathematica code. This means that the mapping from Modelica to MathModelica-FullForm format should be injective, e.g. the source code can be recreated from the intermediate form, and that transformations from Modelica via MathModelicaFullForm into Mathematica style Modelica form should be reversible.
- *Explicit semantic structure.* The format has reserved fixed attribute positions for certain kinds of semantic information, to simplify semantic analysis and queries. There is also a *canonical subset* of the format which is even simpler for semantic analysis, but does not always recreate exactly the same source code since the same declaration often can be stated in several ways.
- *Symbol table and type representation* format. The MathModelicaFullForm format should be possible to use in the symbol table, e.g. to represent types. Types are represented by anonymous type expressions such as the `TYPE` node in the above example. Anonymous means that the type representation is separate from the entity having the type.
- *Internal standard.*

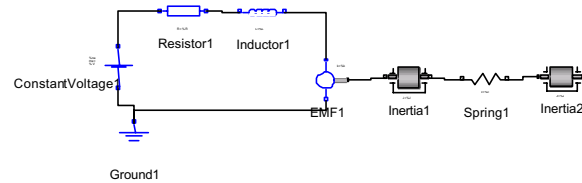
The MathModelicaFullForm format should be used by all the components in the MathModelica system.

4.2 Extracting and Simplifying Model Equations

This section will illustrate a few user-accessible symbolic operations on equations, such as obtaining the system of equations and the set of variables from a Modelica model, and symbolically simplifying this system of equations with the intention of performing symbolic Laplace transformation.

4.2.1 Definition and Simulation of Model1

The example class `Model1` has been drawn in the graphic model editor and imported into the notebook below:



Fi

Figure 15. Connection diagram of Model1.

We simulate the model, smooth the result, and make a plot.

```

res0 = Simulate[Model1, {t, 0, 25},
  ParameterValues -> {Resistor1.R == 0.9}];
res1 = SmoothInterpolation[res0];

```

The plot is parametric where we plot the `Resistor1.i` current against its derivative for both the original result and the smoothed version:

```

ParametricPlotSimulation[
  {(Resistor1.i)[t],
  (Resistor1.i)'[t]}, {t, 0, 25},
  SimulationResult -> {res0, res1}];

```

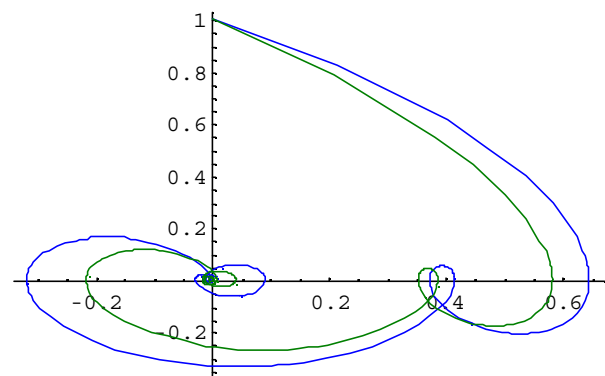


Figure 16. Parametric plots of the Resistor1 current against its derivative, both original and smoothed.

4.2.2 Some Symbolic Computations

Now, flatten `Model1` and extract the model equations and the model variables as lists, and compute the lengths of these lists:

```

eqn = GetFlatEquations[Model1];
Length[eqn]
48

Length[GetFlatVariables[Model1]]
49

```

There is one equation less than the number of variables. Therefore, add an equation for zero torque on the right flange to the equation system:

```

eqn = Append[eqn,
  Inertia2.flange.b.tau == 0];

```

We would like to simplify the equations by eliminating the connector variables before further symbolic processing. First obtain the connector variables from the flattened model:

```

connvars = GetFlatConnectionVariables
           [Model1]

{Resistor1.p.v, Resistor1.p.i,
 Resistor1.n.v, Resistor1.n.i,
 ..... ,
 Inertia2.flange[] a.tau}

Use the Eliminate function for symbolic elimination of
some variables from the system of equations.

eqn2 = Eliminate[eqn, connvars]

der[Inertia1.phi] == Inertia1.w &
der[Inertia1.w] == Inertia1.a &&
... ..
Inertia2.flange[] b.tau == 0 &
der(-1)[EMF1.w] == Inertia2.phi -
Spring1.phi[] rel

```

4.3 Symbolic Laplace Transformation.

We would now like to perform a Laplace transformation of the symbolic equation system obtained in the previous section. This can be done by the application of two transformation rules: $der^{(-1)}[a_]\rightarrow \frac{a}{s}$, $der[b_]\rightarrow sb$.

Note that $der^{(-1)}$ is the inverse of taking a derivative, i.e. an integration operation. Note also that the second rule contains an implied multiplication.

```

eq3 = eqn2 /. {der(-1)[a_] -> a/s, der[b_] -> s b}

s (Inertia1.phi) == Inertia1.w &
s (Inertia1.w) == Inertia1.a &&
.....
EMF1.w
s == Inertia2.phi - Spring1.phi[] rel

```

Introduce short names for the model parameter to obtain a more concise symbolic notation:

```

shortnames =
{Resistor1.R -> R, Inductor1.L -> L,
 EMF1.k -> k, Inertia1.J -> J1,
 Spring1.c -> c1, Spring1.phi[] rel0 -> 0,
 Inertia2.J -> J2};

```

Derive the relation between Inertia2.w and the input voltage

```

eq4 =
Eliminate[eq3,
Complement[
GetFlatNonConnectionVariables[Model1],
{Inertia2.w}]] /. shortnames

(k c1 (ConstantVoltage1.V) ==
k2 c1 (Inertia2.w) +
.....
R s3 J1 J2 (Inertia2.w) +
L s4 J1 J2 (Inertia2.w)) && s != 0

```

The transfer function H is obtained by symbolically solving for Inertia2.w in the equation system eq4, and using the obtained solution on a form Inertia2.w -> expr to eliminate Inertia2.w, thus obtaining H:

$$H[s_] = \text{First}\left[\frac{\text{Inertia2.w}}{\text{ConstantVoltage1.V}} \text{ /. } \text{Solve}[\text{eq4}, \text{Inertia2.w}]\right]$$

$$(k c_1) / (k^2 c_1 + R s c_1 J_1 + L s^2 c_1 J_1 + k^2 s^2 J_2 + R s c_1 J_2 + L s^2 c_1 J_2 + R s^3 J_1 J_2 + L s^4 J_1 J_2)$$

4.4 Queries and Automatic Generation of Models

This example of advanced scripting shows how the easily accessible internal representation in the form of abstract syntax trees can be used for automatic generation of models. The CircuitTemplateFn is a function returning a symbolic representation of a model. This function has two formal pattern parameters where the second one specifies an internal structure. The first parameter is name_, which matches symbolic names. The underscore in name_ is not part of the parameter identifier itself, it is just a short form of the syntax name:_, which means that name will match any item.

The second pattern parameter is the list {type1_, type2_, type3_}, internally containing the three pattern parameters type1_, type2_, type3_. This second parameter will therefore only match lists of length 3, thereby binding the pattern variables type1, type2, and type3 to the three type names presumably occurring in the list at pattern matching. For example, matching {type1_, type2_, type3_} against the list {Capacitor, Conductor, Resistor} will bind the variable type1 to Capacitor, type2 to Conductor, and type3 to Resistor.

```

CircuitTemplateFn[name_,
{type1_, type2_, type3_}] := (
Model[name,
type1 a;
type2 b;
type3 c;
Modelica.Electrical.Analog.Basic.Ground g;
Equation[
Connect[g.p, a.p];
Connect[a.n, b.p];
Connect[b.p, c.p];
Connect[b.n, g.p];
Connect[c.n, g.p]
]
)

```

The aim of this exercise is to automatically generate models based on this template for all combinations of the types that extend the type OnePort in the library package Modelica.Electrical.Analog.Basic.

First we need to extract all the types that extends the type OnePort in the library package Modelica.Electrical.Analog.Basic. This is done by performing a query operation on the internal form using the Select function which has two arguments: the list to be searched, and a predicate function returning true or false.

Only the elements for which the predicate is true are returned. In this case the query is performed on the list of model names in the package `Modelica.Electrical.Analog.Basic`. This list is returned by the function `ListModelNames`.

First we call `GetDefinition` below to load the `Modelica.Electrical.Analog.Basic` package into the internal symbol table:

```
GetDefinition[Modelica.Electrical.Analog.Basic];
```

Then we perform the actual query:

```
types=Select[
  ListModelNames[
    Modelica.Electrical.Analog.Basic
  ],
  Function[
    modelName,
    Not[
      FreeQ[
        GetDefinition[
          modelName,
          Format->MathModelicaFullForm
        ],
        HoldPattern[
          Extends[
            TYPE[Modelica.Electrical.
              Analog.Interfaces.
              OnePort,{},{},{}]
          ]
        ]
      ]
    ]
  ]
];
```

```
{Modelica.Electrical.Analog.Basic.Inductor,
 Modelica.Electrical.Analog.Basic.Capacitor,
 Modelica.Electrical.Analog.Basic.Conductor,
 Modelica.Electrical.Analog.Basic.Resistor}
```

All 64 three-type combinations, e.g. `{Inductor, Inductor, Inductor}`, `{Inductor, Inductor, Capacitor}`, etc., their prefixes not shown for brevity, of these 4 types are computed by taking a generalized outer product of the three types lists, which is flattened.

```
typecombinations =
  Flatten[Outer
    [List, types, types, types],
    2];
```

```
Length[typecombinations]
```

```
64
```

We generate a list of 64 synthetic model names by concatenating the string "foo" with numbers, using the Mathematica string concatenation operation "<>":

```
names = Table[ToExpression[
  "foo"<>ToString[i]], {i, 64}]
{foo1, foo2, foo3, foo4, foo5, foo6,
 foo7, foo8, foo9, foo10, foo11, foo12,
 ...
 foo55, foo56, foo57, foo58, foo59, foo60,
 foo61, foo62, foo63, foo64}
```

Here all 64 test models are created by the call to `MapThread` which applies `CircuitTemplateFn` to each combination.

```
MapThread[CircuitTemplateFn,
  {names, typecombinations}];
```

We retrieve the definition one of the automatically generated models, `foo53`, and unparse it from its internal representation to the Modelica textual form:

```
GetDefinition[foo53, Format->ModelicaForm]
model foo53
  Modelica.Electrical.Analog.
    Basic.Resistor a;
  Modelica.Electrical.Analog.
    Basic.Capacitor b;
  Modelica.Electrical.Analog.
    Basic.Inductor c;
  Modelica.Electrical.Analog.
    Basic.Ground g;

equation
  connect(g.p,a.p);
  connect(a.n,b.p);
  connect(b.p,c.p);
  connect(b.n,g.p);
  connect(c.n,g.p);
end foo53;
```

5 Conclusion

This paper has presented a number of important issues concerning integrated interactive programming environments, especially with respect to the MathModelica environment for object-oriented modeling and simulation. We have especially emphasized environment properties such as integration and extensibility.

One of the current strong trends in software systems is the gradual unification of documents and software. Everything will eventually be integrated into a uniform, perhaps XML-based, representation. The integration of documents, model code, graphics, etc. in the MathModelica environment is one strong example of this trend.

Another important aspect is extensibility. Experience has shown that tools with built-in extensibility mechanisms can cope with unforeseen user needs to a great extent, and therefore often have a substantially longer effective usage lifetime.

The MathModelica system is currently one of the best existing examples of advanced integrated extensible environments. However, as most systems, it is not perfect. There are still a number of possible future improvements in the system including enhanced programmability and extensibility.

Acknowledgements

We would like to thank Peter Bunus for inspiration and great help in MicroSoft Word formatting and conversion from notebook format when preparing this paper, and Dan Costello for Word advice. Acknowledgements to the following individuals for contributions to the design and implementation of the MathModelica system: Andreas Karström, Pontus Lidman, Henrik Johansson, Yelena Turetskaya, Mikael Adlers, Peter Aronsson, Vadim Engelsson, and to Jan Brugård and Andreas Idebrant for contributions to the MathModelica documentation including a number of the examples used in this paper. Thanks to Kristina Swenningsson for creating a nice working atmosphere at MathCore AB. Acknowledgements also to the members of the Modelica Association for creating the

Modelica language, and to EU under the RealSim project for supporting part of the development of MathModelica.

References

- [Andersson-94] Mats Andersson. *Object-Oriented Modeling and Simulation of Hybrid Systems*. Ph.D. thesis, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, 1994.
- [Bunus-00] Peter Bunus, Vadim Engelson, Peter Fritzson. Mechanical Models Translation and Simulation in Modelica. In *Proceedings of Modelica Workshop 2000*. Lund University, Lund, Sweden, Oct 24-26, 2000.
- [Elmqvist-78] Hilding Elmqvist. *A Structured Model Language for Large Continuous Systems*. PhD thesis, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.
- [Elmqvist-96] Hilding Elmqvist, Dag Bruck, Martin Otter. *Dymola - User's Manual*. Dynasim AB, Research Park Ideon, Lund, 1996.
- [Elmqvist-99] Hilding Elmqvist, Sven-Erik Mattsson and Martin Otter. Modelica - A Language for Physical System Modeling, Visualization and Interaction. In *Proceedings of the 1999 IEEE Symposium on Computer-Aided Control System Design*, Hawaii, Aug. 22-27, 1999.
- [Engelson-99] Vadim Engelson, Håkan Larsson, Peter Fritzson. 1999. A Design, Simulation and Visualization Environment for Object-Oriented Mechanical and Multi-Domain Models in Modelica. In *Proceedings of the IEEE International Conference on Information Visualization*, pp 188-193, London, July 14-16, 1999.
- [Engelson-00] Vadim Engelson. *Tools for Design, Interactive Simulation, and Visualization of Object-Oriented Models in Scientific Computing*. Ph.D. Thesis, Dept. of Computer and Information Science, Linköping University, Linköping, Sweden. 2000.
- [Fritzson-83] Peter Fritzson. Symbolic Debugging through Incremental Compilation in an Integrated Environment. *The Journal of Systems and Software*, 3, 285-294, (1983).
- [Fritzson-92a] Peter Fritzson, Dag Fritzson. The Need or High-Level Programming Support in Scientific Computing - Applied to Mechanical Analysis. *Computers and Structures*, Vol. 45, No. 2, pp. 387-295, 1992.
- [Fritzson-92b] Peter Fritzson, Lars Viklund, Johan Herber, Dag Fritzson: Industrial Application of Object-Oriented Mathematical Modeling and Computer Algebra in Mechanical Analysis, In *Proc. of TOOLS EUROPE'92*, Dortmund, Germany, March 30 - April 2, 1992. Published by Prentice Hall.
- [Fritzson-95] Peter Fritzson, Lars Viklund, Dag Fritzson, Johan Herber. High Level Mathematical Modeling and Programming in Scientific Computing. *IEEE Software*, pp. 77-87, July 1995.
- [Fritzson-98] Peter Fritzson and Vadim Engelson. Modelica - A Unified Object-Oriented Language for System Modeling and Simulation. In *Proceedings of the 12th European Conference on Object-Oriented Programming*, ECOOP'98, Brussels, Belgium, July 20-24, 1998.
- [Fritzson-98b] Peter Fritzson, Vadim Engelson, Johan Gunnarsson. An Integrated Modelica Environment for Modeling, Documentation and Simulation. In *Proceedings of Summer Computer Simulation Conference '98*, Reno, Nevada, USA, July 19-22, 1998.
- [Goldberg-89] Adele Goldberg and David Robson, *Smalltalk-80, The Language*. Addison-Wesley, 1989
- [Jirstrand-99] Mats Jirstrand, Johan Gunnarsson, and Peter Fritzson. MathModelica - a new modeling and simulation environment for Modelica. In *Proceedings of the Third International Mathematica Symposium*, IMS'99, Linz, Austria, Aug. (1999).
- [Knuth-84] Donald E. Knuth. Literate Programming. *The Computer Journal*, NO27(2) (May): 97-111. (1984)
- [Knuth-94] Donald E. Knuth, Silvio Levy. The Cweb System of Structured Documentation /Version 3.0. Addison-Wesley Pub Co; 1994.
- [MA-02a] Modelica Association. *Modelica - A Unified Object-Oriented Language for Physical Systems Modeling - Tutorial and Design Rationale Version 2.0*, March 2002.
- [MA-02b] Modelica Association. *Modelica - A Unified Object-Oriented Language for Physical Systems Modeling - Language Specification Version 2.0*, February 2002.
- [Mattsson-93] Sven-Erik Mattsson, Mats Andersson, and Karl-Johan Åström. Object-oriented modelling and simulation. In Linkens, Ed., *CAD for Control Systems*, chapter 2, pp. 31-69. Marcel Dekker Inc, New York, 1993.
- [Otter-95] Martin Otter. *Objektorientierte Modellierung mechatronischer Systeme am Beispiel geregelter Roboter*, Dissertation, Fortschrittberichte VDI, Reihe 20, Nr 147. 1995.
- [Otter-96] Martin Otter, Hilding Elmqvist, Francois E. Cellier. Modeling of Multibody Systems with the Object-oriented Modeling Language Dymola. *Nonlinear Dynamics*, 9:91-112, Kluwer Academic Publishers. 1996.
- [Saldamli-01] Levon Saldamli, Peter Fritzson. A Modelica-Based Language for Object-Oriented Modeling with Partial Differential Equations. In *Proceedings of the 4th International EuroSim Congress*, Delft, the Netherlands, June 26-29, 2001.
- [Sandewall-78] Erik Sandewall. Programming in an Interactive Environment: the "LISP" Experience. *Computing Surveys*, Vol. 10, No. 1, March 1978.
- [Teitelman-69] Warren Teitelman. Toward a Programming Laboratory. In *Proc. of First Int. Jt. Conf. on Artificial Intelligence*, 1969.
- [Teitelman-74] Warren Teitelman. *INTERLISP Reference Manual*. Xerox Palo Alto Research Center, Palo Alto, CA, 1974.
- [Teitelman-77] Teitelman, W. A display oriented programmer's assistant. *Computer*, 39--50. (1977, August 22--25)
- [Tiller-01] Michael M. Tiller. *Introduction to Physical Modeling with Modelica*. Kluwer Academic Publishers, 2001.
- [Visio] <http://www.microsoft.com/office/visio/>
- [Wolfram-88] Stephen Wolfram. *Mathematica System for Doing Mathematics by Computer*. Addison-Wesley, 1988.
- [Wolfram-97] Stephen Wolfram. *The Mathematica Book*, Wolfram Media, 1997.

Dymola for Multi-Engineering Modeling and Simulation

Dag Brück, Hilding Elmqvist, Sven Erik Mattsson and Hans Olsson

Dynasim AB, Research Park Ideon, SE-223 70 Lund, Sweden

E-mail: info@dynasim.se

Abstract

Dymola is an integrated environment for developing models in the Modelica language. The growing use of Dymola has over time increased the demands on the development environment. Requests for extension and redesign originate from two sources: the need to simplify the use of Dymola to better support new and inexperienced users, and the need to better support “power users” which model extremely large and complex systems.

Key areas in the development of Dymola are: a simplified and more coherent graphical user interface, browsing facilities for navigating large and complex systems, new experiment facilities for managing complex simulation tasks, distributed (parallel) simulation, and integrated version control to help manage model libraries and complete models.

The paper describes the extensively redesigned Dymola 5, with an emphasis on new features compared to Dymola 4.

Introduction

Dymola is an integrated environment for developing models in the Modelica language [Modelica Association, 2002; Tiller, 2001], and a simulation environment for performing experiments. It is used since several years within major companies for complex simulations. For example, Dymola has been used to simulate detailed models of complete vehicles including engine, transmission and chassis [Tiller et al., 2000].

Dymola uses hierarchical object-oriented modeling to describe, in increasing detail, the systems, subsystems and components of a model. Reuse of modeling knowledge is a key issue, and is supported by use of libraries containing model classes and by the use of inheritance. Physical couplings are modeled by defining physical connectors and graphically connecting submodels.

Model libraries are available for electronics, rotational, translational and 3D mechanics, thermodynamics, hydraulics and control systems. The libraries range from basic components to more specialized domains

such as the power train library. Predefined libraries can be expanded with user-written model libraries.

The growing use of Dymola has over time increased the demands on the development environment. Requests for extension and redesign originate from two sources:

- The need to simplify the use of Dymola to better support new users and inexperienced users. This is of particular importance when Dymola is used for teaching.
- The need to better support “power users” which model extremely large and complex systems. In this case, the user needs significant support from the environment to handle very large amounts of information, to document complex systems, and to verify results. The development of large component libraries is a collaborative effort involving several people, which requires adequate tool support. Also, different software packages are used which underlines the need for information exchange.

Key areas in the development of Dymola are:

- Simplified graphical user interface. In addition to better structuring, the use of modern GUI elements (help facilities, dockable windows etc.) makes it easier to use the program.
- Browsing facilities for navigating large and complex systems. This includes class browsers for navigating component libraries and a new model browser for navigating complex models.
- New experiment facilities for managing complex simulation tasks. They handle multiple parameter sets, models of different complexity, and tools for validating models.
- Distributed simulation on several computers, allowing parallel simulation for tasks such as optimization.
- Integrated version control to help manage model libraries and complete models. The user needs support for version control to store/retrieve models and associated data, to compare versions of a model, plus mechanisms for documenting the evolution of models.

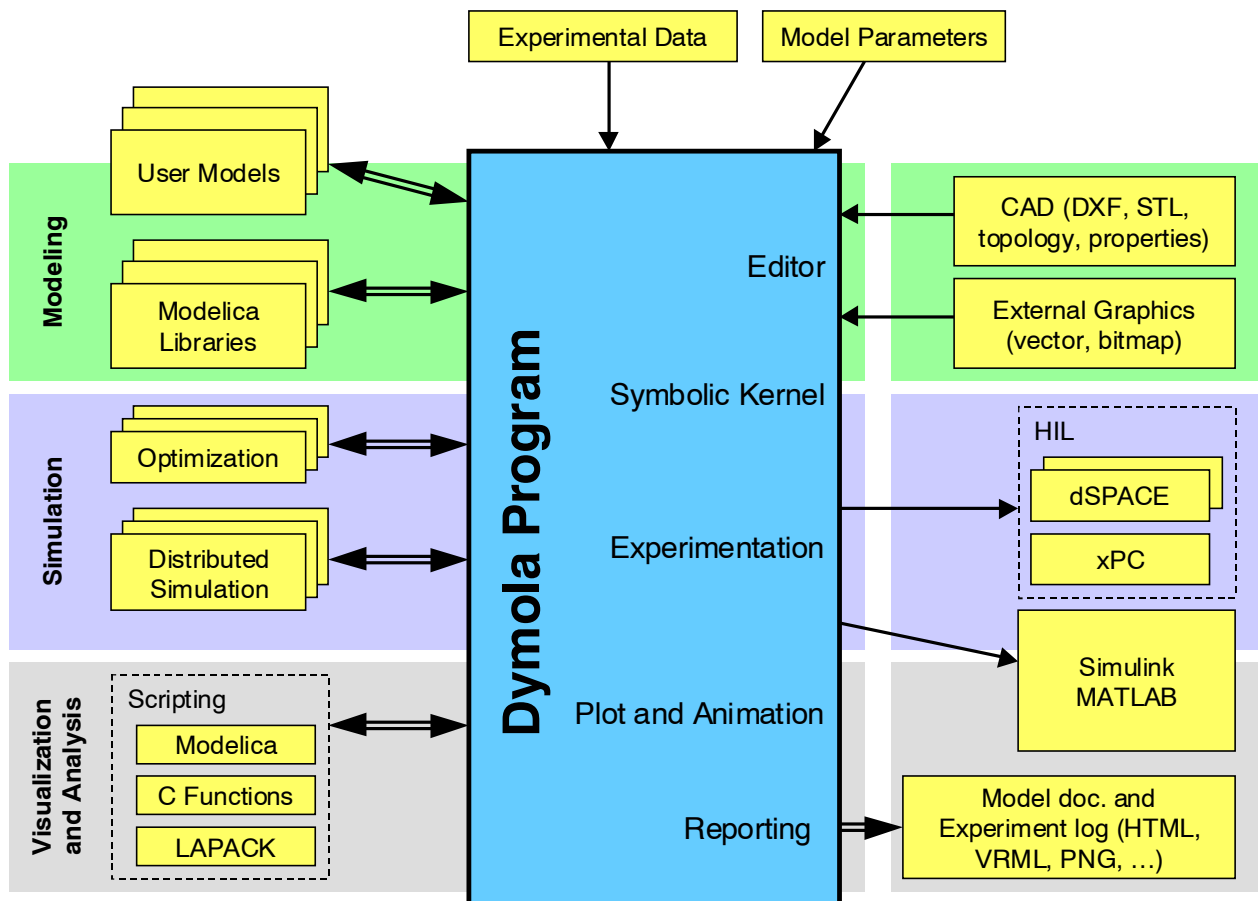


Figure 1. The Dymola architecture.

Dymola architecture

Dymola is an integrated environment for modeling and simulation. Figure 1 describes the architecture and connectivity of Dymola 5.

At the **modeling level**, models are composed from library components (from the Modelica standard library, other free libraries, commercial and proprietary libraries), as well as models developed by the user. Models are either composed of other, more primitive, components, or described by equations at the lowest level. The equation-based nature of Modelica is essential for enabling truly reusable libraries. Measurement data and model parameters cover additional model aspects.

Detailed model knowledge can be imported from CAD packages. Examples of such information are mass and inertia of 3D mechanical bodies, and the topology of a multibody system (bodies and joints). Graphical properties may be described in DXF and STL format. The icons of model components are defined either by drawing shapes in Dymola, or by importing graphics from other tools in common vector or bitmap formats.

At the **simulation level**, Dymola transforms a declarative, equation-based, model description into efficient simulation code. Advanced symbolic manipulation (computer algebra) is used to handle very

large sets of equations. Efficient simulation, including realtime simulation of hydraulic systems, can only be achieved after extensive symbolic transformations of the equations [Elmqvist et al., 2002].

Dymola provides a complete simulation environment, but can also export code for simulation in Simulink. In addition to the usual offline simulation, Dymola can generate code for specialized Hardware-in-the-Loop (HIL) systems, such as, dSPACE, xPC and others.

Recent developments in Dymola 5 allow distributed (parallel) simulation on several computers in a network, for example to perform parameter studies. There are facilities for optimization, also carried out with parallel simulation runs. Such experiments are controlled with a Modelica-based scripting language, which combines the expressive power of Modelica with access to external C libraries, e.g., LAPACK.

The built-in plotting and animation features of Dymola provide the basis for **visualization and analysis** of simulation data. Experiments are documented with logs of all operations in HTML format, including animations in VRML (Virtual Reality Modeling Language) and images. Models and libraries are extensively documented in HTML automatically generated by Dymola from the models themselves.

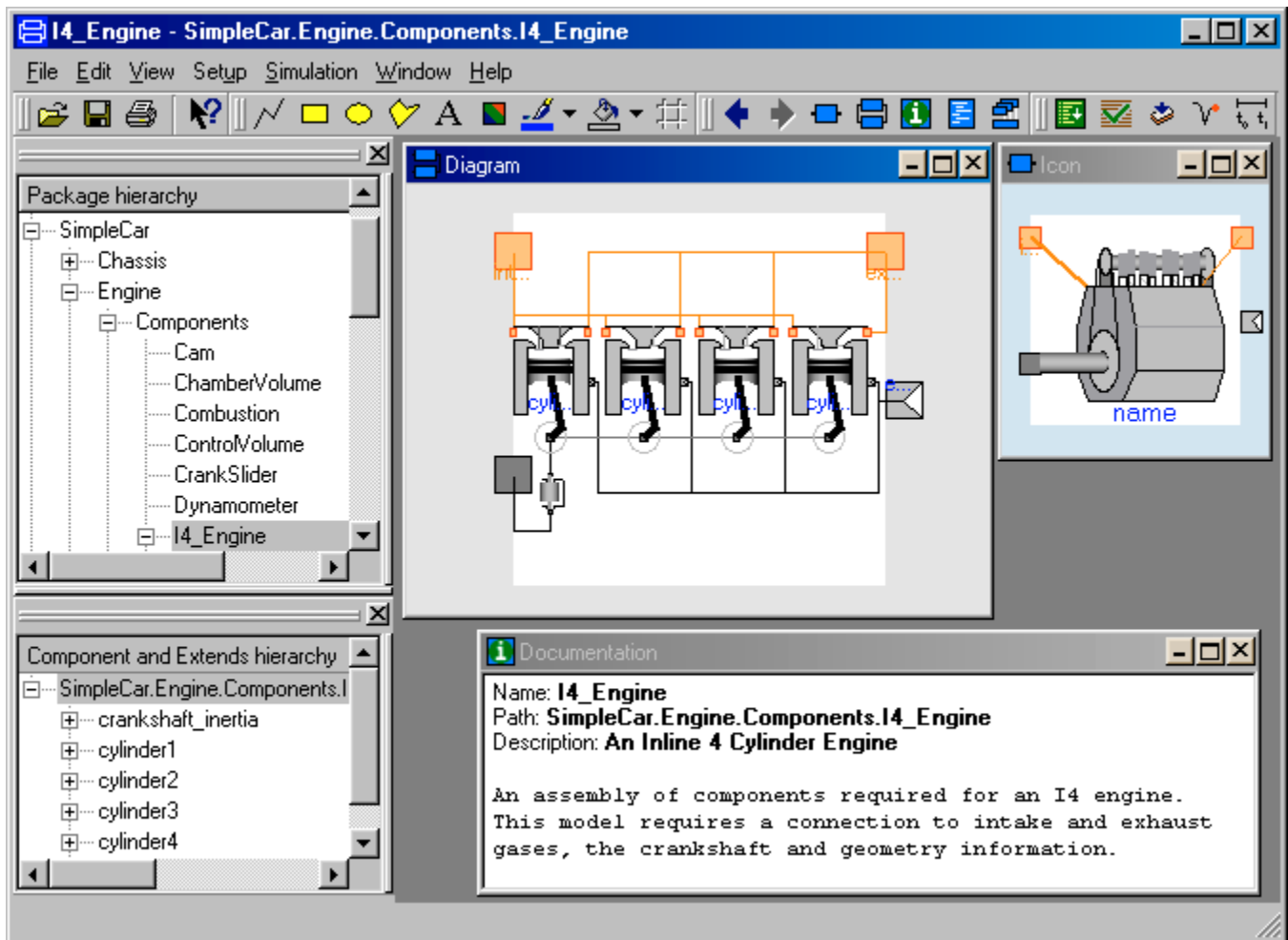


Figure 2. The model editor.

Graphical user interface

The graphical user interface has been extensively redesigned. In Dymola 5 emphasis has been put both on simplifying the task of building models for the novice user and on providing tools for building and managing large and complex models developed by a collaborating team of engineers.

Graphical editor

Figure 2 shows a screen dump of the Dymola modeling environment. The top left tree browser shows the (Package) hierarchy of a library called SimpleCar [Tiller, 2001]. When I4_Engine is chosen different representations (icon and composition diagram) of the model I4_Engine are shown. The lower left tree browser, "Component and Extends hierarchy", shows the hierarchical decomposition, for example, that the engine model contains crankshaft-inertia and the four cylinders: cylinder1, ... cylinder4. A visual representation of that is shown in the Diagram in the middle. An Icon representation of the engine is shown at the top right. A The Documentation window is shown at the lower right. Such a documentation window contains HTML formatted information, i.e.

also graphics and links to other resources may be included.

Editing of models at the fundamental level has been improved by syntax highlighting of the Modelica code, see Figure 3. Another convenience is that models can be dragged from the package browser into the text editor, which gives access to fundamental types in the Modelica library with no typing. Editing in the textual view is instantaneously represented in the graphical view.

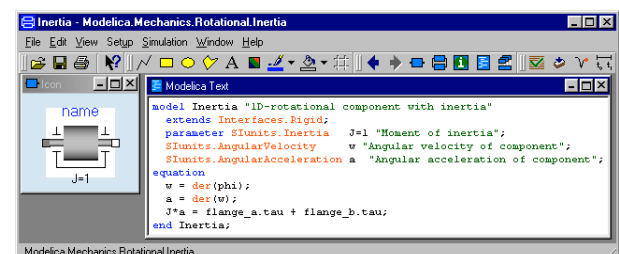


Figure 3. Model editor with syntax highlighting.

The Icon representation can be created with a built-in graphical editor. It allows insertion of lines, rectangles, ellipses, polygons and text strings. Figure 4 shows the tool bar for the graphical editor.



Figure 4. Drawing tools.

It is also possible to insert scalable bitmaps created in other tools like MS Paint and scalable vector graphics from the clipboard. Advanced graphics can thus be created in, for example, MS PowerPoint or MS Visio and inserted into Dymola as Icons or backgrounds for the composition diagrams.

The toolbar also contains controls for setting graphical attributes, e.g., foreground and background color, line style and fill pattern.

As indicated above, Dymola 5 supports Modelica's notion of different layers of information:

- Icon layer
- Diagram layer
- Documentation layer
- Modelica text layer
- Model dependencies layer (generated by Dymola)

It should be noted that Dymola 5 allows several layers to be shown simultaneously.



Figure 5. Navigation tools.

Figure 5 shows the buttons of the navigation tool in Dymola. The first two buttons are used to navigate in the component hierarchy, similar to navigation with a web browser. The back arrow displays the previously visited component; the forward arrow negates the backward move. The other buttons are used to display layers in the graphical editor

Simplifications

In response to user comments, a major design goal was to simplify the graphical user interface. The first step has been to reduce the number windows: both model editing and simulation is controlled from a single window, and plot/animation windows are not opened until a simulation has been performed (or opened explicitly by the user). The design has been influenced by common paradigms, for example, the web-browser approach to navigation.

The design of Dymola 5 more closely follows published guidelines [Microsoft, 1999], and has in general adopted more modern idioms compared to Dymola 4. Common operations are invoked by buttons in addition to menu commands. Dockable windows which either can be part of the main editor window, float on the desktop or be minimized, are used for browsers and similar tools.

The extended use of commonly used GUI elements (toolbars, dockable windows, "what's this" help information) makes Dymola consistent with other applications.

Browsing

The "Package hierarchy" browser shows the library structure and it is possible to drag a component model from the tree into a Diagram in order to add a component to a model, see Figure 6. The browser can either be docked to the editor window as shown in Figure 2, or be dragged onto the desktop.

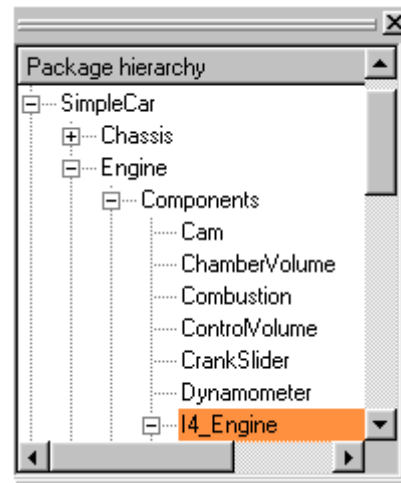


Figure 6. The package browser.

The components of a library can also be viewed as icons in a separate library window, see Figure 7, from which components can be dragged.

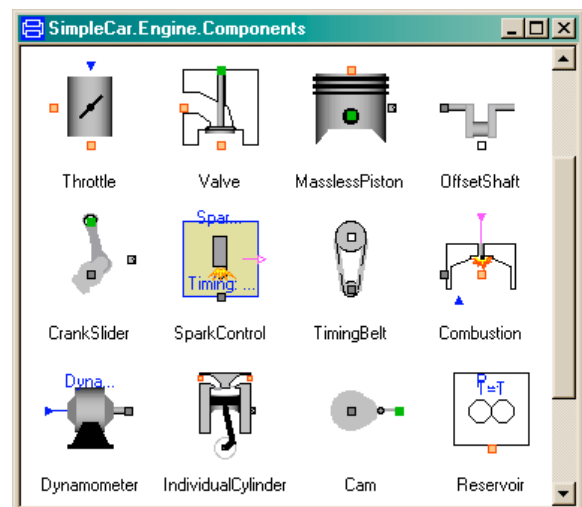


Figure 7. Library window.

The hierarchical structure of a model is shown in the "Component and Extends hierarchy" browser. The top-level components of an engine model are shown in Figure 8.

Maneuvering in this hierarchical structure can be done by clicking in the tree which then changes the view to the selected model. It is also possible to point at an icon and "zoom-in" on the content, i.e. next abstraction layer.

When a model is chosen in the package browser, it becomes the root model of the graphical editor. The root model is used in check, translate and simulate

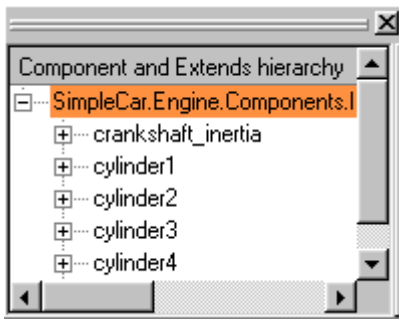


Figure 8. The component browser.

commands. Navigation into its component hierarchy allows inspection of model details, but does not change the root model or permit editing. This view is consistent with the common metaphor used in web browsers.

Dymola 5 has search facilities, for example to search for models that mention particular keywords in the documentation. It may also be useful to find models with a component or a parameter with a known name.

For advanced users, the biggest problem has been to organize the large amount of information in complex models and extensive component libraries. The biggest improvement in Dymola 5 is the use of hierarchical browsers for navigating packages and models. The package browser is also the natural focal point for copying/renaming of models and restructuring of packages.

Advanced Modelica concepts, such as, replaceable classes, is given an intuitive user interface via the component browser. If a class is declared as replaceable, the actual class can be dragged from the package browser onto the replaceable class in the component browser. Other features that benefit from the new user interface are choices (a selection of replaceable classes) and arrays of components.

Visualization in 3D

The graphical editor represents an abstraction of the model, the object diagram. When building 3D mechanical systems, the user greatly benefits from the instantaneous 3D visualization available in Dymola 5. Parameters settings for e.g. the length of a bar can be visually checked in the animation window.

Experimentation

By “experimentation” we mean all the steps necessary to use a model in order to achieve useful results. That includes setting up model parameters and initial conditions, running simulations, analysis of simulation data, and report generation.

Parameter values specific to the studied model have to be entered in a form associated with a component, see Figure 9. Parameters and initial conditions can be set at three different abstraction levels:

- The default values specified in the model of a component, when a reasonable default exists.
- Parameter values that are specified in the modifier list of a specific component. For example, the crankshaft shift is different for each cylinder in an engine.
- Model parameters which are specific for a given top-level model. Such parameters are specified at the top-level of the model, and then propagated through a hierarchical modifier.

Dymola allows the user to set parameters and initial conditions at each of these levels, either through the model editor or while running simulations.

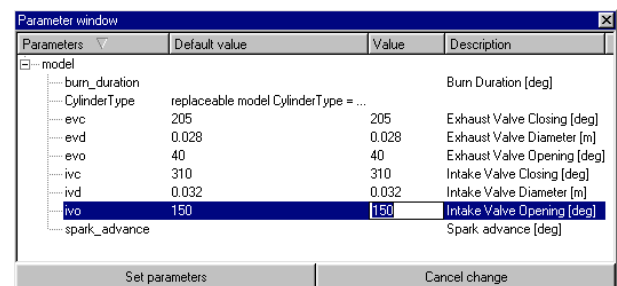


Figure 9. Parameters for specification of details of the engine

For visualization, Dymola offers plotting and 3D animation. Figure 10 shows a window with multiple

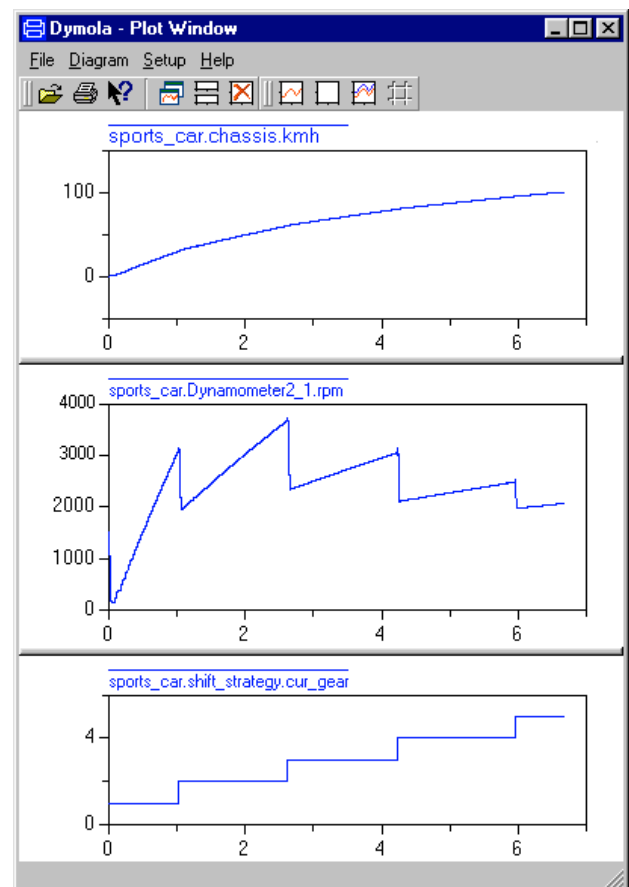


Figure 10. Plot of car speed, engine RPM and selected gear versus time.

plots of car speed, engine RPM and selected gear versus time during such an experiment. The car accelerated to 100 km/h in 6.66 seconds. Plots can be exported as PNG files for inclusion in session log or as vector graphics.

Animation is provided by specialized visualization properties which are present in the mechanical libraries by default. These properties are calculated during simulation and then used to show 3D views in Dymola, as shown in Figure 11. It is also possible to export such animations in VRML format [VRML, 1997], which can be examined with special viewers or with plugins for web browsers.

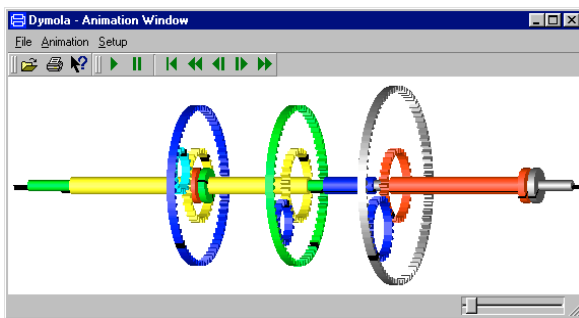


Figure 11. Animation of an automatic gearbox.

Dymola 5 has powerful features for postprocessing of simulation results. It is possible to compare simulation results with experimental data. Data can be imported and exported to other programs like Matlab and Microsoft Excel. There is a scripting language based on Modelica for automating design studies and analysis. Interfaces to subroutine packages such as LAPACK (or other libraries written in C or FORTRAN) enables advanced numerical calculations. The scripting language is also used for running parameter studies in a distributed environment (see below) and for performing optimization.

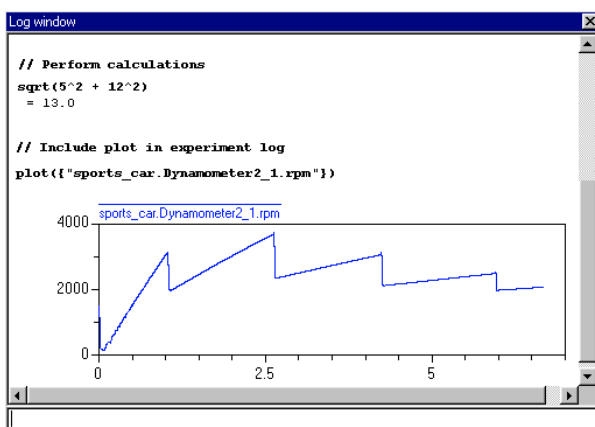


Figure 12. Dymola session window

Automatic logging of design sessions including graphics is provided as HTML code for archiving and sharing over the Internet, see Figure 12. A complete experiment report can be written by editing the session log.

Distributed simulation

During the design phase, hundreds or thousands of simulations have to be performed with different parameter sets. Optimization is used to determine parameters in the model by fitting simulation results to experimental data and to optimize the parameters of a design. It is a task that significantly benefits from parallel simulation. Dymola 5 can use many computers and automatically schedule simulations in parallel to shorten the design cycle.

Figure 13 shows the Dymola monitoring window for parallel simulations. It shows the status of each simulation run: the parameters used and optional criteria result. The Dymola scheduler assigns tasks to computers as they become available. When a simulation finishes, the next task is run on the freed computer. Transfers of the simulation code, input data (parameters and initial conditions) and results are fully automated.

Process	Status	Running on	Started at	Ended at	Parameters	Result
NetDymosim 11%		\\he-notebook\	12:30:48		J1.J=100;	N/A
NetDymosim	Finished	\\he-notebook\	12:30:15	12:30:24	J1.J=1;	5.97979381
NetDymosim	Finished	\\ASTA\	12:30:17	12:30:22	J1.J=5;	9.186667935
NetDymosim	Finished	\\KATJA\	12:30:19	12:30:28	J1.J=10;	9.58768595
NetDymosim	Retrying	\\ASTA\	12:30:24		J1.J=15;	N/A
NetDymosim	Finished	\\he-notebook\	12:30:26	12:30:35	J1.J=30;	9.864166987
NetDymosim	Finished	\\KATJA\	12:30:30	12:30:39	J1.J=50;	9.918705472
NetDymosim	Connecting	\\ASTA\	12:30:34		J1.J=75;	N/A
NetDymosim	Finished	\\he-notebook\	12:30:37	12:30:46	J1.J=80;	9.948200669
NetDymosim 89%		\\KATJA\	12:30:41		J1.J=90;	N/A

Figure 13. Dymola monitoring window for parallel simulations.

During normal simulation on a single computer, a simulation is performed through cooperation between the Dymola program and a separate simulation process. In a distributed environment, a third party, known as the simulation proxy, handles data transfers between Dymola and the simulation task; the use of a proxy allows exactly the same simulation code to run locally and on another computer. As a special case, the “distributed” scheme can utilize multiple CPUs on one computer.

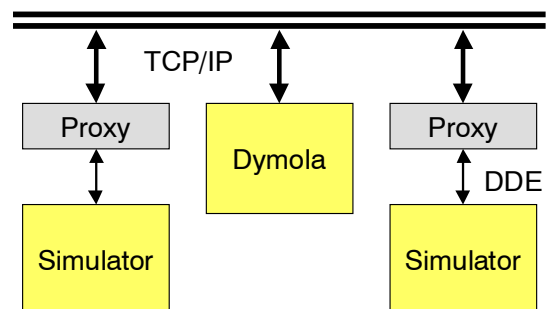


Figure 14. Architecture of distributed simulation.

A proxy is started on each machine willing to act as “compute server”, see Figure 14. On receiving a connection via TCP/IP from a Dymola program, its first task is to help copy the simulation code and input files to a unique area on the server. It then relays parameter settings and commands from Dymola, and

handles data transfers from the simulation to Dymola for online animation and plotting. The Dymola program maintains a list of computers that may be asked to run simulations; the user can control this list by simple commands.

This scheme for distributed simulation is designed for cooperative sharing of resources and quite simple; security measures are limited. First, a computer can only be used as server after the proxy has been started. Second, the proxy runs as an unprivileged process, having only the capabilities of the user starting it. Load is limited because each proxy blocks requests while a simulation is running, but it is possible to start additional proxies to handle multiple simulations (e.g., if the computer has multiple CPUs). Ways to utilize existing system security features need to be further investigated.

Collaborative development

In developing model components for a complex system such as a vehicle, many different kinds of competence are needed. Experts in engines, transmissions and chassis etc. are needed. Because several people are involved in the process, it becomes essential to break up or decompose the overall problem into modular units during development.

The equation-based modeling supported by the Modelica language is fundamental in enabling true reuse of modeling knowledge and the practical use of model libraries. Dymola is able to transform equations of subcomponents as required by the structure of the system. Without the equation-based foundation, several variants of a single model are needed to handle different computational causality. Even worse would have been that the user of a library is given the responsibility to analyze the computational causality of the system in order to pick the right variant.

Inheritance is also important for supporting reuse. Model libraries may include partial models that describe common properties of a set of component types. Such a partial model is conveniently used as a base class to develop models for the individual types of the set by just adding a specific part that distinguish it from the others in the set. This approach makes it simpler to add new component models as well as simplifies maintenance since the common properties of the component types are described only once.

Furthermore, as more people are involved in the process, the development is geographically and chronologically distributed because it is natural to have centers with specific core-competencies. This implies that the modular units developed must be seamlessly integrated to solve the overall problem, and the partitioning should be able to reflect the organizational structure of the model development teams.

In order to increase quality and reduce development time, tools should be made available to

- Provide a structure for organizing, storing and retrieving information (models, documentation, experiment data).
- Support the exchange of information and simplify reuse of models throughout the organization.
- Ensure that correct information is available to each user (versions of libraries, corresponding experiments).

A version control system provides means to track changes to a set of files. A “commit” operation associates a developer and documentation with each change to the common storage of files. The Modelica text of two versions can be compared, and it is possible to back up to any previous version.

The underlying version control system must be able to support multiple concurrent developers working on the same set of models. Extensive locking of files is undesirable in a collaborative environment, and more recent tools also support concurrent development of closely related parts (with appropriate safety nets). A single physical person may have multiple roles in the development or use of the library.

Tracability is essential for maintaining quality over time. Tool enforcement to document modifications before they become publicly available gives the opportunity to review changes and improves quality. The development history and documentation of changes may also be needed for tracing model incompatibilities, for example.

Model testing should be integrated with model development, which implies that the version control system must be able to handle test scripts, support utilities and binary test data. Regression testing, where models are simulated and compared with known good simulation results, is very powerful in detecting involuntary changes to model libraries. A failed regression test may cause either a change of a model, or the revision of the test itself.

Multiple libraries are often used together. In this case, version compatibility across libraries becomes essential. It must be possible to “tag” releases of multiple libraries to indicate compatibility at the project level.

Dymola will support storing, retrieving, etc. of models in version control systems such as CVS (Concurrent Version System) [CVS]. We have deliberately chosen to build on existing version control systems, which offers greater flexibility and better integration than a proprietary system. Because of the textual representation of models in the Modelica language, existing text-based tools can be used, for example, to compare versions. To browse changes in large systems, support in the graphical environment of Dymola is needed.

The use of public libraries has increased in industry over several years. More recent is “open source

development”, which can be described as the loosely organized development (typically of software) by several geographically separated parties. Public websites, such as SourceForge, support Open Source development with web-based tools and CVS. The Modelica Standard Library is maintained as a project at SourceForge.

Library protection

There are many closed simulation packages on the market where you are not able to see what model is used. Modeling is an art in the sense of describing the relevant aspects of the object under observation. It is thus very important to be able to see what assumptions and approximation that the author of a model made. Dymola is open to view all and possibly modify the details by showing of the Modelica code. However, if a company want to protect proprietary information when shipping models, Dymola will support encryption of model details.

A protected library typically consists of parts that are open, and other parts that need protection. Protected parts may require different degree of information hiding, for example

- Preventing unauthorized modification of models (but viewing is unrestricted).
- Parameters and documentation are visible, but model structure and equations are protected.
- The model is regarded as a “black box”. Only model connectors and the icon are available to the user.

The other aspect of library protection is to ensure authorized use. In this case, any use of the library is controlled by options in a license file. A special license is also needed to *make* protected libraries in order to prevent unauthorized distribution.

Acknowledgements

The authors would like to thanks all users of Dymola, who through their suggestion have directly influenced the development of Dymola.

This work was in parts supported by the European Commission under contract IST-199-11979 with Dynasim AB under the Information Societies Technology as the project entitled “Real-time simulation for design of multi-physics systems”.

References

CVS: <http://www.cvshome.org/>

Elmqvist, Hilding, Sven Erik Mattsson and Hans Olsson (2002): “New Methods for Hardware-in-the-loop Simulation of Stiff Models”, Modelica 2002, Modelica Association.

Microsoft (1999): Microsoft Windows User Experience, Microsoft Press.

Modelica Association (2002): “Modelica — A Unified Object-Oriented Language for Physical Systems Modeling”, Language specification version 2.0, January 30, 2002.

Tiller, Michael, Paul Bowles, Hilding Elmqvist, Dag Brück, Sven Erik Mattsson, Andreas Möller and Hans Olsson (2000): “Detailed Vehicle Powertrain Modeling in Modelica”, Modelica 2000, Modelica Association.

Tiller, Michael (2001): Introduction to Physical Modeling with Modelica, Kluwer Academic Publ.

VRML (1997): “Information technology — Computer graphics and image processing — The Virtual Reality Modeling Language (VRML) — Part 1: Functional specification and UTF-8 encoding”, International Standard ISO/IEC 14772-1:1997.

Session 3a

Automotive Powertrains and Hardware-in-the-Loop Simulation

New Methods for Hardware-in-the-loop Simulation of Stiff Models

Hilding Elmqvist, Sven Erik Mattsson and Hans Olsson

Dynasim, Lund, Sweden, E-mail: {elmqvist, svenerik, hans}@dynasim.se

Abstract

The possibilities of multi-domain hierarchical modeling in Dymola often lead to models with both fast and slow parts and the simulation problems become stiff. The usual use of the explicit Euler method for hardware-in-the-loop simulations is not appropriate, because it requires very small step sizes and thus too large computational efforts. The implicit Euler method allows larger step sizes to be used. However, a non-linear system of equations needs to be solved at each step. Reducing the size of the non-linear problem is advantageous. The method of inline integration was introduced to support this. The discretization formulas of the integration method are combined with the model equations and structural analysis and computer algebra methods are applied on the augmented system of equations. This paper describes and illustrates some very important improvements in Dymola's support of the inline integration method. The symbolic analysis and manipulation have been improved and it reduces, in many cases the, size of the non-linear system drastically. Analytic Jacobians for the nonlinear system also increase efficiency and robustness. Support of inline integration of higher order leads to better accuracy for larger steps.

Introduction

Real-time simulation of physical models is a growing field of applications for simulation software. One goal is to be able to simulate more and more complex models in real-time with fast sampling rates. Many of those models are multi-engineering models, which means, that they contain components from more than one engineering domain. Mechanic, electric, hydraulic or thermodynamic components are often coupled together in one model. This leads to a large span of time-constants in the model. The usual use of the explicit Euler method is not appropriate because the fastest time-constant determines the computational effort (step size) for the simulation. In order to maintain stability of the integration method the step size must be less than the smallest time constant. Typically, the fastest modes are not excited to a degree that it is necessary to resolve them for the intended purpose. In such cases the problem is referred as stiff. The implicit Euler method solves the numerical *stability* problem and allows larger step sizes to be used. It is the *accuracy* required that restricts how large

step sizes that can be used. Using the implicit Euler method, on the other hand, implies that a nonlinear system of equations needs to be solved at every step. The size of this system is at least as large as the size of the state vector, n . Solving large nonlinear systems of equations in real-time somewhat problematic because the number of operations is $O(n^3)$ and the number of iterations might vary for different steps. Reducing the size of the nonlinear problem is advantageous. Due to the hybrid nature of the system the Jacobian of the nonlinear system can change drastically between steps. This makes it difficult to apply methods relying on Jacobian updating.

The method of inline integration [3] was introduced to handle such cases. The discretization formulas of the integration method are combined with the model equations and structural analysis and computer algebra methods are applied on the augmented system of equations. For a robotics model with 66 states, the size of the nonlinear system of equations could be reduced to only 6. This method has had little practical use, because certain pragmas about the structure of the model equations had to be put into the model by the user.

Another method, "mixed-mode integration", of reducing the size of the system of nonlinear equations is to use explicit discretization on slow states and implicit on fast states. The problem is then to find the partitioning of the state vector into slow and fast states. A method based on linearization and eigenvalue analysis was presented in [6]. Since the partitioning is based on linearization, special care is needed for highly non-linear and partly discrete model such as friction. In addition it requires a pre-processing step that includes off-line simulation and "suitable" inputs. It is thus not straightforward to use this method.

This paper describes and illustrates some important improvements in Dymola's [1,2] support of the inline integration method.

1. The symbolic analysis and manipulation have been improved and it reduces, in many cases the, size of the non-linear system drastically.
2. The generation of analytical Jacobians has been improved.
3. Inline integration of higher order methods are supported.

The large possible reduction of the size of the implicit non-linear system of equations is due to the fact that certain subsystems might be linear even after amendment of the corresponding discretization formulas. Dymola is now able to automatically detect such structures during the structural analysis of the equations. Furthermore, in certain cases the corresponding linear subproblem is sparse. This is, for example, the case for discretized PDE's. For a one dimensional PDE, a band structure is obtained. The usual technique of tearing then implies a reduction of the size of the problem. For a PDE model with 10 elements, the size of the nonlinear problem, i.e. the number of iteration variables, can often be reduced to one when a first order spatial discretization is used.

The implicit inline Euler technique solves the numerical stability problem. However, the step size need to be chosen small enough to get desired accuracy. Dymola support of inline integration has been extended with higher order methods to meet the accuracy requirements. The use of higher order methods is necessary for e.g. hydraulics systems where one can have oscillations in the kHz-range and want to use step-sizes for external sampling in the same range.

Exploiting sparse structures

Consider a system of differential algebraic equations (DAE)

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

where t is time, x and y are vectors of unknown variables. The elements of x are called dynamic variables since their time derivatives, \dot{x} , appear in the equations and the elements of y are called algebraic variables since none of its derivatives appear in the equations.

When making inline discretization, the model equations are combined with the discretization formulas of the integration method. For implicit Euler we get the nonlinear problem

$$F(t_i, x_i, \dot{x}_i, y_i) = 0$$

$$\dot{x}_i = (x_i - x_{i-1}) / h$$

to solve for x_n and y_n at each step. Also for an ODE on explicit state space form,

$$\dot{x} = f(t, x)$$

the inlined integration method using implicit Euler gives a non-linear problem. The size of the problem is the size of the state vector.

The non-linear systems obtained when combining the discretization formulas of implicit integration methods with model equations are sparse, because typically a model has hundreds or thousands of unknowns, while each equation refer to very few, say ten, variables. There is much structure to exploit.

Let us represent the structure of a system by a structure Jacobian, J , where each row represents a scalar equation and each column represents an unknown variable of the system. If variable j does not appear in equation i then $J_{ij} = 0$. Otherwise it is one. The representation can be extended to indicate how it appears, for example, whether it appears linear or not.

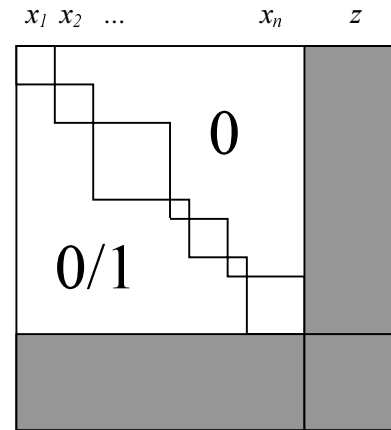


Figure 1: A desired structure for the Jacobian.

Consider a structure Jacobian of the form as shown in Figure 1. The elements of the right and lower borders (the grey part) can have any values. It is the structure of the upper left part (the white part) that is important. It shall be block lower triangular (BLT) and each diagonal block shall be non-singular.

If the z variables are assumed known, the problem of solving for the x variables is decomposed into a sequence of smaller problems that be solved in turn giving x_1, x_2, \dots, x_n .

It means that when using a numerical solver to solve the total problem, the numerical solver needs only iterate over the z variables which is a smaller problem. A numerical solver needs residuals to be calculated, when it provides a value for the z . The residual is calculated in the following way

1. Solve in turn the sequence of problems for the x_i values using the given z value and the x_j ($j < i$) values already calculated.
2. Use the z value and the calculated x_i values to calculate the residuals of the remaining equations at the bottom.

To obtain efficient simulation, the aim is to obtain a small number of z variables while keeping the sequence of problems to solve x_i simple. It is favourable if the calculations of the x variables are just a sequence of assignment involving no numerical solvers. Small linear systems of equations are also acceptable. It is very important that the subproblems to solve for the x_i variables are nonsingular. If the original problem is non-singular, then the manipulation must not introduce singularities or divisions by zero. Unfortunately, it is not only a question avoiding divisions by zero, but also divisions by too small

numbers. When solving linear equations this is commonly solved by pivoting in order to avoid large condition numbers of the factorized matrices.

When solving the outer nonlinear problem, it is favourable to use Newton methods. Fixed point iteration cannot be used for stiff problems. Newton methods need the Jacobian of the problem. Let n_z denote the number of elements of z . The Jacobian can be calculated numerically, by performing additional n_z residual calculations, which may be costly. By generating code for analytic calculation of the Jacobian the effort to calculate all non-zero elements of the Jacobian typically is of the same magnitude as one residual calculation, which is a considerably less effort. This reduction is due to common subexpression elimination.

Higher order methods

In order to get sufficient accuracy for large steps it was necessary to extend the basic method to higher order methods. Higher order methods indicate that they have order greater than one, and the ones considered have orders 2 to 4.

The higher order methods implemented for the new method are L-stable singly diagonally implicit Runge-Kutta methods [4]. The L-stability implies that they are stable for all stable linear systems and do not exhibit oscillations for very stiff systems. The class of methods, singly diagonally implicit Runge-Kutta methods, require the solution to the same equation systems as implicit Euler.

Other high order methods

More general implicit Runge-Kutta methods can often be made more efficient in off-line simulations. However, this requires more costly factorizations that can be shared between many steps and is thus not suitable for realtime simulations. Multi-steps methods and other methods that propagate more information from one step to the next are not suited for real-time simulations of hybrid systems.

Example: One dimensional PDE

Discretized partial differential equations (PDE) have special sparse structures because each unknown appears only in a few equations. For a one dimensional PDE, a band structure is obtained.

Consider the following PDE, modeling one-dimensional heat diffusion.

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}$$

with boundary conditions

$$u(x, t = 0) = 20 \sin\left(\frac{\pi x}{2L}\right) + 300$$

$$u(x = 0, t) = 20 \sin\left(\frac{\pi}{12}t\right) + 300$$

$$\frac{\partial u}{\partial x}(x = L, t) = 0$$

where L is the length of the object. By discretizing in space

$$\frac{\partial^2 u}{\partial x^2}(x, t) \cong \frac{u(x + \Delta x, t) - 2u(x, t) + u(x - \Delta x, t)}{\Delta x^2}$$

where $\Delta x = L/n$ with n being the number of discrete elements, the PDE can be transformed into a set of ODEs. The matrix notation allows convenient description of the discretized model.

```

model PDE
  parameter Real L = 1;
  parameter Integer n = 50;
  Real Dx = L/n;
  constant Real Pi=3.14159265;
  Real u[n+1];
equation
  u[1] = 20*sin(Pi/12*time) + 300;
  der(u[2:n]) = (u[3:n + 1] -
    2*u[2:n] + u[1:n - 1]) / (Dx*Dx);
  u[n + 1] = u[n - 1];
initial equation
  u[2:n] = 20*sin(Pi/2*(1:n-1)*Dx) +
    300*ones(n-1);
end PDE;
```

The discretized ODE is conveniently written by use of shifted sub-ranges of the vector u . The boundary condition at $t=0$ is given as an initialization equation. The sine function is evaluated elementwise on the sequence. The boundary condition at $x=0$ is handled by making $u[1]$ an algebraic variable with given time dependency. The boundary condition at $x=L$ is handled by adding one element to u , namely $u[n+1]$, and the equation $u[n+1] = u[n-1]$.

By discretizing in time using implicit Euler

$$\text{der}(u[i]) = (u[i] - \text{old}(u[i]))/h$$

where h is the step size and the expression $\text{old}(u[i])$ denotes the value of $u[i]$ at the previous step, the ODE

$$\text{der}(u[2:n]) = (u[3:n + 1] - 2*u[2:n] + u[1:n - 1]) / (Dx*Dx)$$

is transformed into

$$u[3:n + 1] = (2+a)*u[2:n] - u[1:n - 1] - a*\text{old}(u[2:n])$$

where a is the constant

$$a = Dx \cdot Dx / h$$

The first component of the discretized ODE is

$$u[3] = (2+a) \cdot u[2] - u[1] - a \cdot \text{old}(u[2])$$

The variable $u[1]$ is known because it simply calculated from the boundary condition given as a pure time dependent expression. Thus the equation has two unknowns, $u[2]$ and $u[3]$, since all old expressions are known quantities when taking a new step. If $u[2]$ is known, it is simple to calculate $u[3]$.

Let us assume $u[1:3]$ to be known and consider the second component of the discretized ODE

$$u[4] = (2+a) \cdot u[3] - u[2] - a \cdot \text{old}(u[3])$$

which is simple to use to calculate $u[4]$. Proceeding in the same way for all components of the discretized ODE, we find equations for calculating $u[3:n+1]$ in a simple way when $u[2]$ is assumed to be known.

The remaining equation is

$$u[n+1] = u[n-1];$$

which now is used to give the residual $u[n+1]-u[n-1]$ for calculating $u[2]$ iteratively. In other words the numerical solver need only iterate over one variable.

Since this problem is linear, Dymola continues the symbolic manipulation and uses the explicit expressions for $u[3:n+1]$ to back-substitute the residual equation to get an equation for $u[2]$ and solves this equation symbolically. Dymola has transformed the model to a simple sequence of assignments and there is no need for a numerical solver.

This model for heat diffusion is not stiff, but it illustrates very well how the sparse structures of discretized PDEs can be exploited. Moreover, such a model can be part of a model that is stiff. Dymola is then able to find and treat these equations as described.

Models of hydraulics systems are stiff. Models to describe pressure wave oscillations in the kHz range in long lines have the same banded structure as discussed above and Dymola is able to find and to reduce the size of the non-linear system of equations automatically.

Example: Multi-body systems

Consider modeling of multi-body systems. The equation of motion can be written as

$$M(q)\ddot{q} = F(q, \dot{q})$$

where q is a vector of generalized coordinates representing the system's position (distances or angles), M is the non-singular mass matrix, and F represents applied forces. Let n denote the number of elements of q or in other words the degree of freedom for the

mechanical systems. The states are q and \dot{q} . Thus the number of states is $2n$.

When simulating this using an explicit ODE solver, it is a major task to invert the mass matrix to solve for the accelerations. When using implicit inlining, inverting the mass matrix can be avoided and the size of the non-linear system to be solved can be reduced from $3n$ to n . The approach is to iterate over the accelerations \ddot{q} and use the the discretization formulas to calculate q and \dot{q} , and use $M(q)\ddot{q} - F(q, \dot{q})$ as the residual. This approach was presented in [1]. However, this method has had little practical use, because certain pragmas about the structure of the model equations had to be put into the model by the user.

The new structural analysis methods of Dymola automatically rediscovers well-known $O(n)$ method by Luh, Walker, and Paul for calculating the joint forces and torques from the motion of the joints (q , \dot{q} and \ddot{q}). Dymola is able to find this approach automatically without no hints or exploiting facts that it is a multi-body model. Dymola makes it by only analyzing the structure of the equations and manipulate them properly. The component models of the library ModelicaAdditions.MultBody result typically in a hundred unknowns for each degree of freedom. Thus, it is far from trivial to transform an inlined model to this efficient form for numeric solution. Moreover, Dymola is able to find the core problem in more complex settings such as for a robot with drivelines and controllers. This is illustrated in the following application.

Application: Robotics model

Consider the model `r3.robot` in the Modelica [5] library ModelicaAdditions.MultiBody.Examples.Robots as shown in Figures 2 and 3.

The model describes an industrial robot with six degrees of freedom. The model is composed of basic mechanical components such as joints and bars as shown in part 3 of Figure 2. At every joint, a drive train as shown in part 4 of Figure 2 is present. Each drive train contains a motor, a gearbox and an actuator as well as a control system. The elasticity of the gears of the first three joints is modelled by one spring for each gearbox. The elasticity of the last three joints is neglected. In part 5 of Figure 2, the model of the motor and the actuator of one joint is shown. This component is defined, most naturally, as an electrical circuit. Finally, in Figure 3, the control system with tachometer filters for one drive train is defined in block diagram format. To simplify the discussion, we omit potential locking in the joints due to bearing friction.

The model consists of 12 states for the mechanical part of the robot, two states for every gearbox with modeled elasticity, two states for every motor/actuator component, three states for every tachometer filter, and three

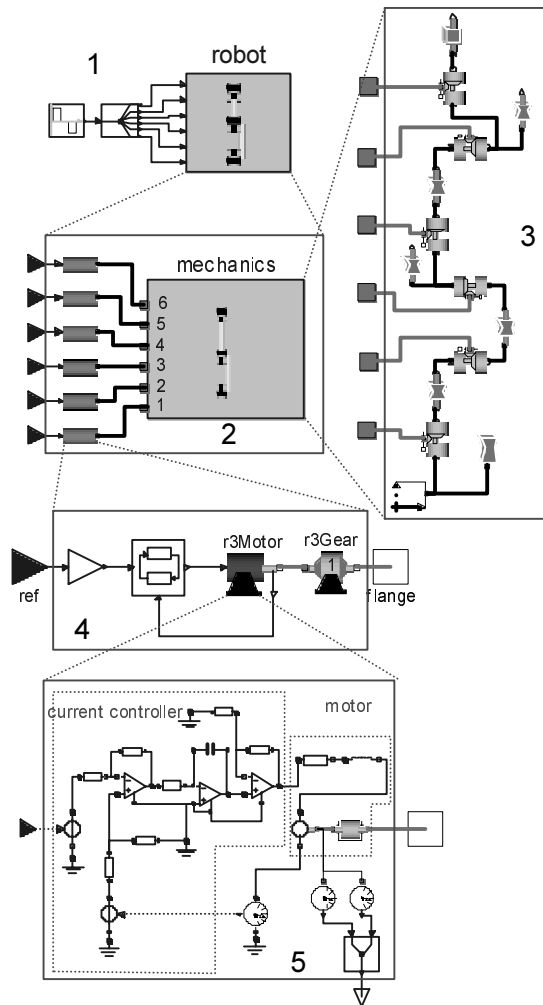


Figure 2: The robot model r3.

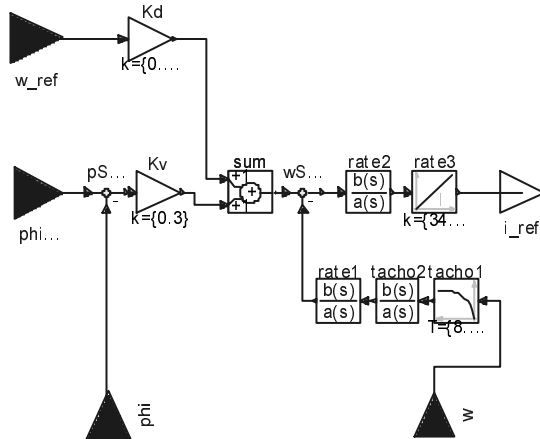


Figure 3: The controller of one joint.

states for every controller. The overall model has thus $12+3\cdot 2+6\cdot (2+3+3) = 66$ states. The simulation problem has additional 12 states for generating the reference path.

The model has 5963 unknowns. After Dymola's elimination of constant and alias variables at translation, 932 nontrivial and time-varying variables remain. For explicit methods there is one linear equation system to solve. It is of size six. It corresponds to the inversion of the mass matrix. Dymola has solved all other equation systems symbolically.

When using inlined explicit Euler, a step size of 0.05 ms has to be selected to achieve stable behavior. The paper [6] reports that the fastest eigenvalues of the linearization of the system are about 7000 in magnitude.

For the inlined implicit Euler, Dymola translates the simulation problem to a non-linear system of size 6 with no additional local equation systems. The equation systems from discretizing the drive trains and their controllers are linear and Dymola is able to solve them symbolically.

Table 1: Performance of the methods for the robot problem that is simulated for 1 s using a Pentium IV 1.6 GHz processor.

	Inl. Expl. Euler	Inl. Impl. Euler	Inl. Impl. RK 3	Inl. Impl. RK 3
Step size [ms]	0.05	1	5	10
Pos. error [mm]	0.1	3	0.1	0.4
Vel. error [mm/s]	5	20	5	20
CPU time [s]	1.97	0.16	0.11	0.06

The resulting execution times and maximum position and velocity errors compared to a reference solution calculated using DASSL are shown in Table 1. When judging the errors it may be of interest to know that the robot is of meter size and the maximum speeds are 2-4 m/s.

For easy interpretation of the execution times the problem was simulated for one second. It means that if the CPU time is less than one second, the simulation runs faster than real-time.

When using explicit Euler the simulation runs slower than real-time. The solution has good accuracy, but the computational burden is high. It is very interesting to see that the inlined third order implicit Runge-Kutta method gives a solution with the same accuracy only needing 6% of the effort for the explicit Euler method.

The implicit methods run all faster than real-time. As reported above Dymola is able to reduce the size of the non-linear system to six. Before the new improvements Dymola reduced the size of the non-linear system to 39 giving a CPU of 1.1 s for the simulation. Using the new approach the implicit Euler method needs only 0.16 s

for the simulation. The simulation is speeded up more than six times and it runs faster than real-time.

The table shows that high order methods pay off. The third order Runge-Kutta method gives with less effort a better result than implicit Euler does.

If we let the robot model allow potential locking in the joints due to bearing friction when using the inlined third order implicit Runge-Kutta method with a step size of 5 ms, the CPU time needed is 0.16 s. Thus, this model runs also much faster than real-time.

Conclusions

This paper has described and illustrated Dymola's new approach to inlined implicit integration. The new features include more advanced analysis and manipulation of the inlined problem giving in many cases a drastic reduction of the non-linear problem that has to be solved numerically. Generation of analytic Jacobians also increases performance. Support of inline integration of higher order methods leads to better accuracy for larger steps. Thus allowing faster simulation.

Reported experiences of applying the new approach to simulation of an industrial robot have shown very promising results. The method has also been applied successfully to simulating hydraulic systems with long pipes exhibiting pressure wave oscillations.

Acknowledgements

This work was in parts supported by the European Commission under contract IST-199-11979 with Dynasim AB under the Information Societies Technology as the project entitled "Real-time simulation for design of multi-physics systems".

References

- [1] D. Brück, H. Elmqvist, S.E. Mattsson, H. Olsson: *Dymola for Multi-Engineering Modeling and Simulation*, Proceedings of Modelica 2002. Modelica homepage: <http://www.Modelica.org>,
- [2] Dymola. *Dynamic Modeling Laboratory*, Dynasim AB, Lund, Sweden, <http://www.Dynasim.se>
- [3] H. Elmqvist, F. Cellier, M. Otter Inline Integration: A new mixed symbolic/numeric approach for solving differential-algebraic equation systems. Proceedings: European Simulation Multiconference June 1995 Prague, pp: XXIII-XXXIV.
- [4] Hairer, Wanner: *Solving Ordinary Differential Equations II*, Springer Verlag

[5] Modelica. Modelica homepage:
<http://www.Modelica.org>,

[6] Schiela, Olsson: Mixed-mode Integration for Real-Time Simulation, Modelica Workshop 2000, October 23-24, 2000, Lund University, Lund, Sweden

Application of mixed mode integration and new implicit inline integration at Toyota

Shinichi Soejima
Toyota Motor Corporation

Takashi Matsuba
Toyota Techno Service Corporation

The HILS (Hardware In the Loop Simulation) is a popular technique to debug control logic of vehicles. Previously, only simplified models could be used to achieve real time performance in the simulations. On the other hand, quite detailed models of engine, drivetrain, hydraulics and brake system were developed with Dymola in recent years. Therefore we would like to use these models also in HILS. However, real time is difficult to obtain for stiff model components, such as the hydraulics, because integrators with fixed step size must be used. With explicit methods very small step sizes are needed to ensure stability. With implicit methods large nonlinear systems of equations have to be solved. Both approaches seem to be not feasible. To improve this situation, the new inline and mixed mode integration technique introduced by Dymola is evaluated for an engine model and results are reported.

1. Introduction

Concerns over fuel consumption and environmental problems have brought about a demand for higher performance in automobiles. To achieve this, the development of highly advanced systems using control technologies that incorporate the use of numerous actuators and sensors has been progressing. The composition and control of such systems is becoming increasingly more complicated. However, at the same time, reducing the length of their development period is also necessary. Applying simulation is essential for achieving this task. Particularly, HILS (Hardware In the Loop Simulation) is widely utilized in the debugging of ECUs (Electronic Control Units). In HILS, the ECU carries out its operation in real time, and as a result, the model is also required to carry out its operation in real time. Simple models with experimental data tables and transfer functions have been used for HILS so far. However, the demand is rising for Dymola models, that have been developed during the design of control logic, to be used in HILS without changes.

Physical models have typically a large span of time constants making them stiff for real time calculation. When using the explicit Euler method, the step size must be less than the fastest time constant in order to maintain numerical stability. However, in real-time simulation using

HILS, the step size cannot be set shorter than the length of time necessary for calculating the new values of the model variables. The implicit Euler method allows a larger step size to be used. However, it implies that a nonlinear system of equations needs to be solved at each step. Reducing the size of the non-linear problem is advantageous. Dymola [1] exploits the method of inline integration [3,4] to support this. The discretization formulas of the integration methods are combined with the model equations. To reduce the size of the resulting non-linear problem, Dymola analyses the structure of the problem and manipulates it symbolically. The symbolic manipulation has recently been improved [4]. The improvements include also inline integration of higher order methods to obtain better accuracy for larger steps.

Another method, "mixed-mode integration", of reducing the size of the system of non-linear equations is to use explicit discretization on slow states and implicit on fast states. The problem is then to find which states that are slow and which that are fast. A method based on linearization and eigenvalue analysis was presented in [6]. Unfortunately, it is not straightforward to use this method.

This paper reports results from applying inline integration and mixed mode integration to two real applications.

2. Application: Engine model

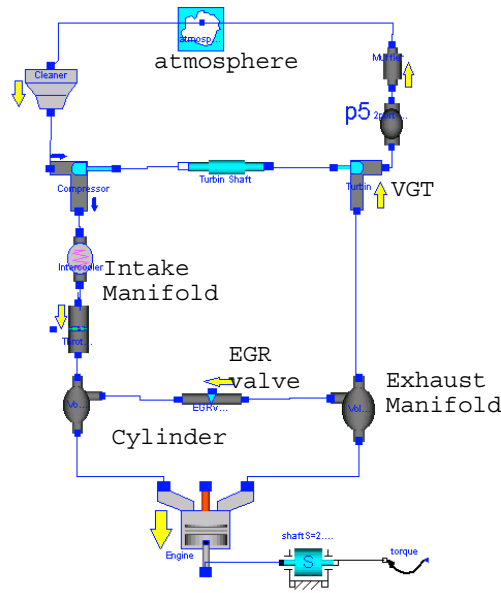


Figure 1: Engine model

Figure 1 shows the structure of the engine model which was used for evaluation. The engine has a Variable Geometry Turbo (VGT) and an Exhaust Gas Recirculation (EGR) system. The EGR system reduces nitrogen oxide (NOx) by recirculating exhaust gases back to the intake manifold. The VGT system increases the exhaust pressure by restricting the flow of burned gas using vanes installed at the entrance. To achieve low emission levels, it is important to control the VGT and EGR correctly. A complication is that both the VGT and the EGR influence the intake manifold pressure and fresh air/EGR gas flow into the engine.

The model is a mean value engine model. The variation of torque or cylinder pressure during a cycle is not calculated. The model builds on conservation of energy and mass and Newton's equations of motion. The amount of air mass flow and EGR gas flow, and the pressure and temperature of every part of the engine are calculated. The model has 796 unknowns. After Dymola's elimination of constant and alias variables at translation, 183 nontrivial and time-varying variables remain. The model has 26 continuous time states. The mass flows of the air or EGR gas are calculated by the Equation (1).

$$\dot{m} = CA\sqrt{2p_1\rho_1}\Phi \quad (1)$$

$$\Phi = \begin{cases} \frac{\kappa}{\kappa-1} \left\{ \left(\frac{p_2}{p_1} \right)^{\frac{2}{\kappa}} - \left(\frac{p_2}{p_1} \right)^{\frac{\kappa+1}{\kappa}} \right\} & \text{if } \left(\frac{p_2}{p_1} \right) > \left(\frac{2}{\kappa+1} \right)^{\frac{1}{\kappa-1}} \\ \left(\frac{2}{\kappa+1} \right)^{\frac{1}{\kappa-1}} \sqrt{\frac{\kappa}{\kappa+1}} & \text{if } \left(\frac{p_2}{p_1} \right) < \left(\frac{2}{\kappa+1} \right)^{\frac{1}{\kappa-1}} \end{cases}$$

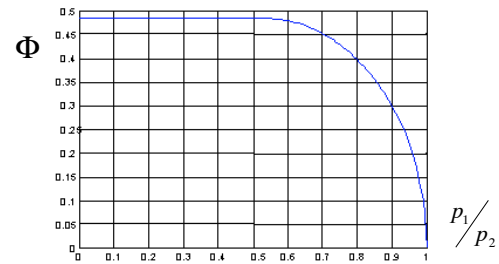


Figure 2: The flow characteristics

The equation shows that the flow changes are large when there is a small difference between the upper pressure and the lower pressure. It implies that the time constants of the dynamics change. As a result, the use of explicit Euler method with fixed step size, requires a small step size for the simulation.

2.1 Evaluation of inlined explicit Euler

We compared the calculation time of two 10 seconds simulations, one in which the explicit Euler inline integration method was applied and one in which a non-inlined explicit Euler method was used. Table 1 shows the results. We can see a 39% performance improvement when using the inlined explicit Euler in Dymola.

The results are labelled "Simulink" when simulating the S-function generated by Dymola in Simulink. The 'sim', 'tic' and 'toc' commands were used for timing. It should be noted that using the simulate button in Simulink 3.0 (Matlab 5.3) made the simulation of these models about 50% slower. This situation has been improved in Simulink 4.0 (Matlab 6.0).

It should be noted that there was not any significant improvement in speed when simulating in Simulink. The reason has not yet been determined.

Table 1: Performance for explicit Euler when simulating the engine model for 10 seconds using an Intel Pentium II 350 MHz processor.

	Integration Method	Step [ms]	CPU Time [s]
Dymola	Explicit Euler	0.1	41
	Inlined Expl Euler	0.1	25
Simulink	Explicit Euler	0.1	46
	Inlined Expl Euler	0.1	44

2.2 Evaluation of mixed mode integration

Next we evaluated mixed mode integration. The partition with 4 fast state variables discussed in [6] was used. Table 2 shows the results of a comparison of calculation time when mixed mode integration was used with the Engine model and when a non-inlined explicit Euler method was used. The mixed mode integration method showed very high performance. The improvement of the performance is about 85%. And without mixed mode integration, the model simulation needed 0.1 ms of step size to ensure calculation stability. On the other hand, with mixed mode integration the calculation was still stable with a step size of 1.0 ms.

The explicit Euler method cannot be used for HIL simulation, because a step size of 0.1 ms is needed, which is too short for HILS calculation. The mixed mode integration method allows HIL simulation as the results from running on dSPACE 1005 (PowerPC 750, 480 MHz) indicate. However, there are problems of using the mixed mode approach in general

1. It is difficult to find the suitable partitioning.
The operation is difficult to generalize because the partitioning depends on the characteristic of each model, and thus requires trial and error analysis on each occasion.
2. Once partitioning has been made, stability is not guaranteed when the input is changed.

Table 2: Performance for mixed mode integration when simulating the engine model for 10 seconds using an Intel Pentium II 350 MHz processor.

	Integration Method	Step [ms]	CPU Time [s]
Dymola	Explicit Euler	0.1	41
	Mixed Mode	1	6.1
Simulink	Explicit Euler	0.1	46
	Mixed Mode	1	7.1
dSPACE	Mixed Mode	1	Not Realtime
	Mixed Mode	1.3	Realtime

2.3 Evaluation of inlined implicit Euler

In order to provide more easy to use method than the mixed mode integration method, the implicit line method reported in [6] has been considerably improved [4]. The performance results of using this method are shown in Table 3.

Table 3: Performance for inlined implicit Euler integration when simulating the engine model for 10 seconds using an Intel Pentium II 350 MHz processor.

	Integration Method	Step [ms]	CPU Time [s]
Dymola	Explicit Euler	0.1	41
	Inlined Impl Euler	0.1	14.3
Simulink	Explicit Euler	0.1	46
	Inlined Impl Euler	0.1	15

It can be noted that the simulation time was about 15 seconds on the Pentium II processor, i.e. 1.5 ms/step. Since dSPACE DS1005 according to Table 2 needs more CPU time per step it would need at least 2 ms to calculate one step. However, using offline simulation in Dymola, it was determined that convergence was not obtained when the step size 2 ms was used.

Faster HILS environments based on Pentium processors are available, for example xPC from MathWorks. Dynasim made test using an Intel Pentium 4, 1.6 GHz processor. Table 4 shows the performance results for simulating in Dymola on xPC. The model could then be run in real-time.

Table 4: Performance for inlined implicit Euler integration when simulating the engine model for 10 seconds using an Intel Pentium 4, 1.6 GHz processor.

	Integration Method	Step [ms]	Comp Time [s]
Dymola	Explicit Euler	0.1	11.4
	Inlined Impl Euler	1	4.6
Simulink	Explicit Euler	0.1	13.6
	Inlined Impl Euler	1	4.6
xPC	Inlined Impl Euler	1	Realtime/4.5s

3. Application: Hydraulic system

The hydraulic system is one of the most important systems in an automobile. It plays a crucial role in the power train and the drive train. As the hydraulic system exhibits very complicated characteristics, it is very useful to simulate it. Additionally, the ability to simulate such systems in real time is very important. So far real-time simulation of hydraulic systems has not been possible. Hydraulic systems are typical examples of very stiff systems. It is necessary to use implicit solvers.

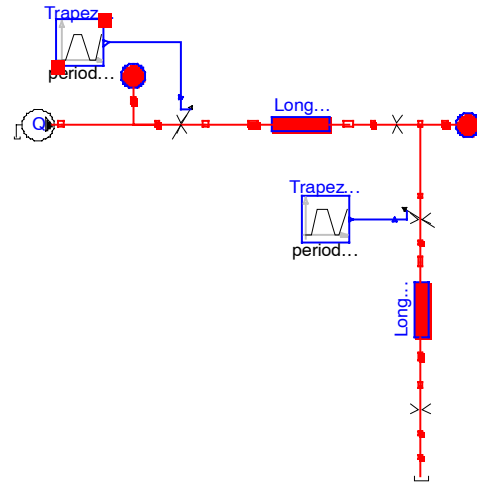


Figure 3: Hydraulic actuator system.

As the next application, consider simulation of an actuator which is usually used to control the pressure in automobiles. Figure 3 shows the structure of the model developed by using components from the HyLib hydraulics library [2]. There are a number of valves and pipes between a high-pressure source and low-pressure parts such as the brake at a wheel. The dynamics within the pipes cannot be neglected, because they are several meters long. The valves adjust the pressure of the lower part by very fast actuation. This introduces very fast pressure changes and the pressure wave propagate with the speed of sound which is approximately 1300 m/s. The result may be oscillations in the range 400-500 Hz generating vibration and noise within the automobile system. See Figure 4. It is important to analyze this phenomenon in order to develop ways of reducing the vibration and noise.

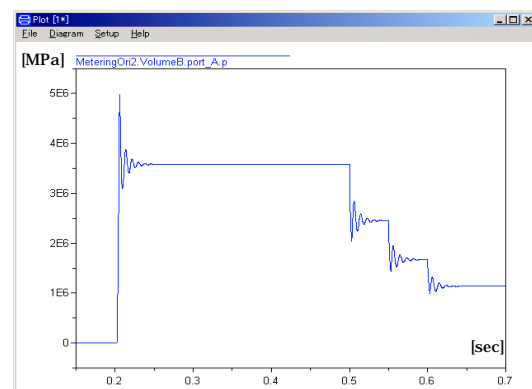


Figure 4: Pulsation in the long hydraulic line

The two pipes of the model are each 3 meters long. Each pipe is modeled by 15 segments, where each segment is 0.2 m long and has 5 continuous time states. The model has in total 164 states.

It is not possible to use explicit Euler for real-time simulation, because the model requires the step size to be less than 0.001 ms to ensure the stability of the simulation.

At the third order implicit inline Runge-Kutta integration method, RK3, with a step size of 0.5 ms was used to obtain both stability and desired accuracy. The symbolic manipulation of Dymola reduces the size of the nonlinear system to be solved by a nonlinear solver from 164 to 10.

Table 5 shows the performance results for the Pentium II processor when simulating 1 second.

Table 5: Performance for inlined implicit RK3 when simulating the hydraulic actuator model for 1 s using an Intel Pentium II 350 MHz processor.

	Integration Method	Step [ms]	CPU Time [s]
Dymola	Explicit Euler	0.001	283
	Inlined Impl RK3	0.5	3.1
Simulink	Explicit Euler	0.001	253
	Inlined Impl RK3	0.5	3

It should be noted that the inlined implicit RK3 gave a speed up of a factor of 91. However, the simulation is 3.1 times too slow to run in real-time. Therefore Dynasim made corresponding test shown in Table 6 for Pentium 4, 1.6 GHz. Real-time performance was then achieved.

When attempting to test on xPC, the diagnostic 'Failed to download' was received. More investigations have to be made to determine the cause.

Table 6: Performance for inlined implicit RK3 when simulating the hydraulic actuator model for 1 s using an Intel Pentium 4, 1.6 GHz processor.

	Integration Method	Step [ms]	CPU Time [s]
Dymola	Explicit Euler	0.001	59.1
	Inlined Impl RK3	0.5	0.71
Simulink	Explicit Euler	0.001	50.9
	Inlined Impl RK3	0.5	0.6

These results show that the improved implicit inline integration method is very effective in this case. Conventionally it was difficult to handle hydraulic models using fixed step explicit Euler and process the model within a practical calculation time. Using the improved implicit inline integration method it is possible to handle hydraulic models and process them at high speeds even in fixed step situations.

This characteristic is desirable for real time simulation. Implicit inline integration produced a remarkable improvement for this simple hydraulic model. Furthermore, it has also shown considerable effectiveness even for models that are more complicated. The representative characteristics of this method are as follows.

- This method enables the generation of a code which is suitable for real-time simulation, even in the case of models that have the property of oscillation, such as the hydraulics system.
- This method makes the handling of models with larger step size possible.

4. Conclusions

We evaluated inline integration and mixed mode integration which were developed to improve calculation performance. Improved implicit inline integration, which has recently been developed, was also evaluated. It was confirmed that these methods effectively increased the ability to handle models in real time. In particular the improved implicit inline 3rd order integration method could handle the model very efficiently even if it was a hydraulic model.

On the other hand, it was generally the case that setting step size for fixed step size integrators remained a problem. Even inline integration needs trial and error testing to find a suitable step size. In addition, an increasing number of modeling beginners are using these kinds of tools. These users are not always experts in modeling or control. It would therefore be desirable if they could more easily obtain improved performance using the techniques we evaluated. We hope that such method will be established and be applied to the physical models, which were made for controller designing in order to enable HILS.

The improved implicit higher order integration method has given a break-through in simulation speed for stiff models and for discretized partial differential equations originating for example in long hydraulic pipes.

References

- [1] Dymola. *Dynamic Modeling Laboratory*, Dynasim AB, Lund, Sweden, <http://www.Dynasim.se>.
- [2] P. Beater, “*Modeling and digital simulation of hydraulic systems in design and engineering education using Modelica and HyLib*”, Modelica Workshop 2000 proceedings pp 33-40
- [3] H. Elmqvist, F. Cellier, M. Otter: *Inline Integration: A new mixed symbolic/numeric approach for solving differential-algebraic equation systems*. Proceedings European Simulation Multiconference, June 1995, Prague, pp: XXIII-XXXIV.
- [4] H. Elmqvist, S.E. Mattsson, H. Olsson: *New Methods for Hardware-in-the-loop Simulation of Stiff Models*, Proceedings of Modelica Conference 2002. Modelica homepage: <http://www.Modelica.org>.
- [5] Modelica. Modelica homepage: <http://www.Modelica.org>.
- [6] A. Schiela, H. Olsson, “Mixed mode integration for Real-time Simulation”, Modelica Workshop 2000 proceedings, pp 69-75
- [7] S. Soejima, “Examples of usage and the spread of Dymola within Toyota”, Modelica Workshop 2000 proceedings, pp 55-59

HIL-Simulation of the Hydraulics and Mechanics of an Automatic Gearbox

Clemens Schlegel

cs@schlegel-simulation.de
Schlegel Simulation GmbH
Munich, Germany

Marco Bross

Marco.Bross@bmw.de
BMW AG
Munich, Germany

Peter Beater

Beater@mailso.uni-paderbon.de
Universität -GH Paderborn
Soest, Germany

Abstract

In this article, hardware-in-the-loop (HIL) simulation of a passenger car automatic gearbox is discussed. The simulation includes detailed models of the mechanics and hydraulics and less detailed models of the other parts of the car's drive train like its engine, torque converter, differential gearbox, chassis and driving resistances. After a short description of the components to be modeled, special issues of simulating variable-structure mechanical systems (coupled frictional elements), simulating hydraulics and simulating in real time with the gearbox control electronics hardware in the loop are discussed. A simulation based, detailed assessment of the dynamics of the gearbox hydraulics show that it might be modeled (under certain assumptions) with fixed causality without major loss of accuracy. Therefore nonlinear systems of equations in the hydraulic parts of the model can be avoided. This enables the usage of a model based on hydraulic component submodels, rather than on overall global dynamics to be used for real time simulations with standard HIL-simulation hardware. The article ends with a short discussion of HIL-simulation results and an outlook on future work.

1. Introduction

The motivation to realize tests in a HIL-environment is manifold, but two main reasons are:

Shorter development time. The time available for the development of new components and cars is becoming shorter and shorter. Thus, a lot of time has to be saved during the development phase. HIL-

simulation and testing is a possibility to achieve this, as

- There is no need to wait for prototype production, if the data of these are available for modeling,
- No driver and test circuit is needed,
- Test conditions can be reproduced precisely,
- Tests can even be automated.

Rising complexity due to interacting electronic control systems. Cars have always been aggregations of several subsystems like engine, gearbox, brakes and so forth, and thus showed a certain complexity. But in former times those systems worked rather independently and could therefore be developed and tested separately. Nowadays the subsystems of passenger cars are strongly interdependent:

- Different control systems act on the same dynamics (e.g. both motor management (DME) and gearbox controller (EGS) influence the longitudinal dynamics (fig. 1).
- Different control systems share sensor information that is exchanged via CAN bus for control purposes, but also for self-diagnosis.
- Functions are spread over several controllers.

As a consequence systems can no longer be tested separately and the number of different error cases that have to be tested increases drastically. The test environment has to include all essential parts or functionalities of all interacting systems. Optimal testing should be automated in order to handle the number of error cases. Both requirements lead to automated HIL simulation and testing.

This article describes the test environment that was installed at BMW in order to test the control system

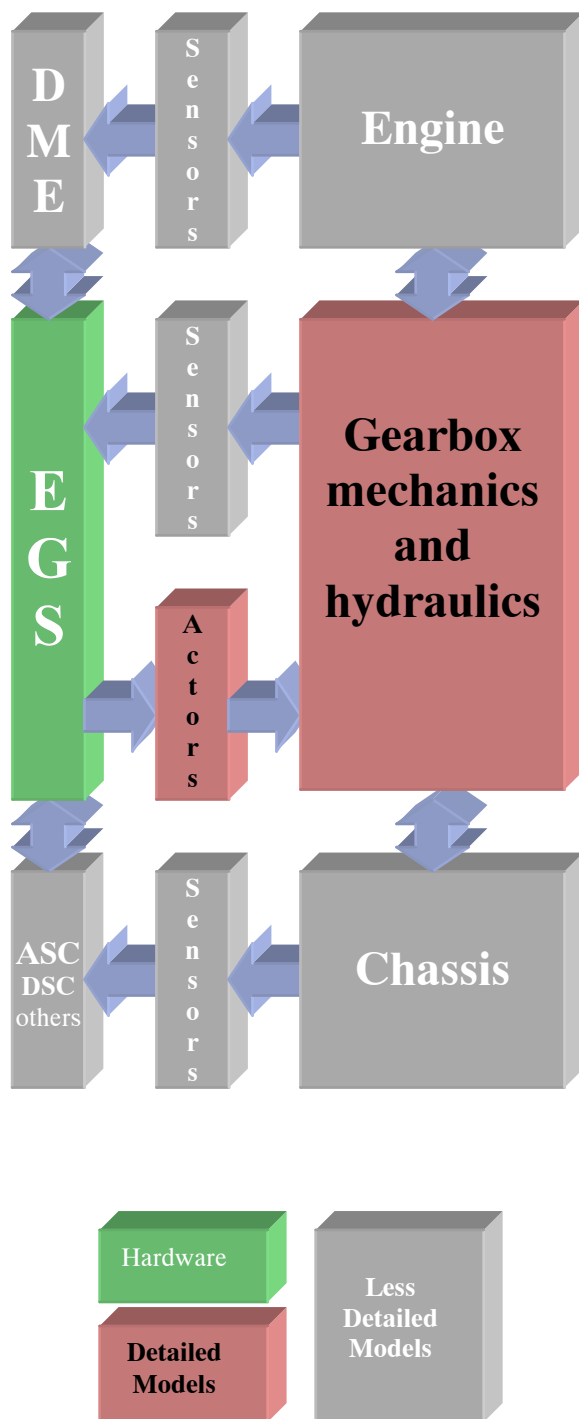


Figure 1: System overall view

(EGS) of the automatic gearbox. For the above mentioned reasons, it was not sufficient to model only the gearbox itself that is controlled by the EGS, but also the remainder of the powertrain and parts of its controllers and communication structures. Figure 1 gives an overview of the components, physical interactions and information flow:

- The EGS represents the hardware in the loop and is the item under test. All other parts are simulated.
- Gearbox mechanics, hydraulics and actuators have been modeled in detail. This was necessary, as one of the goals of this setup was the possibility to simulate the effects of failure of one of the actuators or the hydraulic valves.
- The less detailed models contain only those functionalities that are necessary for the simulation, e.g. the model of the DME does not control a full model of the engine, but is necessary to transmit the required signals via CAN bus to the EGS.

2. Modeling driveline and gearbox mechanics

An automatic gearbox can be simulated only if the input and output torques or speeds are known. Therefore, at least the engine and the longitudinal dynamics of the vehicle also have to be modeled. Figure 2 shows a corresponding model: Engine (controlled by a control unit and a driver model), torque converter, gearbox, final drive, brake wheels, vehicle inertia and driving resistances. The engine is modeled by a torque map, the torque converter by static characteristics, and all other components, apart from the gearbox, by the well known physical relations.

Figure 3 shows an outlined sketch of the 5 speed gearbox ZF 5HP24 [1] which was investigated. Apart from the hydrodynamic torque converter it consists of three planetary wheel sets and seven switching elements: Three clutches (A, B, C), three brakes (D, E, F), and a freewheel (FF). The gearshift pattern (fig. 4) indicates which switching elements have to be active to engage a certain gear.

If appropriate component models are given, the object-orientation of Modelica allows to derive the complete simulation model (fig. 5) easily from the gearbox scheme of figure 3. For the component models the standard Modelica library “Mechanics.Rotational” [2] and the Modelica powertrain library [3] have been used. For more details of modeling automatic gearbox mechanics see [4].

Clutches, brakes and freewheels in a simulation model result in a variable structure system, this is because two shafts can stick or slip relative to each

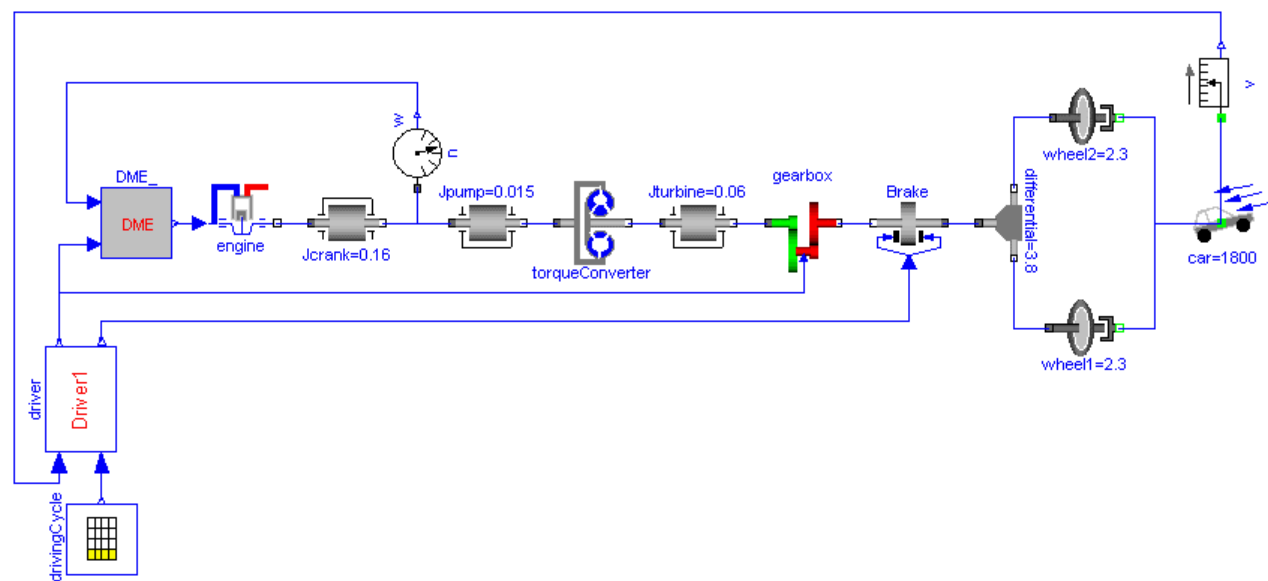


Figure 2: Drive train simulation model

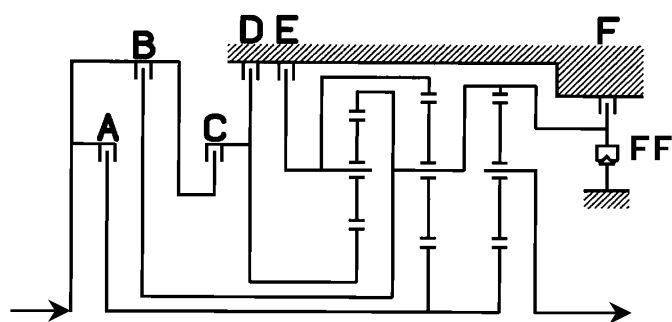


Figure 3: Outlined sketch of 5 speed automatic gearbox ZF 5HP24

Gear	A	B	C	D	E	F	FF
R			x			x	
N						x	
1	x						x
2	x				x		
3	x			x			
4	x	x					
5		x		x			

Figure 4: Gearshift pattern

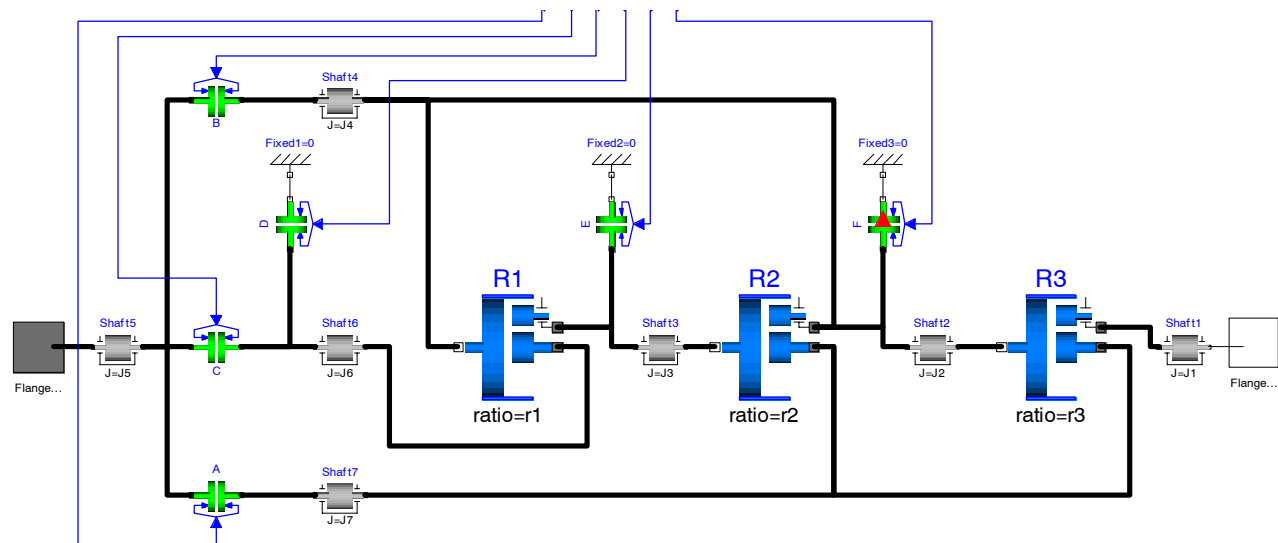


Figure 5: Gearbox mechanics simulation model

other. The number of states is changing during a transition from stick to slip and vice versa. Neglecting some “fast” dynamics in order to reduce simulation time results in a typical idealized friction characteristic shown in figure 6. The friction torque is a discontinuous and in part non-unique function of the relative speed of the clutch disks. Therefore additional equations have to be set up for a complete system description.

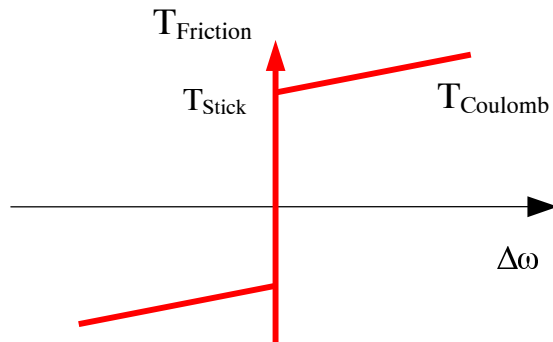


Figure 6: Idealized friction characteristic

In the Modelica libraries used, friction is modeled in a parameterized form (in contrast to [4]) with a curve parameter included plus a state machine describing the transitions between the unique and non-unique parts of the idealized friction characteristic. Because the relative speed in the clutch is an output of the integration algorithm and computed with a limited precision only, finding the transition between the unique and non-unique parts of the friction characteristic is not trivial. This holds especially for systems with several interacting clutches, like the system treated here.

Modeling a clutch by a parameterized friction description in connection with a state machine results in a mixed system of discrete and continuous equations, which cannot be solved by standard methods like Gaussian elimination. There are a few methods to solve such mixed systems [5], all of them need iteration at an event instance (transition from stuck to sliding mode and vice versa). Using Dymola [6] for processing of the Modelica models, these iterations proved to converge quite quickly. Therefore the real-time condition was met in the HIL setup with only a few exceptions.

3. Modeling gearbox hydraulics

The hydraulic system of an automatic gearbox consists of different elements with the following functions:

- Electro-hydraulic elements provide a hydraulic pressure as a function of the electrical current flowing through the element.
- Switching valves open or close canals.
- Proportional valves amplify pressures and / or transform hydraulic impedances.
- Cylinders generate a normal force on a clutch pack if a hydraulic pressure is applied on them.

Figure. 7 gives an overview over the elements and their interactions. In the following section a short outline of modeling techniques for hydraulic systems is given.

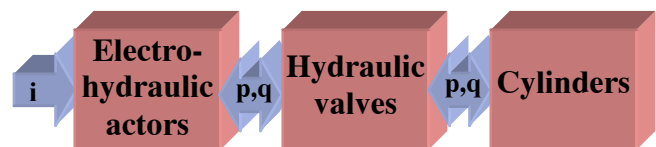


Figure 7: Interaction of hydraulic subsystems

The early simulation languages were block-oriented [7] and emulated analog computers. They were very well suited for the simulation of control systems where the output signal of a control block doesn't influence the input. Hydraulic systems, however, work differently: The state at the input port of a component is dependent on the state of the output port. A hydraulic line illustrates this: If the line is closed at the end the pressure at the entrance will rise according to the input flow rate. If the line is open at the end the pressure at the input will fall almost to atmospheric pressure. These dependencies can be modeled with block-oriented software but lead to awkward models because of the necessary feedback loops. It is very difficult to build modular models with this approach.

Modelica enables *acausal* modeling, i.e. it is possible to describe the behaviour of a component without defining which variables are input and which are output variables. As a consequence it is possible to use the same library model for a hydraulic pump (input is the mechanical power, output the flow rate) and a hydraulic motor (input is the hydraulic power,

output the torque at the shaft). This object oriented modeling approach thus resembles the design strategies of component manufacturers: They use (to a great extent) the same parts for pumps and motors. [8].

Hydraulic systems can be described by differential-algebraic equations (DAE). The differential equations are usually non-linear first-order equations that model the pressure build up in lumped volumes. Only special cases require partial-differential equations (PDE) to describe the behaviour of long lines. Usually these PDEs are discretized to arrive at a system of first order ODEs.

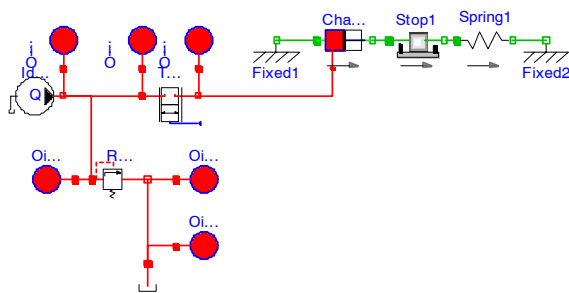


Figure 8: Modeling approach using lumped volumes.

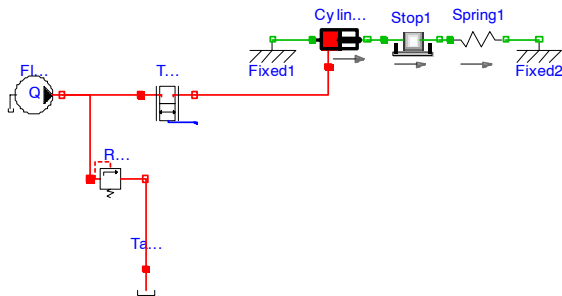


Figure 9: Library models; the lumped volumes at the ports are included but not shown in the icons.

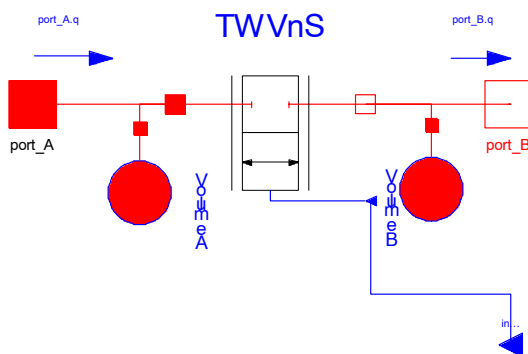


Figure 10: Diagram layer of library valve model with included volumes at the ports shows more details.

For standard applications it has proven very helpful to place a lumped volume at each port of a component to model the behaviour of the compressible oil (fig. 8). This leads to a simple structure of the resulting DAE-system. However to be able to solve this DAE with standard solvers it is necessary to reduce the index. In former times this was done by hand from the modeling engineer by adding the amount of oil of all components connected at a particular node, nowadays it can be done automatically by the tool.

To avoid the manual placement of volumes and the resulting cluttering of the diagram layer library models are available that have already included the lumped volumes at the ports but don't show them in the icons. The resulting diagram layer is almost identical to a standard hydraulic circuit diagram (fig. 9 + 10). It can therefore be read also by engineers with training in hydraulics but no deeper experience in modeling and simulation [9].

When modeling hydraulic systems it makes sense to follow the path of the oil: The source is the pump, the sink is the tank, the cylinders, motors and valves are in between. Using an appropriate library even complex circuits can be modeled in a short period of time if the required parameters of the components are known [10].

The advantages of the outlined concept are obvious. Hydraulic components can be modeled in a truly modular way. They can be arranged in an arbitrary structure – parallel or in series. The resulting nonlinear DAE system can be solved for the derivatives of the state variables thus avoiding the numerical solution of systems of nonlinear equations. There are however also some drawbacks. The lumped volumes between components can become very small, they may contain less than a thimble full of oil. As a consequence the pressure builds up very rapidly. In mathematical terms this means a *stiff system* that has eigenvalues near the origin and almost at minus infinity. Using advanced integration algorithms with automatic step size control these DAEs can be solved successfully but the required computing time will usually be greater than the simulated time. Considerations of the numerical stability will restrict the permissible step size for fixed step-size algorithms that are used for HIL simulations.

One way to reduce the required computing time is the observation that not all pressure states (lumped volumes) are significant for the overall behaviour of the model. In that case it is possible to eliminate a

state. As an example figure 11 shows two orifices in series.

If the pressure dynamics of the lumped volume between the two orifices is not significant one can neglect it and assume that the flow rate through both orifices is identical. It is then possible to calculate the flow rate through both orifices as a function of the pressure differential across both orifices. This approach is identical to the assumption of a zero volume.

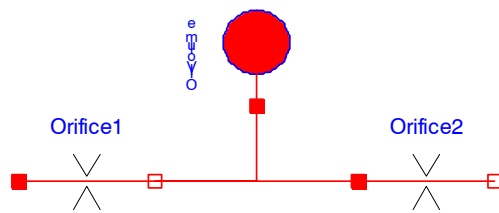


Figure 11: Two orifices in series.

In general, using these techniques, one has to find a compromise between placing a lumped volume at each connector and not using them at all. The first approach avoids nonlinear systems of equations, but generates a stiff system. The second approach does not generate a stiff system, but the resulting system of nonlinear algebraic equations has to be solved numerically. Thus, both approaches will lead to long simulation times (compared to simulated time), the optimum is a combination of both.

Unfortunately, using this method simulation times are still far from real-time using a standard HIL simulation processor (we used a Motorola PowerPC 750 processor running at 480MHz). Thus, another simplification has to be made. Detailed analysis of the hydraulic system shows that it is possible to use a causal approach for some elements: For the majority of the valves, the generated pressure of one valve can be considered to be independent of the valve that is driven by that pressure, as the volume flow of oil is usually small. Thus, a model can be derived from an acausal model where the majority of the elements is modeled in a causal way, which speeds up simulation times to an extent that real-time simulation becomes possible.

4. Gearbox electronics & HIL

After having combined all necessary simulation models (all subsystems shown in fig. 1 apart from the gearbox controller EGS), they have to be implemented on an appropriate real-time processor together with all interfaces needed. For the Modelica implementation of the gearbox mechanics model, we used Dymola and exported the processed model as a Simulink S-function [11]. The fixed causality hydraulics model and the software interfaces to the hardware have been implemented in Simulink too. Since the gearbox controller provides no trigger signal the simulated plant model has to be sampled much faster than the controller. The EGS under test operates at 100 Hz, requiring a sampling rate of 1 kHz for the simulation model. For the real-time simulation hardware we used boards by dSPACE [12].

Setting up a HIL simulation often non-standard interfaces are needed due to I/O reversal: Sensors and actuators are simulated, but they interface in part directly to the power-electronics part of the control unit which needs the respective electric loads for proper operation. In contrast, standard real-time I/O interfaces provide TTL-level signals only.

The EGS senses the speed of the gearbox input- and output shafts and oil temperature. Based on these signals (interfaced directly) and other signals like vehicle speed, throttle position, and estimated engine torque (interfaced indirectly via CAN bus), the actual gearshift is performed according to a shift map and a set of parameters adjusting the slope of the hydraulic forces acting on the respective clutch packs to the actual driveline and vehicle state. During a gearshift the EGS may require via CAN bus the engine controller to reduce engine torque for a smooth transition.

On the output side the EGS interfaces directly to electro-hydraulic components of the gearbox. The respective original parts are included in the HIL setup to provide proper electrical loads. That parts are combined in a load box which may be exchanged for simulation of another automatic gearbox type. Without proper electric loads at the power-electronics interfaces the EGS would operate in emergency mode only (4th gear, no gear shift) due to implemented watchdog functions. For the same reason health monitoring signals of other controllers have to be provided via CAN bus, too.

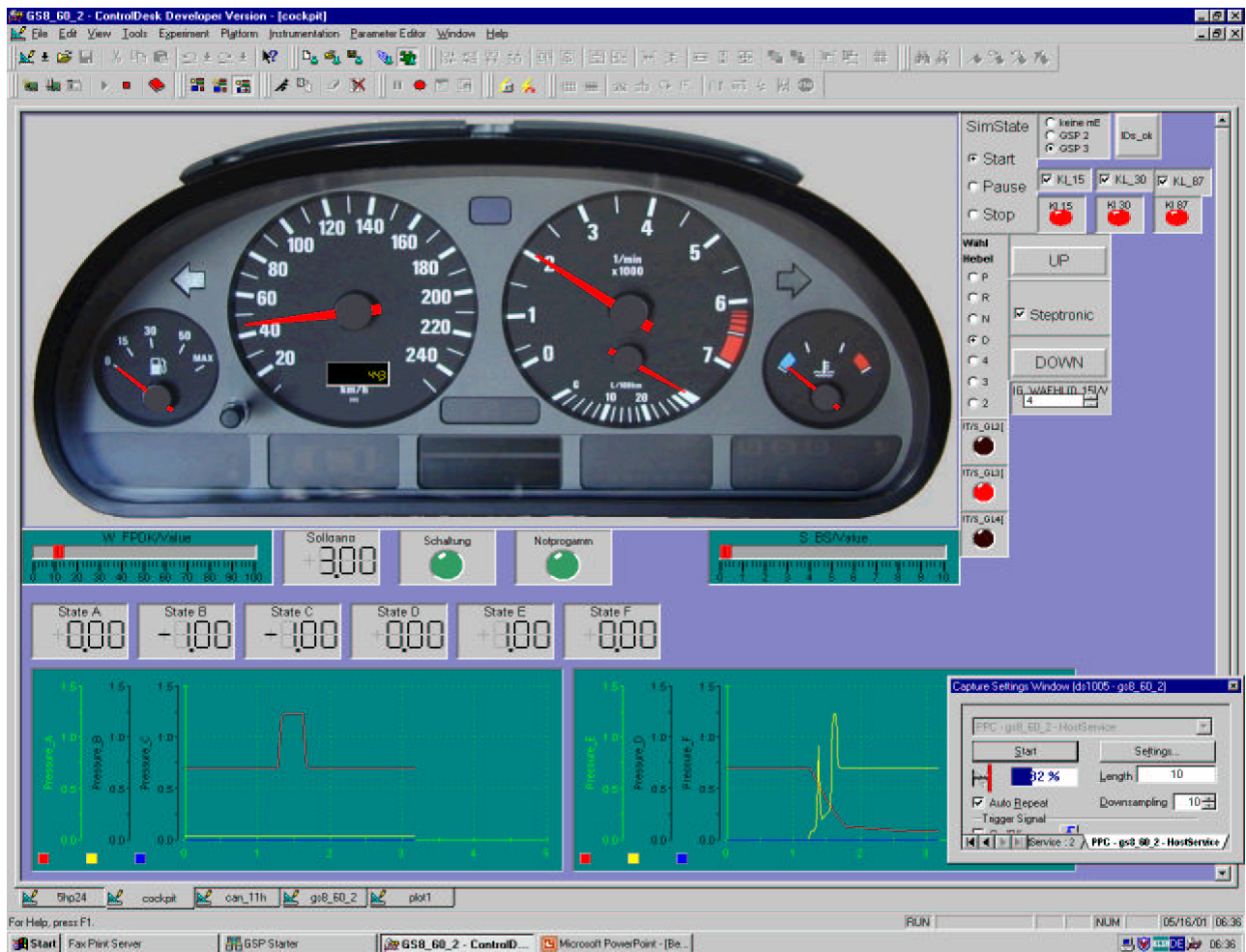


Figure 12: HIL simulation control main panel

For the operator interface to the simulation we used the board vendors software ControlDesk [12]. Figure 12 shows the main panel with standard passenger car instrumentation, gearshift control, simulation control, and simulation output of the actual state and the pressure history of all six clutches of the gearbox.

With the HIL setup described the effects of partial or total failure of one or more mechanic, electric, or hydraulic components of the gearbox can be studied in detail. For interfacing to the EGS software, e.g. for changing parameters, disabling certain parameter adaptation functionalities, etc. an additional device is needed. We used INCA [13] for that task.

5. Simulation Results

The following simulation results show the hydraulic pressure (in N/mm^2) for two cylinders as a result of

two gear shifts. Until $t = 1\text{ s}$, the neutral gear is engaged. Then, the first gear is engaged, and the gear-box switches to the second gear at $t = 3\text{ s}$. Figure 13 shows the simulation results for the acausal model, simulated with Dymola. Figure 14 shows the same, but the results are based on a causal model with the same parameters.

The results for both models are fairly similar, proving the assumption to be correct for most of the time. This is not the case for the pressure in cylinder A around $t=3.5\text{ s}$ (red circle). In the acausal, precise model, the pressure in A falls slightly, because cylinder E gets filled by a considerable volume flow. Thus, the working pressure drops, which is also reflected in the pressure in cylinder A. As it can be expected, the causal model does not show this effect.

Figure 15 shows the influence of a EGS parameter modification (application parameter). The result represents an uncomfortable gear shift, as the pres-

sure in cylinder E shows a peak (blue circle). The fact that changes in these parameters are reflected in the pressure buildup opens the possibility to use these models for application purposes, too.



Figure 13: Simulation results: Acausal model

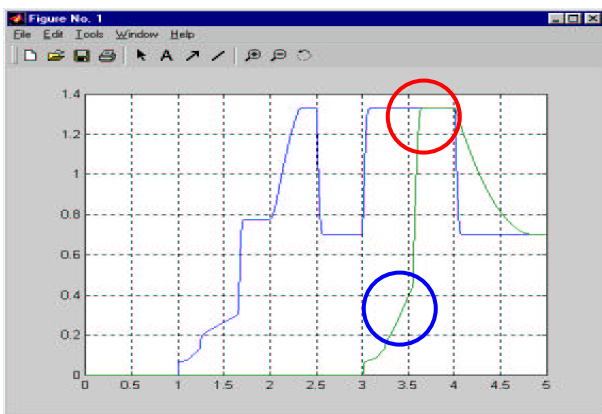


Figure 14: Simulation results: Fixed causality model

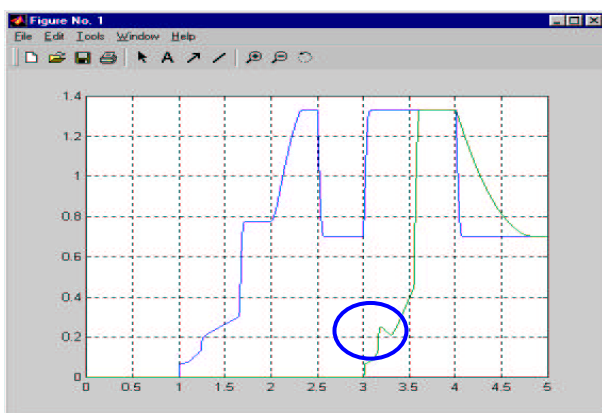


Figure 15: Simulation results: Effects of poor application parameters.

6. Conclusion & Outlook

Using the available component models of Modelica, quite detailed models of gearbox hydraulics and mechanics have been developed. Further investigation showed the possibility to model the gearbox hydraulics in part with fixed causality, which allowed real-time simulation of both hydraulics and mechanics. This model was implemented on a HIL environment together with the gearbox controller. For fully automated component failure tests of the EGS the respective models have to be enhanced by failure injection inputs.

The fixed causality hydraulics model may also be implemented in Modelica. This would enable to split up the combined mechanics and hydraulics model in “slow” and “fast” parts and thus using the potential advantage of Dymola’s inline integration scheme [14]. A limitation may be that the presumable “slow” mechanic parts of the model need “fast” sampling too, in order to meet the real-time condition if iterations occur at an event instance in the clutch models.

An other area of future investigation might be the use of simulation models for application purposes. This creates the need for further improvement of the models without loss of simulation speed. Since only a limited set of signals are available for measurement with reasonable effort, setting up procedures for identification and validation of those refined models needs to be addressed.

References

- [1] Funktionsbeschreibung Automatikgetriebe 5HP24. ZF Getriebe GmbH, Saarbrücken
- [2] Modelica Association. Modelica.Mechanics.Rotational, <http://www.modelica.org/library/library.html>
- [3] PowerTrain library, <http://www.dynasim.se>
- [4] M. Otter, C. Schlegel, H. Elmqvist, Modeling and Realtime Simulation of an Automatic Gearbox using Modelica, 9th European Simulation Symposium ESS’97, Passau, Germany, Oct. 19.-22., pp. 115-121, 1997.
- [5] M. Otter, H. Elmqvist, S.E. Mattsson, Hybrid Modeling in Modelica based on the

Synchronous Data Flow Principle. IEEE International Symposium on Computer Aided Control System Design, Hawaii, August 22-27, USA, Proceedings of CACSD'99, S. 151-157, 1999.

- [6] Dymola, <http://www.dynasim.se>
- [7] *J.C. Strauss, D.C. Augustine, B.B. Johnson, R.N. Linebarger, F.J. Sanson*,. (1967) The SCI Continuous System Simulation Language (CSSL). *Simulation*, IX(6)281-303, 1967.
- [8] *P. Beater*, Entwurf hydraulischer Maschinen – Modellbildung, Stabilitätsanalyse und Simulation hydrostatischer Antriebe und Steuerungen. Berlin, Heidelberg, New York, Springer Verlag. 1999.
- [9] *P. Beater*, Modeling and Digital Simulation of Hydraulic Systems in Design and Engineering Education using Modelica and HyLib. Lund, Modelica Workshop 2000, pp 33 – 40, 2000.
- [10] HyLib. Library of Hydraulic Components
<http://www.hylib.com>
- [11] Simulink, <http://www.mathworks.com>
- [12] <http://www.dspace.de>
- [13] <http://www.etas.de>
- [14] *A. Schiela, H. Olsson*, Mixed-mode Integration for Real-time Simulation. Lund, Modelica Workshop 2000, pp 33 – 40, 2000

Modelica Applications for Camless Engine Valvetrain Development

**Christopher Puchalsky, Thomas Megli, Michael Tiller, Nate Trask,
Yan Wang, Eric Curtis**
Ford Motor Company

Abstract

Several variable valvetrain technologies are being aggressively pursued to increase vehicle fuel economy and reduce engine exhaust emission levels. Electromechanical Valve Actuation (EMVA) is a promising alternative that uses electromagnetic actuators to replace the conventional camshaft and provide fully flexible valve timing control. This "camless" valvetrain provides new opportunities and challenges for engine control optimization. In this work, we present two Modelica applications for EMVA development.

Control and prediction of the Air to Fuel (A/F) ratio in a port fuel injected spark-ignited (PFI SI) engine is an important factor for emissions, performance, and fuel economy. A Modelica model to simulate the dynamic behavior of fuel vaporization and storage inside a PFI SI engine has been developed. This "wall wetting" model was developed from an existing FORTRAN based model and employs several control volumes to represent fuel in various phases and locations in the engine. A multi-component fuel model (i.e. containing different constituents with a wide range of molecular weights) is used where the fuel component masses are the state variables and the mass flow rates are the flow variables. The fuel model can be easily re-declared so that different numbers and types of fuel components can be used to simulate the distillation characteristics of various fuels. For the control volumes that represent liquid fuel puddles, the connectors have additional information such as puddle area, puddle height, fuel component vapor pressure, puddle temperature, and puddle heat transfer. The processes of fuel injection, vaporization, liquid flow, and shattering are used to move fuel between the various control volumes. The Modelica model can be coupled by various degrees to engine simulation models. By comparison, in the original FORTRAN model, engine operating inputs to the wall wetting model were made by rough approximation with no opportunity for feedback from the wall wetting model to affect the operating conditions. In this application we fully couple the wall wetting dynamics to a single cylinder engine model. The complete model is then more generally applicable to the increased number of degrees of freedom afforded by the variable valve

timing control. The engine model incorporates a simple valve actuator model to replace the conventional camshaft motion with the flexible timing and transition characteristics of EMVA. The engine model predicts gas flows, temperatures, and pressures that were inputs to the FORTRAN wall-wetting model. The wall wetting model then determines the fuel vaporization rate, which in turn determines the A/F ratio input to the engine model. This subsequently changes the temperatures, pressures, and flows in the combustion chamber and port sub-models. Initial comparison of results to the FORTRAN model show reasonable agreement in A/F prediction but the FORTRAN version currently runs faster.

The other use of Modelica involves actuator development. Actuator design and control is a significant challenge for EMVA engines. To achieve performance, durability and fuel economy objectives, valve motion must be carefully controlled via electromagnets to achieve both fast transitions and low contact velocities. The actuator system must also be designed to minimize electrical power consumption. A detailed actuator model is developed to study valve transition characteristics. The model incorporates mass, spring, and electrical elements from the Modelica standard translation and electrical sub-libraries. A detailed sub-model of a solenoid with an "E-shaped" core has been developed to predict magnetic forces and inductive characteristics. The magnetic force is coupled to a reciprocating mass which represents the armature and valve assembly. Various actuator design modifications have been investigated. The effect of a simple voltage control scheme on valve motion is investigated here.

Introduction

The global automotive industry is under increasing pressure from governmental, consumer, and non-governmental groups to improve the fuel economy of motor vehicles. Reasons for improvement range from concerns about global warming to the need to reduce the dependence on foreign, and often volatile, petroleum sources. Consumer demand and competitive

forces demand that improvements in fuel economy not be accompanied by decreases in other metrics of vehicle performance – safety, power, interior space, emissions, price, and NVH. It is often required that these other metrics improve along with fuel economy.

One group of technologies that holds promise for improving fuel economy while maintaining or improving most other areas of vehicle performance is variable valve timing (VVT). VVT reduces or eliminates many of the tradeoffs between low and high speed torque, fuel economy, idle quality, and emissions that are currently made with fixed valve timing. VVT includes current production technologies like variable cam timing, cam switching, and variable valve lift. All of these technologies use a cam to open and close the valves. A new VVT technology that holds promise is Electromechanical Valve Actuation (EMVA). EMVA uses electromagnets to open and close the valves. The valve timing is then independent of crankshaft position, and valve opening and closing times can be optimized to reduce throttling losses and to control residual gas fractions. Additionally, valves may be deactivated to reduce power consumption or to deactivate cylinders for improved fuel economy.

EMVA presents several engineering challenges in which modeling plays an important role. One such area is the development of strategies for transient air fuel control. We will discuss the development of the plant model to predict liquid fuel dynamics. Another engineering challenge is the development of the electromagnetic actuator. Both will be discussed and results will be presented.

Wall Wetting Model Development

Prediction of transient fuel dynamics is difficult with conventional port fuel injected (PFI) engines running at fixed valve timing. In PFI engines, a fuel injector is placed in the intake port as close to the intake valve as packaging will allow. A schematic of the fuel injection and wall wetting process in a standard camshaft engine is shown in Figure 1. Fuel is injected towards the intake valve and port walls just before intake valve opening. Some fuel becomes entrained in the air stream, but most lands on the valve and port where it forms small "puddles". The fuel evaporates off the hot port walls (~95 C) and the hotter intake valve (~175 C). The fuel, gasoline, is composed of many different chemical species with widely different characteristics. The lighter, more volatile, components will evaporate easily while the heavier, less volatile, components will tend to evaporate slowly and stay in the puddle. The evaporation rate increases dramatically when the intake valve is open and the air speed in the port is high. The high air speed in the port also produces a forward flow phenomenon, which causes some of the liquid fuel on the port walls and the intake valve to be sucked into the

combustion chamber where it forms a puddle. Additionally, right at the moment of intake valve opening, the pressure in the intake port is much less than that in the cylinder. This produces a backflow pressure wave that splatters some of the fuel off of the valve and up into the port. In both the forward and backward flow processes some of the fuel is entrained in the air stream before it lands.

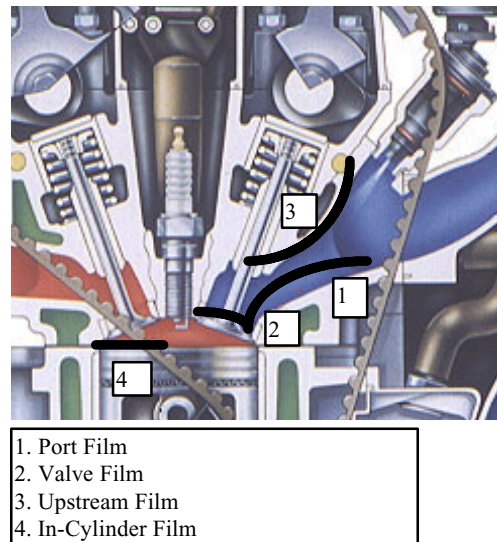


Figure 1: Schematic of Fuel Injection.

Prediction of transient air fuel dynamics becomes even more difficult under certain EMVA engine operating modes (*e.g.* late intake valve closing (IVC), alternating valve closing, and cylinder deactivation). A detailed wall wetting model to predict transient air fuel dynamics has been created in Modelica. A starting point for the Modelica model was a FORTRAN model developed by Curtis, *et. al.* [1]. The FORTRAN wall wetting model contains models of all of the processes described above, all of which have also been implemented in the Modelica version. Additionally, the Modelica version is also tied to an engine cycle simulation that provides data such as air speeds, pressures, and temperatures to the wall wetting model. The FORTRAN version used approximations for this data.

Basic Models

In the most basic form, the wall wetting model is a collection of fuel puddles (control volumes) linked together by processes that move fuel between the various puddles. This is similar to modeling in the thermal domain where a series of thermal capacitances

exist with thermal resistances and convective elements to move the thermal energy between them.

The control volume model has the following connector instantiated as `cv` (control volume):

```
connector MassConnector
  import Modelica.SIunits;
  parameter Integer n "# of species";
  parameter String FuelNames;
  SIunits.Mass m[n];
  flow SIunits.MassFlowRate mdot[n];
end MassConnector;
```

The control volume model also contains the following equation to link the flow and across variables:

```
der(cv.m) = cv.mdot;
```

A multi-component fuel model is used. The number, type, and injected mass fraction of each component (species) is selected to match the distillation characteristics of the fuel. There are 21 chemical species from which to select. The fuel model contains both fuel composition and material property data. It has the following code:

```
model Fuel
  extends FuelIcon;
  replaceable Two_Component_Test
    Fuel_Comp ;
  FuelsDataAdjustable
    Data(Fuel_Comp=Fuel_Comp) ;
end Fuel;
```

The replaceable `Two_Component_Test` model defines the fuel. This model contains the injected mass fractions, the fuel names string, and an array of integers that specifies which components are used. This information is then passed to the `FuelsDataAdjustable` model. The `FuelsDataAdjustable` model extracts the material property data for the used species from the list of possible species. `FuelsDataAdjustable` is implemented as a model and not as a record because of an assert statement in the equation layer of the model. This forces `Fuel` to be a model because one of its used classes is a model and not a record. Knowing which fuel, and hence which species, will be used at translation time decreases the number of variables and run time.

Note that the connector definition has a string parameter `FuelNames`. The string `FuelNames` is a concatenation of abbreviations for the names of all the various fuel components that make up the current fuel model. An example of `FuelNames` for the indolene fuel model is: "ispnt|ioctn|tolun|ndecn|cyhex|naph|ethylb". This is on the connector to assure

that all the parts of the model are using a consistent fuel model.

The control volumes used to represent the liquid fuel are placed inside a wrapper model. The wrapper model contains the fuel model and a thermal connector that is connected to a thermal model that predicts the temperature of the puddle. It also has two models that calculate the surface tension and the vapor pressure of the fuel mixture in the puddle from the puddle temperature and fuel properties. The geometry (area, height, perimeter) of the liquid puddle is calculated in the equation layer of the model. It also has a liquid fuel connector that is similar to the control volume connector:

```
connector LiquidMixture
  import Modelica.SIunits;
  parameter Integer n "# of species";
  parameter String FuelNames;
  SIunits.Mass m[n];
  flow SIunits.MassFlowRate mdot[n];
  SIunits.Pressure Pv[n];
  SIunits.Temperature T;
  flow SIunits.HeatFlowRate q;
  SIunits.DynamicViscosity mu;
  SIunits.SurfaceTension SurfTen;
  SIunits.Area A;
  SIunits.Height H;
  WallWetting.Types.Perimeter Pwet;
end LiquidMixture;
```

The liquid puddle model and liquid mixture connector allow all of the information pertinent to the puddle to be calculated in one place. This prevents, for example, both the evaporation and forward flow models from calculating the puddle geometry.

The FORTRAN version used liquid puddle models to represent the liquid fuel puddles – one on the intake valve, one on the cylinder, and two in the port (see Figure 1). The fuel in the port is split into two puddles – one downstream in the port near the valve and one upstream away from the valve and close to the injector. The downstream puddle is nominally hotter than the upstream port. The modular nature of Modelica permitted easy creation of two different models with different numbers of control volumes. One is identical to the FORTRAN wall wetting model with 4 puddles. Both 4-puddle models can be used to model engines with multiple intake valves, bifurcated and non-bifurcated ports, and charge motion control valves by the use of multipliers. For example, an engine with two valves and a bifurcated port with fuel injected evenly into both ports would have the amount of fuel injected divided by 2 and the amount of fuel vaporized multiplied by 2.

To model liquid fuel dynamics with EMVA it was necessary to use 7 control volumes because each cylinder has two intake valves and a single fuel injector. Only one cylinder puddle control volume was used, but all of the other puddles were doubled to represent the two intake valves. The use of 7 control volumes instead of 4 with multipliers was necessary because of certain EMVA modes that are non-symmetrical. One such mode occurs when the intake valves open on alternating cycles, but the single fuel injector sprays fuel into both ports on each cycle.

The FORTRAN version of the wall wetting model had a control volume to keep track of the fuel evaporated. Some versions of the Modelica wall wetting model have a separate control volume to keep track of the fuel that has been vaporized. Others simply convert the multicomponent evaporation mass flow rate into a single component mass flow rate that can be applied directly to the medium connectors that Ford uses for cycle simulation [2].

In addition to the fuel model, four different records are used to pass information to different sections of the model. They are passed down the hierarchy as replaceable records or models.

Processes

The wall wetting model has several processes that add fuel to the liquid puddles, move the liquid fuel between the puddles, and remove the evaporated fuel. The dominant process is evaporation [3]. Each puddle is connected to the air-stream using an evaporation model. The evaporation model uses the Reynolds number of the flow over the puddle, the free stream gas state (temperature, pressure, composition), and puddle information from the puddle connector. It calculates a total mass convection rate from the puddle to the air stream. The evaporation mass flow rate is governed by:

$$\dot{m}_{evap} = Sh \rho_{mix} \frac{A_{puddle}}{d_{port}} D \ln\left(\frac{X_{fvp} - X_{fvi}}{1 - X_{fvp}}\right) \quad (1)$$

where Sh is the Sherwood Number (dimensionless concentration gradient which is dominated by the Reynolds Number), ρ_{mix} is the density of the air/fuel mixture in the gas phase directly above the puddle, A_{puddle} is the area of the puddle, d_{port} is the port diameter, D is the diffusion coefficient, X_{fvp} is the mass fraction of fuel in the vapor phase above the puddle, and X_{fvi} is the mass fraction of fuel vapor in the inlet stream. The total mass convection rate is divided among the various fuel components (species) in the puddle based on their mass fractions in the vapor phase.

Liquid fuel is added to the puddles via an injector model. The injector apportions the total fuel injected by means of data about the engine hardware (e.g. injector targeting info) and calibration parameters (e.g. how much fuel dribbles off of the injector as opposed to being sprayed). Most of the fuel during closed valve injection goes to the valve puddle and the downstream port puddle. During the rare event of open valve injection a large portion of the fuel goes directly to the cylinder puddle. The injector model also calculates an amount of fuel that is either vaporized or entrained in the air stream before it reaches the puddle. It does this by calculating a Roslin-Rammler distribution of the fuel droplet size in the injection spray. Then it assumes that all the drops under a certain diameter are entrained, and half of the drops between that size and a larger size are entrained. Both sizes are calibration parameters.

A forward flow model simulates the dragging effects of the air-flow in the port. The forward flow model moves liquid fuel from the upstream puddle to the downstream puddle, and liquid fuel from the downstream and valve puddles into the cylinder puddle. All of these flows are modeled by instances of the same flow model.

Using the mass flow rate of air in the port, the forward flow model makes several assumptions in order to calculate a mass flow rate. First, it is assumed that there is no slip at the surface between the puddle and the engine. Next, there is an equal shear force between the puddle and the air stream. Finally, there is a laminar flow distribution in the air and fuel film. The model then divides the total mass flow rate among the various fuel components in the puddle by their mass fractions in the puddle. The forward flow model also has an entrainment model similar to that in the injector model.

The process of backflow shattering is also modeled. This occurs at intake valve opening (IVO) when the pressure in the port at part throttle operating conditions is much less than that in the cylinder. At typical operating conditions, the cylinder pressure at IVO would be at atmospheric (100 kPa) and the pressure in the port would be about 50 kPa. This pressure difference produces a short duration but large magnitude, sometimes sonic, backflow event. This shatters the downstream and valve puddles. A percentage of the fuel that was shattered will be blown up into the port, a percentage will fall back into the puddle, and a percentage will be entrained in the air. The process is modeled as an event in Modelica. At IVO an event is triggered and a submodel calculates the redistribution of fuel. This information is then passed up to higher levels so that all of the control volume models are children. This model uses `reinit` statements to move the percentages among the various control volumes. This method is not entirely

satisfactory because the process is not completely represented in one submodel. We are currently evaluating the experimental impulse handling functionality in Dymola to rewrite the backflow shattering model.

Thermal Warm-up Model

One of the most important features in predicting transient A/F dynamics is good prediction of the temperature of the liquid fuel. The liquid fuel puddle is thin and is assumed to be in thermal equilibrium with the engine surface. The task is therefore to make a thermal model of the intake valve, seat, and cylinder walls. The Modelica version of the wall wetting model is essentially the same as the FORTRAN version [4]. The valve and valve seat are modeled by thermal capacitances connected by thermal resistances. This resistor-capacitor network is connected to the combustion gases, backflow gases, fresh charge gases, and coolant fluid via thermal convective resistances. Twenty-six thermal capacitances are used. Four are for the valve stem, three are for the valve seat, one is for a thermocouple, and 18 are for the valve head. The thermal capacitance and thermal resistance models are the `HeatCapacitance` and `HeatResistance` modes from the `HeatFlow1D` package found in the `ModelicaAdditions` library. The `Convection` model in the `HeatFlow1D` library was not suitable because the convection coefficient is a parameter. In our model the convection coefficient changes throughout the simulation so we made our own convection model with a variable convection coefficient. When a formal heat transfer library is available in the Modelica Standard Library, we will migrate our models to use the standard components.

The temperature of the valve puddle is calculated as a weighted average of the cells on the valve head. The downstream port puddle is connected to one of the seat cells. The upstream port puddle is connected to an average of the coolant and the seat.

Interface with Cycle Simulation

The original FORTRAN wall wetting model was not coupled or integrated into to a detailed engine cycle simulation model. Therefore simple but useful approximations for information such as in-cylinder pressure, burned gas temperatures, and in-cylinder and port air velocities were used as inputs.

The Modelica version of the wall wetting model was designed to permit integration with engine cycle simulations of varying complexity. The simplest cycle simulation would be to use the approximations that the original FORTRAN model uses. The next level of complexity would be to have a simple cycle simulation (e.g. using a single species ideal gas model, prescribed

burn model, and no in-cylinder heat transfer effects) to provide results for input to the wall wetting model, but not visa versa. A more complex cycle simulation could also be used (e.g. using a multiple species gas model with detailed property models, a predictive burn model, and in-cylinder heat transfer effects). Finally, the most complex form of integration would involve the two-way communication of results between the wall wetting model and the cycle simulation model. In other words, the cycle simulation would provide the wall wetting models with the necessary temperatures, pressures, air flow velocities, and heat transfer coefficients while the wall wetting model would provide the cycle simulation with the air/fuel ratio.

For our purposes we have built two versions of the wall wetting model. This first was for model verification. Here we used a simple cycle simulation that was coupled one way to the wall wetting model. Then for the camless application we chose a slightly more complex cycle simulation model (four gas species, thermodynamic relations by polynomial, prescribed burn, and no in-cylinder heat transfer effects) that was fully coupled to the wall-wetting model.

Actuator Model Development

Both simplified and detailed models of the EMVA have been developed. The simplified model is incorporated into the wall wetting simulation of the camless engine, while the more detailed "stand-alone" model is used for actuator controls development.

The actuator, shown schematically in Figure 2, is comprised of an upper and lower electromagnet and a moving armature which pushes on the engine poppet valve. Compression springs of equal stiffness (k_s) are placed above and below the armature, and are pre-loaded during assembly (by positioning the threaded top spring housing) to center the armature between the solenoid pole faces as shown in the left figure. During engine start-up, the valve is pulled from the center position to one of the pole faces corresponding to the open or closed position of the poppet valve. During normal engine operation, the armature and engine valve essentially operate as a reciprocating system. The motion during a transition from one pole face to the other is then primarily harmonic with the transition speed being determined by the effective armature/valve mass (m_{eff}) and the effective stiffness ($k_{eff} = 2k_s$) of the upper and lower springs. The electromagnets are used to (1) hold the valve in either the open or closed positions position, and (2) to inject enough magnetic energy into the armature to overcome frictional losses during transitions.

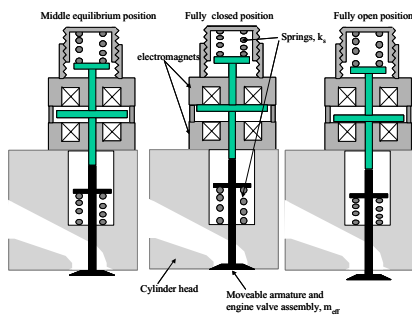


Figure 2: Schematic of the EMVA actuator in middle, fully open and fully closed positions.

Simplified EMVA Model

A simplified sub-model is developed for use with the engine cycle simulations. The simplified model provides valve profiles to the valve port flow models which subsequently determine the gas flow to and from the engine cylinder. From a free-body diagram of the effective reciprocating mass m_{eff} , the equation of motion during a transition can be expressed in terms of the viscous friction damping coefficient c , the effective spring constant k_{eff} , the upper $F_{mag,u}$ and lower magnet $F_{mag,l}$ forces, and the gas pressure and flow forces F_{gas} :

$$m_{eff} \ddot{z} + c \dot{z} + k_{eff} z = F_{mag,u} - F_{mag,l} - F_{gas} \quad (2)$$

In Equation 2, z is the distance from the center position (the upper magnet face is at $z = L/2$ and the lower magnet is at $z = -L/2$. Lift L is the total armature travel). The magnetic force drops off as with the square of the armature distance from the pole face; therefore, during most of the transition, $F_{mag,u}$ and $F_{mag,l}$ are small compared to the spring forces. Additionally, the damping coefficient is very small, and for light to moderate engine loads, the gas forces are relatively small. A reasonable first approximation to the valve lift $x = z - L/2$ is harmonic motion at a frequency of $\omega_n = (k_{eff}/m_{eff})^{1/2}$. For example the position for movement from the closed position at time t_0 is given by:

$$x = \frac{L}{2} (1 - \cos\{\omega_n (t - t_0)\}) \text{ for } t - t_0 \leq \pi/\omega_n \quad (3)$$

and

$$x = L \text{ for } t - t_0 > \pi/\omega_n \quad (4)$$

This simply generates a time based one-half period harmonic transition from closed to open position. A similar expression is used for the valve closing transition.

Figure 3 illustrates an instance of the simplified EMVA model within the context of the camless engine exhaust valvetrain model. The sub-model incorporates a rotational connector to sense engine position and a control connector that provides opening and closing timing signals from higher levels of the model. The output is the harmonic lift profile which is then connected to the exhaust port flow model.

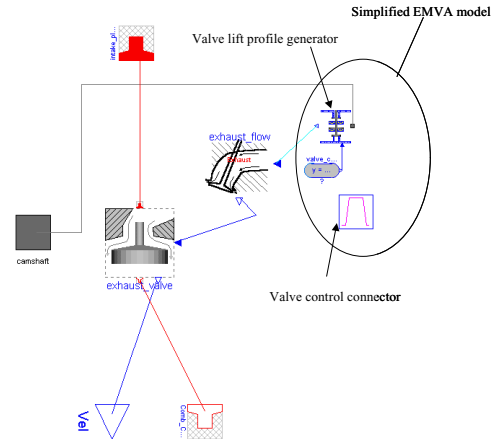


Figure 3: Exhaust valvetrain model showing the simplified EMVA model

Detailed EMVA Model

Modelica standard libraries for linear masses and springs are used to model the mechanical characteristics of the system. Additionally a model of an E-core type electrical solenoid is developed and used in conjunction with the electrical libraries. This provides a plant model for evaluation of passive and active motion control schemes.

The model, shown in Figure 4, is used to evaluate the dynamics of the armature motion during catching near the end of a transition. It includes the mechanical system, a catching electromagnet, and simple voltage supply.

The mechanical system is modeled as a reciprocating mass that is connected to 4 spring-damper elements. These elements are piece-wise linear with a change in stiffness and damping coefficient defined by the positions where the armature meets the magnet pole face. Two of the spring-dampers represent the mechanical stiffness of the upper and lower actuator springs, while two other high-stiffness elements simulate the collision between the armature and the electromagnets. The lower stiffness spring-damper parameters are active during mid-travel ($-L/2 < z < L/2$) and are tuned to match the free oscillation motion of the armature. The high stiffness spring-dampers (active for $|z| \geq L/2$) are then tuned to match the experimental data to simulate the inelastic collision of the armature with either of the magnet pole faces.

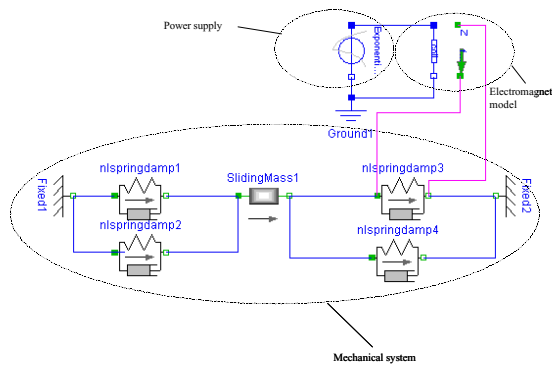


Figure 4: Actuator model

Also shown in Figure 4 is an e-core magnet sub-model. Electrical connectors are provided to connect the coils to a voltage source. In addition, a translational connector is provided to apply the magnetic force to the spring-mass-damper system.

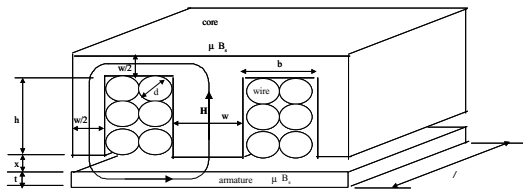


Figure 5: "E"-core magnet schematic

The magnet model development begins with a consideration of the e-core geometry and flux path, which is shown schematically in Figure 5. By applying Gauss' law for magnetostatics:

$$\oint \vec{B} \cdot d\vec{A} = 0 \quad (5)$$

where \vec{B} is the magnetic field and Ampere's law:

$$\oint \vec{H} \cdot d\vec{l} = I_{enclosed} = Ni \quad (6)$$

where H is the magnetic excitation, i is the current, and N the number of coil turns, the flux can be expressed in terms of the geometry, windings, material properties, air gap x , and current i for both the linear (where magnetic field $\vec{B} = \mu \vec{H}$) and magnetic saturation regions of operation. In the linear region the flux is given by:

$$\lambda = \frac{ai}{k + x} \quad (7)$$

where a and k are constants determined by the core and armature dimensions and material properties.

Integrating λ with respect to current i gives the co-energy, which can be differentiated with respect to the air gap to give the magnetic force F_{mag} :

$$F_{mag} = \frac{ai^2}{2(k + x)^2} \quad (8)$$

The flux and magnetic force will vary according to Equation 7 and Equation 8 until either the core or armature begins to saturate at higher current levels. Here, an exponential form for the flux is defined which permits the characterization of the flux and magnetic force in terms of the B-H curve characteristic of the materials.

With the flux characterized, the equation which describes the voltage V_a applied across the coil can be expressed using Kirchhoff's, Faraday's and Ohm's laws:

$$V_a = \frac{d\lambda}{dt} + Ri \quad (9)$$

where R is the coil resistance which is parameterized in terms of the e-core dimensions and wire diameter d .

Equations for the magnetic force and the coil voltage essentially describe the magnet sub-models. The model interfaces with the electrical and mechanical subsystems through translational connectors and electrical pins.

Results and Discussion

Wall Wetting Simulations

The camless wall wetting model has been used to model 1200 RPM 300-second engine "cold-start" tests. The engine starts from near ambient conditions, and is then operated at 1200 RPM. The engine load (or torque) is periodically moved between a lower and higher level, with the excursions being made during a 1-second interval. The load changes are accomplished by changing the engine airflow induction rate. Both "throttled" and "unthrottled" operating modes are investigated, and simulation results are compared to experimental data. In the throttled mode, the camless engine is operated in a conventional way. The valve timings are fixed and load changes are executed by throttling the air flowing into the intake manifold. In the unthrottled mode, the intake manifold air is at atmospheric pressure. Load changes are accomplished by changing the intake valve closing timing (IVC) to change the length of the induction stroke.

Figure 6 shows the results for the throttled engine operation. Both experimental and predicted results are shown for injected air-fuel ratio (the ratio of inducted air mass per cycle to injected fuel mass per cycle) and for the air-fuel ratio in the engine exhaust (inferred from measuring exhaust species concentrations). Note that these are in general different under transient conditions due to the wall wetting fuel dynamics. The model prediction for injected air/fuel ratio is significantly higher than the experimental injected air fuel ratio during the high load operating condition. This difference may be attributed to modeling and experimental error. The difference could be due to over-prediction of the inducted air mass during high load conditions. The exhaust air fuel ratio for the experimental data and modeling simulation behave similarly during low load engine operation, but during high load operation conditions the experimental and modeled exhaust air fuel ratio diverge. This may be due to the differences in the injected air fuel ratios and experimental error.

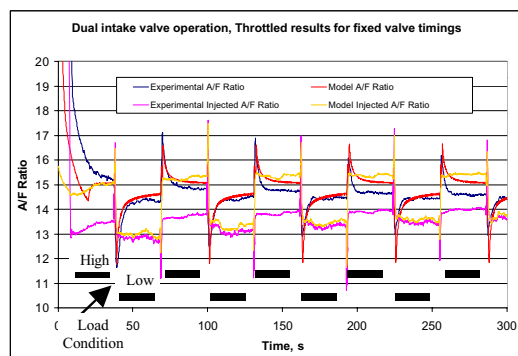


Figure 6: Throttled Operation.

Figure 7 shows the results for unthrottled engine operation. The injected air fuel ratio for the model closely matches the experimental data, demonstrating that the air charge estimation is improved compared to the throttled operating condition. However, the modeled exhaust air fuel ratio does not yield similar results. Although the experimental exhaust air fuel ratio tracks close to the desired stoichiometric conditions, the modeled exhaust air fuel ratio is calculated to be much richer. The model reasonably represents the air fuel excursions during load transitions, but most likely underestimates the quantity of fuel lost to the crankcase. If the model calibration of the lost fuel becomes more representative, the simulation is expected to more closely match the experimental results.

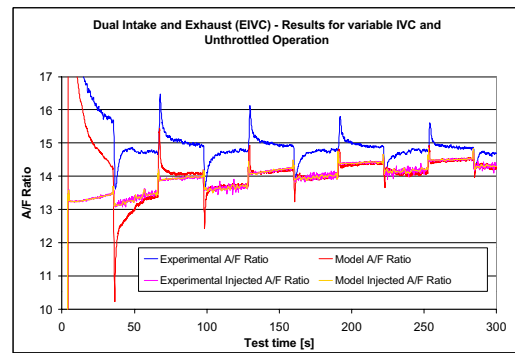


Figure 7: Unthrottled Operation.

These results show reasonable agreement for the general trends in air fuel-ratio behavior and demonstrate that Modelica is suitable for modeling transient air fuel dynamics; however, they also underscore the need for good model calibration and experimental air charge estimation. The FORTRAN version of the wall wetting model has a routine to calibrate the model by adjusting several parameters. The values of these calibrated parameters were used for the Modelica wall wetting model. However, the results show that the Modelica version of the model needs a different calibration process. Once a calibrated version of the model is available, it should be generally useful for both hardware and control strategy development.

Actuator Simulations

The actuator model has been exercised to investigate various design and motion control scenarios. Here we present results that compare model predictions to experimental data for armature catching using a simple square-wave catching pulse.

The model flux and force relationships are tuned to e-core and armature properties for a 200V prototype actuator using data from [5]. Mass, spring and damping parameters are selected to provide reasonable agreement between the predicted and measured free oscillation data. Experimental data are obtained by using a bench-top experimental set-up described in [6]. An actuator is installed on a cylinder head, and instrumentation is provided to drive the coil and to measure the position, velocity, current, and voltage. Figure 8 illustrates the experiment. The actuator is held in either the open or closed position with a low holding current in the corresponding coil. This holding coil current is then quenched at the time of the release command. After a delay time t_d , a square wave catching pulse of amplitude V_{app} and pulse-width t_{pw} is applied to the opposite coil to catch the armature at the magnet pole face. The catching coil voltage is then decreased to provide the lower current required to hold the valve in position.

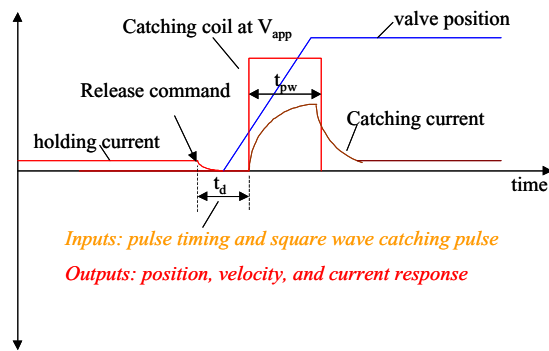


Figure 8: Schematic of armature catching experiment.

Figure 9 shows both predicted and measured position and velocity traces for a $V_{app} = 117$ V, $t_{pw} = 10$ ms catching pulse applied at $t_d = 1.4$ ms from the armature release point. The release spring first accelerates the armature and valve assembly to peak velocity of about 3.7 m/s. As the armature approaches the coil seat, it decelerates due to the catching spring force, but the magnetic force increases to pull the armature in to the open position. The predicted and measured contact velocities are about 0.3 m/s and 0.5 m/s respectively, and occur at 3.2 ms from the release point. Note that under these conditions the armature bounces and contacts a second time at about 0.6 m/s at around 4.5 ms.

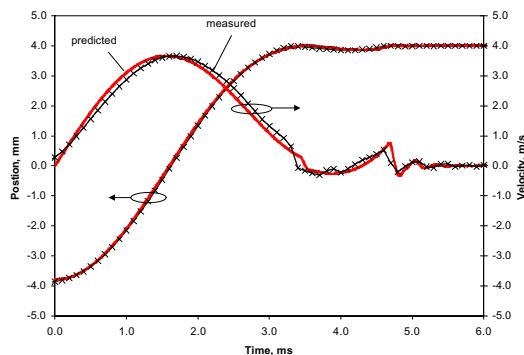


Figure 9: Predicted and measured position and velocity for apply voltage $V_{app} = 117$ V, delay time $t_d = 1.4$ ms, and pulse-width $t_{pw} = 10$ ms.

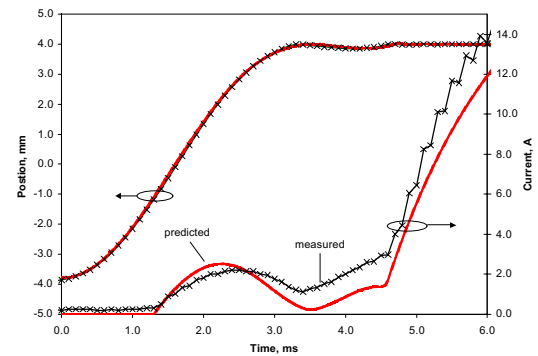


Figure 10: Predicted and measured position and current for apply voltage $V_{app} = 117$ V, delay time $t_d = 1.4$ ms, and pulse-width $t_{pw} = 10$ ms.

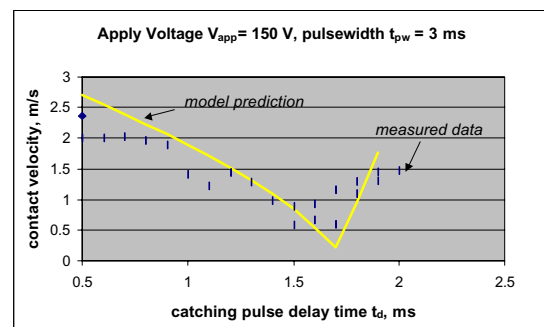


Figure 11: Predicted and measured contact velocity versus catching pulse delay time t_d for apply voltage $V_{app} = 150$ V and pulse-width $t_{pw} = 3$ ms

Figure 10 shows the predicted and measured current responses. The measured current first increases to about 2.0 amps, and then decreases as the armature lands due to the counter electromotive force (EMF) induced when the armature moves toward the magnet pole face. After bouncing, the armature moves away from the pole face and induces a reinforcing EMF. The current then increases, and this subsequently increases the magnetic force to pull the armature in with a higher contact velocity during the second impact. After the armature lands, the current then increases even more rapidly due to magnetic saturation effects. The model over-predicts current until very near the landing point. Here the model predicts a much sharper current decay than is shown by the measurement. The overall trends agree; however, model refinements are being developed to improve the current prediction.

An important issue for actuator design and control is the poppet valve and armature contact velocities. Valve seating velocities must be low enough so that valvetrain durability and noise level targets are met. Figure 11 shows the predicted contact velocities for the pulse timing sweep experiment shown in Figure 8. Here the applied voltage is $V_{app} = 150$ V, the pulse width is $t_{pw} = 3$ ms, and the pulse timing delay t_d is varied. As the pulse timing delay t_d is varied from 0.5

ms to 1.7 ms, the contact velocity decreases from about 2.7 m/s to a minimum of about 0.2 m/s. The minimum occurs when the injected energy from the magnetic force is about equal to the frictional losses from damping. For $t_d > 1.7$ ms the contact velocity begins to increase (due to reinforcement of the coil current as the armature motion reverses near the landing point) until $t_d = 2.0$ ms. Beyond this point, the magnetic force is not sufficient to catch the armature. Experimentally measured contact velocities are also shown in Figure 11. The predicted and measured trends agree reasonably well.

Conclusion

The Modelica language proved to be useful for creating a model for transient fuel dynamics in port fuel injected engines. The model was easily integrated into a cycle simulation model, and was suitable for modeling the transient fuel dynamics in a camless engine, as the predicted trends agreed reasonably with measured data. Modelica was also useful for developing camless engine valve actuator models. An actuator model was developed by using an e-core solenoid sub-model and a mixture of elements from the standard translational and electrical libraries. The model predictions for valve motion agreed reasonably well with experimental data.

References

1. Curtis, E., Russ, S., Aquino, C., Lavoie, G., Trigui, N., "The Effects of Injector Targeting and Fuel Volatility on Fuel Dynamics in a PFI Engine During Warm-Up: Part II – Modeling Results", SAE 982519
2. Newman, C., Batteh, J., Tiller, M., "Spark Ignited-Engine Cycle Simulation in Modelica", 2002 Modelica Conference Proceedings
3. Spalding, D.B., Combustion and Mass Transfer, Pergamon Press, 1979.
4. Curtis, E., Aquino, C., Plensdorf, W., Trumpy, D., Davis, G., Lavoie, G., "Modeling Intake Valve Warmup", ICE-Vol. 29-1, Proceedings of the 19th Annual Fall Technical Conference of the ASME Internal Combustion Engine Division, 1997.
5. Wang Y., Stefanopoulou A. G., Haghgooe M., Kolmanovsky I., Hammoud M., "Modeling of an Electromechanical Valve Actuator for a Camless Engine", AVEC 2000, 5th International Symposium on Advanced

Vehicle Control, No. 93, Ann Arbor, USA, 2000.

6. Wang Y., "Camless Engine Valvetrain: Enabling Technology and Control Techniques" Ph.D. Dissertation, University of California, Santa Barbara, 2001

Session 3b

Electrical Systems

An Incorporated Use of Genetic Algorithm and a Modelica Library for Simultaneous Tuning of Power System Stabilizers

Komsan Hongesombut, Yasunori Mitani, and Kiichiro Tsuji

Osaka University, Graduate school of engineering
2-1 Yamada-oka, Suita, Osaka 565-0871, JAPAN

Abstract

ObtectStab package that has been successfully applied to power system studies is a general-purpose simulation tool developed by the Modelica language. It takes advantages from the capability of physical modeling of Modelica language that make ones readily develop new models and use them for complex and large cases of power system studies based on object-oriented programming. However, in the situation that control of complex power systems is not easy to be realized by traditional methods, genetic algorithm (GA) becomes an alternative powerful method that can be used to solve several difficult problems without any prior or little knowledge of the systems being solved. Proposed in this paper is an incorporating the use of GA to an ObjectStab library to enhance the use of this library into optimization environment. The idea has been applied to one challenging problem of simultaneous tuning power system stabilizers in a multimachine power system. The simulation results show that the resulting controller obtained by a GA can achieve good performance.

Index Terms – ObjectStab, genetic algorithms, Simulink interface, simultaneous tuning, power system stabilization.

1. Introduction

Until recently, there has been widespread interest using genetic algorithms (GA's) to search and optimize in several difficult problems. Compared to traditional search and optimization procedures, such as calculus-based approach, GA's are robust, conceptually simple to apply in problems where little or no prior knowledge is available for the problem being solved. Problems on modern power systems are more and more difficult to be solved by using only conventional techniques due to large complex networks and nonlinear characteristic of power systems. The need of using other alternative tools such as genetic algorithms to solve such difficult problems become evi-

dent in case many conventional techniques get into difficulties. Incorporating the use of GA and power system simulation tools, among them such as PSCAD/EMTDC, EMTP, EuroStag, etc, ObjectStab [1] in Dymola [2] which is a library developed by Modelica language [3] for power system studies is more flexible than those in the view point of its easiness to realize the physical models and its powerful interface with MATLAB and Simulink that can allow ObjectStab be used with optimization methods such as GA. This paper describes a method of how a GA can be applied to a Modelica library named ObjectStab. An example of simultaneous tuning of power system stabilizers in a multimachine power system is used to validate the effectiveness of the incorporated use of these two features. It opens up a new idea of the use GA and Modelica library together allowing designers to design more sophisticated controllers. The idea does not limit only the applications to power systems, but also other Modelica users can adapt this idea to their own works. The simulation tools used in this paper are the Dymola, ObjectStab library, MATLAB [4] and Simulink [5] and Genetic and Evolutionary Algorithm Toolbox (GEATbx) [6].

2. Genetic Algorithms

A Genetic algorithm (GA) is a biologically inspired search algorithm pioneered by Holland [7]. The approach is based on Darwin's survival of the fitness hypothesis. In GA's, candidate solutions to a problem are analogous to individuals in a population. A population of individuals is maintained within search space for a GA, each representing a possible solution to a given problem. The initial population can be a random collection of bizarre individuals. The individuals will interact and breed to form future generations (offspring). The stronger individuals will reproduce more often than weaker individuals. Presumably, the population will get collectively stronger as generations pass and weaker individuals die out. Unlike other optimization methods, GA's do not limit by constraints on the form of fitness function. The fitness

function does not need to be differentiable or continuous. This flexibility in which GA's use a fitness function to search for the solution makes GA's become a power tool for optimization in many difficult problems in many fields.

GA's work with coding of the parameters themselves (called string) and then use the genetic operators to evolve the solution with minimum computation. An optimal solution can be found and represented by the final winner in the competitive environment. GA's consist of simple three operators; selection, crossover and mutation. Selection is the operation in which the fittest individual of the population in the current generation forms part of the population to the new generation. Crossover is responsible for providing new offspring by selecting two individuals and exchanging some parts of their structures. Mutation is an operator which is applied for altering the value of a random position in a string. A simple algorithm flowchart is shown in Fig.1.

3. Combination of Modelica library and Genetic Algorithms

In this section, we will generally describe how a Modelica library combines with a GA. One of the most powerful features of MEX files, including C format S-functions is it allows ones to incorporate existing code into a Simulink model. The key idea of combination a Modelica library and GA is using this feature by converting a Modelica model to a compiled MEX-file used in Simulink as an S-function block. Then a GA that exists in MATLAB environment will adjust some parameters of a Modelica model according to the fitness values. Briefly, incorporating a Modelica library and GA can be achieved by these following steps:

1. Build a Modelica model. The model is build up in Dymola environment.
2. Build a Simulink model named model 1 by using a DymolaBlock which is a new interface to Simulink that can be found in Simulink's library browser. This block is shielded around an S-function MEX block that interfaces to the C code generated by Dymola for the Modelica model. Model 1 is constructed for serving as an interfacing block for editing and compiling for two environments by switching the current active window between Dymola and Simulink environment.
3. Compile to Simulink dll file. It is possible to converted a Modelica model to a compiled MEX-file

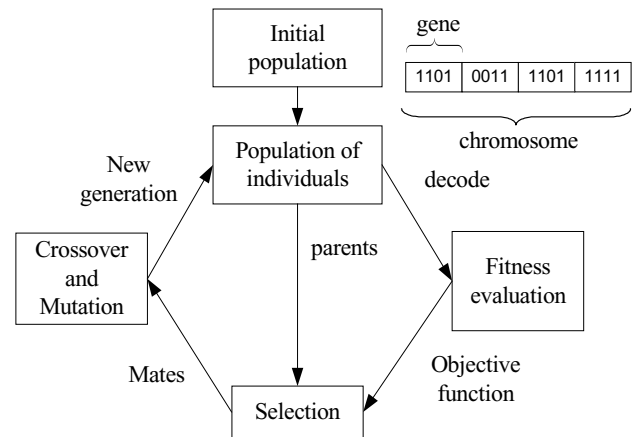


Fig.1 Simple algorithm flowchart of GA

SimStr.dll to be used as one block in Simulink environment. By doing this, command *dymcomp* is used.

4. Build a Simulink model named model 2. This model is served as a main system for connecting with a GA. It contains an S-function block representing a model as in Dymola and Simulink model for calculating fitness values used in a GA. Parameters and initial conditions are be defined or changed by passing these variables as inputs to S-function block.

5. Build a main m-file and a function used in a GA.

Details of above procedures are summarized and given in Fig.2. After this short summary of how a Modelica model combines with a GA, we will continue by real building a model for simultaneous tuning PSSs in a multimachine power system. We will show the flexibility of using GA by using two objective functions with the same Modelica model.

4. Problem Formulation

The objective of this problem is to tune an appropriate set of PSSs to damp local and inter-area modes. This problem is not easy by using traditional analytical methods to simultaneously tune all PSSs. The fixed structure of i^{th} PSS as shown in equation 1 is used for all 4 generators. It consists of a two-stage lead lag compensation with time constants $T_{li} - T_{4i}$, and a gain K_i . We set the wash out time constant T_{wi} with large enough value so that it can be considered as a constant.

$$PSS(s)_i = K_i \frac{sT_{wi}}{1 + sT_{wi}} \left(\frac{1 + sT_{li}}{1 + sT_{2i}} \frac{1 + sT_{3i}}{1 + sT_{4i}} \right) \quad (1)$$

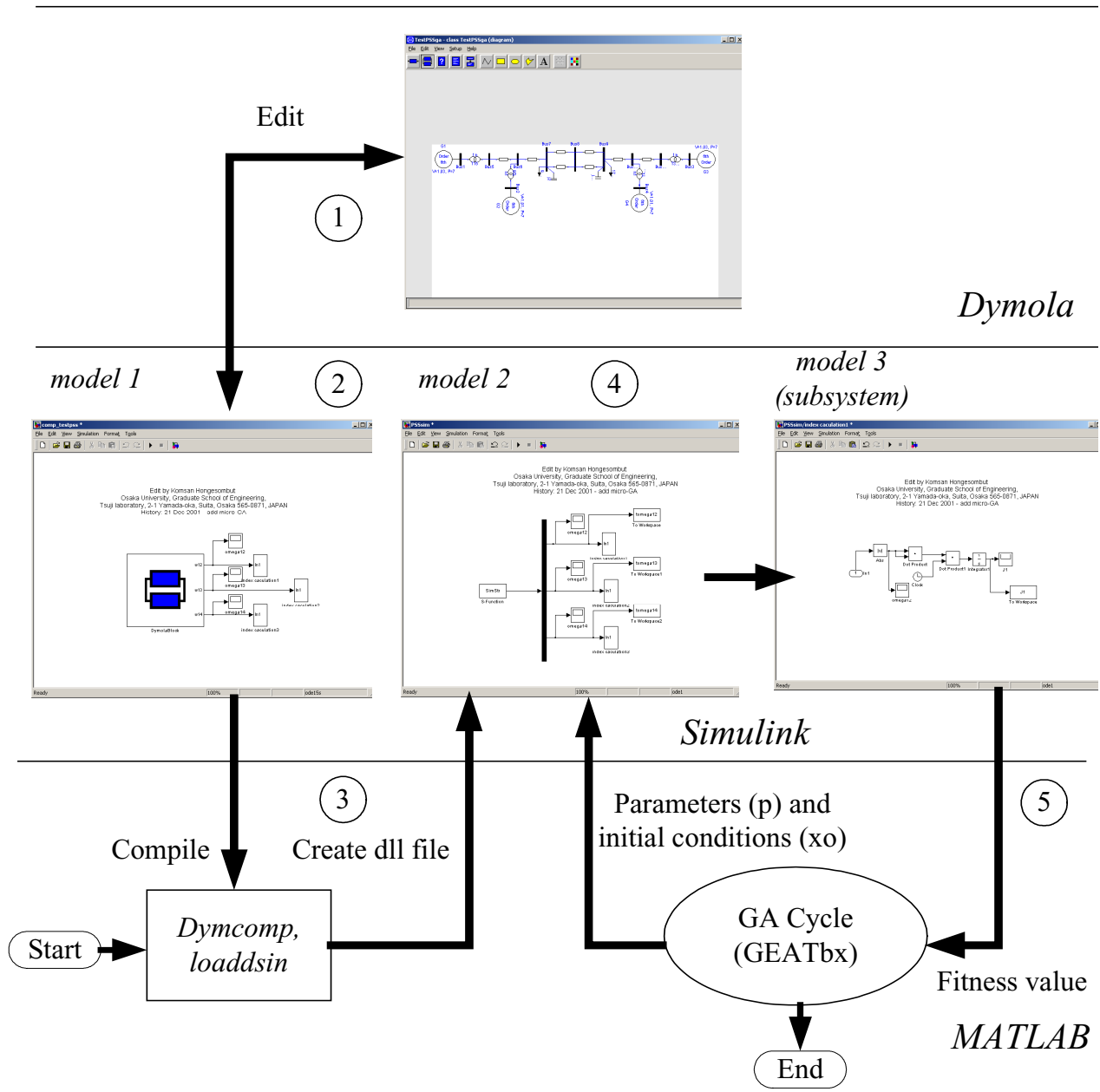


Fig. 2 Summary of how to combine a Modelica library with a GA

where

$$T_{1i} = T_{3i} = \frac{\sqrt{\gamma_i}}{\delta_i} \quad \text{and} \quad T_{2i} = T_{4i} = \frac{1}{\delta_i \sqrt{\gamma_i}} \quad (2)$$

We present two methods to satisfy the objective of tuning PSS as follows,

4.1 Method 1: time domain-based performance index

Typically, the performance of the design controller is measured directly from the output responses varying with time. This is a straightforward approach that can guarantee the performance of controllers under scenarios which are predefined by the designer. Equation 3 shows the objective function used in a GA meaning that we are trying to minimize the deviation of generator speed for local and inter-area modes by applying the suitable set of PSS control parameters.

$$\min F = \int_{t=0}^{10} \left(|\Delta\omega_{12}|^2 + |\Delta\omega_{13}|^2 + \dots + |\Delta\omega_{1n}|^2 \right) \cdot t \, dt \quad (3)$$

where n is the number of generators by assuming that generator 1 is a reference.

4.2 Method 2: eigenvalue-based performance index

For every operating conditions under consideration, here, it is supposed that a linearized model of power system is obtained first. The problem of selecting the parameters for power system stabilizers that can assure minimum damping performance over the considered set of operating point is converted to a simple optimization problem and then is solved by a GA with an eigenvalue-based performance index. The GA objective function is derived in this following way:

A linear model of power system is extracted around a certain operating condition. The system can be expressed in the linear state-space form as shown in the following equations

$$\dot{x} = Ax + Bu \quad (4)$$

$$y = Cx + Du \quad (5)$$

The equation expressed for the controllers is shown in (6) where in this study, the controller $K(s)$ is a lead-lag type that is the same as described by the transfer function in (1). $y(s)$ is the measuring signal and $V(s)$ is the output signal from the controller which provides additional damping by shifting under damped or unstable oscillation modes to the left hand side of the s -plane.

$$V(s) = K(s)y(s) \quad (6)$$

Combining equation 4 through 6, a closed-loop eigenvalues of the system can be obtained. Here, let $\lambda_i = \alpha_i \pm j\beta_i$ be the i^{th} mode of the closed-loop system. Damping coefficient δ_i of the i^{th} mode is calculated by

$$\delta_i = -\frac{\alpha_i}{\sqrt{\alpha_i^2 + \beta_i^2}} \quad (7)$$

If p is a number of operating conditions where each condition contains the matrix of damping coefficient δ_i , $i = 1, \dots, n$ where n is the number of oscillation modes of the closed-loop system. The optimization

problem to be solved by a GA can be written in the following form:

$$\max F = \min(\min(\delta_i))_p \quad (8)$$

For simplicity, we will choose only one operating condition for considering in this paper.

5. Test Power System and Scenarios

Fig.3 shows a single line diagram of a test power system constructed by using a graphical editor of Dymola and ObjectStab. The data of this power system network is given in [8]. The disturbance considered in this study is a three-phase to ground fault near Bus 7 by the following situations:

$t = 1 \text{ s}$: fault is applied,

$t = 1.1 \text{ s}$: fault is cleared by tripping one of two parallel lines.

$t = 2.5 \text{ s}$: line is reclosed.

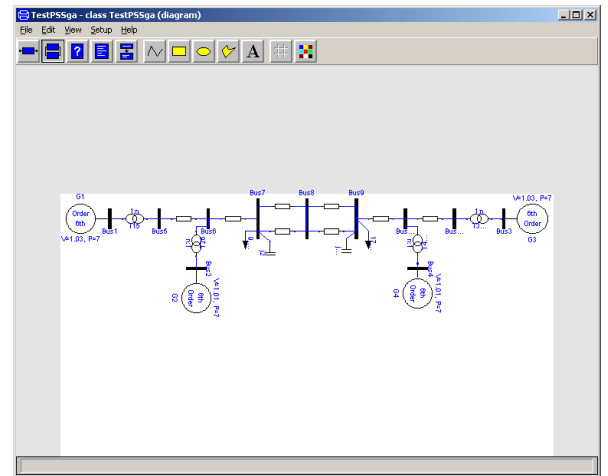


Fig.3 Power system model

6. Demonstration Example

In this section, we will describe the implementation of a GA to a Modelica library called ObjectStab by using 2 different objective functions as described in section 4. Considering the procedure chart in Fig.2, we need to follow 5 steps. It should be noted that only step 5 is different when changing the objective function of a GA. This is due to the manner in which a GA uses the fitness function to evaluate the goodness of solutions that provides greater flexibility of using GA to realize many difficult problems.

In order to follow the procedure in Fig.2, first task is to build a power system model in Dymola. By using the objective function in method 1, we need 3 output variables which are the speed difference of generator 1 and 2, the speed difference of generator 1 and 3, and the speed difference of generator 1 and 4. It should be noted that this is because generator 1 is taken as a reference, hence, the speed difference of generators within the same area is represented as the local mode and the speed difference of generator with different area is represented as the inter-area mode.

Next, we build 3 models in Simulink. Model 1 as shown in Fig.4 can be constructed by drag and drop a DymolaBlock which can be found in Simulink's library browser to a Simulink model. Model name and its path of the Modelica model must be specified in the DymolaBlock in order to point the location of a created Modelica model. It is possible that users can modify a Modelica model directly by using the editing command in the DymolaBlock or compiling a Modelica model by using compiling command. In order to make a Modelica model useful in Simulink and a GA, we will declare external outputs of a Modelica model. These outputs are used for evaluating the fitness value in a GA. The following script is an example of external output declaration in a Modelica model.

```
class TestPSSga
  extends ObjectStub.Examples.Kundur126.linefault;
  Real w1, w2, w3, w4;
  output Real w12;
  output Real w13;
  output Real w14;
equation
  w1 = G1.w;
  w2 = G2.w;
  w3 = G3.w;
  w4 = G4.w;
  w12 = w1 - w2;
  w13 = w1 - w3;
  w14 = w1 - w4;
end TestPSSga
```

After compiling the model, the declared outputs will appear in the DymolaBlock. These outputs can be connected with other Simulink blocks. Now, we can convert a Modelica model to a compiled MEX S-function file by using the following MATLAB commands

```
dymcomp;
[p, x0, pnames, x0names, inputnames, outputnames] = loadssin;
```

The first command line is used to generate a compiled MEX S-function file (dll file). The second command line is used to load values such as parameters, initial conditions and their names from *dsin.txt*

which are necessary for input parameters to S-function block. GA will change parameters p every iterations according to the decoded chromosome.

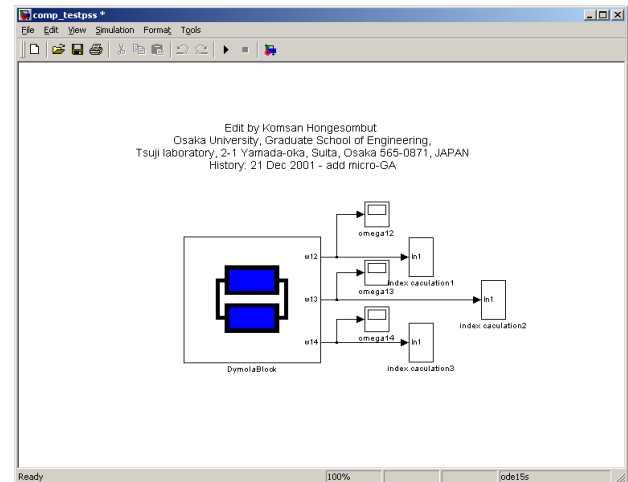


Fig.4 Model 1 in Simulink

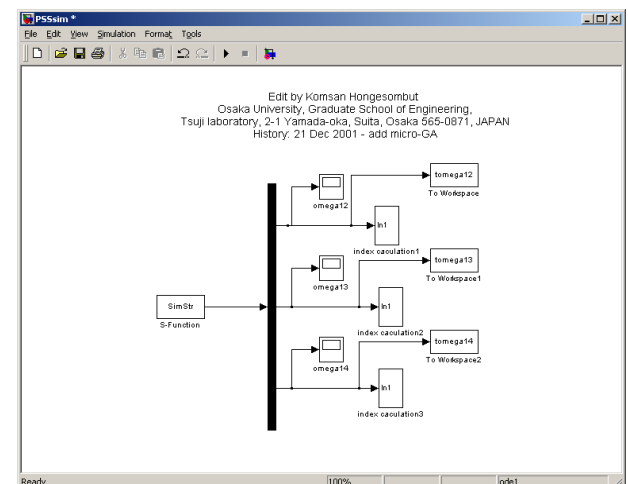


Fig.5 Model 2 in Simulink

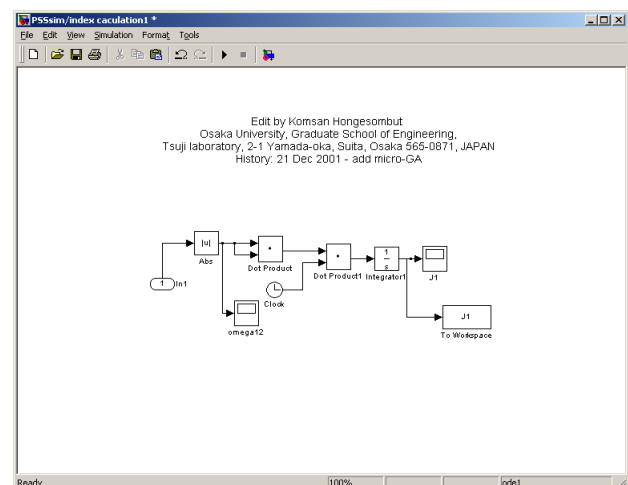


Fig.6 Model 3 in Simulink

```

function objval=Obj_cal1(x)
global p x0 pnames x0names inputnames outputnames
global J1 J2 J3
for i=1:size(x,1) % for ith individual
clear J1 J2 J3 ;
%----Step 1 : decoding -----
KPSS1=x(i,1); g1=x(i,2); d1=x(i,3);
KPSS2=x(i,4); g2=x(i,5); d2=x(i,6);
KPSS3=x(i,7); g3=x(i,8); d3=x(i,9);
KPSS4=x(i,10); g4=x(i,11); d4=x(i,12);
%----Step 2: Find parameter index -----
pindex=[
    tnindex(pnames,'G1.Exc.Kstab');
    tnindex(pnames,'G1.Exc.T1');
    tnindex(pnames,'G1.Exc.T2');
    tnindex(pnames,'G1.Exc.T3');
    tnindex(pnames,'G1.Exc.T4');
    tnindex(pnames,'G2.Exc.Kstab');
    tnindex(pnames,'G2.Exc.T1');
    tnindex(pnames,'G2.Exc.T2');
    tnindex(pnames,'G2.Exc.T3');
    tnindex(pnames,'G2.Exc.T4');
    tnindex(pnames,'G3.Exc.Kstab');
    tnindex(pnames,'G3.Exc.T1');
    tnindex(pnames,'G3.Exc.T2');
    tnindex(pnames,'G3.Exc.T3');
    tnindex(pnames,'G3.Exc.T4');
    tnindex(pnames,'G4.Exc.Kstab');
    tnindex(pnames,'G4.Exc.T1');
    tnindex(pnames,'G4.Exc.T2');
    tnindex(pnames,'G4.Exc.T3');
    tnindex(pnames,'G4.Exc.T4');
];
T11=sqrt(g1)/d1; T21=1/(d1*sqrt(g1));
T31=T11; T41=T21;
T12=sqrt(g2)/d2; T22=1/(d2*sqrt(g2));
T32=T12; T42=T22;
T13=sqrt(g3)/d3; T23=1/(d3*sqrt(g3));
T33=T13; T43=T23;
T14=sqrt(g4)/d4; T24=1/(d4*sqrt(g4));
T34=T14; T44=T24;
%----Step 3: change parameter values ----
p(pindex)=[KPSS1;T11;T21;T31;T41;...
            KPSS2;T12;T22;T32;T42;...
            KPSS3;T13;T23;T33;T43;...
            KPSS4;T14;T24;T34;T44;];
sim('PSSsim',[0 10])
index1=max(J1);index2=max(J2);index3=max(J3);
total_index = sum(index1+index2+index3);
objval(i,1)= total_index
end

```

(a) Objective function by method 1

```

function objval=Obj_caleig1(x)
global p x0 pnames x0names inputnames outputnames
global J1 J2 J3
for i=1:size(x,1) % for ith individual
%----Step 1 : decoding -----
KPSS1=x(i,1); g1=x(i,2); d1=x(i,3);
KPSS2=x(i,4); g2=x(i,5); d2=x(i,6);
KPSS3=x(i,7); g3=x(i,8); d3=x(i,9);
KPSS4=x(i,10); g4=x(i,11); d4=x(i,12);
%----Step 2: Find parameter index -----
pindex=[ ...
    tnindex(pnames,'G1.Exc.Kstab');
    tnindex(pnames,'G1.Exc.T1');
    tnindex(pnames,'G1.Exc.T2');
    tnindex(pnames,'G1.Exc.T3');
    tnindex(pnames,'G1.Exc.T4');
    tnindex(pnames,'G2.Exc.Kstab');
    tnindex(pnames,'G2.Exc.T1');
    tnindex(pnames,'G2.Exc.T2');
    tnindex(pnames,'G2.Exc.T3');
    tnindex(pnames,'G2.Exc.T4');
    tnindex(pnames,'G3.Exc.Kstab');
    tnindex(pnames,'G3.Exc.T1');
    tnindex(pnames,'G3.Exc.T2');
    tnindex(pnames,'G3.Exc.T3');
    tnindex(pnames,'G3.Exc.T4');
    tnindex(pnames,'G4.Exc.Kstab');
    tnindex(pnames,'G4.Exc.T1');
    tnindex(pnames,'G4.Exc.T2');
    tnindex(pnames,'G4.Exc.T3');
    tnindex(pnames,'G4.Exc.T4');
];
T11=sqrt(g1)/d1; T21=1/(d1*sqrt(g1));
T31=T11; T41=T21;
T12=sqrt(g2)/d2; T22=1/(d2*sqrt(g2));
T32=T12; T42=T22;
T13=sqrt(g3)/d3; T23=1/(d3*sqrt(g3));
T33=T13; T43=T23;
T14=sqrt(g4)/d4; T24=1/(d4*sqrt(g4));
T34=T14; T44=T24;
p(pindex)=[KPSS1;T11;T21;T31;T41;...
            KPSS2;T12;T22;T32;T42;...
            KPSS3;T13;T23;T33;T43;...
            KPSS4;T14;T24;T34;T44;];
[t,xx,y]=sim('PSSsim_eig',[0 0.9]);
x0=xx(size(xx,1),:);
[A,B,C,D]=linmod('PSSsim_eig',x0);eigs=eig(A);
eigs=eigs(find(abs(eigs)>eps));%remove zero eigenval
damping= -real(eigs) ./sqrt(real(eigs).^2+imag(eigs).^2);
w=min(damping);objval(i,1)= -w
end

```

(b) Objective function by method 2

Fig.7 Comparison of two objective function used by a GA

After we get a compiled MEX S-function file which has a default name SimStr.dll, as stated earlier, we need to calculate $\int |\Delta\omega|^2 dt$ for each generator

speed deviation. Model 2 shown in Fig.5 is served for this function where model 3 shown in Fig.6 is a subsystem for calculation $\int |\Delta\omega|^2 dt$ of each speed sig-

nal. The summation of 3 speed signals become the objective function of a GA by using the method 1. Particularly useful in conjunction with a GA is the way to write the objective function. It is worthwhile to discuss the construction of the objective function. In Fig.7, it shows the comparison of two objective functions used in this study. The meaning behind each style is

Method 1:

1. Decode the chromosome of a GA.
2. Find the index of parameters which correspond to the tuning parameters in a Modelica model. The syntax of this command is

```
pindex = tindex(pnames, 'parameter name')
```

where pnames is obtained from *loadsin* command

3. Replace current parameters with new parameters obtained by a GA.
4. Run the Simulink model with *sim* command. Simulink will run the model 2 and save the index calculation of each signal when the simulation is complete.
5. Calculate the fitness value by summing 3 signals which each signal is calculated by subsystem model 3.

Method 2:

1. Decode the chromosome of a GA.
2. Find the index of parameters which correspond to the tuning parameters in a Modelica model.
3. Replace current parameters with new parameters obtained by a GA.
4. Run the Simulink with *sim* command in order to find good initial condition x_0 .
5. When the initial values are obtained, we can then proceed to use the MATLAB *linmod* function to determine the [A, B, C, D] matrices of the small-signal model of the nonlinear system about the chosen steady-state operating point. The syntax of the linearization command is as follows:

```
[A, B, C, D] = linmod('model name', x0)
```

- It should be noted that when calculating the eigenvalues, it is not necessary to have an input, but there should be at least one output of a Modelica model.
6. Calculate the fitness value by (8).

7. Simulation Results

A GA is applied to solve the problem of simultaneous tuning by using 2 different objective functions. In this study, routines from GEATbx were used with

bounds for PSS parameters shown in Table 1. The implementation of a GA in this work used real encoding chromosome, a population size 30, maximum generation 50, a uniform crossover rate of 0.9 and a uniform mutation rate of 0.01. The approach also adopted an elitist strategy that copied the best string found in the current generation to the next generation. Selection was performed by using the tournament selection with tournament size of 2. After executing a GA, the final result as shown in Table 2 were obtained. Fig.8 and 9 show the screen outputs of a GA by using the objective function by method 1 and method 2 respectively.

Table 1 Bounds for PSS parameters

PSS parameter	value
K_{\min}	0
K_{\max}	20
γ_{\min}	0.1
γ_{\max}	10
δ_{\min}	1
δ_{\max}	10

Table 2 Final result obtained by a GA

Method 1	K	$T_1 = T_3$	$T_2 = T_4$
PSS1	20.000	0.331	0.139
PSS2	20.000	0.107	0.291
PSS3	17.791	0.127	0.153
PSS4	18.319	0.201	0.055
Method 2	K	$T_1 = T_3$	$T_2 = T_4$
PSS1	19.175	1.583	0.632
PSS2	20.000	0.161	0.184
PSS3	14.209	0.198	0.239
PSS4	7.513	0.051	0.218

To demonstrate the effectiveness of the resulting controller obtained by using 2 objective functions, nonlinear simulation and plot of close-loop eigenvalues were performed. In nonlinear simulations of Fig.10 to 12, the responses of generator speed deviation for local and inter-area modes confirm the effectiveness of the results obtained by a GA. It should be noted that the method 1 gives better result than the method 2 when using time domain-based performance index. The system is well damped and is stabilized in less than 5 seconds.

Fig. 13 to 14 show the plot of dominant eigenvalues of the closed-loop system. It can be observed that using PSS parameters obtained by both methods, the system is sufficiently damped with all modes of the

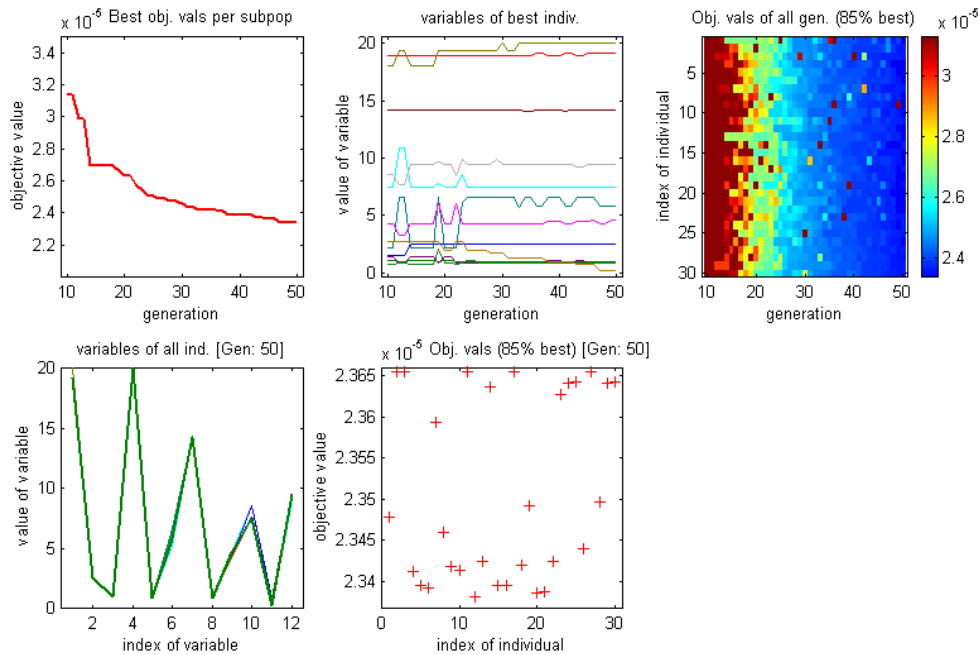


Fig.8 Screen output from a GA by using the objective function in method 1

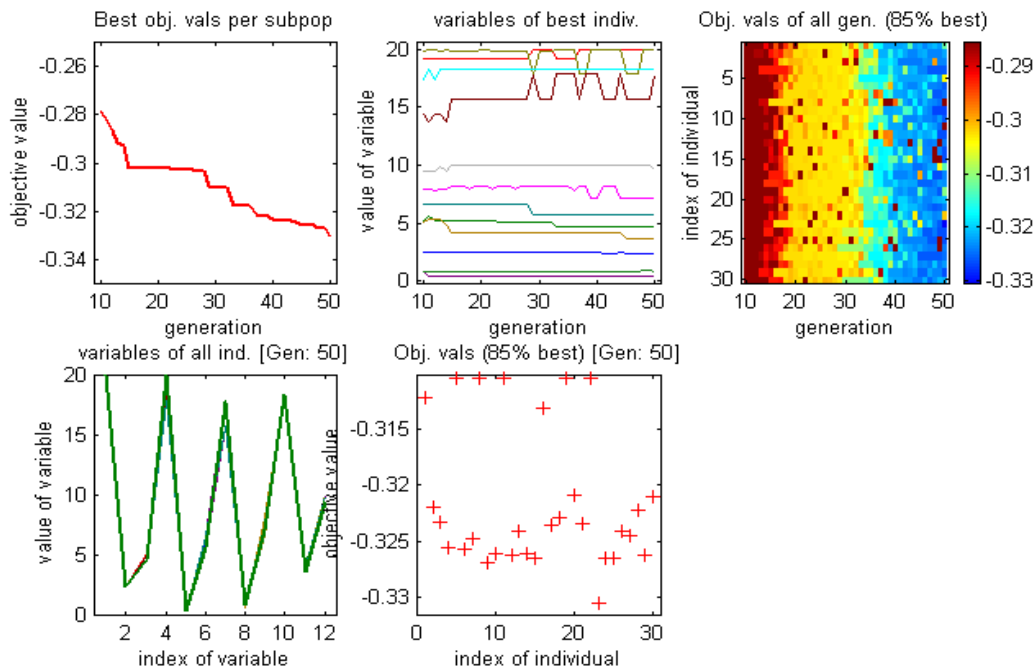


Fig.9 Screen output from a GA by using the objective function in method 2

system having the minimum damping greater 5% which is a typical requirement in PSS tuning. It is also found that the method 2 gives better result than the method 1 in case of using eigenvalue-based performance index.

It is become clear that using different GA objective function, the final result may be quite different. In addition, GA is a time consuming search procedure. Thus, GA is not generally used for problems easily optimized.

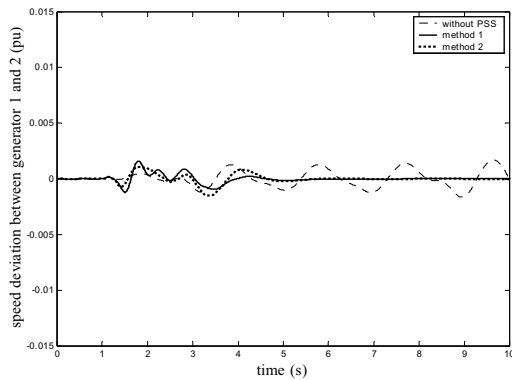


Fig.10 Speed deviation of generator 1 and 2

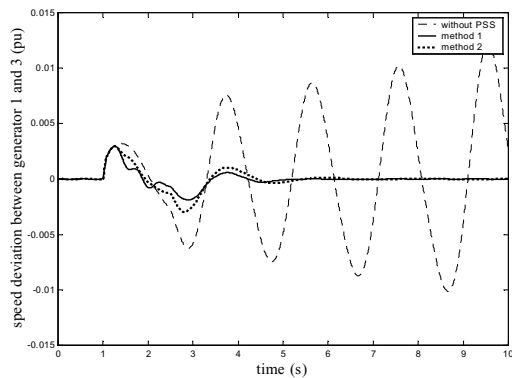


Fig.11 Speed deviation of generator 1 and 3

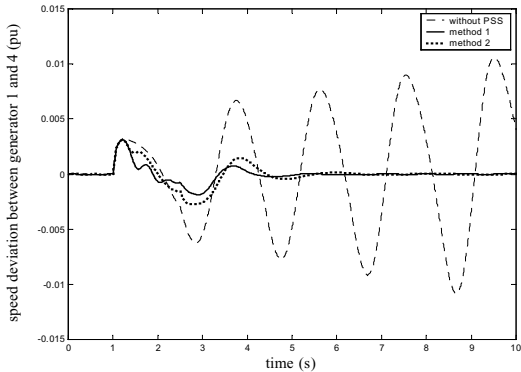


Fig.12 Speed deviation of generator 1 and 4

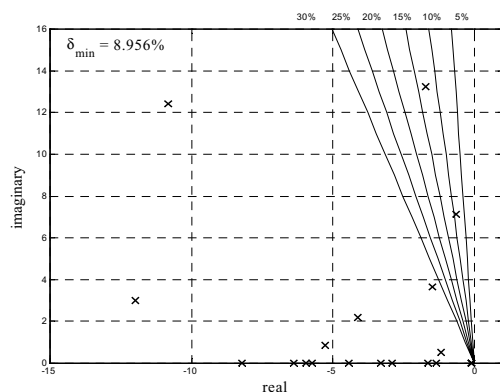


Fig.13 Closed-loop eigenvalues obtained by method 1

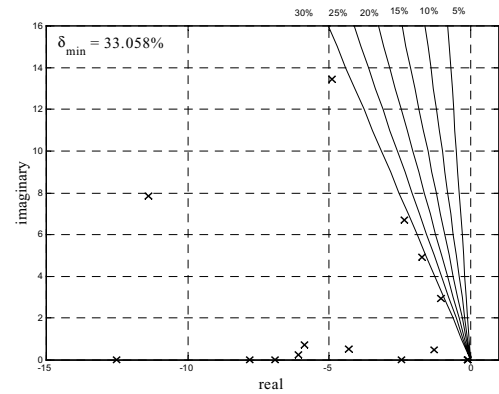


Fig.14 Closed-loop eigenvalues obtained by method 2

8. Conclusions

This paper deals with the incorporated use of a Modelica library called ObjectStab and a GA and application of a GA for simultaneous tuning of power system stabilizers in a multimachine power system. The power system modeling can be realized by using ObjectStab where the behavior of dynamic systems can be expressed by using advance features of Modelica language for detailed physical modeling. We also showed how to link a GA and a Modelica model by using the Simulink interface of the Dymola. We showed the flexibility of optimization by a GA with two different objective functions without modifying the original Modelica model. Given a suitable objective function, the final solution will satisfy the required controller performance. It is important to point that the idea does not limit only the applications to power systems as shown in an example of this paper, but also other Modelica users can adapt this idea to their own works.

9. Acknowledgement

We gratefully acknowledge helpful discussions with Dr. Mats Larsson from ABB Corporate Research Ltd., Switzerland.

10. References

- [1] M. Larsson, "ObjectStab - a Modelica library for power system stability studies", Proc. of the 2000 Modelica Workshop.
- [2] Dymola, *Dynamic Modeling Laboratory*, Dynasim 2001.
- [3] M. M. Tiller, *Introduction to Physical Modeling with Modelica*, Kluwer Academic Publishers, Massachusetts 2001.

- [4] D. Hanselman and B. Littlefield, *Mastering Matlab 6*, Prentice Hall, New Jersey 2001.
- [5] J. B. Dabney and T. L. Harman, *Mastering Simulink 4*, Prentice Hall, New Jersey 2001.
- [6] H. Pohlheim, *Genetic and Evolutionary Algorithm Toolbox for use with MATLAB*, Publication on internet web at <http://www.geatbx.com>.
- [7] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison Wesley, New York, 1989.
- [8] P. Kundur, *Power system stability and control* McGrawHill, New York 1993.

Komsan Hongesombut received his B.Eng.(first class honors) and M.Eng. degrees from the Department of Electrical Engineering, King Mongkut's Institute of Technology Ladkrabang, Thailand in 1997 and 1999 respectively. He is currently a Ph.D student at Osaka University, Japan. His research interests include the applications of intelligent techniques to power systems. He is a student member of the Institute of Electrical Engineers of Japan, IEE, and IEEE.

Yasunori Mitani received his B.Sc., M.Sc., and Dr. of Engineering degrees in electrical engineering from Osaka University, Japan in 1981, 1983, and 1986 respectively. He joined the Department of Electrical Engineering of the same university in 1990. He is currently Associate Professor. His research interests are in the areas of analysis and control of power systems. He is a member of the Institute of Electrical Engineers of Japan, the Institute of Systems, Control and Information Engineers of Japan, and the IEEE.

Kiichiro Tsuji received his B.Sc and M.Sc. degrees in electrical engineering from Osaka University, Japan, in 1966 and 1968, respectively, and his Ph.D in systems engineering from Case Western Reserve University, Cleveland, Ohio in 1973. In 1973 he jointed the Department of Electrical Engineering, Osaka University, and is currently Professor. His research interests are in the areas of analysis, planning, and evaluation of energy systems, including electrical power systems. He is a member of the Institute of Electrical Engineers of Japan, the Japan Society of Energy and Resources, the Society of Instrument and Control Engineers, the Institute of Systems, Control and Information Engineers, and the IEEE.

DC, AC Small-Signal and Transient Analysis of Level1 N-Channel MOSFET with Modelica

A. Urquía and S. Dormido

Dep. Informática y Automática, Facultad de Ciencias, U.N.E.D.

Avda. Senda del Rey 9, 28040 Madrid, Spain.

E-mail: aurquia@dia.uned.es, sdormido@dia.uned.es

ABSTRACT

The “translation” of the SPICE capabilities into Modelica language would allow combining the best of each tool: the SPICE expertise at circuit analysis and the Modelica/Dymola expertise at object-oriented modelling and simulation of hybrid systems. This contribution intends to be a first step to achieve this goal. A reduced group of SPICE device models are translated into Modelica language for OP, AC and TRAN analyses. It includes passive components (resistor and capacitor), independent voltage and current sources, and the SPICE2 level1 n-channel MOSFET.

1. INTRODUCTION

The simulator SPICE is an essential computer-aid for circuit design. Originally, SPICE2 was conceived as a stand-alone, general purpose, analog circuit simulator. However, since the development of SPICE2 at the University of California in 1975, many commercial and freeware SPICE-compatible simulators have been developed for a variety of systems (UNIX, PC, etc). Most of these tools

- run in connection with other simulation programs used in the circuit design flow,
- support analog, digital and mixed analog/digital simulation, and
- include improved device models, additional analyses and device model libraries.

They provide some support to the multi-domain system simulation facilitating the *analog behavioral modelling* (ABM). Behavioral parts allow defining a circuit segment as a mathematical expression or a lookup table. PSpice (OrCAD, 1999) is a commercial, PC-version, SPICE-compatible simulator. PSpice ABM library includes math functions, limiters, Chebyshev filters, integrators, differentiators, etc. However, the SPICE-based simulators impose a hard restriction to ABM: the function continuity (OrCAD, 1999; Kielkowski, 1998).

Device equations built into SPICE are continuous. For instance, voltage- or current-controlled switches are not ideal: they have a finite (very small) “on” resistance and (very large) “off” resistance. The switch resistance changes smoothly between the two

as its control voltage or current changes. Equally, the functions available for ABM are also continuous (for instance, the *int* function can not be implemented). The reason behind this requirement is the heavy use that SPICE numerical algorithms make of continuity (OrCAD, 1999; Kielkowski, 1998). In consequence, SPICE-based simulators are not suited for the simulation of hybrid models (i.e., combined continuous/discrete models) due to its inability to handle discrete events.

On the contrary, general-purpose modelling languages are intended for the simulation of multi-domain hybrid models. To this respect, the object-oriented modelling language Modelica (Modelica, 2000) is intended to serve as a standard format so that models arising in different domains can be exchanged between tools and users (Aström, Elmqvist and Mattsson, 1998). The “translation” of the SPICE capabilities (device models and analysis modes) into Modelica language is one of the Modelica library improvements that have been suggested (Clauss et al., 2000). It would allow combining the best of each tool: the SPICE expertise at circuit analysis and the Modelica/Dymola (Elmqvist et al., 2000) expertise at object-oriented modelling and simulation of hybrid systems. This contribution intends to be a first step to achieve this goal.

An important feature of SPICE device models is their *variable-structure* nature. A model is said to have a variable structure when its mathematical description changes during the simulation run. A different device model is formulated for each analysis mode:

- static model (DC analysis),
- AC small-signal model (AC analysis), and
- large-signal model (transient analysis).

The transitions among these three device formulations are carried out in simulation time. A DC analysis (Massobrio and Antognetti, 1993)

- can be performed prior to a transient analysis to determine the transient initial conditions, and
- it is automatically performed prior to an AC small-signal analysis to determine the linearized, small-signal models for the non-linear devices.

In addition, some DC analysis algorithms require the combined use of the three device formulations.

In this contribution three analysis modes are considered:

- bias point (OP),
- AC sweep (AC), and
- transient analysis (TRAN),

for three analog device types:

- *Passive devices*: linear resistor and capacitor.
- *Independent voltage and current sources*.
- *Semiconductor device*: SPICE2 level1 n-channel MOSFET. It is composed of linear resistors, voltage-dependent capacitors and voltage-controlled current sources.

In addition, IC1 and IC2 pseudo-components are modelled for setting initial conditions.

Model structuring into libraries and the interaction between models are discussed in Section 2. The way of using the model libraries to analyse the user-defined circuits is also outlined. Initial condition setting is described Section 3. Two procedures are supported: IC symbols and the capacitor IC property. The translation into Modelica language of the passive device and source models is addressed in Section 4. Device models have a variable structure and signals are defined to control the model structure transitions. Each analysis mode consists on an ordered sequence of elementary operations implying changes in the device model structure. Analysis models set the control signals in order to accomplish the required device-model structure changes. Analysis models are discussed in Section 5. Bias point calculation is the most problematic step from the numerical point of view. Four alternative bias point calculation algorithms are implemented. Finally, level1 NMOS model is outlined in Section 6.

For the sake of simplicity, neither parameter dependence with temperature nor TEMP analysis have been considered in the present library release. Temperature is considered a constant variable intervening in some device constitutive relations (for instance, the MOSFET source-substrate pn-junction model).

2. ARCHITECTURE

A two-level architecture is proposed (see Fig. 1):

- Upper (controller) level is composed of the analysis models.
- Lower (controlled) level is composed of the device models.

- Unidirectional *control signals* (arrow in Fig. 1) and *global variables* transmit the information from analysis models to device models. In addition, global parameters sets properties common to both analysis and device models.

Controller level: analysis models

ANALYSES package contains the *OP*, *TRAN* and *AC* models. Bias point calculation is a part of OP and AC analyses and it is an option of TRAN analysis. Therefore, the bias point calculation algorithms are programmed in a separate partial model, called *BiasPointCalculation*, inherited by the analysis models (see Fig. 1). Control signals (see Table 1) and global variables (see Table 2) are evaluated in the analysis models.

Control signal	T	W	R
Ctrl_AC	B	*	S
Ctrl_CBREAK_resetTran	B	BPC	C
Ctrl_CBREAK_Tran2DC	B	*	C
Ctrl_CBREAK_Tran2IC	B	*	C
Ctrl_DC	B	BPC	S
Ctrl_IC_clampDC	B	BPC	C, IC
Ctrl_IC_clampTran	B	BPC	C, IC
Ctrl_IC_mode	I	BPC	C, IC
Ctrl_IS_inhibit	B	BPC	S
Ctrl_IS_TranOP	B	BPC	S
Ctrl_log_AC	B	*	S, R
Ctrl_log_DC	B	BPC	S, R
Ctrl_OP_mode	I	BPC	S
Ctrl_OP_value	I	*	S
Ctrl_RBREAK_Tran2DC	B	BPC	R
Ctrl_Tran	B	*	S

Table 1. Control signals.

T: Variable type. (B): Boolean. (I): Integer (0,1)

W: Control signal written during the...

(BPC): bias point calculation. (*): other steps of the analyses.

R: Control signal read by ...

(S): source. (C): capacitor. (R): resistor. (IC): IC symbols

scaleGMIN	Scale factor of the "GMIN stepping" algorithm for bias point calculation.
Freq	AC small-signal frequency.
Temp	Analysis temperature.

Table 2. Global variables.

Controlled level: device models

Device models are grouped in three packages:

- BREAKOUT,
- SOURCE, and
- SPECIAL.

The models of BREAKOUT and SOURCE packages allow the composition of user-defined circuits, while the SPECIAL's provide one way to specify the simulation initial conditions. In addition, a fourth package containing the device model interfaces has been defined: INTERFACE.

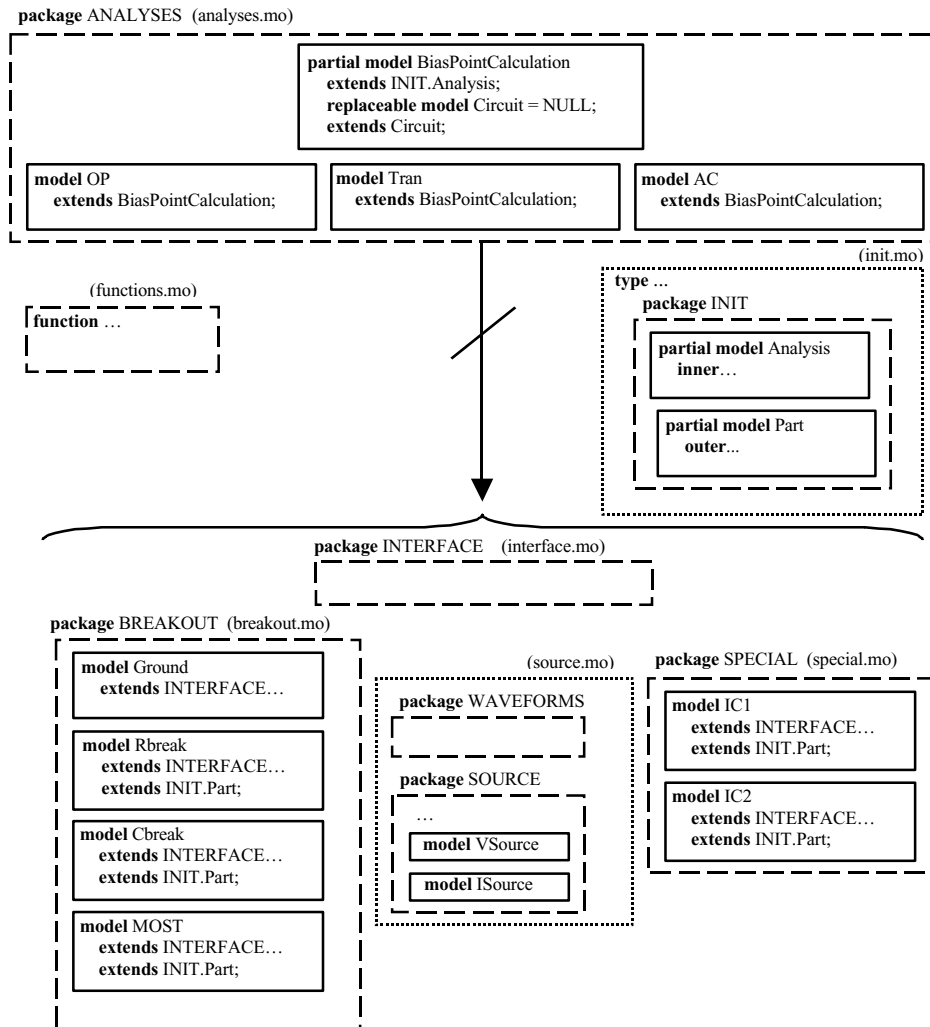


Figure 1. Two-level architecture.

Initialisation file

The initialisation file, *init.mo*, contains:

- Type definitions. Types conform to the Modelica *SIunits* package. However, they are redefined for the sake of conciseness when used. For instance:

```
type Voltage =
  Modelica.SIunits.Voltage;
```
- INIT package. The control signals, the global variables and the global parameters are defined in the INIT package. It contains two partial models (see Fig. 1):
 - *Analysis*, inherited by the analysis models.
 - *Part*, inherited by the device models.

The same set of control signals, variables and parameters is defined in both partial models: *Analysis* model variables are **inner** ones, while *Part* variables are **outer** ones.

Global parameters

Two global parameters have been defined (see Table 3). **TIME_SCALE** is used for setting the length of the source-ramping processes of some bias point calculation algorithms. In addition, it is used for establishing the time elapsed between consecutive control signal transitions (conceptually similar to the system clock period). To this end, the integer parameter **TIME_SLOT** is defined in the analysis models. It represents a percentage (1 to 100). The

time between consecutive events, **CLOCK**, is defined as follows:

$$\text{CLOCK} = \text{TIME_SLOT} * \text{TIME_SCALE} / 100$$

TIME_SCALE	It is intended for providing an (rough) approximate value of the circuit time-constant.
LOG_RESULTS	It determines the amount of information to be logged during the bias point calculation and the AC small-signal analysis.

Table 3. Global parameters

TIME_SCALE parameter plays another important role (not implemented in the current release of the libraries): redefine the units of the *time* variable in order to allow the adequate numerical solution of the system. Circuit simulation for microelectronics applications requires very small time values in comparison with the by-default time-related DAE-solver parameters. For this reason, it is best to include a scale factor between the circuit time and the DAE-solver time (i.e., the *time* variable).

Similar considerations will be made when discussing the pn-junction model. The use of the international system of units for the current is inadequate, because it leads to numerical problems. Large differences in the order of magnitude of the variables (for instance, the current and the voltage) makes impossible to set

adequate values for the numerical algorithm tolerances, the Dymola *evaps* parameter for event detection (Elmqvist, Cellier and Otter, 1993), etc. This fact is taken into account by re-formulating the model constitutive relations. In order to keep the compatibility with Modelica standard libraries, the international system of units is used for all the model terminal variables.

Performing circuit analyses

Two pieces of information are needed to perform a circuit analysis: the analysis model and the circuit model. The analysis models inherit (as a *replaceable model*, called *Circuit*) the circuit model (see *BiasPointCalculation* in Fig 1). The analysis model instantiations have to contain the redeclaration of the *Circuit* model. Consider the following example:

(File: *my_circuit.mo*)

```
model my_circuit
...
end my_circuit;

model circuitAnalysis_OP =
  ANALYSES.OP ( redeclare model Circuit =
    my_circuit);

model circuitAnalysis_Tran =
  ANALYSES.Tran ( redeclare model Circuit =
    my_circuit);

model circuitAnalysis_AC =
  ANALYSES.AC ( redeclare model Circuit =
    my_circuit);
```

The analysis to perform (only one per run) is selected in the script file. For instance, AC analysis:

(File: *my_circuit.mos*)

```
openModel("pspice.mo");
openModel("my_circuit.mo");
checkModel(problem="circuitAnalysis_AC");
translateModel(problem=
  "circuitAnalysis_AC");
```

The file *pspice.mo*:

- imports the library files (see Table 4), and
- defines the graphic windows containing the model icons.

File	Package
<i>analyses.mo</i>	ANALYSES
<i>breakout.mo</i>	BREAKOUT
<i>functions.mo</i>	
<i>init.mo</i>	INIT
<i>interface.mo</i>	INTERFACE
<i>pspice.mo</i>	PSPICE
<i>source.mo</i>	WAVEFORMS SOURCE
<i>special.mo</i>	SPECIAL

Table 4. Complete list of files and packages.

3. SETTING INITIAL CONDITIONS

Adopting the PSpice methodology (OrCAD, 1999), two equivalent procedures are provided to specify the analysis initial conditions:

- Setpoint pseudo-components: IC1 and IC2 (see Fig. 1, SPECIAL package). IC1 is a one-pin symbol that allows setting the initial voltage on a node. IC2 is a two-pin symbol that allows setting the initial voltage between two nodes.

- The IC property of capacitors (inductor model is not included in this library release).

IC property allows associating the initial condition with a device, while the IC symbols allow the association to be with a node or a node pair. Note that these ways of specifying the simulation initial condition substitute the Dymola standard procedures to set the initial value of the state variables.

Two operations require the static model solution:

- *bias point* calculation (during OP and AC), and
- *transient initial condition* calculation.

When the transient initial condition calculation is skipped (a Boolean parameter controls this option), the devices with the IC property defined start with the specified value. However, all other such devices have an initial state of zero. IC symbols are ignored.

IC symbols clamp the voltage for the entire bias point calculation. PSpice attaches a voltage source with a 0.0002 ohm series resistance (*R_EPS*) at each net to which an IC symbol is connected. This is the set-up of the IC-symbol Modelica model. The model of the capacitor IC-property depends on whether the bias point is calculated or the calculation is skipped:

- During the bias point calculation, the capacitor IC property is implemented using an IC2 symbol in parallel with the capacitor. The capacitor model contains this voltage-clamp circuit.
- When the initial transient solution is skipped, the capacitor voltage is initialised to its IC value using a “when clause”.

Control signals have been defined to set the state (open/close) of the IC symbols switches, initialise the capacitor voltage drop, etc.

4. DEVICE MODELS

Resistor, capacitor and independent source models are discussed.

4.1. Interface

Device models are composed of three formulations: static, AC small-signal and large-signal. Each model formulation is described by its own set of equations and variables. Pin model is conceived to allow the simultaneous connection of the three formulation terminal variables. AC small-signal currents and voltages (complex numbers) are represented in rectangular coordinates (i.e., real and imaginary). The current is positive when flows into the pin.

The interface of the two-pin devices is composed of two *Pin* connectors. PSpice sign criterion for current is adopted: *positive current flows from the (+) node through the device to the (-) node*.

(File: *interface.mo*)

```
connector Pin
  Voltage vDC "Static model";
  Voltage vTran "Large-signal model";
  Voltage vAC_Re "AC small-signal";
  Voltage vAC_Im "AC small-signal";
  flow Current iDC "Static model";
  flow Current iTran "Large signal";
  flow Current iAC_Re "AC small-signal";
  flow Current iAC_Im "AC small-signal";
...
end Pin;
```



```

partial model TwoPin
  Pin    p    "(+) node";
  Pin    n    "(-) node";
  ...

```

4.2. Linear resistor

Resistor static model is shown in Fig 2. The purpose of the IC1-like circuits (switches, R_{EPS} resistors and voltage sources) is clamping the DC-formulation voltage at the pins. The bias point calculation algorithm “dynamic model ramping” requires the following operation: clamping the DC-formulation voltage to the instantaneous value of the large-signal formulation. The `ctrl_RBREAK_Tran2DC` signal controls this information transfer between formulations. When `ctrl_RBREAK_Tran2DC` becomes true:

- The source voltages ($vDCclampP$ and $vDCclampN$) are set to the instantaneous value of the transient voltage at the correspondent pin. Then source voltages are held constant.
- The switches are closed. They remain closed only while the signal is true.

The large-signal and AC small-signal models do not include these IC1-like circuits.

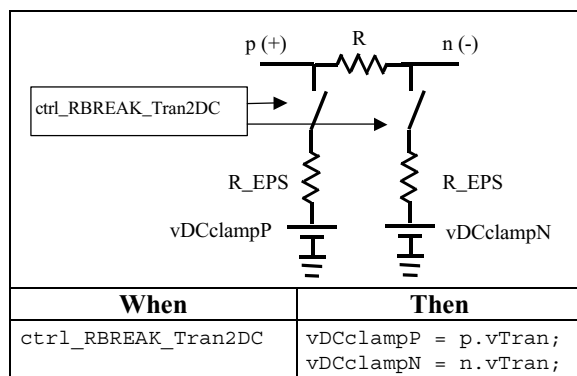


Figure 2. Resistor static model.

4.3. Capacitor

Linear and voltage-dependent capacitors have to be modelled. The partial model *Capacitor* describes all the capacitor behavior except its large-signal and AC small-signal capacitance. *Cbreak* model (linear capacitor) and MOS1 capacitors extend *Capacitor*.

Capacitor static-formulation is shown in Fig. 3. The implementation of the IC property requires the IC2-like circuit (switch, R_{EPS} resistor and $vClampDC$ source). Large-signal formulation is shown in Fig. 4. IC2-like circuit is also included because the “dynamic model ramping” algorithm uses the large-signal formulation during the bias point calculation. The Boolean signals

- `ctrl_IC_clampDC`, and
- `ctrl_IC_clampTran`.

controls the static and large-signal model switches respectively.

The capacitor parameter `IC_ENABLED` enables or disables the IC property. It allows distinguishing between the cases when IC is intentionally set to zero and those cases when the IC property is not enabled (and its by-default value is also zero).

The signal `ctrl_IC_mode` controls $vClampDC$ and $vClampTran$ voltages. Some bias point calculation algorithms need the independent sources ramping from zero up to their nominal initial values. When implementing these algorithms, the voltage clamping sources of the IC symbols and the capacitor IC property need also be ramped from zero to their respective IC values. Two cases are distinguished:

- `ctrl_IC_mode==0`, the clamping voltage ($vClampDC$ or $vClampTran$) is constant and equal to the IC value.
- `ctrl_IC_mode==1`, the clamping voltage is ramped from zero up to its IC value.

In addition, control signals trigger instantaneous changes in the capacitor large-signal voltage drop (see Fig. 4).

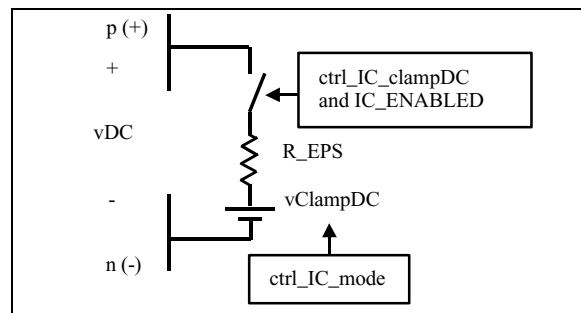


Figure 3. Capacitor static model.

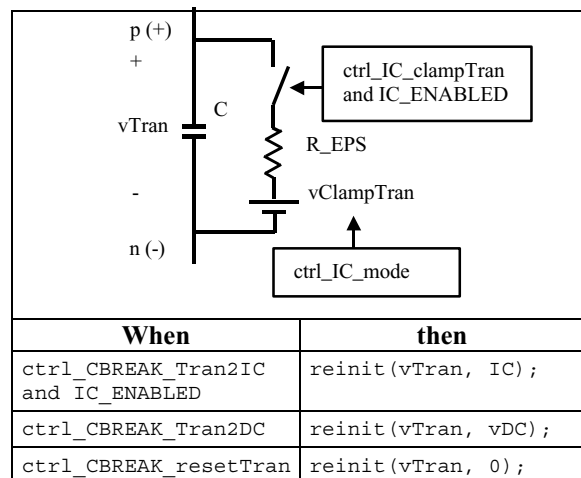


Figure 4. Capacitor large-signal model.

4.4. Independent sources

There are a lot of similarities between the models of the voltage and the current independent sources:

- the interface,
- the DC and transient analysis signals, etc.

The elements in common are defined in the partial model *Stimulus* (SOURCE package) and the source models (*VSource* and *ISource*, see Fig. 1) inherit it.

Source model parameters allow defining the DC and AC characteristics of the source:

- DC analysis: `DC_VALUE`.
- AC analysis: `AC_MAG` and `AC_PHASE`.

Time-dependent waveforms used in the transient analyses are defined in the WAVEFORMS package (see Fig. 1): EXP, PULSE and PWL. PSpice standard has been adopted for waveform parameter names. The *Stimulus* model inherits the waveform model as

a *replaceable model*. Therefore, the waveform model can be declared when instantiating the source model (no waveform is selected by default). Some examples are provided in Table 5.

DC and AC specifications: SOURCE.VSource V1(DC_VALUE=3, AC_MAG=10, AC_PHASE=45);
EXP waveform: SOURCE.VSource V1(DC_VALUE=3, AC_MAG=10, AC_PHASE=45, redeclare model TransientSpecification = WAVEFORMS.EXP(S1=1,S2=2,TD1=1,TC1=1, TD2=3,TC2=1));
PULSE waveform: SOURCE.VSource V1(DC_VALUE=3, AC_MAG=10, redeclare model TransientSpecification = WAVEFORMS.PULSE(S1=1,S2=2, TD=1,TR=1, PW=3,TF=1, PER=8));
PWL waveform: SOURCE.VSource V1(DC_VALUE=3, AC_MAG=10, AC_PHASE=30, redeclare model TransientSpecification = WAVEFORMS.PWL(signalCorners = { 1, 2, 4, 8, 16 }, timeCorners = { 0, 1, 2, 3, 4 }));

Table 5. Examples of source instantiations.

DC analysis

The control signal `ctrl_DC` enables or disables the DC model:

- While `ctrl_DC==false`, the DC value of all the independent sources of the circuit is zero.
- While `ctrl_DC==true`, the DC value of the sources is determined by the integer parameters:
 - `ctrl_OP_mode`, and
 - `ctrl_OP_value`.

In order to set the source value when calculating the initial transient condition, a parameter is associated to each waveform model: `TRANS_INITIAL`. This parameter coincides with the waveform initial value.

The parameter `ctrl_OP_value` determines the source value during the static model solution:

- `ctrl_OP_value==0`: source value is `DC_VALUE`.
- `ctrl_OP_value==1`: value is `TRANS_INITIAL`.

The parameter `ctrl_OP_mode` determines the mode of reaching the previous value:

- `ctrl_OP_mode==0`: the source is hold constant to the value.
- `ctrl_OP_mode==1`: the source value is increased linearly from zero with a slope equal to the value divided by `TIME_SCALE`.

The “dynamic model ramping” algorithm requires the cancellation of the independent sources. The control signal `ctrl_IS_inhibit` allows this operation. While it is true:

- voltage independent sources are substituted by opens (current=0), and
- current independent sources by shorts (voltage=0).

Transient analysis

The control signal `ctrl_Tran` determines:

- whether the transient analysis is enabled, and the source signal is calculated of its associated waveform (`ctrl_Tran==true`),
- or the static bias point calculation is enabled (`ctrl_Tran==false`). The algorithm “dynamic model ramping” requires the circuit large-signal model simulation in order to calculate a “good” initial value for static model iteration.

While `ctrl_Tran==false`, the source value is determined by the parameter `ctrl_IS_TranOP`:

- While `ctrl_IS_TranOP==false`, the value is zero.
- While `ctrl_IS_TranOP==true`, the value depends on the parameters `ctrl_OP_mode`, and `ctrl_OP_value`. The response associated to these parameters is the same than the previously discussed for the static formulation.

AC small-signal analysis

While the control signal `ctrl_AC` is true, the AC small-signal value of the source is set according to the source parameters `AC_MAG` and `AC_PHASE`. Otherwise, the value is zero.

Model of the disabled formulations

It is important to notice that while a model formulation is not enabled, the correspondent values of the independent sources are zero. In this situation, the circuit node voltages are trivially calculated and the simulation computational effort is not unnecessarily increased. The control signals that enable each of the three formulations are:

- `ctrl_DC`,
- `ctrl_Tran`, and
- `ctrl_AC`.

Total power dissipation

The bias point calculation includes the evaluation of the total power dissipation. It is calculated adding the contribution of all the independent voltage sources:

$$W_{DC} = \sum_{\substack{\text{all indep.} \\ \text{V sources}}} v_{DC}(-i_{DC})$$

The calculation is implemented thanks to the Modelica capability of describing “physical fields” (see Table 6). The `PowerDisipation` connector is defined. The model of the voltage source contains:

- an instantiation of this connector,
- the declaration of an outer connector of this type,
- the connection between them.

The “environment” (inner) connector is defined in the *BiasPointCalculation* model.

4.5. Log of analysis results

The analysis results are logged to the *dslog.txt* file using the Dymola’s *LogVariable* function. Two parameters control this information log:

- **LOG_RESULTS** (global parameter). It allows specifying the required detail level at logging results (see Table 7).
- **HIDDEN_COMPONENT**. This device-dependent parameter classifies the circuit devices into two types: those whose variables have to be logged always (**HIDDEN_COMPONENT==false**), and those whose variables have to be logged only in special cases (**HIDDEN_COMPONENT==true**).

The complex AC small-signal voltages and currents are logged in Cartesian and polar coordinates. In addition, the polar magnitude is also expressed in decibels (defined as $20\log_{10}()$).

(File: <i>interface.mo</i>)
connector PowerDisipation
flow Power disipatedPower;
...
(File: <i>source.mo</i>)
model VSource
...
outer INTERFACE.PowerDisipation
TotalPowerDisipation;
INTERFACE.PowerDisipation powerDisipation;
...
equation
when ctrl_log_DC then
powerDisipation.disipatedPower =
vDC*(-iDC);
end when;
connect (powerDisipation,
TotalPowerDisipation);
...
(File: <i>analyses.mo</i>)
partial model BiasPointCalculation
inner INTERFACE.PowerDisipation
TotalPowerDisipation;
...

Table 6. Total power dissipation calculation.

	HIDDEN_COMPONENT	
	False	true
Voltage at resistor pins	0, 1, 2	2
Current through independent voltage sources	0, 1, 2	2
Total power dissipation	0, 1, 2	2
Voltage drop at resistors	1, 2	2
Current through resistors	1, 2	2
Power dissipation of each independent voltage source	1, 2	2

Table 7. LOG_RESULTS values producing the variable log as a function of HIDDEN_COMPONENT value.

5. ANALYSES

PSpice OP, AC and TRAN analyses are translated into Modelica language. Note that analysis models force the simulation end when they have completed their operations (*terminate* function is used). Large simulation times should be selected in the Dymola program window to avoid interfering with analysis execution.

5.1. Bias point calculation

PSpice provides three alternative algorithms for solving the circuit static model (OrCAD, 1999):

- static model iteration,
- static model ramping, and
- GMIN stepping.

PSpice first tries to solve the static model of the circuit using the Newton-Raphson algorithm. If a solution is not found and “GMIN stepping” is enabled (using .OPTION STEPGMIN) then GMIN algorithm is applied. If it also fails or it is not enabled then “static model ramping” is applied. In addition to these three algorithms, a fourth one is programmed in the *BiasPointCalculation* model: the “dynamic model ramping” algorithm, proposed in (Cellier, 1991). The SOLVE_STATIC parameter determines which of the four algorithms to use.

Two control signals, internal to the analysis models, are defined to synchronize the bias point calculation with other analysis operations:

- **biasPoint**. Its transition from false to true indicates that the static-model solution must start.
- **biasPointCalculated**. When the static-model solution is just finished, it becomes true.

The *BiasPointCalculation* model reads the value of **biasPoint** signal and writes **biasPointCalculated**.

Next, the four algorithms are briefly discussed. The control signal transitions required for algorithm completion are shown, but for the sake of clarity, their cause-effect relationships are omitted. Two additional comments:

- **ctrl_OP_value** signal is not written by the bias point calculation algorithms.
- Control signals evaluated at bias point calculation (see Table 1) and hold to false during the whole algorithm, are omitted.

Static model iteration (SOLVE_STATIC:=0)

The solution of the static problem is left in hands of the modelling language. PSpice has two symbols to provide an initial guess for Newton-Raphson algorithm: NODESET1 and NODESET2 (OrCAD, 1999). These symbols have not been translated into Modelica language because they do not represent any advantage compared to Dymola Initial Calculation methods (Elmqvist et al., 2001). The Modelica implementation of the algorithm is shown in Fig. 5.

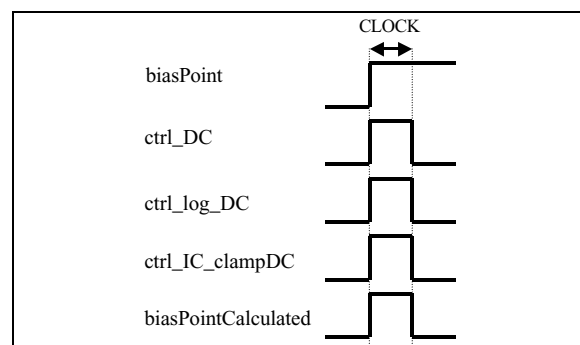


Figure 5. Static model iteration algorithm.

Static model ramping (SOLVE_STATIC:=1)

PSpice cuts back the power supplies to almost zero (0.001%) so that all non-linearities are turned off. When the circuit is linear, a solution can be found (very near zero, of course). The initial condition of this first step is zero for all voltages. Then, PSpice works its way back up to 100% power supplies using

a variable step size (OrCAD, 1999). The process relies heavily on the equation continuity with respect to the power supplies.

This algorithm is translated into Modelica language ramping the static-formulation value of the independent sources from zero up to their target values. The clamping voltages of the IC symbols and the capacitor IC property are also adequately ramped. The value of the parameter `TIME_SCALE` determines the length of the ramping. The algorithm is implemented by means of the signal transitions shown in Fig. 6.

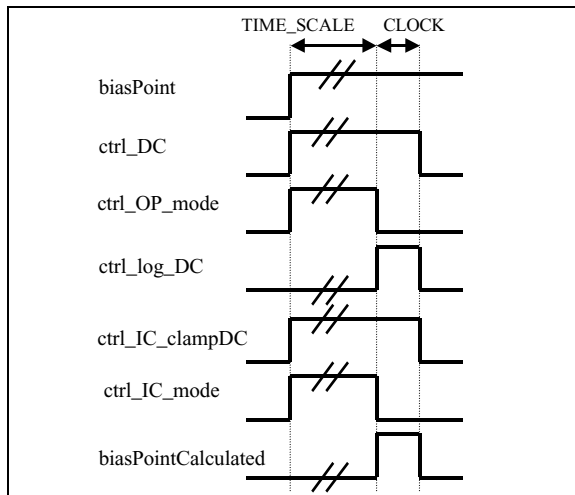


Figure 6. Static model ramping algorithm.

GMIN stepping (SOLVE_STATIC:=2)

GMIN stepping attempts to find a solution for the static model (with power supplies at 100%) by starting with a large value of GMIN, initially 1.0×10^{10} times the nominal value. If a solution is found at this setting, PSpice reduces GMIN by a factor of 10 and tries again. This continues until either GMIN is back to the nominal value, or a repeating cycle fails to converge. This algorithm makes heavy use of equation continuity with respect to GMIN model parameters. The Modelica implementation of this algorithm is shown in Fig. 7.

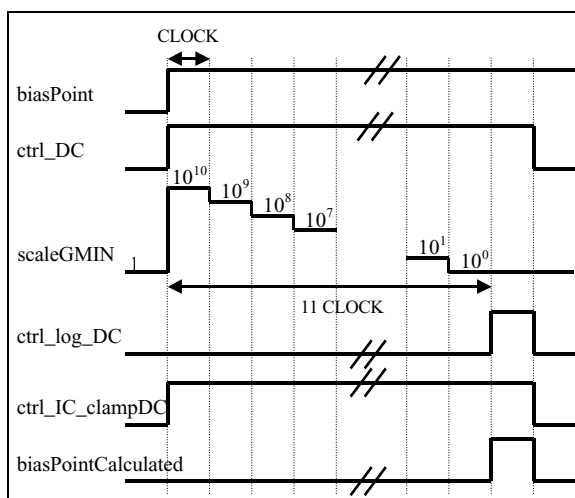


Figure 7. GMIN stepping algorithm.

Dynamic model ramping (SOLVE_STATIC:=3)

The initial condition to iterate the static model is obtained by simulating the large-signal model (Cellier, 1991). A transient analysis is performed: all sources are ramped up from zero to the desired initial value for the simulation and this value is held for some time to allow the circuit to stabilise. Then the large-signal formulation voltages are transferred to the static model (using `ctrl_RBREAK_Tran2DC` and `ctrl_IS_inhibit`). This static-circuit setting is held for a clock cycle. Then, the power supplies are connected to the circuit, the resistor voltage-clamping circuits are disconnected, and the static model is solved. The Modelica implementation of the algorithm is shown in Fig. 8.

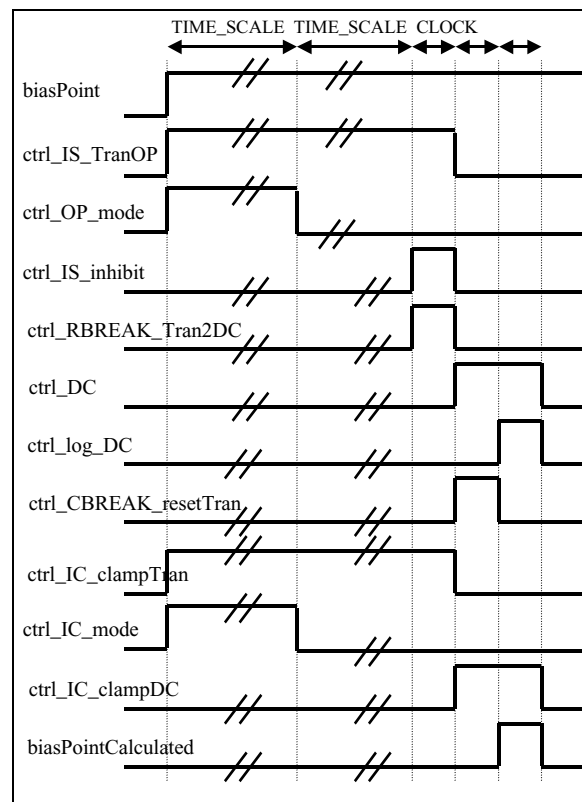


Figure 8. Dynamic model ramping algorithm.

5.2. Bias point analysis (OP)

The OP analysis (see Fig. 9):

- forces the `biasPoint` signal to become true,
- sets `ctrl_OP_value` signal to zero, and
- finish the simulation one clock cycle after the `biasPointCalculated` signal becomes true.

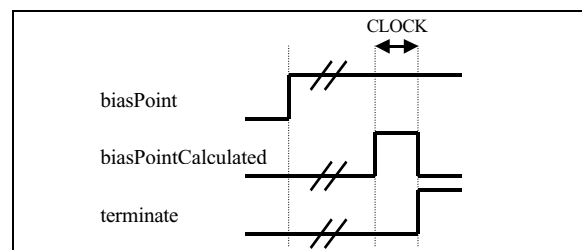


Figure 9. OP analysis signals.

Example

Consider the application of the OP analysis algorithms to the trivial circuit shown in Fig 10. Dymola's experiment *StopTime* variable is set to an arbitrary large value: 100. The *TIME_SLOT*, *TIME_SCALE* and *LOG_RESULTS* parameters are left to their by-default values: 10%, 1s and 0 respectively.

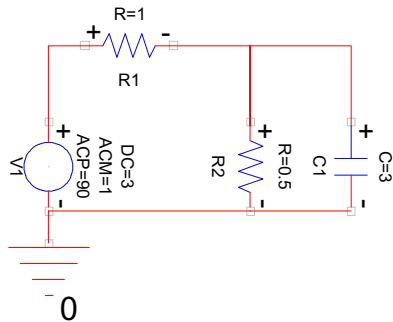


Figure 10. Simple example of a RC circuit.

- **SOLVE_STATIC:=0.** Once finished the simulation (at $T=0.1$), *dslog.txt* file contains the results:

```
V1_iDC(1e-010) = -2
V1_vDC(1e-010) = 3
R1_n_vDC(1e-010) = 1
ctrlx_0logx_0DC(1e-010) = 1
V1_powerDisipation_disipatedPower(1e-010)=6
```
- **SOLVE_STATIC:=1.** The *dslog.txt* file contains the results, logged at $T=1$. The simulation terminates at $T=1.1$ (see Fig 11).
- **SOLVE_STATIC:=2.** The circuit does not contain any device with the *GMIN* parameter, so this algorithm is equivalent to *SOLVE_STATIC:=0*. Results are logged at $T=1.1$ and the simulation finishes at $T=1.2$ (see Fig. 12).
- **SOLVE_STATIC:=3.** Results are logged at $T=2.2$ and the simulation finishes at $T=2.3$. Large-signal and static voltages at *R1.n* node are shown in Fig. 13. At $T=2.0$: large-signal to static info. transfer. At $T=2.1$: Static model solution.

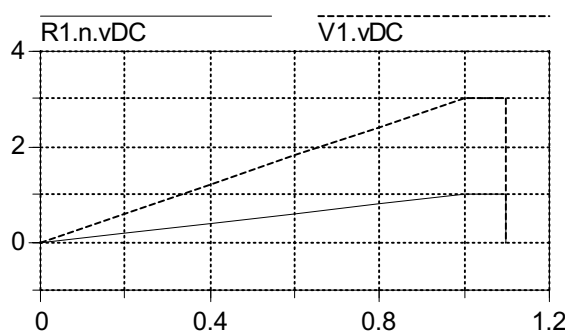


Figure 11. Voltage at circuit nodes.

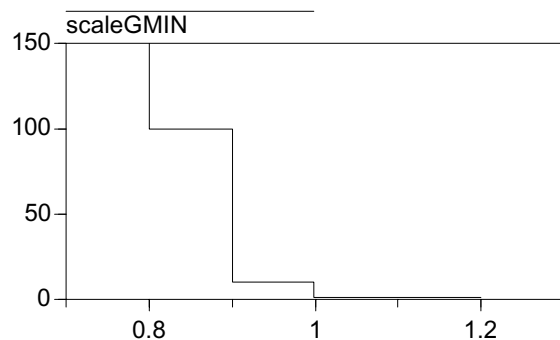


Figure 12. GMIN scale factor

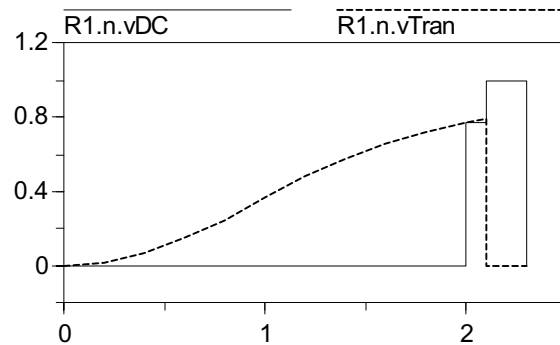


Figure 13. Static and large-signal voltages.

5.3. AC sweep analysis (AC)

The *TYPE_AC_SWEEP* parameter defines the frequency sweep type (LIN and DEC PSpice arguments):

- *TYPE_AC_SWEEP:=0*: frequency linear sweep.
- *TYPE_AC_SWEEP:=1*: the frequency is swept logarithmically by decades.

AC small-signal analysis (see Fig. 14):

- forces the *biasPoint* signal to become true, and
- sets *ctrl_OP_value* signal to zero.

When *biasPointCalculated* becomes true, the AC analysis:

- forces *ctrl_AC* to become true, enabling the AC model.
- Starts the frequency sweep. The frequency variation in time depends on the sweep type. In both cases, the required log frequencies are spaced at regular time-intervals of length $2 \times \text{CLOCK}$. Therefore, the *ctrl_log_AC* signal is a pulse train of period $2 \times \text{CLOCK}$.

The simulation is finished one clock cycle after the frequency reaches *END_FREQUENCY*. An AC analysis of the Fig 10 circuit is shown in Fig 15.

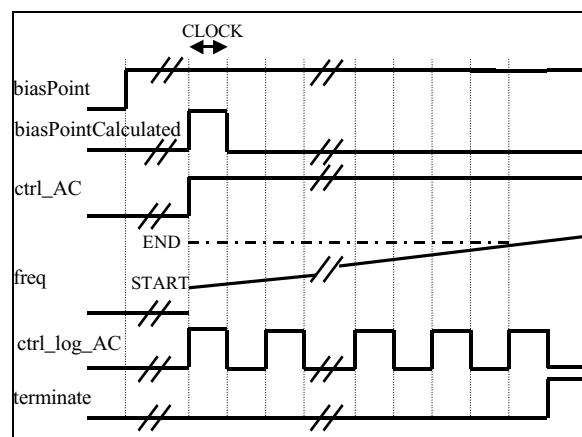


Figure 14. AC analysis implementation.

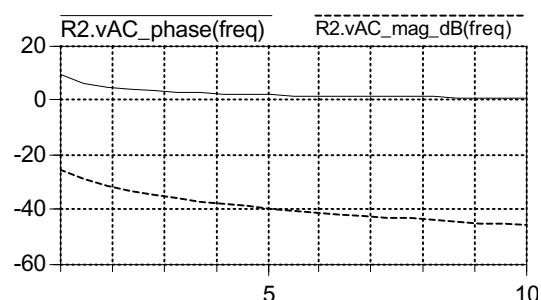


Figure 15. Example of AC small-signal analysis.

5.4. Transient analysis (TRAN)

When the transient simulation is started, the value of the `time` variable is different of zero. For this reason, a variable is defined to measure the transient simulation time: `timeTran`. The length of the transient simulation is set by the `TRAN_STOP_TIME` parameter. The transient analysis depends on the `SKIP_INITIAL_TRAN_SOLUTION` parameter.

`SKIP_INITIAL_TRAN_SOLUTION:=false`

When `biasPointCalculated` becomes true, the circuit static model contains the transient initial solution. Then (see Fig. 16):

- `ctrl_CBREAK_Tran2DC` becomes true. The large-signal circuit state is initialised to the static-circuit voltage values.
- `ctrl_Tran` becomes true. The large-signal device models are enabled.

The simulation terminates when `timeTran` reaches the value `TRAN_STOP_TIME`.

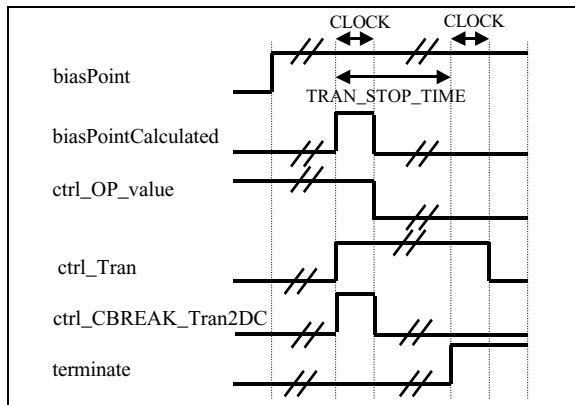


Figure 16. Transient analysis with initial calculation.

`SKIP_INITIAL_TRAN_SOLUTION:=true`

At initial time (see Fig. 17):

- `ctrl_CBREAK_Tran2IC` becomes true. The large-signal circuit state is initialised to the IC-property correspondent values.
- `ctrl_Tran` becomes true. The large-signal device models are enabled.

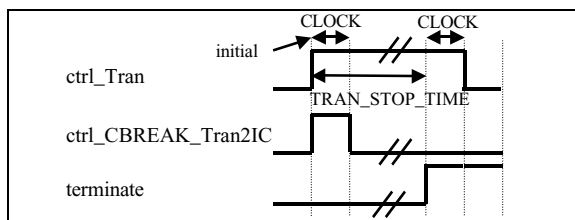


Figure 17. Transient analysis w/o initial calculation.

6. SPICE2 LEVEL 1 NMOS

The SPICE2 level1 MOS model is basically the model proposed by Shichman and Hodges (Massobrio and Antognetti, 1993). The Dymodraw diagram of the model is shown in Fig 18. Each substrate junction is modelled as a voltage-controlled current source (diode-like icon in Fig. 18) in parallel with a voltage-controlled capacitor. I_{DS} is a non-linear current source controlled by the voltages V_{DS} ,

V_{GS} and V_{BS} . The gate capacitance is modelled using three voltage-controlled capacitors: C_{GB} , C_{GS} and C_{GD} .

Voltage-controlled capacitors have been modelled extending the *Capacitor* model. Expressions for the large signal capacitance are provided and the small-signal capacitance is evaluated at the bias point (i.e., when `ctrl_AC` signal becomes true). Large-signal and static formulations of controlled current sources are equal (of course, each one is described by its own set of variables). Their small-signal models (conductance) are evaluated at bias point.

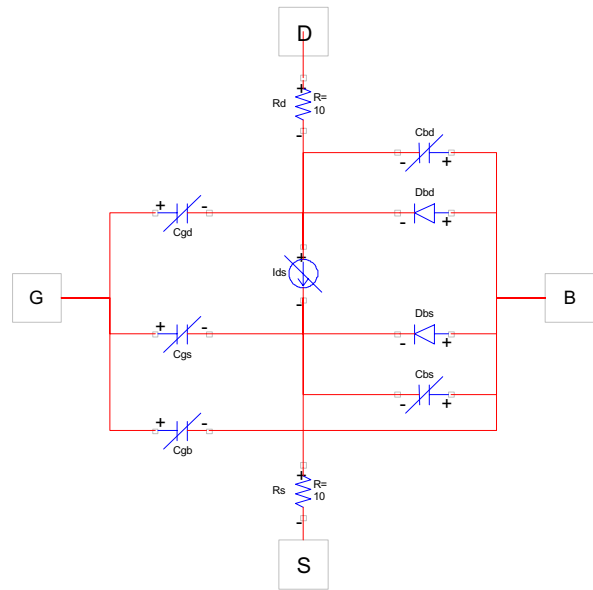


Figure 18. SPICE2 level1 NMOS

Conclusions

A reduced set of SPICE device models has been successfully translated into Modelica language for OP, AC and transient analyses.

References

- Aström, K. J., H. Elmqvist and S. E. Mattsson (1998). *Evolution of Continuous-Time Modeling and Simulation*. 12th ESM, Manchester, UK.
- Cellier, F. E. (1991). *Continuous System Modeling*. Springer-Verlag.
- Clauss, C., et al. (2000). *Modelling of Electrical Circuits with Modelica*. Modelica Workshop 2000, Lund, Sweden.
- Elmqvist, H., F. E. Cellier and M. Otter (1993). *Object-Oriented Modelling of Hybrid Systems*. ESS'93, Delft, The Netherlands.
- Elmqvist, H., et al. (2001). *Dymola. Dynamic Modeling Laboratory. User Manual*. Dynasim.
- Kielkowski, R.M. (1998). *Inside SPICE*. McGraw-Hill, Inc. Second Edition.
- Massobrio, G and P. Antognetti (1993). *Semiconductor Device Modeling With SPICE*. McGraw-Hill, Inc.
- Modelica (2000). *Modelica, Language Specification & Tutorial*. Modelica Association.
- OrCAD. (1999). *OrCAD PSpice A/D. Reference Guide & User's Guide*. OrCAD, Inc.

Simulating permanent magnet brushless motors in DYMOLA

G. Ferretti, G. Magnani, P. Rocco
Politecnico di Milano, Dipartimento di Elettronica e Informazione,
Piazza L. da Vinci 32, 20133 Milano, Italy

L. Bonometti, M. Maraglino
C.M.S. S.p.A., Via A. Locatelli 49, 24019 Zogno (BG), Italy

Abstract

Multi-domain dynamic simulation is becoming an issue in the design of high performance mechatronic systems, where advances are foreseen only if the mutual interaction of different parts of the system is well understood. The modelling environment provided by DYMOLA with Modelica language proved to be ideal for studying the mutual effects of mechanics, electronics and control in a brushless motor, whose model has been conceived as one of the building blocks of a wider project, aimed at simulating a complete machining centre. Details on the model of the brushless motor as well as on its simulation are given in the present paper.

1 Introduction

The most common actuation systems adopted in robotics, machine tools industry and machining centers are by far servomechanisms with permanent magnet brushless motors, connected to the loads by transmission chains (or gearboxes).

In a brushless motor the electromechanical commutation typical of brushed DC motors is replaced by an electronic commutation of the currents in the three phases of the stator windings. This should in principle guarantee that the electromagnetical torque delivered on the motor shaft is independent of the rotor position. However some constructive imperfections in the motor or in the drive, where electronic commutation is implemented, produce an undulation (ripple) [4] on the actual torque. While this problem could be considered minor in the static dimensioning of the actuation system, it is of utmost importance for its dynamic performance. Torque ripple might in fact excite the resonances of the mechanical system, usually associated to the elastic couplings between motors and loads.

Dynamic simulation [2], or virtual prototyping in a more recent jargon, is a valuable tool to study these phenomena, and in particular to separate the effects of the single sources of disturbances on the performance of the system. Mechanics, electronics and control are different domains involved in this truly mechatronic problem. *Multi-domain simulation environments* are required to simulate with a reasonable effort the system, while the particular electrical configuration of the stator windings (Y connected) calls for the adoption of modelling languages where *algebraic constraints* on state variables can be easily specified.

DYMOLA (with Modelica language [7, 5]) has been found to fit easily both the above requirements. Mechanical, electrical and control systems can be combined in a natural and physics-driven way, while the acausal modelling based on DAE equations, proper of this environment, allows to specify the constraint on the phase currents as it is, avoiding reformulation of the system's equations in terms of two out of three currents, typical of procedural modelling languages.

In the present work DYMOLA has been used to simulate a brushless motor controlled with an analogue driver and with a full digital driver. The simplified model ([3]) of torque ripple has been validated through these simulations. The model of the brushless motor with its analogue or digital drivers has been actually used as one of the building blocks of a wider project, where the simulation of a complete machining center (detailed simulation of the mechanical parts of the system and of various features of the CN) has been implemented.

2 Torque ripple modelling

The functional scheme of a sinusoidal PMAC machine is represented in Fig. 1. If a reference torque $\bar{\tau}$ should be delivered by the motor, typically as re-

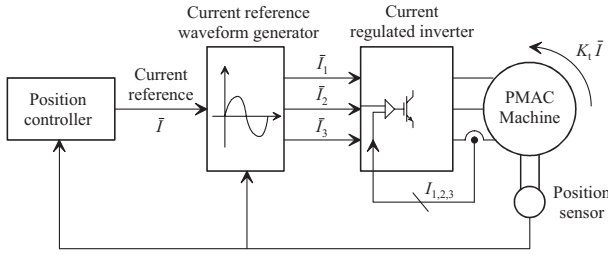


Figure 1: Functional scheme of a brushless motor

quired by a position controller, the current reference \bar{I} has to be given the value $\bar{I} = \bar{\tau}/K_t$, where K_t is the torque constant. This scalar setpoint is then modulated through three sinusoidal functions of the electrical angle $\alpha = pq_m$, p being the number of pole pairs and q_m being the motor angle, that are offset by an angle $2\pi/3$ one from each other. The three resulting signals become the current references for the three phases. High bandwidth current controllers make the currents track their setpoints in each phase (actually two out of the three Y connected phases are closed loop controlled). If the current reference in each phase is given the same dependence on the electrical angle characterizing the back EMF (ideally sinusoidal or trapezoidal), a torque τ is produced, approximately equal (in a band of frequencies limited by the current loops) to the desired torque $\bar{\tau}$, and thus proportional to the scalar current reference \bar{I} .

Brushless motors, however, introduce a disturbance in the system in the form of a ripple on the torque. Several constructive imperfections of the motor and the servodrive sum up to form this pulsating disturbance. Examples are cogging torque, offsets in the current sensors, imperfections in the construction of the motor and the drive, implying that both the back EMF profiles and the phase currents may be affected by undesired higher order harmonics.

As it is shown in [3], the following relation can be used to represent in a compact form the effects of the disturbances on the torque production:

$$\tau = \tau(\alpha, \bar{I}) = \gamma(\alpha) + K_t \bar{I} (1 + \delta(\alpha)) \quad (1)$$

The term $\gamma(\alpha)$ accounts for the disturbances due to the cogging torque and to the current offset in the drives, while the second term is responsible for the nominal torque (with $\delta(\alpha) = 0$) and for the disturbances related to the harmonic content. It is also possible to include in $\delta(\alpha)$ the effects of the amplitude imbalances and the phase misalignments of the current and back EMF shapes profiles [3].

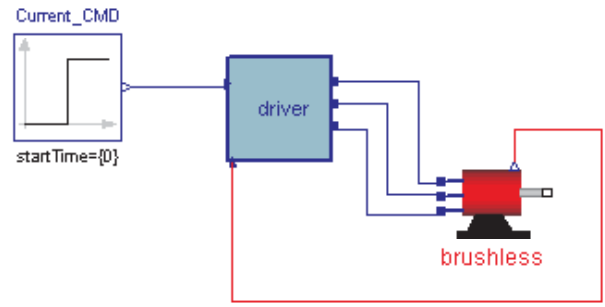


Figure 2: Complete model of the system

3 Modelling the system in DYMOLA

The model of the system is obtained by the feedback connection of two sub-models, one representing the brushless motor, the other one the driver (Fig. 2). The two models are connected through three electrical connectors (the three phases of the motor) as well as through a control connector (the measure of the rotor position).

The brushless model is shown in Fig. 3. The three phases are Y connected in the block emf3, that generalizes the EMF model in the Modelica.Electrical.Analog.Basic library. In the emf3 model the back-emf profiles on the single phases are assigned. The nominal sinusoidal profiles can then be modified to study ripple due to higher order harmonics. The torque at the flange of the emf3 model derives from the equilibrium with the sum of the products of currents and back emf profiles on the single phases. Remarkably, the acausal modelling environment provided by DYMOLA allows to specify in the most natural way the algebraic constraint on the currents (the sum of the currents must be zero). This constraint would obviously generate troubles in other simulation environments based on causal specifications of the models, expressed with ODE systems. Just for comparison, Fig. 4 shows the SIMULINK model of the electrical part of a brushless motor, obtained by resolving the algebraic constraint and expressing the whole model in terms of two out of three currents. The derivation of the model is time consuming and error prone and the result lacks readability.

Modelling of the mechanical part of the motor is on the other hand standard.

The analogue version of the current controller is shown in Fig. 5. The current reference (usually the output of a position/velocity controller) is modulated through sinusoidal functions of the electrical angle, to form the references for two of the three Y connected currents. Current sensors are included in the drive and

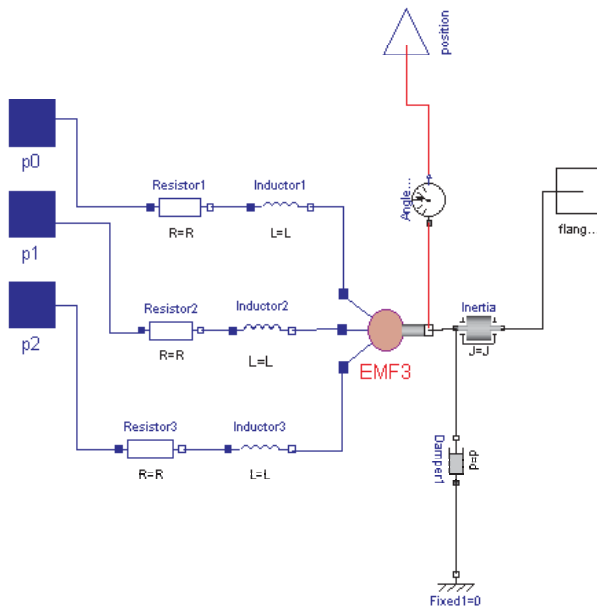


Figure 3: Model of the brushless motor

possible offsets can be added to the measures, in order to simulate their effects on the generation of torque ripple. Two anti-windup PI controllers close the current loops. Their outputs are then linearly amplified and form the voltages to be applied to the single phases of the motor.

A full digital version of the current controller has been implemented as well (Fig. 6). A vector control scheme [6] has been adopted, where the phase currents are first transformed into direct and quadrature currents through Park's transformation. Two digital loops are closed on these currents, the quadrature reference being the output of the outer position/velocity controller, the direct reference being zero as usual. The voltage commands output of the two antiwindup PI controllers are then back transformed to voltages on the three phases through inverse Park's transformation.

The PWM amplifier has not been simulated since, operating with a frequency 10 or 20kHz and with a modulation of the pulse width of $1\mu s$, it requires an integration step size less than $1\mu s$, which might be acceptable for the simulation of the electronics of a motor drive but is far too small in the combined simulation of the mechanics and the electronics. This is particularly true if the model of the motor is instantiated several times, for the simulation of a complete machine.

4 Simulating the motor without load

Simulations obtained with the analogue version of the driver will be presented here, in order to show the utility of the model. The input of the system is a step on the current reference: the signal is expressed in Volt and has been given the value 1V (corresponding to 10% of the entire scale and to a current of 1.9A). The current-to-torque gain of the motor (K_t) is equal to $1.1Nm/A$, while the number of pole pairs is equal to 3. Both the current loops have been tuned for a bandwidth of 1kHz. The inertia of the motor is equal to $0.012Kg\cdot m^2$ while the damping factor is $0.371Nms/rad$.

Fig. 7 and Fig. 8 show, on different time scales, the responses of the electromagnetic torque and of the motor velocity in nominal conditions. The responses match the expectations¹ both from the transient point of view and from the steady state one (no oscillations is produced).

¹The negative sign of the torque is of no particular meaning, being associated just to the way the balance of torques is written in the block emf3.

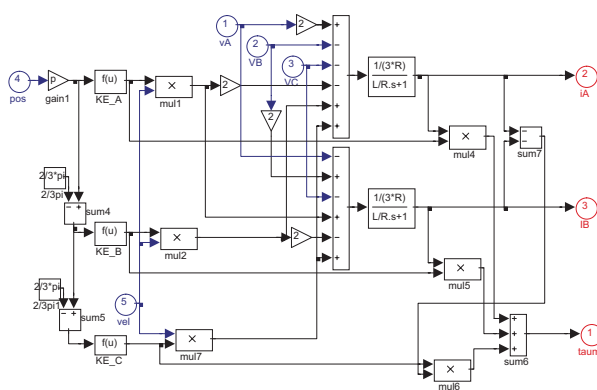


Figure 4: Model of the brushless motor in SIMULINK

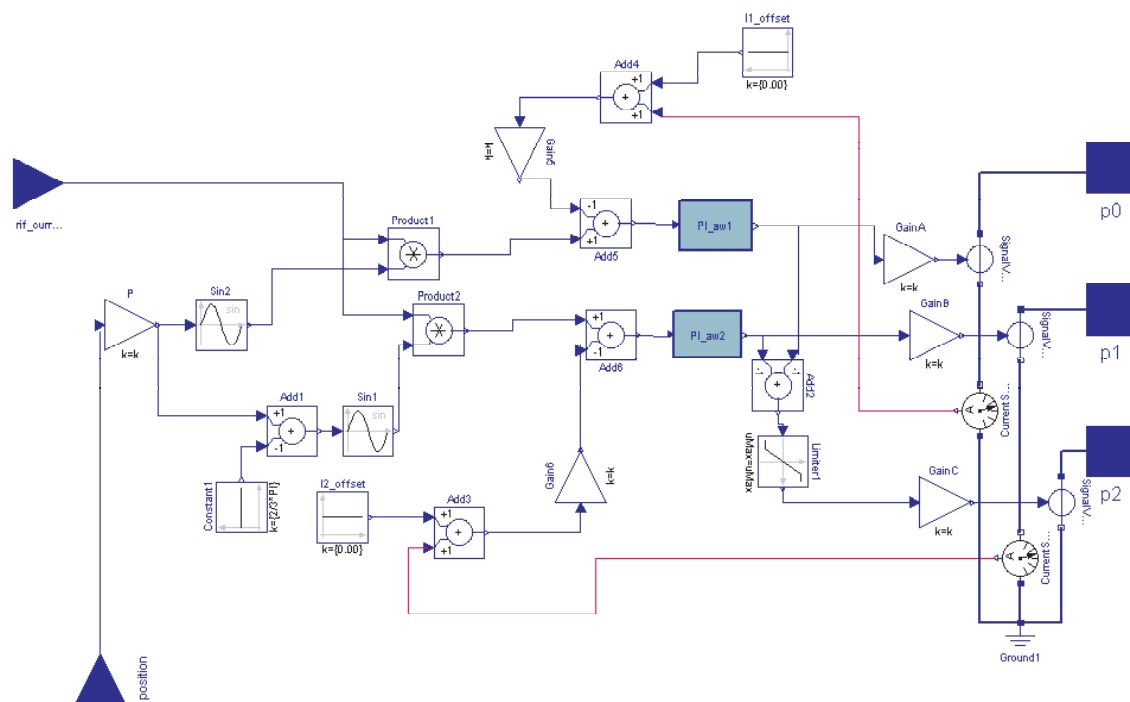


Figure 5: Model of the analogue driver

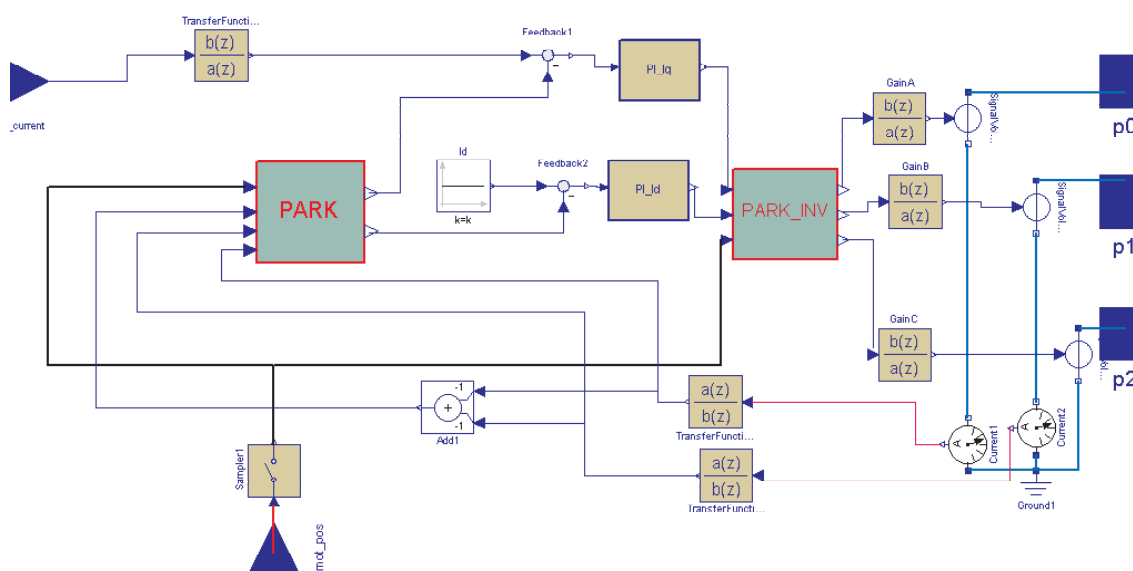


Figure 6: Model of the digital driver

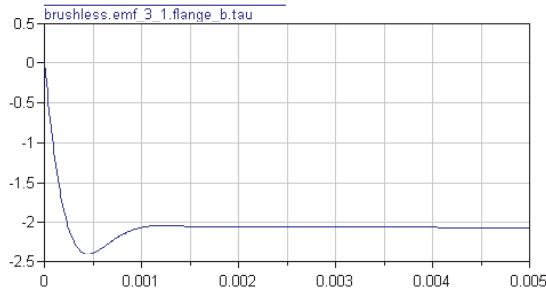


Figure 7: Electromagnetic torque in nominal conditions

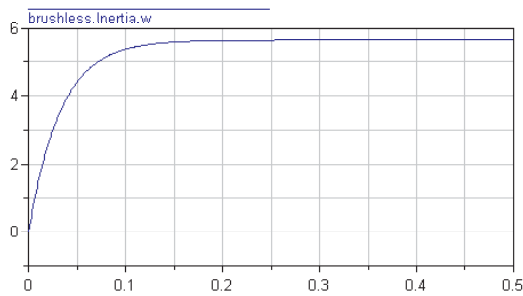


Figure 8: Motor velocity in nominal conditions

In a second simulation, an offset of 1% of the rated current has been introduced on both the current sensors. The resulting electromagnetic torque is shown in Fig. 9. As the average velocity is equal to the value obtained in nominal conditions $\Omega \approx 5.6 \text{ rad/s}$, the periodicity of the disturbance is consistent with theory [3] ($T = 2\pi/(3\Omega) = 0.37 \text{ rad/s}$).

The effect of higher order harmonics in the back e.m.f. profiles has been simulated introducing the fifth harmonic on all the three profiles, with amplitude 5% of the main harmonic and no misalignments or unbalances. The result in terms of electromagnetic torque is reported in Fig. 10. Again the periodicity of the disturbance is consistent with theory [3] ($T = 2\pi/(18\Omega) = 0.062 \text{ rad/s}$).

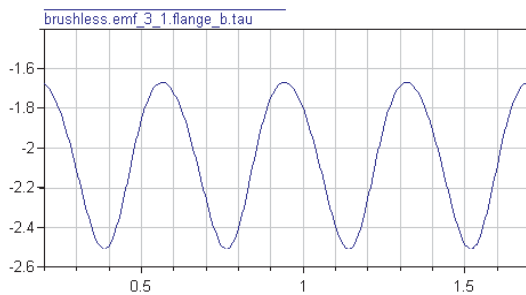


Figure 9: Electromagnetic torque with a current offset

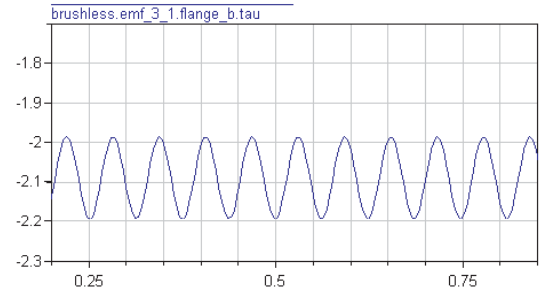


Figure 10: Electromagnetic torque with a high order back emf harmonic

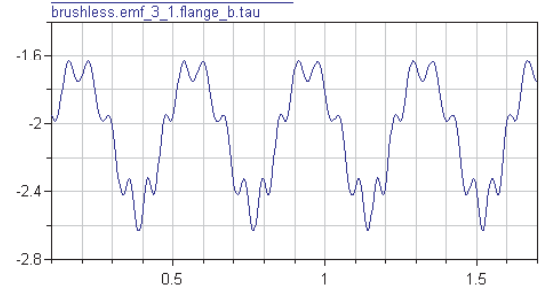


Figure 11: Electromagnetic torque with both disturbances

The superimposition of the two disturbances (the offset on the current sensor and the high order back emf harmonic) yields the electromagnetic torque reported in Fig. 11.

It is not difficult to verify (for example exporting the results of the simulation in Matlab) that the above torque profile corresponds, apart from the sign inversion, to (1), where:

$$\gamma(\alpha) = 3KI_{off} \sin(\alpha + \frac{2}{3}\pi) \quad (2)$$

$$\delta(\alpha) = -\frac{K_5}{K} \cos(6\alpha) \quad (3)$$

where I_{off} is the current offset, K_5 is the amplitude of the fifth harmonic of the back emf profile, K is the amplitude of the main harmonic ($K = 2/3K_f$).

5 Simulating the motor with a load

As already mentioned in the Introduction, one of the reasons why torque ripple deserves accurate modelling and possibly compensation is that it may act as an excitation signal for the usually lightly damped dynamics of the two-mass system made up by the motor coupled with a load through an elastic transmission. As the torque ripple frequency is proportional to the motor

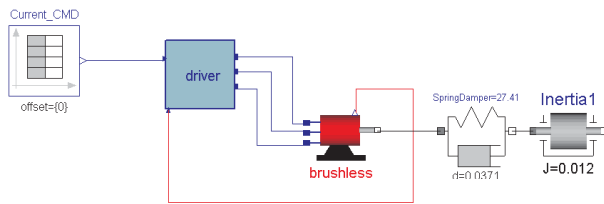


Figure 12: Model of the system including a load

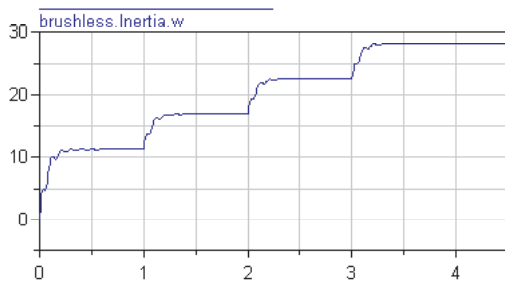


Figure 13: Motor velocity with a load, without ripple

velocity, this problem is particularly critical at those operating velocities when the multiple of the motor velocity is comparable to the natural frequency of the system. In order to confirm this analysis with simulation results, the model of the motor has been coupled to the models of an elastic transmission and a load, both taken from the Modelica.Mechanics.Rotational library (see Fig. 12).

The load has been given the same value as the inertia of the motor while the elastic parameter has been selected so as to have a resonance frequency approximately equal to 70rad/s . In a first simulation, four consecutive steps on the current command have been given, corresponding to 20%, 30%, 40% and 50% of the entire scale, in nominal conditions (i.e. with all the sources of ripple disabled). The result, in terms of the velocity of the motor is shown in Fig. 13, where the natural oscillations due to elasticity are evident, but also damped out by the natural damping on the system.

Then a ripple induced by the same offset on the current sensor as in the previous Section has been introduced. Notice that, as the natural frequency of the system is about 70rad/s , major problems to the system are expected when the average velocity of the motor is about one third (23rad/s) of this value, namely in the third interval of the simulation. The result is confirmed in the plot of Fig. 14, where the effect of the matching between ripple frequency and natural frequency is most evident (once triggered in the third interval, the oscillations remains also in the fourth one).

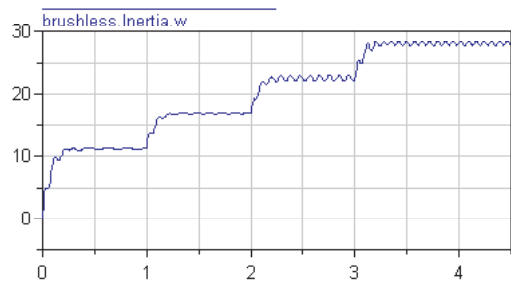


Figure 14: Motor velocity with a load, with ripple

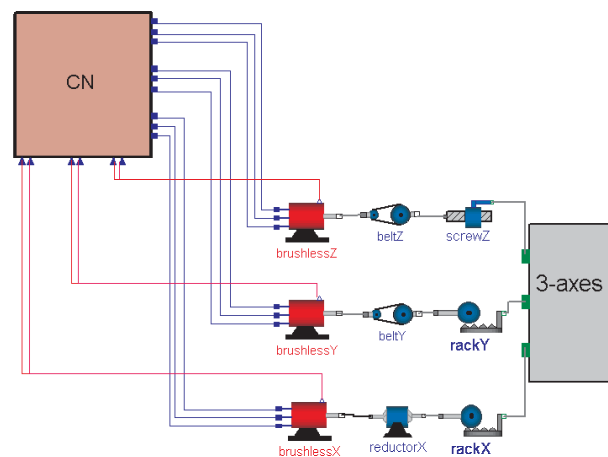


Figure 15: Top view of the simulator of a machining center

6 Use of the brushless motor in the simulation of a machining center

As already mentioned in the Introduction, the model of the brushless motor has been included in a library of elements used to simulate a complete machining center. Fig. 15 shows the top level of the simulator for a three axes machine. The model is composed of three parts: the simulation of the CN and the servodrive, entirely realized with the DYMOLA blocks, the simulation of the transmission chain for each axis, where the brushless motor has been used, and the simulation of the kinematic chain, realized with the blocks of the ModelicaAdditions.MultiBody library.

Again the multi-domain nature of DYMOLA and the physics driven assembly of the model turned out to be essential elements to fulfill the task, namely to realize a reliable simulation environment, easy to use for a non specialist of dynamic modelling.

7 Conclusions

DYMOLA proved to be a valuable tool to specify in the most natural way the model of the three phase brushless motor, in terms of a high index DAE system [1]. Simulations have been run to test various non nominal situations in brushless motors, where torque ripple can occur.

8 Acknowledgements

The support of student A. Samarani in building the model and performing simulations is acknowledged.

References

- [1] Brennan, K.E., S.L. Campbell and L.R. Petzold: *Numerical solution of initial-value problems in differential algebraic equations*, North-Holland (1989).
- [2] Cellier, F.: *Continuous system modelling*, Springer Verlag (1991).
- [3] Ferretti, G., G. Magnani and P. Rocco: Modelling, identification and compensation of pulsating torque in permanent magnet ac motors. *IEEE Transactions on Industrial Electronics*, **46**, (1998), pp. 912–920.
- [4] Jahns, T. M. and W. L. Soong: Pulsating torque minimization techniques for permanent magnet ac motor drives - a review. *IEEE Transactions on Industrial Electronics*, **43** (1996), pp. 321–330.
- [5] Mattsson, S.E., H. Elmqvist and M. Otter: Physical system modeling with Modelica. *Control Engineering Practice*, **6**, (1998), pp. 501–510.
- [6] Texas Instruments: Digital signal processing solution for permanent magnet synchronous motor. *Application Note. Literature Number: BPRA044*, (1997).
- [7] Tiller, M.: *Introduction to physical modeling with Modelica*, Kluwer (2001).

PQLib- A Modelica Library for Power Quality analysis in Networks

Sergej N. Kalaschnikow

VA TECH ELIN EBG Elektronik GmbH

Ruthnergasse 1, A-1210 Vienna Austria

e-mail: s.kalaschnikow@eel.elinebg.at

Abstract

In power supply networks, the quality of the voltage is becoming more and more of a determining factor.

Non-linear loads such as diode or thyristor converters contribute to the degradation of the supply voltage quality. Non-sinusoidal currents of the non-linear loads result in the distortion of the supply voltage wave form at the point of common coupling due to the finite supply impedance.

To improve the power quality of the supply voltage active filters and reactive current compensators are used. The optimal power rating and topology of these units are very important, but also the determination of the best compensation strategy for a specific application is very important as well. Different topologies and strategies can often perform related compensation functions, resulting in a situation where financial implications determine the best solution. In this situation the power quality analysis in network using simulation tools is very useful.

This paper describes a Modelica library called PQLib (**P**ower **Q**uality **L**ibrary) designed for power quality analysis in networks using simulation tools written in Modelica.

The PQLib contains the following components:

- Definition of connectors for three phase networks
- Models for:
 - three phase passive electrical elements like resistor, capacitor and so on.
 - three phase electrical machines and transformers
 - three phase transmission lines
 - semiconductor controlled dc and ac electrical drives

- power factor correction devices (passive filters)
- mains active restoring devices using semiconductors (active filters)
- measuring instruments: true rms voltmeter and amperemeter, digital frequency analyser
- Examples

1. Building of the PQLib

The PQLib is based on the package concept. The package concept was introduced into Modelica to help organize definitions of models, connectors, etc. [1, 2]. Fig.1 shows the components of the package PQLib.

1.1. Types

In the PQLib for currents, voltages and impedances the per unit (p.u.) quantities with the definitions according to [3] are used. Currents and voltages are related to their rated peak- values:



Fig. 1. Components of the package PQLib

$$u(t) = \frac{U(t)}{\sqrt{2} \cdot U_N}, \quad i(t) = \frac{I(t)}{\sqrt{2} \cdot I_N}. \quad (1)$$

Impedances are referred in the same way as (1) to

$$z(\omega) = \frac{Z(\omega) \cdot I_N}{U_N}, \quad (2)$$

with: U_N and I_N are nominal values of the voltage and the current accordingly.

Consequently, the types in the PQLib are defined as (for example for the first phase):

```

type Voltage1stPhase = Real (
  final quantity="Voltage",
  final unit="p.u.",
  displayUnit="p.u.");
type Current1stPhase = Real (
  final quantity="Current",
  final unit="p.u.",
  displayUnit="p.u.");
type Resistance = Real (
  final quantity="Resistance",
  final unit="p.u.",
  min=0,
  displayUnit="p.u.");
type Conductance = Real (
  final quantity="Conductance",
  final unit="p.u.",
  min=0,
  displayUnit="p.u.");
type Reactance = Real (
  final quantity="Reactance",
  final unit="p.u.",
  min=0,
  displayUnit="p.u.");
type SignalAnalog = Real;
type SignalBoolean = Boolean;
type SignalDiscrete = Real;

```

1.2. Interfaces

Usually every package includes some interface definitions which are used throughout the package. In the PQLib package the basic interface definition is the three phase pin, which is a connector. At the pin the pin three phase voltages va, vb and vc and the pin three phase currents ia, ib and ic are defined. The positive pin is described in the following way:

```

connector Pin3Ph
  Voltage1stPhase va;
  Voltage2ndPhase vb;
  Voltage3rdPhase vc;
  flow Current1stPhase ia;
  flow Current2ndPhase ib;
  flow Current3rdPhase ic;
end Pin3Ph

```

The negative pin differs in its graphical representation only.

The TwoPin interface is defined as a partial model:

```

partial model TwoPin3Ph
  PQLib.Interfaces.Voltage1stPhase Vr;
  PQLib.Interfaces.Voltage2ndPhase Vs;
  PQLib.Interfaces.Voltage3rdPhase Vt;

```

```

  PQLib.Interfaces.Current1stPhase Ir;
  PQLib.Interfaces.Current2ndPhase Is;
  PQLib.Interfaces.Current3rdPhase It;

```

```

  PQLib.Interfaces.Pin3Ph P;
  PQLib.Interfaces.NegPin3Ph N;

```

```

equation
  Vr = P.va - N.va;
  Vs = P.vb - N.vb;
  Vt = P.vc - N.vc;
  P.ia + N.ia = 0;
  P.ib + N.ib = 0;
  P.ic + N.ic = 0;
  Ir = P.ia;
  Is = P.ib;
  It = P.ic;
end TwoPin3Ph

```

For the control package of the PQLib the analog and digital as well as logical interfaces are defined in the classical way of the Modelica interface definition with the exception of the definition for three phase vectors [3]. For example, the vector of voltages \mathbf{u} is derived from the instantaneous values of the three phase voltages u_a , u_b and u_c as follows:

$$\mathbf{u} = \frac{2}{3} \left(u_a + u_b \cdot e^{j\frac{2\pi}{3}} + u_c \cdot e^{-j\frac{2\pi}{3}} \right) = u_\alpha + j u_\beta \quad (3)$$

Thus, the connectors for three phase vectors can be described in the following way:

```

connector InAB
  input SignalAnalog alfa;
  input SignalAnalog beta;
end InAB

```

```

connector OutAB
  output SignalAnalog alfa;
  output SignalAnalog beta;
end OutAB

```

2. Main Components of the PQLib

Any network consists of passive electrical elements like resistors, capacitors and so on. The three phase transmission line itself can be

represented as a circuit of passive electrical elements. The passive shunt harmonic filter, which is the traditional method of controlling harmonic distortion levels, consists of a tuning reactor in series with a capacitor bank.

At the same time, each network consists of active electrical elements as well. These elements are: generators, electrical motors, four quadrant electrical drives, active harmonic filters and so on. The PQLib packages imply both passive and active electrical elements. Fig. 2 shows, for example, the package of electrical elements which are based on the TwoPin interface.

2.1 Passive electrical elements

The three-phase elements like resistors, capacitors and inductors are equally defined. The three phase capacitor, for example, is defined as:

```
class C
  extends PQLib.Interfaces.TwoPin3Ph;
  parameter PQLib.Interfaces.Reactance xc[3]={1,1,1};
equation
  1/w/xc[1]*der(Vr) = Ir;
  1/w/xc[2]*der(Vs) = Is;
  1/w/xc[3]*der(Vt) = It;
end C
```

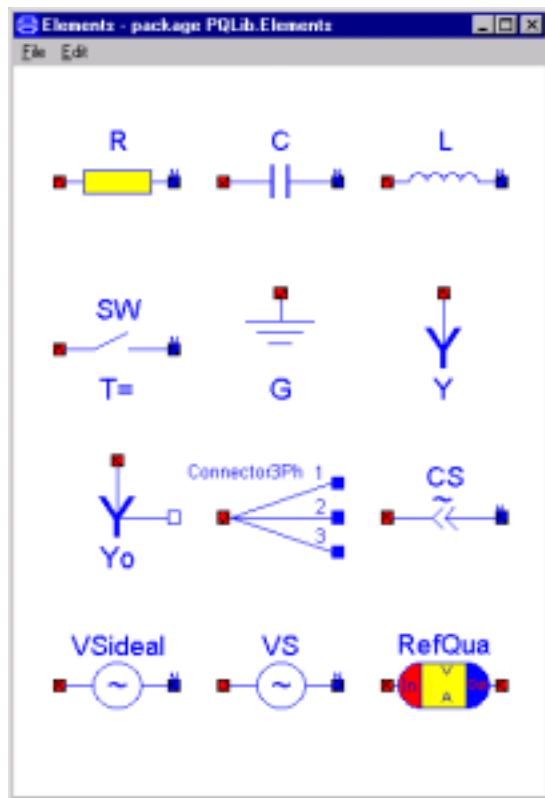


Fig. 2. Components of the package Elements

In the same way the three phase switch can be defined:

```
class SW
  extends PQLib.Interfaces.TwoPin3Ph;
  parameter Real OnTime(unit="[s]") = 0 "switch ON Time";
  parameter Real Ron(final min=0) = 1.E-5 "Closed switch resistance";
  parameter Real Goff=1.E-5 "Opened switch conductance";
protected
  Real s1;
  Real s2;
  Real s3;
equation
  Vr = s1*(if time >= OnTime then Ron else 1);
  Ir = s1*(if time >= OnTime then 1 else Goff);
  Vs = s2*(if time >= OnTime then Ron else 1);
  Is = s2*(if time >= OnTime then 1 else Goff);
  Vt = s3*(if time >= OnTime then Ron else 1);
  It = s3*(if time >= OnTime then 1 else Goff);
end SW
```

To get the star connection of the three phase elements the class Y can be used:

```
class Y
  PQLib.Interfaces.Pin3Ph v0;
equation
  v0.ia + v0.ib + v0.ic = 0;
  v0.va = v0.vb;
  v0.vb = v0.vc;
end Y
```

To use one-phase electrical elements of the Modelica standard library the class Connector3Ph (see Fig. 2 and Fig.3) is used. The class Connector3Ph is described in the following way:

```
class Connector3Ph
  PQLib.Interfaces.Pin3Ph InOut3Ph;
  Modelica.Electrical.Analog.Interfaces.Pin Ph1;
  Modelica.Electrical.Analog.Interfaces.Pin Ph2;
  Modelica.Electrical.Analog.Interfaces.Pin Ph3;
equation
  InOut3Ph.va = Ph1.v;
  InOut3Ph.vb = Ph2.v;
  InOut3Ph.vc = Ph3.v;
  InOut3Ph.ia = -Ph1.i;
  InOut3Ph.ib = -Ph2.i;
  InOut3Ph.ic = -Ph3.i;
end Connector3Ph
```

The other passive elements like the three-phase full wave converter, the three-phase transformers, the passive harmonic filters and so on are created by using graphical model editing tools. Fig. 3 shows, for example, a model of the three-phase full wave converter.

2.2 Active electrical elements

The voltage source is defined in the following way:

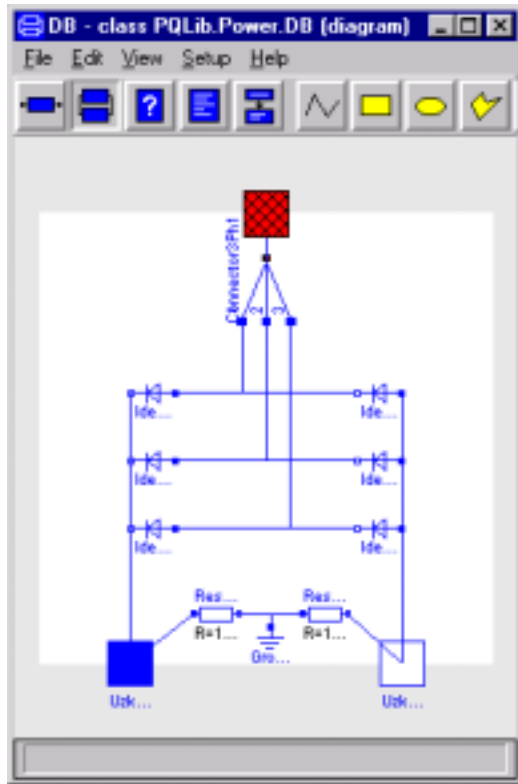


Fig.3. Diagram of the three-phase full wave converter

```

class VS
  extends PQLib.Interfaces.TwoPin3Ph;
  parameter Real N_harmonic[:]={0,0} "Array of
  numbers of harmonics";
  parameter Amplitude
  V_harmonic[size(N_harmonic, 1)]={0,0};
  parameter Phase Ph_harmonic[size(N_harmonic,
  1)]={0,0};
  parameter Amplitude V1=1.0;
  parameter Phase Ph1=0;
  parameter Amplitude V1_opposite=0.0;
  parameter Phase Ph1_opposite=0;

equation

  Vr = V1*cos(w*time + Ph1*pi/180) +
  V1_opposite*cos(w*time + Ph1_opposite*pi/
  180) + V_harmonic*cos(N_harmonic*w*time +
  Ph_harmonic*pi/180);

  Vs = V1*cos(w*time - 2*pi/3 + Ph1*pi/180) +
  V1_opposite*cos(w*time + 2*pi/3
  + Ph1_opposite*pi/180) +
  V_harmonic*cos(N_harmonic*(w*time - 2*pi/3) +
  Ph_harmonic*pi/180);

  Vt = V1*cos(w*time + 2*pi/3 + Ph1*pi/180) +
  V1_opposite*cos(w*time - 2*pi/3
  + Ph1_opposite*pi/180) +
  V_harmonic*cos(N_harmonic*(w*time + 2*pi/3) +
  Ph_harmonic*pi/180);

end VS

```

This voltage course definition makes it possible to simulate all possible kinds of the voltage distortion in industrial supply systems.

The three-phase current source is defined in the same way.

The very important part of the PQLib package are the switch mode power devices such as four quadrant frequency controlled electrical drives and active harmonic filters. These devices use the pulse-width modulation IGBT-inverter technology. Fig. 4 shows basic configuration of a IGBT-inverter.

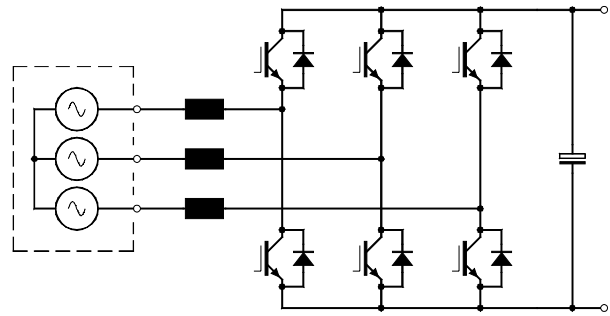


Fig. 4. Basic configuration of a IGBT-inverter

Owing to the fact that the goal of the power quality analysis is to study the network itself and in order to simplify the model of the IGBT-inverter, the following equivalent circuit for IGBT-inverters can be used [3]:

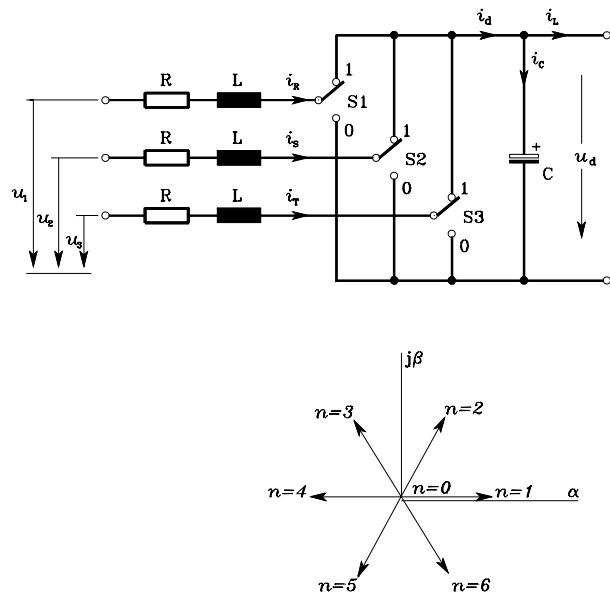


Fig. 5. Equivalent circuit for IGBT-inverters

Assuming a balanced three-phase system without the neutral connection and neglecting the resistance of the power switches, the circuit in

Fig.5 with a voltage-source inverter can be described as [3]:

$$\left. \begin{aligned} L \frac{d\mathbf{i}_F}{dt} &= \mathbf{u}_S - \mathbf{u}_W - R\mathbf{i}_F \\ C \frac{du_d}{dt} &= \frac{3}{2} \sigma(n) \operatorname{Re}(\mathbf{i}_F e^{-j\frac{\pi}{3}(n-1)}) - i_L \end{aligned} \right\} n = 0, 1, 2, \dots, 6 \quad (1)$$

with

$$\mathbf{u}_W = \frac{2}{3} \sigma(n) u_d e^{j\frac{\pi}{3}(n-1)},$$

where the used symbols denote:

- \mathbf{i}_F complex vector of the line currents;
- \mathbf{u}_S complex vector of the mains voltage;
- \mathbf{u}_W complex vector of the inverter voltage;
- u_d dc-link voltage;
- i_d dc-link current;
- i_L dc-link load current;
- L inductance of the line choke;
- R resistance of the line choke;
- C capacitance of the dc-link capacitor;
- n switching- state of the converter (Fig.5)
- σ switching function: $\sigma(n) = \begin{cases} 1, & \text{if } n > 0 \\ 0, & \text{if } n = 0 \end{cases}$.

For more information about the control unit of IGBT- inverters see [4]. The description of the control system for the active filter for industrial mains can be found, for example, in [5]. The control system of the four quadrant adjustable speed drives is described in [6].

2.3 Measuring instruments

Often the goal of the power quality analysis in networks is to get a value of the total harmonic distortion factor (THD) or values of harmonic amplitudes at the point of common coupling (PCC). In this case, it is very useful to use the built-in measuring instruments.

Fig. 6 shows the components of the package Instruments. There are instruments for voltages and currents. Together with model RMS it is possible to measure the rms-values of the three-phase voltages and currents and the THD-factor as

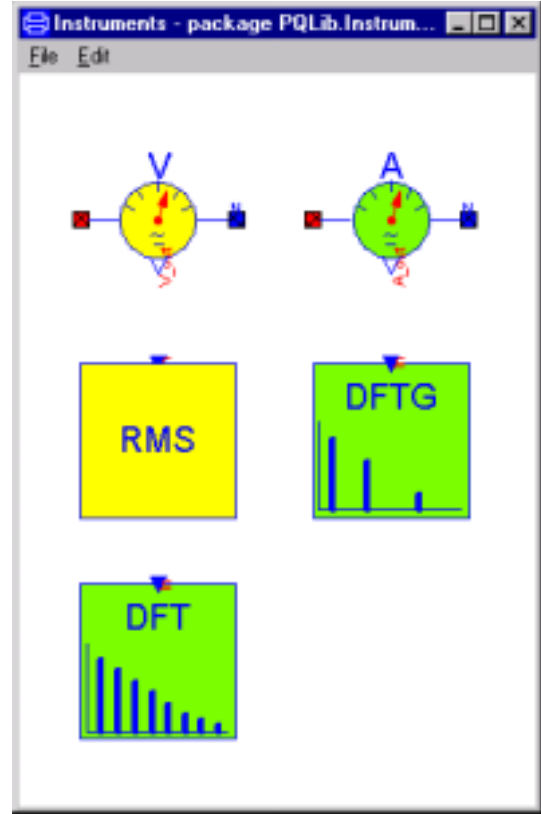


Fig.6. Components of the package Instruments

well. To get the spectra for voltages and currents the class DFT can be used. The class DFTG is very useful for the measuring of single reference number harmonics. The classes DFT and DFTG are based on DFT-technique.

Several aspects of the PQLib-package usage are demonstrated in the following example.

3. Examples

Non-linear loads such as diode or thyristor converters contribute to the degradation of the supply quality. Non-sinusoidal currents of the non-linear loads result in the distortion of the supply voltage wave form at the point of common coupling due to the finite supply impedance.

In industrial mains, the passive filters have traditionally been used to absorb harmonics generated by the load, primarily due to their low cost and high efficiency. This is a good approach when power factor correction is needed too. However, they have the following drawbacks:

- the mains impedance strongly influences the compensation characteristics of the filter;

- they result in new resonances and therefore magnify the levels of the other harmonics;

Compared with the passive filter, the active filters can be used to reduce harmonics in the industrial mains without worrying about all the problems associated with applying passive filters. Additionally they can not be overloaded by harmonics from the power system. Due to the fact that active filters use the same IGBT-inverter technology that is used in adjustable speed drives, their cost is not high.

The next Modelica model shows the utilization of the active filter to achieve harmonic cancellation for an adjustable speed drive (see Fig.7). The simulation results are shown in Fig. 8-10.

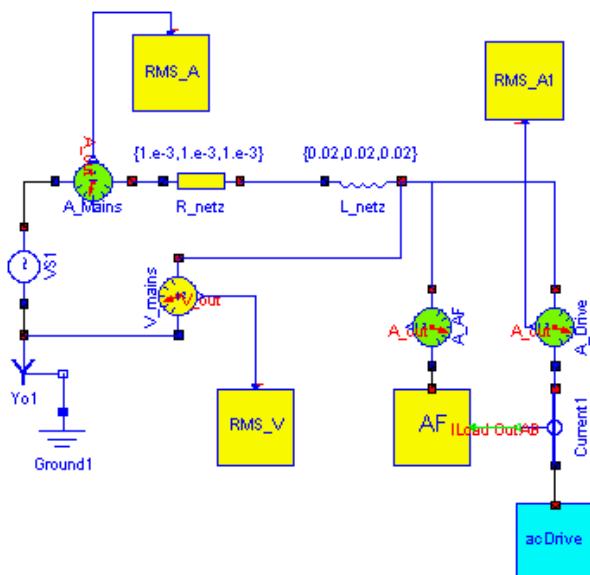


Fig. 7. Using active filter connection to achieve harmonic cancellation for adjustable speed drives

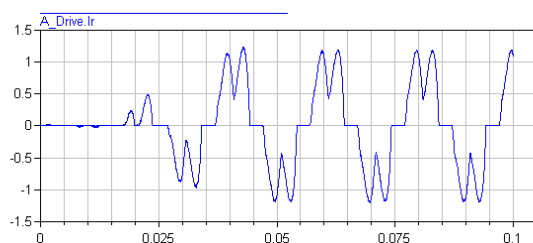


Fig. 8. Simulation results: one-phase current of the adjustable speed drive

The current of the adjustable speed drive in one phase is shown in Fig.8. Fig. 9 shows the current of the active filter in the same phase.

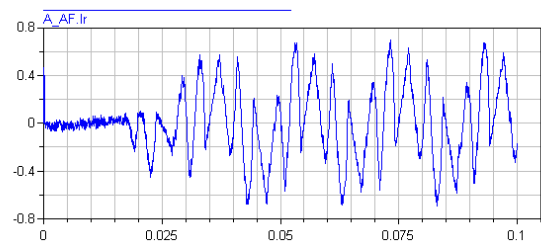


Fig. 9. Simulation results: one-phase current of the active filter

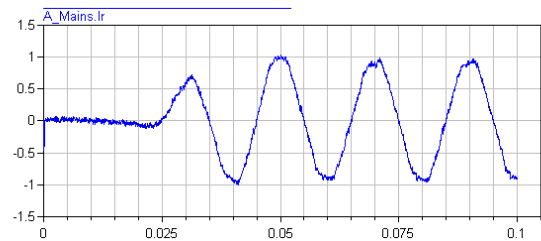


Fig. 10. Simulation results: one-phase current in the mains

The current in the mains (sum current) is presented in Fig.10. From Fig.10 it is seen that the sum current has practically sinusoidal wave form. The harmonics of the ac drive current are practically eliminated.

4. Conclusion

The presented Modelica package PQLib is very useful for power quality analysis in networks. Using the library, the user can quickly create the network with different kind of mains loads and measuring instruments using the graphical editor of the Dymola. The library has an open structure and all models can be modified.

References

- [1] Tiller, M.: Introduction to physical modeling with Modelica, Kluwer Academic Publishers, 2001
- [2] Otter, M.; Elmqvist, H.; Mattsson, S.E.: Objektorientierte Modellierung physikalischer Systeme, Teil 1-8. at Automatisierungstechnik, 1999.
- [3] Kalachnikov, S.: Regelung des netzseitigen Puls-stromrichters eines Vier-Quadranten-Spannungszwischenkreis-Umrichters, ELIN-Zeitschrift, Heft 3/4 pp. 46-55, 1994 (in German)
- [4] Mohan, N., Undeland T.M. und Robbins, W.P. Power Electronics, John Wiley & Sons, Inc. 1995, pp.805
- [5] Kalachnikov S., Berger, H.: AC-Drive with Three-Phase PWM-Rectifier as a Reactive Power Compensator, IEEE Stockholm Power Tech Conference, Stockholm, Sweden, June 18-22, 1995, Electrical Machines and Drives, pp 426-431
- [6] Kalachnikov S., Wieser, R.: AC Drives with IGBT Recovery Inverter System in the Power Range up to 500 kW, PEMC'96, Budapest, Hungary, September, 1996, pp 3.151-3.155

Session 6a

Automotive Powertrains

Development of Fuel Cell Powered Drive Trains With Modelica

Peter Treffinger, Martin Goedecke

DLR Institute of Vehicle Concepts, Pfaffenwaldring 38-40, 70569 Stuttgart
peter.treffinger@dlr.de martin.goedecke@dlr.de, Tel: 0049-711-685-7468/58

Abstract

The DLR Institute of Vehicle Concepts uses MODELICA in the area of fuel cell powered vehicles, where multidisciplinary simulation is required. The paper gives an overview of the existing libraries of DLR Institute of Vehicle Concepts and discusses our approach writing models in MODELICA.

An investigation of hybrid concepts of fuel cell powered vehicles is presented using the so called hyzem cycle as reference cycle. The results show that hybridisation of the energy supply, i.e. combining fuel cell and battery, yields to lower fuel consumption compared to vehicles only powered by fuel cells.

Introduction

The DLR Institute of Vehicle Concepts is investigating the potential and design of fuel cell powered vehicles.

One of the main issues is the simulation of the operational behavior in order to find suitable designs and operational strategies. Basis of the vehicle modeling is an appropriate block diagram, where the vehicle is separated in several sub systems. Figure 1 shows a block diagram representing a fuel cell hybrid vehicle.

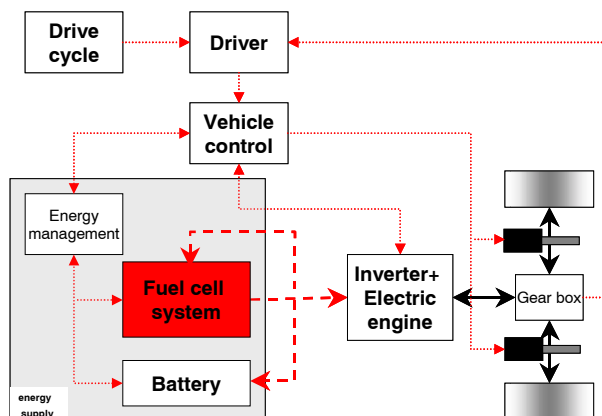


Figure 1: Block diagram of a fuel cell hybrid vehicle

It consists of the following sub systems: drive cycle, driver, vehicle control, energy management, fuel cell system, battery, inverter + electric engine, gear box and driving resistance. The thin lines represent the flow of data, the thick lines the flow of energy. The drive cycle gives the requested velocity as function of time. The driver model compares the actual velocity of the vehicle with the requested velocity and determines a request for acceleration or deceleration, which is given to the vehicle control. For safety reasons a direct mechanical or hydraulic connection between brake pedal and mechanical brake is mandatory. Therefore the vehicle control gives priority to the mechanical brakes above a certain level of deceleration. This approach should be sufficient to evaluate the potential of regenerative braking. However, our analysis does not cover the final realization of the braking system.

Finally, the vehicle control determines the signal for the inverter of the electric engine with respect to the state of the energy supply system (i.e. battery and fuel cell system). This state is determined by the energy management module, which receives the signals of the sensors installed in the battery and fuel cell system. The state of the energy supply system is also transferred to the vehicle control. For example, regenerative braking is not possible with fully charged battery. The state of the fuel cell system depends on the state of the fuel cell itself and also the state of several supply systems; e.g. air supply, fuel supply and heat and water management. Details to the fuel cell system are given below.

The short introduction should make clear that modeling of fuel cell powered vehicles is a multidisciplinary task, requiring electric, mechanical, electrochemical (fuel cell, battery), control, thermo-hydraulic models.

Libraries

Table 1 gives an overview of the existing libraries.

Library	Content
Property	Thermal and caloric properties of fluids (air, water, hydrogen, glycol-water, ..)
Piping	Ducts, fans, compressors, blowers, valves
Heatex	Heat exchanger
Accu	Battery models
Carmechnic	Mechanical components of cars
Control	Control units
Power_electronics	Inverters, DCDC-converters
Fuel cell	Fuel cells
Tank	Models of fuel tanks compressed gas, metal hydride

Table 1: Listing of libraries

Additionally, standard MODELICA libraries as electric library and block library were used. As simulation results were needed for ongoing projects, the libraries were designed under a very limited time scale. We felt that the hierarchical layout of the libraries should not be too complex. We therefore decided to limit the usage of base models creating a usable model to only one base model. Our general approach is outlined by two examples of the property and piping library.

The following listing shows a property model for wet air. Wet air describes a mixture of ideal gases (O_2 and N_2) with one condensable component (H_2O). As we are working at moderate pressures and temperatures, the ideal gas assumption lead to sufficiently accurate results. General equations for such a mixture have been put into the base class `base_prop_01` (extend statement) setting the modifier `n` to three, which means the mixture is composed of three components: `water[1]`, `nitrogen[2]` and `oxygen[3]`. The specific properties of the components are given by functions.

```

bzvt.property.air01.wet_air

see bzvt.property.bass_classes.base_prop_01

Modelica definition

model wet_air "siehe bzvt.property.bass_classes.base_prop_01"
  extends bzvt.property.bass_classes.base_prop_01(n=3);
equation
  //-----properties-----
  ps = bzvt.property.h2o.satproperties.ps03(t);
  rhom_l_tp*mm_i[1] = bzvt.property.h2o.propfunctions.rho_l_ofp(t);

  mm_i[1] = bzvt.property.h2o.constants.mm;
  mm_i[2] = bzvt.property.n2.constants.mm;
  mm_i[3] = bzvt.property.o2.constants.mm;

  hm_g_i[1]=xm_g_tp*bzvt.property.h2o.janaf.h_g(t+ 273.15)*mm_i[1];
  hm_g_i[2] = xm_i[2]*bzvt.property.n2.janaf.h(t + 273.15)*mm_i[2];
  hm_g_i[3] = xm_i[3]*bzvt.property.o2.janaf.h(t + 273.15)*mm_i[3];
  hm_l_tp=xm_l_tp*
    bzvt.property.h2o.janaf.h_l_ofp(t+273.15,p)*mm_i[1];
  sm_g_i[1]=xm_g_tp*
    bzvt.property.h2o.janaf.s_g(t+273.15,p)*mm_i[1];
  sm_g_i[2] = xm_i[2]*bzvt.property.n2.janaf.s(t + 273.15, p)*mm_i[2];
  sm_g_i[3] = xm_i[3]*bzvt.property.o2.janaf.s(t + 273.15, p)*mm_i[3];

  sm_l_tp = xm_l_tp*bzvt.property.h2o.janaf.s_l(t + 273.15)*mm_i[1];
  //-----
end wet_air;

```

We use the same base model for other mixtures of gases with one condensable component. The thermodynamic state of such a mixture is determined by two state variables and $n-1$ compositions. After a period of refinement, we achieved a very good robustness even with several sets of input variables:

- Composition, p, T
- Composition, p, h .
- Composition, p, s
- Relative humidity, p, T

Key points to get this good performance have been the appropriate formulation of the general equations, i.e. structure of the equations. Molar or mass specific state variable can be used, which underlines the general formulation of this routine.

Figure 2 shows the icon and object diagram of a compressor. The fluid connectors inlet and outlet contain the potential variables pressure p , enthalpy h , composition x_i and the flow variable mass flow rate \dot{m} . The compressor is described by an efficiency model with the isentropic efficiency as a parameter. The calculation scheme is indicated in the object diagram by means of an enthalpy, entropy diagram. Three similar property models are embedded into the model: prop1, prop2, isentrop. Prop1 and prop2 are used to calculate the inlet and outlet state. Isentrop calculates the state for an isentropic compression.

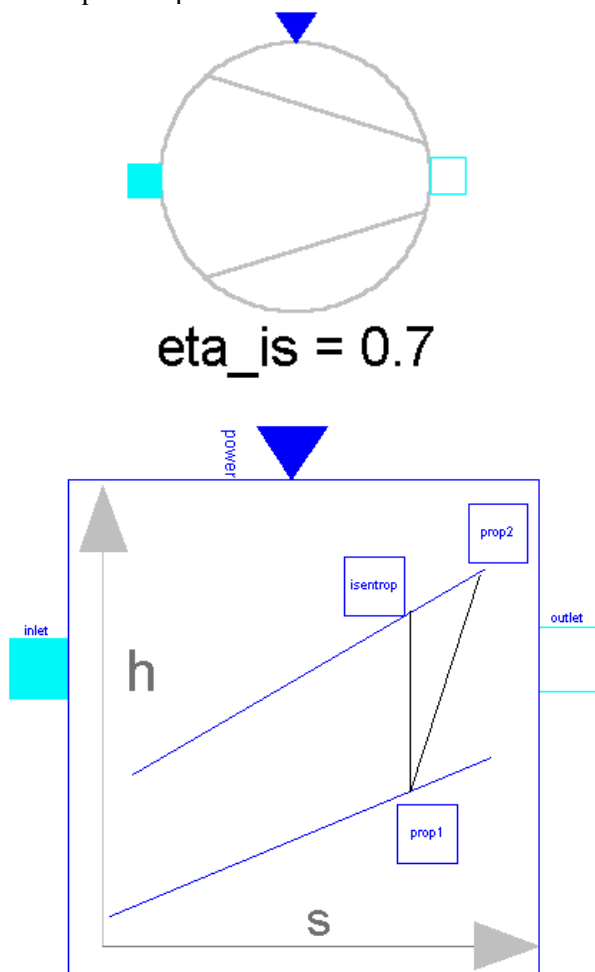


Figure 2: Above: icon diagram of compressor; below: object diagram of compressor.

Formulating our models we have taken special care to handle cases like mass flow rates getting very small values down to zero and to get a robust and stable formulation. Therefore we tried several formulations of heat exchangers: description by piecewise discretisation, log mean temperature and efficiencies.

In the beginning we had problems with robustness of our models, e.g. the convergence of the simulation run depended on the parameter values used. Also the appropriate initialisation of the model was difficult. To overcome the initialisation problems we have written special initialisation routines, which allow us to formulate the equations with numerically appropriate state variables and use common variables for parameterisation. For example inside a volume the internal energy u is used as state variable. However, by means of the initialisation function temperature t is used to parameterise the model.

As mentioned above we have restricted ourselves primarily to alternative drive trains and fuel cell systems due to time limitations. To simplify the usage of models for people working in DLR Stuttgart e.g. on solar thermal engineering our colleague Wolf-Dieter Steinmann has generalized a property and a thermohydraulic library [2].

In the following we will give some examples how we are using the libraries.

Modeling of Fuel Cell cars

Figure 3 shows the object diagram of a fuel cell powered car composed from models of the libraries listed in table 1. The object diagram represents the layout of the scheme given in figure 1, whereby the object dcsupply corresponds to the gray lighted (energy supply) in figure 1.

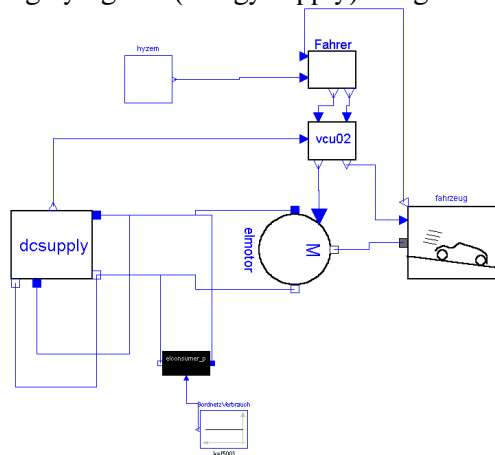


Figure 3: Vehicle model as MODELICA object diagram.

The object diagram consists of four main parts:

1. **the overall control of the vehicle**, which includes the representation of the drive cycle (object hyzem), which has specially been derived for hybrid vehicles [1], the driver model (object Fahrer) and the control unit (object vcu02).
2. **the supply of electrical energy** (object dcsupply), which includes the fuel cell system.
3. **the inverter and electrical engine** (object elmotor), which convert the DC current to mechanical energy.
4. **the mechanical parts and driving resistance of the car** (object fahrzeug)

Our special interest is the energy supply, which provides energy for elmotor, auxiliaries and for to energy supply system itself (indicated by the two additional electrical connectors of the object dcsupply).

Figure 4 shows the object diagram of the object dcsupply in detail.

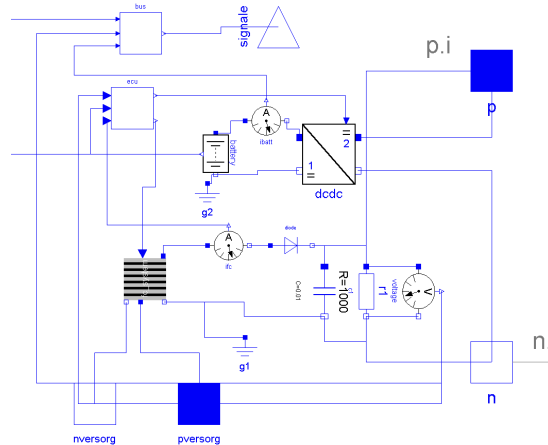


Figure 4: Object diagram of energy supply

Fuel cell system and battery are combined to a hybrid energy supply system. A bi-directional DCDC-converter adjusts the battery voltage to the fuel cell voltage. The DCDC-converter is controlled by the energy control unit (object ecu), which gets input from sensors installed in the battery and fuel cell system.

DC power is supplied to the electrical engine by the two electrical connectors p and n (right). Compressors and pumps of the fuel cell system are supplied with electrical energy by the two additional electrical connectors (nversorg and pversorg).

The dcsupply object can be parameterized in order to generate three different hybrid fuel cell vehicle designs.

1. **Fuel Cell Vehicle:** vehicle with a fuel cell solely to supply energy
2. **Fuel Cell Vehicle + Booster** battery: a vehicle with high power fuel cell and relatively low power battery to recover brake energy and provide additional peak power
3. **Fuel cell as battery loader:** high power battery and low power battery. The fuel cell is mainly used as a battery loader.

For the following examples we used a reference vehicle with the following parameters: vehicle mass 1240 to 1280 kg depending on option, drag resistance times front area 0.6m^2 and rolling resistance 0.01. The total installed power in all designs is 60 kW.

first design “**Fuel Cell Vehicle**” the fuel cell power is 60 kW.

second design “**Fuel Cell Vehicle + Booster**” the fuel cell power is 40 kW

third design “**Fuel cell as battery loader**” the fuel cell power is 20 kW

Figure 5 compares the requested velocity of the hyzem cycle and the achieved velocity of the fuel cell car for the third design “**Fuel cell as battery loader**”. The first 500 s of the hyzem cycle represent the urban part of the drive cycle, then up to approximately 1500 s the extra urban part follows. The last part is the motorway part.

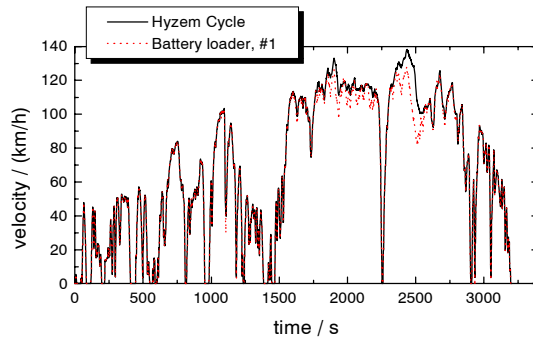


Figure 5: Results of drive cycle simulation; velocity as function of time

The battery loader can not follow the drive cycle at higher velocities in contrast to the other designs. Due to the relatively weaker voltage curve compared to the other designs, which yields to lower maximum torques of the electric engine.

Figure 6 compares the state of the battery for the booster design and battery loader design. Over the urban (first 500 s) and the extra urban part (to 1500 s) of the drive cycle the degree of discharge decreases, which means the battery is loaded. During the motorway part of the drive cycle, the battery loader design needs a significant amount of energy from the battery, whereas the state of the battery for the booster design keeps almost stable.

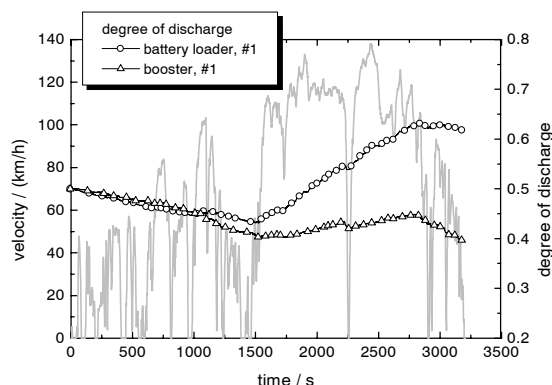


Figure 6: Results of vehicle simulation; degree of discharge as function of time.

Table 2 summarizes the results of the investigation. It shows the energy consumption per km obtained for the simulation of the three vehicle designs. The energy content of hydrogen is approximately 120 MJ/kg. Thus the vehicles would consume around 1 kg hydrogen per 100 km.

The operation strategy reflects the control strategy of the fuel cell system. Operation strategy #1 minimizes the auxiliary energy consumed to operate the fuel cell system. The results show that from the point of energy efficiency the second design “**Fuel Cell Vehicle + Booster**” battery combined with operation strategy #1 is the most energy efficient. The results clearly show that the energy consumption of the vehicle without battery suffers from the lack of energy recovery.

Degree of hybridisation	Operation Strategy	Cons. Energy (MJ/km)
Fuel Cell Vehicle	#1	1.57
	#2	1.65
	#3	1.75
Fuel Cell Vehicle + Booster Battery	#1	1.09
	#2	1.15
	#3	1.21
Fuel Cell as a battery loader	#1	1.17
	#2	1.20
	#3	1.23

Table 2: Energy efficiency for Hybrid fuel cell vehicles

Modeling of Fuel Cell Systems

Figure 7 shows the object diagram of a fuel cell system where two stacks are electrically connected in series. The stacks have to be cooled and supplied with fuel and oxygen. In this example hydrogen is used as the fuel. The complete hydrogen supply containing tank and pressure reducer etc. is hidden in the object H2 (left in figure 7).

Oxygen is taken from air. As hydrogen H2 the components of air supply (compressor, pressure control, mass flow control etc.) are hidden in the air object (right in figure 7).

The fuel cell stacks (each with 120 cells) are

directly cooled with water. Each stack has separate inlets for air and water. Inside the stacks air and water are combined and dragged out at one single outlet. Therefore a separator is needed to separate air and water.

The water(dark blue) and glycol(green) modules includes pumps, heat exchangers, ducts and control valves. The glycol module additionally contains a fan and an air – glycol heat exchanger to cool the glycol.

An electric load, which is connected to the anode and cathode of the fuel cells, is used to examine the fuel cell system with different load profiles.

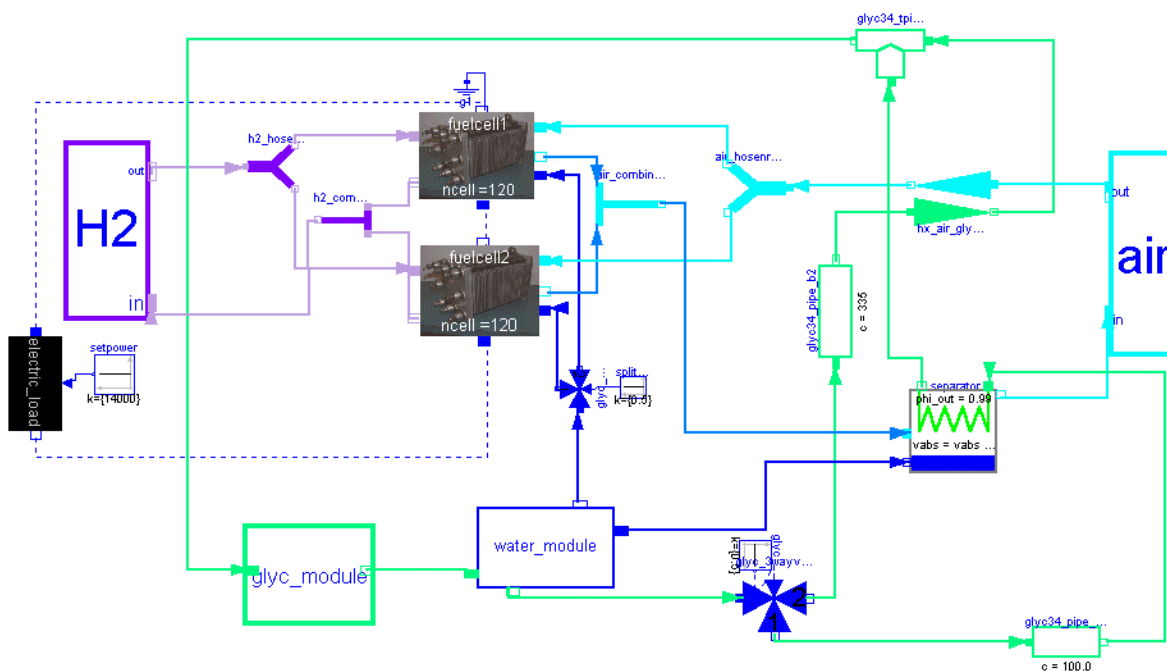


Figure 7: Object diagram of a fuel cell system

The description of the fuel cell systems shows that,

- thermo-hydraulic components are important
- the system has several closed loops
- advanced property routines are needed e.g. covering phase change

One main problem of fuel cell systems is that water is dragged with the exhaust stream. Despite of the production of water in the fuel cell a negative water balance could occur. In such a situation an additional water tank would be needed. The water balance is influenced by a number of parameters which influence each other, e.g. air supply strategy, cooling strategy, pressure drops, load profile. Beyond the task of layout and design of the components of fuel cell system, we use our simulation models to adjust the system in order to achieve water neutrality.

Summary

The DLR Institute of Vehicle Concepts uses MODELICA in projects, where components of fuel cell powered cars are developed. During last year a number of libraries for several disciplines have been created. We find MODELICA a very promising tool to analyse the complex interaction in such systems. Our models have brought us a lot of insight on how the design of fuel cell powered cars should be. They are especially useful for the design of the components of the fuel cell system and for the definition of operation strategies. The work with MODELICA will be continued in the future.

Literature

- [1] Gossen, F., Grahl, M.: Vergleich von Brennstoffzellen- und weiteren zukünftigen Antrieben hinsichtlich Wirkungsgrad und Wirtschaftlichkeit. 8. Aachener Kolloquium Fahrzeug- und Motorentechnik, Aachen, October 1999
- [2] Steinmann, W-D.: 2nd MODELICA Conference, Munich 2002

Spark-Ignited-Engine Cycle Simulation in Modelica

Charles Newman, John Batteh, and Michael Tiller

Ford Motor Company, USA

Abstract

This paper details the use of the Modelica modeling language for the cycle simulation of a spark-ignited engine. After a brief overview of the physical processes which must be modeled by a predictive cycle simulation model, this work emphasizes the two main challenges to the developer of such a model in Modelica: zone formation/destruction and calculation of realistic thermodynamic properties of the cylinder contents. The results illustrate that Modelica is capable of handling the complex physical models required by cycle simulation programs.

1 Introduction

Computer programs, which simulate the thermodynamic cycle of an internal combustion engine, have been developed over the last several decades both to assist in understanding the observed behavior of engines and to predict engine performance and efficiency as functions of engine design parameters (see [1], [2]). At Ford Motor Company the internally-developed General Engine Simulation (GESIM) program [3-6] has matured sufficiently that it can accurately predict the effects of intake and exhaust port design, combustion chamber geometry, and valve timing on combustion rate, fuel economy, and emissions for a spark-ignited engine.

Although very useful, GESIM has some significant limitations: it simulates only one cylinder of an engine running at constant angular velocity and reports the cycle averaged output torque. The result is essentially a simulation of a dynamometer test point for the engine. Currently, GESIM is written in procedural languages (FORTRAN and C), and its capabilities cannot readily be extended to include the transient multi-cylinder behavior required to simulate real engine operation in a vehicle. An effort is now under way to capture GESIM's physical models in Modelica [7, 8],

thereby retaining its current capabilities, while removing the limitations on its applicability. Previous work [9, 10] proved the feasibility of this type of detailed powertrain modeling in Modelica. This paper, after a brief overview of the physical processes which must be modeled by a predictive cycle simulation model, focuses on the two main challenges to the developer of such a model in Modelica: zone formation/destruction and calculation of realistic thermodynamic properties of the cylinder contents.

2 Overview of Cycle Simulation Physics

The goal of a cycle-simulation program is to perform a thermodynamic analysis of the engine cylinder contents through each engine cycle, an overview of which is shown in Figure 1:

- a) The mixture is prepared during the gas exchange period which extends from the time the exhaust valve opens (EVO) until the intake valve closes (IVC); during this period the burned gases are expelled, and a fresh mixture of fuel and air is inducted into the chamber. The mixture is then compressed until the piston reaches a position near top dead center (TDC).
- b) In a spark-ignited engine, combustion is then initiated by the firing of the spark plug.
- c) The mixture is then burned, raising the in-cylinder pressure and temperature considerably. In contrast to the process in a diesel engine, where combustion occurs throughout the chamber simultaneously, the mixture is consumed through the propagation of a well-defined flame front across the combustion chamber.
- d) After the flame consumes all the combustible mixture, usually some time after TDC, the gas continues to expand, transferring energy to the piston as it continues its downward trajectory.

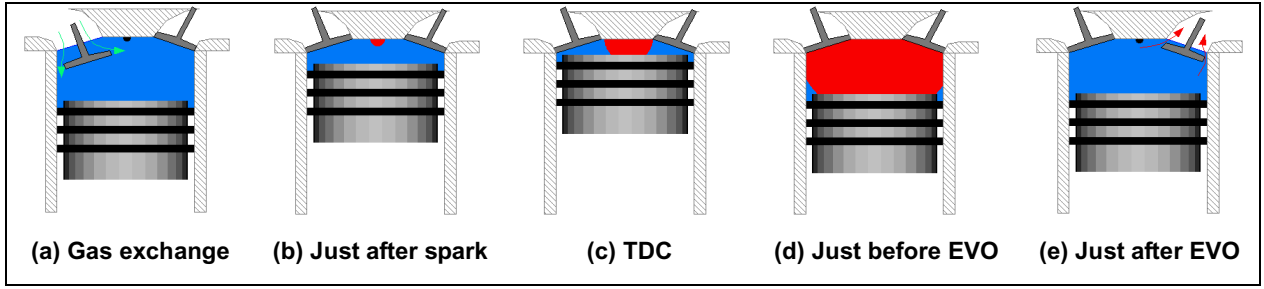


Figure 1. Modeling of cylinder contents at various points in engine cycle

- e) When EVO is again reached, the process begins anew.

Successful modeling of the cycle requires capturing the essential physics of all the processes described above. In this paper, however, we concentrate on those processes critical to the thermodynamic analysis: combustion by a propagating flame and the computation of realistic thermodynamic properties of the gases comprising the in-cylinder mixture.

3 Combustion Modeling

The traditional approach taken to model the spark-ignition combustion process is suggested by Figure 1: we divide the cylinder contents into two (or more) thermodynamic zones, each with its own temperature and composition. Behind the flame is a zone comprised of only burned gases at high temperature (the burned zone). Ahead of the flame is the unburned zone, containing the remnants of the original mixture at a much lower temperature.

Each zone is regarded as a homogeneous mixture of N species (or pseudo-species), each modeled as an ideal gas. The zone must then satisfy the First Law of Thermodynamics,

$$\frac{dU}{dt} = \dot{Q} - P \frac{dV}{dt}, \quad (1)$$

the ideal gas law,

$$PV = MRT, \quad (2)$$

and the conservation of mass for each species

$$\frac{dm_i}{dt} = \dot{S}_i \quad (3)$$

where

$$\dot{Q} = \dot{H} + \dot{q}_w \quad (4)$$

$$M = \sum_i m_i \quad (5)$$

$$m_i = MX_i \quad (6)$$

$$U = Mu \quad (7)$$

$$R = \sum_i X_i R_i = \bar{R} \sum_i \frac{X_i}{\mu_i} \quad (8)$$

and

U	is the total internal energy of the zone
\dot{Q}	is the total energy flow into the zone
P	is the cylinder pressure
V	is the volume of the zone
\dot{H}	is the total flow of enthalpy entering the zone
\dot{q}_w	is the heat transferred from the chamber walls to the zone
M	is the total mass in the zone
R	is the overall (mass-specific) gas constant for the zone
T	is the temperature of the zone
m_i	is the mass of species i in the zone
X_i	is the mass fraction of species i in the zone
\dot{S}_i	is the total flow of species i of the mass flow entering the zone
u	is the specific internal energy of the zone
$u_i(P, T)$	is the specific internal energy of species i and is a known function of P and T
\bar{R}	is the universal gas constant
$\mu_i(P, T)$	is the average molecular weight of species i and is a known function of P and T

In addition to a set of equations (1)-(8) for each zone z , a constraint on the total volume must be added:

$$V_T = \sum_z V_z \quad (9)$$

where

V_T is the total volume of the chamber

The task of the modeler is to complete the system of equations by supplying conditions to specify the flows appearing in equations (1)-(8) for all zones.

3.1 GESIM Implementation

GESIM makes two modifications to the above approach:

1. During mixture preparation (gas exchange and compression), the contents are treated as a single zone. No solution is sought for the quantities in the burned zone.
2. After combustion is initiated and as soon as the flame front makes contact with the surface of the chamber, the burned zone is further subdivided into an adiabatic core and a thermal boundary layer between the core and the wall.

GESIM implements the engine cycle of Figure 1 as follows:

- a) The system to be solved consists of equations (1)-(9) for a single (unburned) zone, supplemented by the following relationships for the flows:

$$\dot{S}_{ui} = \sum_v \dot{m}_v \tilde{X}_{vi} \quad (10)$$

$$\dot{H}_u = \sum_v \dot{m}_v \tilde{h}_v \quad (11)$$

where

\dot{m}_v is the mass flow into the zone through valve v

\tilde{h}_v is the specific enthalpy of the mass flow entering the zone through valve v

\tilde{X}_{vi} is the fraction of species i of the mass flow entering the zone through valve v

- b) At spark, the solution is interrupted and a kernel (diameter ~ 1 mm) of burned gases is instantly created from the unburned mixture. This forms the initial state for the adiabatic zone. Equations (1)-(8) for the adiabatic zone are added to the system, and flows for both zones are now specified by

$$\dot{S}_{ui} = -\dot{m}_b X_{ui} \quad (12)$$

$$\dot{H}_u = -\dot{m}_b h_u \quad (13)$$

for the unburned zone and

$$\dot{S}_{bi} = \dot{m}_b B_i(\{X_u\}) \quad (14)$$

$$\dot{H}_b = \dot{m}_b h_u \quad (15)$$

for the burned zone where

\dot{m}_b is the burn rate as calculated by the flame propagation model

$B_i(\{X_u\})$ is the fraction of species remaining after a mixture of composition X_{ui} is burned.

h_u is the specific enthalpy of the unburned zone

- c) When the flame contacts the wall, the solution is interrupted and a thin boundary layer is initialized and the system of equations altered in a manner similar to b) above. The details are omitted here.
- d) At EVO, the solution is again interrupted. All zones are mixed together instantly to form a single zone, which represents the initial state for the unburned zone for the next cycle.
- e) The set of equations is reduced to those of a) above and the solution is resumed.

3.2 Modelica Implementation

As GESIM is an in-house product written in FORTRAN, it has complete control over the solution method- it can interrupt the solution at will to expand or shrink the system of equations, reinitialize, and resume. In Modelica, however, where the number of equations is fixed, we adopt two modifications to GESIM's approach:

1. The burned zone (*i.e.* the adiabatic zone and boundary layer) exists throughout the simulation. Each zone satisfies equations (1)-(8), and equation (9) is imposed on the volumes. During the mixture preparation period, when the burned zone does not exist in GESIM, we require that it have a small mass (less than the initial spark kernel) and have temperature and composition equal to that of the unburned zone. The solution for the two zones degenerates to an equivalent single-zone simulation during this portion of the cycle.
2. GESIM effects the transitions between 1-zone and multi-zone behavior essentially by simulating impulses. In the absence of a stable and mature impulse capability in Modelica, we choose to perform these transitions over a finite, but short time.

Since all zones are always mathematically active, enough conditions must be supplied to specify all the flows. While our model includes all three zones, in this paper we discuss only the aspects of modeling two zones in order to illustrate the approach. Adding the boundary layer is a relatively straightforward extension of the technique.

During mixture preparation, the "burned" zone is just a dummy placeholder, containing a small

amount of mass in its unburned state, which will be the first mass to be burned in forming the initial kernel after spark. We require that both zones have identical temperature and composition. Since (5) and (6) imply that $\sum_i X_i = 1$, only $N-1$ components of the composition vector can be independently constrained. Hence, our condition can be expressed as

$$\Delta T = T_u - T_b = 0 \quad (16)$$

$$\Delta X_i = X_{ui} - X_{bi} = 0, \quad 1 \leq i \leq N-1 \quad (17)$$

$$\Delta V = V_b - V_K = 0 \quad (18)$$

where

V_K is a volume small compared to that of the initial spark kernel

Denoting the time of EVO as t_0 , we achieve the transition from combustion to the above conditions by mixing the contents of the two zones over a short time $\tau_s \sim 100\mu s$; *i.e.*, for $t_0 \leq t \leq t_F = t_0 + \tau_s$:

$$\Delta T = \Delta T_0 \sigma(t) \quad (19)$$

$$\Delta X_i = \Delta X_{i0} \sigma(t), \quad 1 \leq i \leq N-1 \quad (20)$$

$$\Delta V = \Delta V_0 \sigma(t) \quad (21)$$

where the subscript 0 denotes the value of a quantity at EVO and

$$\sigma(t) = \left(\frac{t - t_F}{\tau_s} \right)^2. \quad (22)$$

Referring again to Figure 1, the Modelica implementation of the cycle is as follows:

- a) During mixture preparation, a dummy burned zone (shown in black) exists. Except for the transition time at EVO, its volume is V_K . Otherwise it is indistinguishable from the main unburned zone. To specify the flows, we first introduce modified forms of (10)-(11):

$$\dot{S}_{ui} + \dot{S}_{bi} = \sum_v \dot{m}_v \tilde{X}_{vi} \quad (23)$$

$$\dot{H}_u + \dot{H}_b = \sum_v \dot{m}_v \tilde{h}_v \quad (24)$$

The constraints (16)-(18) or (19)-(21) are sufficient to complete the specification for the transition at EVO or the main portion of mixture preparation, respectively. Differentiating both of the above sets of

equations, we can combine them into a single set. Between EVO and spark, then

$$\frac{d\Delta T}{dt} = \Delta T_0 \lambda(t) \quad (25)$$

$$\frac{d\Delta X_i}{dt} = \Delta X_{i0} \lambda(t), \quad 1 \leq i \leq N-1 \quad (26)$$

$$\frac{d\Delta V}{dt} = \Delta V_0 \lambda(t) \quad (27)$$

where

$$\lambda(t) = \frac{2}{\tau_s} \min(0, \frac{t - t_F}{\tau_s}). \quad (28)$$

- b) The transition at spark from mixture preparation to combustion is accomplished in two steps.

1. Denoting the time of spark as t_I , we first burn the contents of the "burned" zone over a time $\tau_1 \sim 1\mu s$; *i.e.*, for $t_I \leq t \leq t_M = t_I + \tau_1$, the flows are specified by

$$\dot{S}_{ui} = 0 \quad (29)$$

$$\dot{H}_u = 0_u \quad (30)$$

$$\dot{S}_{bi} = \frac{M_{bl}}{\tau_1} [B_i(\{X_I\}) - X_{bi}] \quad (31)$$

$$\dot{H}_b = 0 \quad (32)$$

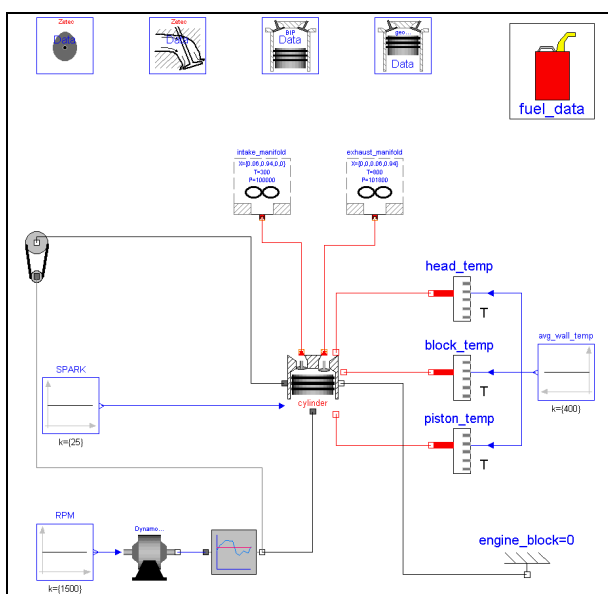
where the subscript I denotes a value at ignition.

2. At $t = t_M$, two-zone combustion begins. Equations (12)-(15) are used unchanged, just as in GESIM, with the burn rate \dot{m}_b initially set high enough to assure that, by $t = t_M + 1\mu s$, a burned zone volume will be achieved equal to that of GESIM's initial spark kernel. As soon as the required volume is attained, the flame propagation model is used to calculate the burn rate.
- c) The main phase of combustion is identical to that of GESIM.
- d) At EVO, expansion ends. The current time is assigned to t_0 , and we change over to the mixture preparation phase.
- e) The next cycle begins.

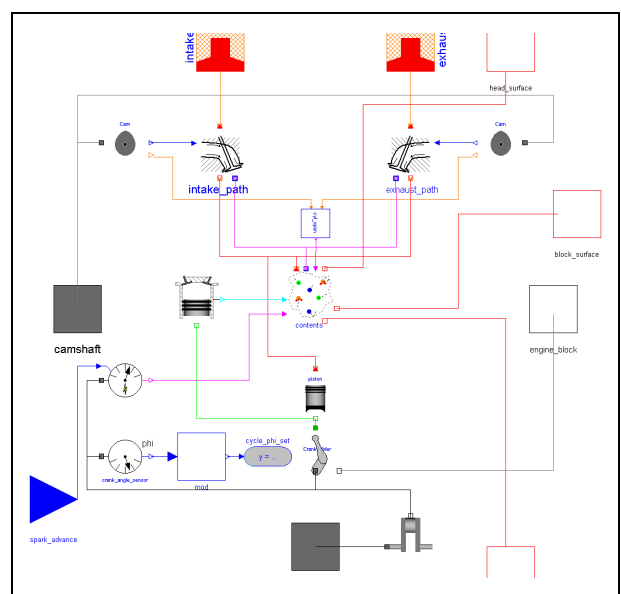
Figure 2 shows schematics for successive levels of the instance hierarchy in the Modelica implementation of a single-cylinder version of this model. The engine itself is shown in Figure 2(a). Its cylinder component appears in Figure 2(b); it has been designed to facilitate construction of multi-cylinder configurations through its replication. The contents component of the cylinder, shown in Figure 2(c), models the thermodynamics of all gases residing in the cylinder. Finally, in Figure 2(d) we see the combustion component, the main focus here.

The combustion component controls the creation, evolution, and destruction of the burned

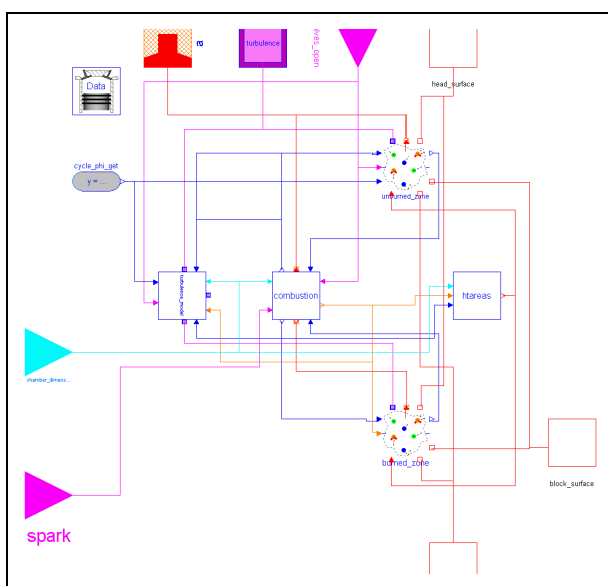
zone in the manner discussed above by coordinating the activities of a parallel configuration of three subcomponents: `mix_zones` to mix burned and unburned together at EVO by specifying the flows according to a) above; `kernel_burn` to create the initial spark kernel as described in b) above; `flameadv` to grow the burned zone according b) and c). Each of these components supplies non-vanishing contributions to the total flow only during the portion of the cycle that it is meant to control. Figures 6-9, included at the end of the paper, contain code fragments that provide some insights into how the models described in this section have been implemented.



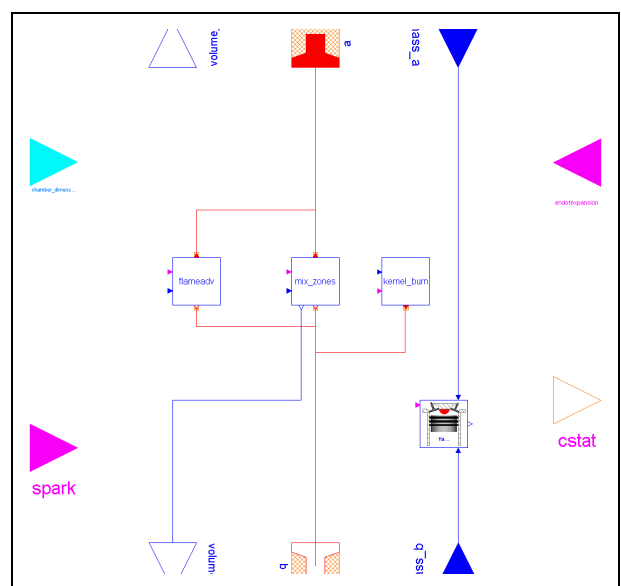
(a) single cylinder engine



(b) one cylinder



(c) cylinder contents



(d) combustion model

Figure 2. Schematic representations of the engine model hierarchy

4 Thermodynamic Properties

4.1 The "MediumModel" Idiom

Different engine simulation applications require different levels of detail. One of the important determinations to be made is what level of detail is required in computing the thermodynamic properties of the cylinder contents. In some cases, we can treat the medium flowing through the engine as simply air but in other cases we might need to allow for changes in composition of the gas that would require tracking several chemical species.

At first glance, it would appear that different component models (*e.g.* valves, control volumes) would be required for each of the possible media. But if we look carefully at the issue, we find that the properties of the selected medium are orthogonal to the equations of the various thermodynamic processes. In other words, if the models are formulated correctly, the choice of media can be made independently of the components used to model the engine cycle.

In practice, this is achieved by using what we refer to as the MediumModel idiom. The basic idea behind this idiom is to define a partial package that describes the interfaces of the various models, connectors, *etc.* that will be required to implement all of our component models. However, no implementation is provided by this partial package. This is essentially a Modelica adaptation of the "Kit" or "Abstract Factory" pattern found in [13]. In the same way that a "Kit" might be used as a means of instantiating compatible GUI toolkit components such as scrollbars, menus, *etc.*, the MediumModel is used to instantiate consistent sets of property models, connectors, *etc.*

The complete definition of the MediumModel package definition is too lengthy to include here, but it consists mainly of three things. First, it contains a partial model definition that defines the interface for computing medium properties. Second, it contains partial connector definitions that include the appropriate number of chemical species flowing between components. Finally, it contains several partial function definitions for computing useful quantities (*e.g.* air fuel ratio) using medium composition information.

4.2 Property Calculations

As discussed previously, the conservation of energy for the various combustion zones in the cylinder is at the heart of the cycle simulation tool. In addition, a specific medium model is needed to determine the thermodynamic properties (*e.g.* specific enthalpy, h , and specific internal energy, u) of the cylinder contents used in Eqs. (4), (7).

In simple combustion simulations the cylinder contents can be treated as a single ideal gas. Constructing a medium model for a single ideal gas is relatively easy. Since the thermodynamic properties vary as a function of temperature only, they can be calculated from a look-up table or a polynomial regression of tabulated data. However, for detailed combustion systems the medium is assumed to be a reacting **mixture** of ideal gases (*e.g.* the fuel vapor, air, and combustion products). Therefore, in order to compute the contribution of each species to the mixture property we must first determine the relative amounts of the various species in the mixture. This calculation requires the solution of the nonlinear system of equations that define chemical equilibrium for the mixture.

The steps required for the property calculation are detailed in [11] and can be summarized as follows:

1. Solve the nonlinear system of equations that defines chemical equilibrium for the combustion mixture to yield the mixture composition
2. Calculate the contribution of each species to the mixture property
3. Calculate the mixture property from the individual species contributions and the mixture composition

While the calculations in steps 2 and 3 above are simple evaluations, the nonlinear solution of the chemical equilibrium problem is certainly nontrivial. In the GESIM property models the combustion products are comprised of twenty-one species; thus, obtaining the mixture composition requires the solution of a set of twenty-two nonlinear equations for chemical equilibrium. Furthermore, recall that h and u are functions of P , T , and ϕ (the equivalence ratio of the combustion products). Therefore, the property calculations, including the determination of the equilibrium composition, must be computed for each thermodynamic zone in the engine model whenever there is a change in the pressure,

temperature, and/or composition of any of the thermodynamic zones.

While this method for calculating the mixture properties could be implemented directly in Modelica, it would require the cycle simulation tool to repeatedly compute the solution of the chemical equilibrium problem, a formulation that has several drawbacks. First, the repeated solution of the nonlinear equations used to determine the equilibrium chemistry is computationally demanding when compared to the other behavior equations involved, a formulation which would result in slower simulation times. In addition, the chemical systems introduce other issues such as robustness of the nonlinear solution method and scaling of the chemical concentrations. So, rather than calculate the needed properties on-demand during the simulations, an alternative approach is to pre-compute the properties throughout the expected domain of operation and simply interpolate as needed during the simulations. The ModelicaAdditions package contains a Tables package that includes models for linear interpolation in one and two dimensions, CombiTable1D and CombiTable2D respectively. However, the mixture properties are functions of three variables (P , T , and ϕ). Even trilinear interpolation would not suffice as continuity of the mixture properties and gradients could not be insured. It can be shown [11] that this continuity in gradients is important for fast and accurate simulation. Furthermore, maintaining continuous gradients allows for index reduction in Modelica and would certainly be of benefit to the numerical integration schemes in Dymola [12].

As a result, higher order interpolation schemes are required to provide the desired continuity. They involve the construction and evaluation of polynomials to yield the interpolated values and are more difficult to implement since more information is needed about the function other than simply its value at each grid point (*i.e.* the derivatives of the function, additional function values at adjoining cubes, *etc.*).

Though not detailed in this work, a flexible modular scheme has been developed to automatically formulate the chemical equilibrium problem and solve for the equilibrium chemistry and mixture properties over a wide range of engine operating conditions (P , T , and ϕ). The remainder of this section discusses the implementation of a higher order interpolation scheme in Modelica with

the assumption that a file has been created that contains all the necessary data to perform the interpolation.

4.3 Hermite Interpolation

Based on the requirements detailed in the previous section, the interpolation scheme must be three-dimensional and provide continuity of the interpolated function value and its derivative. One scheme that satisfies those criteria is Hermite interpolation [14]. The Hermite interpolating function for a generic property p is defined in standard tensor notation as follows:

$$p(u, v, w) = F_l(u)F_m(v)F_n(w)\mathbf{b}_{lmn} \quad (33)$$

where the following vectors define the cubic blending functions:

$$\begin{aligned} F_1(u) &= 2u^3 - 3u^2 + 1 \\ F_2(u) &= -2u^3 + 3u^2 \\ F_3(u) &= u^3 - 2u^2 + u \\ F_4(u) &= u^3 - u^2 \end{aligned} \quad (34)$$

and similarly for $F_m(v)$ and $F_n(w)$. The blending functions clearly show the cubic nature of the interpolating polynomial and are evaluated based on the point within the cube at which the interpolated value is sought, denoted by the star in Figure 3.

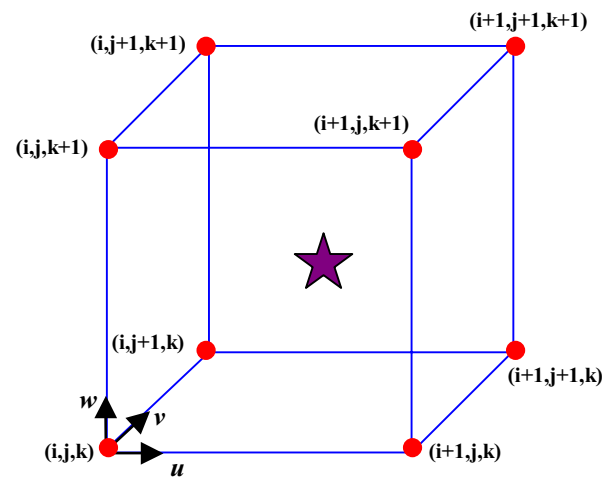


Figure 3. Hermite interpolation cube

The tensor \mathbf{b}_{lmn} is comprised of externally-provided data about the function p and its derivatives, data that is required at the eight cube vertices shown and labeled in Figure 3. The data consists of eight pieces of information at each of the eight vertices, a total of 64 pieces of information for a single cube: the function value, three tangent vectors, three twist vectors, and a vector defined by the third-order mixed partial

derivative of the function. See [14] for a complete description of the Hermite interpolation scheme and data required.

Clearly a significant amount of data is required for the interpolation of the thermodynamic properties h and u . With some symbolic manipulation of the property equations, it is possible to derive all the necessary function and derivative data analytically without resorting to numerical differentiation. This data is available to Modelica in the form of a Matlab .mat file. A typical $30 \times 45 \times 45$ data file is approximately 7.6 MB.

4.4 Modelica Implementation

Once we have decided on an appropriate interpolation scheme and collected all the property data required, the next step is to implement the interpolation scheme so that it can be used from within our Modelica models. In our implementation, the interpolation and gradient calculations (associated with the interpolation function via the derivative annotation) were written in "C". These functions are then called as external functions by native Modelica functions.

While the steps required are straightforward, there are several implementation details worth discussing. For example, in order to perform the interpolation, the property data must be loaded and made available to the "C" language routines. Rather than load the data (which is quite voluminous) into Modelica arrays and pass it as an argument to the various functions, we chose instead to load the data into memory and simply refer to it using an integer identification number. As a result, the only data passed around in the Modelica models is the unique ID number that identifies where the data can be found in memory. In the future, the interpolation routines will be upgraded to use the newly adopted `ExternalObject` class in Modelica 2.0 [8] that was introduced to provide more direct support for these kinds of applications.

Another issue with the interpolation routines is to improve performance by implementing some form of caching mechanism. There are two reasons to implement a cache mechanism. First, the simulation tool may not entirely optimize away redundant function calls (*i.e.* calls with the same arguments and therefore the same results). In such cases, a cache can be used to store previous results

and avoid expensive recalculations. Another reason to implement a cache is to allow for common calculation to be shared among the various interpolation-related functions. For example, calculating the gradient of the interpolating function requires much of the same data as the function evaluation itself. These common quantities can also be stored in a cache and reused across calculations.

Once we have implemented the interpolation routines, we can move on to implementing a medium model that utilizes these interpolation routines. This calculation involves two different interpolations. First, the properties of the gaseous air-fuel mixture (which is treated as a non-reacting mixture of ideal gases) can be computed via interpolation in temperature. Then, the properties of the reacting combustion products are computed using interpolation in pressure, temperature and equivalence ratio. These two sets of properties are then combined to form a single set of properties for the entire air, fuel and combustion products mixture.

Although we implemented our own interpolation routines for this purpose, we will work toward incorporating similar functionality into the Modelica Standard Library so that the routines can be more fully optimized and so that future users will be able to simply reuse what is in the library rather than having to create their own.

5 Cycle Simulation Results

The single-cylinder Modelica model was run for 1 second of simulation time at 1500 rpm at slightly lean conditions. By the end of the tenth engine cycle, approximately 0.8 seconds, the model has effectively converged to "steady-state"; *i.e.*, each cycle is a repeat of its predecessor. Some results for the tenth cycle are shown in Figures 4-5.

Figure 4 plots the temperature of all three zones. During mixture preparation the temperatures are equal, as is required. At spark, they separate, with the adiabatic temperature exceeding 2600 K, the unburned zone peaking at around 900 K, and the boundary layer achieving a value in between. These temperatures agree well with those computed in GESIM and are typical of those encountered in spark-ignited engines. At EVO, when the zones are remixed, all temperatures again quickly collapse to a single value, as expected.

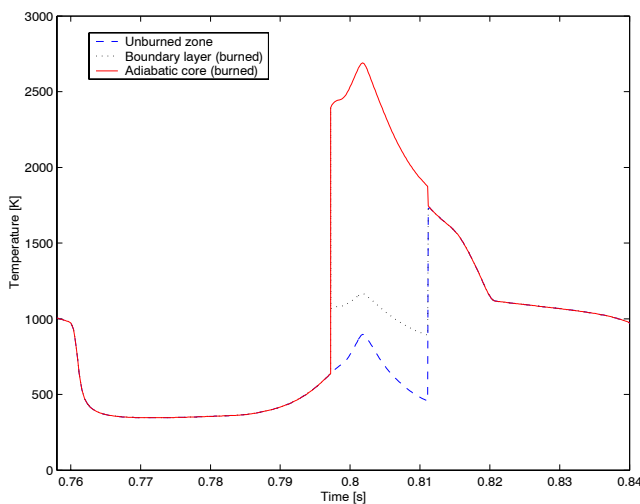


Figure 4. Zone temperatures (converged cycle)

The volumes of the three zones, along with the total chamber volume, are plotted in Figure 5. Since some of the zones during parts of the cycle are artifacts of our modeling approach, we cannot use measurements or GESIM to gauge their accuracy. However, they do behave as anticipated. During mixture preparation, when the adiabatic zone and the boundary layer do not appear in GESIM, their volumes are indeed insignificant. When combustion begins those two zones grow rapidly at the expense of the unburned zone, eventually reducing the size of the latter to insignificance at the end of combustion. At EVO, when the zones are remixed, all zones quickly revert to their values for mixture preparation.

6 Conclusions

This paper outlines the handling of zone formation/destruction and calculation of realistic thermodynamic properties of the cylinder contents in Modelica for engine cycle simulation. The results illustrate that Dymola and Modelica are capable of handling the complex physical models required for predictive cycle simulation. Furthermore, the techniques used provide illustrative examples for the handling of similar behavior in different applications.

Acknowledgements

The authors would like to acknowledge Dr. Anne Marsan at Ford Research Laboratories for the helpful discussions on geometry and interpolation. In addition, we would also like to acknowledge the support provided by Dynasim AB during the development of these models.

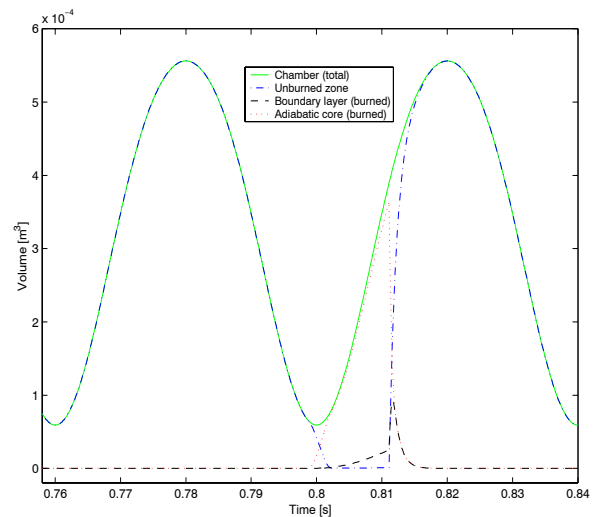


Figure 5. Zone volumes (converged cycle)

References

1. Heywood, J.B., 1988, *Internal Combustion Engine Fundamentals*. McGraw-Hill.
2. Tiller, M. M., 2001, *Introduction to Physical Modeling with Modelica*. Kluwer.
3. Borgnakke, C., *et al.*, 1980, "A Model for the Instantaneous Heat Transfer and Turbulence in a Spark Ignition Engine," SAE-80-0287, Society of Automotive Engineers.
4. Newman, C.E., *et al.*, 1989, "The Effects of Load Control with Port Throttling at Idle---Measurements and Analyses", SAE-89-0679, Society of Automotive Engineers.
5. Brehob, D. D., and C. E. Newman, 1992, "Monte Carlo Simulation of Cycle by Cycle Variability," SAE-92-2165, Society of Automotive Engineers.
6. Miller, R., *et al.*, 1998, "Comparison of Analytically and Experimentally Obtained Residual Fractions and NO_x Emissions in Spark-Ignited Engines", SAE-98-2562, Society of Automotive Engineers.
7. Modelica Association, 2000, "Modelica Language Specifications (Version 1.4)", www.modelica.org.
8. Modelica Association, 2002, "Modelica Language Specifications (Version 2.0)", www.modelica.org.
9. Tiller, M.M., *et al.*, 2000, "Detailed Vehicle Powertrain Modeling in Modelica", Modelica Workshop 2000 Proceedings, pp. 169-178.
10. Bowles, P., *et al.*, 2001, "Feasibility of Detailed Vehicle Modeling", SAE-2001-01-0334, Society of Automotive Engineers.
11. Olikara, C. and Borman, G. L., 1975, "A Computer Program for Calculating Properties of Equilibrium Combustion Products with Some Applications to I.C. Engines", SAE-75-0468, Society of Automotive Engineers.
12. Dymola. Dynasim AB, Lund, Sweden, www.dynasim.se.
13. Gamma, E., 1995, *Design Patterns*. Addison-Wesley.
14. Mortenson, M.E., 1985, *Geometric Modeling*. John Wiley and Sons.

Code Fragments

This section contains code fragments to illustrate some of the points raised during discussion of zone formation and destruction.

```

constant Integer mixprep=1;
constant Integer ignstep1=2;
constant Integer ignstep2=3;
constant Integer propagating=4;
constant Integer expanding=5;
parameter Modelica.SIunits.Time tau=1e-4;
parameter Modelica.SIunits.Time ign_delta=1e-6;
Integer status;
discrete Modelica.SIunits.MassFlowRate
  ignition_rate "Combustion rate during ignition";
discrete Modelica.SIunits.Time endstep(start=tau);
equation
mix_zones.mix.signal[1] = status == mixprep;
mix_zones.tfinal.signal[1] = endstep;
kernel_burn.burn.signal[1] = status == ignstep1;
kernel_burn.burn_rate.signal[1] = burn_rate;
flameadv.burn.signal[1] = status == ignstep2 or
  status == propagating;
flameadv.burn_rate.signal[1] = burn_rate;
.
.
if status == ignstep1 or status == ignstep2 then
  burn_rate = pre(ignition_rate);
elseif status == propagating then
  // post_ignition_rate is computed by flame
  // propagation model
  burn_rate = post_ignition_rate;
else
  burn_rate = 0.0;
end if;

algorithm
when status == mixprep and spark.signal[1] then
  status := ignstep1;
  endstep := time + ign_delta;
  kernel_mass := pre(burned_mass);
  ignition_rate := kernel_mass/ign_delta;
end when;

when status == ignstep1 and time > endstep then
  status := ignstep2;
  endstep := time + ign_delta;
  ignition_rate := pre(burned_mass)*
    (2.0*initial_kernel_size/pre(burnedV) -
    1.0)/ign_delta;
end when;

when status == ignstep2 and
  (time > endstep or
  burnedV/initial_kernel_size > 1.0) then
  status := propagating;
end when;

when status <> mixprep and
  endofexpansion.signal[1] then
  status := mixprep;
end when;

when status == mixprep then
  endstep := time + tau;
end when;

```

Figure 6. Excerpt from the combustion model

```

parameter Modelica.SIunits.Volume Vbtarget
  "Unburned kernel size";
constant Integer nindep=MediumModel.nspecies - 1
  "Number of independent species fractions";
discrete Modelica.SIunits.Time endstep;
Modelica.SIunits.MassFraction dX[nindep];
Modelica.SIunits.Temperature dT=a.T - b.T;
Real dV=1.0 - Vbnorm;
Real Vbnorm(start=5, fixed=true);
discrete Modelica.SIunits.Temperature deltaT;
Modelica.SIunits.MassFraction deltaX[nindep];
discrete Real deltaV;
discrete Real r;
Real rate_expr;
.
.
equation
// component a refers to the unburned zone, b to
// the burned zone
a.P = b.P;
a.q + b.q = 0.0;
a.mdot = -b.mdot;
// volume_b.signal[1] is the volume of the
// burned zone
Vbnorm = volume_b.signal[1]/Vbtarget;
rate_expr = 2.0*r^2*min(0.0, time - pre(endstep));
dX = a.X[1:nindep] - b.X[1:nindep];
if mix.signal[1] then // true at EVO
  der(dV) = pre(deltaV)*rate_expr;
  der(dX) = pre(deltaX)*rate_expr;
  der(dT) = pre(deltaT)*rate_expr;
else
  a.q = 0.0;
  a.mdot = zeros(MediumModel.nspecies);
end if;

algorithm
when mix.signal[1] then
  deltaX := dX;
  deltaT := dT;
  deltaV := dV;
  endstep := tfinal.signal[1];
  r := 1.0/(endstep - time);
end when;

```

Figure 7. Excerpt from the mix_zones model

```

equation
// Component medium is the burned zone
if burn.signal[1] then
  // MediumModel.BurnMixture computes the
  // composition after a given mixture is burned
  medium.mdot = burn_rate.signal[1]*
    (Xbase - MediumModel.BurnMixture(Xbase));
else
  medium.mdot = zeros(MediumModel.nspecies);
end if;
medium.q = 0.0;

algorithm
when burn.signal[1] then
  Xbase := medium.X;
end when;

```

Figure 8. Excerpt from the kernel_burn model

```

equation
a.q + b.q = 0;
if burn.signal[1] then
  a.q = burn_rate.signal[1]*a.h;
  a.mdot = burn_rate.signal[1]*a.X;
  // MediumModel.BurnMixture computes the
  // composition after a given mixture is burned
  b.mdot = -burn_rate.signal[1]*
    MediumModel.BurnMixture(a.X);
else
  b.q = 0;
  a.mdot = zeros(MediumModel.nspecies);
  b.mdot = zeros(MediumModel.nspecies);
end if;

```

Figure 9. Excerpt from the flameadv model

Session 6b

Thermodynamic Systems I

Modeling and Simulation of Refrigeration Systems with the Natural Refrigerant CO₂

Torge Pfafferott* Gerhard Schmitz†
 Technical University Hamburg–Harburg (TUHH)
 Department of Technical Thermodynamics (6-08)

March 2002

Abstract

This paper presents the current results of the development of a ModelicaTM library for CO₂-Refrigeration systems based on the free Modelica library ThermoFluid.

The development of the library is carried out in a research project of EADS Airbus and the TUHH and is focused on the aim to get a library for detailed numerical investigations of refrigeration systems with the rediscovered, natural refrigerant carbon dioxide (CO₂).

A survey of the CO₂-Library is given and the modeling of CO₂-Heat exchangers is described in detail. A comparison with steady state results of heat exchangers is presented and results of a transient simulation run are discussed with respect to plausibility.

1 Introduction

The fact of climate changes due to ozone depletion and global warming has been directed to significant research activities on the field of refrigeration and air-conditioning since the 1990s [7]. The objective of the investigations may yield to a long-term solution. Therefore so called natural, resp. alternative refrigerants with no Ozone Depleting Potential (ODP) and no or a very low Global Warming Potential (GWP) are investigated and new technical developments are driven. Carbon dioxide (CO₂, R 744) as a natural refrigerant was rediscovered and has recently a very high potential to substitute currently used refrigerants in the area of mobile/automotive air-conditioning and refrigeration. This development is caused by the excellent thermodynamic, transport and environmental properties of CO₂. Due to the critical data of CO₂ the process must be re-

alized as a transcritical cycle, which requires special control strategies.

In order to obtain a better understanding of the complex thermodynamic and hydraulic behaviour of CO₂-Refrigeration processes under steady and dynamic boundary conditions the modeling of components of a CO₂-System has been realized. A CO₂-Model library in ModelicaTM was built up by using base classes of the free Modelica library ThermoFluid [14]. The scope of the CO₂-Library is the modeling of the system behaviour by consideration of the most important physical effects like compressible flow, heat transfer, pressure drop, large capacities and time delays.

The development of a CO₂-Library is carried out in a research project of European Aeronautic Defence and Space Company (EADS) Airbus and the Department of Technical Thermodynamics of the Technical University Hamburg–Harburg (TUHH). The main objective of the project is a proof of concept of a CO₂ based integrated cooling system on board of future airliners. For this purpose numerical and experimental investigations are in progress.

2 Carbon dioxide as refrigerant

Carbon dioxide was used as a refrigerant until the 1930s, but was then replaced by the synthetic refrigerants (HCFCs) that offered lower absolute pressures, simpler techniques and higher efficiencies in conventional vapor compression cycle. Due to the ODP and the GWP of the synthetic refrigerants substantial research activities on the field of refrigerants are initiated since the 1990s. Recent research on carbon dioxide is pushed for mobile, resp. automotive air-conditioning and refrigeration and has focused on the development of a transcritical cycle [2]. Figure 1 illustrates the GWP for the three refrigerants R 12, R 134a and CO₂; the use of R 12 is forbidden in Europe since the be-

*pfafferott@tu-harburg.de

†schmitz@tu-harburg.de

gining of 1990s. The refrigerant R 134a today is the most common refrigerant in mobile and automotive air-conditioning systems.

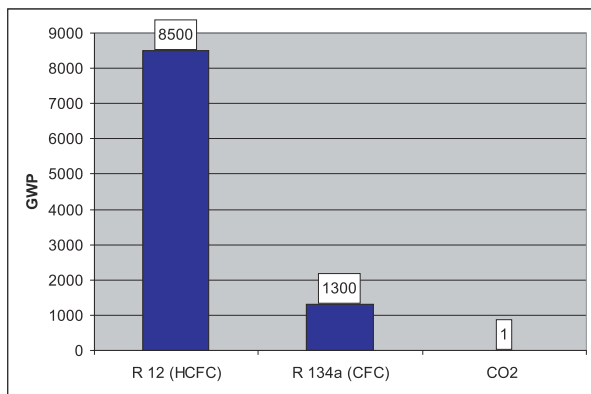


Figure 1: GWP of three different kind of refrigerants; GWP is standardised to 1 for CO₂

2.1 CO₂-Refrigeration cycle

The temperature and pressure at the critical point of CO₂ are 304,13 K and 73,77 bar. Therefore, the refrigerant cycle has to be operated transcriticalally when the ambient temperature is near or higher than the critical temperature. In this case the evaporation takes place at subcritical pressure and temperature and the heat rejection at supercritical state. At the supercritical status area pressure and temperature are not coupled anymore; so a CO₂-System has one more degree of freedom than conventional vapour compression cycles.

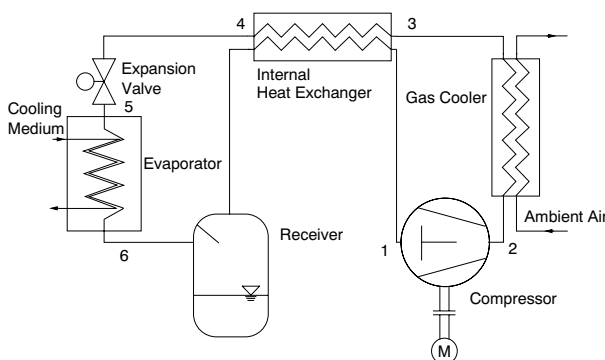


Figure 2: Schematic diagram of a CO₂-Refrigeration cycle

As shown in figure 2, the main components of a CO₂-Refrigeration cycle are compressor, gas cooler (instead of a condenser because of the supercritical heat rejection,

that occurs sometimes), internal heat exchanger, expansion valve, evaporator and low-pressure receiver. The process path of a transcritical CO₂-Cycle is shown in figure 3. The path represented by 1-2-3-4-5-6 shows compression (1-2), isobaric heat rejection at gas cooler (2-3), isobaric cooling in the internal heat exchanger (3-4), adiabatic expansion (4-5), isobaric evaporation (5-6) and isobaric superheating at internal heat exchanger (6-1). In steady state the low-pressure receiver has no influence of the process. For more detailed explanation of the CO₂-Cycle see [6], [5].

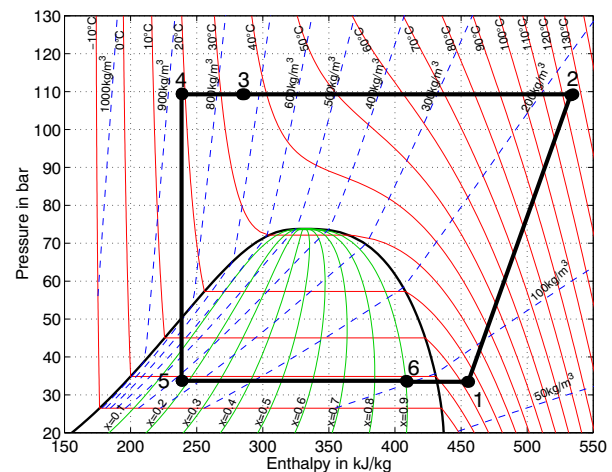


Figure 3: p,h -Diagram with states of a CO₂-Refrigeration cycle

3 CO₂-Library

The aim of the modeling is to create a library with physical based models of the above mentioned components. Such a library with models of these components and of additional components for testing, like sinks and sources, can be used for investigations of both, single components and complete refrigeration cycles. Furthermore it is of great interest to make dynamic simulation as well as steady state simulation of CO₂-Systems and single components, especially heat exchanger. Up to now, there is no commercial or free available simulation tool enabling dynamic and steady state simulation of CO₂-Cycle and -Components with only physical based models. There are some tools for steady state simulation but they need measured characteristics of the heat exchangers as an input.

The numerical investigation of heat exchanger components is of particular interest to find optimized heat exchangers for limited space. On the other hand the

concept of connectors in Modelica provides the opportunity using the same heat exchanger models for single component simulation as well as for a complex cycle simulation.

There are different backgrounds for modeling and simulation of complex, closed CO₂-Refrigeration cycles. The first aim is a better understanding of the complex, coupled thermodynamic, fluidmechanic and heat transfer effects in a transcritical operating CO₂-System. Here the influence of some typical system parameters like compressor speed, heat exchanger and receiver geometry and refrigerant filling can be tested. Furthermore aspects of the control of the system should be investigated. Finally, the library is used for simulation and evaluating of different system design in various applications.

The library is based on free Modelica library ThermoFluid. The ThermoFluid library, especially the base classes and partial components, is in regard to the implementation of the three balance equations (energy, mass, momentum) and the method of discretization (finite volume) very well suited for modeling of CO₂-Systems. In cooperation with the developers of ThermoFluid a high accuracy medium model for CO₂ based on an equation of state was implemented for the whole fluid region [9].

3.1 Survey of CO₂-Library

So far, the following models and classes have been implemented:

- **Heat transfer and pressure loss relations for the whole fluid region:**
This constitutive equations are used for the calculation of heat flux and pressure drop due to friction, which are added to the balance equations of energy and momentum [10], [11].
- **Models for the air side of heat exchangers:**
The balance equation of energy is implemented by the finite volume method [8]; well suited heat transfer correlations for the air side have been implemented [4].
- **Pipes and heat exchangers:**
Based on the medium model, classes of ThermoFluid, the heat transfer and pressure drop correlations and the air side models pipes and heat exchangers have been modelled. The pipes are modelled with discretized parameters.

- **Compressor:**

The model is made for a reciprocating compressor. Therefore, the mass flow is calculated by the general equation of a reciprocating compressor and enthalpy change is calculated according to the isentropic efficiency. The compressor is modelled with lumped parameters.

- **Expansion valve:**

The throttling process is treated as isenthalpic and the pressure drop is calculated according to the flow coefficient of the valve [1]. The flow coefficient results by the specific valve construction and the opening ratio of the valve. Therefore, the flow characteristic of the valve has to be known and the model has to be parameterized with the corresponding values. For the valve model lumped parameters are used.

- **Receiver:**

Up to now, a simple receiver model is implemented. The model separates the incoming two phase flow into its vapour and liquid phase. As long as the liquid level of the receiver is lower than the outlet height saturated vapour leaves; if the liquid level reaches the outlet height a two phase flow leaves up to a height only liquid leaves. Due to the sophisticated construction of CO₂-Receivers in most of the operating modes a two phase flow leaves the receiver even if the liquid level is much lower the outlet height. It seems to be not easy to model this components with physical correlations; so the modelling is in progress.

- **Flow splits and junctions:**

For this models classes of ThermoFluid are used; for the pressure drop in the momentum equation special correlations for splits and junctions have been implemented taking the ratio of mass flow into account [3]. The change of mass flow direction is taken under account in the implementation.

4 Examples of Modeling

4.1 Modeling of heat exchangers

So far, available heat exchangers for CO₂-Refrigeration systems are compact prototype components from the automotive application, see figure 4. The heat exchangers are built up as follows: The CO₂-Flow is splitted in different streams through so called Flat-Tubes (or Multiport-Micro-Tubes),

see figure 5. The Flat-Tubes consists of a number of parallel bores in which the CO_2 flows. The refrigerant is splitted and collected at the feeder and manifold of the heat exchangers. Outside the heat exchanger air passes over slitted fins enhancing the air side heat transfer area and heat transfer coefficient, see figure 6.



Figure 4: CO_2 -Gas cooler

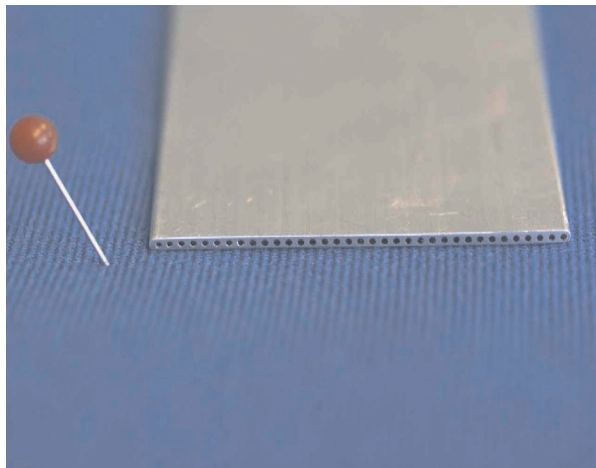


Figure 5: Cross section of a Flat-Tube

In a heat exchanger different flow paths for the CO_2 are possible; usually gas coolers are constructed as crossflow and evaporators are built up as cross-counterflow heat exchangers. In figure 7 the schematic flow path of CO_2 through a crossflow heat exchanger is shown; e.g. here the CO_2 has three transits through the heat exchanger. At every transit the CO_2 -Flow is splitted in a number of parallel Flat-Tubes, the bores of every Flat-Tube are flowed through concurrent. For the modeling of the CO_2 -Flow a homogenous distribution of the flow is supposed. By this assumption the flow is modelled by one single pipe. The

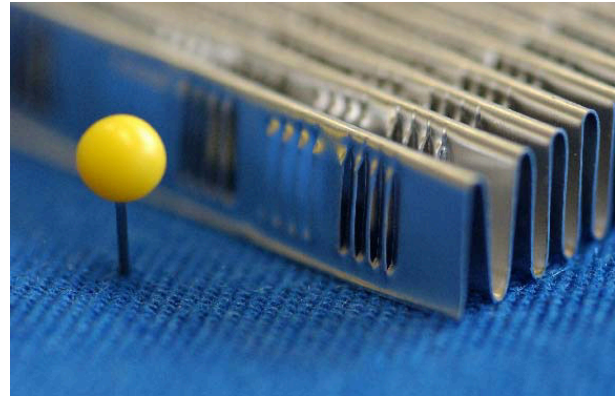


Figure 6: Slitted fins

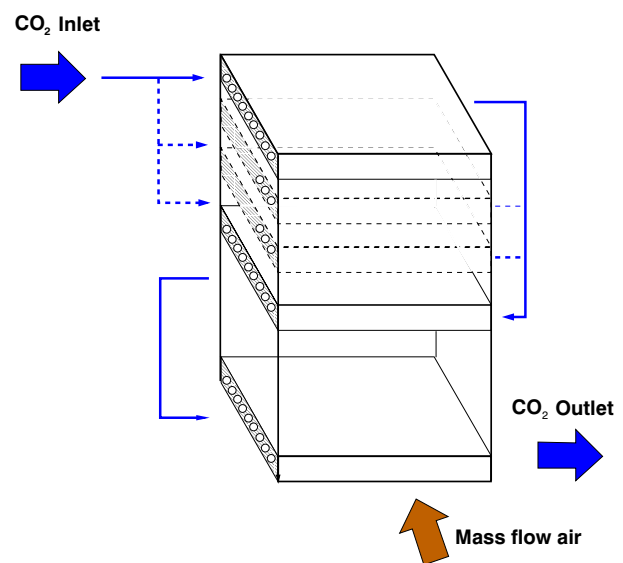


Figure 7: Flow path through a cross flow heat exchanger

heat transfer area and the flow cross section are determined by the geometry and the number of all concurrent flowed pipes; whereas the heat transfer coefficient and the pressure loss is calculated with the mass flow rate and the geometry of a single bore.

The assumption of homogenous mass flow and temperature distribution is also made for the air side. Therefore, it is possible to model the air flow through one air channel. In the modeling can be assumed that the slitted fins and the Flat-Tubes create a triangular channel, see figure 6. So the total mass flow of air is scaled down to the mass flow through one channel by division through the total number of air channels of the heat exchanger. At the air side only the energy balance equation is implemented by the finite volume method. The medium properties are introduced by polynomi-

nales fitting the properties well in the temperature intervall of 253.15 K to 342.15 K. Because the air side heat transfer is much lower than on the refrigerant side, a detailed physical model based on characteristic numbers and geometry parameters and validated on experimental investigation has been found by a literature review and is implemented [4].

The wall model is taken from the ThermoFluid library. It is modeled as a capacitive, cylindric wall.

This specific models of CO₂-Pipe, wall and air have to be connected in the right way to get a reasonable model of a heat exchanger. For the connection the heat connectors of ThermoFluid can be used; the connecting variables are temperature and heat flux. The implementation, especially the connection is as follows: First the same number of air channel objects is created like the discretization number of the pipe and wall. The air channel model itself can be discretized in air flow direction with another number. At the connection to the air side the calculation of the heat flux for one single air channel has to be taken into consideration. Therefore it has to be scaled up by a factor of the numbers of total air channels and the discretization number. In the modeling a class is programmed where the air channel objects are declared and where the scaling is programmed. Furthermore, every air channel object is connected with the wall temperature of the equivalent, discretized wall element. So every volume of a discretized air channel gets the same wall surface temperature. The following code example shows this implementation; here *geoHX.pipe_n* means the discretization of pipe and wall and *geoHX.AC* means the discretization of air flow:

```
model AirChannelDCrossFlow
...
Co2Flow.Air.DiscAirChannelDDry
AirChannels[geoHX.pipe_n];
ThermoFluid.Interfaces.HeatTransfer.HeatFlowD
AirHT(n=geoHX.pipe_n) "Heat connector";
equation
  for ac in 1:geoHX.pipe_n loop
    for i in 1:geoHX.AC_n loop
      AirChannels[ac].T_W[i] = AirHT.T[ac];
    \\ Air surface temp. connected
    \\ with heat connector
    end for;
    ...
    AirHT.q[ac] = AirChannels[ac].Q_dot_total*
      geoHX.total_channels/geoHX.pipe_n;
    \\ Heat flow at the connector is scaled
    end for;
    ...
end AirChannelDCrossFlow;
```

A schematic illustration of the modeling idea and the connections is shown in figure 8. The implementation

of a heat exchanger in Modelica is shown in figure 9 as the graphical representation in the modeling and simulation tool DymolaTM.

The implementation of a cross counter heat exchanger can be realized now easily. Only the connections of temperature and heat flow have to change in the class *AirChannelDCrossFlow* in a specific way.

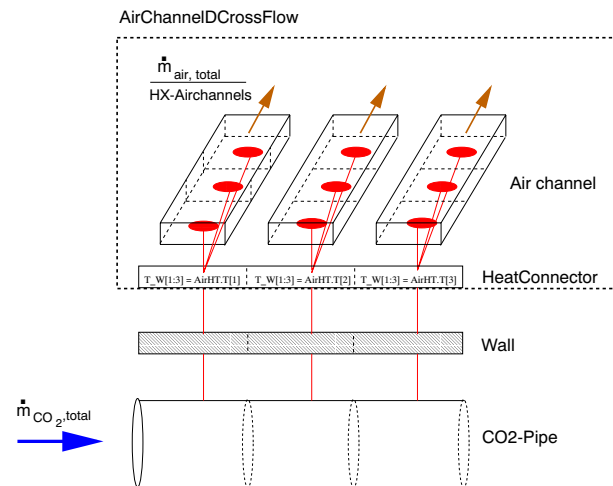


Figure 8: Schematic illustration of the modeling of heat exchangers

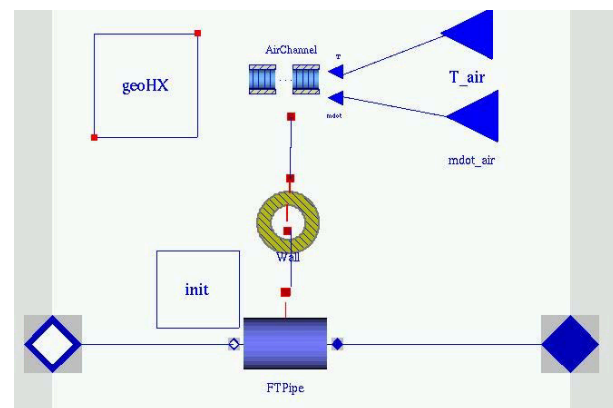


Figure 9: Graphical representation of the heat exchanger model

4.1.1 Comparison of steady state simulation and measurement

With these models simulations in a test configuration have been run. The test configuration consists of a source providing pressure and enthalpy at the heat exchanger inlet and a mass flow sink generating a defined mass flow at the outlet. The source and sink are used

to set the boundary conditions resulting from the measured data at the component.

The following comparison is made for a crossflow gas cooler and cross-counter flow evaporator from the CO₂-Experimental system built up at the Department of Aircraft Systems Engineering of the TUHH. The geometry parameters of the components are known. In the tables 1 and 2 the measured data and the results of the simulations at the point of steady state are shown.

The comparison of experimental data and simulation results shows a very good correspondence, especially if you take under account that the printed experimental data are taken as is. The tolerance of the sensors has not been taken into account, yet.

Table 1: Comparison of measured data at a gas cooler with simulation results in steady state

Boundary conditions from measured data					
\dot{m}_{air}	\dot{m}_{CO_2}	\bar{p}_{CO_2}	$T_{CO_2,in}$	$h_{CO_2,in}$	$T_{air,in}$
[kg/s]	[kg/s]	[bar]	[K]	[kJ/kg]	[K]
0,605	0,013	96,0	395,4	538,5	308,9
0,593	0,032	87,5	355,0	487,2	312,9
0,598	0,036	88,3	373,4	513,8	312,9
Measured data			Simulation		
$T_{CO_2,out}$	$T_{air,out}$	\dot{Q}_{CO_2}	$T_{CO_2,out}$	$T_{air,out}$	\dot{Q}_{CO_2}
[K]	[K]	[kW]	[K]	[K]	[kW]
309,5	315,7	-3,02	312,7	313,5	-2,85
314,7	320,2	-3,45	316,2	318,0	-3,14
315,3	323,3	-4,99	317,4	320,0	-4,37

Table 2: Comparison of measured data at an evaporator with simulation results in steady state

Boundary conditions from measured data					
\dot{m}_{air}	\dot{m}_{CO_2}	\bar{p}_{CO_2}	$T_{CO_2,in}$	$h_{CO_2,in}$	$T_{air,in}$
[kg/s]	[kg/s]	[bar]	[K]	[kJ/kg]	[K]
0,21	0,032	49,1	286,7	295,3	301,6
0,21	0,036	40,3	278,7	281,3	294,7
0,21	0,013	34,6	272,9	222,1	285,2
Measured data			Simulation		
$h_{CO_2,out}$	$T_{air,out}$	\dot{Q}_{air}	$h_{CO_2,out}$	$T_{air,out}$	\dot{Q}_{CO_2}
[kJ/kg]	[K]	[kW]	[kJ/kg]	[K]	[kW]
372,8	289,85	-2,48	374,8	289,6	2,54
357,4	281,95	-2,74	357,4	282,0	2,75
378,2	277,65	-2,03	380,3	275,6	2,04

4.2 Implementation of constitutive equations

In order to obtain a most physical modeling of CO₂ flow through pipes and any kind of heat exchangers constitutive equations for pressure drop and heat transfer for the whole fluid region are implemented according to [11], [10]. A comparison of implemented relations with experimental data from the SINTEF [12] shows a good correspondence [13]. The pressure drop and heat transfer correlations are empirical equations which only are exactly valid for steady state. Due to the fact that such correlations for dynamic state are not available it seems to be the best and a very common method for describing these effects in a dynamic simulation.

The correlations have been implemented with regard to numerical robustness and simulation time. At the foldover between laminar and turbulent flow the describing empirical equations of heat transfer and pressure drop have no steady transition. By avoiding event iterations in this case a function for the smooth transition of the pressure drop coefficient at a Reynolds number of 2300 is shown in figure 10 with a solid line. The dashed line between Reynolds numbers of 2000 and 3000 shows the run of the interpolated pressure drop coefficient. The interpolation function fulfills the following requirements:

- The gradient inbetween the limits of validity is always smaller than infinity.
- The gradient near the limits of the intervall is nearly zero.
- Exactly at the limits of validity the interpolation function calculates the exact value of the current function.

The interpolation function is implemented by using the *tanh*- and the *tan*-function as follows:

```
function Stepsmoothen
//Interpolationsfunction to avoid event iterations
input Real func;
//value, where function value becomes 100%
input Real nofunc;
//value, where function value becomes 0%
input Real x;
//Variable generating the event
output Real result;
protected
Real m;
Real b;
algorithm
m := Pi/(func - nofunc);
```



```

b := -Pi/2 - m*nofunc;
result := (tanh(tan(m*x + b)) + 1)/2;
end Stepsmoothen;

```

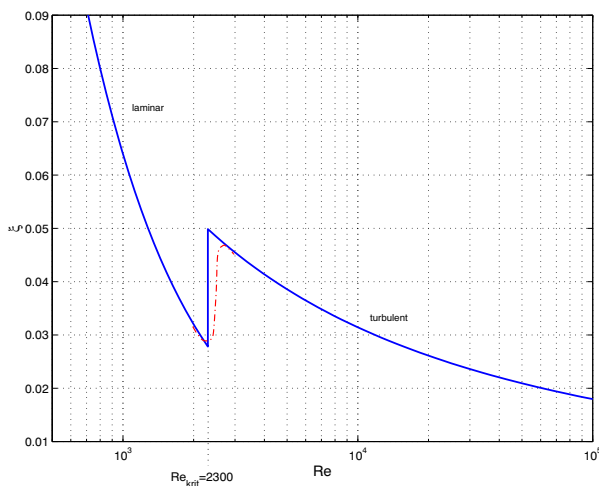


Figure 10: Example for the unsteady transition of the pressure drop coefficient (solid line) and the implemented interpolation function (dashed line)

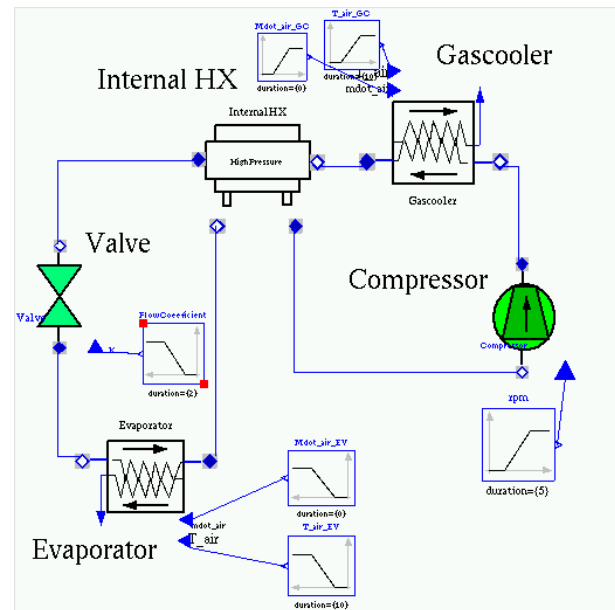


Figure 11: Object diagram of simulated CO₂-Cycle

Table 3: Boundary conditions and initial values of the simulation run

Compressor	$\lambda = 0.75, \eta_{is} = 0.75$
Gas cooler	$\dot{m}_{air} = 3200 \text{ kg/h}, T_{air,in} = 305 \text{ K}$
Evaporator	$\dot{m}_{air} = 580 \text{ kg/h}, T_{air,in} = 305 \text{ K}$
System volume	$V_{tot} = 1.13 \text{ l}$
Refrigerant filling	200 kg/m^3
Initial value	$p_0 = 66 \text{ bar}, h_0 = 420 \text{ kJ/kg}$

5 Simulation results of a CO₂-System

In the following simulation results of the start up of a CO₂-System are presented. The results are discussed with respect to plausibility since reliable data of transient processes from a test rig are only available for a few weeks. The simulated model is shown in object diagram in figure 11. This configuration does not consist of a receiver since the receiver model is not implemented in the right way, see subsection 3.1.

In table 3 the boundary conditions and initial values are listed. The following boundary conditions are changed during the simulation run:

- Start up of compressor speed $n = 120 \rightarrow 1000 \text{ rpm}$ in 2 seconds;
- Variation of flow coefficient $K_v = 0.03 \rightarrow 0.02 \text{ m}^3/\text{h}$ in 0.1 seconds starting at 60 seconds simulation time.

5.1 Results

In figure 12 the pressure at compressor inlet and outlet is plotted versus time. What can be seen from the results is the divergent run of the pressures and a typical overshoot, resp. undershoot at the beginning.

This system behaviour is plausible as well as the divergent run of pressure after changing the flow coefficient of the valve. This can be made clear by looking at the mass flow rates at the compressor and the expansion valve in figure 13. At the beginning the compressor mass flow rate is much higher than the mass flow at the valve. The compressor mass flow increases proportional with the compressor speed, whereas the flow rate at the valve just increases with the increasing pressure difference at the valve. The difference between both mass flows effects a shifting of refrigerant mass from the low pressure section to the high pressure section of the system. The decreasing density of the sucked refrigerant at the compressor causes in the strong decreasing of the compressor mass flow after 2 seconds. The valve mass flow rate is mostly affected by the pressure difference, so the mass flow does not decrease; the system time delay causes a higher valve

flow rate for a few seconds resulting in the shown over- and undershooting of pressures. The same effect of displaced mass explains the divergent run of the pressures after changing the flow coefficient.

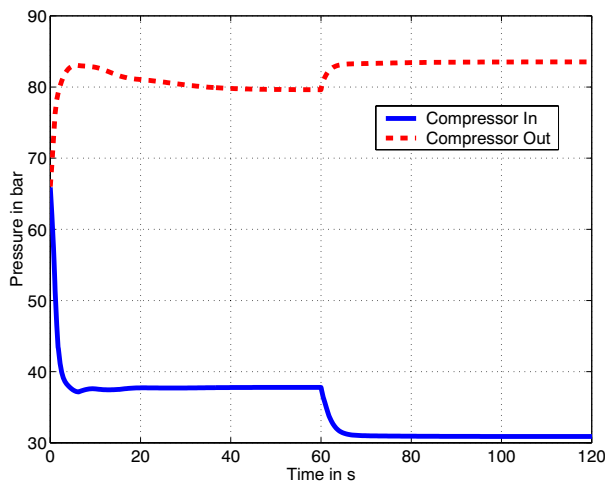


Figure 12: Pressure run at compressor in- and outlet

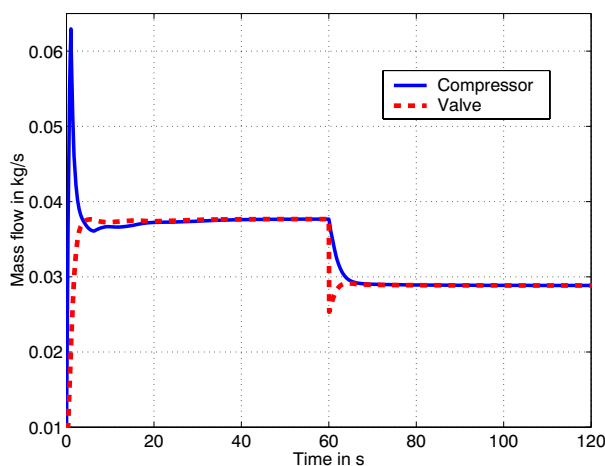


Figure 13: Mass flow rate at valve and compressor

This model contains about 2600 nontrivial scalar equations. The simulating of the start up and the changing of flow coefficient was performed on a PC with a Pentium 1000 MHz and 256 MB of main memory and took 5 minutes. The length of time of simulation is very sensitive due to the initial values. To realize a simulation of 5 minutes for this model you have to provide very suitable initial values; furthermore up to now the initialization in the two phase region needs wide experience. To get suitable initial values for the gas region we are using matlabTM based script to predict the steady state pressure drop for the initialized mass

flow rate. Starting a simulation with a mass flow rate of zero and equal pressure in every object increases the executing time extremely or generates a termination of simulation.

Nevertheless, the simulation results show that Modelica, the free Modelica library ThermoFluid and the CO₂-Library are very well qualified for the simulation of the complex processes in a CO₂-Refrigeration cycle.

6 Conclusion

A developed CO₂-Library based on free Modelica library ThermoFluid was presented, which contains models for all important components of a CO₂-Refrigeration system. The intention is to create a library for the simulation of single components and complete cycles. Such a library can be used to get a better understanding of the thermodynamic, fluid-mechanic and heat transfer effects in a CO₂-System. Furthermore, it can be used for the optimization of specific heat exchangers, for the evaluating of optimal system configuration and for the layout and optimization of the system control.

The presented simulation results for the steady state of two different types of CO₂-Heat exchangers show a very correspondence with measured datas. The results of transient simulation show a plausible system behaviour due to the thermodynamic and hydraulic effects. Up to now a validation with transient measured datas was not possible since an available CO₂-Test rig operates just for a few weeks.

Future work contains the validation of the models and the improvement of the initialization due to new features in Modelica. If the models are verified the control of the system will implemented.

6.1 Acknowledgement

This work is carried out in a research project financed by EADS Airbus in Hamburg, Germany.

The presented results and efforts would not have been possible without the help and the work of some people apart from the authors. We would like to thank Hubertus Tummescheit for providing ThermoFluid and the given support. Thanks to Stefan Wischusen, Guido Ströhlein, Till Gundlach, Torsten Meyer and Jörg Eiden for the implementation of various models in master thesis and project works. Thanks to Dirk Limperich for providing a lot of parameters of CO₂-Components, for the support of the implementation of CO₂-Medium

model, for the testing of several models and for constructive discussions and several ideas. Thanks to Oliver Schade for providing first data from his CO₂-Experimental system.

References

- [1] DIN EN 60534-2-1:
Stellventile für die Prozeßregelung Teil 2-1: Durchflußkapazität und Bemessungsgleichungen für Fluide unter Einbaubedingungen Deutsches Institut für Normung e.V., Beuth Verlag, Berlin, 2000 (in german)
- [2] Haffner, A., Pettersen, J., et al.:
An automotive HVAC system with CO₂ as refrigerant IIR Conference Natural Working Fluids - Preprints, pp. 289-298, Oslo, 1998
- [3] Idelchick, I.E.:
Handbook of Hydraulic Resistance CRC Press, Florida, 1994
- [4] Kakac, S. et al.:
Handbook of Single-Phase convective Heat Transfer John Wiley & Sons Inc., New York, 1987
- [5] Kauffeld, M., Hesse, U., Pettersen, J.:
Kohlendioxid in der Kälte-, Klima- und Wärmepumpentechnik Die Kälte- und Klimatechnik, No. 11, pp. 768-781, 1993
- [6] Lorentzen, G.:
Revival of carbon dioxide as refrigerant Int. Journal of Refrigeration, Vol. 17, No. 5, 1994
- [7] McMullan, J.T.:
Refrigeration and the environment - issues and strategies for the future Int. Journal of Refrigeration, Vol. 25, No. 1, pp. 89-99, 2002
- [8] Patankar, S. V.:
Numerical Heat Transfer and Fluid Flow Hemisphere Publ. Corp., Washington, 1980.
- [9] Span, R., Wagner, W.:
A New Equation of State for Carbon Dioxide Covering the Fluid Region from the Triple-Point Temperature to 1100 K at Pressures up to 800 MPa Journal of Physical and Chemical Reference Data, Vol. 25, No. 6, pp. 1509-1596, 1996.
- [10] Stephan, K.:
Wärmeübergang beim Kondensieren und beim Sieden Springer-Verlag, Berlin, 1988 (in german)
- [11] N.N.: *VDI-Wärmeatlas - Berechnungsblätter für den Wärmeübergang* VDI-Verlag, 7. Edition, Düsseldorf 1994 (in german)
- [12] Pettersen, J., Rieberer, R., Munkejord, S.T.:
Heat transfer and pressure drop for flow of supercritical and subcritical CO₂ in microchannel tubes Technical Report A5127, SINTEF Energy Research, Trondheim 2000
- [13] Pfafferott, T., Schmitz, G.:
Numeric Simulation of an integrated CO₂ Cooling System Proceedings of the Modelica Workshop 2000, Lund, 2000
- [14] Tummescheit, H., Eborn, J., Wagner, F.:
Development of a Modelica Base Library for Modeling of Thermo-Hydraulic Systems Proceedings of the Modelica Workshop 2000, Lund, 2000

Simulation of Thermal Building Behaviour in Modelica

F. Felgner, S. Agustina, R. Cladera Bohigas, R. Merz, L. Litz (felgner@eit.uni-kl.de)
 Fachbereich Elektrotechnik und Informationstechnik, Universität Kaiserslautern,
 Erwin-Schrödinger-Straße, D 67663 Kaiserslautern,
 March 2002

1. Abstract

During the past decades heating and air conditioning systems were usually designed and consequently oversized according to simplified, mostly static calculating procedures. The increase in primary energy costs, rising cost pressure felt by private and public clients as well as increased demands on comfort forced engineers to change the customary procedure. Thus the dynamic simulation of building and system behaviour plays an increasingly important role in planning and dimensioning heating and air conditioning systems. This change is supported by the growing performance of personal computers in use. This means that calculating methods which used to be too expensive and time-consuming became practicable and could even be improved.

Building and system simulation aims at emulating the thermal and energetic behaviour of an existing or a fictitious building and of its HVAC system as well as their interaction. For this purpose the external influences through the outdoor climate, user behaviour and internal loads are to be taken into account. The comprehensive building design requires the adequate description of real processes within a broad spectrum of mathematical, physical and engineering disciplines. The model of just an uncomplicated heating system includes various components from thermodynamics, fluid dynamics, mechanics, electrical and control engineering.

It is true there is a great variety of simulation tools - mostly conceived for architects and building engineers - varying according to the methods they use, the effects they consider as well as to their objectives. Such simulation tools pretend to offer a high transparency and flexibility through their menu-guided modelling but can often not be completely overlooked by the user as to their numeric methods, the effects considered and approximations applied. Operations going beyond what is provided by the menu are either not possible or can only be realized at great expense.

Therefore we intended to take another way. Using an open simulation system, which provides the mathematical formalism, the model specification is done by the description of basic physical laws describing the

relevant properties [Fel-01], [Mer-01], [Sit-01]. An object-oriented, non calculation-causal simulation language like *Modelica* offers perfect conditions for this concept.

In the context of our work a model library for the simulation of thermal building behaviour has been developed in *Modelica*. Due to the interdisciplinary character of building simulation this domain is an ideal application of *Dymola/Modelica*. We used *Dymola 4.1a* from *Dynasim* (<http://www.dynasim.se>).

The new model library is divided into four sublibraries:

- *Building* (chapter 2),
- *Weather* (chapter 3),
- *Heating* (chapter 4),
- *controller* (chapter 5).

The building models have been validated in exemplary configurations with the building simulation system *TRNSYS* [Trn-02], [Kle-00], [Kie-01].

In the following chapters the most interesting components or those sublibraries will be presented.

2.1 Basic Building Elements

The characteristic thermal behaviour of a building structure is determined by the storage and the conduction of heat within walls, ceilings, floors and the air inside and outside the building as well as the heat transmission between those components [VDI-01]. The processes of heat storage and transmission are described by basic building elements, which are the primary components of a thermal building model.

Heat storing elements (fig. 2.1a, b) correspond to electrical capacitors, where electrical current is replaced by heat flow j and the place of the electrical potential is taken by the temperature T :

$$m \cdot c \cdot \dot{T} = j \quad (2.1)$$

(m : mass of heat storing body, c : specific heat capacity).

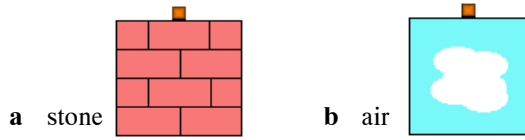


Fig. 2.1: Heat storing components of a building

Heat conducting elements (fig. 2.2) correspond to electrical conductors:

$$j_{1 \rightarrow 2} = \frac{\lambda \cdot A}{d} \Delta T = \underbrace{\frac{\lambda \cdot A}{x_2 - x_1}}_{\text{conductance } G} (T_1 - T_2) \quad (2.2)$$

(λ : heat conductivity, A : area perpendicular to heat flow j , d : distance between two heat storing elements with the temperatures T_1 and T_2).

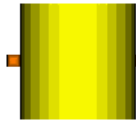


Fig. 2.2: Heat conductor

Convection (fig. 2.3) is described with the same mathematical structure if the convective heat transfer coefficient α is supposed to be a constant:

$$j_{1 \rightarrow 2} = \alpha \cdot A \cdot (T_1 - T_2). \quad (2.3)$$

Convection takes place between the air and walls, floors and ceilings inside the building as well as outside.

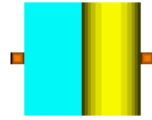


Fig. 2.3: Convection with constant α

Radiation – the third way of heat transfer – plays an important role, too, especially as far as solar radiation, the emission from radiators and the heat exchange between walls are concerned. The power $P_{\text{rad}} = j_{\text{rad}} = |\vec{j}_{\text{rad}}|$, emitted from a surface with the temperature T , is given by Stefan-Boltzmann's Law,

$$j_{\text{rad}} = \sigma \cdot \varepsilon \cdot A \cdot T^4 \quad (2.4)$$

(σ : Stefan Boltzmann constant, ε : emission coefficient of surface),

which is implemented in a model class (fig. 2.4) describing the exchange of radiation between a surfaces A with the temperature T_1 (e.g. the surface of a wall) and a fictive black body with the same surface and the temperature T_2 :

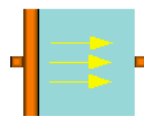


Fig. 2.4: Radiation to a black body

$$j_{1 \rightarrow 2} = \sigma \cdot \varepsilon_1 \cdot \underbrace{\varepsilon_2}_{=1} \cdot A \cdot (T_1^4 - T_2^4). \quad (2.5)$$

Those components will be used for the so-called *two-star room model* (see ch. 2.2).

The library also contains components simulating the radiation between two parallel or two perpendicular surfaces (fig. 2.5a, b). For this purpose the equation (2.5) has to be modified by an additional factor taking

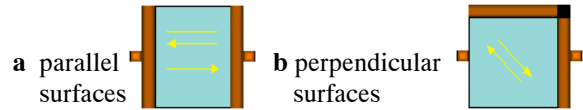


Fig. 2.5: Radiation between two surfaces

into account the surfaces' dimensions and relative site.

2.2 Composed Building Model Classes (walls, windows, doors, rooms)

The basic building elements presented in ch. 2.1 serve to compose models of more complex parts of a building. As a first instance the model of a *wall* be considered:

Within solid matter heat transport is provided by conduction. This means that in case of one-dimensional heat flow in x -direction the temperature $T(x)$ is given by the well-known partial differential equation

$$\rho \cdot c \cdot \frac{\partial T}{\partial t} = \lambda \cdot \frac{\partial^2 T}{\partial x^2}, \quad (2.6)$$

which cannot be implemented directly in *Modelica* as there is only one independent variable (time) provided. But the derivation in x can be approximated by discretizing the coordinate x into x_i ($i = 1, 2, 3, \dots$) with $\Delta x := x_i - x_{i-1} = \text{const. } \forall i$:

$$\frac{\partial^2 T}{\partial x^2} \approx \frac{T(x_{i+1}) - 2T(x_i) + T(x_{i-1}))}{(\Delta x)^2} := \frac{T_{i+1} - 2T_i + T_{i-1}}{(\Delta x)^2} \quad (2.7)$$

Thus the equation (2.6) can be approximated by

$$\begin{aligned} c\rho\dot{T} &= \lambda \frac{T_{i+1} - 2T_i + T_{i-1}}{(\Delta x)^2} \Leftrightarrow \\ c \cdot \underbrace{\rho A \Delta x}_m \cdot \dot{T} &= \underbrace{\frac{\lambda A}{\Delta x}}_G (T_{i-1} - T_i) - \underbrace{\frac{\lambda A}{\Delta x}}_G (T_i - T_{i+1}). \end{aligned} \quad (2.6')$$

Using the components for heat storage and conductance introduced in ch. 2.1 the equation (2.6') can

be implemented by the *Modelica* model in fig. 2.6, which is a simple thermal model of a wall with mass m , specific heat capacity c , surface A and thickness $2\Delta x$.

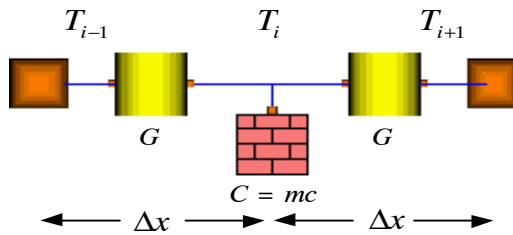


Fig. 2.6: Model of conduction within a wall

A more precise model of a wall is achieved by dividing the wall into several layers – at least two – and adding the convection model from fig. 2.4:

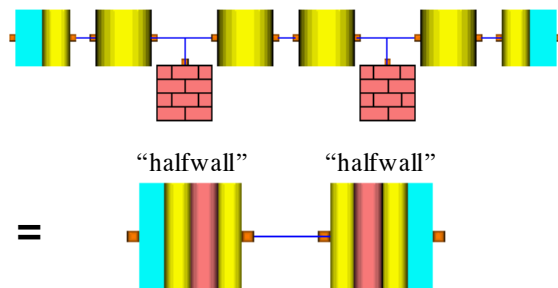


Fig. 2.7: Two-layer model of a wall (or floor / ceiling) with convective heat transmission to air

The model in fig. 2.7 is of course suitable for floors and ceilings, too.

Before constructing the thermal model of a room the radiative heat transmission between its walls, floor, ceiling and other radiating surfaces (e.g. radiator surface) shall be considered first. Using the models from fig. 2.5 would lead to a complicated network of connections between each surface and all the others. A room of a simple rectangular geometry would demand three instances of the parallel surfaces model (fig. 2.5a) and 12 instances of the perpendicular surfaces model (fig. 2.5b). A non-rectangular room geometry and the radiation from a radiator surface would demand very special additional model classes which are much more complex or even do not exist in a parameterised form, respectively.

For this reason the building library presented here makes use of a certain approximation – the so-called *two-star model* (see for example [Fei94]). In this model all radiating surfaces are connected to a fictive massless black body, which has an infinite heat conductivity and fills in the whole volume of the room. In the model diagram this body is simply a nodal point. Each long wave radiation emitting surface is connected to that nodal point via a radiation junction component from fig. 2.4 (ch. 2.1). A second nodal point provides the convective connection of the sur-

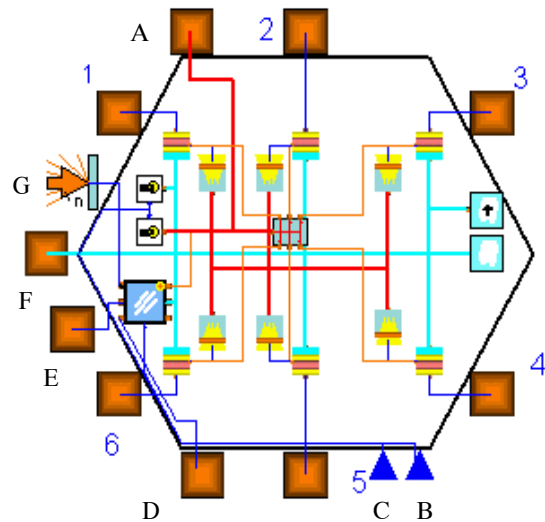


Fig. 2.8: Two-star model of room with four walls.

External cuts: 1 – 6 walls, floor, ceiling;
A radiation from external model (radiator);
B signal for air exchange through window;
C signal for internal gains;
D Radiation to sky through window;
E convection on the outside of the window;
F convection from or to external model;
G connection to solar radiation model.

faces to the air via convective junction components (fig. 2.3). Fig. 2.8 shows the two-star model of a room with four walls (plus floor and ceiling). Only one half of each wall is represented in the model of a room, the second half is part of a neighbouring room.

Additionally models for internal gains (with a light bulb in the icon) as well as the model of a window (fig. 2.8) describing air exchange and heat transmission by conduction, convection and radiation. There is a special cut in the window model and in the room model (“G” in fig. 2.8) providing a connection between the window and a model from the solar radiation library (see ch. 4). A radiator (as an external model) can be connected to the room via two cuts: Cut “A” provides the connection to the radiation nodal point, cut “F” transmits convective heat transport from the radiator to the air.

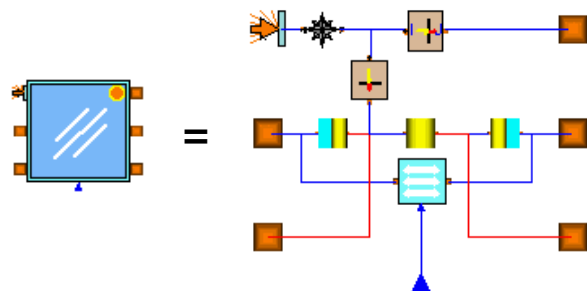


Fig. 2.9: Model of a window with controlled air exchange and heat transmission by conduction, convection and radiation (including solar radiation)

3. Solar Radiation Models

3.1 Introduction

Comprehensive modelling of thermal building dynamics requires considering the influence of nature. In order to simulate the impact solar radiation has on energy consumption and controller behaviour in a realistic manner a highly refined solar radiation library has been developed. With its models it is possible to calculate the solar radiation on a tilted surface at any location [Duf-74]. The models are very user-friendly since simulation periods can be defined by entering date and local clock time. Moreover, real weather data can be integrated in the simulation model by reading external ASCII files and interpolating the data in different ways. The models reproduce the broad spectrum offered by modern building simulation tools (e.g. *TRNSYS* [Trn-02], [Kle-00], [Spr-01]).

3.2 The Models

The solar radiation models contain a large number of algorithms, which cannot be explained within the limitations of this article. In the following a brief overview of some important components will be described.

The encapsulated model in fig. 3.1 calculates the solar radiation on surfaces of any orientation. Two versions of that model will be presented now:



Fig. 3.1: calculation of solar radiation on tilted surfaces

The diagram layer of version I is shown in fig. 3.2 (without the cuts on the highest hierarchy level). There are eight important components performing the calculation:

(1) distributes the information about location (*longitude* and *latitude*) and *time zone* to all components that need this data.

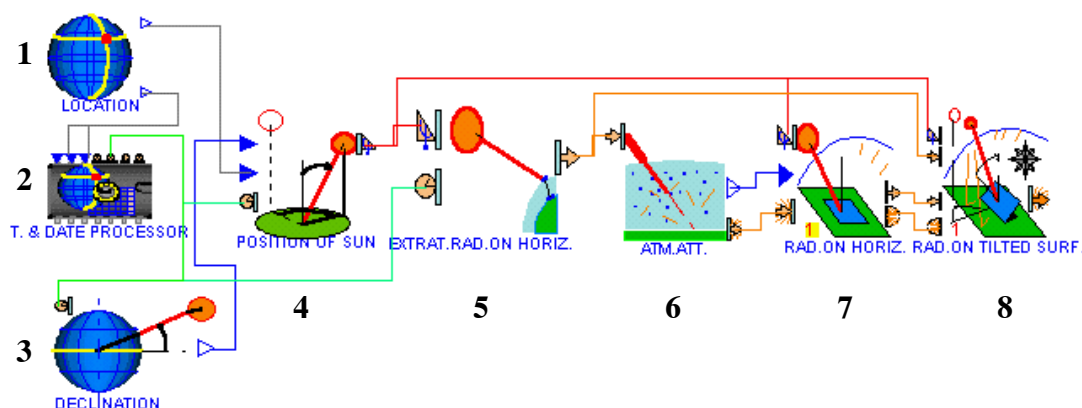


Fig. 3.2: Aggregation pattern of solar radiation model (Version I)

(2) calculates the time variables that are needed in addition to the (physical) simulation time: *solar time* and *standard meridian time*. (Switching to daylight time is possible, too.)

(3) produces the *declination angle*.

(4) calculates the position of the sun determined by *zenith angle* and *azimuth angle*.

(5) calculates the *solar radiation on an extraterrestrial horizontal surface*.

(6) determines the *atmospheric attenuated solar radiation*. For this a so-called *clearness index* k_T is used. The calculation with a constant k_T is an approximation, which should be used only for clear days. (And even in that case k_T is not exactly constant.) In a further model (version II) this component is replaced by importing real or fictive weather data from an external file.

(7) divides the total radiation on a horizontal terrestrial surface into *beam radiation* and *diffuse radiation*.

(8) transforms the results of component (7) into total radiation on a tilted surface.

In version II (fig. 3.3) the disadvantage of component 6 (only suitable for clear days) has been removed by importing radiation data from an external file. The radiation data may be based on measurements of a weather station or a typical climatic conditions of the location. The radiation data (total radiation on a horizontal surface) is imported by a new *ASCII table reader* (component 6A). Table readers from the Modelica standard libraries could not be used for they perform a linear interpolation using a C function. The new table reader makes use of a special C function without linear interpolation. As tables contain sampled values – official weather data for a German test reference year are available at hourly intervals (DIN 4710) – a linear interpolation would produce big mistakes during the hours of sunrise and sunset. To avoid such mistakes the table data has to be read in

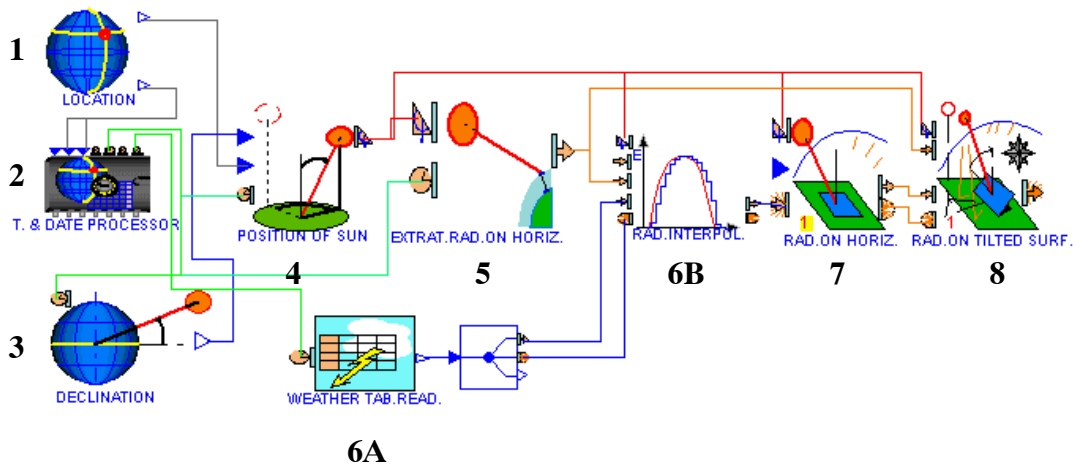


Fig. 3.3: Aggregation pattern of solar radiation model with table reader and interpolator (Version II)

advance. In this way sunrise and sunset are detected in time. Afterwards an appropriate interpolation is performed (component 6B). The library offers a selection of different interpolation methods.

4. Models of Heating Systems

4.1 Introduction

The heating library allows composing models of electrical heating systems as well as hot-water heating systems. In view of physical modelling hot-water heating systems (HWH) are more interesting than the electrical ones, which produce almost immediately a certain heat flow prescribed by the controller. Yet the dynamics of an HWH is determined by transient processes caused by the thermal inertia of various components. Those are the water, the pipes, the radiators and – especially in case of floor heating systems – the stone of the floor. In addition slow transport of water through long pipe systems in large buildings bring about dead time. All those characteristics complicate controlling an HWH for its delayed reactions may have a negative influence on stability.

In the following chapter a dynamic pipe model will be presented.

4.2 Pipes and Radiators

A universal model of a water pipe must combine mechanical and thermal aspects of flowing water. Both aspects are interdependent: The temperature profile depends on the mass flow rate, whereas the mass flow rate is influenced by the viscosity, which is rather strongly depending on temperature. The impact viscosity has on a pipe's flow resistance is determined by the form of flow.

In case of steady laminar flow *Hagen-Poiseuille's Law* is valid for cylindrical pipes:

$$\dot{m} = \frac{\rho \pi D^4}{128 \eta L} (p_1 - p_2) \quad (4.1)$$

(\dot{m} : mass flow rate, D : diameter, L : length of pipe, η : dynamic viscosity, $p_1 - p_2$: pressure drop).

If the Reynolds Number $Re = \rho v D / \eta$ is greater than $Re_{crit.} \approx 2300$ the flow becomes turbulent, and pressure drop is usually approximated by

$$|p_1 - p_2| = \lambda \cdot \frac{L}{D} \cdot \frac{\rho v^2}{2} \quad (4.2)$$

(v : average speed, λ : pipe friction coefficient).

If $Re < 100\,000$, the factor λ is given by

$$\lambda = \frac{0.3164}{\sqrt[4]{Re}} \quad (\text{formula of Blasius}). \quad (4.3)$$

In the pipe model developed here λ is approximated by pieces of straight lines – as well as the viscosity $\eta(T)$.

The thermal dynamic of a fluid within a cylindrical pipe can be described by the partial differential equation (PDG)

$$\underbrace{-c \dot{m} \cdot \left(\frac{\partial T}{\partial x} \right)_t}_{\text{steady release of heat (per meter)}} - \underbrace{c \rho \pi \frac{D^2}{4} \left(\frac{\partial T}{\partial t} \right)_x}_{\text{unsteady release of heat (per meter)}} = \underbrace{\alpha \cdot \pi D \cdot (T - T_{\text{wall}})}_{\text{heat transmitted by convection to pipe wall (per meter)}} \quad (4.4)$$

(c : specific heat capacity of water, ρ : density of water, α : convective heat transfer coefficient),

where heat conduction in x -direction within the water has been neglected. To solve this problem the PDG has to be transformed to a system of ordinary differential equations by a discrete coordinate x : A long pipe is composed out of short pipe elements, each of the length $L := \Delta x$:

$$-cm \cdot \frac{T_{\text{out}} - T_{\text{in}}}{L} - c\rho\pi \frac{D^2}{4} \frac{dT_{\text{out}}}{dt} = \alpha \cdot \pi D \cdot (T_{\text{out}} - T_{\text{wall}}). \quad (4.4')$$

As an approximation of α one might take for instance *Schack's formula* [Rec-97]:

$$\alpha = 3370 \cdot \left(1 + 0.014 \frac{T_{\text{out}}}{^\circ\text{C}}\right) \left(\frac{v}{\text{m/s}}\right)^{0.85} \frac{\text{W}}{\text{m}^2\text{K}} \quad (4.5)$$

$(D = 15 \dots 100 \text{ mm}).$

That formula is, however, not suitable for $v = 0$ as water being at standstill would release no heat at all in such a model.

The equations (4.1) to (4.5) are implemented in model class describing a cylindrical water element (of Length L) flowing through a pipe (fig. 4.1). The cut variables are pressure, temperature and mass flow rate (blue cuts) or temperature and heat flow rate (red cut), respectively.

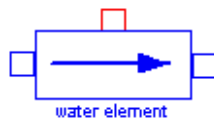


Fig. 4.1: Water element within a pipe

A model describing the conductance and storage of heat within the pipe's wall and heat insulation material is composed out of heat capacitors and cylindrical special heat conductors (fig. 4.2).

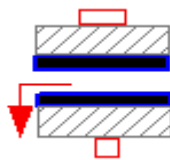


Fig. 4.2: Pipe wall and insulation

With the help of a `for`-loop a series connection of n water elements, each connected to a wall and insulation component, is generated. The complete pipe model (fig. 4.3) can be connected to a temperature source or another (big) heat capacitor (e.g. a wall or a floor in which the pipe is embedded), which absorbs steady heat loss.

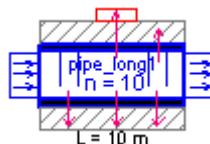


Fig. 4.3: Long water pipe with insulation

Fig. 4.4 shows a circuit with a 20 m-pipe (consisting of 20 pipe elements, i.e. $n = 20$), a pump producing a constant pressure and a boiler switched on at $t = 600\text{s}$. At $t = 0$ all components have the temperature $T_0 = 10^\circ\text{C}$. The boiler has a two-state controller with a

hysteresis between 90°C and 95°C . The results of the *DYMOLA* simulation are shown in diagram 4.1.

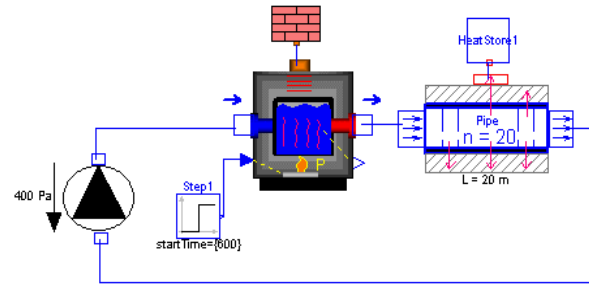


Fig. 4.4: Circuit with long pipe, boiler and pump

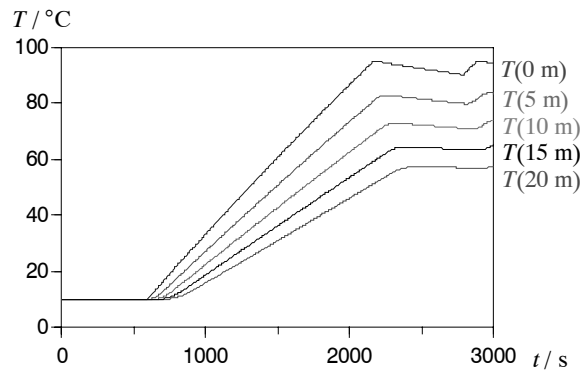


Diagram 4.1: Results of *DYMOLA* simulation of model in fig. 4.4: water temperatures at different positions in the pipe

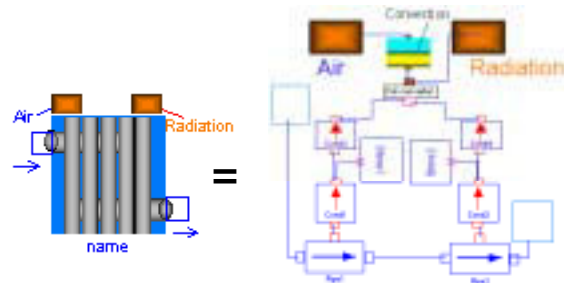


Fig. 4.5: Hot-water radiator

By analogy with the pipe model a *radiator model* can be designed by leaving out the heat insulation (fig. 4.5). The radiator can be connected to the *two-star room* presented in ch. 2 via two cuts: one cut for convective heat transmission to air and another cut for a connection to the room's radiation nodal point.

5.1 Standard Controllers

The first part of the controller library contains several components simulating standard control algorithms. Some components, however, have extended functions. There are two sublibraries, one with continuous and one with discrete controllers.

The *continuous controller sublibrary* contains PI and PID controllers with an anti-wind-up reset function and a pulse width modulator.

The *discrete controller sublibrary* contains special PI and PID controllers emulating the signal processing of digital controllers. In this way the influence of important quantities such as the number of bits or the sampling period can be included in the simulation.

5.2 Fuzzy Control

The Heating, air-conditioning and ventilation of a building requires the control of many interdependent quantities belonging to rather complex physical processes. Therefore, Fuzzy Control is an appropriate alternative to standard control strategies. With the new *Modelica Fuzzy Control library* to design a Fuzzy Controller and to test different rule bases,

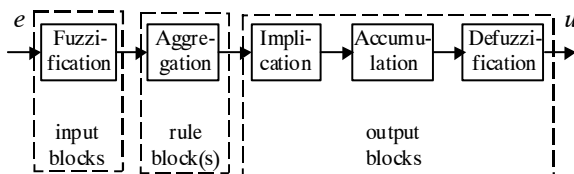


Fig. 5.1: The five steps of FC processing and their assignment to input, output and rule blocks

methods etc. on a Modelica model.

The Fuzzy Control library has a modular structure. A Modelica Fuzzy Controller (FC) has to be composed by the user out of special blocks for the linguistic input and output variables and for the rules. Due to

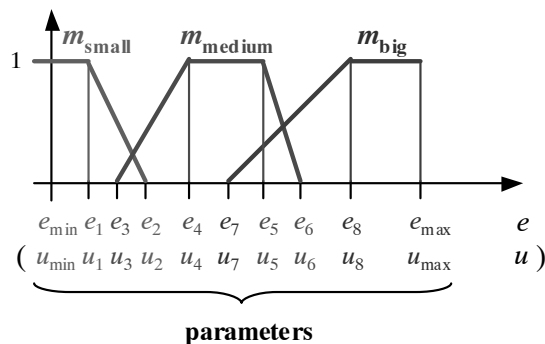


Fig. 5.2: Predefined fuzzy sets for input or output variable (e/u) with parameters of corresponding input or output block

the large variety of settings defining an FC (numbers of inputs and outputs, fuzzy sets, rules, methods of inference and defuzzification) a modular structure supports the clarity of the FC model. Fig. 5.1 shows the assignment of the FC-processing steps to instances of different model classes: input blocks, output blocks and rule blocks.

The fuzzy sets for each input variable and each output variable are defined by choosing an input or output block, respectively, and entering the blip abscissas of the predefined membership functions (fig. 5.2). Currently there are input and output blocks with three or five membership functions. The linguistic values have predefined names.

As fuzzy implication, accumulation and defuzzification are performed in the output block of each linguistic output variable, the library contains different output blocks according to the methods used for implication, accumulation and defuzzification and according to the number of fuzzy sets.

Two different versions of the FC library have been developed so far. In the first place the two versions vary in the way the rules are formulated. Fig. 5.3 shows the *Modelica* diagram layer of an FC according-

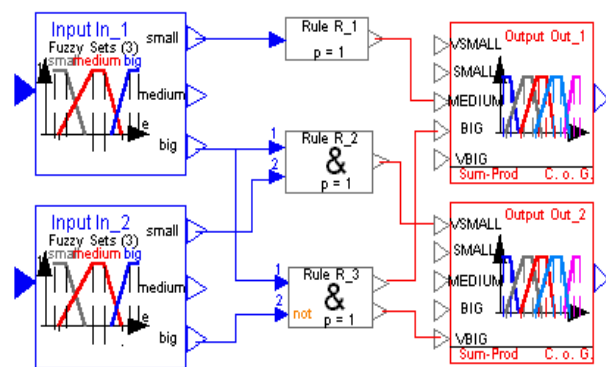


Fig. 5.3: FC according to Version I with the rules:

1. If In_1 small, then Out_1 medium.
2. If In_1 big and In_2 small, then Out_2 very small.
3. If In_1 big and In_2 not big, then Out_1 big and Out_2 very big.

ing to Version I of the library. The methods of implication, accumulation and defuzzification are *sum/product/centre of gravity*. The diagram is similar to a circuit diagram of Boolean logic: There is a rule block for each rule that has to be connected to the input and output blocks. This means that the rules are visualised by the connections the user must draw. There are various types of rule blocks in the library according to the numbers of input values and output values appearing in a rule. Each input value can be negated by entering a certain parameter into the rule block concerned.

Fig. 5.4 is the diagram layer of the same FC composed in Version II: In this case, only one central rule block is needed. The rules are entered into its parameter table as text using a special syntax. The rule block is prepared for up to ten input variables, five output variables and fifty rules. It interprets the aggregation by comparing the components of each rule vector with the predefined linguistic values "vsmall", "small", "medium", "big", "vbig" and "nvsmall",

“nsmall”, “nmedium”, “nbig”, “nvbig” (v = “very”, n = “not”, output values in capital letters for better readability).

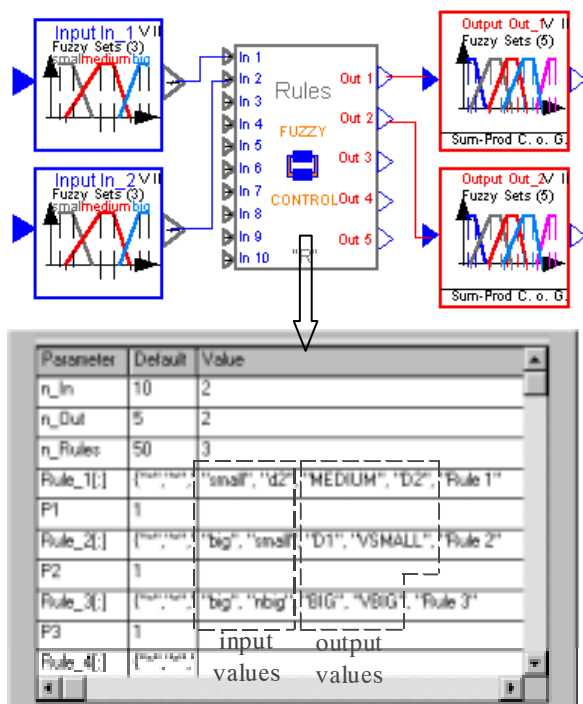


Fig. 5.4: FC in Version II
(corresponding to FC in fig. 5.3)

6. References

- [Duf-74]
Duffie J.A., Beckman W.A.: Solar Energy thermal process, Wiley, New York (1974)
- [Fei-94]
Feist W.: Thermische Gebäudesimulation Kritische Prüfung unterschiedlicher Modellansätze, Verlag C.F. Müller, Heidelberg (1994) pp.135
- [Fel-01]
Felgner F.; Merz R.: Thermohydraulische Simulationsmodelle für Heizungsanlagen ASIM 2001, 15.Sym. Sim.tech. Paderborn 2001
- [Kie-01]
Kienzlen K. und da Silva P.: Das Haus im Entwicklungslabor, TAB (Technik am Bau), Bertelsmann Fachzeitschriften GmbH, Gütersloh, 10 (2001) p. 35-39
- [Kle-00]
Klein, S.A., Duffie, J.A., Mitchell, J.C., Kummer, J.P., Thornton, J.W., Beckman W.A., Duffie, N.A., Braun J.E., Urban, R.R., Blair, N.J., Mitchell, J.W., Freeman T.L., Evans B.L., Fiksel A.: TRNSYS, a transient system simulation program, Solar Energy Laboratory University of Wisconsin, Madison 1996

[Mer-01b]

Merz R., Litz, L.: Objektorientierte mathematische Modellbildung zur Simulation thermischen Gebäudeverhaltens, ASIM 2001, 15.Sym. Sim.tech. Paderborn 2001

[Rec-97]

Recknagel, Sprenger, Schramek: Taschenbuch für Heizungs- und Klimatechnik. 68.Auflage. Oldenbourg Verlag. München, Wien, 1997.

[Sit-01]

Sitompul E., Merz R.: Anwendung objektorientierter Entwurfsmethoden zur generischen Entwicklung von Regelungsalgorithmen in der Anwendungsdomäne Gebäudeautomation ASIM 2001, 15.Sym. Sim.tech. Paderborn 2001

[Spr-01]

Sprengard C. Merz R.: Simulation des energetischen und thermischen Verhaltens eines Niedrigenergiehauses mit dem Gebäudesimulationsprogramm TRNSYS ASIM 2001, 15.Sym. Sim.tech. Paderborn 2001

[Trn-02]

TRNSYS, www.transsolar.de, Homepage der Fa. Transsolar Energietechnik GmbH. Stuttgart 2001

[VDI-01]

VDI Richtlinie 6020: Anforderung an Rechenverfahren zur Gebäude- und Anlagensimulation, VDI, Düsseldorf, Beuth Verlag GmbH, Berlin (2001)

Session 7

Poster session

Methods for Structural Analysis and Debugging of Modelica Models

Peter Bunus, Peter Fritzson

PELAB, Programming Environment Laboratory, Department of Computer and Information Science, Linköping University, SE-581 83, Linköping, Sweden
{petbu,petfr}@ida.liu.se

Abstract

A significant part of the simulation design effort is spent on detecting deviations from the specifications and subsequently localizing the source of errors. Employment of debugging environments that control the correctness of the developed source code is an important factor in reducing the time and cost of software development in classical programming languages. Currently, few or no tools are available to assist developers when debugging declarative equation based modeling languages. To begin to address this need we have developed an efficient debugging framework for Modelica and have adapted traditional debugging techniques and algorithms to it. The developed algorithms and methods help to statically detect and repair a broad range of errors without having to execute the simulation model. Several simulation models and examples are given in this paper in order to illustrate the main situations when over and under-constraining equations can appear in the system. Error detection and error solving strategies for those cases are also given.

1 Introduction

Obviously, each simulation problem is associated with a corresponding mathematical model. In dynamic continuous simulation the mathematical model is usually represented by a mixed set of algebraic equations and ordinary differential equations. A typical problem which appear in physical system modeling and simulation is when too many (or few) equations are specified in the system inevitably leading to an inconsistent state of the simulation model. In such situations numerical solvers fail to produce correct solutions to the underlying system of equations.

For example, a physical system simulation model specified in a declarative object-oriented equation based modeling language may consist of several hundreds of classes resulting in over 100 000 flattened equations. If one of these equations over-constrains the overall system it cannot be simulated. It can be easily imagined that, if a subset of six over-constraining equations can be provided by a static debugger from where the user can choose one equation to eliminate, in order to form a structurally well posed simulation problem, it would be

extremely useful. This could greatly reduce the amount of time required to get the simulation working.

Our goal is to contribute to the methodology of algorithmic debugging and automated debugging of object-oriented equation based modeling languages and to develop programming environments to support it. Although, what we present in this paper applies to the whole area of equation based debugging, our primary target is debugging of Modelica models and more specifically static analysis techniques for diagnosability of physical system models specified with Modelica.

The simulation models presented in this paper are so trivial as to be almost beneath consideration, but they serves as a straightforward vehicle for the introduction of several fundamental debugging concepts with the purpose of illustrating concepts of structural analysis. These models are extremely useful from that point of view because they keep the associated structural graphs to a minimum size and complexity, but in the meantime exposing interesting structural and debugging problems.

This paper is organized as follows: Section 2 provides graph theoretical preliminaries necessary to understand the algorithms used in this paper together with the canonical decomposition algorithm. In Section 3 simple over constrained simulation models are diagnosed and debugged with the help of graph decomposition techniques and our algorithmic debugging approach. Details are briefly presented about the structures used to annotate the underlying equations of the simulation model, in order to help the debugger to eliminate the heuristics when multiple choices are available to fix an error. In Section 4 the particulars of debugging an under-constrained systems are given. Implementation details of the debugger are given in Section 5. Finally, Section 6 concludes and summarizes the work.

2 Preliminaries

Many practical problems are examples of a model of interaction between two different types of objects and can be phrased in terms of problems on bipartite graphs. The expressiveness of the bipartite graphs in concrete practical applications has been demonstrated many times in the literature (Dolan and Aldous [2]), (Asratian et al. [1]). We will show that the bipartite graph representations are general enough to efficiently accommodate several numeric analysis methods in order to reason about the solvability and unsolvability of the

flattened system of equations and implicitly about the simulation model behavior. Another advantage of using the bipartite graphs is that it offers an efficient abstraction necessary for program transformation visualization when the equation based declarative specifications are translated to procedural form.

The bipartite graph representation and the associated decomposition techniques are widely used internally by compilers when generating the procedural form from the declarative equation based description of the simulation model (Elmqvist [4]) (Maffezzoni et. al. [9]) but none of the existing simulation systems use them for debugging purposes or expose them visually for program understanding purposes.

Definition 1: A bipartite graph is an ordered triple $G = (V_1, V_2, E)$ such that V_1 and V_2 are sets, $V_1 \cap V_2 = \emptyset$ and $E \subseteq \{\{x, y\}; x \in V_1, y \in V_2\}$. The vertices of G are elements of $V_1 \cup V_2$. The edges of G are elements of E .

Definition 2: A *matching* is a set of edges from graph G where no two edges have a common end vertex.

Definition 3: A matching M of a graph G is *maximal* if it is not properly contained in any other matching.

Definition 4: A vertex v is *saturated* or *covered* by a matching M if some edge of M is incident with v . An unsaturated vertex is called a *free vertex*.

Definition 5: A *perfect matching* P is a matching in a graph G that covers all its vertices.

Definition 6: A path $P = \{v_0, v_1, \dots, v_k\}$ in a graph G is called an *alternating path* of M if it contains alternating free and covered edges.

In Figure 1 all the possible perfect matchings of a simple bipartite graph are presented. It should be noted that a maximum matching and the perfect matching of a given bipartite graph is not unique.

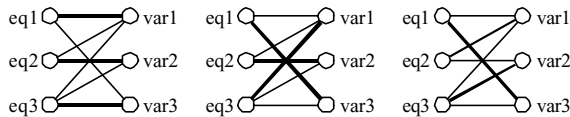


Figure 1. An example bipartite graph with all the possible perfect matchings marked by thick lines.

A structural decomposition of a bipartite graph associated with a simulation model which relies on the above presented vertex coverings is due to (Dulmage and Mendelsohn [3]) and canonically decomposes any maximum matching of a bipartite graph in three distinct parts: *over-constrained*, *under-constrained*, and *well-constrained* part.

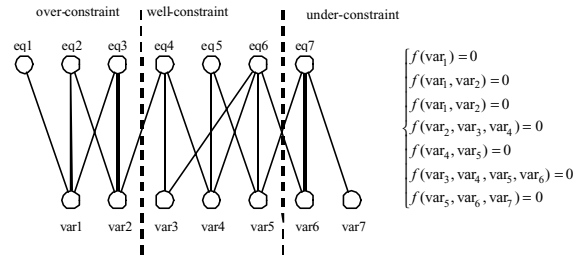


Figure 2. Dulmage Mendelsohn's canonical decomposition of a bipartite graph.

The canonical decomposition algorithm is given below:

Algorithm: Dulmage and Mendelsohn canonical decomposition

Input Data: A bipartite graph G

Result: Three subgraphs: well-constrained G_w , over-constrained G_o and under-constrained G_u .

begin:

- Compute the maximum matching M of G .
- Compute the directed graph D by replacing each edge in M by two arcs and orienting all other edges from the equations to the variables.
- Let be U the set of all descendants of sources of the directed graph D .
- Let be W the set of all ancestors of sink of the directed graph D .
- Calculate $G_o = G - U - W$.

end.

The *over-constrained* part: the number of equations in the system is greater than the number of variables. The additional equations are either redundant or contradictory and thus yield no solution.

The *under-constrained* part: the number of variables in the system is greater than the number of equations. A possible error fixing strategy would be to initialize some of the variables in order to obtain a well-constrained part or add additional equations to the system.

Over and under-constrained situations can coexist in the same model. In the case of over-constrained model, the user would like to remove the over-constraining equations in a manner which is consistent to the original source code specifications, in order to alleviate the model definition.

The *well-constrained* part: the number of equations in the system is equal to the number of variables and therefore the mathematical system of equations is structurally sound having a finite number of solutions. This part can be further decomposed into smaller solution subsets. A failure in decomposing the well-constrained part into smaller subsets means that this part cannot be decomposed and has to be solved as it is. A failure in numerically solving the well-constrained part means that no valid solution exists and there is somewhere a numerical redundancy in the system.

3 Debugging of Over-Constrained Models

A typical problem which appears in physical system modeling and simulation is when too many equations are specified in the system inevitably leading to an inconsistent state of the simulation model. In such situations numerical solvers fail to compute correct

In Figure 3 the Modelica source code of a simple simulation model consisting of a resistor connected in parallel to a sinusoidal voltage is given. The intermediate form is also given for explanatory purposes. The Circuit model is represented as an aggregation of the Resistor, Source and Ground model instances, R1, AC and G connected together by means of physical ports.

<pre> connector Pin Voltage v; Flow Current i; end Pin; model TwoPin Pin p, n; Voltage v; Current i; equation v = p.v - n.v; 0 = p.i + n.i; i = p.i end TwoPin; model Resistor extends TwoPin; parameter Real R; equation R*i == v; end Resistor; model VsourceAC extends TwoPin; parameter Real VA=220; parameter Real f=50; protected constant Real PI=3.141592; equation v=VA*(sin(2*PI*f*time)); end VsourceAC; model Ground Pin p; equation p.v == 0 end Ground; model Circuit Resistor R1(R=10); VsourceAC AC; Ground G; equation connect(AC.p,R1.p); connect(R1.n,AC.n); connect(AC.n,G.p); end Circuit; </pre>	<pre> Flat equations 1. R1.v == -R1.n.v + R1.p.v 2. 0 == R1.n.i + R1.p.i 3. R1.i == R1.p.i 4. R1.i*R1.R == R1.v 5. AC.v == -AC.n.v + AC.p.v 5. 0 == AC.n.i + AC.p.i 7. AC.i == AC.p.i 8. AC.v == AC.VA*Sin[2*time*AC.f*AC.PI] 9. G.p.v == 0 10. AC.p.v == R1.p.v 11. AC.p.i + R1.p.i == 0 12. R1.n.v == AC.n.v 13. AC.n.v == G.p.v 14. AC.n.i + G.p.i + R1.n.i == 0 Flat Variables 1. R1.p.v 2. R1.p.i 3. R1.n.v 4. R1.n.i 5. R1.v 6. R1.i 7. AC.p.v 8. AC.p.i 9. AC.n.v 10. AC.n.i 11. AC.v 12. AC.i 13. G.p.v 14. G.p.i Flat Parameters R1.R -> 10 AC.VA -> 220 AC.f -> 50 Flat Constants AC.PI -> 3.14159 </pre>
---	--

solutions to the underlying system of equations.

Figure 3. Modelica code of a simple electrical circuit and the associated flattened equations

During the first stage of the static analysis the associated bipartite graph of the intermediate flattened form of the equations is constructed and the maximum cardinality matching is computed as it is shown in Figure 4.

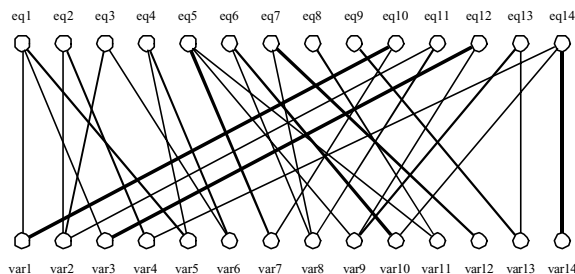


Figure 4. Associated bipartite graph and the corresponding perfect matching (thicker lines) to the simple electrical circuit.

It's worth noting, that in this case, the maximal matching is also a perfect matching of the associated bipartite graph. In this case all the vertices are covered

by a matching and the canonical decomposition algorithm will yield to only one well-constrained component without any under or over-constraining part. The well-constrained part can be safely sent to the numerical solver and the simulation can be successfully performed if no other numerical redundancies are present in the system of equations..

Let us now consider the same electrical circuit where an additional equation ($i=23$) was intentionally introduced inside the Resistor component in order to obtain a generally over-constrained system. The D&M canonical decomposition will lead to two parts: a well-constrained part and an over-constrained part (see Figure 5). Equation “*eq11*” is a non-saturated vertex of the equation set so it is a source for the over-constrained part. Starting from “*eq11*” which is the non-saturated vertex, the directed graph can be derived from the undirected bipartite graph as is illustrated in Figure 6. by exchanging all the matching edges in bi-directional edges and orienting all other edges from equations to variables. An immediate solution of fixing the over-constrained part is to eliminate “*eq11*” which will lead

to a well-constrained part and therefore the equation system becomes structurally sound.

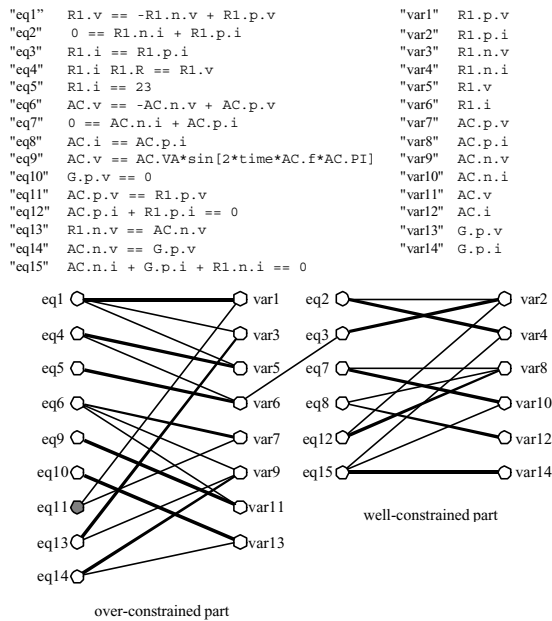


Figure 5. Canonical decomposition of the over-constraining system.

However, examining the equation “eq11” one can note that the equation is generated by a `connect` statement from the `Circuit` model, and the only way to remove the equation is to remove the `connect` (`AC.p`, `R1.p`) statement. But removing the above-mentioned statement will remove two equations from the flattened model which is unacceptable.

In order to support an automatic reasoning about the equations the flattened equations from the intermediate code are annotated by a structure which resembles the one presented in Table 1.

Table 1. The structure of the annotated equation

Attribute	Value
Equation	$R1.i * R1.R == R1.v$
Name	“eq4”
Description	“Ohm’s Law for the resistor component”
Nr. of associated eq	1
Class Name	“Resistor”
Flexibility Level	3
Connector generated	no

The *Class Name* tells from which class the equation is coming. This annotation is extremely useful in exactly locating the associated class of the equation and therefore providing concise error messages to the user.

The *No. of associated eqs.* parameter specifies the number of equations which are specified together with the annotated equation. In the above example the *No. of associated eqs.* is equal to one since there are no additional equations specified in the `Resistor` component. In the case of the `TwoPin` component the number of associated equations is equal to 3. If one

associated equation of the component need to be eliminated the value is decremented by 1. If, for example, during debugging, the equation $R1.i * R1.R == R1.v$ is diagnosed to be an over-constraining equation and therefore need to be eliminated, the elimination is not possible because the model will be invalidated in that way (the *No. of associated eqs.* cannot be equal to 0) and therefore other solutions need to be taken into account.

The *flexibility level*, in a similar way as it is defined in (Flannery and Gonzales [5]), allows the ranking of the relative importance of the constraint in the overall flattened system of equations

The *Connector generated* is a Boolean attribute which tells whether the equation is generated or not by a `connect` statement. Usually these equations have a very low flexibility level.

It is worth noting that the annotation attributes are automatically initialized by the static analyzer, incorporated in the front end of the compiler, by using several graph representations.

Having the equations annotated, the next step is to traverse the associated directed graph, shown in Figure 6, to the over-constraining part, obtained from the D&M decomposition.

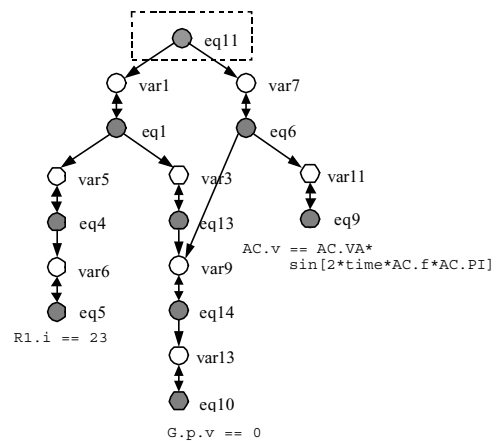


Figure 6. The associated directed graph of the over-constraining subgraph.

One important property of the over-constrained bipartite graph is that it only contains alternating paths because it is constructed from perfect matchings and a supplementary free edge. We can easily obtain all the maximal matchings in the over-constrained graph by exchanging matching edges with other edges along an alternating path. Therefore eliminating any of the constitutive nodes that represent an equation we can easily find a corresponding matching of the sub-graph will yield to a well-constrained subsystem. But eliminating some of the constitutive nodes that represent equations will disconnect the sub-graph as it is illustrated in Figure 7 where *eq1* was eliminated. Even if this situation, when two disconnected graphs are obtained, are mathematically sound they are not very common from the modeling point of view and therefore they are not further considered.

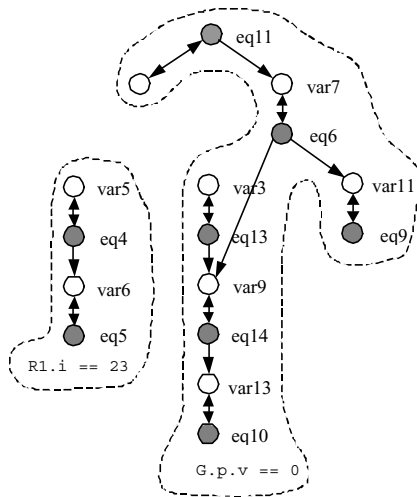


Figure 7. Disconnected graph obtained by eliminating *eq1*.

In our case the set of equivalent over-constraining equations is {"eq11", "eq13", "eq10", "eq5", "eq9"} after eliminating those equations which disconnect the bipartite graph. "eq11" was already analyzed and therefore can be eliminated from the set. "eq13" is eliminated too for the same reasons as equation "eq11". Analyzing the remaining equations {"eq10", "eq5", "eq9"} one should note that they have the same flexibility level and therefore they are candidates for elimination with an equal chance. But analyzing the value of the *No. of associated eqs.* parameter, equation "eq10" and "eq9" have that attribute equal to one, which means that they are singular equations defined inside the model. Eliminating one of these equations will invalidate the corresponding model, which is probably not the intention of the modeler.

Examining the annotations corresponding to equation "eq5" one can see that it can be safely eliminated because the flexibility level is high and eliminating the equation will not invalidate the model since there is another equation defined inside the model. After choosing the right equation for elimination the debugger tries to identify the associated class of that equation based on the *Class name* parameter defined in the annotation structure. Having the class name and the intermediate equation form ($R1.i=23$) the original equation can be reconstructed ($i=23$) indicating exactly to the user which equation needs to be removed in order to make the simulation model mathematically sound. In that case the debugger correctly locates the faulty equation previously introduced by us in the simulation model.

We now construct a simple electrical circuit model (Figure 8) by connecting two resistors in parallel with a voltage source as is shown in Figure 9. The Modelica definition of the Ground, *VsourceAC* and *Resistor* component are reused from the previous examples. The *TwoPin* class is modified by introducing an additional over-constraining equation ($i=10$) in the model definition. This extra equation will be inherited

by all the classes which extends the *TwoPin* class. Therefore each instance of the *Resistor* and *VsourceAC* models will contribute to one extra over-constraining equation to the final flattened system of equations.

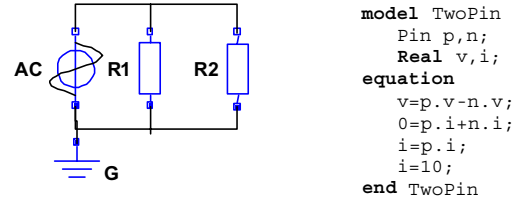


Figure 8. An electrical circuit with an over-constraining equation in the *TwoPin* component.

During the model translation the corresponding flattened set of equations from the simulation model is derived and the associated bipartite graph G is constructed. The overall flattened model corresponding to the simple electrical circuit contains three extra over-constraining equations (*eq9*, *eq18*, *eq7*). Therefore three vertices from the equations sets are not covered by a matching, as it is illustrated in the derived directed graph in Figure 9.

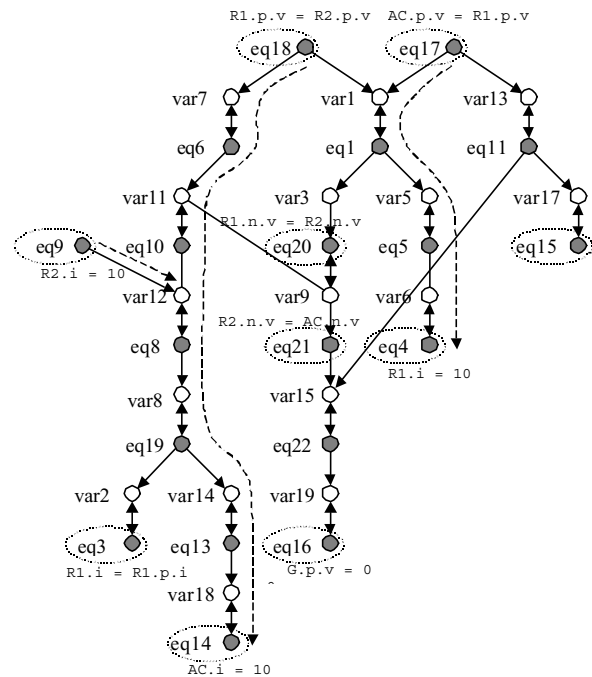


Figure 9. The over-constrained directed graph

While traversing the directed graph, after eliminating all the equations that disconnect the graph and performing reasoning based on the equation annotations we found that the equations *eq9*, *eq14*, *eq15* need to be eliminated from the intermediate form which are generated from the equation $i=10$ in the *TwoPin* partial component. The elimination of *eq9*, *eq14* and *eq15* is safe because they can be made free vertices by exchanging the matching edges with non matching edges along the paths indicated with dashed lines in Figure 9.

4 Debugging of Under-Constrained Models

The issue of under-constrained simulation models considered in an object-oriented declarative equation-based frameworks, has been discussed in (Ramirez [10]). The work presented in (Ramirez [10]) is particularly concerned with the issues involved in the modeling and solutions of conditional models where the system of equations in the model is different for each of the alternatives.

Let us consider the number of equations m from a model and the number of variables incident in those equations n . For a typical under-constrained situation the number of variables is greater than the number of equations ($n > m$).

Definition 7: We call the *degree of under-constraining* the difference between the number of variables and the number of equations $D_u = n - m$. In a similar way in (Ramirez [1]) D_u is called the number of degrees of freedom of the problem.

In the following we are going to illustrate the possible error fixing solutions for a typical under-constraining situation and the reasoning involved in the graph transformation system. Let us consider the following system of equations with the corresponding bipartite graph presented in Figure 11 and with the degree of under-constraining $D_u = 1$.

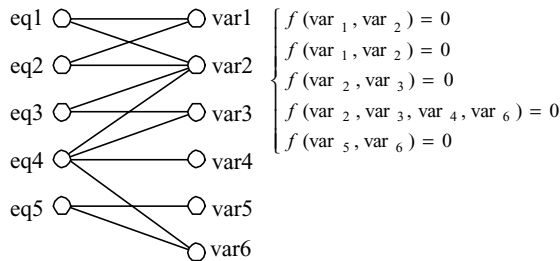


Figure 10. A simple system of equations with the associated bipartite graph.

One possible corresponding maximal matching (represented by thicker edges) to the bipartite graph and the D&M canonical decomposition is presented below:

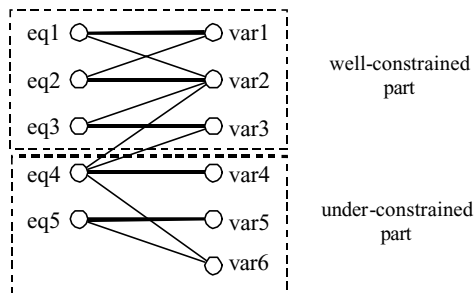


Figure 11. Maximum matching and canonical decomposition of the bipartite graph

In performing the canonical decomposition algorithm the associated directed graph to the bipartite graph was constructed by exchanging all the edges which are part of the maximal matching by bi-directional edges and orienting all other edges from equations to variables. The obtained directed graph is shown in Figure 12:

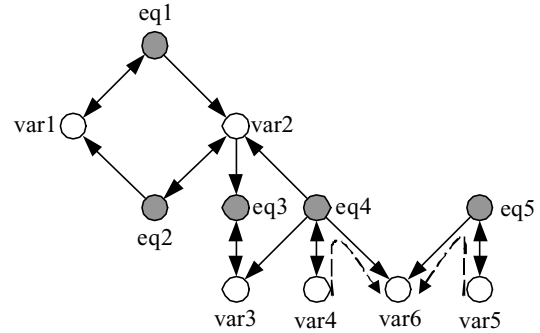


Figure 12. Directed graph associated with the system of equations.

The variables contained in an under-constrained part constitutes the eligibility set. In our small example the eligibility set is $\{var4, var5, var6\}$ which means that any of these variable can be taken away and the remaining associated graph will be well constrained.

Variable $var6$ is not covered by the maximal matching and therefore is a free vertex. In the directed graph, it can be seen that these are two alternating paths to the free vertex $var6$ (indicated by the dashed arrows in Figure 12):

$\{(var4, eq4), (eq4, var6)\}$ and $\{(var5, eq5), (eq5, var6)\}$.

Exchanging the matching edges with normal edges and the normal edges with matching edges along an alternating path a new matching can be obtained which cover the free vertex $var6$ but will uncover another vertex from the eligible set. Therefore for an error fixing strategy all the possible combinations should be taken into account.

During the first stage of the error fixing process only those solutions which involve the elimination of a variable from the eligibility set are taken into account. We have the following possible solutions illustrated in Figure 13.

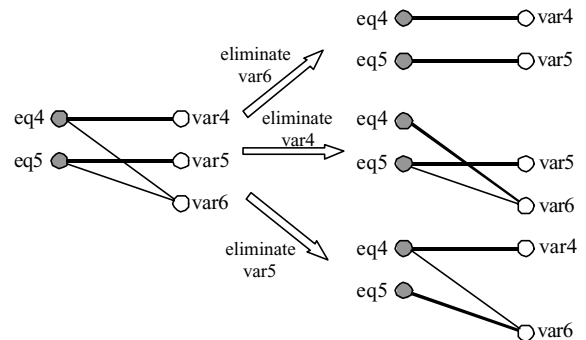


Figure 13. Error fixing solution when one variable is taken away from the eligibility set.

By removing *var6* from the under-constrained sub system the considered maximum matching becomes a perfect matching of the associated bipartite graph and therefore the associated system of equations is structurally sound. However, by removing *var6* the bipartite graph will be disconnected and an independent edge (*eq5, var5*) appears in the system, which is not connected to the main bipartite graph. This situation is extremely unusual in physical system modeling and it means that some variables are computed locally inside a component without contributing to the general behavior of the simulated system. As an example the following Modelica Resistor component integrated in a circuit model will produce two disconnected sub-graphs.

```

model Resistor
  extends TwoPin;
  parameter Real R;
  Real s;
equation
  R*i=v;
  s=10;
end Resistor

```

The variable *s* and the equation *s=10* are redundant in the system. Therefore the situation when an extra variable is eliminated and the remaining bipartite graph is disconnected needs to be further analyzed. In our case, for example, a solution which involves the elimination of variable *var6* and the presence of an extra variable *var1*, *var2*, *var3* or *var4* in equation *eq5* might be acceptable.

It should also be noted that multiple error fixing strategies are possible in the case of an under-constrained subsystem. Another error fixing situation for the under-constrained systems is to add one extra equation to the system and link the free variable to the added equation instead of eliminating the free variable. This strategy applied for the free variable *var6* is presented in Figure 14.

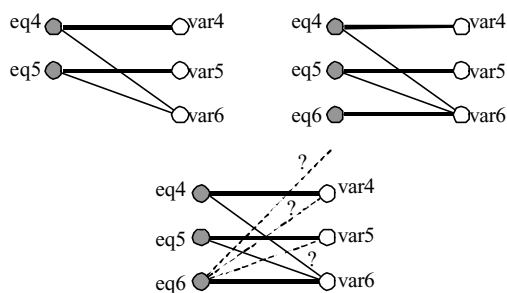


Figure 14. Error fixing strategy involving adding an extra equation.

This strategy involves two steps: at the first step an extra equation is added and linked together with the free variable and then at the second step is checked if other variables from the system might be present in the recently added equation. This last step turns out to be very useful from the users point of view because is

helps them to reconstruct missing equations from simulation models.

Let us again analyze the simple circuit model when the Resistor component is changed again by declaring an extra variable (Real *s*) and introducing this variable into the Resistor component constitutive equation.

```

model Resistor
  extends TwoPin;
  parameter Real R;
  Real s;
equation
  R*i=v*s;
end Resistor

```

The directed graph obtained from the associated bipartite graph of the flattened underlying system of equations and a corresponding maximum cardinality matching, is given below:

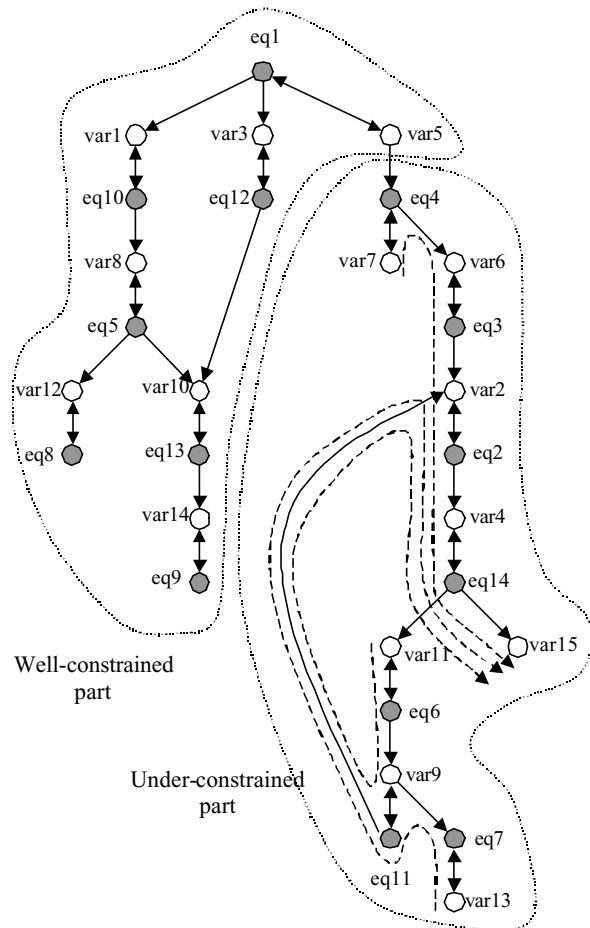


Figure 15. Directed graph corresponding to the under-constrained simple electrical circuit.

The uncovered variable by the considered maximum cardinality matching is *var15*, the eligibility set being:

{*var15*, *var4*, *var2*, *var6*, *var7*, *var11*, *var9*, *var13*}

with the corresponding variables:

{G.p.i, R.n.i, R.p.i, R.i, R.s, AC.n.i,
AC.p.i, AC.i}

From the under-constrained subgraph we can derive the following alternating paths (indicated in Figure 15 by the dashed arrows) to the uncovered variable:

$\{(\overline{\text{var}_{15}, \text{eq}_{14}}, \overline{\text{eq}_{14}, \text{var}_4}), (\overline{\text{var}_4, \text{eq}_2}, \overline{\text{eq}_2, \text{var}_2})$
 $(\overline{\text{var}_2, \text{eq}_3}, \overline{\text{eq}_3, \text{var}_6}), (\overline{\text{var}_6, \text{eq}_4}, \overline{\text{eq}_4, \text{var}_7})\}$

$\{(\overline{\text{var}_{15}, \text{eq}_{14}}, \overline{\text{eq}_{14}, \text{var}_4}), (\overline{\text{var}_4, \text{eq}_2}, \overline{\text{eq}_2, \text{var}_2})$
 $(\overline{\text{var}_2, \text{eq}_{11}}, \overline{\text{eq}_{11}, \text{var}_9}), (\overline{\text{var}_9, \text{eq}_6}, \overline{\text{eq}_6, \text{var}_{11}})\}$

$\{(\overline{\text{var}_{15}, \text{eq}_{14}}, \overline{\text{eq}_{14}, \text{var}_4}), (\overline{\text{var}_4, \text{eq}_2}, \overline{\text{eq}_2, \text{var}_2})$
 $(\overline{\text{var}_2, \text{eq}_{11}}, \overline{\text{eq}_{11}, \text{var}_9}), (\overline{\text{var}_9, \text{eq}_7}, \overline{\text{eq}_7, \text{var}_{13}})\}$

By following each alternating path and eliminating the variables from the eligibility set it can be noticed that eliminating only the variables

$\{\text{var}_{15}, \text{var}_4, \text{var}_7, \text{var}_{13}\}$ will not disconnect the bipartite graph. Therefore only this reduced set will be further analyzed at this stage. Based on a similar reasoning as in the over-constrained situations and on variable associated annotations, the results is that only *var7* can be safely removed from the Modelica code in order to obtain a well specified underlying equation system. We call the set of variables obtained after performing the reasoning based on annotations *the reduced eligibility set*

In the above presented situation the fault was detected during the first stage of the debugging of under-constrained equations. But if the user is not satisfied with the given solution or the reduced eligibility set is empty, the debugger can enter in the second stage when possible connections of the adjacent equations to those variables that disconnect the bipartite graph are checked. If a possible coupling of a variable to those equations is found the adjacent disconnecting variable might be also considered for elimination. The possible coupling of variables with equations is performed by a *variable reachability analysis* based algorithms applied to the inheritance graph of the underlying simulation system. The variable reachability analysis computes the list of variables which can be inserted into certain equations. The description of the variable reachability analysis algorithm is not the subject of this paper.

A third stage in the debugging process of the under-constraining equations is when extra equations need to be added and coupled to the free equation. For example, in our case, adding an extra equation $s=10$ in the Resistor component is a mathematically sound solution even it might not reflect the modelers intent. In a similar way extra equations can be added to each variable from the eligibility set.

The user has the possibility of specifying which level of debugging he/she would like to perform on the erroneous model, in that way, filtering out some of error messages and performing an incremental error fixing on the modeling source code.

5 Prototype Implementation

A prototype debugger has been built and attached to the *MathModelica* simulation environment as a testbed for evaluating the usability of the above presented graph decomposition techniques for debugging declarative equation based languages. *MathModelica* is an integrated problem-solving environment (PSE) for full system modeling and simulation (Fritzson et. al.[6]) (Jirstrand [7]) (Jirstrand et. al.[8]). The environment integrates Modelica-based modeling and simulation with graphic design, advanced scripting facilities, integration of code and documentation, and symbolic formula manipulation provided via *Mathematica* (Wolfram [11]). Import and export of Modelica code between internal structured and external textual representation is supported by *MathModelica*. The environment extensively supports the principles of literate programming and integrates most activities needed in simulation design: modeling, documentation, symbolic processing, and transformation and formula manipulation, input and output data visualization.

In order to attach the debugger it was absolutely necessary to have access to the intermediate form of the code because the presented algorithm makes use of the intermediate flat form of the equations. The implemented debugger was successfully tested on Modelica models involving several hundreds of algebraic and differential algebraic equations.

The general architecture of the implemented debugger is presented in Figure 16. The debugging algorithm proceeds as follows: based on the original declarative source code the intermediate representation is generated. From the intermediate representation the overall system of equations is extracted and transformed into bipartite graph form. The associated bipartite graph is canonically decomposed. Error-fixing strategies are applied if the decomposition leads to over- or under-constrained components. The debugger will try to solve the errors automatically without explicit intervention of the user. If automatic error solving is not possible due to missing information the user will be consulted regarding the repair strategy.

When the user is interrogated, all valid options that will lead to a structurally sound underlying system of equations are presented. As was mentioned earlier, the error fixing strategies for over- and under-constrained subcomponents might involve several stages, especially for under-constrained situations. Due to the equation and variable annotations the error messages output by the debugger are understandable relative to the user perception of the simulation source code, in our case the Modelica code. The information output by the debugger will of course lead to a mathematically sound system of equations. However, some of the solutions might not be acceptable from the modeling language point of view or from the physical system model perspective. The debugger focuses on those errors whose identification would not require the solution of the underlying system of equations.

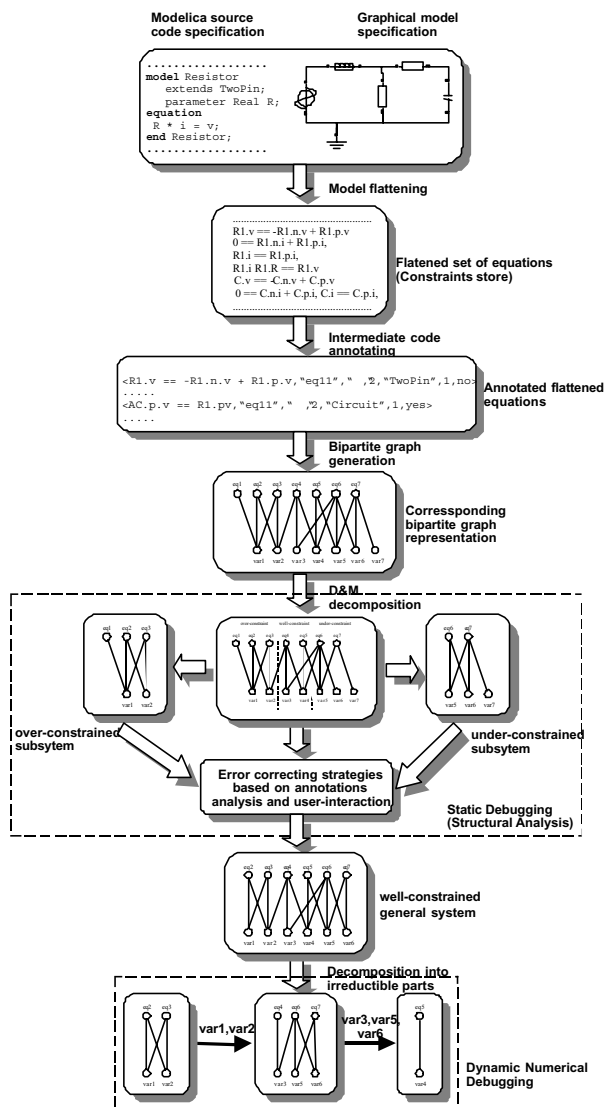


Figure 16. Debugger architecture.

6 Conclusions

Determining the cause of errors in models of physical systems is hampered by the limitations of the current techniques of debugging declarative equation based languages. We have presented a new approach for debugging such languages by employing graph decomposition techniques and have given several usage examples for debugging erroneous models. It has also been demonstrated that it is possible to create a tool with an enhanced user interaction capability that can be used explicitly in understanding complicated simulation models.

The contributions of this paper are twofold: the proposal of integrating graph decomposition techniques for debugging declarative equation based languages and an efficient equation annotation structure which helps the debugger to eliminate some of the heuristics involved in the error solving process. The annotations also provide an efficient way of identifying the equations and therefore helps the debugger in providing

error messages consistent with the user's perception of the original source and simulation model. The implemented debugger helps to statically detect a broad range of errors without having to execute the simulation model. Since the simulation system execution is expensive the implemented debugger helps to greatly reduce the number of test cases needed to validate a simulation model.

The merits of the proposed debugging technique are as follows:

- The user is exposed to the original source code of the program and is therefore not burdened with understanding the intermediate code or the numerical artifacts for solving the underlying system of equations.
- The user has a greater confidence in the correctness of the simulation model.
- The error fixing strategies are also prioritized by the debugger, which benefits the user in choosing the right error fixing solution.

References

- [1] Asratian A.S.; Denley T. and Häggkvist R. *Bipartite Graphs and their Applications*. Cambridge University Press 1998.
- [2] Dolan A. and Aldous J. *Networks and algorithms – An introductory approach*. John Wiley & Sons 1993 England.
- [3] Dulmage, A.L., Mendelsohn, N.S. *Coverings of bipartite graphs*, Canadian J. Math., 10, 517-534.
- [4] Elmqvist, H. *A Structured Model Language for Large Continuous Systems*. PhD thesis TFRT-1015, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden. 1978.
- [5] Flannery, L. M. and Gonzalez, A. J. *Detecting Anomalies in Constraint-based Systems*, Engineering Applications of Artificial Intelligence, Vol. 10, No. 3, June 1997, pages. 257-268.
- [6] Fritzson P.; Gunnarsson J; Jistrand M.; "MathModelica - An Extensible Modeling and Simulation Environment with Integrated Graphics and Literate Programming". *In Proceedings of the 2nd International Modelica Conference* (March 18-19, Munich, Germany, 2002)
- [7] Jistrand, M.; Gunnarsson J. and Fritzson P. "MathModelica – a new modeling and simulation environment for Modelica." *In Proceedings of the Third International Mathematica Symposium (IMS'99, Linz, Austria, Aug), 1999.*
- [8] Jistrand, M. "MathModelica – A Full System Simulation Tool". *In Proceedings of Modelica Workshop 2000* (Lund, Sweden, Oct. 23-24), 2000.
- [9] Maffezzoni C.; Girelli R. and Lluka P. *Generating efficient computational procedures from declarative models*. Simulation Practice and Theory 4 (1996) pages 303-317.
- [10] Vicente Rico Ramirez. Representation, Analysis and Solution of Conditional Models in an Equation-Based Environment. PhD Thesis, Carnegie Melon University, Pittsburgh, Pennsylvania, August 1998.
- [11] Wolfram S. *The Mathematica Book* . Wolfram Media Inc. (February 1996)

A Behavioral Model for DC-DC Converters using Modelica

David A. Torrey Ugur Savas Selamogullari
 Department of Electrical, Computer and Systems Engineering
 Rensselaer Polytechnic Institute
 Troy, NY 12180-3590 USA
 Email: torred@rpi.edu, selamu@rpi.edu

Abstract

This paper describes the development of a behavioral model of a dc/dc converter. The focus is on developing a model that simulates quickly, yet retains the behavioral features of a physical converter. Based on a physically motivated behavioral circuit model, the model is then implemented in the Modelica modeling language and simulated using the Dymola simulation environment. Simulation results are given. A theoretical method for tuning the simulation model to a physical converter is presented.

1 Introduction

Dc/dc converters are power electronic circuits that convert a dc voltage to a different regulated dc voltage level. In this respect, ideally a dc/dc converter can be considered as a dc transformer that provides lossless transfer of energy between circuits at different voltage or current levels. There are several topological variations of dc converters; the converter is generally described in terms of its voltage conversion characteristics. For example, a Buck Converter generally produces an output voltage that is lower than the input voltage.

The main objective of the dc/dc converter is to control one or more power semiconductor switches to transform dc input from one level to another. This is usually accomplished by controlling the on and off durations of the semiconductors; filters are then used to remove the associated ac components of the input current and the output voltage [1, 2, 3, 4].

Our objective here is in describing the macroscopic dynamic behavior of the dc/dc converter without getting caught up in the control and switching operations taking place within a physical converter. We are trying

to create a simulation model that faithfully emulates the behavior of a commercial dc/dc converter. The next section motivates the behavioral model.

2 The Behavioral Model

To model the true behavior of a dc/dc converter, a detailed model of the converter and its controller has to be built where every switching cycle is taken into account. However, the goal of our modeling effort is to emulate the behavior of a commercial dc/dc converter where the dynamics at the switching frequency are barely perceptible at the two ports of the converter. Modeling the cycle to cycle operation within the converter merely adds to the execution time of the model and potentially introduces numerical issues.

A more efficient way to simulate the behavior of a dc/dc converter is to use a circuit model that produces dynamic voltages and currents, but without considering internal converter quantities on an instantaneous or averaged basis [4]. The equivalent circuit contains no switching or switching ripple, and only the important macroscopic components of the waveforms are modeled [2]. With this approach, a behavioral model (input current, output voltage changes) of a dc/dc converter is obtained. The circuit and Modelica implementation are shown in Fig. 1 and Fig. 2, respectively.

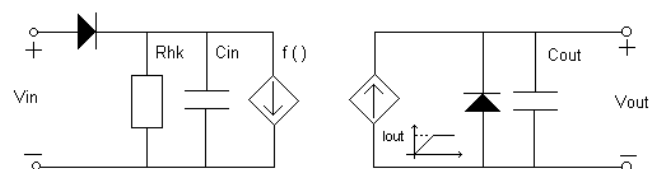


Figure 1: A behavioral model for a dc/dc converter.

A reflected load current is used on the input side of the converter. Any change in output voltage, load, and input voltage is reflected in this current since these are

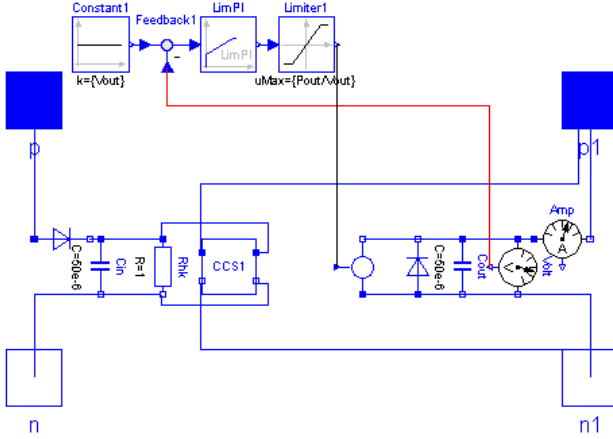


Figure 2: A schematic of the dc/dc converter behavioral model.

variables of the current equation $f(v_{in}, v_{out}, i_{out})$. At the output stage, a controlled current source is used to simulate the output current supplied by the converter. This current is controlled to regulate the output voltage. The model emulates the current limiting function of a practical dc/dc converter design that uses current-mode control. In order to control the output current, output voltage feedback is used. This is exactly what would be done in a physical converter. The output voltage is compared with the nominal voltage of the converter (V_{ref}) and an error is fed to a proportional plus integral (PI) controller. The output of this controller is used to drive a limiter with a limit value of $I_{out} = P_{out}/V_{ref}$. The output of the limiter drives a signal current source and defines the current drawn by the load.

On the input side there is a resistor (R_{hk}) to account for internal power needs of the converter. The power consumed by this resistor models the internal power consumed by the control circuitry; the power lost to switching, conduction and other parasitic losses within the converter are reflected in the efficiency used to determine the load current reflected to the input. The input capacitor is to provide energy storage for the input side and to simulate the input capacitance of a typical dc/dc converter. The output of the converter is only active if the input voltage is in within a valid range of input voltages $[V_{in_{min}}, V_{in_{max}}]$. Ideal diodes are used to restrict the flow of energy to be from the input to the output.

The model uses existing Modelica Library components such as resistors, ideal diodes, capacitors and two new components: a controlled current source (CCS) and a PI controller with limiting.

2.1 Controlled Current Source (CCS)

This component is the redefinition of existing Voltage Controlled Current Source within the Modelica Electrical Library with a new current definition equation and the addition of extra parameters. The relationship between input and output power of any dc-dc converter can be written as

$$P_{in} = \frac{P_{out}}{\eta} ; \quad (1)$$

$$V_{in} \cdot I_{in} = \frac{V_{out} \cdot I_{out}}{\eta} . \quad (2)$$

The load current reflected to the input is

$$I_{in} = \frac{V_{out} \cdot I_{out}}{\eta \cdot V_{in}} . \quad (3)$$

In the model, V_{out} is the output voltage, I_{out} is the load current measured through the current sensor, η is efficiency of the converter and V_{in} is the input voltage.

The Modelica code of this component is given below. Since the output voltage is already used in the current equation and there will be no output voltage when input voltage is outside of range, parameters $V_{in_{min}}$ and $V_{in_{max}}$ are not used in this component. The current equation i_2 is used under the complete model equation section due to dependency on outside variables, which are not defined inside the class itself. The Modelica code given below is complete with i_2 in it. When i_2 is used outside of the *class CCS*, the class becomes a partial class.

```
class CCS
  extends Electrical.Analog.Interfaces.TwoPort;
  parameter Real Vout;
  parameter Real Pout;
  parameter Real eff;

  equation

  i1 = 0;
  i2 = v1*(CurrentSensor.i)/((v2 + 1e-10)*eff);
end CCS;
```

2.2 PI Controller

This component is the combination of two standard Modelica library components: Gain and LimIntegrator. These two components are connected in parallel to form the proportional plus integral (PI) controller. The error signal is fed to the input of the PI controller and controller output is one that is proportional to both

magnitude and the integral of the input signal. Proportional control increases the speed of the response while using a term proportional to the integral of the error signal eliminates the steady state error.

Component LimIntegrator provides the option of turning the integrator off when the integral reaches a given upper or lower limit. This is used to prevent integrator wind-up. This way, the overall system has less overshoot and uses less control effort. The Modelica implementation of this component follows.

```
class LimPICont

parameter Real Pout;
parameter Real Vref;
Blocks.Continuous.LimIntegrator
LimIntegrator1(outMax[:]={Pout/Vref});
Blocks.Math.Add Add1;
Blocks.Math.Gain Gain1;
Blocks.Interfaces.InPort inPort;
Blocks.Interfaces.OutPort outPort;

equation

connect(LimIntegrator1.inPort,inPort);
connect(Gain1.inPort,inPort);
connect(Gain1.outPort, Add1.inPort1);
connect(LimIntegrator1.outPort,Add1.inPort2);
connect(Add1.outPort,outPort);

end LimPICont;
```

A theoretical method for determining the PI controller parameters is explained in the next section.

3 Determining PI Controller Parameters

The following method can be used to determine the PI controller parameters from a physical converter. A ripple is introduced on the output voltage by using an offset sinusoidal voltage source in series with a resistor as shown in Fig. 3. Using the sinusoidal voltage source provides control over the ripple frequency.

The capacitor voltage and current can be calculated from the circuit equations once the resistor voltage and current are known. These two currents add up to the dc/dc converter output current:

$$i_{out} = i_r + i_c \quad (4)$$

The output current can also be written as:

$$i_{out} = H(s)(V_{ref} - V_{out}) \quad (5)$$

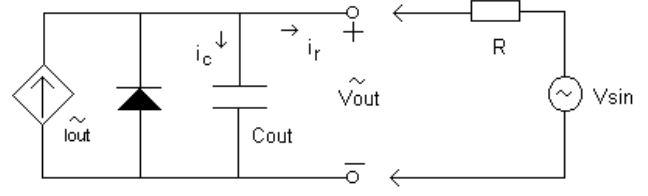


Figure 3: Circuit for PI parameter extraction.

$$H(s) = K_p + \frac{K_i}{s} \quad (6)$$

where $H(s)$ is the transfer function of the PI controller, V_{ref} is nominal output voltage of the converter and V_{out} is the output voltage. If only ripple quantities are considered, then

$$\tilde{i}_{out_d} = \tilde{i}_{r_d} + \tilde{i}_{c_d} \quad (7)$$

and

$$\tilde{i}_{out_d} = H(s) \cdot \tilde{v}_{out_d} \quad (8)$$

\tilde{v}_{out_d} is the ripple voltage across the output capacitor. Since both \tilde{i}_{out_d} and \tilde{v}_{out_d} are known, PI controller parameters K_p and K_i can be calculated.

For high frequency values, the integrator effect of the PI controller is minimized and the proportional effect will be dominant. Under these conditions

$$\tilde{i}_{out_d} = K_p \cdot \tilde{v}_{out_d} \quad (9)$$

so

$$K_p = \frac{\tilde{i}_{out_d}}{\tilde{v}_{out_d}} \quad (10)$$

For low frequencies, the integrator will be dominant, so

$$\tilde{i}_{out_d} = \frac{K_i}{s} \cdot \tilde{v}_{out_d} \quad (11)$$

$$= K_i \int \tilde{v}_{out_d} dt \quad (12)$$

The values found from this method are an approximation and give a starting point for final values. Test results using a physical converter can be used for fine-tuning of the simulation model.

4 Simulation Results

Simulations have been completed for 10Ω and 20Ω loading. For each simulation the following waveforms are plotted:

- Input current.
- Output voltage and current.

Simulation parameters and their values are:

- $V_{in_{min}} = 145 \text{ V}$
- $V_{in_{max}} = 208 \text{ V}$
- $V_{out} = 300 \text{ V}$
- $P_{out} = 5000 \text{ W}$
- $\eta = 0.95$
- LimIntegrator gain = 1 (actual gain would be determined from the test explained in Section 3)
- Gain = 1 (actual gain would be determined from the test explained in Section 3)
- $C_{out} = C_{in} = 50 \mu\text{F}$

In order to show the effect of input voltage range on converter operation, a sinusoidal voltage source is placed in series with a constant voltage source to form the input voltage (see Fig. 4). Both 10Ω and 20Ω loading simulations are run for this case as well. For

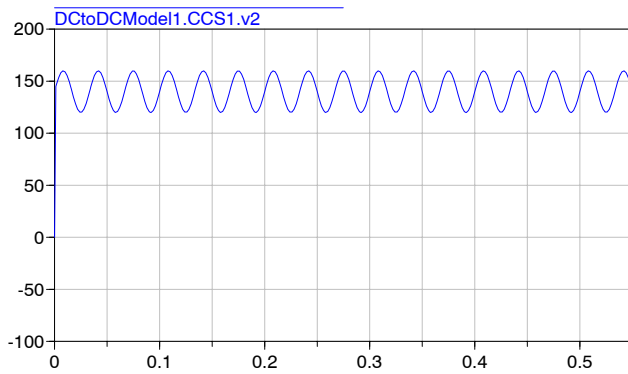


Figure 4: The time-varying input voltage.

the constant input voltage case, the simulation results for 10Ω loading are shown in Fig. 5 and Fig. 6. The output current limit can be seen in Fig. 5.

For the time-varying input voltage case, the simulation results for 20Ω loading are shown in Fig. 7 and Fig. 8.

The input current waveform in Fig. 8 for the time-varying input voltage case can be explained by considering its equation:

$$I_{in} = \frac{V_{out} \cdot I_{out}}{\eta \cdot V_{in}} \quad (13)$$

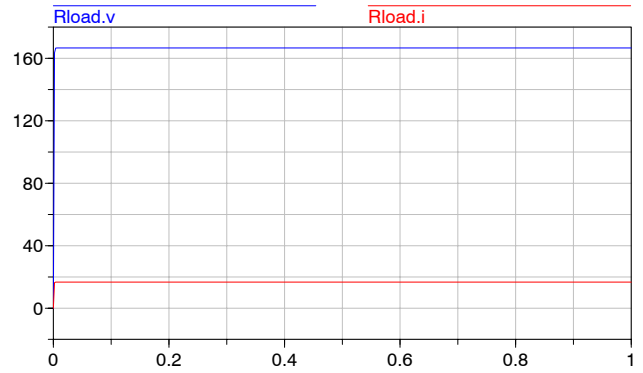


Figure 5: The output voltage and current waveforms for constant input voltage.

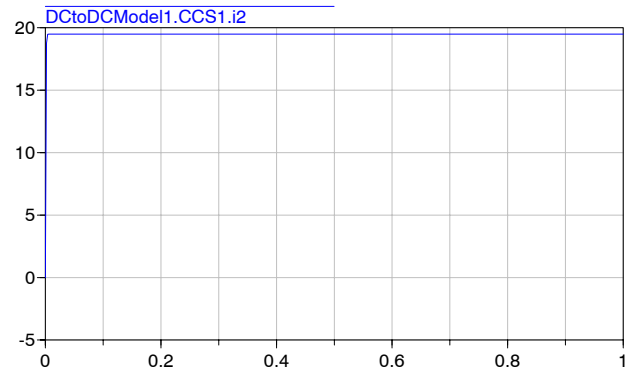


Figure 6: Input current waveform for the constant input voltage.

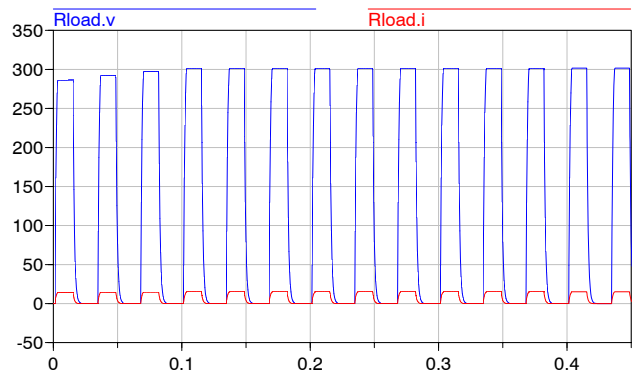


Figure 7: The output voltage and current waveforms for time-varying input voltage.

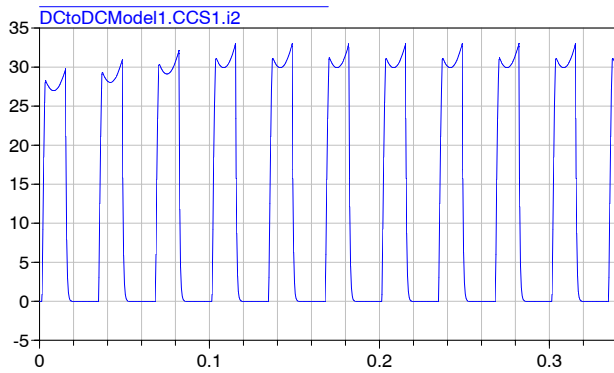


Figure 8: The input current waveform for the time-varying input voltage.

The output power has a constant value and the input voltage varies sinusoidally. Thus, the current value is found by division of a constant with a sinusoid. This explains why the input current decreases as the input voltage increases and vice versa.

5 Conclusion

This paper has described the development of a behavioral model for dc/dc converters. The model has been developed by using the modeling language Modelica. Two new Modelica components are developed from the existing library components. Model itself is also a new addition to Modelica Electrical Library. Simulation results for $10\ \Omega$ and $20\ \Omega$ loading are given. Information about the tuning of PI controller parameters to a physical converter is explained.

References

- [1] J. G. Kassakian, G. C. Verghese, and M. F. Schlecht, *Principles of Power Electronics*, Addison-Wesley, 1991.
- [2] P. T. Krein, *Elements of Power Electronics*, Oxford University Press, 1998.
- [3] N. Mohan, T. M. Undeland, W. P. Robbins, *Power Electronics*, 2nd Edition, John Wiley and Sons Inc., 1995.
- [4] D. W. Hart, *Introduction to Power Electronics*, Prentice Hall Inc., 1997.
- [5] Modelica Web Page (www.modelica.org)

[6] Dymola User's Manual

[7] Modelica Standard Electrical Library

A Modelica Code of DC-DC Converter Model

```

package DCModel

model DctoDCModel "DC/DC Converter Model"

parameter Real Pout;
parameter Real Vref;
parameter Real Vinmin;
parameter Real Vinmax;
parameter Real eff;
Real Iout;

model SignalCurrent
"Generic current source using the input signal as
source current"

extends Electrical.Analog.Interfaces.OnePort;
Blocks.Interfaces.InPort inPort (final n=1) ;

end SignalCurrent;

class LimPICont

parameter Real Pout;
parameter Real Vref;
Blocks.Continuous.LimIntegrator
LimIntegrator1 (outMax[:]={Pout/Vref}) ;
Blocks.Math.Add Add1 ;
Blocks.Math.Gain Gain1 ;
Blocks.Interfaces.InPort inPort ;
Blocks.Interfaces.OutPort outPort ;

equation

connect (LimIntegrator1.inPort, inPort) ;
connect (Gain1.inPort, inPort) ;
connect (Gain1.outPort, Add1.inPort1) ;
connect (LimIntegrator1.outPort, Add1.inPort2) ;
connect (Add1.outPort, outPort) ;

end LimPICont;

partial class CCS

extends Electrical.Analog.Interfaces.TwoPort;

equation

i1 = 0;

end CCS;

CCS CCS1 ;
Electrical.Analog.Basic.Resistor Rhk ;
Electrical.Analog.Basic.Capacitor Cout (C=50e-6) ;
Electrical.Analog.Ideal.IdealDiode Din ;
Electrical.Analog.Basic.Capacitor Cin (C=50e-6) ;
Electrical.Analog.Interfaces.PositivePin p ;
Electrical.Analog.Interfaces.NegativePin n ;
Electrical.Analog.Interfaces.PositivePin p1 ;
Electrical.Analog.Interfaces.NegativePin n1 ;
Electrical.Analog.Sensors.CurrentSensor Amp ;
Blocks.Math.Feedback Feedback1 ;
Blocks.Sources.Constant Constant1 (k={Vout}) ;
Electrical.Analog.Sensors.VoltageSensor Volt ;
Blocks.Nonlinear.Limiter Limiter1 (uMax={Pout/Vout}, uMin={0});
LimPICont LimPI (Pout=Pout, Vref=Vref) ;
Electrical.Analog.Ideal.IdealDiode Dout ;
SignalCurrent SignalCurrent1 ;

equation

connect (Volt.p, Amp.p) ;
connect (Din.n, Cin.p) ;
connect (Din.p, p) ;
connect (Cin.n, n) ;
connect (Rhk.p, Cin.p) ;
connect (Cin.n, Rhk.n) ;
connect (CCS1.p2, Rhk.p) ;
connect (CCS1.n2, Rhk.n) ;
connect (CCS1.p1, p1) ;
connect (CCS1.n1, n1) ;
connect (Cout.p, Volt.p) ;
connect (Cout.n, Volt.n) ;
connect (Volt.n, n1) ;
connect (Amp.n, p1) ;
connect (Constant1.outPort, Feedback1.inPort1) ;
connect (Feedback1.outPort, LimPI.inPort) ;
connect (LimPI.outPort, Limiter1.inPort) ;
connect (Volt.outPort, Feedback1.inPort2) ;
connect (Limiter1.outPort, SignalCurrent1.inPort) ;
connect (Dout.n, SignalCurrent1.n) ;
connect (SignalCurrent1.p, Dout.p) ;
connect (Dout.p, Cout.n) ;
connect (Dout.n, Cout.p) ;

Iout = Pout/Vref;

CCS1.i2 = (CCS1.v1)*(Amp.i)/((CCS1.v2 + 1e-10)*eff);

SignalCurrent1.i = if p.v > Vinmin and p.v < Vinmax then
SignalCurrent1.inPort.signal[1]
else 0;

end DctoDCModel;

end DCModel;

```

Modelica Implementation of Field-oriented Controlled 3-phase Induction Machine Drive

David A. Torrey Ugur S. Selamogullari
 Department of Electrical, Computer and Systems Engineering
 Rensselaer Polytechnic Institute
 Troy, NY 12180 USA
 Email: torred@rpi.edu, selamu@rpi.edu

Abstract

This paper focuses on the modeling of a cage induction machine drive under direct-field oriented control, also referred to as flux-vector control. The interest is to create a behavioral model of an induction machine drive under field orientation for system simulations since field-oriented control is now commonplace in commercial adjustable speed drives. First, the 3-phase induction machine model is developed. Then, the field orientation requirements are applied to this model and a voltage source inverter is used to emulate a controlled current source. Rotor field orientation is used because of fewer limitations than other field-orientation approaches. The inverter is assumed to provide the desired phase currents instantaneously and ripple free at some efficiency. Finally, these three components of the overall drive system, machine model, field orientation and inverter power supply, are combined together in a block using the Modelica language and simultaneously solved using the Dymola user interface.

1 Introduction

As a mature technology the induction machine enjoys use in many established applications and is frequently the first machine considered for emerging applications. The machine is comprised of a stator and a rotor. The windings on the stator and the rotor are assumed to be sinusoidally distributed in space to simplify the analysis of the machine [5]. The windings in the induction machine are coupled. This coupling is described through the inductance matrix, which describes how current in any one winding contributes to the flux linking the other windings. In a closed form,

the matrix equation can be written as

$$\lambda_{abc} = L_{abc}(\theta) i_{abc} \quad , \quad (1)$$

where λ_{abc} and i_{abc} are 1×6 vectors and $L_{abc}(\theta)$ is a 6×6 matrix dependent on rotor position.

The electrical dynamics for the induction machine can be written very succinctly using vector notation as

$$v_{abc} = \frac{d\lambda_{abc}}{dt} + R_{abc} i_{abc} \quad . \quad (2)$$

The electromagnetic torque is

$$\tau_{em} = \frac{1}{2} i^T \frac{dL(\theta)}{d\theta} i \quad , \quad (3)$$

and the mechanical dynamics are

$$H \frac{d\omega}{dt} = \tau_{em} - \tau_l \quad , \quad (4)$$

where the load torque τ_l includes windage and friction in addition to the shaft load. The moment of inertia (H) is assumed to include the inertia of the induction machine and whatever is connected to the induction machine through its shaft.

Taken together Eqs. 1 through 4 summarize the electromechanical dynamics of the induction machine. This description, however is inconvenient for studying dynamics and control for two reasons:

1. The order of the system is large.
2. The dependence on θ gives rise to a time-varying model.

To obtain a much simpler induction machine model, two power invariant transformations are used. The $\alpha\beta$ transformation converts a balanced three-phase machine into an equivalent balanced two-phase machine.

This is valuable because in a three-phase machine each phase couples into the other phase. A two-phase machine, on the other hand, has phase windings that do not couple because the axes of the magnetic fields are orthogonal (Fig. 1). In addition, it reduces the machine from six windings to four windings.

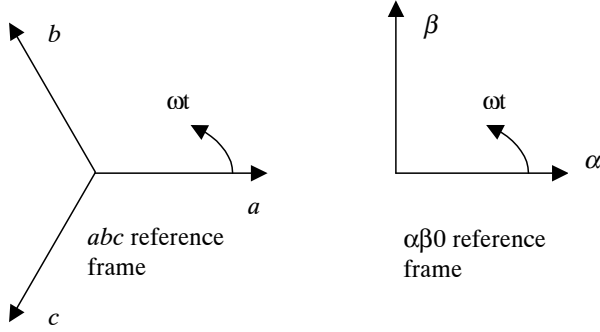


Figure 1: Space vectors for the abc reference frame and the $\alpha\beta 0$ reference frame.

If the phase a and phase α axes are coincident, N_3 is the number of turns of the three-phase winding and N_2 is the number of turns for the two-phase winding, then resolving the three mmfs of the abc frame along the α and β axes and equating the three-phase quantities gives

$$N_2 i_\alpha = N_3 i_a + N_3 i_b \cos\left(\frac{2\pi}{3}\right) + N_3 i_c \cos\left(\frac{4\pi}{3}\right), \quad (5)$$

$$N_2 i_\beta = N_3 i_b \sin\left(\frac{2\pi}{3}\right) + N_3 i_c \sin\left(\frac{4\pi}{3}\right). \quad (6)$$

For completeness, a third variable which is independent of i_α and i_β is needed:

$$N_2 i_0 = k N_3 i_a + k N_3 i_b + k N_3 i_c. \quad (7)$$

These relationships can be summarized in vector form as

$$\begin{bmatrix} i_\alpha \\ i_\beta \\ i_0 \end{bmatrix} = \frac{N_3}{N_2} \overbrace{\begin{bmatrix} 1 & -1/2 & -1/2 \\ 0 & \sqrt{3}/2 & -\sqrt{3}/2 \\ k & k & k \end{bmatrix}}^T \begin{bmatrix} i_a \\ i_b \\ i_c \end{bmatrix}. \quad (8)$$

In order to have invariance of power, $T = T^{-T}$ must be satisfied, and this is satisfied if $N_3/N_2 = \sqrt{2/3}$ and $k = 1/\sqrt{2}$ [7], giving

$$T = \sqrt{\frac{2}{3}} \begin{bmatrix} 1 & -1/2 & -1/2 \\ 0 & \sqrt{3}/2 & -\sqrt{3}/2 \\ 1/\sqrt{2} & 1/\sqrt{2} & 1/\sqrt{2} \end{bmatrix}. \quad (9)$$

Since 0 components do not couple to either the α or β phases and do not contribute to torque production, it is

best not to include them in the model. As a result, the model order is reduced from six states to four states. The T matrix becomes T_{23} for converting abc quantities to $\alpha\beta$ quantities and becomes T_{32} for the inverse transformation:

$$T_{23} = \sqrt{\frac{2}{3}} \begin{bmatrix} 1 & -1/2 & -1/2 \\ 0 & \sqrt{3}/2 & -\sqrt{3}/2 \end{bmatrix}; \quad (10)$$

$$T_{32} = T_{23}^T. \quad (11)$$

In a closed format, the electrical dynamics become

$$v_{\alpha\beta} = \frac{d\lambda_{\alpha\beta}}{dt} + R_{\alpha\beta} i_{\alpha\beta}. \quad (12)$$

The second transformation is another power-invariant transformation that is tied to the rotating magnetic fields in the airgap of the machine. This dq transformation eliminates the rotor position from the machine dynamics by projecting the dynamics onto a reference frame that moves with the airgap magnetic field. Fig. 2 shows an arbitrary vector \vec{a} decomposed into reference frames where one frame is displaced from the other by an angle ϕ . Each reference frame is denoted by a direct axis and a quadrature axis; the direct and quadrature axes are orthogonal. It can be shown that

$$\begin{bmatrix} a_{d2} \\ a_{q2} \end{bmatrix} = \begin{bmatrix} \cos(\phi) & \sin(\phi) \\ -\sin(\phi) & \cos(\phi) \end{bmatrix} \begin{bmatrix} a_{d1} \\ a_{q1} \end{bmatrix}. \quad (13)$$

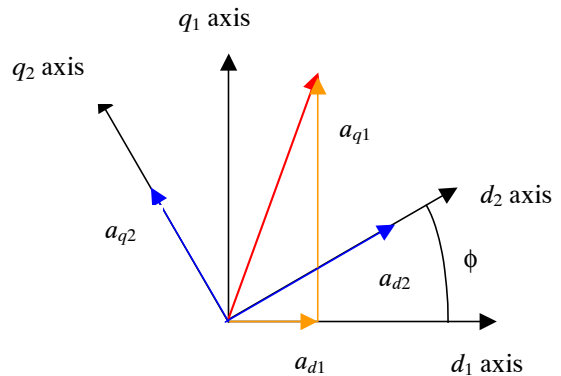


Figure 2: The vector \vec{a} decomposed into two reference frames, with angular displacement ϕ between them.

Fig. 3 shows the relationship among the three coordinate systems, where superscript s and r indicate stator and rotor frames, respectively. Accordingly, the $\alpha\beta$ dynamics of the stator and rotor are transformed to dq reference frame through an angle $P\phi$ for the stator and

$P(\phi - \theta)$ for the rotor quantities (Fig. 3). It follows that

$$\begin{bmatrix} \lambda_{sd} \\ \lambda_{sq} \\ \lambda_{rd} \\ \lambda_{rq} \end{bmatrix} = \begin{bmatrix} e^{P\phi J} & 0 \\ 0 & e^{P(\phi-\theta)J} \end{bmatrix} \begin{bmatrix} \lambda_{s\alpha} \\ \lambda_{s\beta} \\ \lambda_{r\alpha} \\ \lambda_{r\beta} \end{bmatrix}, \quad (14)$$

where

$$J = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}; \quad (15)$$

$$e^{J\phi} = \begin{bmatrix} \cos(\phi) & \sin(\phi) \\ -\sin(\phi) & \cos(\phi) \end{bmatrix}. \quad (16)$$

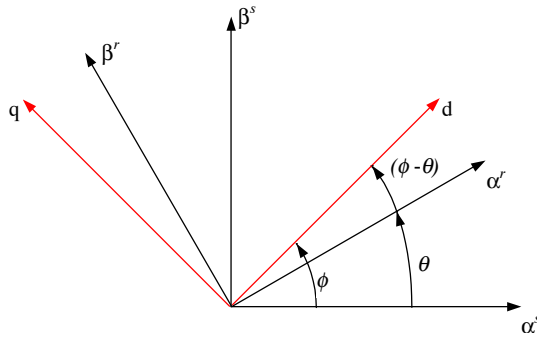


Figure 3: The relationship among the stator $\alpha\beta$ axes, the rotor $\alpha\beta$ axes, and the axes of the dq reference frame.

The machine dynamics become

$$v_{dq} = \frac{d\lambda_{dq}}{dt} + R_{dq}i_{dq} - \begin{bmatrix} \omega_s J & 0 \\ 0 & \omega_{sl} J \end{bmatrix} \lambda_{dq}, \quad (17)$$

where

$$\begin{aligned} \lambda_{dq} &= \begin{bmatrix} L_S I & M I \\ M I & L_R I \end{bmatrix} i_{dq}; \\ L_S &= L_s + L_{ss}, \\ L_R &= L_r + L_{rr}, \\ M &= \frac{3}{2} L_{sr}. \end{aligned} \quad (18)$$

ω_s is the synchronous angular velocity of the air-gap magnetic field, ω_{sl} is the slip frequency, and P is the number of pole pairs.

The torque equation is

$$\tau_m = \frac{3PL_{sr}}{2} (i_{sq}i_{rd} - i_{sd}i_{rq}) = PM(i_{sq}i_{rd} - i_{sd}i_{rq}). \quad (19)$$

2 Field Orientation

Field oriented control is a technique that structures the control of an induction-machine to be entirely parallel to that of a separately excited dc machine. That is, the field flux is oriented to be orthogonal to the torque-producing current. There are three commonly discussed versions of field orientation: rotor, stator and airgap. In each, the torque is given by vector product between flux and current. The flux involved is tied to the type of orientation, for example rotor orientation uses rotor fluxes. In the direct method, the airgap flux is measured directly by Hall sensors to determine the magnitude and orientation of the rotor flux vector, while the indirect field orientation is based on calculating the slip speed required for proper field orientation, and imposition of this speed on the motor [1, 2, 3, 4].

In direct field orientation, the orientation of the rotor flux is determined as follows:

1. The currents $i_{s\alpha}$ and $i_{s\beta}$ are calculated from the measured stator currents.
2. The fluxes $\lambda_{r\alpha}$ and $\lambda_{r\beta}$ are calculated from the measured airgap flux and the stator currents:

$$\lambda_{r\alpha} = \frac{L_R}{M} \lambda_{m\alpha} - (L_R - M) i_{s\alpha}; \quad (20)$$

$$\lambda_{r\beta} = \frac{L_R}{M} \lambda_{m\beta} - (L_R - M) i_{s\beta}. \quad (21)$$

3. The magnitude and orientation of the rotor flux is determined using the rectangular to polar coordinate transformation:

$$|\lambda_r| = \sqrt{\lambda_{r\alpha}^2 + \lambda_{r\beta}^2}, \quad (22)$$

$$\phi = \tan^{-1} \frac{\lambda_{r\beta}}{\lambda_{r\alpha}}. \quad (23)$$

Under field-orientation, we are forcing the induction machine to maintain orthogonality between appropriate flux and current through active control. The commands for flux and torque are generated by a higher-order controller. The block diagram of the field oriented-control of an induction machine is given in Fig. 4. Based on commanded flux and torque, the desired values for i_{sd}^* and i_{sq}^* are generated. The outputs of the flux and torque calculators are used to close the flux and torque feedback loops. By knowing the rotor position, the corresponding values for $i_{s\alpha}^*$ and $i_{s\beta}^*$ are determined using the rotary transformation:

$$\begin{bmatrix} i_{s\alpha}^* \\ i_{s\beta}^* \end{bmatrix} = e^{-J\phi} \begin{bmatrix} i_{sd}^* \\ i_{sq}^* \end{bmatrix}, \quad (24)$$

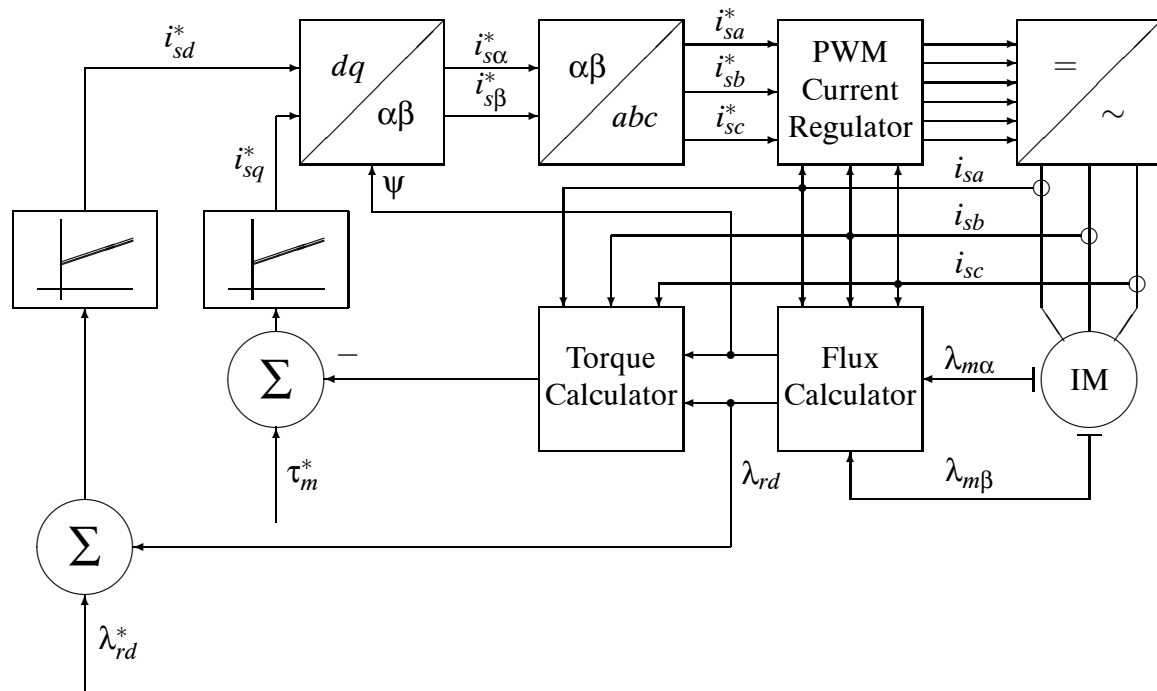


Figure 4: A general block diagram of how field oriented control is implemented.

where ϕ is the electrical angle that comes out of the flux calculator shown in Fig 4. The excitation angle is determined by simulating the induction motor in the $\alpha\beta$ reference frame. In our model, this is an easy task since the simulation gives us $\lambda_{r\alpha}$ and $\lambda_{r\beta}$ directly. Once $i_{s\alpha}$ and $i_{s\beta}$ are determined, they can be converted into the desired phase currents i_{sa}^* , i_{sb}^* and i_{sc}^* using the $\alpha\beta$ transformation. Because stator currents are imposed on the induction motor, phase voltages are no longer prescribed. Instead, the phase voltages reflect the self-consistent resolution of the induction motor model and the imposed currents. Accordingly, the number of states within the induction motor model is reduced by two. The rotor dynamics on the $\alpha\beta$ axes can be written as

$$\frac{d\lambda_{r\alpha}^s}{dt} = -\frac{1}{\tau_r}\lambda_{r\alpha}^s + \frac{M}{\tau_r}\dot{i}_{s\alpha}^s - P\omega_r\lambda_{r\beta}^s \quad , \quad (25)$$

$$\frac{d\lambda_{r\beta}^s}{dt} = -\frac{1}{\tau_r}\lambda_{r\beta}^s + \frac{M}{\tau_r}i_{s\beta}^s + P\omega_r\lambda_{r\alpha}^s \quad , \quad (26)$$

where $\tau_r = L_R/R_r$.

The stator voltages are given by

$$v_{s,\alpha\beta}^s = R_s i_{s,\alpha\beta} + \frac{d\lambda_{s,\alpha\beta}}{dt} \quad , \quad (27)$$

where

$$\lambda_{s,\alpha\beta}^s = \sigma L s i_{s,\alpha\beta} + \frac{M}{L_R} \lambda_{r,\alpha\beta} \quad , \quad (28)$$

and

$$\sigma = 1 - \frac{M^2}{L_S L_R} \quad . \quad (29)$$

Stator three-phase voltages and currents can be calculated using the T_{23} transformation matrix:

$$v_{s,abc} = T_{23} v_{s,\alpha\beta}^s \quad ; \quad (30)$$

$$i_{s,abc} = T_{23} i_{s,\alpha\beta}^s \quad . \quad (31)$$

It is common to use a voltage-source inverter to feed the induction motor with closed-loop current control. The inverter is assumed to be instantaneous and modeled as a current source. The current values are determined from the feedback loops of flux and torque. Pulse width modulation (PWM) is routinely employed to control the inverter switches [6]. Ideally the phase currents would be without ripple and perfectly track the commanded phase currents, while reality is somewhat different but close enough to the ideal. Thus, the inverter is assumed to be ideal and the current drawn from the DC side of the inverter can be calculated using conservation of instantaneous power:

$$i_{dc} = \frac{i_{s,abc}^T v_{s,abc}}{\eta_{inv} v_{dc}}, \quad (32)$$

where η_{inv} is the inverter efficiency.

There are three sections to this model. The first captures the electromechanical dynamics of the induction

machine when operating under direct field orientation. These dynamics when applied to the induction machine model prescribe the corresponding stator voltages and currents. The stator voltages and currents in turn dictate the current that must be supplied by the inverter power supply.

As a block diagram, the developed system model is shown in Fig. 5. The pins are provided for inverter connections, inports are used to get the flux and torque commands and a flange is used for the shaft of the machine. This way, the model emulates the reality. The Modelica code for the model is given in Appendix A. Since including the $v_{s,\alpha\beta}$ calculations in the simulation code causes a DAE index problem in the translation stage, the voltages are calculated separately using the derivative and gain blocks from the Modelica library. Then, the instantaneous power is calculated.

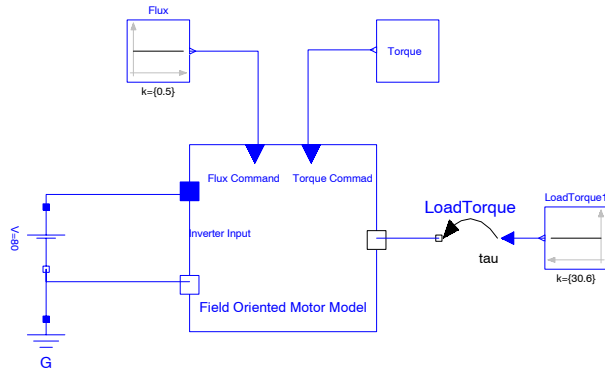


Figure 5: Block diagram of the field-oriented controlled 3-phase induction machine drive.

To illustrate the performance of the direct rotor flux field orientation system, the model is simulated under a commanded torque and flux profile[3]. The load torque is taken as 30.6 Nm: the total inertia of the system is 0.5 kg.m². The commanded torque profile is

$$\tau_m = \begin{cases} 135.3 & \text{Nm for } 0 < t \leq 0.5 \text{ sec} \\ 30.6 & \text{Nm for } 0.5 < t \leq 1 \text{ sec} \\ -74.1 & \text{Nm for } 1 < t \leq 1.5 \text{ sec} \\ -135.3 & \text{Nm for } 1.5 < t \leq 2 \text{ sec} \\ -30.6 & \text{Nm for } t > 2 \text{ sec.} \end{cases} \quad (33)$$

The rotor flux is to be maintained at 0.5 Wb. The programmed torque and flux commands are given in Fig. 6 and Fig. 7, respectively.

Simulation results for the torque, speed and flux of the model are shown Fig. 8 and Fig. 9. The inverter DC side current is plotted in Fig. 10. Comparing the simulation results with the desired torque and flux com-

mands shows that the model follows the commanded torque and flux.

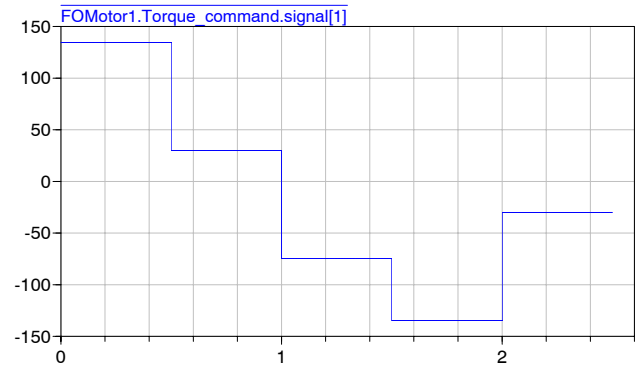


Figure 6: The commanded torque profile.

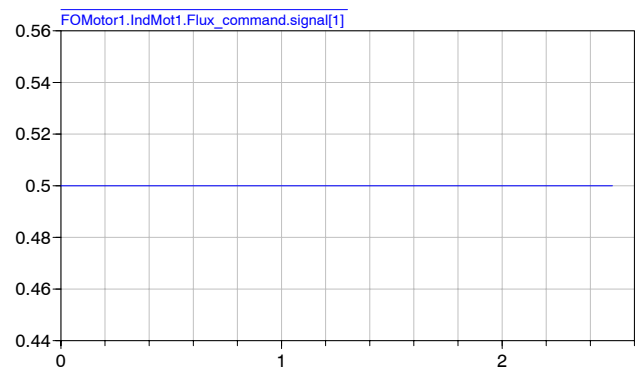


Figure 7: The commanded flux.

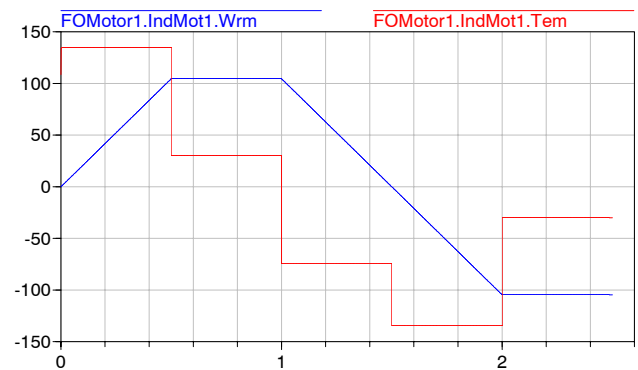


Figure 8: The torque and speed response of the induction machine.

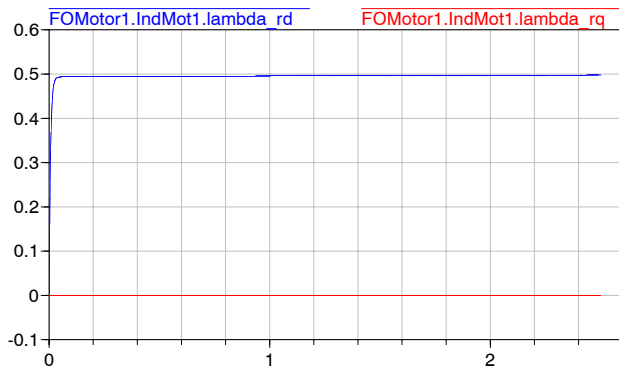


Figure 9: The simulated rotor fluxes.

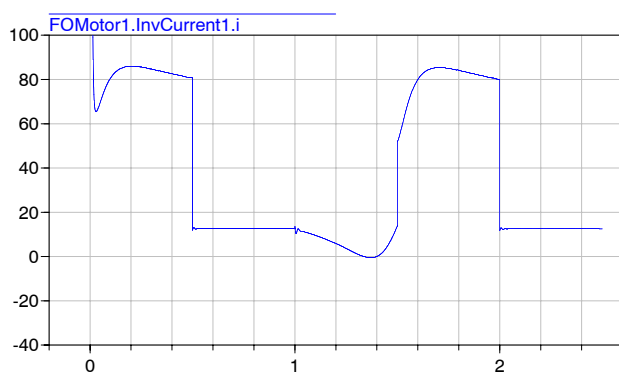


Figure 10: The inverter DC side current.

3 Conclusion

The underlying motivation for this study is the construction of a field-oriented controlled 3-phase induction machine drive model for system simulations. The final block diagram (Fig. 5) is comprised of three components: an induction machine model, field orientation requirements and an inverter power supply that provides the desired phase currents under field orientation. The inverter is assumed ideal and loaded consistent with the induction machine drive system. Since the physical phase currents are needed to obtain the inverter DC side current, the induction machine is simulated in $\alpha\beta$ reference frame under rotor direct field orientation. This is required to determine the orientation of the rotor flux. Pins, inports and a flange are used to provide the connection points to the user. The model is simulated under a programmed flux and torque profile and results are given. The Dymola simulation tool is used to solve the simultaneous resolution of three sections.

References

- [1] F. Blaschke, *Das Verfahren der Feldorientierung zur Regelung der Drehfeldmaschine (The method of field orientation for control of three phase machines)* Ph.D. Dissertation, TU Braunschweig, 1973.
- [2] F. Blaschke, *The principle of field orientation as applied to the new transvektor closed-loop control system rotating-field machines*, Siemens Review, Vol. 34, pp. 217-220, May 1972.
- [3] A. M. Trzynadlowski, *The Field Orientation Principle in Control of Induction Motors*, Kluwer, 1994.
- [4] D. M. Novotny and T. A. Lipo, *Vector Control and Dynamics of AC Drives*, Oxford University Press, 1997.
- [5] A. E. Fitzgerald, C. Kinglesy, Jr., and S. D. Umans, *Electric Machinery*, 5th ed., McGraw-Hill, 1990.
- [6] B. K. Bose, ed., *Power Electronics and Variable Frequency Drives*, IEEE Press, 1997, Chapter 4.
- [7] N. N. Hancock, *Matrix Analysis of Electric Machinery*, 2nd ed., Pergamon Press, 1974.
- [8] Dymola User's Manual
- [9] Modelica Web page (www.modelica.org)

A Modelica Code

```

package FieldOriented

class Tork
  Real T ;
  Modelica.Blocks.Interfaces.OutPort Torque ;
equation
  T = if (time <= 0.5) then (135.3) else if (time <= 1 and time > 0.5) then
    30.6 else if (time > 1 and time <= 1.5) then -74.1 else if (time > 1.5
    and time <= 2) then -135.3 else if time > 2 then -30.6 else 0 ;
  Torque.signal[1] = T ;
end Tork ;

partial class InvCurrent "Source for constant current"
  extends Modelica.Electrical.Analog.Interfaces.OnePort ;
end InvCurrent ;

class IndMot
  parameter Real Rs=0.294 "stator resistance(abc, dq frames)" ;
  parameter Real Rr=0.156 "rotor resistance (abc, dq frames)" ;
  parameter Real Lsl=0.00139 "abc frame stator leakage inductance" ;
  parameter Real Lrl=0.00074 "abc frame rotor leakage inductance" ;
  parameter Real Lsr=0.041 "abc frame mutual inductance" ;
  parameter Real P=3 "number of pole pairs" ;
  parameter Real H=0.5 "inertia of rotor" ;
  parameter Real f=60 "applied source frequency" ;
  Real M "dq frame mutual inductance" ;
  Real LS "dq frame stator inductance" ;
  Real LR "dq frame rotor inductance" ;
  Real D "leakage factor" ;
  Real Tem "electromechanical torque" ;
  Real Wrm(start=0) "Motor mechanical speed" ;
  Real theta "Rotor position angle" ;
  Real Isd "d axis stator current" ;
  Real Isq "q axis stator current" ;
  Real Is_alpha "alpha axis stator current" ;
  Real Is_beta "beta axis stator current" ;
  Real Isa "stator phase a current" ;
  Real Isb "stator phase b current" ;
  Real Isc "stator phase a current" ;
  Real lambda_alpha(start=1e-3) "alpha axis rotor flux" ;
  Real lambda_rbeta(start=1e-3) "beta axis rotor flux" ;
  Real lambda_rd "d axis rotor flux" ;
  Real lambda_rq "q axis rotor flux" ;
  Real lambda_salpha "alpaha axis stator flux" ;
  Real lambda_sbета "beta axis stator flux" ;
  Real Tem_fo "Torque under field orientation used for closed loop control" ;
  Real m_flux "magnitude of the rotr flux" ;
  Real p_flux "angle of the rotor flux" ;
  Real Tref "Reference Torque" ;
  Real Lref "Reference Flux" ;
  Real Terror "Torque error" ;
  Real Ferror "Flux error" ;
  Real errort(start=0) "integral of torque error" ;
  Real errorf(start=0) "integral of flux error" ;
  parameter Real Kif=500 "PI controller integral gain for Flux" ;
  parameter Real Kpf=1000 "PI controller proportional gain for Flux" ;
  parameter Real Kit=500 "PI controller integral gain for Torque" ;
  parameter Real Kpt=1000 "PI controller proportional gain for Torque" ;
  Modelica.Blocks.Interfaces.OutPort Lambda_salpha ;
  Modelica.Blocks.Interfaces.OutPort Lambda_sbета ;
  Modelica.Blocks.Interfaces.OutPort I_salpha ;
  Modelica.Blocks.Interfaces.OutPort I_sbета "OutPorts used above are
  used to calculate the Vs_alpha and Vs_beta since including them
  into the code brings DAE index problem" ;
  Modelica.Blocks.Interfaces.InPort Flux_command ;
  Modelica.Blocks.Interfaces.InPort Torque_command ;
  Modelica.Mechanics.Rotational.Interfaces.Flange_b shaft ;
equation
  Lambda_salpha.signal[1] = lambda_salpha ;
  Lambda_sbета.signal[1] = lambda_sbета ;
  I_salpha.signal[1] = Is_alpha ;
  I_sbета.signal[1] = Is_beta ;

```

```

M = 3/2*Lsr ;
LS = Lsl + M ;
LR = Lrl + M ;
D = (LS*LR - (M*M)) ;

Tref = Torque_command.signal[1] ;
Lref = Flux_command.signal[1] ;

Terror = Tref - Tem_fo ;
Error = Lref - m_flux ;

der(errorr) = Tref - Tem_fo ;
der(errorf) = Error ;

Isd = Kpf*Error + Kif*errorf ;
Isq = Kpt*Terror + Kit*errorr ;
Is_alpha = Isd*cos(p_flux) - Isq*sin(p_flux) ;
Is_beta = Isd*sin(p_flux) + Isq*cos(p_flux) ;

der(lambda_ralpha) = -Rr/LR*lambda_ralpha + M*Rr/LR*Is_alpha - P*Wrm*lambda_rbeta ;
der(lambda_rbeta) = -Rr/LR*lambda_rbeta + M*Rr/LR*Is_beta + P*Wrm*lambda_ralpha ;
lambda_rd = lambda_ralpha*cos(p_flux) + lambda_rbeta*sin(p_flux) ;
lambda_rq = -lambda_ralpha*sin(p_flux) + lambda_rbeta*cos(p_flux) ;
lambda_salpha = D/(LS*LR)*LS*Is_alpha + M/LR*lambda_ralpha ;
lambda_sbета = D/(LS*LR)*LS*Is_beta + M/LR*lambda_rbeta ;

m_flux = sqrt((lambda_ralpha^2) + (lambda_rbeta^2)) ;
p_flux = atan2(lambda_rbeta, lambda_ralpha) ;

der(Wrm) = if Wrm >= 0 then (1/H)*(Tem - shaft.tau) else (1/H)*(Tem + shaft.tau) ;
der(theta) = Wrm ;
shaft.phi = theta ;

Tem = (P*M/LR)*(Isq*lambda_rd - Isd*lambda_rq) ;
Tem_fo = (P*M/LR)*(Isq*m_flux) ;

Isa = Is_alpha*sqrt(2/3) ;
Isb = sqrt(2/3)*(-0.5*Is_alpha + sqrt(3)/2*Is_beta) ;
Isс = sqrt(2/3)*(-0.5*Is_alpha - sqrt(3)/2*Is_beta) ;

end IndMot ;

class PowerCalculator
block P2toP3
  extends Modelica.Blocks.Interfaces.BlockIcon ;
  parameter Integer n=1 "Dimension of input and output vectors." ;
  Modelica.Blocks.Interfaces.OutPort a(final n=n)"Connector 1 of Real input signals" ;
  Modelica.Blocks.Interfaces.OutPort b(final n=n)"Connector 2 of Real input signals" ;
  Modelica.Blocks.Interfaces.OutPort c(final n=n)"Connector 3 of Real input signals" ;
  Modelica.Blocks.Interfaces.InPort alfa(final n=n)"Connector of Real output signals" ;
  Modelica.Blocks.Interfaces.InPort beta(final n=n) ;
equation
  a.signal[1] = alfa.signal[1]*sqrt(2/3) ;
  b.signal[1] = sqrt(2/3)*(-0.5*alfa.signal[1] + sqrt(3)/2*beta.signal[1]) ;
  c.signal[1] = sqrt(2/3)*(-0.5*alfa.signal[1] - sqrt(3)/2*beta.signal[1]) ;
end P2toP3 ;

P2toP3 P2toP3_1 ;
P2toP3 P2toP3_2 ;
Modelica.Blocks.Math.Gain Gain1 ;
Modelica.Blocks.Continuous.Derivative Derivative1 ;
Modelica.Blocks.Math.Add Add1 ;
Modelica.Blocks.Math.Add3 Power ;
Modelica.Blocks.Math.Product Product1 ;
Modelica.Blocks.Math.Gain Gain2 ;
Modelica.Blocks.Continuous.Derivative Derivative2 ;
Modelica.Blocks.Math.Add Add2 ;
Modelica.Blocks.Math.Product Product2 ;
Modelica.Blocks.Math.Product Product3 ;
Modelica.Blocks.Interfaces.InPort inPort ;
Modelica.Blocks.Interfaces.InPort inPort1 ;
Modelica.Blocks.Interfaces.InPort inPort2 ;
Modelica.Blocks.Interfaces.InPort inPort3 ;
Modelica.Blocks.Interfaces.OutPort outPort ;

```

```

equation
  connect(Power.inPort1, Product1.outPort) ;
  connect(Power.inPort3, Product3.outPort) ;
  connect(Derivative2.inPort, inPort3) ;
  connect(Gain1.inPort, inPort) ;
  connect(Add2.inPort2, Derivative2.outPort) ;
  connect(Add2.inPort1, Gain2.outPort) ;
  connect(Add1.inPort1, Gain1.outPort) ;
  connect(Derivative1.inPort, inPort1) ;
  connect(Derivative1.outPort, Add1.inPort2) ;
  connect(P2toP3_1.alfa, inPort) ;
  connect(P2toP3_2.alfa, Add1.outPort) ;
  connect(P2toP3_2.beta, Add2.outPort) ;
  connect(P2toP3_1.a, Product1.inPort1) ;
  connect(P2toP3_2.a, Product1.inPort2) ;
  connect(P2toP3_1.b, Product2.inPort1) ;
  connect(P2toP3_2.b, Product2.inPort2) ;
  connect(P2toP3_1.c, Product3.inPort1) ;
  connect(P2toP3_2.c, Product3.inPort2) ;
  connect(Power.inPort2, Product2.outPort) ;
  connect(inPort2, Gain2.inPort) ;

  connect(P2toP3_1.beta, inPort2) ;
  connect(Power.outPort, outPort) ;
end PowerCalculator;

class FOMotor
  parameter Real Inveff=0.9 ;
  IndMot IndMot1 ;
  Modelica.Electrical.Analog.Interfaces.PositivePin p ;
  Modelica.Electrical.Analog.Interfaces.NegativePin n ;
  Modelica.Blocks.Interfaces.InPort Flux_command ;
  Modelica.Blocks.Interfaces.InPort Torque_command ;
  InvCurrent InvCurrent1 ;
  Modelica.Mechanics.Rotational.Interfaces.Flange_b shaft ;
  PowerCalculator PowerCalculator1 (Gain1.k={IndMot1.Rs}, Gain2.k={IndMot1.Rs}) ;
equation
  connect(IndMot1.Flux_command, Flux_command) ;
  connect(IndMot1.Torque_command, Torque_command) ;
  connect(InvCurrent1.p, p) ;
  connect(InvCurrent1.n, n) ;
  connect(IndMot1.shaft, shaft) ;
  connect(PowerCalculator1.inPort, IndMot1.I_salpha) ;
  connect(PowerCalculator1.inPort1, IndMot1.Lambda_salpha) ;
  InvCurrent1.i = (PowerCalculator1.Power.outPort.signal[1]/(Inveff*InvCurrent1.v)) ;
  connect(PowerCalculator1.inPort2, IndMot1.I_sbета) ;
  connect(PowerCalculator1.inPort3, IndMot1.Lambda_sbета) ;
end FOMotor;

end FieldOriented ;

```


Numerical Simulation of Complex Cooling and Heating Systems

Dipl.–Ing. Stefan Wischhusen Prof. Dr.–Ing. Gerhard Schmitz
Technical University Hamburg–Harburg
Department of Technical Thermodynamics
Denickestraße 17, D–21073 Hamburg, Germany

Abstract

In cooperation with the Imtech Deutschland GmbH & Co. KG (formerly known as Rudolf Otto Meyer GmbH & Co. KG) a research project was conducted. The aim of the project is to develop a simulation tool for heating and cooling systems in building applications. This tool should enable configuration studies and dynamic system simulations with time scales from a few seconds up to one year within short computational times. Therefore, the simulation environment of Dymola, containing the programming language Modelica, is used to model complex heterogeneous systems. In this paper the recent library for heating components is presented and the implemented models are subsequently used for a verifying simulation of an existing thermal power plant. Furthermore, the graphical user interface HKSIM is introduced as an applied tool for project management and post-processing, integrating Dymola for model editing and simulation only.

1 Introduction

So far, there is no simulation tool known which enables the dynamic simulation of both, complex heating and cooling systems in building applications, allowing free choice of parts and system layout. Therefore, this research project was conducted in 2001 with the goal of developing such an object-oriented library. Partners involved in this project are the Department of Technical Thermodynamics of the Technical University Hamburg–Harburg and Imtech Deutschland. From the viewpoint of a system engineer it is desirable to predict the dynamic behaviour of a complex plant during the concept and definition phase of a project. The development of running costs is due to the gas and electric power consumption of every single component, like for instance pumps, boilers etc.. These are the key optimisation numbers of such

a system for the operator as well as for the system builder. Since many owners of heating (and cooling) plants neither have the knowledge nor the financial budget for a reconfiguration, *contracting* companies are commissioned with the optimisation. There are of course various kinds of contracts possible. One could be the optimisation of an existing plant, another the supply with heating and/or cooling where the customer just pays for the delivered energy and not for the plant, which has to be build for that purpose (outsourcing). The benefit resulting from that simulation tool is not only of economical nature but also a reduction of energy consumption which means a decreased production of carbondioxid. This is the background for the work which is described in this paper.

2 Concept of Simulation

Since there is already a lot of building simulation software available, like e.g. TRNSYS, BLAST and others, the development is focused on the plant components. Due to the separation of the building from the system simulation some work has to be invested in the linkage between both calculations. The simulation of the building is supposed to be done first. The results from this simulation, basically the heat demand (requirement of refrigeration, respectively) and room temperature, are stored in a data file which can be read in subsequently by a table–interpolation–model `CombiTableTime` from the `Modelica–Standard–Libraries` [3] (Fig.1). To the interpolation–model connects a model of a radiator, which is not a single heating element in that sense but represents one heating circuit or even a whole building. The main idea is, that the heat demand is directly translated into a corresponding mass flow rate by functions implemented to the heat consuming model. Usually, for the modeling of thermo–hydraulic control volumes two state variables are needed, like e.g. pressure p and enthalpy h . During the integration process the speed

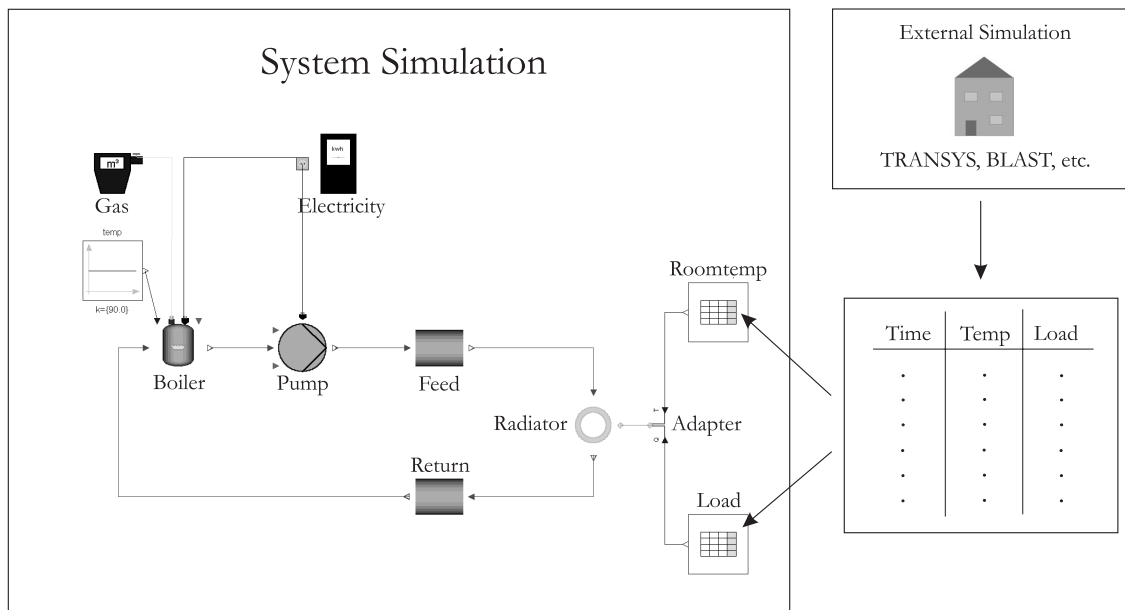


Figure 1: Concept of separated simulations

of the solver is influenced by the dynamic behavior of the state variables, or better, the time scales in which the state variables change. These time scales differ by up to three magnitudes in heating installations. This is the reason why an incompressible fluid model is selected with a simple algebraic mass balance in order to reduce the calculation time. Thus, the mass flow rate is just considered as a signal, which is not calculated by a detailed momentum balance, including the actual pressure difference between inlet and outlet, but is limited by the pumps pressure difference and maximum flow rate. The used concept results in calculation times of a few minutes for a plant simulation of one year. In addition to that, the influence of the pressure on the gas and electric power consumption of a heating installation is considered to be low so that it can be neglected. Due to these reasons only the water temperature is taken as a thermodynamic state variable of each component.

In favour of a conservative energy balance a control volume formulation is chosen and the temperature is projected downstream. In case of an adiabatic control volume V from the energy balance follows:

$$V \rho c_w \dot{T} = c_w \cdot (\dot{m}_{in} \cdot t_{in} - \dot{m}_{out} \cdot t_{out}) \quad (1)$$

with the thermodynamic temperature T , the Celsius-temperature t and mass flow rate \dot{m}

$$\begin{aligned} T &= t_{out} + 273.15 \\ \dot{m}_{in} &= \dot{m}_{out} \quad (\text{incomp. fluid}) . \end{aligned}$$

A water property model is needed to calculate the specific heat capacity c_w and density ρ . This can be done efficiently by assuming constant values or more accurately by providing polynomial functions [1] depending on temperature. Since the accuracy is hardly enhanced by less than 1% but the calculation times are increased by 300% the constant value approach is considered to be accurate enough.

For the calculation of the gas consumption of a boiler the efficiency coefficient η is needed on the one hand, which is defined as the ratio of heat output to burner output and on the other hand the feed temperature, which is provided by the controller. Boiler manufacturers usually specify η depending on different states with regard to the biggest impact, like e.g. load ratio, return temperature or average boiler temperature. The specified values can be interpolated by a characteristic diagram model which also refers electric power consumption of the burner. The common methods, functions and interfaces are implemented in a class (BaseBoiler), whereas the characteristic diagram class Characteristic Diagram can be replaced by "drag and drop" or modified to generate a new boiler model in an convenient and easy way (see Fig.2). The same concept of an object-oriented model is also used for the modeling of other components of a heating installation, like for instance pumps or combined heat and power plants (CHP)[2].

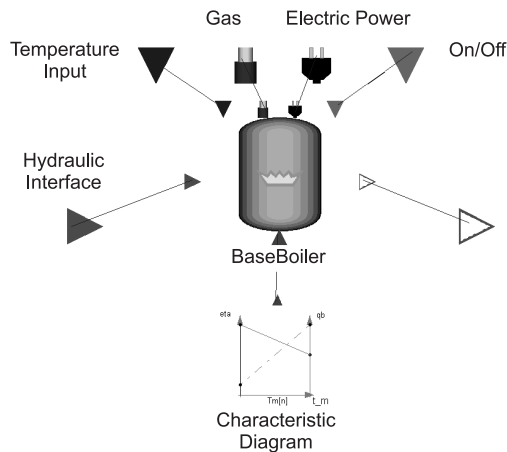


Figure 2: Diagram layer of a boiler model

3 Library Content

The recent library contains models for components of heating and cooling installations in subpackages. Since the later users of the library shall only modify the existing component classes by "drag and drop" and parameterisation, the base classes will be stored in an extra library which cannot be modified. An important requirement for the parameterisation (and also for the component model itself) is that the needed information is made available by the manufacturer. This has been checked in the beginning of every model development. Especially, when programming components for cooling systems it was found out that the supplied information level is very low.

So far, the boiler sublibrary consists of models for a broad range of small to large gas-fired boiler types (oil fuel could be easily introduced), as there are condensing boilers and normal boilers equipped with atmospheric to modulating burners. The electric power consumption of the gas burners ventilation motor is also taken into account.

The pump sublibrary provides models for uncontrolled and controlled pumps. These models can be used for heating, cooling and service water supply.

Pipes are modeled by discretised control volumes and wall classes, which enable the calculation for heat losses. Usually, these losses are neglected when the heating installation and the pipes are part of the building which has to be heated because it is considered to contribute to the heat demand. Thus, in most cases adiabatic pipes without any wall model are used to model the systems delay.

For the simulation of domestic hot water systems a sublibrary provides models for hot water storage tanks.

The heat is either stored in horizontal or upright standing tanks which may have water layers with different water temperatures. The heat is transferred in internal or external heat exchangers (loading systems).

Since the combined heat and power technology is becoming more and more important – not only in very large heating installations – a model for gas driven CHP's has been added to the library.

Currently, the work focuses on the development of components for cooling systems, like water chillers and cooling towers. The previously developed components of heating systems will be reused as far as possible, for example the pump models, pipe models and storage tank models.

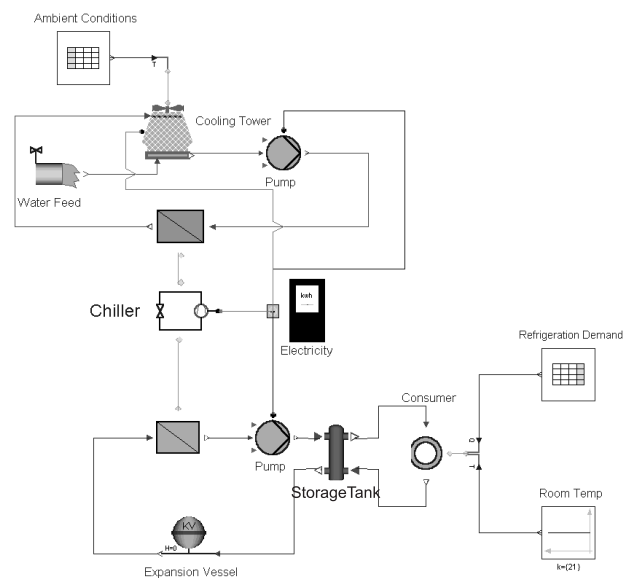


Figure 3: Schematic of a cooling installation

The water chiller model is not numerically described by a control volume because the needed state equations for this purpose include pressure and enthalpy which decrease the speed of simulation too much as discussed in section 2. Although with the current computing facilities real-time simulations of thermo-hydraulic systems are partly possible (good guess of initial conditions provided) these models are not suitable for simulations of one year. Thus, characteristic diagrams are implemented in the chiller model which refer to the used refrigerant and type of compressor. The supplied functions were derived from technical data of various manufacturers and device sizes [4] with a maximum spread of 10%. A simple model example of a cooling system is presented in Fig.3.

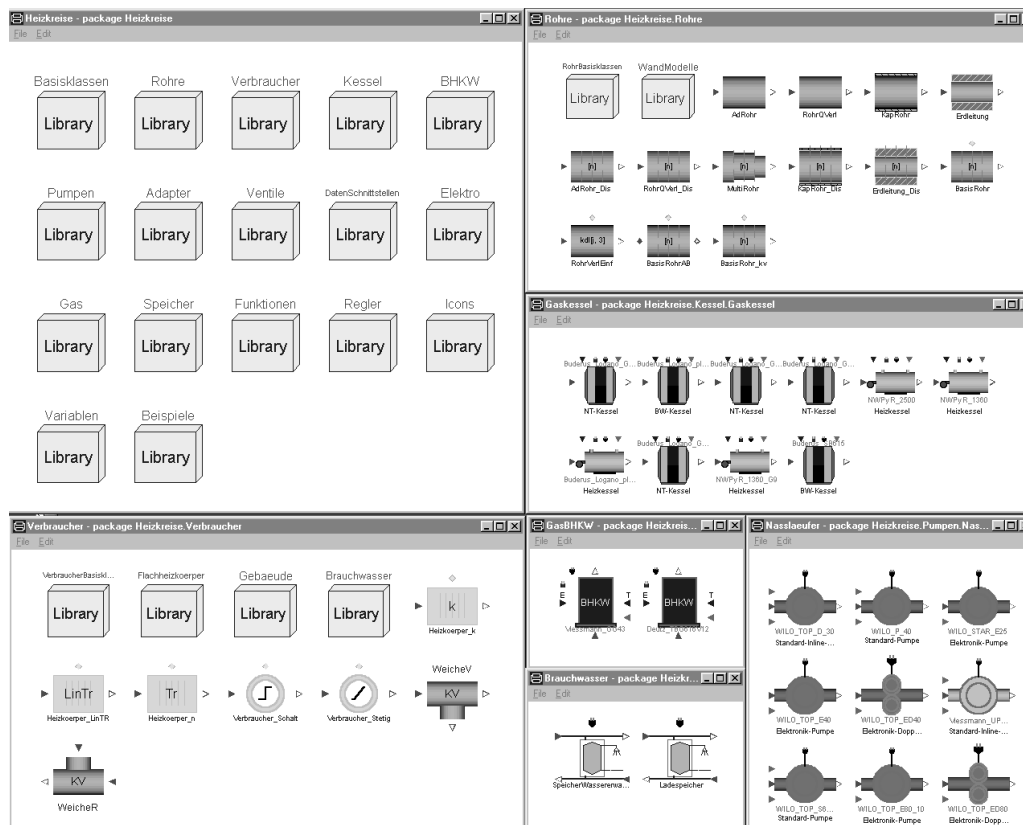


Figure 4: Recent package Heizkreise for heating installation components

4 Verification of Existing Models

The following section is focused on the simulation and verification of models of heating installations with measurement data. At first, the fundamental question, why a dynamic simulation environment is chosen, shall be answered by means of a simple example.

4.1 Dynamic Simulation – Why?

Due to the fact that real heating and cooling installations may have large fluid capacities a dynamic simulation of such systems is necessary. In order to demonstrate the difference between static and dynamic simulation a simple feed temperature step is performed with three different boilers and the same hydraulic input ($\dot{m} = 2.0 \text{ kg/s}$ and $t_{\text{return}} = 60^\circ \text{C}$). The results from the three simulations are shown in Fig.5 where the step of the feed temperature is triggered after one hour. In a static simulation, represented by the continuous line, the boiler follows the step input ideally (sufficient burner output of at least 265kW assumed). In a dynamic calculation the water volume within the boiler has to be heated up by the gas burner's rated output until the switch-off temperature is reached (1K below

set value). Thereover, the boiler model switches into the ideal mode which means the heat input is adjusted ideally without triggering further state events by burner starts and stops. It is evident that the dynamic calculated feed temperature with regard to the static simulation increases in a slower way and that the warm-up time is influenced by the size of the boiler. The temperature rise is slowed down by the continuous mass flow rate which has a bigger impact on smaller boilers. In the case of a too small output rate of the boiler (boiler with a nominal capacity of 140kW) the feed temperature input is actually not reached.

During the warm-up of the feed temperature the gas burner operates at full capacity which means a higher gas consumption than calculated in a static simulation. In addition to that, the operation efficiency drops due to the rising water temperature, the increased load and rising exhaust gas temperature. Comparing the gas consumption for a simulation time of two hours including one temperature step a deviation of 3% (400kW) to 13% (2500kW) is found depending on the total water volume of the boiler. Therefore, a dynamic simulation of complex heating and cooling installations is considered to be necessary.

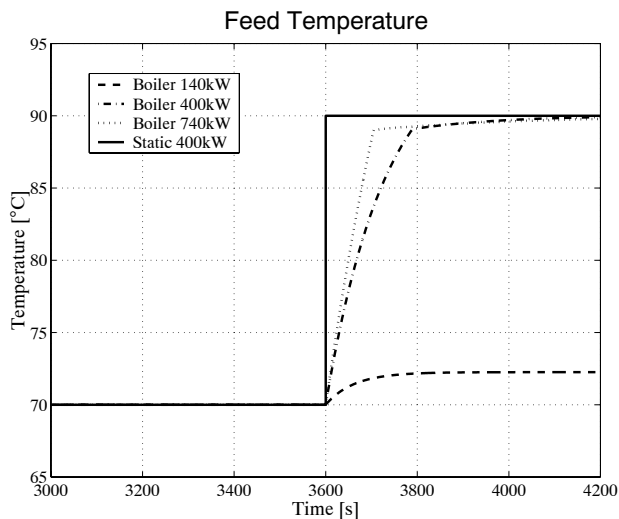


Figure 5: Feed temperature step for different boiler sizes and simulation types

4.2 Simulation of a Thermal Power Plant

Furthermore, the existing models have been used to simulate a thermal power plant which is operated by the contracting subsidiary company of Imtech Deutschland, Imtech Contracting. The schematic diagram of the heating installation model is shown in Fig.6. The measurement data with regard to mass flow rate, return, feed and ambient temperature is interpolated by the interpolation table model `CombiTableTime DataInput` and `AmbientTemp` which is supplied by the tables sublibrary of the `Modelica Additions Libraries`. The feed temperature input is used to compare the energy flow of the model with the real system. The source model `HydSource` represents the heat consuming part. This is done because measurement data is not supplied for the consumers and the associated buildings were not simulated. To make sure that the same amount of energy is transferred like in the real system, the mass flow rate is related to the energy flux, rising when the simulated feed temperature is falling. The shown system consists of the following components:

- **Boiler:** gas-fired boiler (nominal capacity of 1,360kW) equipped with a modulating (output can vary in a certain range) burner
- **Pump:** pump with integrated electronic speed regulation for variable head control, maximum flow rate 64m³/h
- **AdmixingPump:** 3-speed inline pump, maximum flow rate 32m³/h

- **MixingValve:** 3-way valve, adjusting a constant temperature difference ΔT between return and feed of 20K
- **Controller:** ambient temperature lead feed temperature controller
- **ExpansionVessel:** model is used as a data sink for the mass flow signal

Finally, the gas and electric power consumption is summed up in the gas and electric meter model.

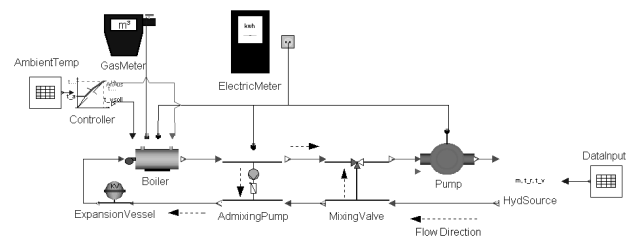


Figure 6: Schematic of a thermal power plant

Measurement data with an interval length of 15 minutes has been provided by Imtech Contracting over a period of 9 days in February. The data set contains ambient, feed and return temperature of the pipeline as well as of the boiler. Furthermore, the volume flow rate has been measured. With use of these input values the heat demand of the consumers can be calculated directly (Fig.8).

As Fig.7 reveals, the boiler feed temperature of the simulation follows the measured temperature understanding the mentioned ideal model behaviour which does not produce noise resulting from measurement tolerances and a chopping burner output below 250kW heat demand.

With regard to Fig.8 it is evident that the simulated burner output is just a little higher than the heat demand because of the boiler's high efficiency level of 94%. The profile of the heat demand shows typical events like a lower load at night followed by a warm-up peak in the early morning. Two times the plant was even turned off completely which was not due to weather conditions but to maintenance reasons.

4.3 Case Study

In this section a comparison of four often used boiler configurations is undertaken. Thus, the boiler model

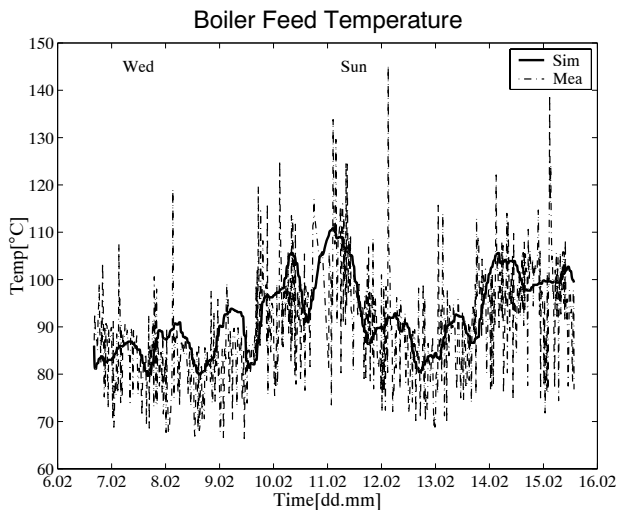


Figure 7: Boiler feed temperature simulated over a period of 9 days in Feb.

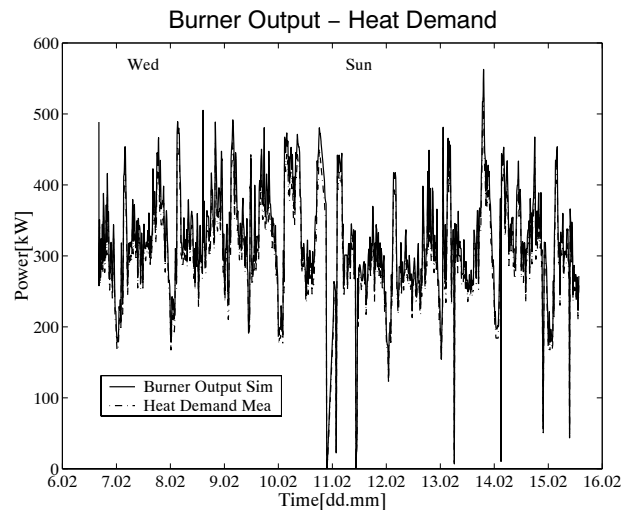


Figure 8: Simulated burner output and measured heat demand

in Fig.6 was replaced by the following configurations, which are all state-of-the-art:

- two parallel boilers (rated output 740kW each) (Fig.10)
- one condensing boiler (1350kW)
- a condensing boiler (640kW) followed by a simple boiler (740kW) for higher duties (Fig.11)

For reasons of comparison, the heat demand data of the previous simulation run was doubled by just increasing the mass flow rate. The prices for gas and electricity have been set with regard to the usual contract conditions (Gas: 3.6Ct/kWh, Electricity: 14(day)–10(night)Ct/kWh).

The outcome is presented in the chart diagram of Fig.9. Obviously, the existing configuration is also the most expensive with regard to running costs (as mentioned before the average efficiency of the implemented boiler model is up to 94%) which has two reasons. First, the gas burner of the boiler model is driven by an electric motor with a rated power of 6.5kW, which is considerably higher than that of the two installed pumps and of the smaller boilers (simple boiler: 2.6kW and cond. boiler: 1.4kW). Second, the average efficiency of the condensing boilers is 98% and would be even higher if the return temperature was lower than 60°C as in this case.

Less expensive is the configuration with two parallel boilers because of the smaller burner motor as mentioned before and the fact that only one boiler operates

in part load. The gas consumption is even rising slightly, since the efficiency of a high loaded boiler drops and this effect can not be fully compensated by reduced standby losses of the second boiler.

A single condensing boiler configuration reduces the fuel costs by 230 EUR in 9 days while the electric costs are not affected, since the same burner type is implemented.

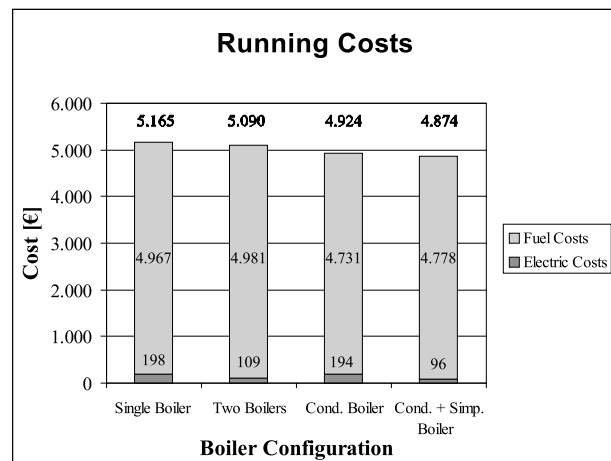


Figure 9: Running costs of case study

Apparently, the highest reduction (approx. 300 EUR or 6%) in this case study could be realised by the serial configuration of a condensing boiler with a simple boiler because it combines the positive effects of two smaller boilers and condensing technology. This is a reason why this concept is preferred by

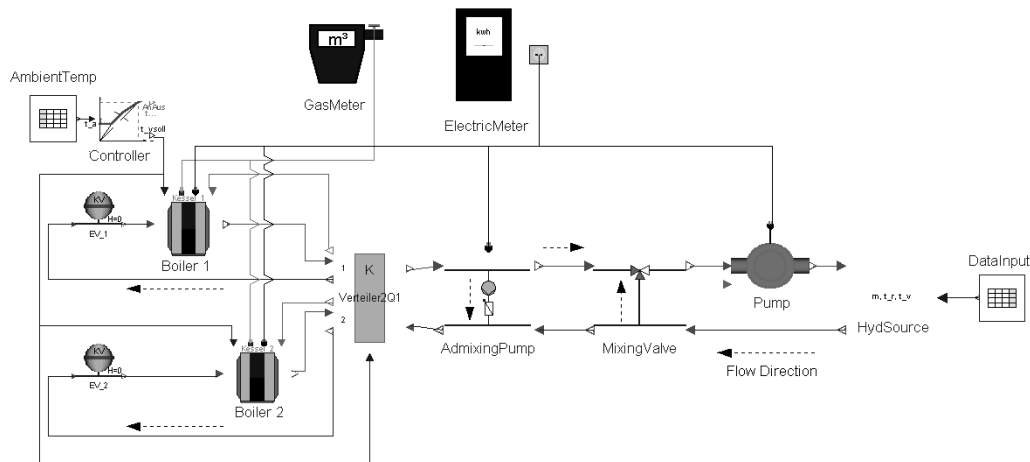


Figure 10: Configuration with two parallel boilers

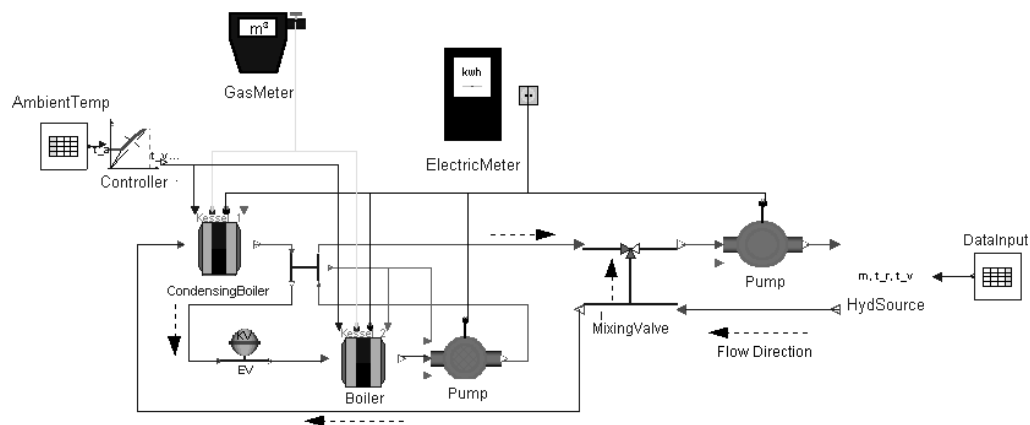


Figure 11: Configuration with one condensing boiler followed by a normal boiler

Imtech Deutschland when it comes to the design of new heating installations with low return temperatures. Nevertheless, it has to be emphasised that in this special case the calculated reduction of running costs may be too low to be worthwhile, especially if the investment costs of the more efficient configurations are much higher.

5 Graphical User Interface – HKSIM

As mentioned before a graphical user interface for Windows is developed by the Zentrale Ingenieurtechnik (ZIT) department of Imtech Deutschland for a number of reasons:

1. As an expert tool, the used simulation environment, Dymola, needs to be controlled by an applied user interface which is focused on the end-user and simulation background.

2. A data base connection is needed to save different projects and to give information about former simulations and their outcome.
3. The results from the simulations can be presented in a chart, which can be printed out as a standard information sheet for customers or can be exported to other applications.

In fact, Dymola is a powerful, but also complex simulation tool, too complex for a straightforward usage when results are needed fast. Thus, the graphical user interface is utilising the applied features of Dymola, like the model editor (system building by "drag and drop") and the simulator. Other applications, like the plot and animation window are faded out. Also, the end-user, for instance the project engineer, is not expected to program with use of Modelica.

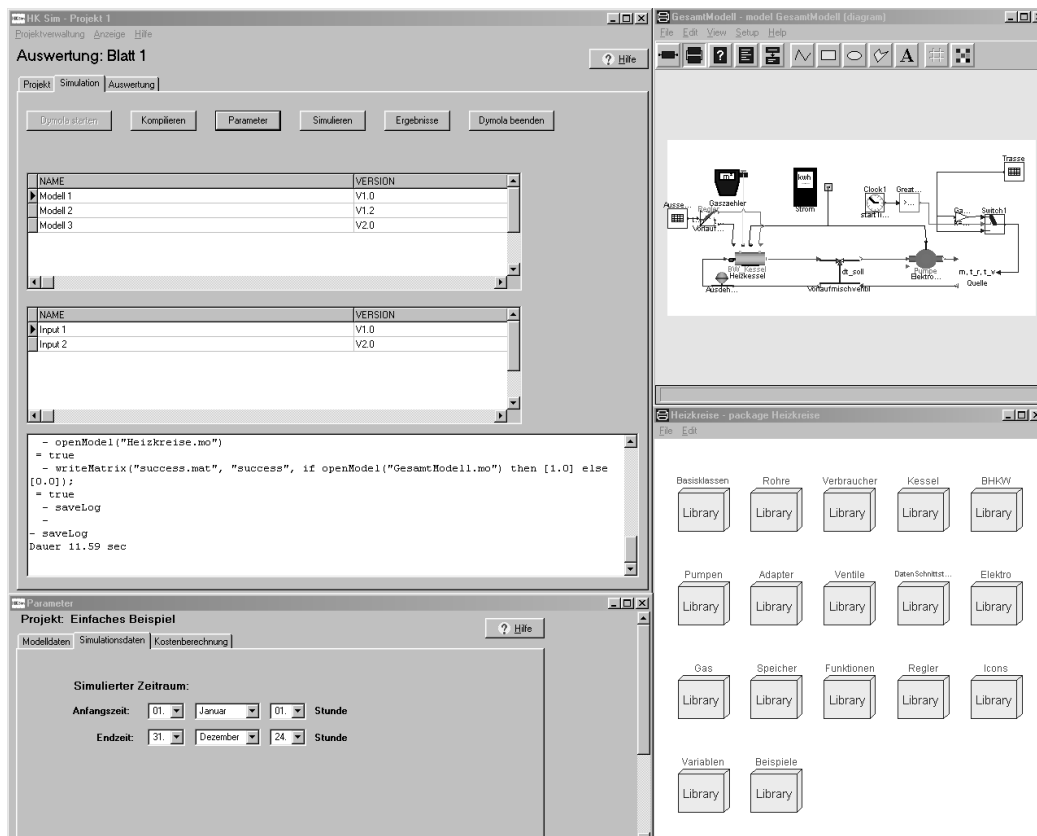


Figure 12: Screen shot of HKSIM's main (top 1.) and parameter window (bottom 1.)

The typical procedure can be described as follows: The user chooses a project from the list of existing installations (Fig.12) or a new one is opened, alternatively. It has to be emphasised that every modification of an existing project is saved under a new model name and can be restored later. Afterwards, Dymola can be started and the model library is loaded. From this point, the user can decide if the Dymola environment or HKSIM is used. After the configuration or modification of the project model is completed, parameter settings can be performed within the Dymola diagram layer. The same applies for the model translation. A convenient setting of the simulation time is enabled by means of a submenu of HKSIM, which converts start and stop times from a pull down date and hour menu to start and stop times in seconds (Fig.12). In other submenus data input files for the boundary conditions (e.g. heat demand, ambient temperature) can be chosen as well as parameters with regard to operation costs calculation before the simulation is finally started. The results are then displayed by an implemented post processing tool. For clarity reasons only a predefined choice of relevant variables can be plotted from a result browser which also offers the results of for-

mer simulation runs in numerical order. An example of a plot diagram is shown in Fig.13.

6 Conclusion

A simulation tool for heating and cooling processes in building applications is needed to calculate the performance of complex system layouts with regard to economical and ecological aspects. In this paper, the development of a system library is described and it was pointed out that a dynamic simulation in this field is necessary as a matter of accuracy. The library components can be used in an applied way as a case study in section 4.3 shall demonstrate. Since the end-users will operate this simulation tool among other applications in a predetermined way, a graphical user interface is programmed, integrating Dymola just as a model editor and simulator, while focusing more on the needs of a project engineer. So, case studies and post-processing operations can be undertaken conveniently with use of an integrated data base and chart utility.

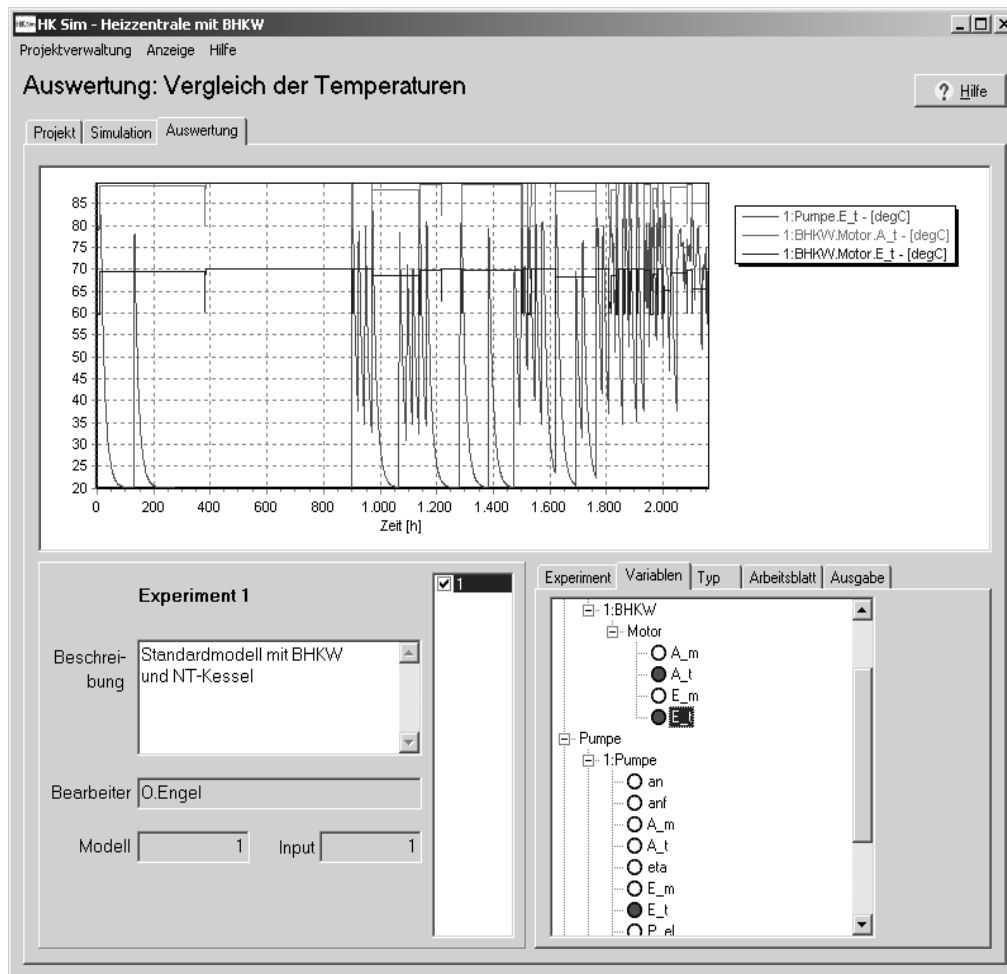


Figure 13: Screen shot of HKSIM's post processing window

7 Acknowledgement

The measurement data used in the simulation of the thermal power plant has been kindly provided by Imtech Contracting. Especially, I would like to thank Bruno Lüdemann and Ole Engel, who is developing the graphical user interface, for their help and advice throughout this project. I have to thank Admir Hadžikadunić for his work on modeling combined heat and power plants during his diploma thesis. Thanks must also go to Hicham Nabil for developing concepts with regard to model complex cooling systems in his diploma thesis.

References

- [1] Glück, B.:
Zustands- und Stoffwerte: Wasser, Dampf und Luft
2. Aufl., Verlag für Bauwesen, Berlin, 1991.
- [2] Hadžikadunić, A.:
Simulation von Blockheizkraftwerken mit Modelica / Dymola
Diploma thesis, Department of Technical Thermodynamics, Technical University Hamburg–Harburg, 2001.
- [3] Modelica Standard Library
www.modelica.org/libraries.shtml, 2002.
- [4] Recknagel, Sprenger, Schramek:
Taschenbuch für Heizung und Klimatechnik
69. Aufl., R. Oldenbourg Verlag, München /Wien, 1999.

Session 8a

Discrete Event Modeling

Using Modelica for Testing Embedded Systems

**Wolfgang Freiseisen, Robert Keber, Wilhelm Medetz,
Petru Pau, Dietmar Stelzmueller**

[wfreisei,rkeber,ppau]@risc.uni-linz.ac.at
[wilhelm.medetz,dietmar.stelzmueller]@scch.at

Abstract

In this paper, we give an overview of a simulation environment based on Modelica, dedicated to testing PLC programs. The main components of the system are a compiler of a Modelica subset and a runtime environment, which provides the necessary tools for simulating the evolution of models.

Introduction

Due to the high complexity of embedded software systems, it is more and more desirable to provide programmable logic control (PLC) programmers with *virtual test environments*. Usually, tests of the complete software, including PLC programs, high-level task control and human-machine interface (HMI) visualization, can only be performed when the mechanical environment, which is controlled by the software, has been finished.

This is the reason why a *simulation environment* based on Modelica has been developed. We describe in this paper a simulation system that handles models written in Modelica and uses C++ as intermediate language. Developed in the frame of the project VirtMould, supported by RISC institute and company ENGEL from Schwertberg, Austria, the compiler was meant to provide a tool for simulating *injection-molding machines*. The compiler accepts only a subset of Modelica language: this subset suffices for obtaining a model that simulates faithfully an injection-molding machine.

In order to minimize the necessary simulation modelling time, Modelica descriptions of many components are generated automatically from CAD models. Models are translated first to XML, the resulting files containing, in fact, the syntactic structure of the Modelica programs. The XML files are further parsed and provide the input for a pushdown automaton, which creates the internal data structures that store the essential content of the future C++ classes.

This C++ code is compiled and linked to specific simulation libraries; the resulting software component – a dynamic-link library or a static library – can be linked to external tools for visualization or process simulation.

In fact, the runtime environment is interfaced with a simulation of the embedded software environment. It allows an almost real time execution and simulation of the embedded software. In addition,

an open, OPC (OLE for process control) based interface for visualization and monitoring of the process is provided.

As we have already mentioned, the stable version of our product accepts only a subset of Modelica language. Thus, among the restricted classes, only blocks are currently translated, and the syntax is restricted. A version that can handle Modelica models is currently in the testing phase.

The paper is structured as follows: We begin by giving a short overview of the project. The architecture is detailed in Section 2. In Section 3 we present the main features of the C++ generated code. Section 4 describes the interface provided by the software component obtained after compiling the model. In Section 6 we give some examples and snapshots of the visual interfaces.

1. Project overview

ENGEL is a leading manufacturer of injection moulding machines, producing and selling integrated flexible manufacturing cells. A typical manufacturing cell consists of:

- an injection-moulding machine, whose individual components are selected from a wide variety of available product features;
- a handling system built upon a free programmable robot.

The whole manufacturing cell is controlled by an *embedded software system* that integrates an IEC 1131 based PLC, a high-level task-coordination language, a Java-based HMI and communication components for manufacturing execution system (MES) integration.

The software engineers and service technicians should have the possibility to perform software tests *offline* on their desktop with a “virtual injection-moulding machine”, whenever a new PLC is produced. In order to achieve a high user acceptance, a *test environment* (see [6] for a description of the main features of such software components) must be closely integrated with the development environment used by the PLC programmers; also, the PLC programmer should not be bothered with building simulation models for his specific target machine. Therefore the simulation models used for testing must be automatically generated from CAD designs.

The main goal is to increase software quality through simulation-based testing without increasing the

time spent for testing. In addition to this, *VirtMould* should also be applicable for other application domains, like *customer support offline diagnosis*, *computer based training* or *sales support*.

2. System architecture

VirtMould environment contains four major components:

- automatic generation of a simulation model;
- programming environment;
- runtime environment;
- visualization.

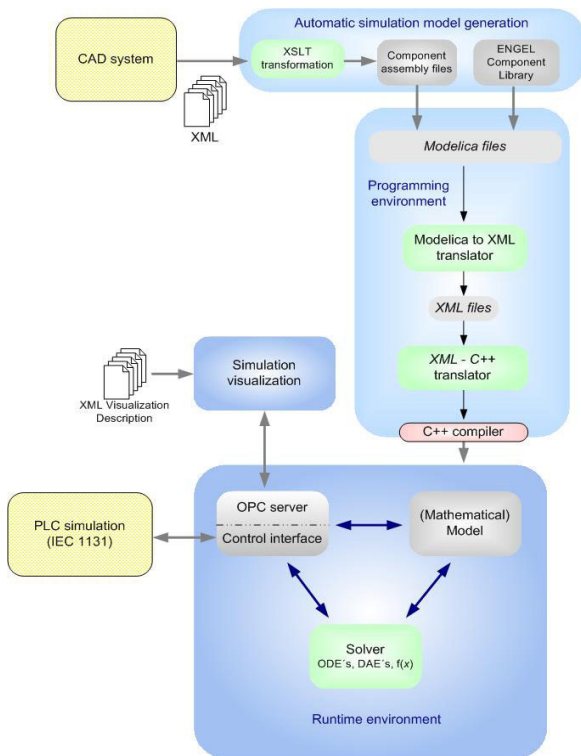


Figure 1: System architecture of the VirtMould environment.

Automatic generation of the simulation model

In order to minimize the effort for building simulation models, XML-based tools have been developed for translating CAD files (containing electrical, hydraulic, etc. components), together with information about product configuration, to Modelica. These tools work with *XLST transformations* based on the XML files exported from CAD, and generate *component assembly files*.

A library containing the *mechatronic blocks* and *handling system components* of the injection-moulding machine has been developed and is continuously improved and extended (*ENGEL Component Library*). The goal has not been to achieve the highest possible simulation accuracy, but rather to provide the accuracy required for software testing, together with a high degree of flexibility and fast simulation execution. This library also contains components for interactive test manipulation, interaction with the panel of the virtual machine, and simulation of machine failures.

Programming Environment

A compiler for a Modelica subset has been implemented. The focus of this compiler was to provide an easy integration into the overall system architecture and to allow the efficient simulation of discrete events. This is done in two phases: The *Modelica - XML translator* parses the Modelica files and generates a XML representation, and the *XML - C++ translator* generates C++ class files. Details on the Modelica compiler are presented in Section 3.

Runtime Environment

The runtime environment is loaded by the PLC program simulation and contains some major components:

- the *model* is the C++ collection of classes corresponding to the Modelica program;
- the *solver* is the C++ framework for computing trajectories of the variables of the model;
- there are two kinds of interfaces for controlling the simulation and for data exchange:
 - the *control interface* is a COM in-process DLL interface, which has been chosen for coupling the PLC program simulation with the Modelica simulation runtime environment. The PLC program simulation allows program execution in either *soft real-time* or *virtual simulation time* mode.
 - for accessing the simulating state we have added an *OPC server*. A more detailed description is given in Section 4.

Visualization

In addition to using third party visualization tools, a built-in configurable user interface allows the design of *virtual hardware panels*. The user has the possibility to:

- inspect all model variables in a hierarchical manner;
- monitor some variables by drawing their trajectories;
- change the values of some variables, provided that those variables permit this kind of user interaction.

The user interface can be configured through an XML file.

3. Short description of the Modelica compiler

The compilation of a Modelica model comprises three steps. After each step, one or more files of specific types are generated. Thus:

1. The model is parsed and the syntactic structure is stored in an XML file. In the same format (XML) are stored the libraries.
2. One or more XML files are taken as input by a program that produces C++ code. This code mirrors the Modelica code, a Modelica

class having a C++ counterpart, which is a C++ class.

3. The C++ file is compiled, and the object is linked with some libraries, which provide the framework for solving the model, i.e., generating a simulation, and for interfacing with other software tools.

In the following we focus on the characteristics of the C++ code.

Base classes

C++ as intermediate code was a normal choice, due to the object-oriented philosophy of Modelica (see Modelica tutorial [1]). A Modelica class is translated into a C++ class. Among the restricted Modelica classes, our system accepts **type**, **block**, **connector** and **package**. Models are accepted with some restrictions.

To each component of a Modelica class corresponds a member data in its C++ image. According to the specifics of the Modelica class, a number of other members are included in the C++ code, like, e.g., one additional member for each differentiated variable.

A Modelica block and its C++ image are shown in Figure 1. We have kept in the C++ code only the member declarations and function definitions that correspond directly to their Modelica counterparts.

The C++ correspondent of a Modelica **package** is derived from a special class, which has no method with equations but can contain a number of inner classes.

The **connectors** provide a set of template member functions, in order to allow connections with components of various types.

We must emphasize that the granularity of Modelica code is preserved: The models are described hierarchically in Modelica, by giving the mathematical description of components and connecting them in composite objects; this structure is transmitted to the C++ code.

Equations

The *equations* that describe the behavior of a component generate a few member functions. Thus:

1. *discrete* equations are collected in a member function that is executed only when special events occur;
2. *differential* equations go to another member function;
3. the remaining equations are divided in two groups, depending on the relation of their variables with differential equations. Thus, we call *dynamic* these equations that are related to differential equations, and
4. *independent algebraic* the other ones.

These methods are easily generated for blocks – provided that the Modelica code contains all equations in explicit form. If *flow* variables occur and Kirchhoff laws have to be generated, implicit equations should be added and the translation requires further processing. Moreover, for equations that cannot be explicitized (like transcendental equations, or polynomial of high degree), numerical solvers are required. These features will be covered in a future version of our system.

Events

The set of discrete equations is solved only when some *events* occur. During the simulation, an event queue is maintained and used for triggering the evaluation of these equations.

Changes of values of discrete variables usually generate events; also, the special functions *sample* and *edge* are event-generators. On the other hand, events can be generated *externally*, by the interaction of the user with the visual interface: Recall that the output of our system is a *software component* that can be embedded or attached to other software tools; this component communicates with the environment in two ways, that is, it exposes names, types and values of variables, and it accepts, with some restrictions, modifications of these variables.

Simulation

A *simulation* consists in the computation of trajectories of variables during a specific time period. In fact, a finite set of points on these trajectories is computed, for values of time discretized by a *time step* whose value is set externally.

For solving differential-algebraic equations we have implemented a few numerical integrators: Euler, Runge-Kutta, Runge-Kutta with variable step size (see [4], [5] for detailed descriptions of these methods). In order to ensure a higher stability, the integrators have an *internal time step* (fixed or variable), which is usually much smaller than the external. The differential and dynamic equations are evaluated after each internal time step, whereas the independent algebraic equations are evaluated after the external time step.

Basically, the simulation is performed following the same rules as described in the Modelica language specifications (see [2], Chapter 4): The integrator solves numerically the equations between two events. When an event occurs, the set of discrete equations is solved and any change of the values of the discrete variables can influence the set of differential and algebraic equations.

Ordering their enclosing blocks gives the order in which the equations are evaluated, so that every variable that occurs in the right-hand side of an equation has already been given a value before the equation is evaluated.

```

block Sine
  parameter Real amplitude[:]={1.};
  parameter Slunits.Frequency freqHz[:]={1.};
  parameter Slunits.Angle phase[:]={0.};
  parameter Real offset[:]={0.};
  parameter Slunits.Time startTime[:]={0.};

  extends Interfaces.MO(final nout=max((size(amplitude, 1); size(freqHz, 1); size(phase, 1);
    size(offset, 1); size(startTime, 1))));

protected
  constant Real pi=Modelica.Constants.pi;
  parameter Real p_amplitude[nout]=(if size(amplitude, 1) == 1 then ones(nout)*amplitude[1] else amplitude);
  parameter Real p_freqHz[nout]=(if size(freqHz, 1) == 1 then ones(nout)*freqHz[1] else freqHz);
  parameter Real p_phase[nout]=(if size(phase, 1) == 1 then ones(nout)*phase[1] else phase);
  parameter Real p_offset[nout]=(if size(offset, 1) == 1 then ones(nout)*offset[1] else offset);
  parameter Slunits.Time p_startTime[nout]=(if size(startTime, 1) == 1 then ones(nout)*startTime[1] else startTime);
equation
  for i in 1:nout loop
    y[i] = p_offset[i] + (if time < p_startTime[i] then 0.
      else p_amplitude[i]*Modelica.Math.sin(2*pi*p_freqHz[i]*(time - p_startTime[i]) + p_phase[i]));
  end for;
end Sine;

```

```

class Sine : public Interfaces::MO {
  // Variables ...
public:
    ParamVectorN<Real>          amplitude;
    ParamVectorN<Slunits::Frequency> freqHz;
    ParamVectorN<Slunits::Angle> phase;
    ParamVectorN<Real>          offset;
    ParamVectorN<Slunits::Time> startTime;

  // protected members
    ConstScalar<Real>          pi;
    ParamVectorN<Real>          p_amplitude;
    ParamVectorN<Real>          p_freqHz;
    ParamVectorN<Real>          p_phase;
    ParamVectorN<Real>          p_offset;
    ParamVectorN<Slunits::Time> p_startTime;

  // methods
public:
    Sine() { // Default Constructor....
      // ... initialization of parameters with constant vals
      // ...
    }
    void resize() { // resizing vector components
      p_amplitude.resize(nout);
      p_freqHz.resize(nout);
      p_phase.resize(nout);
      p_offset.resize(nout);
      p_startTime.resize(nout);
    }
    void init() {
      // initializing parameters with non-constant expr.
      //
      // initializing nout
      nout.parameter(max ( size ( amplitude,1 ) ,
        size ( freqHz,1 ) ,size ( phase,1 ) ,
        size ( offset,1 ) ,size ( startTime,1 ) ), 1);
      MO::init();
      //...
      resize();
    }
}

```

```

void start() { // Settings for start
  MO::start();
}

// ...

void equation_dyn(const Time &time){
  // Dynamic Equations for this Block
  MO::equation_dyn(time);
  {
    Integer _initialCond;
    Integer _finalCond;
    _initialCond = 1;
    _finalCond = nout;
    for (int i=_initialCond;i<=_finalCond;i++){
      Real _if5;
      if (TimeLt(time, p_startTime [ i - 1 ] )){
        _if5 = 0.;
      } else {
        _if5 = p_amplitude [ i - 1 ] *
          Modelica->Math->sin (
            2*pi*p_freqHz [ i - 1 ] *
            ( time-p_startTime [ i - 1 ] ) +
            p_phase [ i - 1 ] );
      }
      y [ i - 1 ] = p_offset [ i - 1 ] + ( _if5 );
    }
  }
  return;
}

// ...
}; // end class: Sine

```

Figure 2: A Modelica block and its C++ translation.

4. Communication with other programs

The runtime system contains two interfaces for connections with other programs:

- a proprietary COM interface for communicating with the PLC program simulation;
- an open OPC-based interface for communication with visualization tools.

After compiling the initial Modelica text, a *library* (.lib file) or a *dynamic-link library* (.dll file) is generated. This library encapsulates both the model description and the integration algorithms. Among the possible services provided by this library we mention: start/stop a simulation process, transmit the names, types and values of all the internal variables of the model, at every time moment during the simulation run, and accept new values for some of the variables.

Interaction with other programs is realized through a common interface: we have used the COM (*Common Object Mode*) technique. A COM interface component for loading and unloading simulations, calculating time steps and exchanging data with the simulation has been designed.

For visualizing the results of the simulation an OPC (*OLE for process control*, see [3] for more information) server has been developed and implemented. The simulation results are provided via OPC and these values can be visualized with standard OPC client tools. OPC is the most commonly used standard for inter-process communication in the area of manufacturing automation.

The OPC specification is a non-proprietary technical specification that defines a set of standard interfaces based upon Microsoft OLE/COM technology. An OPC standard interface makes possible interoperability between automation/control applications, field systems/devices and business/office applications.

Traditionally, each software or application developer was required to write a custom interface, or server/driver, to exchange data with hardware field devices. OPC eliminates this requirement by defining a common, high performance interface that permits this work to be done once, and then easily reused by HMI, SCADA (Supervisory Control and Data Acquisition), control and custom applications.

The advantage of using the OPC Data Access Specification is that it provides a hierarchically structured namespace that can be directly used to map Modelica variables. Clients can then retrieve OPC items (i.e. Modelica variables) either synchronously or asynchronously. OPC also provides possibilities to specify the desired update rates for items and to browse the available item name space. The OPC server defines the access status (read, write) for each item together with additional descriptive information. As OPC is implemented by a COM object, the inter-process communication can be realized either as an highly efficient in-process communication or as distributable (DCOM) out-of-process communication. Also several clients can communicate in parallel with one OPC server.

5. Usage and examples

The following figure shows a standard ENGEL injection moulding machine. The ENGEL HL is a highly accurate, fast and energy-saving injection moulding machine in tiebarless design for use in the range from 200 to 6,000 kN clamping force.



Figure 3: An injection moulding machine.

After a thorough analysis of the machine structure, a simulation model written in Modelica has been developed, which should describe its components with an appropriate accuracy for testing its general behavior. As we have already mentioned, Modelica description of many components has been automatically generated from CAD specifications.

An excerpt of the main Modelica model of the injection model machine is given in the following. In the instantiation sector of this model all the functional units of the machine are defined.

```

block InjectMoldMachine
  "Tiebarless Injection Molding Machine"
  Lib.IMM.FunctionalUnits.MainPowerSupply MainPower;
  Lib.IMM.FunctionalUnits.ControlVoltages ContrVolt;
  Lib.IMM.FunctionalUnits.EmergencyOff EmergOff;
  Lib.IMM.FunctionalUnits.FilterMotor FilterMotor1;
  Lib.IMM.FunctionalUnits.Motor Motor1;
  Lib.IMM.FunctionalUnits.SafetyGateMoldFront SGMoldFront;
  Lib.IMM.FunctionalUnits.SafetyDoor SafetyDoor;
  Lib.IMM.FunctionalUnits.SafetyGateInject SGInject;
  Lib.IMM.FunctionalUnits.Heating
    Heat1 (startTemp = 30.);
  Lib.IMM.FunctionalUnits.TraverseCooling Cool1;
  Lib.IMM.FunctionalUnits.PumpBlock Pumps;
  Lib.IMM.FunctionalUnits.Ejector
    Ejector1 (startPos = 0.3);
  Lib.IMM.FunctionalUnits.Core
    Core1 (startPos = 10.),
    Core2 (startPos = 10.);
  Lib.IMM.FunctionalUnits.InjectionUnit
    InjUnit1 (startPos = 5.);
  Lib.IMM.FunctionalUnits.Mold
    Mold1 (startPos = 3.);
  Lib.IMM.FunctionalUnits.InjectionPlasticize
    InjPlast1 (startPos = 1.);
  Lib.VMLib.Electrics.AlarmLamp Alarm;
  PLC_Interface PLC_IO;
  ButtonBlock Buttons;
equation
  // input connections for "SafetyDoor"
  connect(ContrVolt.VAC24, SafetyDoor.VAC24);
  connect(ContrVolt.VE24, SafetyDoor.VE24);
  connect(Buttons.SafetyDoor.outPort, SafetyDoor.HandleGate);
  connect(Buttons.QuitKey.outPort, SafetyDoor.QuitKey);
  // ...
end InjectMoldMachine;

```

The simulation model contains all components necessary for characterizing the behavior of the machine, including its electrical parts (e.g. power supply with different control voltages), hydraulic movements (e.g. ejector, mold) and interaction with the user (e.g. open/close safety gate).

Currently, the typical testing environment consists of the following windows:

- HMI of the injection moulding machine; this graphical object has the appearance and functionalities of the touch screen of the real machine;

- Graph window: for visualizing simulation results of selected variables;
- Control panel: buttons and switches for opening/closing safety gates, moving hydraulic units, etc.; this object is a faithful copy of the control panel of the injection moulding machine;
- PLC simulation window: gives internal information of the PLC runtime environment.

These components can be seen in Figure 4.

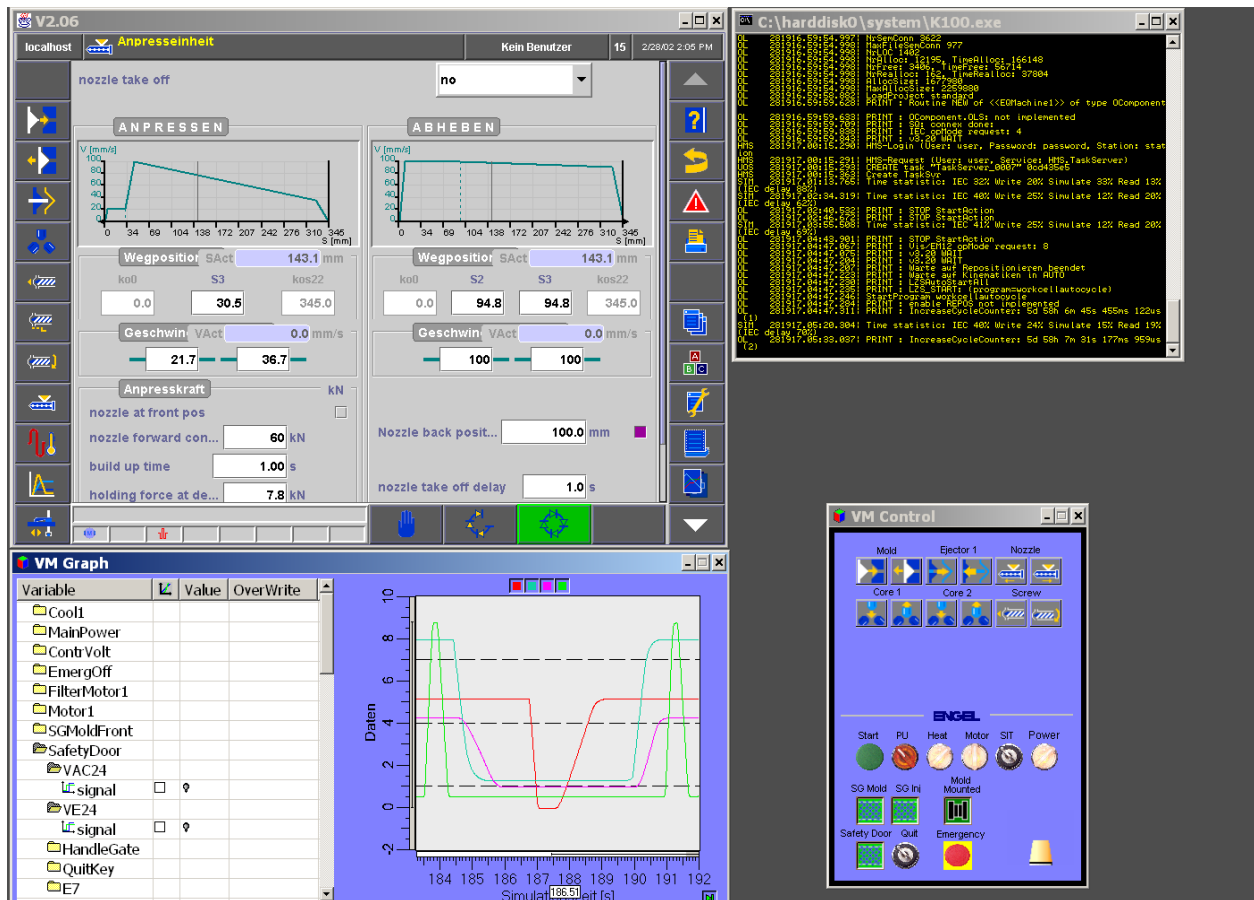


Figure 4: Screenshot from a simulation session.

6. Conclusions and future work

The quality of the overall system architecture of the VirtMould test environment has been proved by a high user acceptance. About 10 PLC programmers in their daily work are currently using the environment; this number should be increased to 50 or more users, testing more than 1000 systems per year. Modelica has proved to be the ideal object oriented modelling language for building reusable libraries of simulation components, which is essential for automatic model generation.

The next step will be to add a test automation framework, such that *regression tests* can be performed automatically. An SVG (Scalable Vector Graphic) based visualization client will provide enhanced 2D animation of the simulation process. A 3D mechanical component visualization will also provide the possibility for collision checking.

In addition to software testing, the environment will also be used for computer-based training.

References

- [1] *Modelica™ – A Unified Object-Oriented Language for Physical Systems Modeling*, Tutorial, Version 1.4, by **Modelica Association**, Dec. 2000, downloadable from <http://www.modelica.org>.
- [2] *Modelica™ – A Unified Object-Oriented Language for Physical Systems Modeling*, Language Specification, Version 1.4, by **Modelica Association**, Dec. 2000, downloadable from <http://www.modelica.org>.
- [3] F. Iwanitz, J. Lange - *OLE for Process Control*, **Huethig GmbH**, Heidelberg, 2001.
- [4] U. M. Ascher, L. R. Petzold – *Computer methods for ordinary differential equations and differential-algebraic equations*, **SIAM**, 1998.
- [5] K. E. Brennan, S. L. Campbell, L. R. Petzold – *Numerical solution of initial-value problems in differential-algebraic equations*, **SIAM** 1996.
- [6] C. Kaner, J. Falk, H. Q. Nguyen - *Testing Computer Software*, **John Wiley & Sons**, 1999.
- [7] M. Fewster, D. Graham - *Software Test Automation*, **Addison-Wesley**, 1999.

Combining Discrete Event Models and Modelica – General Thoughts and a Special Modeling Environment

Manuel A. Pereira Remelhe

Process Control Laboratory
Department of Chemical Engineering
University Dortmund, D-44221 Dortmund
Tel.: +49/231/755-5127, Fax: +49/231/755-5129
Email: m.remelhe@ct.uni-dortmund.de

Abstract

This contribution consists of two parts. In the first part general possibilities for the combination of Modelica models and discrete event models are discussed on a conceptual level. It is shown that it is necessary to support asynchronous behavior and that it is useful to represent sampled data behavior of discrete event systems in an interrupt-driven style for fast simulation. The characterizations of the alternatives are summarized in table 1.

In the second part a modeling environment prototype that provides dedicated editors for different discrete event formalisms and supports hierarchical and heterogeneous models is presented briefly. It transforms a discrete event model into a Modelica class whose behavior is given by a Modelica algorithm and several Modelica functions. Such a discrete event component can be inserted intuitively into a model of a physical system and simulated by standard Modelica-Tools.

Introduction

Sophisticated technological systems often require complex discrete event control. For instance, sequential control is needed for the execution of recipes on chemical batch plants. Redundancy control is crucial for the safety of aircraft. And resource booking systems are needed for coordinating several interacting sequential controllers, e.g., to avoid collisions of robots or to prevent the mixing of parallel running batches in chemical batch plants. These discrete parts often involve hierarchical execution schemes as well as concurrency with synchronous and asynchronous communication. The physical part of such technological systems normally is very large and include complex hybrid dynamics such as Friction, collisions and instantaneous equilibrium reactions.

If simulation is to be used for the estimation of throughput, power consumption, quality quantities

etc. the simulation model has to incorporate the discrete event as well as the physical part of the system. Hence, powerful modeling formalisms are required for both, the physical and the discrete part, and an intuitive integration of these parts has to be supported. In addition to the clearness of the modeling paradigm the simulation efficiency is an important aspect.

For the modeling and simulation of general hybrid physical systems Modelica [1] is particular suitable, because it facilitates multi-domain models, allows intuitive modeling of the most hybrid phenomena and enables efficient simulation. Furthermore it is possible to combine physical models with complex discrete event dynamics. In this contribution 4 general approaches for the integration of physical models and discrete-event models are discussed:

1. declarative style (based on equations),
2. imperative style (based on Modelica algorithms and functions),
3. external secondary simulator (based on the external function interface) and
4. external primary simulator (Modelica in the loop).

One well known example for the first approach is the Petri nets library created by Mosterman, Otter and Elmqvist [2]. The Petri net modeling objects, i.e., places and transitions, are represented by library components. These can be instantiated and connected via the connectors to constitute a specific Petri net graph. The equations of the transition and place objects were specified in a way such that the composition of these equations behaves like a Petri net would do.

The second approach implies that the behavior of a discrete event system is encoded imperatively into one Modelica algorithm that may call Modelica functions. In order to support high-level modeling formalisms a special modeling environment has to

be implemented that provides dedicated editors for specific discrete event formalisms and translates automatically a discrete event model into a Modelica component containing such an algorithm. A prototype of such a modeling environment will be presented in the second part of this contribution.

The third and fourth approaches allow the usage of existing discrete event simulators. In the third case a Modelica simulator controls the external simulator, whereas in the fourth case the Modelica simulator is controlled from outside, e.g., by a Stateflow-Simulink model.

Part I: General Thoughts

Interrupt-driven Models

The task of discrete event systems is normally to wait for certain state events in the physical system and react instantly when they occur, e.g., when a predefined temperature is reached, the next step of a recipe is started. Hence, from a functional point of view discrete event systems are interrupt-driven, or to be more precise, driven by state events (and sporadic time events).

A prerequisite for realizing this behavior is the detection and localization of state events during continuous integration. Since integration methods require continuous model equations, in Modelica all inequality expressions of a model are fixed during integration in order to guarantee that discontinuous changes of variables do not occur. In addition, inequality expressions which depend on continuous state variables and which are critical, are monitored during integration. When the logical value of an inequality expression changes, the time instant of the switching point is determined up to a certain precision and the integration is stopped, i.e., a state event is localized. In the case of time events the inequality expression depends only on the time variable and the integration simply stops directly at the predetermined time.

Modelica-Tools such as Dymola [3] support this event handling, if the events can be deduced from the Modelica code, i.e., the inequality expressions must be included in the code. Therefore an interrupt-driven model of a discrete event system can only be realized accurately with the first two approaches. If an external simulator is to be used, a wrapper code could be written in Modelica that defines the externally caused state events. But that would not be feasible for very complex discrete

event models and the advantage of using an approved discrete event simulator would get lost due to error-prone hand coding. However, the fourth approach allows to detect the state events at the end of an integration step without using event localization, so that the event instants depend on the step size of the integrator.

Sampled-data Models

Discrete event systems are normally implemented as sampled-data systems. In the most cases the sampling rates are very high, so that the sampling can be neglected which results into interrupt-driven models. However, sometimes it is necessary to consider the sampling. The simplest way to do so is to use the `sample`-operator. In the following example a tank is filled continuously with the rate 1. A controller with 100 samplings per second opens the outlet when the tank level becomes higher than 10 and it closes the outlet when the tank level becomes lower than 0.1 .

```

model ControlledTank
  Real level;
  Boolean valveOpen;
equation
  der(level)= if valveOpen then
    1 - sqrt(level*2.173) else 1;
  when sample(0, 0.01) then
    valveOpen =
      if level > 10 then true
      else if level < 0.1 then false
      else pre(valveOpen);
  end when;
end ControlledTank;

```

The sample-Operator generates regular time events each 0.01 seconds. At every time event the integrator is stopped, the switching equation is evaluated and the integrator is started again. Since the initialization of the integrator consumes the largest amount of time, the simulation can be speed up significantly by transforming this sampled-data model into an interrupt-driven model that emulates the sampled-data behavior. This is done by replacing the `when` clause by the following two `when` clauses:

```

when {level<0.1, level>10} then
  sampleTime=(floor(time*100)+1)/100;
end when;
when time > sampleTime then
  valveOpen = if level>10 then true
              else if level<0.1 then false
              else pre(valveOpen);
end when;

```

In the first **when** clause an effective sampling time is determined, when a significant state event occurs. It is the next regular sample time the **sample** operator would have. In the second **when** clause then the switching equation is evaluated when the sampling time is reached. While no state events occur, the integration does not stop. For the given example the simulation of the interrupt-driven model is 70 times faster than the simulation of the sampled-data model. In consequence, the **sample**-operator should not be used for reactive discrete event systems. Instead, it is always possible to emulate sampled-data behavior, if required.

Asynchronous Behavior

In the first approach the modeling objects of a formalism are represented by Modelica objects whose behavior is defined declaratively using equations.

These equations are treated in the same way as the equations of the physical systems part so that a discrete transition of the discrete event model is connected to a complete evaluation of the whole system of equations including the physical systems equations. This synchronous behavior is disadvantageous when a formalism is used that performs a sequence of intermediate transitions in order to achieve a consistent state. Here, the effective transition that should be observable from the physical system incorporates several internal transitions that are asynchronous to the those of the physical system. Otherwise inconsistent intermediate states of the discrete event model could have an illegal effect on the physical system. Furthermore complex discrete event systems often involve complex hierarchical execution schemes: on each level a process can include sub-processes, and concurrent subsystems may run asynchronously. Therefore, it is crucial to support asynchronous behavior for modeling complex discrete event controllers. Hence, the equation-based approach is only suitable for simple cases where as the other three approaches can realize such complex behavior due to the algorithm-based imperative definition.

Other Aspects

Modelica algorithms are more limited in comparison to real programming languages, because Modelica's data structure is static, i.e., it is not possible to instantiate new variables or components at run-time. Therefore all variables that might be needed possibly at run-time have to be installed at compile-time. For example an event list of a scheduler of a discrete event simulator has to be

realized by a fixed length vector in principle. If during a simulation the number of elements for the event list exceeds the length of the vector, the simulation has to abort.

Some discrete event systems have only discrete-time input signals. In this case state or time events are generated outside of the discrete event component, e.g., by limit switches in the physical system, and the usage of an external simulator is straightforward.

The models created with the Petri nets library look pretty much like Petri nets. The difference to the original formalism is that there no ports, so that the objects can be connected directly with lines and therefore only two classes of objects are needed. However, many formalisms have more complex syntax and graphics that can not be represented adequately using object-oriented composition diagrams. Statecharts for example has a state hierarchy concept without encapsulation of inner states. Consequently, dedicated graphical editors should be used in general. The following table summarizes the characterizations of the 4 approaches.

Table 1: Characterization of the 4 approaches

Approach	1	2	3	4
allows interrupt-driven models (localization of state events)	x	x	-	-
allows asynchronous behavior (hidden iterations)	-	x	x	x
allows dynamic data structures	-	-	x	x
adequate graphics	-	x	x	x
heterogeneous discrete event models	?	x	?	?
reliable discrete event simulation	?	?	x	x

Part II: A Prototype of a Modeling Environment

In order to enable heterogeneous discrete event models including domain-specific formalisms a modeling environment prototype has been developed that provides dedicated editors for the different formalisms. The environment is based on the meta-modeling tool DoME [4]. This tool generates automatically graphical editors based on a formal specification of the syntax and on parameters for the graphical appearance of the formalism. Therefore prototypes of new editors can be implemented within a few hours and a final version can be achieved within a few days. This is important for further development of domain-specific formalism.

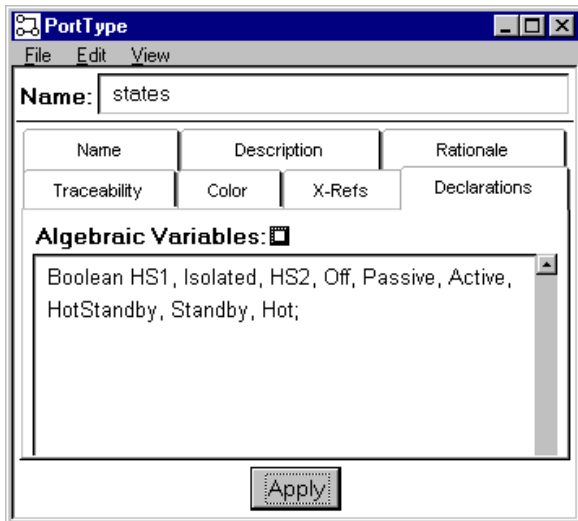


Figure 1: Defining a port type named "states" using Modelica syntax

An hierarchical block diagram formalism is implemented that allows the basic blocks to be modeled with any reactive formalism. At present, state-charts [5] and sequential function charts [6] are integrated, but other formalisms can be added in future.

Hierarchical Block Diagram Formalism

In the framework of the block diagram formalism an archetype concept is used. Each archetype defines a specific class of objects and can be instantiated several times. The definition of an archetype incorporates the ports of the objects (fig. 3), archetype attributes, object attributes and one or more alternative internal implementations, i.e., different interchangeable realizations. In order to cope with the possible combinations of different implementations one or more configurations can be defined that determine unambiguously which implementation is used for which instance.

In our case port types and block types can be defined. Port types are archetypes with archetype attributes (port variables and sub-ports), but without an implementation (fig.1). Ports can be instantiated within block archetypes and port archetypes. The block archetypes contain ports and no further attributes. The implementations of a block archetype can be a block diagram or a statechart or a sequential function chart (fig. 2 and 4). A block diagram contains the outer ports of the corresponding archetype and sub-blocks adorned with their own ports. In a block diagram one output port can be connected to several input ports, but one input port must not be connected with more than one output port (fig. 2).

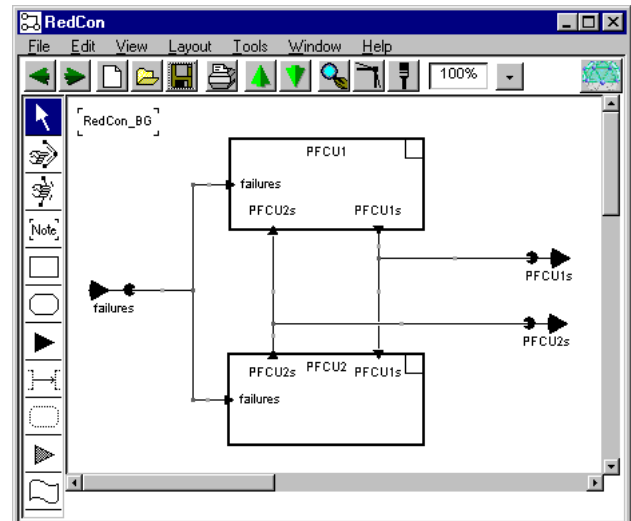


Figure 2: A block diagram implementation named "RedCon_BG"

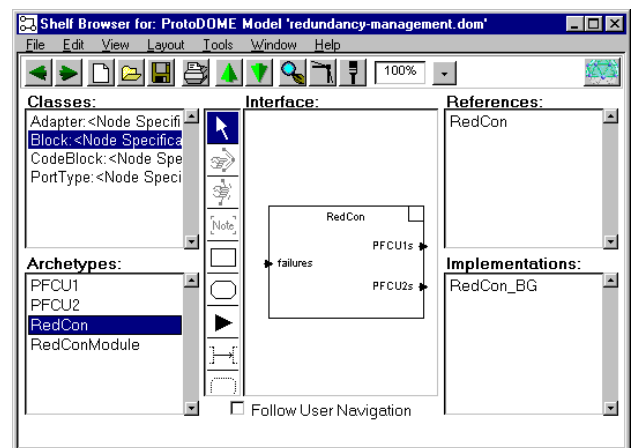


Figure 3: The shelf view on the archetypes

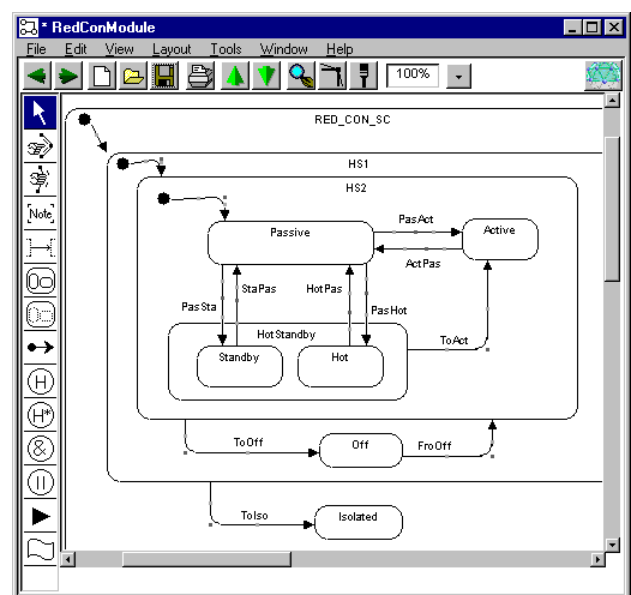


Figure 4: A statechart model that represents an implementation of a block

The Translation into Modelica

The translation procedures for the different formalisms are programmed using the extension facilities of DoME: The *Scheme* variant *Alter*, a *Lisp* like functional programming language, and *Small-talk* in which DoME itself is implemented.

For the translation of a hierarchical model a specific implementation configuration is used. All parts that are automatically generated are encapsulated in one Modelica model class. The connectors of this class correspond to the outer ports on the highest model level. Each port type is directly represented by a connector definition. For each block implementation a record is generated that defines the components (variables and/or sub-blocks) of the corresponding implementation. Since the instantiation of such a record is connected to the instantiation of its components, the instance of the top-level block contains the whole data structure of the hierarchical model.

```
// block1 contains block2 and block3:
record Data_block1
  ...
  data_Block2 block2;
  data_Block3 block3;
  ...
  Boolean trig;
end Data_block1;
```

In addition for each block implementation a set of Modelica functions is generated that define its behavior. The following tasks are realized by single Modelica functions:

- initialization of the components (variables and sub-blocks) of the block implementation
- detecting the need for state transitions based on given input values
- performing state transitions based on given input values
- data logging for visualization

For a specific block implementation the argument type and the output type of these functions is the corresponding record. Hence, the initialization function of the top-level block gets its data object (a record) as an argument and calls the initialization function of its sub-blocks using the corresponding sub-objects included in the record:

```
function init_block1
  input Data_block1 par;
  output Data_block1 data;
algorithm
  data := par;
  data.block2 :=
```

```
    init_block2(data.block2);
  data.block3 :=
    init_block2(data.block3);
  ...
end init_block1;
```

Finally, the generated Modelica model class contains an algorithm that originates all function calls:

```
model DiscreteController
  < Port Definitions >
  < Record Definitions >
  < Function Definitions >
  InputPort inputConnector;
  OutputPort outputConnector;
  Data_block1 block1;
algorithm
  block1.inputPort := inputConnector;
  when initial() then
    block1 := init_block1(block1);
  end when;
  block1 := detect_block1(block1);
  if block1.trig then
    block1 := perform_block1(block1);
  end if;
  outputConnector :=
    block1.outputPort;
end DiscreteController;
```

Such a Modelica model of a discrete event component can be inserted intuitively into a model of a physical system and simulated by standard Modelica-Tools.

References

- [1] Modelica. Homepage: <http://www.Modelica.org/>
- [2] Mosterman P.J., Otter M. and Elmqvist H.: Modeling Petri-Nets as Local Constraint Equations for Hybrid Systems using Modelica. *Proceedings of the 1998 Summer Computer Simulation Conference (SCSC'98)*, Reno, U.S.A., 19.-20. Juli 1998.
- [3] Dymola. Homepage: <http://www.Dynasim.se/>.
- [4] DoME. <http://www.htc.honeywell.com/dome/>.
- [5] D. Harel: Statecharts: A Visual Formalism for Complex Systems. *Sc. of Comp. Prog.* 8, pp 231-274, 1987.
- [6] International Electrotechnical Commission. *International Standard IEC 1131 Programmable Controllers, Part 3, Programming Languages*. IEC, Geneva, 1993.
- [7] M. Otter, M. A. Pereira Remelhe, S. Engell, P. Mosterman, "Hybrid Models of Physical Systems and Discrete Controllers," *at - Automatisierungstechnik*, vol. 48, no. 09, pp. 426-437, 2000.
- [8] M. A. Pereira Remelhe: Simulation and Visualization Support for User-defined Formalisms Using Meta-Modeling and Hierarchical Formalism Transformation. *Proceedings of the 2001 IEEE International Conference on Control Applications*, México City, 2001

Hybrid Modeling of Communication Networks Using Modelica*

Daniel Färnqvist[†], Katrin Strandemar[†], Karl Henrik Johansson^{†,‡} and João Pedro Hespanha[§]

Abstract

Modeling and simulation of communication networks using Modelica is discussed. Congestion control in packet-switched networks, such as the Internet, is today mainly analyzed through time-consuming simulations of individual packets. We show, by developing a model library based on a recent hybrid systems model, that Modelica provides an efficient platform for the analysis of communication networks. As an example, a comparison between the two congestion control protocols is presented.

1 Introduction

The interaction with a variety of networks plays an important role in everyone's life. A growing use of networks, such as the Internet, with their widening set of services increases the demand on the control of the network. The objective is often to improve traffic throughput and to better accommodate different service demands. Communication networks experience major problems due to traffic congestion. Today's congestion control is in most networks implemented as end-to-end protocols, e.g., [4, 12, 10]. The protocols have proved to form the basis of a remarkably robust and scalable system, though the understanding of the basic principles of these complex systems is far from satisfactory. There is intensive research on modeling and simulation of the Internet. It has been pointed out that classical network models from telecommunication based on Poisson modeling are not suitable for the Internet [8]. Recent models capturing the self-similar nature of the traffic has been developed [5]. The general opinion in the network area is still that Internet modeling and simulation are open research problems [9].

*The authors want to thank Håkan Hjalmarsson and Gunnar Karlsson for helpful discussions.

[†]Dept. of Signals, Sensors & Systems, Royal Institute of Technology, SE-100 44 Stockholm, Sweden

[‡]Corresponding author. Tel. +46 8 7907321. Fax +46 8 7907329. kallej@s3.kth.se

[§]Dept. of Electrical & Computer Engineering, University of California, Santa Barbara, CA

The intention of this paper is to describe initial work on modeling packet-switched communication network using Modelica. The standard modeling and simulation environment targeted at networking research is the discrete event simulator *ns-2* [7]. *ns-2*, which was originally developed at UC Berkeley, implements network protocols such as the transmission control protocol (TCP) and traffic source behaviors such as file transfer protocol (FTP) and Telnet. Since *ns-2* directly implements the Internet protocols and simulates individual packets, it provides on one hand accurate simulation results but on the other hand a rather slow simulation speed. The result of this is that *ns-2* is mainly for studying relatively small networks over a short time scale. The other extreme is to use flow models, i.e., to approximate the packet transmission with a continuous flow and basically neglect the network protocols. Flow models capture average transmission rates but ignores events such as packet drops. Flow models are hence suitable for the study of steady-state behavior but not for evaluating transient phenomena. This is for instance due to that certain congestion control strategies are based on the implicit feedback information from packet drops, which are not included in the flow model. A hybrid systems model, which is based on the average rates but takes packet drops into account, was recently proposed in the literature [3]. The motivation for this model is to capture the network behavior on a time scale in between packet models and flow models. Studies have shown that the hybrid model is able to model many important network phenomena [3, 1]. In this paper, we will show that the hybrid model is suitable for Modelica. Moreover, we show that simulating the model in Dymola provides an efficient environment for studying congestion control in computer networks.

The outline of the paper is as follows. Section 2 presents a brief introduction to congestion control in communication networks. The hybrid model is described in Section 3 and its Modelica implementation is discussed in Section 4. An example, where two TCP versions are compared for a small wireless network, is given in Section 5.

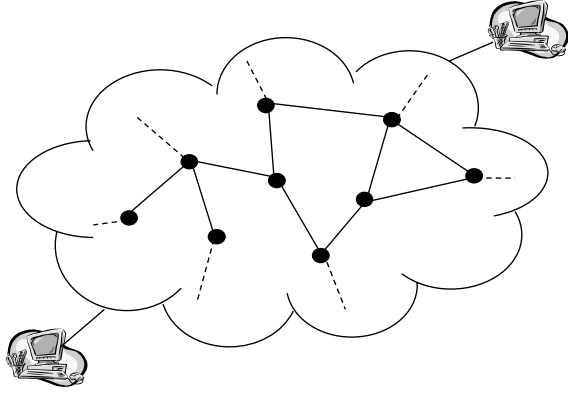


Figure 1: Communication network.

2 Communication Networks

A packet-switched communication network can be described by a directed graph. The nodes represent the routers, which direct the packets from sender to receiver, and the edges correspond to wired or wireless links. Figure 1 illustrates a connection with one sender and one receiver. The bandwidths of the links are limited, so each router has a buffer where packets are stored if more packets are entering the router than is going out. In this way, it is possible to deal with minor traffic congestion in the network. If too many packets enter a router in a short amount of time, however, packets will be dropped due to that the buffer has finite size. The way this congestion problem is handled by the senders on the Internet is through a control mechanism denoted transmission control protocol (TCP). The receiver sends acknowledgments back to the sender, when packets have arrived. In order to efficiently use the network resources, the TCP sender adjusts its sending rate according to a control variable called the window size w . The TCP sender sends w number of packets and waits for acknowledgments for them to return. Hence, w corresponds to the number of unacknowledged packets the sender may have in the network. When the sender has received acknowledgments for all w packets, w is increased by one. If a packet is dropped (so that no acknowledgment for that packet is received), w is decreased by a factor two. Hence, TCP uses additive increase and multiplicative decrease (AIMD) to regulate the congestion window size based on explicit acknowledgments and implicit negative acknowledgments. Although, the AIMD control strategy has proved to be efficient, robust and remarkably scalable for the Internet, it is believed that it might be too abrupt for emerging applications such as streaming of audio and video.

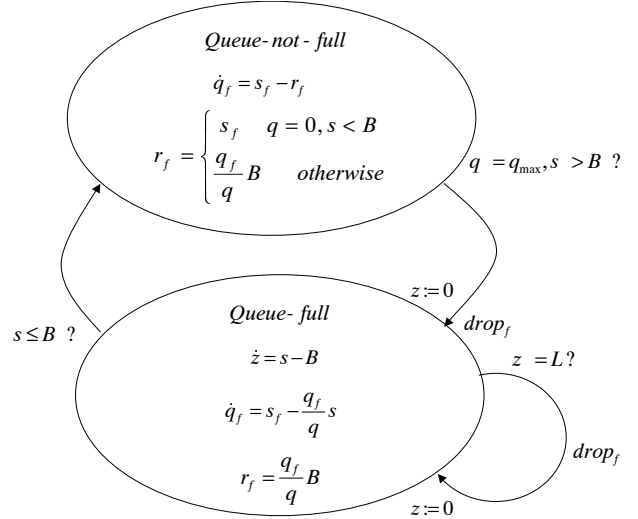


Figure 2: Hybrid queue model.

3 Hybrid Model

In the hybrid model for communication networks proposed by Hespanha et al. [3, 1], the network dynamics and the TCP dynamics is modeled as hybrid systems (e.g., [2, 11]). Note that the hybrid model is based on traffic flows, but is more accurate than a classical flow model since it handles discrete events such as packet drops and window adjusting.

3.1 Network Dynamics

Packet transmission rates are in the hybrid model treated as continuous-time real-valued variables. The received rate of packets at a router is denoted r and the sent rate is denoted s . The number of packets stored in a router queue is denoted q , which is also treated as a continuous variable. The dynamics of the queue is depending on if the queue is full ($q = q_{\max}$) or not ($0 \leq q < q_{\max}$). We thus for each router introduce the hybrid system with two discrete states shown in Figure 2. In this model, subscript f refers to flow f , so that the windows w_f for all flows are updated according to given equations. Moreover, introduce the variables $q = \sum_f q_f$ and $s = \sum_f s_f$, and let the bandwidth of the outgoing link be equal to a constant B . Note that when the queue is full, a drop will be generated as soon as the variable z is equal to the predefined packet size L . Which flow f of the incoming flows that will lose a packet is determined by the distribution of the flows in the queue.

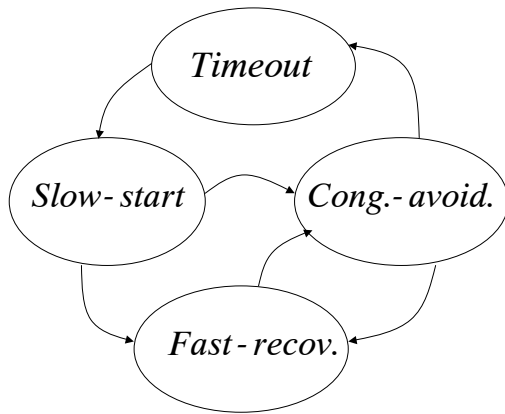


Figure 3: Hybrid TCP model.

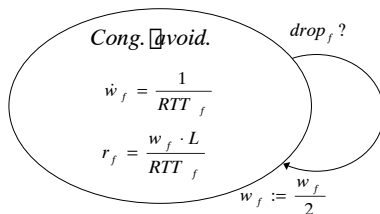


Figure 4: Simplified model of congestion control in TCP.

3.2 TCP Dynamics

The hybrid TCP model consists of four discrete states as shown in Figure 3. We will not detail the continuous dynamics and the transition rules for all states, but rather focus on the most important discrete state, namely, congestion avoidance. It is through the congestion avoidance state the additive increase and multiplicative decrease control strategy described in Section 2 is implemented.

Introduce the round-trip time RTT_f for flow f as the time between sending a packet and receiving the corresponding acknowledgment. It is given by the sum of the physical propagation time and the queueing times. The sender estimates RTT_f and uses it in the congestion control algorithm. Congestion avoidance in TCP can be described by the hybrid system in Figure 4. If RTT_f is approximately constant, we note that the window size w_f grows linearly. The corresponding transmission rate r_f for the sender is proportional to w_f . If a drop occur in flow f , the window size is reduced by a factor two.

Figure 5 shows a simulation of the queue size q (solid) and the window size w (dashed) for a typical session. When the queue becomes full ($q = q_{\max} = 57$), a packet is dropped and w is reduced by a factor two. The window size has a sawtooth shape, which is characteristic for TCP flows. The reason why the growth

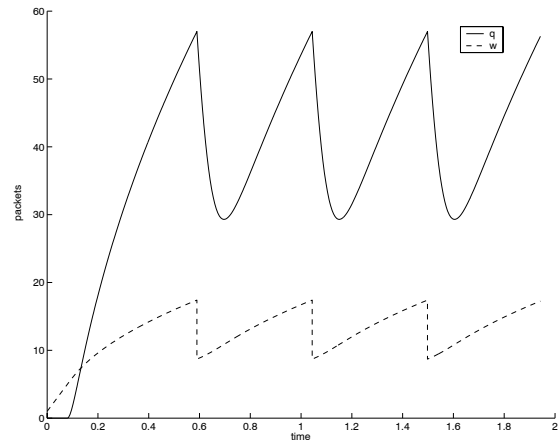


Figure 5: Illustration of congestion avoidance for a network with a single router. When the queue q (solid) exceeds $q_{\max} = 57$, one packets is dropped. TCP reduces accordingly the window w (dashed) by a factor two. Note the characteristic shape of w with intervals of approximately linear growth and periodic jumps.

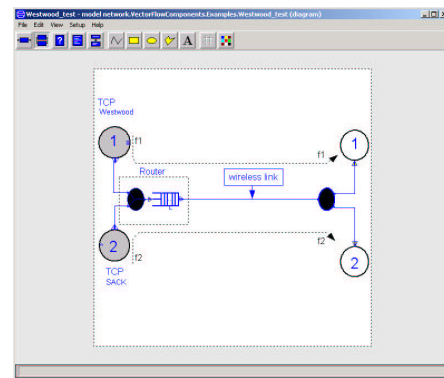


Figure 6: Example of a simple communication network implemented in Modelica.

in w is not linear in the beginning of the session is that RTT varies substantially due to the rapid increase in q : when q is small RTT is approximately equal to the physical propagation time, while if q is large packets spend a relatively large amount of time in the queue.

4 Modelica Implementation

An example of a communication network implemented in Modelica is shown in Figure 6. Two senders and two receivers are connected to the network. Their flows are sharing the same link capacity. If the sum of the flows at some time instance is larger than the bandwidth of the link, packets will be queued. The hybrid model described in previous section consists of continuous equations and discrete events. Modelica was chosen as implementation language

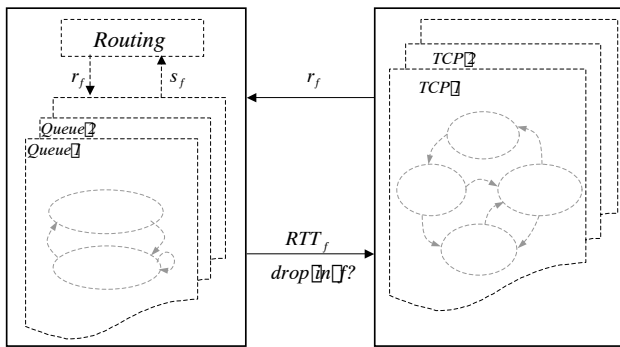


Figure 7: Composition of hybrid communication network model. The network dynamics and the TCP dynamics are separated.

since it supports efficient handling of such models. We have developed a communication network library. The library contains standard building blocks for network simulation, such as TCP senders, routing tables, and queues. Figure 7 shows the schematic layout of our communication network model. Note that the network dynamics and the TCP controllers are separated. The only information shared between the two submodels are the sending rates r_f , the round-trip times RTT_f , and the drop events. The modular structure allows an easy testing of for instance different TCP controllers applied to the same network topology.

Appropriate handling of the switching between discrete states is important for accurate and efficient simulation of the hybrid model. Implementations in Simulink showed some problems in this respect. Our implementation in Modelica and simulation in Dymola works well. Note that the simulation time is not depending on the number of flows, but instead by the number of discrete events generated by packet drops. The simulation time grows considerably when the number of discrete events becomes large, which hence limits the complexity of the model that can be studied. In ns-2, where individual packets are simulated, the simulation time is also depending on the size of the packet flows as well as the number of flows.

5 Example

Let us simulate the simple computer network in Figure 6. Sender 1 sends Flow f_1 using a version of TCP called TCP Westwood (TCPW) [6] and Sender 2 sends Flow f_2 using TCP SACK [3]. TCP SACK corresponds approximately to “standard” TCP used for the Internet today. The flows are sent over the same wireless link. For a wireless link transmission losses are

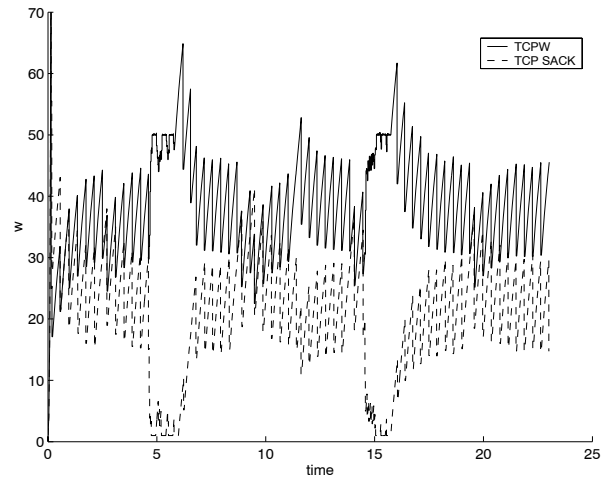


Figure 8: Simulation of the system in Figure 6. The windows size for TCP SACK is considerably reduced when the wireless link is losing a lot of packets (at about $t = 5$ and $t = 15$), while TCPW is able to cope with the losses very well.

more likely than for a wired link. For the network in Figure 6, there can be packet losses both due to that the router queue is full and due to that the wireless transmission lose packets. We model the wireless link as having a good and a bad state. In the good state 0.1% of the transmitted packets are lost, while in the bad state 10% of the packets are lost. The link stays in each state a random amount of time, which is exponentially distributed. TCPW was designed taking wireless links into account, while TCP SACK was designed for wired links. Next we will see that TCPW might be a good option for the emerging wireless Internet.

Figure 8 shows the window sizes for a simulation of the network in Figure 6. The solid line corresponds to TCPW and the dashed line to TCP SACK. Note the time intervals at about $t = 5$ and $t = 15$ when the link is in the bad state. The packet losses due to the bad transmission result in a sudden decrease of the window size for TCP SACK, while TCPW are able to compensate for the packet losses of the wireless link. The window size for TCPW is in general larger than for TCP SACK. This gives a larger throughput for the connection using TCPW, as is shown in Figure 9.

Figure 10 shows the throughput when two TCPW's are sharing the same wireless link (upper plots) and when two TCP SACK's are sharing the same link (lower plots). From the simulations we see that the major advantage of TCPW is when the link is in the bad state. When the link is in the good state, the performance of both TCP implementations are roughly equal.

Since TCP (SACK) was developed for wired net-

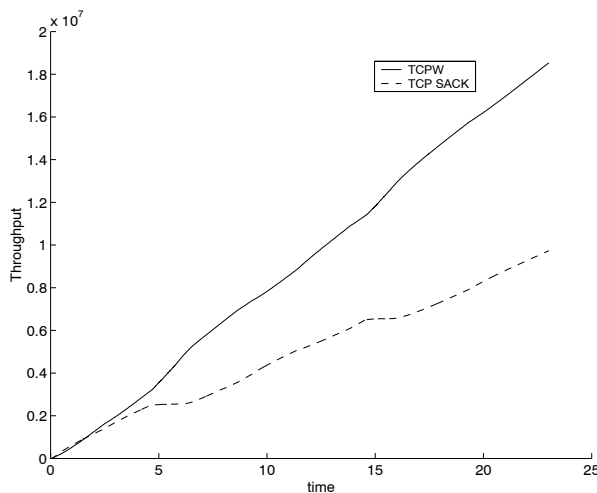


Figure 9: TCPW gives a larger throughput than TCP SACK. The difference is emphasized when there are a large number of packet losses due to the wireless transmission.

works, where packet losses arise only due to buffer overflow, there is a problem using it over wireless links. The reason is that the window size is reduced by a factor two every time a drop occurs, regardless if the drop is due to congestion or to transmission loss. TCPW has another way of updating the window size when a drop occurs. The window size is set to a value based on an estimate of the available bandwidth and the current round-trip time RTT . Since RTT is highly dependent on the queue sizes, so that RTT is small if and only if all corresponding queues are small, a small RTT implies that a detected drop must be due to wireless loss. The aim of TCPW is hence to utilize the available bandwidth more efficiently.

References

- [1] S. Bohacek, J. P. Hespanha, J. Lee, and K. Obraczka. Analysis of a TCP hybrid model. In *Proc. of the 39th Annual Allerton Conference on Communication, Control, and Computing*, 2001.
- [2] R. W. Brockett. Hybrid models for motion control systems. In H. Trentelman and J. Willems, editors, *Essays in Control: Perspectives in the Theory and Its Applications*, pages 29–53. Birkhäuser, Boston, 1993.
- [3] J. P. Hespanha, S. Bohacek, K. Obraczka, and J. Lee. Hybrid modeling of TCP congestion control. In M. Di Benedetto and A. Sangiovanni-Vincentelli, editors, *Hybrid Systems: Computation and Control*, volume 2034 of *Lecture Notes in Computer Science*, pages 291–304. Springer-Verlag, Berlin, Germany, 2001.
- [4] V. Jacobson. Congestion avoidance and control. In *Proc. of SIGCOMM*, volume 18.4, pages 314–329, 1988.
- [5] W. E. Leland, M. S. Taqqu, W. Willinger, and D. V. Wilson. On the self-similar nature of ethernet traffic (extended version). *IEEE/ACM Transactions on Networking*, 2(1):1–15, 1994.
- [6] S. Mascolo, C. Casetti, M. Gerla, M. Y. Sanadidi, and R. Wang. TCP Westwood: bandwidth estimation for enhanced transport over wireless links. In *MobiCom*, Rome, Italy, 2001.
- [7] *The Network Simulator ns-2*. Information Sciences Institute, University of Southern California. <http://www.isi.edu/nsnam/ns>.
- [8] V. Paxson and S. Floyd. Wide-area traffic: the failure of Poisson modeling. *IEEE/ACM Trans. on Networking*, 3(3):226–244, 1995.
- [9] V. Paxson and S. Floyd. Why we don't know how to simulate the internet. In *Proc. of the Winter Simulation Conference*, 1997.
- [10] L. L. Peterson and B. S. Davie. *Computer networks: a systems approach*. Morgan Kaufmann, 2nd edition, 2000.
- [11] A. van der Schaft and J. M. Schumacher. *An Introduction to Hybrid Dynamical Systems*. Number 251 in *Lecture Notes in Control and Information Sciences*. Springer-Verlag, London, 2000.
- [12] J. Walrand and P. Varaiya. *High-performance communication networks*. Morgan Kaufmann, 2nd edition, 2000.

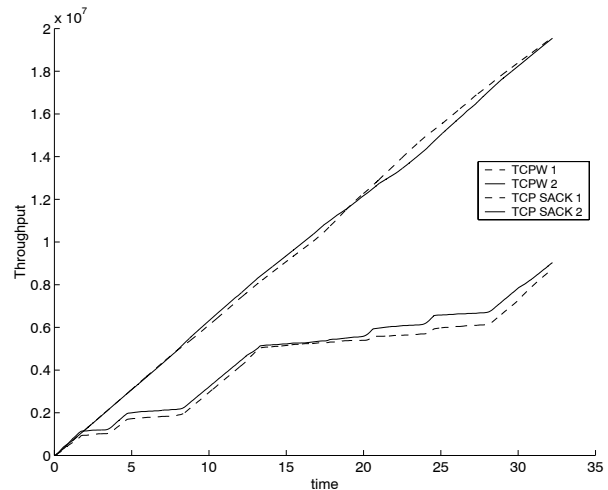


Figure 10: Throughput for two TCP Westwood senders (upper plots) and two TCP SACK senders (lower plots).

Session 8b

Thermodynamic Systems II

TechThermo- A Library for Modelica Applications in Technical Thermodynamics

W.D. Steinmann, S. Zunft

German Aerospace Center (DLR)
Institute of Technical Thermodynamics
Pfaffenwaldring 38-40, 70569 Stuttgart
wolf.steinmann@dlr.de

Abstract

This paper describes the development of the Modelica library TechThermo. This library is intended as a basis for simulation projects in the area of technical thermodynamics. TechThermo enhances the efficiency of simulation activities by providing models describing essential processes which are not restricted to certain applications. The library contains connector-definitions, boundary conditions, basic descriptions of heat and mass transfer, control volumes, general property routines and components like turbines or heat-exchangers. The structure of models can be controlled by parameters thus providing the possibility to choose between different physical descriptions of a process.

Introduction

At DLR's Institute of Technical Thermodynamics various simulation activities cover projects reaching from solarthermal power generation to fuel cell systems. Initial experiences with Modelica show that the efficiency of the simulation activities can be improved significantly by definition of a base library containing models which are not restricted to a special application. By using such a base library developers working on different projects share a common basis and can concentrate on the physical process which are typical for a special application while the common basis undergoes a continuous optimization resulting from the experience in the different projects.

Resulting from the former work with Modelica, some recommendations for improving the acceptance of a library can be made:

- models should be small and easy to understand
- models should be numerically robust
- the dependance on the right choice for the right initial conditions should be as small as possible
- the total number of components in a library should be limited; it's better to build several smaller libraries than building a single one containing everything
- the time period for developing a library should be limited

- an extensive application of object-oriented techniques like multiple inheritance doesn't improve readability

Experienced users should profit from TechThermo primarily by extending the models provided by the library thus minimizing the amount of trivial equations needed for describing a physical process. By standardization of connectors and variable names of input and output variables the reusability and readability of models is enhanced. On the other hand, this library should allow a quick analysis of thermodynamic systems without the necessity of major modifications of the models. This also helps new users to apply Modelica for the simulation of thermodynamic systems.

1 Library Structure

TechThermo is organized in a four-level structure. There are seven main packages, which are stored in separate files:

- **Interface** connectors and general base models
- **Source** boundary conditions
- **Basic** heat/mass transfer, control volumes
- **Medium** thermophysical properties
- **Component** basic components
- **Subsystem** simplified thermodynamic systems
- **Example** examples

Each of these main packages is divided in sub-packages. These sub-packages contain models which can be used without further modifications. If necessary, there are supporting models and data-records on the fourth level:

main package

sub-package

model 1

model 2

model 3

package Support

support-model 1

support-model 2

package Data

data record 1

data record 2

This structure enhances the orientation of the user within the library, since the first two levels contain only packages, while all models which can be used immediately are concentrated within the third level.

2 Control of Model Structure by Parameters

A general-purpose library has to offer a choice between different descriptions for a physical process to adapt the model to a certain simulation task. In Modelica flexibility can be reached by using replaceable models. In TechThermo, this approach is used to exchange models which show significant differences. In other cases, differences in a physical model only affect a single equation. Here, the usage of replaceable models seems not to be effective, since the total number of models will increase significantly. Instead, parameters are used in combination with if-expressions to influence the structure of a model. Two different cases can be distinguished: if there are only two alternatives, parameters of type Boolean are used to select the appropriate physical model. In TechThermo, names of such parameters start with *switch_* to distinguish them from other parameters:

```
model
    parameter Boolean switch_name = true;
    ....
    equation
        if switch_name then
//      this part is only activated, if
//      switch_name== true
            equation 1;
            equation 2;
            equation 3;
            ...
        end if;
```

If there are more than two alternatives parameters of type Integer are applied. The names of these parameters start with *option_*. The alternatives are described in the info-section of a model. Users don't have to get along with names for different replaceable models, the total number of different models is kept small:

```
model
    parameter Integer option_name=1;
    ....
    equation
        if option_name== 1 then
//      this part is only activated, if
//      option_name== 1
            equation 1;
            equation 2;
            equation 3;
            ....
        end if;
        if option_name== 2 then
//      this part is only activated, if
//      option_name== 2
            equation 1;
            equation 2;
            equation 3;
            ....
        end if;
        if option_name== 3 then
//      this part is only activated, if
//      option_name== 3
            equation 1;
            equation 2;
            equation 3;
            ....
        end if;
```

The *switch_* and *option_* parameters are declared with default values describing the standard case, so users can use the models without modifications of these parameters. Dymola translates only the parts which are activated by the parameters, other sections are ignored. As a result, control of model structure by parameters doesn't increase the calculation time. On the other side, modifications of *switch_* and *option_* parameters only become effective after a new compilation.

3 Connectors

The definition of connectors is an essential task during the creation of a base library. There's no unique 'right' definition of connectors. In TechThermo four different kinds of connectors are defined:

mass-flow	\dot{m} [kg/s]	mass-flow rate
	h [J/kg]	spec. enthalpy
	p [Pa]	pressure
	x_i [-]	vector of mass-fractions
heat-flow	\dot{q} [W]	heat-flow rate
	t [°C]	temperature
thermal state	h [J/kg]	spec. enthalpy
	p [Pa]	pressure
	ρ [kg/m ³]	density
	s [J/kg/K]	spec. entropy
	t [°C]	temperature
	u [J/kg]	spec. internal energy
	x [-]	steam quality
	x_i [-]	vector with mass fractions
exergy-flow	\dot{ex} [W]	exergy-flow rate

There are two connectors of each kind with different graphical presentation. The mass-flow connector allows the description of combined heat and mass-transfer. Specific enthalpy and pressure enable a description of the thermal state of the flowing medium. The description of a multi-component flow is possible by the vector x_i which contains the mass-fraction of each component. During the development of TechThermo different alternatives for the mass-flow connector have been discussed: the definition of individual mass-flow connectors for different transported media proves not to be efficient, since the number of connectors became too high. Instead, an universal mass-flow connector was defined. While the choice of mass-flow rate and pressure as connector variables is obvious, alternatives to specific enthalpy have been discussed. One disadvantage of specific enthalpy as a connector variable is that it is neither a 'through' nor an 'across' variable, a connection of more than two elements demands a mixing model. This might be avoided by using enthalpy-flow rate (product of specific enthalpy and mass-flow) as a connector variable. The major drawback of this concept is that information about the energy in the fluid is always related to the mass-flow

rate: if the velocity of the fluid is zero, no information about the specific enthalpy of the fluid is available, if specific enthalpy is calculated from the enthalpy-flow rate division by zero must be avoided. Temperature as connector variable doesn't allow a clear definition of thermal state in the two-phase region, entropy as connector variable is interesting from a theoretical point of view but is probably not accepted by all users.

The heat-flow connectors are used for thermal energy transfer without mass-flow, connector variables are temperature and heat flow rate.

The state connector transports information about the thermal state of a system. A characteristic of thermodynamics is that there are many different options for describing the thermal state. Depending on the application, different sets of variables are preferred. While the steam quality is interesting when simulating a steam generator, it is unnecessary during the analysis of a gas turbine. A general library like TechThermo must offer a variety of state variables at the thermal state connector without urging the user to define all of them to avoid error messages. The state connector contains eight different kinds of state variables: spec. enthalpy h , pressure p , density ρ , spec. entropy s , temperature t , spec. internal energy u , steam quality x and composition x_i . By using the model *NotUsedVariables* variables at the state connector can be equated with default values thus eliminating the necessity to define them explicitly. The choice of variables defined by *NotUsedVariables* depends on parameters of type Boolean.

The set of connectors is completed by the general exergy-connectors which transport exergy-flows like mechanical power or electricity without further specification of the kind of power-flow.

3.1 Adapters to Connectors of Modelica Standard Library

The Connector main package contains a sub-package with models to join models from the Modelica Standard Library to models from TechThermo. These adapters include a connector from each of these two libraries and define the relation between the variables of the two connectors.

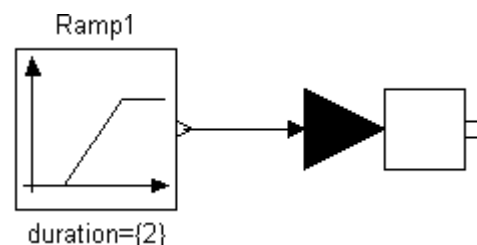


Fig. 1: Adapter-model *OutHeatFlowInSignal* transforming signal to thermodynamic connector variable

Fig. 1 shows the adapter-model *OutHeatFlowInSignal* that connects a heat flow connector to the connector *InPort* from the Modelica Standard Library. The adapter is used to transform a general signal provided by model *Ramp* to a thermodynamic variable. The thermodynamic variable is either temperature or heat-flow rate depending on the value of parameter *option_defsignal* in *OutHeatFlowInSignal*. Similar adapters are implemented for the other connectors in TechThermo.

3.2 TwoPort Models

Many processes in technical thermodynamics can be regarded as systems with an inflow and an outflow. As base classes for models TechThermo provides for all defined connectors models with an inflow and an outflow connector. There are two different kinds of models: while *TwoPortCM* only includes two connectors with different graphical representation, *TwoPort* extends *TwoPortCM* introducing also simple relations for the corresponding variables of the two connectors. These relations can be controlled by parameters. Many processes affect only some of the connector variables while others remain constant. By using *TwoPort* as a base model, the necessity to define explicitly connector variables which remain constant is avoided by setting the corresponding parameter values. For example models describing heat transport can extend the *TwoPort*-model for heat flow:

```
partial model TwoPortCM
  " partial model heat flow element with two
  connectors"
  //-----connectors-----
  In InHeatFlow;
  Out OutHeatFlow;
end TwoPortCM ;
```

```
model TwoPort
  "model heat flow element with two connectors"
  extends Support.TwoPortCM;
  //-----boolean switches for additional
  // equations-----
  parameter Boolean switch_q_dot_const=false
    "if switch_q_dot_const=true then
    q_in_dot+q_out_dot=0";
  parameter Boolean switch_t_const=false
    "if switch_t_const=true then t_in=t_out";

  //Internal variables

  flow SIunits.HeatFlowRate q_in_dot;
  SIunits.CelsiusTemperature t_in;
  flow SIunits.HeatFlowRate q_out_dot;
  SIunits.CelsiusTemperature t_out;

equation

  q_in_dot = InHeatFlow.q_dot;
  t_in = InHeatFlow.t;
```

```
q_out_dot = OutHeatFlow.q_dot;
t_out = OutHeatFlow.t;

//relations between connector variables at
//heat_cut1 and heat_cut2 dependant on boolean
//parameters:

if switch_q_dot_const then
  0.0 = q_in_dot + q_out_dot;
  //heat flow rate remains constant
end if;

if switch_t_const then
  t_in = t_out;
  //temperature remains constant
end if;

end TwoPort;
```

The introduction of variables like *t_in*, *q_in_dot*, *t_out*, *q_out_dot* shorts the names of connector-variables.

There are also *TwoPort* mass-flow elements with additional connectors for heat flow or exergy flow. These models offer the option to activate a stationary energy balance.

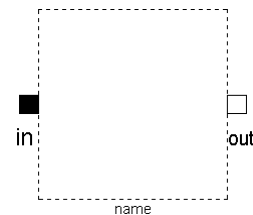


Fig. 2: Icon for *TwoPort*-model for heat-flow

4. Source Models

The Source package contains models representing boundary conditions in thermodynamic systems. Boundary conditions can be introduced for all kind of connectors. The connector variables used as boundary conditions are selected by parameters thus avoiding the necessity to define separate models for various kinds of boundary conditions.

The following Modelica-Code shows the definition of a source for heat flow:

```

model ParameterDefined
  heat-flow source with optional definition of
  heat-flow variables by parameters"

  TTInterface.HeatFlow.Out OutHeatFlow;

  //-----parameters-----
  //switches for variables at connectors:
  parameter Boolean switch_q_dot_def=false
    "if switch_q_dot_def=true, OutHeatFlow.q_dot is
    determined by parameter q_dot_para";
  parameter Boolean switch_t_def=false
    "if switch_t_def=true, OutHeatFlow.t is
    determined by parameter t_para";

  //values for variables at outlet if
  //corresponding switch-parameter=true
  parameter SIunits.HeatFlowRate q_dot_para=1.0
    "value for heat-flow rate HeatFlowOut.q_dot at
    outlet if switch_q_dot_def=true";
  parameter SIunits.CelsiusTemperature
    t_para=25.0
    "value for temperature at HeatFlowOut.t at
    outlet if switch_t_def=true";
  equation
    if switch_q_dot_def then
      OutHeatFlow.q_dot = q_dot_para;
    end if;

    if switch_t_def then
      OutHeatFlow.t = t_para;
    end if;

end ParameterDefined;

```

This model allows to select a constant temperature or a constant heat flow (or both) as boundary conditions. In addition to models providing constant boundary conditions, there are also models with an signal input offering the possibility to control one of the boundary variables by an external signal source.

5. Models for Heat and Mass Transfer

Package Basic offers basic models for describing heat and mass transfer processes. There are models describing thermal conduction, convective heat transfer and radiation heat transfer. Depending on switch-parameters values for heat conductivity, heat transfer coefficient or emissivity can be assumed as being constant or not. The model *HeatTransfer* extends the *TwoPort*-model for heat flow and calculates convective heat-transfer assuming a constant value for the heat transfer coefficient. There's no storage of energy in the element, so parameter *switch_q_dot_const* = *true*, i.e. $q_{in_dot} = -q_{out_dot}$. The temperature difference between the inlet and the outlet is calculated from the heat-flow rate, the area and the heat transfer coefficient:

```

model HeatTransfer
  "basic model describing heat transfer "
  extends TTInterface.HeatFlow.TwoPort
    (switch_q_dot_const=true);

  SIunits.CoefficientOfHeatTransfer alpha_trans;
  parameter SIunits.CoefficientOfHeatTransfer
    alpha_trans_const=1000
    "if switch_alpha_const==true then
    alpha_trans=alpha_trans_const";
  parameter SIunits.Area surface_area=1.0
    "area normal to direction of heat transfer";

  //-----switch-parameter-----
  parameter Boolean switch_alpha_const=true
    "if switch_alpha_const==true then
    alpha_trans=alpha_trans_const";

  equation

    if switch_alpha_const then
      alpha_trans = alpha_trans_const;
    end if;

    q_in_dot = surface_area*(t_in -
    t_out)*alpha_trans;

end HeatTransfer;

```

The model *HeatTransferVariable* extends the more basic *HeatTransfer*-model, parameter *switch_alpha_const* being set to *false* and the heat transfer coefficient *alpha_trans* is now calculated from the properties of the fluid and the velocity. There are two connectors for the mass-flow determining the heat transfer coefficient. An additional state connector is needed to introduce the density of the fluid for calculating the velocity from mass-flow rate. *HeatTransferVariable* offers various possibilities for calculating the heat transfer coefficient, the user can choose the appropriate physical model by setting an option_parameter.

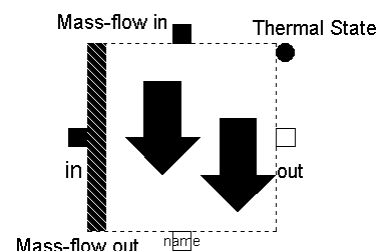


Fig. 3 model *HeatTransferVariable* with connectors for mass-flow and thermal state to calculate heat transfer coefficient.

Similar to the models describing heat transport, the models included in TechThermo for calculating pressure loss due to mass flow offer the possibility to choose between different physical model depending on flow geometry, accuracy and velocity range.

6 Control Volumes

A proper description of control volumes is essential for modelling the dynamic behaviour of a thermodynamic system. In TechThermo, the dynamics of a finite mass with a single connector is described by the model *ThermalCapacity*. The temperature of the system is calculated from the heat flowing into or out of the system. Heat capacity and mass of the system can be determined by parameters c_{heat} , m_{const} , but there's also the possibility to extend the model and introduce other definitions for mass or heat-capacity after a modification of the corresponding switch_parameters. This might be useful if other parameters are preferred (example: pipe with mass calculated from diameter and density).

```

model ThermalCapacity
  "control volume finite thermal capacity"

  TTInterface.HeatFlow.In HeatCut
  SIunits.Mass m;
  // mass of thermal capacity
  SIunits.SpecificHeatCapacity c_heat;
  // specific heat capacity

  parameter SIunits.Mass m_const=1
    "const. value mass if
    switch_m_const==true";
  parameter SIunits.SpecificHeatCapacity
    c_heat_const=500
    "const. value heat-capacity if
    switch_c_heat_const==true";

  //      Boolean switches
  parameter Boolean switch_m_const=true
    "if switch_m_const==true then
    m=m_const";

  parameter Boolean
    switch_c_heat_const=true
    "if switch_c_heat_const==true then
    c_heat=c_heat_const";

equation

  if switch_c_heat_const then
    //      specific heat capacity is
    //      defined by parameter
    c_heat = c_heat_const;
  end if;

  if switch_m_const then
    //      mass is defined by parameter
    m = m_const;
  end if;

  //      transient energy conservation
  HeatCut.q_dot = m*c_heat*der(HeatCut.t);

end ThermalCapacity;

```

A general description for a finite volume with a single mass-flow connector and a heat-flow input is provided by the model *ControlVolume* including energy and mass conservation:

```

model ControlVolume
  parameter Integer n_comp=1;
  parameter SIunits.Volume v_control=1;

  //--connector for thermal state---
  TTInterface.ThermalState.In
  StateCut(n_comp=n_comp);

  //connector for inflow or outflow of mass
  TTInterface.MassFlow.In
  InMassFlow(n_comp=n_comp);

  //----- connector for heat
  //transferred to ControlVolume----
  TTInterface.HeatFlow.In InHeatFlow;

protected
  SIunits.InternalEnergy energy;
  SIunits.MassFlowRate m_inflow_dot;
  SIunits.MassFlowRate m_outflow_dot;

equation

  InMassFlow.p = StateCut.p;
  InMassFlow.x_i = StateCut.x_i;
  InHeatFlow.t = StateCut.t;

  energy =
    v_control*StateCut.u*StateCut.rho;

  der(energy) = m_inflow_dot*InMassFlow.h
    + InHeatFlow.q_dot +
    m_outflow_dot*StateCut.h;

  v_control*der(StateCut.rho) =
    InMassFlow.m_dot;

  m_inflow_dot = if InMassFlow.m_dot <= 0
  then 0.0 else InMassFlow.m_dot;
  m_outflow_dot = if InMassFlow.m_dot > 0
  then 0.0 else InMassFlow.m_dot;

end ControlVolume;

```

Model *ControlVolume* gets information about the density of the fluid by the state connector *StateCut*, usually by an external property routine providing a relation between internal energy u , density ρ and pressure p , so the model is not restricted to a certain kind of fluid. The mass-flow can either be positive or negative, the direction also influences the specific enthalpy at the mass-flow connector.

7 Thermophysical Properties

In technical thermodynamics the calculation of physical properties plays an important role. Since TechThermo should be a general purpose library, the routines included for property calculation are mainly based on universal physical models, descriptions optimized for a certain medium are not applied. For most models, the characterization of a substance by molar mass and critical point properties is sufficient. With respect to calculation time and errors usually introduced by simplified physical models in other parts, the reduced

accuracy of general property models seems to be acceptable.

In TechThermo the property models are separated from other model parts, informations are exchanged by a state connector. Connecting a property model usually presents the final step in creating a model. The general models for property calculation are organized in four sub-packages according to the physical condition:

- Solid
- Liquid
- Gas
- MultiPhase

Descriptions for the solid and liquid state offer the possibility to take variations of density into account or not. Depending on the demanded accuracy, one of the following models describing the gas state can be selected:

- *PerfectGas*:
 $p v = R T$, specific heat capacity is constant
 low pressure, small temperature variations, far from saturation temperature
 example: dry air at ambient conditions
- *IdealGas*:
 $p v = R T$, specific heat capacity is temperature dependant
 medium pressure, significant temperature variation, far from saturation temperature
 example: air in gas turbine
- *RealGas*:
 Redlich Kwong equation:

$$p = \frac{RT}{v - b} - \frac{a}{T^{0.5} v(v + b)}$$
 coefficients a and b are calculated from critical values,
 heat capacity is temperature dependant
 example: superheated water-steam

The Redlich-Kwong equation is a cubic equation of state. In order to determine the correct solution, a cubic equation solver based on the method of Cardano is applied. The calculation of caloric values like spec. enthalpy or entropy demands an equation for the specific heat capacity dependant on temperature. Since no general models are available, polynomes (usually second degree) must be provided for calculating the specific heat capacity.

The model *SaturationTemperature* calculates the saturation temperature from the pressure according to Antoine, the model *EvaporationPressure* calculates the evaporation enthalpy. Together with the models describing liquid and gaseous state the calculation of thermophysical properties with a minimum of medium specific data is possible.

8 Components

Main package *Component* contains models for the fundamental units of a system in technical thermodynamics like compressors, turbines, heat-exchanger, pipes, valves, tanks or burners. These models are composed of general basic models and a specific property routine. For example the general model for a turbine is model *Basic* (located in package *Turbine.Support*):

```

model Basic
  "turbine without specification of working fluid"

  extends TTInterface.MassFlow.TwoPort (
    final switch_m_dot_const=true,
    final switch_h_const=false,
    final switch_p_const=false,
    final switch_x_i_const=true);

  //-----exergy-connector mechanical power
  //provided by expansion
  TTInterface.ExergyFlow.Out PmechCut;

  //-----connector for
  //thermodynamic properties at inlet
  TTInterface.ThermalState.In InletState;

  //-----connector for thermodynamic
  //properties after ideal expansion-----
  TTInterface.ThermalState.Out
  IdealExpansionState;

  //-----parameters
  parameter SIunits.Efficiency
  eta_expansion_const=0.8
  "const. efficiency of turbine if
  switch_eta_expansion_const==true";

  //-----switch-parameters-----
  parameter Boolean
  switch_eta_expansion_const=true
  "if switch_eta_expansion_const==true
  then eta_expansion=eta_expansion_const";

protected
  SIunits.SpecificEnthalpy dh_ideal;
  // spec. enthalpy difference isentropic
  //expansion
  SIunits.SpecificEnthalpy h_out_ideal;
  // spec. enthalpy after isentropic
  //expansion
  SIunits.Efficiency eta_expansion;

```

```

equation

  if switch_eta_expansion_const then
    eta_expansion = eta_expansion_const;
  end if;

  InletState.h = h_in;
  InletState.p = p_in;
  InletState.x_i = x_in_i;

  //      thermal state after isentropic
  //expansion from p_in to p_out
  //      is defined by p2 and entropy
  //entropy_inlet_cut.s
  IdealExpansionState.p = p_out;
  IdealExpansionState.s = InletState.s;
  IdealExpansionState.x_i = x_in_i;
  h_out_ideal = IdealExpansionState.h;

  //      decrease specific enthalpy isentropic
  //expansion
  dh_ideal = h_out_ideal - h_in;
  //      specific enthalpy after real expansion
  h_out = h_in + dh_ideal*eta_expansion;
  //      mechanical power provided by expansion
  P_mechCut.exergy_dot = m_in_dot*(h_out -
    h_in);

end Basic;

```

In this basic version of turbine calculation, the outlet condition is calculated from the inlet condition by assuming an isentropic expansion and multiplying the difference in spec. enthalpy by the efficiency of the turbine. The calculation demands the knowledge of the entropy at the inlet, which is provided by state connector *InletState* and the thermodynamic state after the ideal expansion which is provided by connector *IdealExpansionState*. In this basic version, the efficiency of the turbine is determined by the parameter *eta_expansion_const*; a more elaborate turbine model might be implemented by extending *Basic*, setting *switch_eta_const* = false and introducing a calculation procedure for the efficiency *eta_expansion*.

The model should now be used for expanding air, the model *AirTurbine* extends first the model *TurbineMC*, which contains two mass-flow connectors, an exergy-flow connector for the mechanical work delivered by the turbine and the icon presentation of the turbine. The general turbine model *Basic* is connected to the outer connectors, finally the model is completed by joining two property routines for air to the thermal state connectors *IdealExpansionState* and *InletState* of *Basic*.

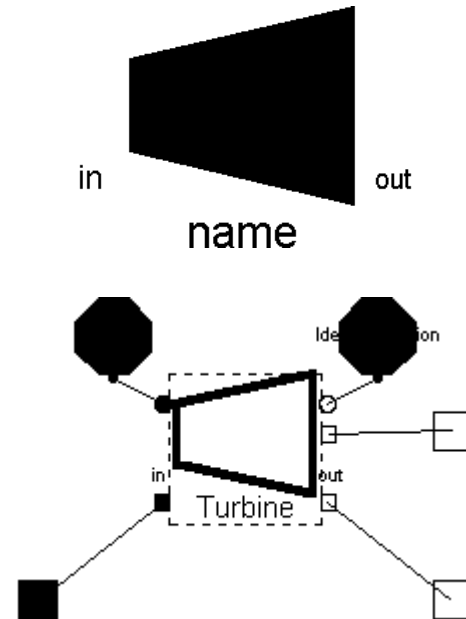


Fig.4 : Icon for model *AirTurbine* and internal view with basic turbine model and two property models.

9 Subsystem

The package subsystem contains simplified models for thermodynamic systems like solar collectors or cyclic processes. These models should be used to complete systems when the main interest of the simulation doesn't focus on the part modelled by the subsystem-component.

10 Current Status and Conclusion

The development of TechThermo started during a project dealing with fuel-cell systems and has taken advantages of the experiences gained in this area. After an initial period of theoretical considerations almost all components identified as essential are implemented. TechThermo is now applied in two different projects, one dealing with high temperature thermal storage, the other dealing with the dynamics of solarthermal steam generation. Compared to first experiences with Modelica for simulating thermodynamic systems, the efficiency of the modelling activities could be improved significantly thus showing the importance of a common base library. At the moment TechThermo undergoes a continuous improvement due to the practical experiences in the projects. While the total number of models in TechThermo should remain more or less constant, future activities will concentrate on the improvement of the numerical stability of the components.

Modelica Library for Hybrid Simulation of Mass Flow in Process Plants

S.M.O. Fabricius and E. Badreddin

Swiss Federal Institute of Technology Zurich (ETH), Switzerland

Laboratory for Safety Analysis

fabricius@lsa.iet.mavt.ethz.ch

Abstract

Operation, control and maintenance of large process plants can be very energy and cost intensive. Optimization of the involved technical, organizational and dependent economic aspects is a non-trivial, multi-criteria problem; solving it can be supported by dynamic plant modeling.

First principles, constitutive and empirical relations are used to derive quasi-steady-state mathematical models of fluid storage and flow for a selection of physical components as are typically installed in process plants. Control logic, both on component and plant-level, is integrated using high-level hybrid language constructs of Modelica and in particular, extended Petri net formalism. The resulting Modelica library facilitates efficient composition of mass flow models of potentially large and complex plants and allows for simulative investigation of plant dynamics.

1 Introduction

Availability of process plants depends mainly on a combination of the reliability of individual installed components, the plant topology, the control system¹ and on the plant maintenance strategy and procedures. Traditional system analysis methods, as known from reliability engineering, e.g., fault tree analysis (FTA), event tree analysis (ETA), hazard and operability study (HAZOP) and failure mode and event analysis (FMEA) can lead to significant insight into the weak points of a plant with respect to performance measures as reliability, availability, safety or profitability. However, these methods lack force of expression when trying to deal with highly *dynamic systems* or systems with extensive *internal feedback loops*. Collaboration with a partner from industry—on topics of maintenance, fault detection and monitoring of process plants—has led to the insight that performance analysis of complex process plants needs to consider not only static aspects, but also dynamic ones, not only technical, but also operational, organizational and economic factors as well.

¹Both automatic control and human operator interaction.

Petri nets are well suited for modeling of discrete-event phenomena, and can be used beneficially to investigate plant availability (as demonstrated e.g., in [Fa01]), but are inherently far less convenient as a modeling formalism when confronted with physical processes exhibiting continuous-time behavior. In order to address all above mentioned aspects of process plant performance, integrated hybrid² dynamic modeling—unifying different formalisms—is in need.

*Modelica*³, still rather new, with its multi-domain and multi-formalism modeling capabilities, seems promising in this respect. It defines a physical object-oriented modeling paradigm suitable for expressing hybrid behavior and offers respective high-level language syntax and semantic. It supports hierarchy, reuse of modeling knowledge and provides an open standard, based upon which, a couple of computer-based tools have already been created or have migrated to⁴. Several open, standard Modelica libraries exist e.g., for electric, mechanic and hydraulic systems. Among other, currently, a thermodynamics library called “Thermofluid” is being developed, see [Tu98], which could be very useful for modeling of process plants in the future as well.

In this text, primarily, mass-flow dynamics in process plants are to be investigated, and for this purpose, a new, customized library is presented. It contains models of basic equipment needed for fluid—in particular liquid—transport and storage as well as logic to control the flow of material. The library considers the flow as *quasi-steady-state*, i.e., no momentum balance is formulated; the flow is assumed to reach steady flow conditions instantly based on the current outer pressure situation and internal pressure drop characteristic.

System-level dynamics are of main interest here, not the very details of component behavior. This distinguishes it also from the available—partly commercial—hydraulics library in Dymola, which focuses on hydraulics as used in control systems. Emphasis is on

²The term “hybrid” is used here to denote combined discrete-event and continuous-time behavior.

³Modelica is a trademark of the Modelica Association (<http://www.modelica.org>)

⁴In this text, the tool “Dymola” is used; Dymola is a registered trademark of Dynasim AB, Sweden, <http://www.dynasim.se>

ease-of-use, scalability and extendibility. Models generated with the library can be used for experimental design studies, control scheme optimization, de-bottle-necking, re-engineering of already commissioned plants. The paper first discusses basic physics and mathematics of modeling fluid flow components before addressing implementation in Modelica and giving several modeling and simulation examples.

2 Basic Physics and Mathematics of Quasi-Steady-State Fluid Flow

This section briefly describes conservation laws, constitutive relations and phenomenological empirical relations as they apply to *liquid* storage and flow in equipment as typically installed in process plants. Note, liquid density ρ is considered constant throughout the text.

2.1 Fundamental Relations for Liquid Storage and Flow

The *mass balance* for liquid storage is

$$\frac{dM}{dt} = \sum_i \dot{m}_i \quad (1)$$

(with total mass M contained within the reservoir and in- and outflow rates \dot{m}) and the relation describing *pressure drop* as a function of volumetric flow rate q is

$$\Delta p = f(q). \quad (2)$$

The well know Bernoulli equation in its original form relates *mechanical energies* of a fluid neglecting friction and considering the fluid as incompressible. It can be extended to include friction (*irreversible* loss of pressure head as energy dissipation) as well as addition of flow energy (e.g., by means of pumps or compressors). The sum of pressure-, velocity-, elevation-, friction- and pump-head remains constant along a flow trajectory (eq. 3, subscripts *fr* and *pu* for “friction” and “pump” respectively). According to the second law of thermodynamics, there is a restriction on the conversion direction of friction head, it cannot be converted back into other heads without interaction with the environment.

$$\frac{p}{\rho} + \frac{v^2}{2} + gz + \sum \Delta p_{fr} + \sum \Delta p_{pu} = \text{const.} \quad (3)$$

2.2 Fluid Reservoirs

For a container with constant cross section A , the *liquid level* obeys

$$\frac{dh}{dt} = \frac{1}{A} \sum_i q_i. \quad (4)$$

If liquid flows via a pipe and flange (velocity v_{pipe}) into a large⁵ vessel (see fig. 1), the velocity head of the inflowing fluid can be considered irreversibly lost due to turbulent friction (see e.g., [Th99]), i.e., no pressure is recovered by slowing (process inside the vessel) the fluid to assumed flow velocity zero⁶ in the vessel itself⁷.



Figure 1: Inflow to a vessel

For *inflow* to a tank, extended Bernoulli can be formulated as

$$p_{vessel} = p_{pipe} + \frac{\rho}{2} v_{pipe}^2 - \Delta p_{fr,inlet}. \quad (5)$$

As indicated, the dynamic pressure is irreversibly lost with inflow, therefore, eq. 6 and 7 are valid.

$$\Delta p_{fr,inlet} = \frac{\rho}{2} v_{pipe}^2 \quad (6)$$

$$p_{vessel} = p_{pipe}. \quad (7)$$

Irreversible pressure loss involved with *outflow* of liquid at a flange of a vessel can be formulated as

$$\Delta p_{fr,outlet} = K_{con} \frac{\rho}{2} v_{pipe}^2, \quad (8)$$

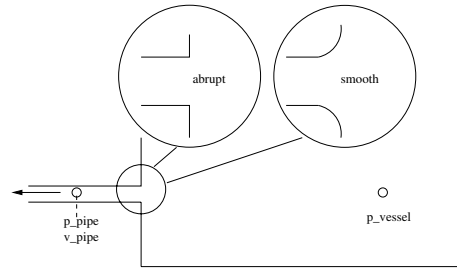


Figure 2: Outflow from a vessel

and is strongly dependent on the geometric form of the flange (see fig. 2). According to [Th99], the contraction coefficient K_{con} of an abrupt exit from the vessel to the outflow pipe is $K_{con,abrupt} \approx 0.5$ whereas for smooth, tapered outlets it is rather small $K_{con,smooth} \approx 0.05$.

⁵“Large” in the sense of the tank diameter being much larger than the flange diameter.

⁶Velocity “far” away from the inlet to the tank where the fluid is essentially not moving and undisturbed by turbulence from the inflow process.

⁷This is assumed true for both fluid inlets below and above the surface of the liquid contained in the vessel.

Pressure in the pipe and vessel relate for outflow as follows

$$p_{vessel} = p_{pipe} + \frac{\rho}{2} v_{pipe}^2 + \Delta p_{fr,outlet}, \quad (9)$$

which can alternatively be formulated with volumetric flow rate q to

$$p_{pipe} = p_{vessel} - \frac{\rho}{2A_{pipe}} (1 + K_{con}) q_{pipe}^2. \quad (10)$$

Note, the relations for pressures and flow velocities are different for the cases of in- and outflow (eq. 7 and 10). In reality, a physical flange at a vessel can normally carry both in- and outflows depending on the surrounding pressure situation. Therefore, in a tank model, a case distinction must be made in order to determine the pressure drop relation valid for the respective current flow situation. In this text, flow *into* a component is always taken positive ($q > 0.0[m^3/s]$), in accordance with Modelica conventions.

2.3 Flow Resistors

When a fluid flows through a flow conduit, frictional effects lead to pressure drops along the flow trajectory. In many flow armatures, e.g., pipe elbows and bends, orifices, diffusers and nozzles, the relation between pressure drop and flow velocity can be described by eq. 11 with pressure drop coefficient ζ as a function of Reynolds number Re and the geometry of the flow channel (eq. 12). In technical applications, ζ is often assumed constant for varying flow rate for many flow armatures.

$$\Delta p = \zeta \cdot \frac{\rho}{2} \cdot v^2 \quad (11)$$

$$\zeta = \zeta(Re, geometry) \quad (12)$$

2.3.1 Pipes

The pressure drop in a pipe is customarily described by

$$\Delta p = \lambda \cdot \frac{L}{D} \cdot \frac{\rho}{2} v^2 \quad (13)$$

with pipe length L , pipe diameter D , flow velocity v and pipe friction factor λ . Depending on the magnitude of Re in the pipe

$$Re = \frac{|v|D}{\nu}, \quad (14)$$

different relations are to be used for λ , see table 1. The linear relation for laminar flow regime is

$$\lambda_{lam} = \frac{64}{Re}. \quad (15)$$

Re	flow regime; relation
$Re < 2300$	laminar; linear
$2000 < Re < 4000$	transient
$Re > 4000$	turbulent; e.g., Blasius, Colebrook Prandtl/Karman/Nikuradse

Table 1: Flow regimes

In the turbulent region and for smooth pipes⁸, the Blasius relation can be used (eq. 16). Colebrook, Karman/Nikuradse and Prandtl/Nikuradse are all implicit relations, less handy to employ and are therefore not further discussed here.

$$\lambda_{turb} = 0.364 \cdot Re^{-\frac{1}{4}}. \quad (16)$$

2.3.2 Valves

A valve shall allow to control fluid flow by changing the valve opening and as a consequence the flow resistance across the valve. Usually the coefficients k_v and k_{vs} are used with valves; k_v indicates how much fluid—usually in $[m^3/h]$ —passes the valve at an outer pressure gradient of $\Delta p = 1$ [bar] at a certain opening position of the valve and k_{vs} indicates the flow when the valve is fully open. The relation for the fully open valve is given in eq. 17.

$$q = k_{vs} \cdot \sqrt{|\Delta p|} \cdot \text{sign}(\Delta p) \quad (17)$$

A continuous control valve shall enable to control fluid flow over the whole continuous range of opening positions possible in the valve. k_v in this case is a function (eq. 18) of the valve opening position x (normalized to $[0..1]$) and k_{vs} and has a characteristic depending on the type and construction of the control valve (e.g., linear, equal-percentage).

$$k_v = k_v(k_{vs}, x) \quad (18)$$

Power control valves are driven by valve positioners (servo), which can be described by a first order exponential lag with time delay τ , relating actual valve travel x to demanded valve travel x_d by

$$\tau \frac{dx}{dt} = -x + x_d. \quad (19)$$

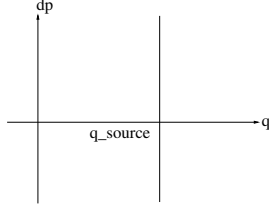
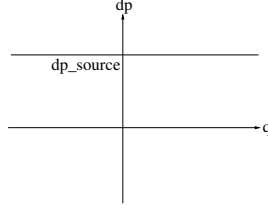
2.4 Pumps

Pumps can be seen as compensators for pressure drops to drive a fluid through its flow channel. Two idealized pump characteristics are shown in fig. 3 and 4. An *ideal flow source* is an abstraction of the behavior of volumetric pumps (piston, displacement pump), which have a very steep relation between pressure increase and flow across the pump. The idealized characteristic

⁸Most commercial pipes for process industry applications can normally be considered “smooth”, if not, the Colebrook relation must be used which accounts for relative roughness ϵ/D .

is simply $q_{pump} = q_{source}$, independent of the pressure gradient. The other idealized representation is that of an *ideal pressure source* (centrifugal pump type). Its parameter can be set as height Δh_{source} (head rise), which determines the pressure rise in the pump (independent of the flow rate).

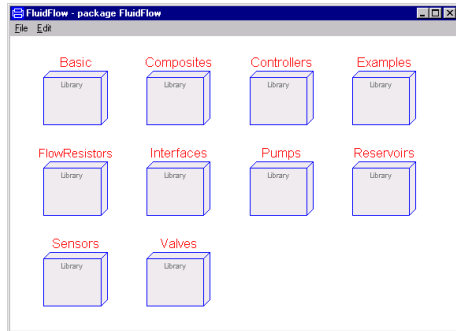
$$\Delta p_{pump} = \Delta h_{source} \cdot \rho \cdot g \quad (20)$$

Figure 3: *Flow source*Figure 4: *Pressure source*

More detailed and realistic pump models are described in the literature and not explicitly discussed here due to lack of space. A model for a centrifugal pump is e.g., presented in [Ge85] as a first order model with nonlinear algebraic constraint equations.

3 Modelica Library Implementation

Having described the basic physics, this section is concerned with fluid flow component modeling to create a Modelica library. Its structure is organized in compliance with Modelica rules, grouping interfaces, various component types and example models in different packages, see fig. 5.

Figure 5: *Library structure*

3.1 Connector, Boundary Conditions and Fluid Properties

A connector called “Flange” (fig. 6) defines the potential variable p for pressure and the flow variable q for volumetric flow rate:

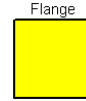
```
connector Flange
  SI.Pressure p;
```

```
  flow SI.VolumeFlowRate q;
end Flange
```

Currently, fluid properties are considered constant for all library components. The respective parameters are defined in an abstract class and given here, for the case of water:

```
partial model Fluid
  constant SI.Density rho=1000.0;
  constant SI.KinematicViscosity nu=1.0e-6;
end Fluid
```

Boundary conditions can either be set for pressure or for flow, the class icons are shown in fig. 7.

Figure 6: *Connector*Figure 7: *Boundaries*

3.2 Flow Resistors

The components presented in this subsection are all modeled as purely resistive, i.e., no mass capacitance and—as indicated earlier—no momentum balance is formulated.

3.2.1 Pipes

With eq. 13 for pressure drop in a pipe and with eq. 15 for friction factor λ_{lam} , a linear relation for pressure drop Δp_{lam} in laminar pipe flow results as

$$\Delta p_{lam} = k_{lam} \cdot v \quad (21)$$

with coefficient

$$k_{lam} = 0.32 \cdot D^{-2} \cdot \nu L \rho. \quad (22)$$

For turbulent flow, the respective non-linear relation is

$$\Delta p_{turb} = k_{turb} \cdot \text{sign}(v) \cdot |v|^{7/4} \quad (23)$$

with

$$k_{turb} = 0.182 \cdot D^{-5/4} \cdot \nu^{1/4} L \rho. \quad (24)$$

The question of how to model the transient region ($Re = [2300..4000]$) arises. In reality, the flow will change from laminar to turbulent depending on the particular pipe geometry, flow disturbances, surface particularities in a somewhat random and discontinuous fashion. Here, a linear model is used to connect laminar to turbulent flow behavior, according to eq. 25 and 26 for positive and negative flow velocity respectively⁹.

⁹Of course, other approaches are possible as well, in particular some with “smooth” transfers, but this linear model is feasible for the investigation purposes in mind.

$$\Delta p_{trans} = -k_{1trans} + k_{2trans} \cdot v \quad (25)$$

$$\Delta p_{trans} = k_{1trans} + k_{2trans} \cdot v \quad (26)$$

The two coefficients k_{1trans} and k_{2trans} can be calculated satisfying the conditions (eq. 27, 28) at the flow regime transition points. Depending on the current flow velocity, the flow regime is determined in the pipe model using the “if-statement” of Modelica for automatic detection of transition points (state events).

$$\Delta p_{lam}|_{Re=2300} = \Delta p_{trans}|_{Re=2300} \quad (27)$$

$$\Delta p_{trans}|_{Re=4000} = \Delta p_{turb}|_{Re=4000} \quad (28)$$

If the pipe connects flanges of different geographical elevations, an additional term Δp_g is included in the pipe model to account for gravitational pressure difference

$$\Delta p_g = \rho g \Delta z. \quad (29)$$

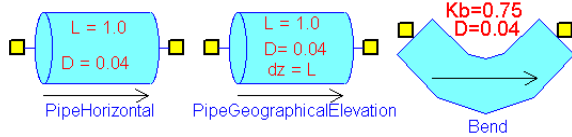


Figure 8: Horizontal and elevated pipe; bend

Fig. 8 displays the pipe component icons. The arrow is intended to indicate the default positive flow direction of the fluid in the pipe. This, in order to facilitate setting reasonable starting values for flow velocity to help the execution algorithms find consistent initial values at the beginning of a simulation.

3.2.2 Bends

The pressure drop across pipe bends (fig. 8, on the right) can be formulated as multiples of velocity heads (subscript “b” for bend)

$$\Delta p_{fr,b} = k_{bf} \frac{\rho}{2} v_b^2. \quad (30)$$

[Th99] lists some numerical values for k_b , see table 2.

type of bend	frictional loss k_b
Standard 45° bend	0.35
Standard 90° bend	0.75
180° bend	2.2

Table 2: Examples of friction factors for pipe bends

3.2.3 Valves

Valves for different valve characteristics are modeled. The one depicted in fig. 10 has a linear characteristic $k_v = k_{vs} \cdot x$ (fig. 9) with a first order servo for the valve positioner. Note, to avoid numerical problems, the valve is modeled to always have a small leakage¹⁰, even for an input signal demanding for complete valve closure. The respective minimum opening valve travel x_{min} can be set to zero if desired and if feasible with the system model topology.

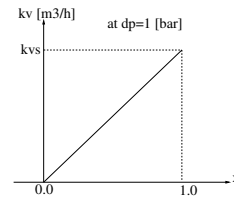


Figure 9: Linear char.

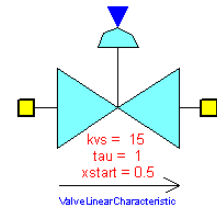


Figure 10: Valve

3.2.4 Other Flow Resistors

An idealized linear (eq. 31) and a general quadratic resistor (eq. 32) are implemented as well.

$$\Delta p_{lin} = k \cdot v \quad (31)$$

$$\Delta p_{gen} = \zeta \frac{\rho}{2} v_{bf}^2. \quad (32)$$

The linear resistor can be useful for simulation test purposes, e.g., when nonlinearities in a model cause problems calculating consistent initial conditions. The pressure loss coefficient ζ can be chosen in order to model orifices, venturi pipes or any other armature for which the pressure loss coefficient is known. Standard texts on fluid dynamics, e.g., [Yo01] or publications from flow armature manufacturers list loss factors for different forms of armatures.

3.3 Pumps

Fig. 11 shows icons of pumps modeled as ideal flow and pressure source as well as a more detailed centrifugal pump model.

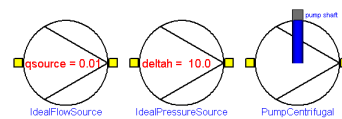


Figure 11: Pump models

¹⁰Possibility of small leakage is also implemented in other armatures allowing otherwise for complete flow interruption.

3.4 Tanks

A variety of tanks have been modeled. Here, a tank with constant cross section, open to atmosphere with a flange at the top and one at the bottom is illustrated. The pressure distribution in such a tank is given in eq. 33 (liquid level h , vertical elevation z in the liquid space, pressure p_s at the surface of the liquid).

$$p(z) = (p_s + \rho gh) - \frac{1}{\rho gh} \cdot z \quad (33)$$

For an open tank, surface pressure equals ambient pressure $p_s = p_\infty$. The pressure at the bottom of the open tank therefore is

$$p|_{z=0} = p_\infty + \rho gh. \quad (34)$$

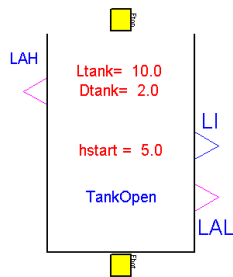


Figure 12: Open tank

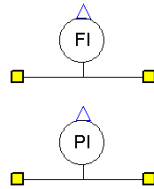


Figure 13: Sensors

Fig. 12 gives a graphical representation of this tank model. Output connectors are added to signal tank level (LI) as well as low and high level alarms (Boolean signals; LAL, LAH). Note, this tank model imposes no constraints on its level and does allow both liquid in- and outflow at its lower flange, depending on the gradient of inner to outer pressure conditions. Clearly, the upper flange is always above the liquid surface in the tank (unless the tank is completely full), and therefore, no liquid outflow is possible at the top.

3.5 Additional Models

A variety of additional component models were created, among them, sensors (fig. 13 shows flow and pressure indicators, realized with or without sensor dynamics), no-return valves (analog to the diode in the electric domain), various switches and commutators to control fluid routing, pipe diameter changes, tanks with constraints on levels and flow directions at its flanges, pressurized tanks and relief valves. Some of the models are thoroughly tested, others have a somewhat experimental character.

4 Modeling and Simulation Examples

A few—rather simple—models shall first be given to demonstrate the basic functionality of the library, before progressing to more comprehensive applications.

4.1 Introductory Demonstration Models

Tank Emptying Fig. 14 shows a model consisting of a tank with a pipe connected to its lower flange. Boundary conditions set the pressure at the outlet of the pipe to ambient pressure and the inflow rate at the upper flange to zero. Initial tank level is $h_s = 5.0m$. As expected, during simulation, the tank level decreases and it can be seen in fig. 15 that the tank becomes empty after about 6000s (pipe of 4cm inner diameter and 10m length).

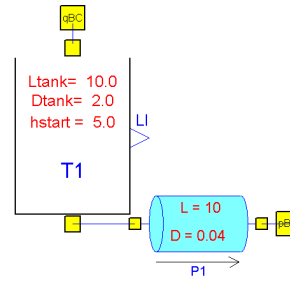


Figure 14: Model one

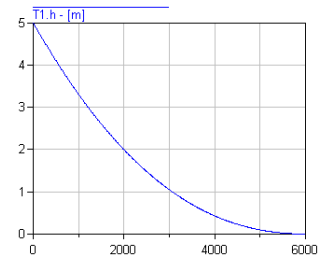


Figure 15: Emptying

Tank Level Equalization In the next model, two tanks of the same dimensions are connected with a pipe at their lower flanges (fig. 16). Initial levels are $h_{1,s} = 8.0m$ and $h_{2,s} = 2.0m$. There is no inflow to the tanks at their upper flanges. During simulation, the levels approach and equalize, as can be seen in fig. 17.

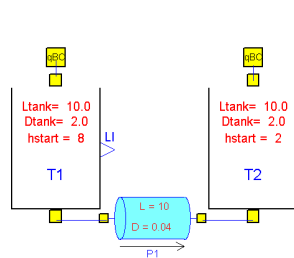


Figure 16: Model two

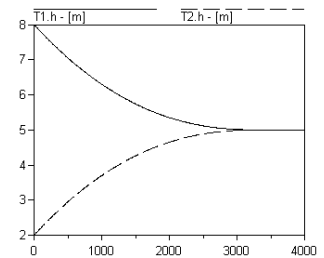


Figure 17: Equalization

Flow Inversion in a Pipe The model of fig. 16 is used again. This time, rather strong liquid inflow at the upper flange of tank 2 is imposed as a step $q_{2,in} = 0.01m^3/s$ at time $t = 1000s$. Fig. 18 and 19 show the tank levels, the inflow step and the flow rate in the connecting pipe. As can be seen, the behavior of the model is initially identical to the one illustrated above, after 1000s, the level in tank 2 increases faster due to the liquid added at its upper flange and surpasses the level of tank 1 after about 1550s, at which point in time the flow in the pipe is inverted. The simulation runs without numerical trouble through the point of flow inversion.

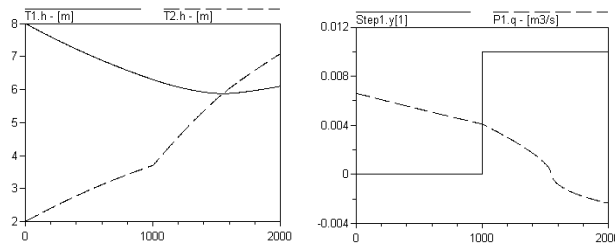


Figure 18: Tank levels

Figure 19: Step, flow

4.2 Laboratory Test-bed

A somewhat larger and more complex model is made of a laboratory test-bed which is currently installed at our institute. It consists of three tanks, a variety of connecting horizontal and vertical pipe segments, a pump and four continuous control valves. Fig. 20 depicts the PI¹¹ flow diagram, fig. 21 gives a 3-D view (CAD-draw from the planning phase) and fig. 22 shows the test-bed in its recent early commissioning phase¹². A respective Modelica/Dymola model is shown in fig. 23.

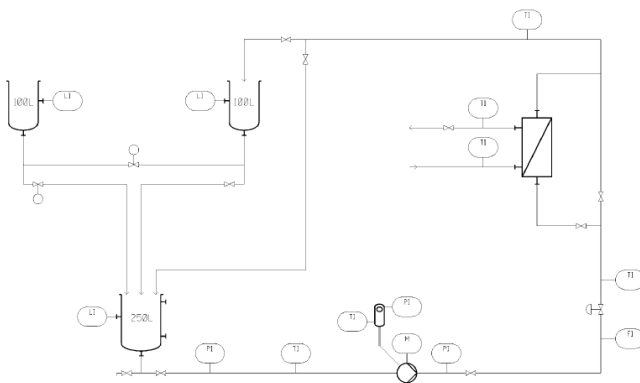


Figure 20: PI scheme of test-bed

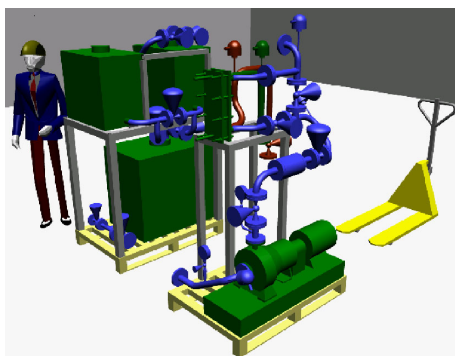


Figure 21: 3-D picture of the test-bed

For control scheme investigation, one flow (“FC” for flow control) and two level (“LC”) controllers are in-

¹¹Process and Instrumentation.

¹²The design in the flow diagram and the actual realization of the test-bed differ somewhat, but this is not of importance here.



Figure 22: Test bed: Commissioning phase

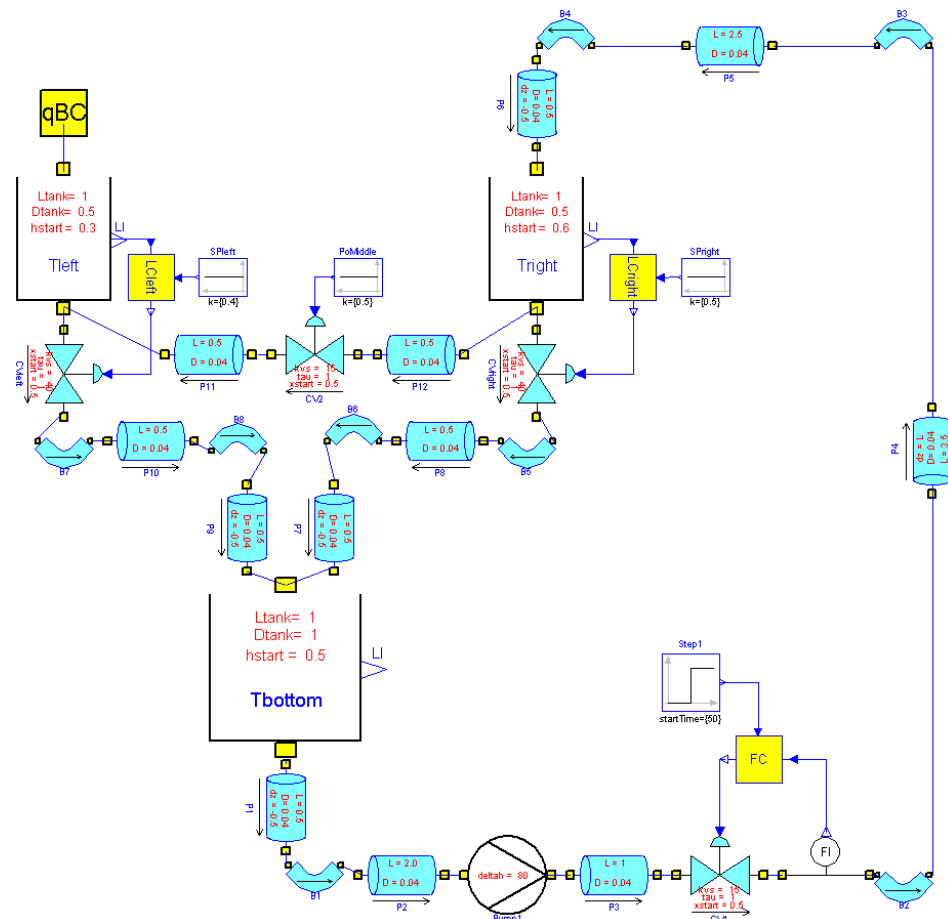
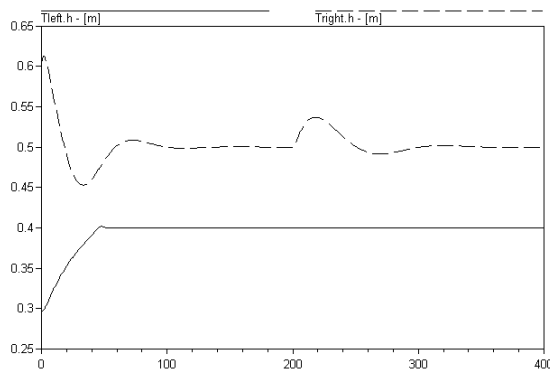
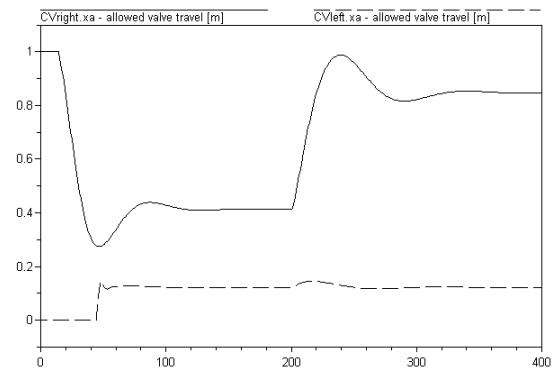
cluded in the model (PID-type controllers with limited output and anti-windup from the standard Modelica library with continuous control valves as actuators). The valves all have positioning dynamics (1st-order lag) with time constants $\tau = 1s$ (fast valve with pneumatic drive) and $k_{vs} = 40m^3/s$ for LC and $k_{vs} = 15m^3/s$ for FC. LC shall keep the levels in the upper tanks “Tleft” and “Tright” at their desired set-points ($T_{left,sp} = 0.4m$, $T_{right,sp} = 0.5m$). FC is used to control the flow rate through the pump (centrifugal pump, modeled as an ideal pressure source with hydraulic head of $\Delta h_{pump} = 80m$). After some simulation experiments, PID control parameters (gain k , integrative and derivative time constants T_i and T_d) were chosen as indicated in table 3.

LC	k	50
	T_i	5
	T_d	0.1
FC	k	100
	T_i	5
	T_d	0.1

Table 3: Controller parameters

Fig. 24 and fig. 25 show the upper tank levels and the respective control valve positions for a simulation experiment of 400s duration. Initial levels of the upper tanks are set to $h_{left,s} = 0.3m$ and $h_{right,s} = 0.6m$. A step is imposed on the set-point of the flow controller, at time $t = 200s$ (increase from $q_{sp1} = 0.00175$ to $q_{sp2} = 0.00275m^3/s$). With the controller parameters given above, the levels are kept at their set-points in a satisfactory fashion.

It can be seen in fig. 23, the system topology (as illustrated in the PI flow diagram and on the photograph) is maintained in the Modelica model. The model presents itself in a graphical manner so that its structure can be very quickly and intuitively understood. This is inherent to the physical object-oriented modeling paradigm

Figure 23: *Dymola model of testbed*Figure 24: *Levels in the upper tanks*Figure 25: *Level control valve positions*

of Modelica and regarded as very beneficial. The effort necessary to compose the test-bed model with the library presented was minor. Compare this to the workload if the modeler had to analytically and manually derive the overall system equations! Such a proceeding would probably be time-consuming and error prone even for this still rather simple three-tank model, especially because there is non-linearities and hybrid phenomena to account for as well. The library therefore can greatly reduce the modeling burden and increase efficiency of carrying out engineering design tasks (in

this case, controller design and investigation of dynamic behavior).

Furthermore, it has been shown (e.g., in [Ti00]) that the Modelica modeling approach allows to scale models to many thousands, if not hundreds of thousands of defining equations. This is of great importance if trying to model large and complex systems.

The presented test-bed is not yet operational in reality; it will be interesting to validate the model against experimental data and possibly refine it at a later time.

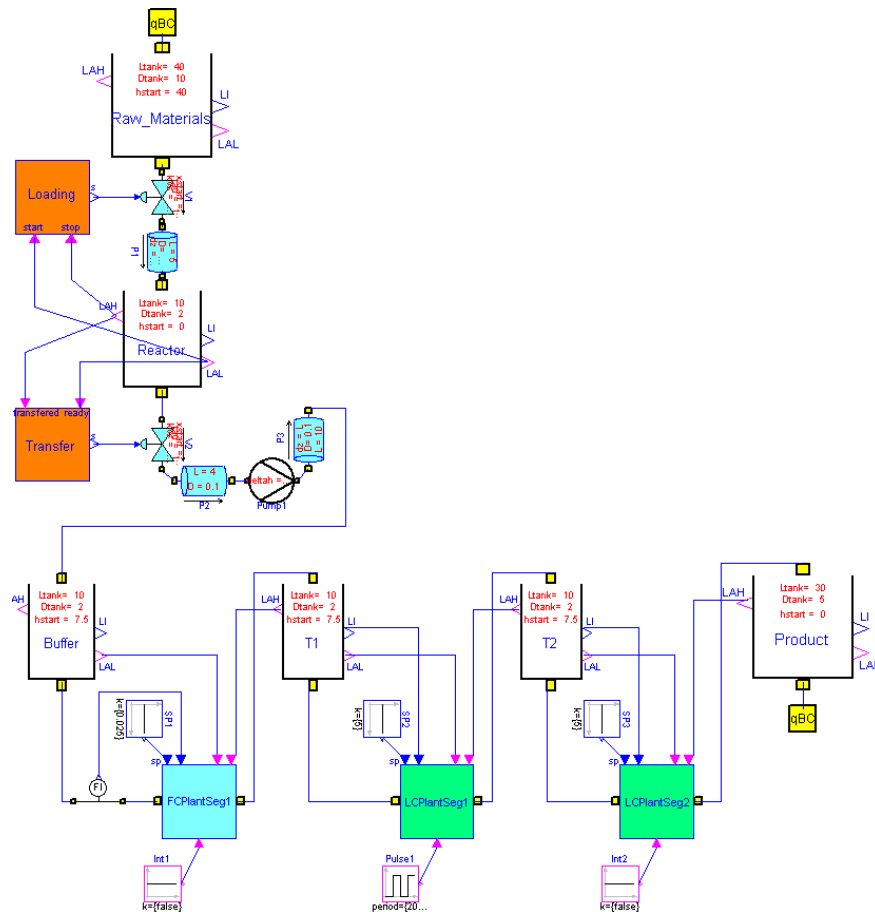


Figure 26: *Plant model top view*

4.3 Modeling of Mass Flow in a Process Plant

For purposes of maintenance management in process plants, it is advantageous to have a good knowledge of plant dynamics. Process plants often can be considered as interconnected flow segments where the linking elements are vessels¹³. Since the flow segments are decoupled from each other to some degree (depending on the sizes of intermediate buffer vessels as well as on the throughput capacities of flow segments), it can be beneficial to take mass flow dynamics into account when defining maintenance strategies and scheduling maintenance tasks.

Fig. 26 illustrates a model of a plant with both a batch and continuous part, consisting of a tank with raw materials, a reactor, a buffer tank enabling synchronization between batch and continuous part, two intermediate tanks in the continuous part and a large storage vessel for the final product. Note, it is not necessary to see all the details in fig. 26, rather, the aim is to illustrate the overall plant structure and the modeling concept.

The loading of the reactor with raw material is organized by control logic in the block “Loading” and the transfer of the reactor content after reaction is completed is controlled by block “Transfer”; the internals of the latter are shown fig. 27. As can be seen, the control logic is realized with Petri net formalism, based on the Modelica library described in [Mo98]. In particular, the reaction duration is modeled as delayed transition firing (transition “TReactionDelay”)¹⁴.

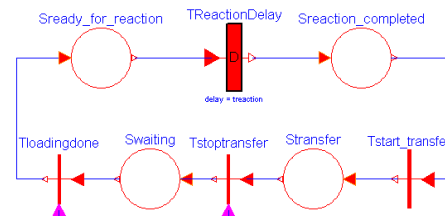


Figure 27: *Transfer control logic*

Two types of flow segments are defined, namely, a flow and a level controlled variant (“FCPlantSeg”, “LCPlantSeg”). Without going into details, it shall be said

¹³An abstraction into flow and storage elements as basic modeling blocks is extensively used in systems dynamics, e.g., see [Fo61].

¹⁴The Modelica Petri net library was extended in this context, supporting timed stochastic Petri nets and places with multiple token capacity, this aspect will be published in the near future.

here, the segments can contain whatever non-storage library component presented so far, but normally consist of at least a flow driver (pump), flow resistor (process steps and pipes) as well as control logic. The flow segment blocks receive measured flow or level signals, reference values as well as level alarms from neighboring vessels as inputs. In addition, an external Boolean signal can be supplied to indicate flow interruption, which can be useful to include deterministic or stochastic component and subsystem failure models in the plant model.

A simulation experiment with a flow interruption in the middle flow segment ("LCPlantSeg1") is carried out for a total simulation duration of 3000 time units. Fig. 28 shows the resulting tank levels, it can be seen how the level in the reactor and the subsequent buffer tank fluctuate. As well, the effect and propagation of the flow interruption (1000 to 1800 time units) in the plant is shown.

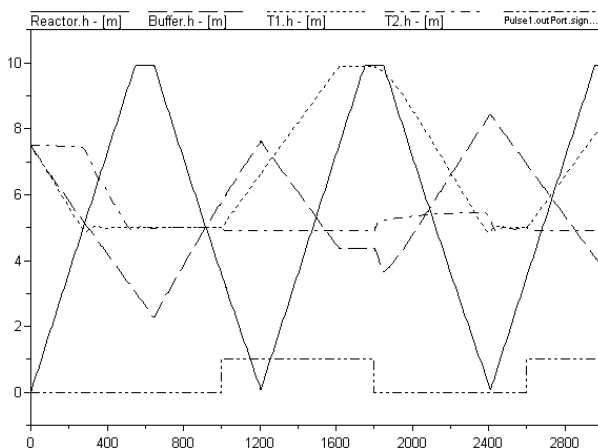


Figure 28: Tank levels in the plant

Plant control strategy, flow segment throughput capacities and tank sizes all influence the sensitivity of the plant to interruptions (e.g., caused by component failures or maintenance action) in flow segments, which can be analyzed with models as just described. The situation of course is yet more interesting if internal feedback loops are present in the plant.

5 Conclusion and Outlook

A library for mass flow simulation in process plants was presented and it has been shown how it can be used to efficiently generate models of possibly large and complex systems.

The presented plant models could be extended to include important energy exchanging equipment (heaters, coolers), more detailed component models as well as stochastic Petri nets for component failure and repair processes.

An important question to be dealt with in the future, is how to efficiently handle the great bandwidth of char-

acteristic times in a system such as a process plant. E.g., components often have life-times (between component failures or preventive overhauls) in the range of month to years, whereas the time constants of buffer tanks are minutes to hours, the controller and actuator dynamics seconds to minutes. If models were to be used for maintenance strategy optimization accounting for plant mass flow dynamics, it is desirable to have models which allow for rapid progress in simulation time for normal, nominal system behavior, switching to others, more detailed ones, in the case of faults or failures; multi-mode models are needed for this.

Another non-trivial topic is initial value calculation, especially as here for the case of quasi-steady state flow models, where nonlinear algebraic equations must be solved for iteratively, to calculate consistent initial conditions. The tool Dymola seems to be quite sensitive to choice of starting values, which is inconvenient in large models. It is hoped, the tool will be further equipped with even stronger algorithms in the future to support the simulation analyst in this respect.

References

- [Ge85] Geiger G. (1985). *Technische Fehlerdiagnose mittels Parameterschätzung und Fehlerklassifikation am Beispiel einer elektrisch angetriebenen Kreislumpumpe*. VDI Verlag, Düsseldorf, Germany
- [Fa01] Fabricius S.M.O., Badreddin E. (2001). *Stochastic Petri Net Modeling for Availability and Maintainability Analysis*. Proceedings of 14th COMADEM, September 2001, Manchester, UK
- [Fo61] Forrester J.W. (1961). *Industrial Dynamics*. M.I.T. Press and John Wiley & Sons, New York, U.S.A.
- [Mo98] Mosterman P.J., Otter M., Elmquist H.: (1998). *Modeling Petri Nets as Local Constraint Equations for Hybrid Systems Using Modelica*. Summer Computer Simulation Conference, July 19-22, S. 314-319, 1998 Reno, Nevada, U.S.A.
- [Ti00] Tiller M., Bowles P. et al. (2000). *Detailed Vehicle Powertrain Modeling in Modelica*. First int. Modelica Workshop 2000, October 23 - 24, 2000, Lund University, Lund, Sweden
- [Th99] Thomas P. (1999). *Simulation of Industrial Processes for Control Engineers*. Butterworth-Heinemann, Oxford, UK
- [Tu98] Tummescheit H., Eborn, J. (1998). *Design of a Thermo-Hydraulic Model Library in Modelica*. The 12th European Simulation Multiconference, ESM '98, June 16-19, 1998, Manchester, UK
- [Yo01] Young D.F., Munson B.R., Okiishi T.H. (2001). *A brief introduction to fluid mechanics*. John Wiley & Sons, New York, U.S.A.

Moving Boundary Models for Dynamic Simulations of Two-Phase Flows

Jakob Munch Jensen[†] and Hubertus Tummescheit[‡]

[†]Department of Mechanical Engineering
Technical University of Denmark
jmj@mek.dtu.dk

[‡]Department of Automatic Control
Lund University, Sweden
hubertus@control.lth.se

Abstract

Two-phase flows are commonly found in components in energy systems such as evaporators and boilers. The performance of these components depends among others on the controller. Transient models describing the evaporation process are important tools for determining control parameters, and fast low order models are needed for this purpose. This article describes a general moving boundary (MB) model for modeling of two-phase flows.

The new model is numerically fast compared to discretized models and very robust to sudden changes in the boundary conditions. The model is a 7th order model (7 state variables), which is a suitable order for control design. The model is also well suited for open loop simulations for systems design and optimization. It is shown that the average void fraction has a significant influence on the system response. A new method to calculate the average void fraction including the influence of the slip ratio is given. The average void fraction is calculated as a symbolic solution to the integral of the liquid fraction profile.

1 Introduction

First principle mathematical models of dynamical systems are made for a range of purposes, but one of the most common ones is to develop and verify controllers. The complexity of the model should be in accordance with the purpose of the model and this simple principle suggests that models for control design should be of low order and preferably easy to linearize. Unfortunately, physical systems are not sticking to this class of models, on the contrary: most mathematical first principle models are of distributed nature. The natural way to describe such a model is partial differ-

ential equations (PDE). PDE are infinite dimensional and their common numerical approximations, spatially discretized PDEs using one of the many possible discretization schemes, are of high order and without further model reduction not well suited for control design. The problem of control-oriented modeling is to derive a model which at the same time fulfills the requirements of control theory and characterizes those features of the system which are needed to satisfy the controller specification.

Moving boundary models for two phase flows in heat exchangers are a good example of low order control design models. They can be used for evaporators, condensers and steam generators. Their only disadvantage is that a number of mathematically rather different models arise depending on the operating conditions of the heat exchanger and the fluid conditions at the inlet of the equipment.

The model presented in this paper covers the most general case of two-phase heat exchangers with subcooled liquid at the inlet and superheated vapour at the outlet. This flow configuration is commonly found in thermal power systems, and the model can easily be extended to condensers and heat exchangers with subcooled liquid at the inlet and two-phase at the outlet. The special case of dry-expansion evaporators for refrigeration has been derived in [6].

The idea of a moving boundary model is to dynamically track the lengths of the different regions in the heat exchanger: the length from the inflow to the onset of boiling and the length of the two phase region. Simulation results are given for an evaporator in an organic rankine cycle, which utilizes the waste heat from a gas turbine in a small power plant. Other references to MB models include B.T. Beck that describes a MB-model for incomplete vaporization [2], a two region MB-model by He [4] and a three region model by Willatzen [7].

Roman and Greek Letters			
A	area	h	enthalpy
C_v	nozzle coef.	\dot{m}	mass flow
C_w	heat cap. of wall	q	heat flux
D	diameter	t	time
L	length	v	velocity
S	slip ratio	x	mass fraction
V_{cyl}	cylinder volume	z	length coordinate
α	heat transfer coef.	ρ	density
η	liquid fraction	ω	pump speed
η_v	volumetric efficiency	Φ	dissipation function
γ	void fraction	Ψ	vapour generation
μ	density ratio		
Subscripts			
1	subcooled	i	inner
2	two-phase	in	inlet
3	superheated	l	saturated liquid
12	interface 1-2	o	outer
23	interface 2-3	out	outlet
amb	ambient	r	refrigerant
c	condensation	w	wall
g	saturated gas		
Superscripts			
$'$	flux per length	$''$	flux per area
$'''$	flux per volume		

Table 1: Notation used in the Moving Boundary Model

2 Governing Equations

The general differential mass balance is

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \vec{v}) = 0 \quad (1)$$

which for the one-dimensional case can be written as

$$\frac{\partial A\rho}{\partial t} + \frac{\partial \dot{m}}{\partial z} = 0 \quad (2)$$

The general differential energy balance is

$$\frac{\partial \rho h}{\partial t} + \nabla \cdot (\rho h \vec{v}) = -\nabla \cdot \vec{q}'' - q''' + \frac{Dp}{Dt} + \Phi \quad (3)$$

which can be simplified by neglecting the axial conductivity, radiation and the viscous stresses and assuming one dimensional flow:

$$\frac{\partial (A\rho h - A p)}{\partial t} + \frac{\partial \dot{m} h}{\partial z} = \pi D \alpha (T_w - T_r). \quad (4)$$

A simplified differential energy balance for the wall is achieved by setting all flow terms in (3) equal to zero and neglecting the axial conductivity.

$$C_w \rho_w A_w \frac{\partial T_w}{\partial t} = \alpha_i \pi D_i (T_r - T_w) + \alpha_o \pi D_o (T_{amb} - T_w) \quad (5)$$

Equations (2), (4) and (5) are the differential balance equations, which will be integrated over the three regions to give the general three region lumped model for a two-phase heat exchanger. A schematic of the model is given in Figure 2. It is assumed in the following analysis that the change in pressure along the evaporator pipe is negligible.

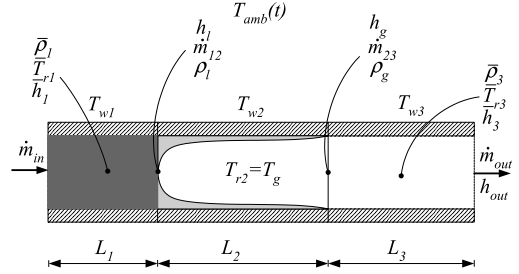


Figure 1: Schematic of the Three Region MB-model. 1 : subcooled, 2 : two-phase and 3 : superheated.

2.1 Mass Balance for the Subcooled Region

Integration of the mass balance (2) over the subcooled region gives

$$\int_0^{L_1} \frac{\partial (A\rho)}{\partial t} dz + \int_0^{L_1} \frac{\partial \dot{m}}{\partial z} dz = 0 \quad (6)$$

Applying Leibniz's rule (see Appendix A) on the first term and integrating the second term give for a constant area pipe:

$$A \frac{d}{dt} \int_0^{L_1} \rho dz - A \rho(L_1) \frac{dL_1}{dt} + \dot{m}_{12} - \dot{m}_{in} = 0. \quad (7)$$

The density at the interface $\rho(L_1)$ is equal to the saturated liquid density ρ_l . Pressure and mean enthalpy \bar{h}_1 define the state of the subcooled region where

$$\bar{h}_1 = \frac{1}{2} (h_{in} + h_l) \quad (8)$$

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

$$\bar{\rho}_1 = \frac{1}{L_1} \int_0^{L_1} \rho dz \approx \rho(p, \bar{h}_1) \quad (9)$$

The mean temperature is calculated from the same states as $\bar{T}_1 \approx T(p, \bar{h}_1)$. The mass balance for the subcooled region can be rewritten as

$$A \left\{ (\bar{\rho}_1 - \rho_l) \frac{dL_1}{dt} + L_1 \frac{d\bar{\rho}_1}{dt} \right\} = \dot{m}_{in} - \dot{m}_{12} \quad (10)$$

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

$$\begin{aligned} \frac{d\bar{\rho}_1}{dt} &= \left. \frac{\partial \bar{\rho}_1}{\partial p} \right|_h \frac{dp}{dt} + \left. \frac{\partial \bar{\rho}_1}{\partial \bar{h}_1} \right|_p \frac{d\bar{h}_1}{dt} \\ &= \left(\left. \frac{\partial \bar{\rho}_1}{\partial p} \right|_h + \frac{1}{2} \left. \frac{\partial \bar{\rho}_1}{\partial \bar{h}_1} \right|_p \frac{dh_l}{dp} \right) \frac{dp}{dt} \\ &\quad + \frac{1}{2} \left. \frac{\partial \bar{\rho}_1}{\partial \bar{h}_1} \right|_p \frac{dh_{in}}{dt} \end{aligned} \quad (11)$$

The term dh_{in}/dt is determined from the boundary conditions to the heat exchanger model. The expression for $d\bar{\rho}_1/dt$ is inserted into the mass balance (10), such that the final mass balance for the subcooled region reads:

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

2.2 Energy Balance for the Subcooled Region

Integration of the energy balance (4) over the subcooled region gives

$$\begin{aligned} \int_0^{L_1} \frac{\partial(A\rho h - A p)}{\partial t} dz + \int_0^{L_1} \frac{\partial \dot{m} h}{\partial z} dz \\ = \int_0^{L_1} \pi D \alpha (T_w - T_r) dz. \end{aligned} \quad (13)$$

Applying Leibniz's rule on the first term and integrating the other terms give for a constant area pipe and a constant heat transfer coefficient α

$$\begin{aligned} A \frac{d}{dt} \int_0^{L_1} \rho h dz - A \rho (L_1) h(L_1) \frac{dL_1}{dt} - A L_1 \frac{dp}{dt} \\ + \dot{m}_{12} h_l - \dot{m}_{in} h_{in} = \pi D \alpha L_1 (T_{w1} - \bar{T}_{r1}). \end{aligned} \quad (14)$$

The first two terms are evaluated as

$$\begin{aligned} \frac{d}{dt} \int_0^{L_1} \rho h dz - \rho(L_1) h(L_1) \frac{dL_1}{dt} \\ = \frac{d}{dt} (\bar{\rho}_1 \bar{h}_1 L_1) - \rho_l h_l \frac{dL_1}{dt} \\ = \left(\frac{1}{2} \bar{\rho}_1 (h_{in} + h_l) - \rho_l h_l \right) \frac{dL_1}{dt} \\ + \frac{1}{2} L_1 \left(\bar{\rho}_1 + \frac{1}{2} (h_{in} + h_l) \left. \frac{\partial \bar{\rho}_1}{\partial \bar{h}_1} \right|_p \right) \frac{dh_{in}}{dt} \\ + \frac{1}{2} L_1 \left\{ \bar{\rho}_1 \frac{dh_l}{dp} + (h_{in} + h_l) \right. \\ \left. \left(\left. \frac{\partial \bar{\rho}_1}{\partial p} \right|_h + \frac{1}{2} \left. \frac{\partial \bar{\rho}_1}{\partial \bar{h}_1} \right|_p \frac{dh_l}{dp} \right) \right\} \frac{dp}{dt}. \end{aligned} \quad (15)$$

where $\bar{\rho}_1 \bar{h}_1 \approx \overline{\rho_1 h_1} = \int_0^{L_1} \rho h dz$. The above equation (15) is inserted into the energy balance (14), which gives the final energy balance for the subcooled region:

$$\begin{aligned} \frac{1}{2} A \left\{ \left(\bar{\rho}_1 (h_{in} + h_l) - 2 \rho_l h_l \right) \frac{dL_1}{dt} \right. \\ \left. + \left(\bar{\rho}_1 L_1 + \frac{\partial \bar{\rho}_1}{\partial h} \right) \frac{dh_{in}}{dt} \right. \\ \left. + L_1 \left\{ \bar{\rho}_1 \frac{dh_l}{dp} + (h_{in} + h_l) \cdot \right. \right. \\ \left. \left. \left(\left. \frac{\partial \bar{\rho}_1}{\partial p} \right|_h + \frac{1}{2} \left. \frac{\partial \bar{\rho}_1}{\partial h} \right|_p \frac{dh_l}{dp} - 2 \right) \right\} \frac{dp}{dt} \right\} \\ = \dot{m}_{in} h_{in} - \dot{m}_{12} h_l + \pi D \alpha L_1 \alpha_{i1} (T_{w1} - \bar{T}_{r1}). \end{aligned} \quad (16)$$

2.3 Mass and Energy Balances for the Two-Phase and Superheated Regions

The mass and energy balances are integrated over the two-phase region and the superheated region using the same procedure as for the subcooled region. The equations are derived in detail in Appendix B.

The flow in the two-phase region is assumed to be homogeneous at equilibrium conditions with a mean density of $\bar{\rho} = \bar{\gamma} \rho_g + (1 - \bar{\gamma}) \rho_l$, where the void fraction is defined as $\gamma = A_g/A$. The average void fraction is defined as $\bar{\gamma} = \frac{1}{L_2} \int_{L_1}^{L_1+L_2} \gamma dz$ and is assumed to be invariant with time. A detailed model of the calculation of the void fraction is derived in section 2.5. The mass balance for the two-phase region is

$$\begin{aligned} A \left\{ (\rho_l - \rho_g) \frac{L_1}{dt} + (1 - \bar{\gamma}) (\rho_l - \rho_g) \frac{dL_2}{dt} \right. \\ \left. + L_2 \left(\bar{\gamma} \frac{d\rho_g}{dp} + (1 - \bar{\gamma}) \frac{d\rho_l}{dp} \right) \frac{dp}{dt} \right\} = \dot{m}_{12} - \dot{m}_{23} \end{aligned} \quad (17)$$

and the energy balance for the two-phase region is

$$\begin{aligned} A \left\{ L_2 \left\{ \bar{\gamma} \frac{d(\rho_g h_g)}{dp} + (1 - \bar{\gamma}) \frac{d(\rho_l h_l)}{dp} - 1 \right\} \frac{dp}{dt} \right. \\ \left. + \left\{ \bar{\gamma} \rho_g h_g + (1 - \bar{\gamma}) \rho_l h_l \right\} \frac{dL_1}{dt} \right. \\ \left. + \left\{ (1 - \bar{\gamma}) (\rho_l h_l - \rho_g h_g) \right\} \frac{dL_2}{dt} \right. \\ \left. = \dot{m}_{12} h_l - \dot{m}_{23} h_g + \pi D \alpha_{i2} L_2 (T_{w2} - T_{r2}) \right. \end{aligned} \quad (18)$$

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

from the pressure. The mass balance for the superheated region reads:

$$A \left\{ L_3 \left(\frac{1}{2} \frac{\partial \bar{\rho}_3}{\partial \bar{h}_3} \Big|_p \frac{dh_g}{dp} + \frac{\partial \bar{\rho}_3}{\partial p} \Big|_h \right) \frac{dp}{dt} + (\rho_g - \bar{\rho}_3) \frac{dL_1}{dt} + (\rho_g - \bar{\rho}_3) \frac{dL_2}{dt} + \frac{1}{2} L_3 \frac{\partial \bar{\rho}_3}{\partial \bar{h}_3} \Big|_p \frac{dh_{out}}{dt} \right\} = \dot{m}_{23} - \dot{m}_{out}. \quad (19)$$

The energy balance for the superheated region is given by

$$A \left\{ \left(\rho_g h_g - \frac{1}{2} \bar{\rho}_3 (h_g + h_{out}) \right) \left(\frac{dL_1}{dt} + \frac{dL_2}{dt} \right) + L_3 \left[\frac{1}{2} (h_g + h_{out}) \left(\frac{1}{2} \frac{\partial \bar{\rho}_3}{\partial \bar{h}_3} \Big|_p \frac{dh_g}{dp} + \frac{\partial \bar{\rho}_3}{\partial p} \Big|_h \right) + \frac{1}{2} \bar{\rho}_3 \frac{dh_g}{dp} - 1 \right] \frac{dp}{dt} + \left(\frac{1}{2} \bar{\rho}_3 L_3 + \frac{1}{4} \frac{\partial \bar{\rho}_3}{\partial \bar{h}_3} \Big|_p (h_g + h_{out}) L_3 \right) \frac{dh_{out}}{dt} \right\} = \dot{m}_{23} h_g - \dot{m}_{out} h_{out} + \pi D_i \alpha_{i3} L_3 (T_{w3} - \bar{T}_{r3}) \quad (20)$$

The mean properties of the superheated region are calculated in the same way as in the subcooled region. Thus $\bar{h}_3 = 0.5(h_g + h_{out})$, $\bar{\rho}_3 \approx \rho(p, \bar{h}_3)$ and $\bar{T}_{r3} \approx T(p, \bar{h}_3)$.

2.4 Energy Balance for the Wall Regions

The energy balances for the walls are derived. Integration of the wall energy equation (5) from α to β gives

$$\int_{\alpha}^{\beta} C_w \rho_w A_w \frac{\partial T_w}{\partial t} dz = \int_{\alpha}^{\beta} \alpha_i \pi D_i (T_r - T_w) dz + \int_{\alpha}^{\beta} \alpha_o \pi D_o (T_{amb} - T_w) dz \quad (21)$$

Applying Leibniz's rule, assuming constant wall properties give and rearranging gives the general energy balance for a wall region:

$$C_w \rho_w A_w \left\{ (\beta - \alpha) \frac{dT_w}{dt} + (T_w(\alpha) - T_w) \frac{d\alpha}{dt} + (T_w - T_w(\beta)) \frac{d\beta}{dt} \right\} = \alpha_i \pi D_i (\beta - \alpha) (T_r - T_w) + \alpha_o \pi D_o (\beta - \alpha) (T_{amb} - T_w). \quad (22)$$

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

$$C_w \rho_w A_w \left\{ L_1 \frac{dT_{w1}}{dt} + (T_{w1} - T_w(L_1)) \frac{dL_1}{dt} \right\} = \alpha_i \pi D_i L_1 (T_{r1} - T_{w1}) + \alpha_o \pi D_o L_1 (T_{amb} - T_{w1}) \quad (23)$$

The wall temperature in the model is discontinuous at L_1 giving

$$T_w(L_1) = T_{w2} \text{ for } \frac{dL_1}{dt} > 0 \\ T_w(L_1) = T_{w1} \text{ for } \frac{dL_1}{dt} \leq 0 \quad (24)$$

Similar expressions are derived for the walls adjacent to the two-phase and the superheated regions see Appendix B. Typically in the literature a simplified mean value for $T_w(L_1)$ has been used, which seems attractive in order to simplify the model see e.g. [4] and [7]. Simulations show that the response times for the system for some test conditions depend significantly on the expression for $T_w(L_1)$, and the full equations given by (23) and (24) should therefore be used.

The general three region moving boundary model is described by the mass and energy balances for the flow stated in equations (12), (16), (17), (18), (19) and (20) and the energy balances for the wall regions as stated in equations (23), (49) and (51). In addition the two discontinuous equations for the wall temperatures as stated in (24) and (50) are needed. This equation system contains 9 equations with the 7 state variables: $(L_1, L_2, p, h_{out}, T_{w1}, T_{w2}$ and $T_{w3})$. The variable h_{in} , which also appears differentiated, is calculated as a boundary condition and is thus not included in the state variables for the MB-model. Dependent variables can be calculated from the state variables and include: $(\bar{\rho}_1, \rho_l, \rho_g, \bar{\rho}_3, \partial \bar{\rho}_1 / \partial \bar{h}_1|_p, \partial \bar{\rho}_1 / \partial p|_h, d\rho_l/dp, d\rho_g/dp, \partial \bar{\rho}_3 / \partial \bar{h}_3|_p, \partial \bar{\rho}_3 / \partial p|_h, \bar{h}_1, h_l, h_g, \bar{h}_3, dh_l/dp, dh_g/dp, \bar{T}_{r1}, \bar{T}_{r2}, \bar{T}_{r3}, \dot{m}_{12}, \dot{m}_{23})$. Parameters are constant during simulation and include: $(A, D_i, D_o, \alpha_{i1}, \alpha_{i2}, \alpha_{i3}, \alpha_o, \gamma, L, T_{amb}, C_w, \rho_w, A_w)$. The boundary models calculate the variables $(\dot{m}_{in}, \dot{m}_{out}, h_{in}$ and $dh_{in}/dt)$, which are boundary conditions to the MB-model.

2.5 Calculation of the Average Liquid Fraction $\bar{\eta}$

The liquid fraction in the two phase region $\eta(z)$ is related to the void fraction $\gamma(z)$ via the equation

$$\eta(z) + \gamma(z) = 1. \quad (25)$$

The same equation holds for the average values $\bar{\eta}$ and $\bar{\gamma}$ over the whole region, which are the parameters of interest. It is computed as the integral over the normalized profile. For the derivation of a $\eta(z)$ profile, a couple of assumptions are necessary:

1. All assumptions made in the derivation of the moving boundary model apply also to the derivation of the liquid fraction profile, in particular that a constant pressure is assumed along the pipe.
2. The profile can be evaluated under steady state conditions. For the purpose of slow, start-up transients as well as for linearization purposes this does not pose any restrictions. This means in particular that the pressure is in steady state.
3. The vapour generation rate Ψ' is uniform over the evaporator length.
4. The slip velocity ratio $S = u_g/u_l$ between the gas and the liquid velocities is constant along the evaporator length and a known function of the model states that also can be evaluated under steady state conditions¹.

A similar derivation but assuming a slip velocity ratio of 1 has been done in [3]. Under the above assumptions, the following coupled ODE boundary value problem holds:

$$\rho_l \frac{\partial(A_l u_l)}{\partial z} = -\Psi' \quad (26)$$

$$\rho_g \frac{\partial(A_g u_g)}{\partial z} = \Psi' \quad (27)$$

Ψ' is the net generation of saturated vapour per unit length $[kg/(ms)]$, A_l and A_g are the cross sectional areas taken up by liquid and vapour respectively and the densities are independent of the length coordinate because we assumed no pressure loss and steady state conditions for the pressure. This equation is normalized by setting $A = A_l + A_g = 1$ and letting the length of the evaporation region run from 0 to 1 so that the cross section area $A_l(z)$ is now equivalent to the liquid volume fraction $\eta(z)$. Then, replacing u_l with u and u_g with Su and dividing by ρ_l the following normalized equation is obtained:

$$\frac{\partial(\eta u)}{\partial z} = -\Psi^* \quad (28)$$

$$\mu S \frac{\partial((1-\eta)u)}{\partial z} = \Psi^* \quad (29)$$

¹Remark: It is possible to weaken this assumption and use a slip ratio $S(z)$ which is a function of the length coordinate. Many of the rather complex empirical slip correlations depend on the local mass fraction $x = \dot{m}_g/\dot{m}$ as well and in this case the profile and the integral over the profile can only be solved numerically. For certain applications this may result in the most accurate approximation of the mean void fraction.

where

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

The boundary conditions at the length coordinates $z = 0.0$ and $z = 1.0$ are

$$\eta(0) = 1, \quad \eta(1) = 0. \quad (30)$$

From (28), (29) and the boundary conditions, the following function for $\eta(z)$ can be derived:

$$\eta(z) = \frac{1-z}{1+z\left(\frac{1}{S\mu} - 1\right)} \quad (31)$$

The influence of the slip ratio S on the amount of saturated liquid in the evaporation region, $\bar{\eta}$ can be seen in Figure 2. $\eta(z)$ can be integrated symbolically to give:

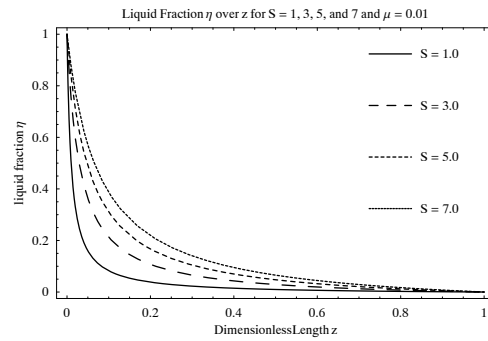


Figure 2: Liquid Fraction $\eta(z)$ along the normalized evaporation region.

$$\bar{\eta} = \int_0^1 \eta(z) dz = \frac{S\mu(S\mu - 1 - \ln(S\mu))}{(S\mu - 1)^2} \quad (32)$$

This $\bar{\eta}$ can only be used together with the dynamic model from the previous section when the time derivative of $\bar{\eta}$ can be neglected. This holds for slow pressure transients. The density ratio μ is a unique and simple function of the pressure, but for the slip ratio S a number of empirical correlations are available to choose from. Because of the assumptions made above, we have to choose a slip ratio which is independent of the local void fraction or mass fraction. A simple and appealing correlation is the one from Zivi (1964) cited in [9] which minimizes the total kinetic energy flow locally at each position z along the pipe:

$$S = \frac{u_g}{u_l} = \left(\frac{\rho_l}{\rho_g}\right)^{1/3} = \mu^{1/3} \quad (33)$$

Using this slip correlation, the average liquid fraction in the pipe becomes a function of only one variable,

the density ratio μ .

$$\bar{\eta} = \int_0^1 \eta(z) dz = \frac{1 + (1/\mu)^{2/3} (2/3 \ln(1/\mu) - 1)}{\left((1/\mu)^{2/3} - 1\right)^2} \quad (34)$$

Both the density ratio μ and the slip S approach 1 when the pressure is rising toward the critical pressure. In the limit, the liquid and vapour densities are equal as well as the flow speeds, so that a mean liquid fraction of 0.5 is expected, compare the plot of $\bar{\eta}$ in Figure 3.

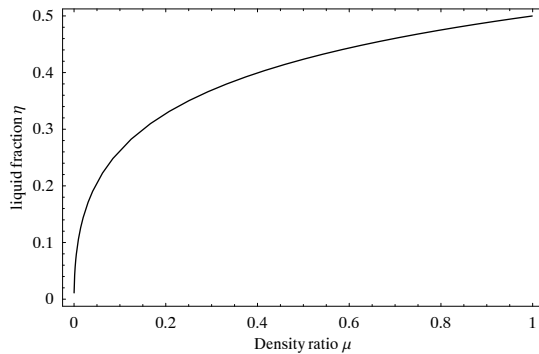


Figure 3: Average Liquid fraction $\bar{\eta}$ as a function of the density ratio μ .

3 Dominating Time Constants of the Linearized Model

In this section the influence of some model parameters on the eigenvalues of a linearization of the system derived in section 2 is investigated. Models for fluid flow exhibit two types of time constants: fast, hydraulic time constants for disturbances traveling with the speed of sound and much slower thermal ones, whose disturbances move at the flow speed. In two phase flows, the coupling between thermal and hydraulic phenomena is much tighter than in one phase flows, because a change in the hydraulic pressure is tightly coupled to a change in the temperature. The eigenvectors reveal that the 7 eigenvalues are tightly coupled, but roughly their physical interpretation is as follows:

- one mode comes from the overall mass balance of the evaporator which depends on the ratio between the total mass and the sum of the mass flows in and out of the evaporator and
- one for the overall energy balance which depends on the ratio of the total heat capacity to the sum of convective and heat transfer energy flows,

- one for each of the lengths of the subcooled and the two-phase regions. These are a combination of the mass and energy balances for the respective region.
- Three more eigenvalues come from the energy balances of the evaporator walls.

In [1] Bauer derived a more detailed, distributed model of heterogeneous flow² and validated it against measurement data for the refrigerant R22. According to [1], the advantage of the heterogeneous model over the homogeneous one is that the void fraction turns out to be more realistic. Therefore, the dominant time constants are modeled more accurately. The average void fraction $1 - \bar{\eta}$ has a strong influence on the total fluid mass in the evaporator, as can be seen clearly from Figure 4. It can be concluded from this argument that

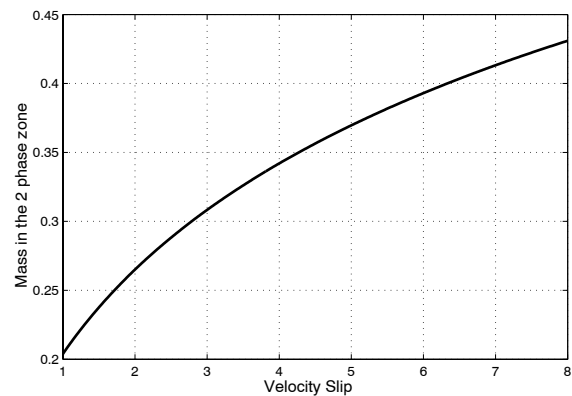


Figure 4: Total mass in the two-phase region as a function of velocity slip.

the void fraction $\bar{\gamma}$ is a crucial parameter in the moving boundary model. Using a good approximation of the void fraction, which may be obtained from a detailed, distributed model like in [1], is important for obtaining realistic dynamic behaviour.

The slow modes in the overall system are mostly influenced by the wall temperatures: higher heat capacities and smaller heat transfer coefficients result in slower modes. This means that the slowest mode usually is governed by the pipe wall in the liquid region. Two model parameters with a large influence on the slow time constants are the void fraction $\bar{\gamma}$ and the ratio of the total heat capacities of fluid and pipe walls of the evaporator pipes. The latter depend on the system pressure and the pipe diameter. Correct estimation of

²Heterogeneous flow means that the flow speeds of the gas and liquid phases can be different. A homogeneous flow assumption is equivalent to the same flow speed for both phases.

the void fraction gets more important at lower pressures because the slip increases due to smaller density ratio μ and the heat capacity of the pipes is usually smaller due to thinner pipe walls.

The root locus plot in Figure 5 shows the slow eigenvalues of the system, which are the dominating ones for control design purposes. These vary significantly when the slip ratio S (and thus the void fraction) is varied from 1.0 to 8.0. In the example with approx. 31 bars the pressure is relatively high for the working fluid R22 and therefore the slip ratio is not very far from 1. Nonetheless, the slow eigenvalues move considerably on the root locus. The change in the model dynamics will be larger at lower pressures.

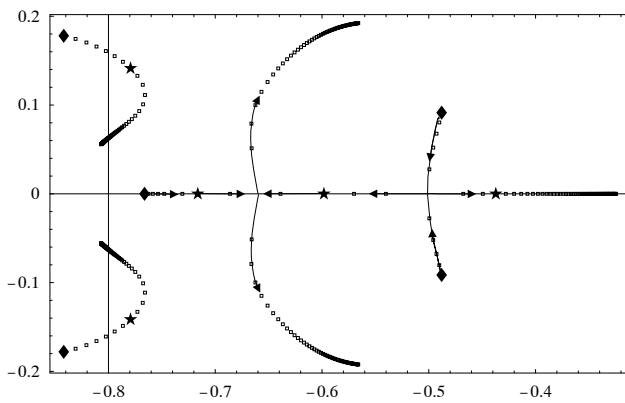


Figure 5: Root Locus for the slow eigenvalues. Diamonds mark a slip ratio of 1.0, stars a slip ratio of the test case of ≈ 1.7 . Slip ratios vary from 1.0 to 8.0.

4 Model Variants

The main effect of using a velocity slip estimate to calculate the average void fraction is an improved model of the fluid mass in the evaporator. In a model for control design around a narrow operating pressure – the short transient case – a constant void fraction based on the profile derived above, will give sufficiently accurate results. Different slip correlations than the one from above and numerical quadrature can be used to find a good estimate of the mean void fraction. A fixed void fraction will be less adequate when “long” transients over wide pressure ranges are to be simulated. During the simulation of the start-up of a near- or supercritical once-through boiler, which is a classical case for a moving boundary model (see [3]), the density ratio μ will change by 3 orders of magnitude. In that case the simple slip correlation $S = \mu^{1/3}$ works well to model the fluid mass in the evaporator.

The model derived above is still too complex and has too many states for some purposes, e.g., online dynamic optimization as it is done in Model Predictive Control (MPC). There are several ways to reduce the number of states in the moving boundary model. One possibility is to assume that the 2-phase heat transfer coefficient is much higher than the outer heat transfer coefficient so that the wall temperature and the fluid temperature in the evaporation region are equal. This assumption may also be extended to the subcooled and superheated regions. The model will lose accuracy in the high frequency range but will be very similar to the full model at low frequency range. Another possible simplification is to get rid of the states in the superheated region, because it is usually short and contains only few percent of the fluid mass. The dynamic model for the region can be replaced by a semi-empirical algebraic relation for the superheat temperature, see [4]. Investigation of these options for model reduction is the goal of future work by the authors.

4.1 Boundary Models

The test simulations for the heat exchanger are performed for a simple cycle containing a pump that supplies the liquid flow into the evaporator and a nozzle (turbine) at the end of the evaporator. The pump model is defined by a simple expression for the mass flow

$$\dot{m}_{pump} = \eta_v \rho_{pump} V_{cyl} \omega \quad (35)$$

where η_v is the volumetric efficiency, ρ_{pump} is the inlet density to the pump, V_{cyl} is the cylinder volume and ω is the number of revolutions per second. The specific enthalpy at the inflow of the evaporator is h_{in} and is a constant below the saturated liquid enthalpy.

The model of the nozzle is computed as:

$$\dot{m}_{nozzle} = C_v \sqrt{\rho_{out} (p - p_c)}. \quad (36)$$

where C_v is a coefficient, ρ_{out} is the outlet density from the evaporator, p is the pressure in the evaporator and p_c is a constant pressure lower than p .

5 Simulation Result

The evaporator in an organic rankine cycle (ORC) is simulated using the three region moving boundary model and the models for the pump and the nozzle. The simulation program Dymola [5] has been used to perform the simulations. The ORC is used to convert thermal energy to electric energy in applications

with small temperature differences between the high and the low temperature heat sources. The ORC can thereby be used to improve the energy efficiency in gas turbine power plants by converting the waste heat energy in the exhaust gas to electricity. Simulation results for a test case of an evaporator pipe are shown in Figure 6 to Figure 8.

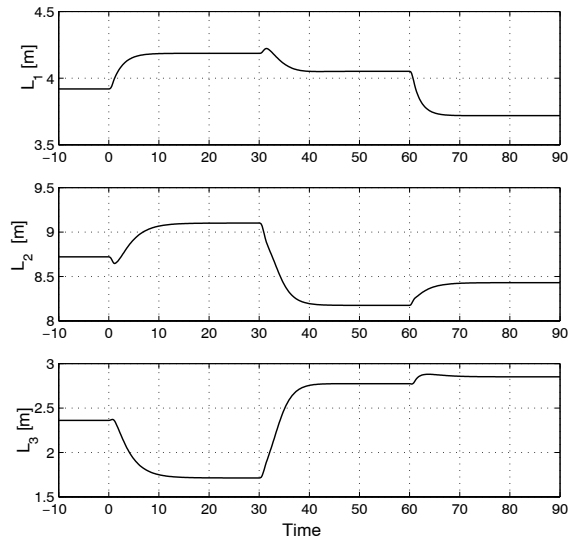


Figure 6: Lengths of the three regions

Pump and nozzle parameters	
$\eta_v = 0.6 [-]$	$C_v = 3.76E-5 [m^2]$
$\omega = 60 [rps]$	$p_c = 1.4 [Mpa]$
$V_{cyl} = 1.30E-5 [m^3]$	
Evaporator steady state results	
$L = 15 [m]$	$Q = 163 [kW]$
$D_i = 0.020 [m]$	$\dot{m} = 0.54 [kg/s]$
$D_o = 0.022 [m]$	$p = 3.6 [Mpa]$
$C_w = 385 [J/kgK]$	$T_{w1} = 388.8 [K]$
$\rho_w = 8.96E3 [kg/m^3]$	$T_{w2} = 371.7 [K]$
$\alpha_{i1} = 2451 [J/m^2K]$	$T_{w3} = 449.5 [K]$
$\alpha_{i2} = 11404 [J/m^2K]$	$T_{r1} = 306.0 [K]$
$\alpha_{i3} = 2071 [J/m^2K]$	$T_{r2} = 352.3 [K]$
$\alpha_o = 500 [J/m^2K]$	$T_{r3} = 384.0 [K]$
$T_{amb} = 573.1 [K]$	$L_1 = 3.9[m]$
$S = 1.67 [-]$	$L_2 = 8.7[m]$
$\bar{\gamma} = 0.665 [-]$	$L_3 = 2.4[m]$

Table 2: Parameters and steady-state results

Three experiments are performed with parameters and initial steady state results in table 2. At $t = 0$ s the pump speed ω is increased by 5%, at $t = 30$ s the outer heat transfer coefficient α_o is increased by 10% and at $t = 60$ s the nozzle coefficient C_v is increased by 10%. Figure 6, 7 and 8 show the transient response of the system.

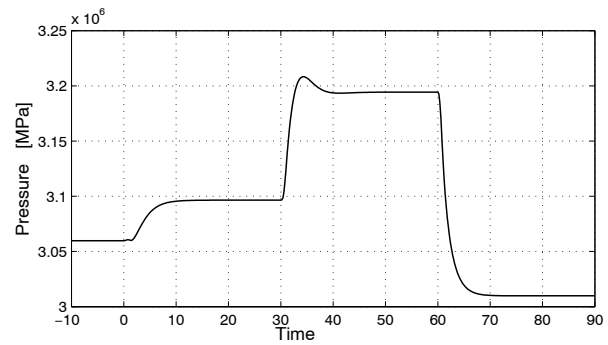


Figure 7: Pressure in the evaporator

The length of the subcooled and the two-phase region is seen to increase as the pump speed is increased (at $t=0$ s). Also the pressure and the heating effect is increased which is expected. An increase in outer heat transfer (at $t=30$ s) results in shorter two-phase and superheated regions as well as in increased heating effect. The larger nozzle coefficient ($t = 60$ s) results in a decrease in pressure. The reduced pressure lowers the boiling point and thus the fluid temperature in the evaporation region. The length of the subcooled region is therefore shrinking. The length of the two-phase region grows in this case but this trend depends on the conditions. The lower evaporating temperature tends to decrease the length of the two-phase region and the larger latent heat increases it. The heating effect rises in this case, but this trend depends on the conditions as well. The model gives the right trends even though no experimental data has been available to validate the model.

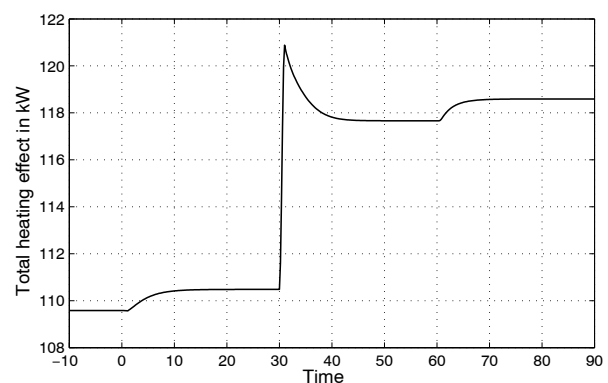


Figure 8: Heating effect to the evaporator

5.1 Conclusions

A new moving boundary model has been presented describing the dynamics of two phase heat exchang-

ers with liquid at the inlet and vapour at the outlet. The new model is numerically fast compared to discretized models and very robust to sudden changes in the boundary conditions. The model is a 7th order model (7 state variables), which is a suitable order for control design. The model is also well suited for open loop simulations for systems design and optimization. It is shown that the average void fraction has a significant influence on the system response. A new method to calculate the average void fraction including the influence of the slip ratio is presented. The average void fraction is computed from the symbolic solution to the integral of the liquid fraction profile.

Appendix A: Leibniz's Rule

Leibniz's rule for differentiation of integrals with time varying limits reads ([8]):

$$\frac{d}{dt} \int_{z_1}^{z_2} f(z, t) dz = f(z_2, t) \frac{dz_2}{dt} - f(z_1, t) \frac{dz_1}{dt} + \int_{z_1}^{z_2} \frac{\partial f(z, t)}{\partial t} dz. \quad (37)$$

Appendix B: Derivation of the Model Equations

Mass Balance for the Two-Phase Region

The mass balance (2) is integrated over the two-phase region from L_1 to $L_1 + L_2$. Applying Leibniz's rule gives for a constant area pipe

$$A \frac{d}{dt} \int_{L_1}^{L_1+L_2} \rho dz + A \rho(L_1) \frac{dL_1}{dt} - A \rho(L_1 + L_2) \frac{d(L_1 + L_2)}{dt} + \dot{m}_{23} - \dot{m}_{12} = 0 \quad (38)$$

The flow is assumed to be homogeneous at equilibrium conditions with a mean density of $\rho = \bar{\gamma} \rho_g + (1 - \bar{\gamma}) \rho_l$. The mass balance for the two-phase region becomes

$$A \left\{ \frac{d}{dt} (\rho_2 L_2) + (\rho_l - \rho_g) \frac{dL_1}{dt} - \rho_g \frac{dL_2}{dt} \right\} = \dot{m}_{12} - \dot{m}_{23} \quad (39)$$

where $\rho_2 = \bar{\gamma} \rho_g + (1 - \bar{\gamma}) \rho_l$. The time derivative of ρ_2 is

$$\frac{d\rho_2}{dt} = \left(\bar{\gamma} \frac{d\rho_g}{dp} + (1 - \bar{\gamma}) \frac{d\rho_l}{dp} \right) \frac{dp}{dt} \quad (40)$$

which inserted into the mass balance (39) gives the final mass balance for the two-phase region as stated in (17).

Energy Balance for the Two-Phase Region

The energy balance (4) is integrated over the two-phase region from L_1 to $L_1 + L_2$. Applying Leibniz's rule gives for a constant area pipe

$$A \frac{d}{dt} \int_{L_1}^{L_1+L_2} \rho h dz + A \rho(L_1) h(L_1) \frac{dL_1}{dt} - A L_1 \frac{dp}{dt} - A \rho(L_1 + L_2) h(L_1 + L_2) \frac{d(L_1 + L_2)}{dt} - A L_2 \frac{dp}{dt} + \dot{m}_{23} h_g - \dot{m}_{12} h_l = \pi D_i \alpha_{i2} L_2 (T_{w2} - T_{r2}) \quad (41)$$

The first term is evaluated as

$$\begin{aligned} \frac{d}{dt} \int_{L_1}^{L_1+L_2} \rho h dz &= \frac{d}{dt} \int_{L_1}^{L_1+L_2} (\bar{\gamma} \rho_g h_g + (1 - \bar{\gamma}) \rho_l h_l) dz \\ &= \frac{d}{dt} \left\{ (\bar{\gamma} \rho_g h_g + (1 - \bar{\gamma}) \rho_l h_l) L_2 \right\} \\ &= L_2 \left\{ \bar{\gamma} \frac{d(\rho_g h_g)}{dp} + (1 - \bar{\gamma}) \frac{d(\rho_l h_l)}{dp} \right\} \frac{dp}{dt} \\ &\quad + \left\{ \bar{\gamma} \rho_g h_g + (1 - \bar{\gamma}) \rho_l h_l \right\} \frac{dL_2}{dt} \end{aligned} \quad (42)$$

Inserting (42) into (41) gives the final energy balance for the two-phase region as state in (18).

Mass Balance for the Superheated Region

The mass balance (2) is integrated over the superheated region from $L_1 + L_2$ to L which for a constant area pipe gives

$$\int_{L_1+L_2}^L \frac{\partial A \rho}{\partial t} dz + \int_{L_1+L_2}^L \frac{\partial \dot{m}}{\partial z} dz = 0 \quad (43)$$

Applying Leibniz's rule on the first term and integrating the second term give for a constant area pipe

$$A \frac{d}{dt} \int_{L_1+L_2}^L \rho dz + A \rho(L_1 + L_2) \frac{d(L_1 + L_2)}{dt} + \dot{m}_{out} - \dot{m}_{23} = 0 \quad (44)$$

The mean density in the superheated region is $\rho_3 = \frac{1}{L_3} \int_{L_1+L_2}^L \rho dz \approx \rho(p, h_3)$, which inserted in the mass balance (44) gives

$$A \left\{ L_3 \frac{d\rho_3}{dt} + (\rho_g - \rho_3) \frac{dL_1}{dt} + (\rho_g - \rho_3) \frac{dL_2}{dt} \right\} = \dot{m}_{23} - \dot{m}_{out} \quad (45)$$

The derivative of ρ_3 is calculated as

$$\begin{aligned} \frac{d\rho_3}{dt} &= \frac{\partial \rho_3}{\partial p} \Big|_h \frac{dp}{dt} + \frac{\partial \rho_3}{\partial h} \Big|_p \frac{dh}{dt} \\ &= \left(\frac{1}{2} \frac{\partial \rho_3}{\partial h_3} \Big|_p \frac{dh_g}{dp} + \frac{\partial \rho_3}{\partial p} \Big|_h \right) \frac{dp}{dt} + \frac{1}{2} \frac{\partial \rho_3}{\partial h_3} \Big|_p \frac{dh_{out}}{dt} \end{aligned} \quad (46)$$

The expression for $\frac{d\rho_3}{dt}$ is inserted into (45), which gives the final mass balance for the superheated region as stated in (19).

Energy Balance for the Superheated Region

The energy equation 4 is integrated over the superheated region from $L_1 + L_2$ to L . Applying Leibniz's rule gives for a constant area pipe

$$\begin{aligned} A \frac{d}{dt} \int_{L_1+L_2}^L \rho h dz + A \rho (L_1 + L_2) h (L_1 + L_2) \frac{d(L_2)}{dt} \\ - A L_3 \frac{dp}{dt} + \dot{m}_{out} h_{out} - \dot{m}_{23} h_g \\ = \pi D_i \alpha_{i3} L_3 (T_{w3} - T_{r3}) \end{aligned} \quad (47)$$

The first term is calculated as

$$\begin{aligned} \frac{d}{dt} \int_{L_1+L_2}^L \rho h dz &= \frac{d}{dt} (\bar{\rho}_3 \bar{h}_3 L_3) \\ &= -\frac{1}{2} \bar{\rho}_3 (h_g + h_{out}) \left(\frac{d(L_1 + L_2)}{dt} \right) \\ &\quad + \frac{1}{2} L_3 (h_g + h_{out}) \frac{d\bar{\rho}_3}{dt} \\ &\quad + \frac{1}{2} \bar{\rho}_3 L_3 \left(\frac{dh_g}{dp} \frac{dp}{dt} + \frac{dh_{out}}{dt} \right) \end{aligned} \quad (48)$$

where $\bar{h}_3 = \frac{1}{2}(h_g + h_{out})$ and $\bar{\rho}_3 = \rho(p, \bar{h}_3)$. Equation (48) and the expression for $\frac{d\bar{\rho}_3}{dt}$ from equation (46) is inserted into (47), which after some rearranging gives the final energy balance for the superheated region as stated in (20).

Energy Balance for the Walls

For the wall region adjacent to the two-phase region $\alpha = L_1$ and $\beta = L_1 + L_2$, which inserted in (22) gives

$$\begin{aligned} C_w \rho_w A_w \left\{ L_2 \frac{dT_{w2}}{dt} + (T_w(L_1) - T_{w2}) \frac{dL_1}{dt} \right. \\ \left. + (T_{w2} - T_w(L_1 + L_2)) \frac{dL_2}{dt} \right\} \\ = \alpha_i 2\pi D_i L_2 (T_{r2} - T_{w2}) \\ + \alpha_o \pi D_o L_2 (T_{amb} - T_{w2}) \end{aligned} \quad (49)$$

$T_w(L_1)$ is given by (24), and $T_w(L_1 + L_2)$ is given by

$$\begin{aligned} T_w(L_1 + L_2) &= T_{w3} \text{ for } \frac{dL_2}{dt} > 0 \\ T_w(L_1 + L_2) &= T_{w2} \text{ for } \frac{dL_2}{dt} \leq 0 \end{aligned} \quad (50)$$

For the wall region adjacent to the superheated region $\alpha = L_1 + L_2$ and $\beta = L$, which inserted in (22) gives

$$\begin{aligned} C_w \rho_w A_w \left\{ L_3 \frac{dT_{w3}}{dt} + (T_w(L_1) - T_{w2}) \frac{dL_1}{dt} \right. \\ \left. + (T_w(L_1 + L_2) - T_{w3}) \left(\frac{dL_1}{dt} + \frac{dL_2}{dt} \right) \right\} \\ = \alpha_i 3\pi D_i L_3 (T_{r3} - T_{w3}) + \alpha_o \pi D_o L_3 (T_{amb} - T_{w3}) \end{aligned} \quad (51)$$

References

- [1] Olaf Bauer, *Modelling of Two-Phase Flows with Modelica*, Masters Thesis ISRN LUTFD2/TRFT-5629-SE, Department of Automatic Control, Lund University, November 1999.
- [2] B.T. Beck and G.L. Wedekind, *A generalization of the system mean void fraction model for transient two-phase evaporation flows*, Int. J. of Heat Transfer **103** (1981), 81 – 85.
- [3] S. Bittanti, M. Bottinelli, A. De Marco, M. Facchetti, and W. Prandoni, *Performance Assessment of the Control System of Once-Through Boilers*, Proceedings of the 13th Conference on Process Control '01, June 2001.
- [4] X.D. He and S. Liu, *Multivariable Control of Vapor Compression Systems*, HVAC Research **4** (1998), 205 – 230.
- [5] <http://www.dynasim.se>.
- [6] J.M. Jensen and H.J. Hoegaard Knudsen, *A new moving boundary model for transient simulations of dry-expansion evaporators*, Proceedings of the 15th International Conference on Efficiency, Costs, Optimization, Simulation and Environmental Impact of Energy Systems, July 2002.
- [7] N.B.O.L. Pettit M. Willatzen and L. Plougs-Sørensen, *A general dynamic simulation model for evaporators and condensers in refrigeration*, Int. J. of Refrigeration **21** (1998), 398 – 414.
- [8] N.E. Todreas and M.S. Kazimi, *Nuclear Systems I, Thermal Hydraulic Fundamentals*, Taylor and Francis, 1993.
- [9] P.B. Whalley, *Boiling, Condensation and Gas-Liquid Flow*, 1987.

Session 9a

Mechatronic Applications

Modelling of Hybrid Electric Vehicles in Modelica for Virtual Prototyping

Jonas Hellgren

Machine and Vehicle Systems

Chalmers University of Technology, SE-412 96, Gothenburg, SWEDEN

jonas.hellgren@me.chalmers.se, www.mvd.chalmers.se/~jonash/

Abstract: This paper presents a Modelica library made for the simulation of Hybrid Electrical Vehicles (HEVs). The library consists of vehicle models, component models and models of surrounding systems. An overview of the models within the library is given and some models are described more in detail. The major purpose of the vehicle models is to predict vehicle characteristics, especially fuel consumption, for a given vehicle and driving cycle. Modelica has been found useful for the simulation of HEVs.

1 Introduction

Environmental concerns and the decrease in cost of electrical components make *Hybrid Electrical Vehicles (HEVs)* more and more competitive. The major difference between conventional vehicles and HEVs is the presence of a buffer for temporary storage of energy in HEVs. The buffer enables regenerative braking and makes the *Primary Power Unit (PPU)* more or less decoupled from the wheels. This gives a potential for reducing fuel consumption and emissions.

Prototyping is necessary for the design and evaluation of HEVs. It is very expensive and time consuming to build real prototypes. Virtual prototyping (computer simulation) is therefore an almost necessary complement. The aim of this paper is to describe a library, developed in Modelica, whose purpose is to evaluate HEVs. In the future, the library can be extended with more components and configurations.

Modelica [1] is a language for modelling physical systems. It is a standard proposed by an international association. Modelica can handle problems in different areas, e.g. mechanics, electricity, chemistry, fluid dynamics and control theory.

The total system of an HEV is very complex. To achieve reasonable computation time, fairly simple models should be used. Another reason for using simple models is that fewer parameters are needed. It will be easier to update models in the future if fewer parameters are used. A rule of thumb when modelling is to use a model that is as simple as possible but still sufficiently accurate.

2 Modelling of different HEV configurations

HEVs can be configured in numerous ways. The arrangement, type and size of components result in a huge number of combinations. The most common way to categorize HEVs is by using the definitions of series and parallel HEVs. In series HEVs, there is no mechanical connection between the PPU and the wheels. In parallel HEVs, a part of the engine torque affects the wheels directly. This nomenclature is not always easy to adopt, however, because some drive-line configurations are neither clear series nor parallel ones. For example, the classification of split HEVs is debatable.

The library includes four different types of HEVs. The arrangements are illustrated on a large scale in Figure 1.

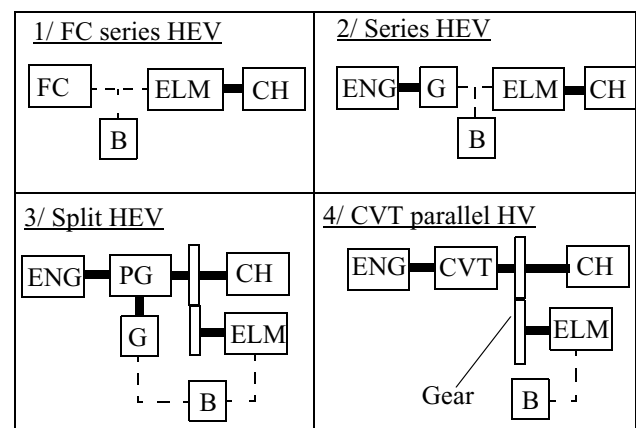


Fig. 1: HEV configurations. Acronyms: *ENG* engine, *FC* fuel cell, *B* buffer, *G* generator, *ELM* electric machine, *CH* chassis, *PG* planetary gear and *CVT* continuous variable transmission. Dashed lines represent electrical connections and continuous lines represent mechanical connections.

The reasons for choosing these HEV types are:

- They are subject to industrial projects. For example, the first HEV ever built for the public market, Toyota Prius, is a split HEV.
- The configurations allow the engine to work at its optimal line.

The reason why a series HEV can work at its optimal line is explained by the fact that the PPU is totally decoupled from the wheels. In other configurations, such as the parallel CVT and split hybrid, a transmission between the engine and wheels makes the engine speed controllable.

The relation between the vehicle and the surrounding systems is illustrated in Figure 2. The environment emits a driving cycle, i.e. speed and slope as functions of time. The driver controls the vehicle in such a way that the desired speed is managed. Figure 3 shows the implementation of a split HEV in Modelica.

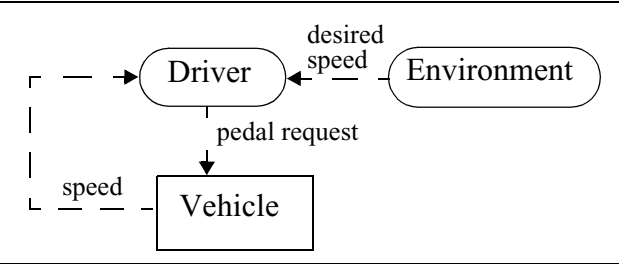


Fig. 2: The relation between the vehicle and the surrounding systems.

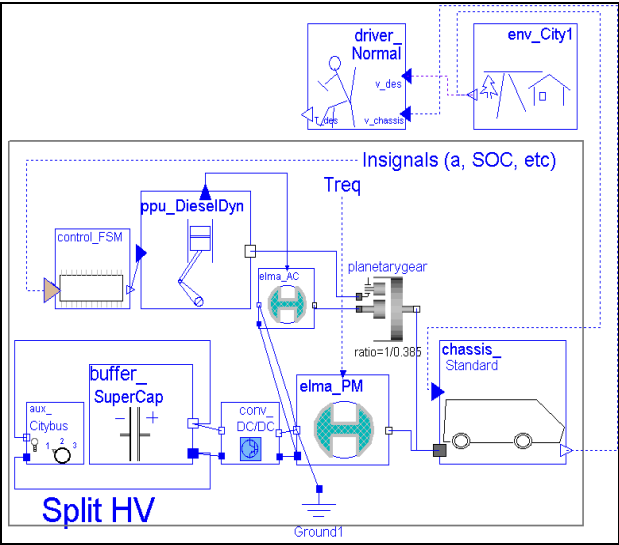


Fig. 3: Modelica model of a split HEV.

2.1 Example of simulation of an HEV

This section presents an example of a simulation of a series HEV. The vehicle has a fuel cell as the primary power unit and a super capacitor as the buffer. A city bus is chosen as an example. Information about the component sizing is given in Table 1 and the layout is illustrated in Figure 4. It is interesting to note that the required PPU size, expressed as maximum power, is about half that of a conventional bus. This is explained by the fact that the buffer assists in heavy accelerations. When the vehicle decreases in speed, power is regenerated into the buffer. The buffer might seem to be oversized with respect to power. The explanation is that the specific energy of the super capacitor is crucial with regard to sizing. The driver controls the target torque of the electric motor. If the driver demands braking, the desired motor torque will be negative. If the desired torque exceeds the torque limit of the motor at braking, a mechanical brake in the chassis will assist. The mechanical brake also assists when the vehicle stands still.

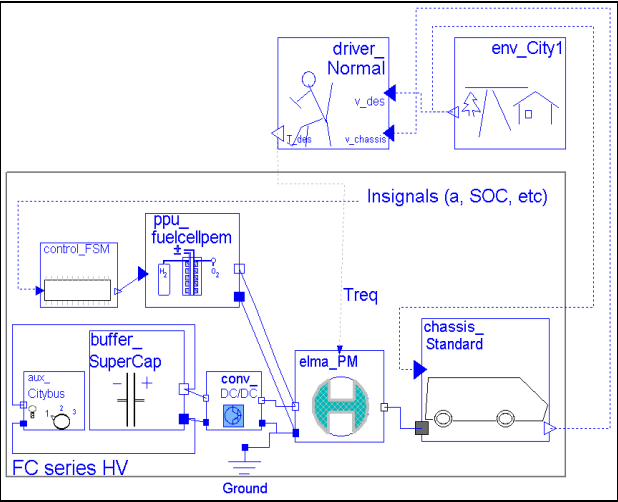


Fig. 4: Modelica model of a fuel cell HEV.

Table 1: Component sizes in the fuel cell HEV.

COMPONENT	SIZE; MASS	MAX POWER; ENERGY
Chassis (with propulsion units)	6; 17600 kg	
Fuel cell	2; 200 kg	100 kW; -
Motor	5; 200 kg	250 kW; -
Super capacitor	8; 500 kg	900 kW; 7.5 MJ

The driving cycle is a part of an urban city bus route and is plotted as speed as a function of time in Figure 5. Figure 6 also shows the cycle but with speed versus position instead of time. Many starts and stops are characteristic for the driving cycle. This makes an HEV more interesting. In this case, the vehicle manages to follow the desired speed almost exactly. Figure 7 shows how *State Of Charge (SOC)* in the buffer and motor temperature change during the driving cycle. It can be seen that the SOC increases at braking and decreases at acceleration. Figure 8 reflects the control of the PPU by comparing the power from the PPU to the power demand from the chassis. According to the simulation, the fuel consumption is 0.2 kg hydrogen/km (the energy density of hydrogen is three times higher than the energy density of diesel). This result is valid for this particular combination of vehicle and driving cycle.

The HEV model in Figure 4 uses 287 variables, 12 states and 80 parameters. This reflects the complexity of the model. Examples of states are: vehicle speed, SOC, temperature of the electric machine and power from the fuel cell.

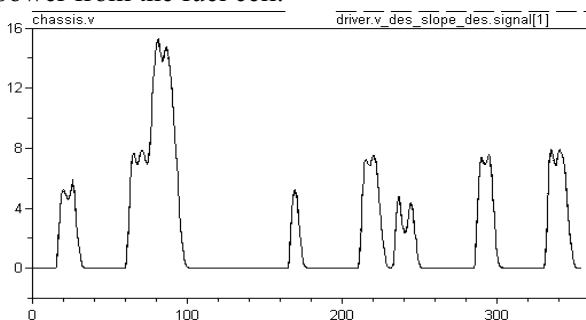


Fig. 5: Driving cycle. Speed [m/s] versus time [s].

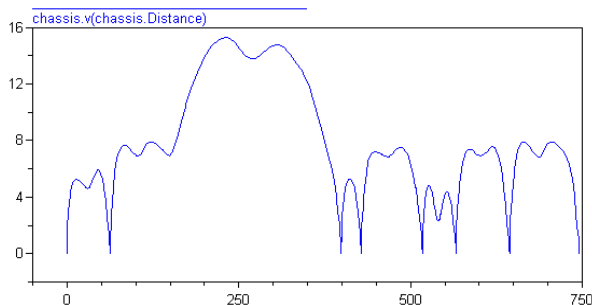


Fig. 6: Driving cycle equal to driving cycle in Figure 5 but with speed [m/s] versus position [m]

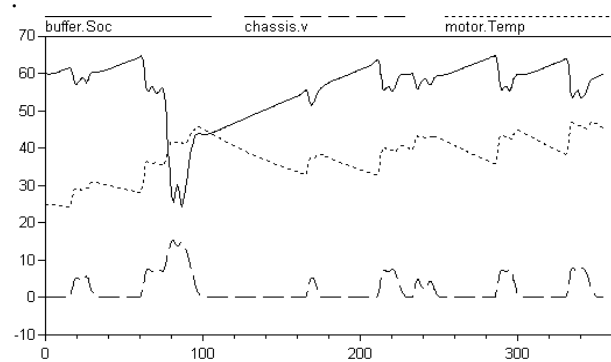


Fig. 7: SOC [%] (continuous line), motor temperature [Celsius] (dotted line) and vehicle speed [m/s] (dashed line) versus time [s].

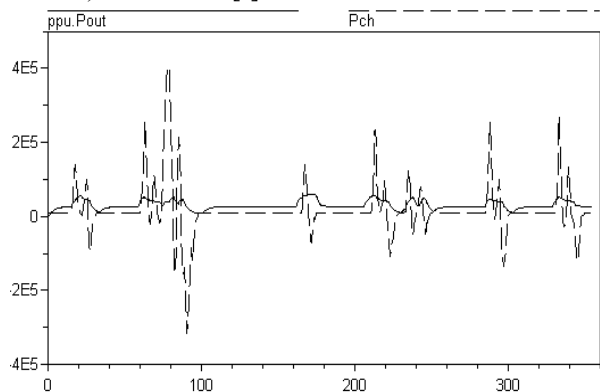


Fig. 8: Power from PPU [W] (continuous line) and power demand from chassis [W] (dashed line) versus time [s].

3 Modelling of components in HEVs

One major principle in modelling a component has been to make it as simple as possible to change its size. This is achieved by a *SIZE* parameter, which is included in most components. A change of the *SIZE* parameter affects some of the other parameters in a component model such that the component model behaves as though it has another size. When the *SIZE* parameter in a component model is changed, the total mass of the vehicle is changed automatically. Another principle has been to use physical parameters and to tune them in such a way that the model behaves similarly to an existing component. For example, the inner resistance in the battery model is defined such that the efficiency of the battery model is similar to a real battery.

Some component models are described more in detail in this paper (underlined in Figure 9). In the described component models, parameters (constant

during simulation) are in upper case letters and variables (changing during simulation) are in lower case letters.

Figure 9 shows the structure of the component model library. The library can be expanded to include more models in the future. The graphical interface makes it very easy to replace a component model in a vehicle model, e.g. replace a battery model with a super capacitor model. This is done by a “drag and drop procedure”.

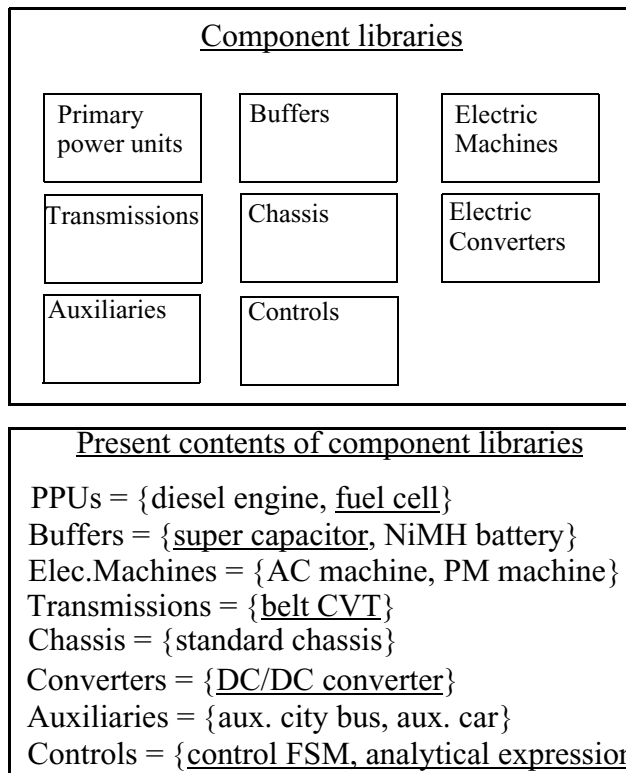


Fig. 9: Structure of component library. *AC* stands for alternating current, *PM* stands for permanent magnet and *DC* stands for direct current. When possible, a component uses sub-models that are included in the standard Modelica library.

Table 2: Number of internal states and number of parameters in the component models.

COMPONENT MODEL	NUMBER OF INTERNAL STATES	NUMBER OF PARAMETERS
AC machine	5	27
Auxiliaries	0	1

Table 2: Number of internal states and number of parameters in the component models.

COMPONENT MODEL	NUMBER OF INTERNAL STATES	NUMBER OF PARAMETERS
Chassis	2	20
CVT	3	10
DC/DC converter	0	2
Diesel engine	2	6+data map
Fuel cell	2	16
NiMH battery	3	26
PM machine	5	30
Super capacitor	3	12

3.1 Model of fuel cell

A fuel cell is an electrochemical device that combines hydrogen fuel and oxygen from the air to produce electricity, heat and water. Fuel cells operate without combustion, so they are virtually pollution free. In theory, a fuel cell can operate at much higher efficiencies than internal combustion engines, but auxiliary systems such as pumps and compressors reduce the efficiency. The efficiency of a fuel cell is approximately 50%. The fuel cell itself has no moving parts, and the fuel cell is thus a quiet and reliable source of power. Individual fuel cells are normally combined into a *stack*. The number of fuel cells in the stack, i.e. the number of cells in series, determines the total voltage. Major disadvantages of fuel cells are high capital cost and difficulties in handling the fuel. For example, hydrogen, gasoline and methanol are proposed as fuel.

Figure 10 illustrates the major idea of the model. Three efficiencies have been defined: the efficiency of fuel delivery η_{fuel} , the internal efficiency of a fuel cell η_{cell} , the efficiency related to parasitic losses and electrical conversion $\eta_{el\,sys}$. The model is strongly influenced by [2]. The *SIZE* parameter represents the number of stacks in the system. The transient performance, i.e. how fast the fuel cell can change power, is expressed by low pass filtering the desired power

P_{des} . The total power output from the fuel cell system is P_{out} .

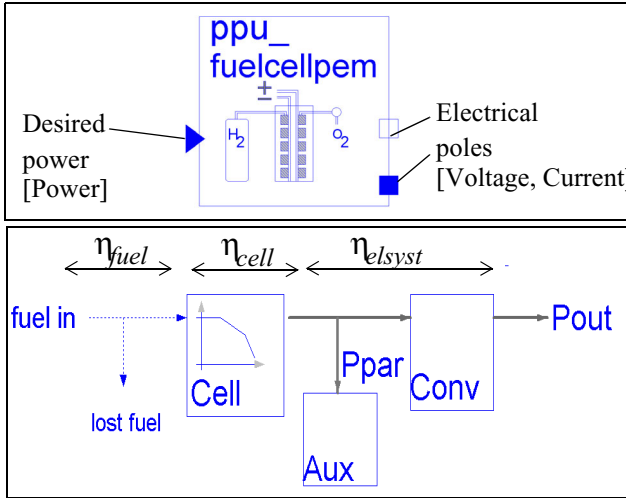


Fig. 10: Model of fuel cell. *Aux* represents auxiliary systems and *Conv* represents a DC/DC converter.

Eq. 1 to Eq. 4 describe how total power output P_{out} , desired power P_{des} , power produced in one cell P_{cell} and normalized power p are calculated and related to each other.

$$P_{out} = \text{filter}(P_{des}) \quad (\text{EQ } 1)$$

$$P_{out} = P_{cell} \cdot N_{SER} \cdot N_{STACKS} \cdot \eta_{elsyst} \quad (\text{EQ } 2)$$

$$P_{cell} = U_{cell} \cdot I_{cell} \quad (\text{EQ } 3)$$

$$p = \frac{P_{cell}}{P_{NOMCELL}} \quad (\text{EQ } 4)$$

The voltage in a cell U_{cell} decreases with p according to Eq. 5 and Eq. 6.

$$U_{cell} = \eta_v \cdot E_{CELL} \quad (\text{EQ } 5)$$

$$\eta_v = \left(1 + \sqrt{\left(1 - \frac{p}{PPC} \right)} \right)^2 \quad (\text{EQ } 6)$$

The calculation of the instantaneous fuel consumption F_c [g/s] and total efficiency of the system η_{tot} are described by Eq. 7 to Eq. 14.

$$F_c = \frac{P_{out} \cdot H_{FUEL}}{\eta_{tot}} \quad (\text{EQ } 7)$$

$$\eta_{tot} = \eta_{cell} \cdot \eta_{fuel} \cdot \eta_{elsyst} \quad (\text{EQ } 8)$$

$$\eta_{cell} = \eta_v \cdot \eta_{MAX} \quad (\text{EQ } 9)$$

$$\eta_{fuel} = U_{FUEL} \cdot \eta_{REF} \quad (\text{EQ } 10)$$

$$\eta_{elsyst} = \eta_{EL} \cdot \eta_{par} \quad (\text{EQ } 11)$$

$$\eta_{par} = \frac{P_{out}}{P_{out} + P_{par}} \quad (\text{EQ } 12)$$

$$P_{par} = \alpha_1 \cdot P_{TOT} + \beta_0 \cdot P_{TOT} \cdot p \quad (\text{EQ } 13)$$

$$P_{TOT} = P_{NOMCELL} \cdot N_{SER} \cdot N_{STACKS} \quad (\text{EQ } 14)$$

Table 3: Notations for the fuel cell model.

NOTATION	DESCRIPTION
α_1, β_0	Parameters that determine parasitic losses
E_{CELL} [V]	Maximal voltage in a cell
η_{EL} [-]	Efficiency of converter
η_{MAX} [-]	Maximal efficiency in one cell
η_{REF} [-]	Efficiency of reformer
H_{FUEL} [g/J]	Mass of hydrogen to achieve one Joule of energy
U_{FUEL} [-]	Amount of hydrogen utilized
N_{SER} [-]	Number of cells in series
N_{STACKS} [-]	Number of stacks
$P_{NOMCELL}$ [W]	Nominal power from a cell
P_{par} [W]	Parasitic losses
PPC [-]	Part of maximal theoretical power that is used in cell
P_{TOT} [W]	Nominal power from fuel cell system
U_{cell} [V], I_{cell} [A]	Voltage and current in a cell

The model is general and fairly simple and can therefore easily be adjusted to imitate almost any fuel cell system by changing the parameters. The system presented here meets the targets set by DOE (the Department Of Energy in USA) [3]. The targets are an efficiency of 44% at full load, an efficiency of 55%

at part load and a response time of three seconds. Part load is defined as 25% of full load and response is defined as how quickly power is changed from zero to maximal load. Figure 11 compares the model with the DOE targets.

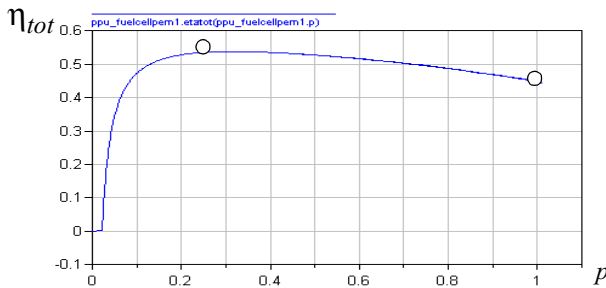


Fig. 11: Efficiency (η_{tot}) versus normalized power p . The continuous line represents the model and the circles represent DOE targets.

3.2 Model of DC/DC converter

The task of a DC/DC converter is to adjust the output voltage of an electrical device. In an HEV the operating voltage of the buffer may be different from the voltage range desired in the electrical machine. This is an example of a situation in which a DC/DC converter becomes useful. DC/DC means that a direct current is transformed to a direct current with another voltage and current. Unfortunately, this power transformation is related to losses. The power losses in a DC/DC converter are accurately described in [4]. The model takes into account the dominant losses according to [4].

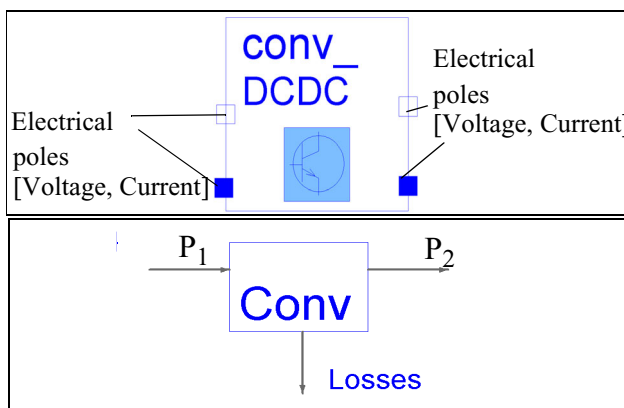


Fig. 12: Model of DC/DC converter.

Eq. 15 to Eq. 18 describe the model. Eq. 16 and Eq. 17 imply that a higher quotient between voltage on outside and inside results in a lower efficiency η .

$$P_{losses} \approx (1 - \eta) \cdot P_1 \quad (\text{EQ 15})$$

$$d = \left| 1 - \frac{v_2}{v_1} \right| \quad (\text{EQ 16})$$

$$\eta = \eta_{MAX} - d \cdot KD \quad (\text{EQ 17})$$

$$P_2 = \begin{cases} P_1 \cdot \eta & (P_1 > 0) \\ \frac{P_1}{\eta} & (P_1 < 0) \end{cases} \quad (\text{EQ 18})$$

Table 4: Notations for the converter model.

NOTATION	DESCRIPTION
d	Ratio of switching
P_{losses} [W]	Total losses in the converter
η_{MAX} , KD	Parameters determining the efficiency of the converter
P_1 , P_2 [W]	Power in and power out from the converter
v_1 , v_2	Voltage on outside and inside

The parameters in the model are adjusted in such a way that the efficiency is close to 95%.

3.3 Model of super capacitor

Super capacitors are an energy storage technology ideally suited for applications that need repeated bursts of power for fractions of a second to several minutes. High specific power but low specific energy is characteristic.

The model illustrated in Figure 13 is influenced by [5]. The parameters are taken from the data sheet for an existing super capacitor “PC2500” described more in detail in [6]. The SIZE parameter represents the number of super capacitors in parallel. Due to the definition of a leaking current I_{leak} , the model takes into account that the capacitor is discharged even when no current is requested. R_I is the internal resistance, C is the capacitance, I_{capPos} is the requested current and U_{cap} is the voltage of the capacitor.

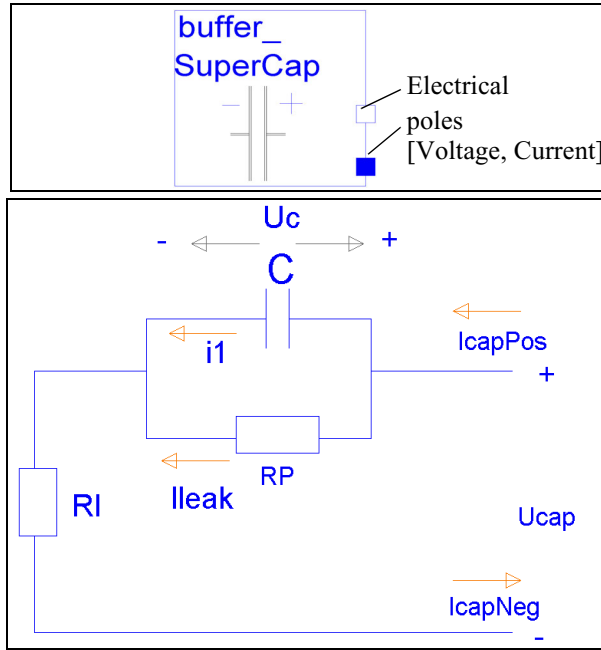


Fig. 13: Model of super capacitor.

Eq. 19 to Eq. 24 describe current and voltage laws for one capacitor. Most of the terms are described in Table 5.

$$I_{capPos} = i_1 + I_{leak} \quad (\text{EQ 19})$$

$$I_{capPos} + I_{capNeg} = 0 \quad (\text{EQ 20})$$

$$U_c = \frac{Q}{C} \quad (\text{EQ 21})$$

$$\dot{Q} = i_1 \quad (\text{EQ 22})$$

$$I_{leak} = \frac{U_c}{R_P} \quad (\text{EQ 23})$$

$$U_{cap} = U_c + R_I \cdot I_{capPos} \quad (\text{EQ 24})$$

The requested power P is positive at discharge and is negative at charge. The following equations define the energy level in the super capacitor Soc and the efficiency η :

$$Soc = \frac{U_c - U_{MIN}}{U_{MAX} - U_{MIN}} \cdot 100 \quad (\text{EQ 25})$$

$$\eta = \frac{|P|}{|P| + P_{Losses}} \quad (\text{EQ 26})$$

$$P = U_{cap} \cdot I_{capPos} \quad (\text{EQ 27})$$

$$P_{Losses} = I_{Leak}^2 \cdot R_P + I_{capPos}^2 \cdot R_I \quad (\text{EQ 28})$$

Table 5: Notations for the super capacitor model.

NOTATION	DESCRIPTION
U_c [V], i_1 [A]	Internal voltage and internal current in capacitor
Q [C]	Charge in capacitor
R_P [Ohm]	Resistance determining the leaking current I_{leak}
U_{MIN} , U_{MAX} [V]	Permitted minimal and maximal voltage in capacitor

Figure 14 shows the simulation of a charge process of a super capacitor model. Power is taken from the capacitor and increases with time. The upper curve corresponds to Soc starting at 70%. The lower curve corresponds to a start value of 30%. The curves show that efficiency is dependant on Soc and decreases with power. The simulated efficiency corresponds to data given in [6].

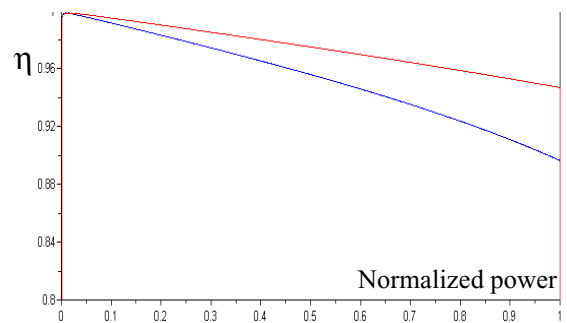


Fig. 14: Efficiency for super capacitor versus normalized power for different start values of Soc . Normalized power is defined as requested power divided by maximum possible power.

3.4 Model of belt CVT

A *Continuous Variable Transmission (CVT)* allows the speed ratio to change in a step less way. This allows the engine to operate on its optimal line. One disadvantage is that a CVT has a lower efficiency than a conventional transmission.

A belt CVT, illustrated in Figure 16, is used here. The clutch slips when the torque is too high, which results in poor efficiency. The efficiency of the belt η_{belt} is a function of utilized torque, see Figure 15. The SIZE parameter determines the mass and maximum torque capacity T_{MAX} of the CVT.

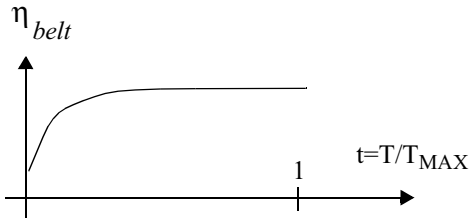


Fig. 15: Efficiency of CVT belt as function of utilized torque t .

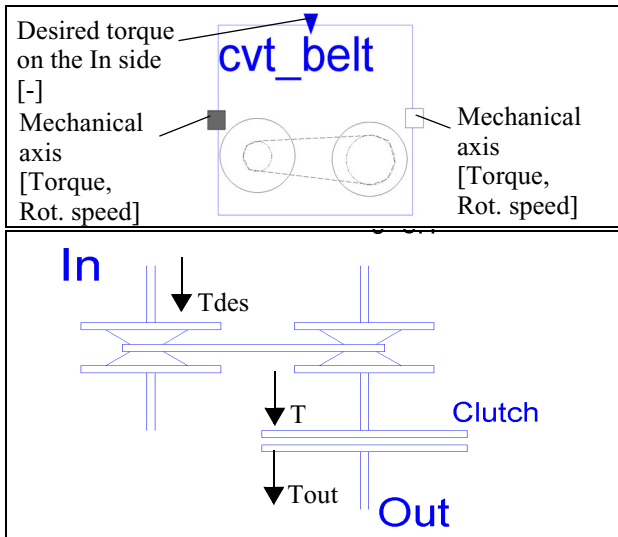


Fig. 16: Model of belt CVT.

Eq. 29 to Eq. 33 describe how the output torque T_{out} , the ratio of the CVT r_{cvt} and the total ratio r_{tot} are calculated and related to each other. The filter function in Eq. 30 reflects the fact that there is a response time in the system.

$$T_{out} = \min(T_{MAX}, T) \quad (\text{EQ 29})$$

$$T = \text{filter}(T_{des}) \cdot r_{cvt} \cdot \eta_{belt} \quad (\text{EQ 30})$$

$$r_{cvt} = \min(r_{MAX}, \max(r_{MIN}, r_{tot})) \quad (\text{EQ 31})$$

$$r_{tot} = \frac{w_{In}}{w_{Out}} \quad (\text{EQ 32})$$

$$\eta_{belt} = B + \frac{A - B}{1 + e^{t \cdot C}} \quad (\text{EQ 33})$$

Table 6: Notations for the belt CVT model.

NOTATION	DESCRIPTION
A, B, C	Constants determining the efficiency of the belt
r_{MAX}, r_{MIN} [-]	Maximum and minimum ratio
T_{des} [Nm]	The desired torque at the In side
T_{MAX} [Nm]	The maximum torque that the CVT can transmit
w_{IN}, w_{OUT} [Rad/s]	Speed at IN and OUT side

The parameters in the model are adjusted in such a way that the maximum efficiency is close to 90%.

3.5 Model of chassis

The chassis model takes into account resistance in the longitudinal direction. This resistance is mainly a result of vehicle mass, air and tire rolling resistance. From the SIZE parameter, it is possible to choose chassis models that correspond to the following vehicles: Volkswagen Golf (small passenger car), Volvo V70 (large passenger car), Toyota Previa (minibus), Chassis corresponding to a mini truck, Volvo FL6 (large truck), Volvo B10M (city bus), Volvo FL20 (heavy truck). The brake is necessary to make the vehicle stand still if a slope is present or to assist if the capacity of the electric braking in an HEV is exceeded. The traction force $F_{traction}$ is limited to prevent the wheels from skidding. Figure 17 shows the layout of the chassis. Only components from the standard Modelica library are used. The gear efficiency depends on vehicle configuration. A low gear efficiency corresponds to the use of a differential while a high gear efficiency corresponds to the use of wheel motors.

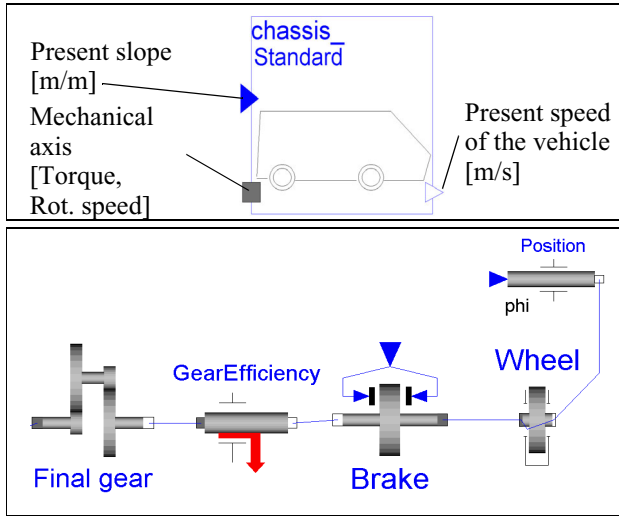


Fig. 17: Model of chassis.

Eq. 34 to Eq. 37 describe the dynamics of the vehicle in the longitudinal direction. $F_{traction}$ calculated in Eq. 37 affects the wheel component in Figure 17.

$$MASS = MASS_{DRIVELINE} + MASS_{COACH} \quad (\text{EQ 34})$$

$$MASS_{FRAME} + MASS_{PASSENGER}$$

$$F_{traction} = \begin{cases} F_{trnom} & (|F_{trnom}| < F_{TRMAX}) \\ F_{TRMAX} \cdot \varsigma & (|F_{trnom}| > F_{TRMAX}) \end{cases} \quad (\text{EQ 35})$$

where

$$\varsigma = \text{sign}(T_{Wheel})$$

$$F_{trnom} = \frac{T_{Wheel}}{R_{WHEEL}} \quad (\text{EQ 36})$$

$$F_{traction} = F_{acc} + F_{air} + F_{roll} + F_{slope} \quad (\text{EQ 37})$$

where

$$F_{acc} = MASS \cdot a \cdot \gamma$$

$$F_{air} = \frac{C_D \cdot A \cdot \rho_{AIR} \cdot v^2}{2}$$

$$F_{roll} = K_F \cdot N \cdot \left(1 - \frac{1}{0.5 \cdot v}\right)$$

$$F_{slope} = MASS \cdot G \cdot \alpha$$

$$N = MASS \cdot G \cdot \cos(\text{asin}(\alpha))$$

Table 7: Notations for the chassis model.

NOTATION	DESCRIPTION
α [m/m]	Road surface grade
a [m/s ²]	Acceleration of vehicle
$C_D A$ [m ²]	Air resistance of vehicle
$F_{traction}$ [N]	Force between ground and wheel
F_{trnom} [N]	Traction force if vehicle is not skidding
γ [-]	Factor for influence of rotational inertias of driven axle
G [m/s ²]	Constant of gravity
K_F [-]	Coefficient of rolling resistance
$MASS$ [kg]	Total mass of vehicle
N [N]	Normal force between vehicle and ground
v [m/s]	Speed of vehicle

3.6 Model of controller

The controller controls the power from the PPU. How this should be done is debatable and the aim of research. In the simulations presented in this paper, a *Finite State Machine (FSM)* is used as the control algorithm. A more detailed description of how this works is found in [7]. In practice, the control model interprets data maps that contain information on the FSM.

Another candidate to control the power produced in the PPU is to use an analytical expression. Eq. 38 proposes how this power can be calculated. The expression is influenced from [8].

$$\frac{d}{dt} (P_{PPU}) = \frac{(P_{ch} + K(Soc_{tar} - Soc) - P_{PPU})}{\tau} \quad (\text{EQ 38})$$

Table 8: Notations for Eq. 38.

NOTATION	DESCRIPTION
P_{PPU}	Power requested from PPU
Soc_{tar}	Desired Soc
K, τ	Parameters

4 Modelling of surrounding systems in HEVs

The surrounding systems are more abstract models and are necessary to be able to make the simulation. The driving cycle in particular is extremely important for the result of a simulation.

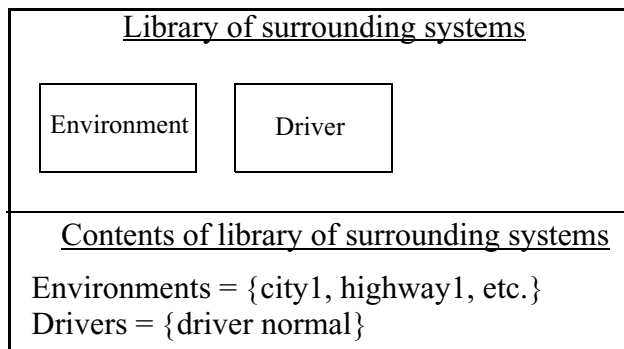


Fig. 18: Library of surrounding systems.

4.1 Model of environment

The environment model interprets data maps that contain the driving cycles. Speed and slope are described as functions of time. Alternatively, speed and slope can be described as functions of position.

4.2 Model of driver

The driver model is simply a PID regulator that forces the vehicle speed to be equal to the desired speed, i.e. the speed that is given in the driving cycle. Desired speed and vehicle speed are inputs and requested traction torque is output.

5 Conclusions and future work

Modelica has been found suitable for modelling and simulating an HEV. Properties such as object orientation, non casual modelling and an equation based syntax have been found useful during the development of the models presented in this paper. Much modelling work would have been saved if more public model libraries for Modelica had been available. Unfortunately there is a lack of such libraries today.

In the future, additional vehicle configuration and component models should be developed. More detailed models that take more phenomena into account should also be developed. Validation of the virtual HEV models towards existing vehicles should also be done.

REFERENCES

- [1] Modelica, www.modelica.org, September 2001.
- [2] Thorstensen B., "A parametric study of fuel cell system efficiency under full and part load operation", Journal of Power Sources, Volume 92, Issues 1-2, Pages 9-16, January 2001.
- [3] www.energy.gov, September 2001.
- [4] Alving B., "Calculating Power Losses in DC-DC converters, including a proposed high efficiency converter topology", Master thesis work, Electrical and Computer Engineering, Chalmers University of Technology, Gothenburg, Sweden, 1999.
- [5] Axelsson F., "Investigation of super capacitors in a series hybrid vehicle", Master Thesis at Control Eng. Lab., Gothenburg, Chalmers University of Technology, Sweden, 1998.
- [6] Powercache, www.powercache.com, September 2001.
- [7] Hellgren J. and Wahde M., "Evolving Finite State Machines for the Propulsion Control of Hybrid Vehicles", Advances in Signal Processing and Computer Technologies, ISBN 960-8052-37-8, WSES, 2001.
- [8] Andersson C., Jonasson K., Strandh P. and Alakula M., "Simulation and verification of a hybrid bus", Nordic Workshop on Power and Industrial Electronics, Aalborg, Denmark, 2000.

NOTATION

NOTATION	DESCRIPTION
CVT	Continuous Variable Transmission
Data map	A set of numbers presented in a vector or matrix
Normalized power	Requested power divided with maximum power that can be delivered to/from the device
Soc [%]	State of charge in an energy buffer

Modeling and Simulating the Efficiency of Gearboxes and of Planetary Gearboxes

Christoph Pelchen, ZF Friedrichshafen, Germany (Christoph.Pelchen@zf.com)

Christian Schweiger, DLR, Germany (Christian.Schweiger@dlr.de)

Martin Otter, DLR, Germany (Martin.Otter@dlr.de)

Abstract

It is shown how to model and simulate frictional effects present in gearboxes and in planetary gearboxes. This includes modeling of gear wheel sticking and sliding due to Coulomb friction between the gear teeth leading to load torque dependent losses. This allows reliable simulation of, e. g., stick-slip effects in servo drives or gear shifts in automatic gearboxes. It is also discussed how the friction characteristics can be measured in a useful way. The presented models are implemented in *Modelica* and demonstrated at hand of the simulation of an automatic gearbox.

1 Introduction

Gearbox dynamics due to friction, elasticity and backlash in the gear has often a strong impact on the performance of the system in which the gearbox is contained, such as for robots, machine tools, vehicles, power trains. It is both difficult to simulate gearbox effects and to get reasonable agreement between measurements and dynamic simulations.

In the current *Modelica* standard library `Modelica.Mechanics.Rotational` [5] several model components are available to simulate gearbox effects, especially bearing friction. *Missing* is the satisfactory handling of *mesh efficiency* due to friction between the teeth of gear wheels which leads to load torque dependent losses. In this article it is shown in detail how this problem can be solved for *any* gearbox that has *two* or *three external shafts*, i. e., standard gears and a large class of planetary gears.

Before going into the details of an appropriate gear efficiency model, it is important to analyse the different frictional effects in a gearbox. Friction is present between two surfaces which slide on each other. In a gearbox, this occurs in the gear bearings and between the gear teeth which are in contact to each other. The effect of these two cases is quite different.

1.1 Bearing Friction

The torques acting at a bearing are shown in figure 1. The shaft in the bearing has the torques τ_A and τ_B on the two sides. Losses due to friction are described by the additional torque τ_{bf} . Torque equilibrium yields

$$\tau_B = \tau_A - \tau_{bf} . \quad (1)$$

where

$$\tau_{bf} = \begin{cases} \geq 0: & \omega > 0 \\ \leq 0: & \omega < 0 \\ \text{so that } \dot{\omega} = 0: & \omega = 0 \end{cases} \quad (2)$$

The friction torque τ_{bf} is essentially a function of the shaft speed ω , the bearing load f_N (=force perpendicular to bearing axis), the bearing temperature T , the bearing construction and the used lubrication (for more details, see, e. g. [6]). Since the bearing load is usually constant (but not zero) and independent of the gearbox load torque, and all other factors can be often regarded as constant for certain operation conditions, the bearing friction is essentially a function of the relative speed, $\tau_{bf}(\omega)$, and has a characteristic as given in figure 2.

If $\omega \neq 0$ the friction torque τ_{bf} is computed from the sliding friction characteristic according to figure 2. If $\omega = 0$ the bearing is stuck due to the bearing load in combination with Coulomb friction, and therefore the friction torque τ_{bf} is an unknown constraint torque which is computed so that $\dot{\omega}$ vanishes. How to model and simulate this effect is described in detail, e. g., in [8].

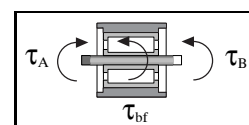


Figure 1: Torques at a bearing

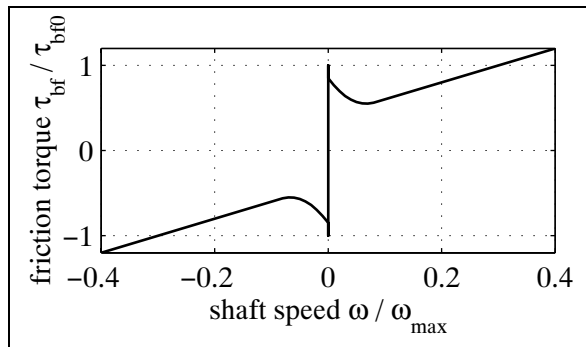


Figure 2: Typical bearing friction characteristic

1.2 Mesh Friction

In figure 3, two gear teeth in contact are shown. In order that the teeth neither penetrate nor separate, the normal velocities in contact point C need to be identical and therefore the tangential velocities are different, i. e., the teeth slide on each other, see, e. g., [3] (the tangential velocities are only identical, if $\omega_A = 0$ or if point C is at W, see figure 3). As a result, in contact point C Coulomb friction $f_R = s_v \mu f_N$ is present, where f_R is the friction force in the contact plane, f_N is the force perpendicular to the contact plane, $s_v = \pm 1$ depending on whether C is below or above pitch circle r_A and $\mu = \mu(v_{rel}, T)$ is the sliding friction coefficient which is essentially a function of the relative velocity v_{rel} between the contact planes and the temperature T at the contact point. Note, that the contact planes

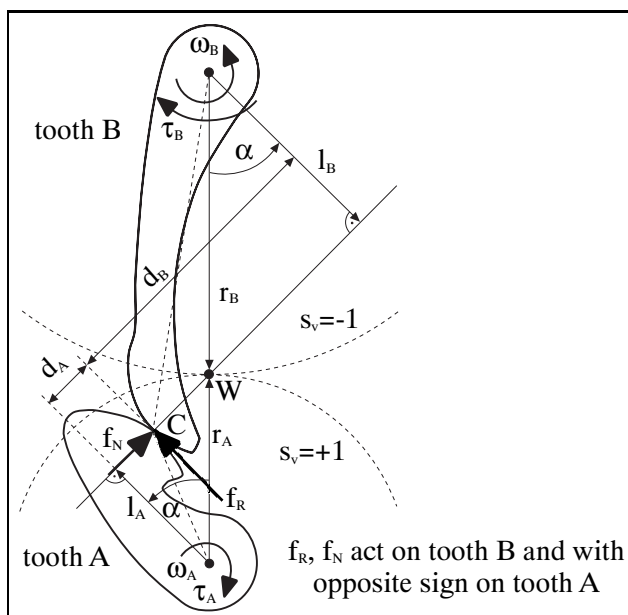


Figure 3: Friction between gear wheel teeth

may become stuck to each other if $v_{rel} = 0$, i. e., if $\dot{\omega}_A = 0$. Then, f_R is a constraint force calculated from the condition that $\dot{\omega}_A = 0$. Torque equilibrium in figure 3 yields

$$\begin{aligned} 0 &= \tau_A - f_N l_A + f_R d_A \\ 0 &= \tau_B - f_N l_B + f_R d_B \\ f_R &= s_v \mu f_N \end{aligned} \Rightarrow \tau_B = \frac{l_B \left(1 - s_v \mu \frac{d_B}{l_B}\right)}{l_A \left(1 - s_v \mu \frac{d_A}{l_A}\right)} \tau_A.$$

Utilizing teeth contact geometry and gear ratio i

$$\cos \alpha = \frac{l_A}{r_A} = \frac{l_B}{r_B} \Rightarrow \frac{l_B}{l_A} = \frac{r_B}{r_A} = i$$

results in

$$\tau_B = i\eta_{\text{mf1}} \tau_A \quad \text{with} \quad \eta_{\text{mf1}} := \frac{1 - s_V \mu \frac{d_B}{l_B}}{1 - s_V \mu \frac{d_A}{l_A}}. \quad (3)$$

Since $d_A/l_A < d_B/l_B$ for $s_v = +1$, $d_A/l_A > d_B/l_B$ for $s_v = -1$, and $0 \leq \mu < 1$, it follows from (3) that η_{mf1} is between 0 and 1.

In a similar way it can be shown (see, e. g., [7]) that the relative velocity v_{rel} is calculated as $v_{rel} = k(\cdot)\omega_A$ where $k(\cdot)$ is a function of the geometric quantities d_A, d_B, l_A, l_B . Since all these quantities can be computed from gear-wheel constants and the absolute angle φ_A of shaft A, $\mu = \mu(\varphi_A, \omega_A, T)$ and therefore $\eta_{mfl} = \eta_{mfl}(\varphi_A, \omega_A, T)$.

The derivation above is only valid if $\omega_A > 0$, because the friction force at the driven tooth is always directed in opposite direction to the relative sliding velocity. If $\omega_A < 0$ the sign of the friction force changes, yielding:

$$\eta_{\text{mf}2} \tau_B = i \tau_A \quad \text{with} \quad \eta_{\text{mf}2} := \frac{1 + s_V \mu \frac{d_A}{l_A}}{1 + s_V \mu \frac{d_B}{l_B}}. \quad (4)$$

The derivations assume that the "right" side of tooth edge A is in contact. If the "left" side is in contact, the sign of the normal force changes. Collecting everything together and neglecting the temperature and position dependency of $\eta_{mf1}(\cdot)$, $\eta_{mf2}(\cdot)$ finally results in the basic formula for mesh friction:

$$\hat{\eta}_{\text{mf}} := \begin{cases} \eta_{\text{mf1}}(|\omega_A|) & : \begin{cases} \tau_A \omega_A > 0 \text{ or} \\ \tau_A = 0 \text{ and } \omega_A > 0 \end{cases} \\ 1/\eta_{\text{mf2}}(|\omega_A|) & : \begin{cases} \tau_A \omega_A < 0 \text{ or} \\ \tau_A = 0 \text{ and } \omega_A < 0 \end{cases} \\ \text{so that } \dot{\omega}_A = 0: & \omega_A = 0 \end{cases} \quad (5)$$

where $\eta_{mf1}(|\omega_A|) \in [0; 1]$ and $\eta_{mf2}(|\omega_A|) \in [0; 1]$ denote the *mesh efficiencies* for the different power flow directions characterized by $P_A = \tau_A \omega_A$. Note, that the two mesh efficiencies are a function of the absolute value of ω_A . Often, $\eta_{mf1} \approx \eta_{mf2}$. However, there are also cases where the two mesh efficiencies are very different, e. g., for worm gears.

2 Standard Gear

In this section, a mathematical description of the frictional effects present in a standard gear is presented and an appropriate *Modelica* model is sketched. The gear type under consideration is shown in figure 4. Here, ω_A denotes the angular velocity of the left shaft

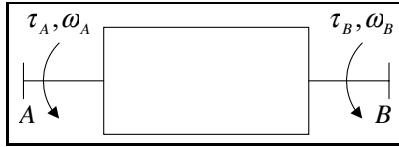


Figure 4: Speeds and cut-torques of a standard gear

and ω_B denotes the angular velocity of the right shaft, respectively. At the cut-planes of the two shafts, the constraint torques τ_A and τ_B are present. The class of gears to be examined in this section is formally defined as:

Definition 1: A gear denoted as *standard gear* in this article has the following properties:

- The gear has *two external shafts*.
- The gear has *one degree of freedom*.
- The time invariant constraint equation

$$\omega_A = i \omega_B \quad (6)$$

holds, where i is *constant* and not zero.

This constant is called *gear ratio*¹

This definition includes a broad class of gears.

2.1 Mathematical Description

A standard gear may have several bearings, several gear stages and several teeth in contact. Based on the observations in section 1.1 and 1.2, and assuming that all bearing losses are either transformed to bearing friction $\tau_{bf,A}$ at shaft A or bearing friction $\tau_{bf,B}$ at shaft B and that mesh friction $\hat{\eta}_{mf}$ is present, the following loss model is obtained

$$\tau_B - \tau_{bf,B} = -i \hat{\eta}_{mf} (\tau_A - \tau_{bf,A}) . \quad (7)$$

Reordering of terms yields

$$\tau_B = i (-\hat{\eta}_{mf} \tau_A + \frac{1}{i} \tau_{bf,B} + \hat{\eta}_{mf} \tau_{bf,A}) . \quad (8)$$

¹The following derivation is also valid for variable gear ratios, such as a CVT gear. The only addition is that the bearing friction term $\hat{\tau}_{bf}$ is not only a function of ω_A but also of the actual gear ratio i , due to (8) and (9).

The torque direction of $\tau_{bf,B}$ depends on the sign of ω_B due to equation (2) and the torque direction of $\tau_{bf,A}$ depends on the sign of ω_A . Since $\omega_B = i \omega_A$, the torque direction of $\tau_{bf,B}/|i|$ depends also on the sign of ω_A . Therefore, all bearing friction terms can be collected to one overall bearing friction variable $\hat{\tau}_{bf}$ which has the properties that (a) the direction of this torque depends on ω_A and (b) the value depends on the same energy flow directions as $\hat{\eta}_{mf}$ does. As a result, the following gear loss model is obtained:

$$\tau_B = i (-\hat{\eta}_{mf} \tau_A + \hat{\tau}_{bf}) \quad (9)$$

where

$$\hat{\eta}_{mf} := \begin{cases} \eta_{mf1}(|\omega_A|) & : \begin{cases} \tau_A \omega_A > 0 \text{ or} \\ \tau_A = 0 \text{ and } \omega_A > 0 \end{cases} \\ 1/\eta_{mf2}(|\omega_A|) & : \begin{cases} \tau_A \omega_A < 0 \text{ or} \\ \tau_A = 0 \text{ and } \omega_A < 0 \end{cases} \\ \text{so that } \dot{\omega}_A = 0 & : \omega_A = 0 \end{cases} \quad (10)$$

describes the *mesh frictions* with $\eta_{mf1}(|\omega_A|)$, $\eta_{mf2}(|\omega_A|) \in [0; 1]$ and

$$\hat{\tau}_{bf} := \begin{cases} \tau_{bf1}(\omega_A) & : \begin{cases} \tau_A \omega_A > 0 \text{ or} \\ \tau_A = 0 \text{ and } \omega_A > 0 \end{cases} \\ \tau_{bf2}(\omega_A) & : \begin{cases} \tau_A \omega_A < 0 \text{ or} \\ \tau_A = 0 \text{ and } \omega_A < 0 \end{cases} \\ \text{so that } \dot{\omega}_A = 0 & : \omega_A = 0 \end{cases} \quad (11)$$

describes the *bearing frictions* with

$$\hat{\tau}_{bf}(\omega_A) = \begin{cases} \geq 0 & : \omega_A > 0 \\ \leq 0 & : \omega_A < 0 \end{cases} . \quad (12)$$

More detailed models are obtained by taking into account that $\eta_{mf1}(\cdot)$, $\eta_{mf2}(\cdot)$ are additionally functions of the absolute position φ_A of shaft A and of the gear temperature T , and $\hat{\tau}_{bf}(\cdot)$ is additionally also a function of T , respectively.

The model above describes especially the case when the gear bearings and the teeth in contact to each other are stuck. This occurs when $\omega = 0$. Then, $\hat{\eta}_{mf}$ and $\hat{\tau}_{bf}$ are constraint variables which are computed from the condition that the gear remains stuck, or formulated mathematically that $\dot{\omega}_A = 0$. If the constraint variables become greater as their respective sliding values at zero speed, the gear leaves the stuck mode and starts sliding. Note, that the stuck mode is both due to bearing friction (because the bearing loads introduce Coulomb friction) and due to mesh friction. Since in stuck mode there are two additional unknowns ($\hat{\eta}_{mf}$, $\hat{\tau}_{bf}$), but only one additional equation

($\dot{\omega}_A = 0$), there is an ambiguity so that either $\hat{\eta}_{mf}$ or $\hat{\tau}_{bf}$ can have an arbitrary value in this mode.

A direct implementation of model (9) is difficult. The key idea from [9] is to transform this model into a form close to the standard bearing friction model which is well understood. This requires to collect all loss effects in an *additive* loss torque $\Delta\tau$, i. e., to describe the mesh and bearing frictions by the equation

$$\tau_B = i(-\tau_A + \Delta\tau) \quad (13)$$

instead of (9). Equation (13) implicitly defines the newly introduced loss torque $\Delta\tau$, i. e., (9) and (13) are two equations for the three unknowns τ_A , τ_B and $\Delta\tau$.

In *sliding* mode, equation (9) is replaced by the combined equation of (9), (13)

$$-\hat{\eta}_{mf} \tau_A + \hat{\tau}_{bf} = -\tau_A + \Delta\tau$$

and therefore

$$\Delta\tau = (1 - \hat{\eta}_{mf}) \tau_A + \hat{\tau}_{bf}. \quad (14)$$

In *stuck* mode, equation (9) is replaced by the constraint equation $\dot{\omega}_A = 0$.

To summarize, the transformed gear loss model is defined by (10), (11) and equations:

$$\tau_B = i(-\tau_A + \Delta\tau) \quad (15)$$

$$\Delta\tau = \begin{cases} (1 - \hat{\eta}_{mf}) \tau_A + \hat{\tau}_{bf} & : \omega_A \neq 0 \\ \text{so that } \dot{\omega}_A = 0 & : \omega_A = 0 \end{cases} \quad (16)$$

Note, that by this transformation the previous ambiguity in stuck mode is removed. Utilizing (10)-(12) in (16) for the sliding mode, results in the equations of table 1. The different regions to compute $\Delta\tau$ are visu-

ω_A	τ_A	$\Delta\tau =$
> 0	≥ 0	$(1 - \eta_{mf1}) \tau_A + \tau_{bf1} \quad (= \Delta\tau_{\max 1} \geq 0)$
> 0	< 0	$(1 - 1/\eta_{mf2}) \tau_A + \tau_{bf2} \quad (= \Delta\tau_{\max 2} \geq 0)$
< 0	≥ 0	$(1 - 1/\eta_{mf2}) \tau_A - \tau_{bf2} \quad (= \Delta\tau_{\min 1} \leq 0)$
< 0	< 0	$(1 - \eta_{mf1}) \tau_A - \tau_{bf1} \quad (= \Delta\tau_{\min 2} \leq 0)$

Table 1: $\Delta\tau = \Delta\tau(\omega_A, \tau_A)$ in sliding mode

alized in the upper part of figure 5. In sliding mode, $\Delta\tau$ has either a value on the upper or on the lower limiting lines, depending on the sign of ω_A . In stuck mode, $\Delta\tau$ has a value between the limiting lines such that $\dot{\omega}_A = 0$. Stuck mode is left, when $\Delta\tau$ reaches one of the limiting lines.

In the lower part of figure 5 the torque loss $\Delta\tau$ is shown using ω_A as abscissa and τ_A as curve parameter.

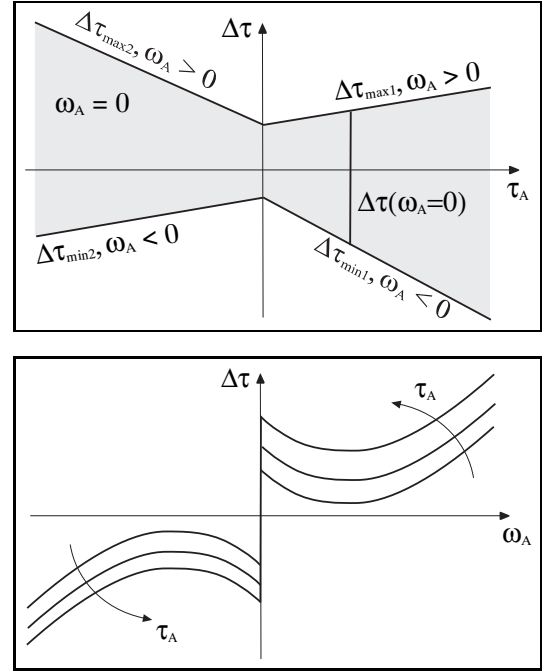


Figure 5: $\Delta\tau$ in sliding and stuck mode

By this figure it can be clearly seen, that the transformation to $\Delta\tau$ results in a friction characteristic which is close to a pure bearing friction model. The stuck mode is described in an identical way. Only the sliding friction torque is no longer a function of solely ω_A , but additionally a function of τ_A .

2.2 Modelica Model

The gear loss model derived in the previous section can be implemented as a *Modelica* model in a straightforward manner. The parameters to be provided are the gear ratio i and table `lossTable` to define the gear losses, see table 2. Tabulated values of the variables η_{mf1} , η_{mf2} , τ_{bf1} , τ_{bf2} have to be given as function of $\omega_A \geq 0$. The values for negative ω_A are automatically taken care off. Whenever η_{mf1} , η_{mf2} , $\hat{\tau}_{bf1}$ or $\hat{\tau}_{bf2}$ are needed, they are determined by interpolation in `lossTable`. The interface of this *Modelica* model is therefore defined as

```
parameter Real i = 1;
parameter Real lossTable[:,5]
           = [0, 1, 1, 0, 0];
```

$ \omega_A $	η_{mf1}	η_{mf2}	$ \tau_{bf1} $	$ \tau_{bf2} $
\vdots	\vdots	\vdots	\vdots	\vdots

Table 2: Format of table `lossTable`.

using the unit gear ratio and no losses as a default.

Internally, the gear loss model is based on the BearingFriction model, because this model already implements the sliding/stuck handling in a satisfactory way. The only enhancement is that the sliding friction torque $\Delta\tau$ is not only a function of ω_A but also of the unknown variable τ_A and of the relations $\tau_A \geq 0$, $\tau_A < 0$.

During code generation, this results in an additional algebraic loop in which τ_A , $\Delta\tau$ and the two relations are contained. Due to the structure of equation (14), the continuous unknowns (τ_A , $\Delta\tau$) enter this loop linearly. The resulting algebraic loop is a mixed Real/Boolean system of equations, which is very similar to the corresponding mixed system of equation of a pure bearing friction model, and can be solved with the same methods, see, e. g., [8].

2.3 Measurement of Gear Losses

Efficiency measurement data provided in gearbox catalogues contain usually not enough information for a dynamic simulation (e. g., the losses for $\omega_A = 0$ are not given). The reason is that in many cases only the overall efficiency is included as a function of load torque for some constant angular velocities $\omega_A \neq 0$.

In order to obtain the data needed for the `lossTable` of the *Modelica* model, the following measurement method is proposed:

For $m \geq 2$ fixed load torques $\tau_{B,j}$ (e. g., nominal torque and half of the nominal torque) and n angular velocities $\omega_{A,k}$, the necessary driving torques τ_A are measured which are needed to drive the gear for positive and negative energy flow $P_A = \omega_A \tau_A$, including measurements near $\omega_A = 0$ (= the gear shafts start to rotate).

As a result, the following values are obtained:

$$\tau_{A,j}(\omega_{A,k}), \quad \tau_{B,j} \quad (j = 2..m, \quad k = 1..n).$$

For every fixed speed $\omega_{A,k}$, equation (9) can be formulated in the unknowns $\hat{\eta}_{mf}$ and $\hat{\tau}_{bf}$. Collecting all equations together results in one linear system of equations

$$\begin{bmatrix} -i\tau_{A,1}(\omega_{A,k}) & 1 \\ -i\tau_{A,2}(\omega_{A,k}) & 1 \\ \vdots & \vdots \\ -i\tau_{A,m}(\omega_{A,k}) & 1 \end{bmatrix} \begin{bmatrix} \hat{\eta}_{mf} \\ \hat{\tau}_{bf} \end{bmatrix} = \begin{bmatrix} \tau_{B,1} \\ \tau_{B,2} \\ \vdots \\ \tau_{B,m} \end{bmatrix} \quad (17)$$

for every fixed speed $\omega_{A,k}$. If more than two load torque measurements are available, (17) has no solution and is solved in a least square sense. For two load

torque measurements, a unique solution exists

$$\hat{\eta}_{mf}(\omega_{A,k}) = -\frac{\tau_{B,1} - \tau_{B,2}}{i(\tau_{A,1} - \tau_{A,2})} \quad (18)$$

$$\hat{\tau}_{bf}(\omega_{A,k}) = \frac{1}{i}\tau_{B,2} + \hat{\eta}_{mf}\tau_{A,2}. \quad (19)$$

Finally, $\eta_{mf1}(\omega_{A,k})$, $\eta_{mf2}(\omega_{A,k})$, $\tau_{bf1}(\omega_{A,k})$ and $\tau_{bf2}(\omega_{A,k})$ can be easily determined from $\hat{\eta}_{mf}(\omega_{A,k})$ and $\hat{\tau}_{bf}(\omega_{A,k})$ based on the sign of $\omega_A \tau_A$ using equations (10) and (11).

As already mentioned, in gearbox catalogues usually the overall efficiency $\eta = -P_B/P_A$ is provided as function of the load torque τ_B . To demonstrate that the presented loss model produces qualitatively the same result, the overall efficiency of the following example with the loss model

$$\begin{aligned} \eta_{mf1} &= 0.97 \\ \tau_{bf1}/\tau_{Bmax} &= 0.01(\omega_B^2 + 2\omega_B + 5) \end{aligned}$$

is shown in figure 6. As can be seen, the typical hyperbolic curves are present, although the mesh efficiency is constant.

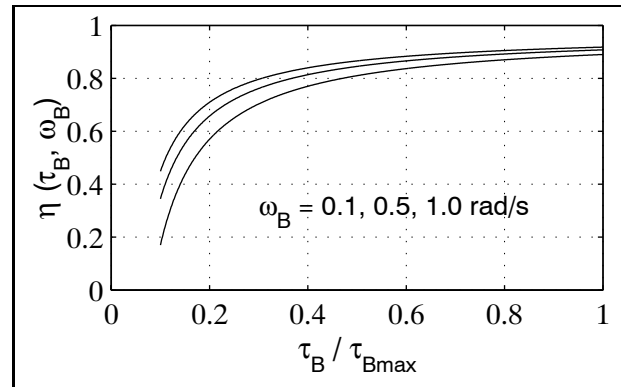


Figure 6: Overall efficiency as function of τ_B .

3 Planetary Gear

In this section the frictional effects of planetary gears are mathematically described in a similar way as in

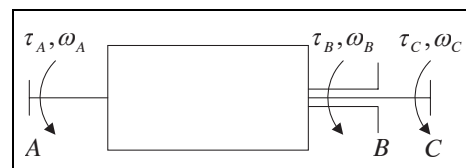


Figure 7: Speeds and cut-torques of a planetary gear

the previous section (based on [9]) and an appropriate *Modelica* model is derived. The variables describing a planetary gear are shown in figure 7, where ω_A , ω_B , ω_C denote the angular velocities of shafts A, B, C and τ_A , τ_B , τ_C denote the torques at the cut planes of the shafts, respectively. The examined gears are defined as follows:

Definition 2: A gear denoted as *planetary gear* in this article has the following properties:

- The gear has *three external shafts*.
- The gear has *two degrees of freedom*.
- The time invariant constraint equation

$$\omega_{AB} = i_0 \omega_{CB} \quad (20)$$

holds, with

$$\omega_{AB} = \omega_A - \omega_B$$

$$\omega_{CB} = \omega_C - \omega_B$$

where the so-called stationary gear ratio i_0 is *constant* and is in the range $i_0 \leq -1$ or $i_0 > 1$ (for i_0 outside of this range, the role of shafts A and C has just to be exchanged. However, $i_0 = 0$ and $i_0 = 1$ is never possible).

Note, that Willis' equation, see, e. g. [4], is equivalent to (20). A large class of planetary gears matches to this definition. Some examples are shown in figure 8. The stationary gear ratio i_0 is usually computed from the teeth number of the gear wheels. For example, for the gearbox in the left upper corner of figure 8, $i_0 = z_r/z_s$, where z_s is the number of teeth for the inner sun wheel and z_r is the number of teeth for the outer ring wheel (teeth numbers are taken negative for internal teeth).

3.1 Mathematical Description

The relationship between the angular velocities of the three shafts shown in figure 7 can be described using relative kinematics yielding

$$\omega_A = \omega_{B0} + \omega_{AB} \quad (21)$$

$$\omega_B = \omega_{B0} \quad (22)$$

$$\omega_C = \omega_{B0} + \omega_{CB} . \quad (23)$$

The structure of (21)-(23) reflects the superposition of two movement types:

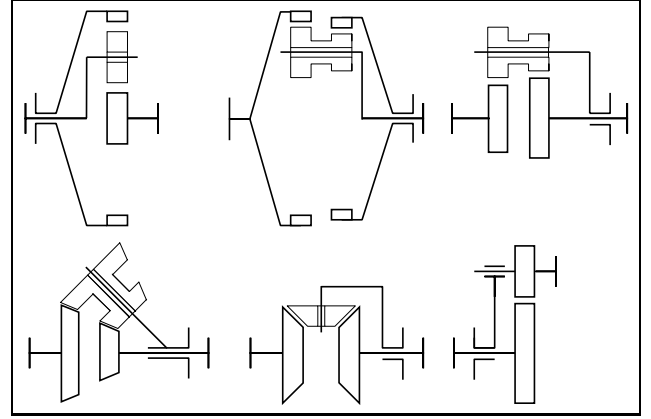


Figure 8: Examples of planetary gears according to definition 2 [4]

- **Block movement.** The whole gear rotates as one fixed block with angular velocity ω_{B0} :

$$\omega_A = \omega_B = \omega_C = \omega_{B0} .$$

During this movement a power P_1 is transmitted solely by rigid coupling of the three shafts. As the three shafts are not rotating relative to each other, any losses due to friction between the teeth of the gear wheels or in internal bearings cannot arise, i. e., the block movement is without losses.

- **Stationary gear movement.** Shaft B is fixed relative to the inertial system:

$$\omega_{B0} = 0 \Rightarrow \omega_A = \omega_{AB} \quad \omega_C = \omega_{CB} .$$

During this movement a power P_2 is transmitted solely by sliding of teeth in all three gear wheels resulting in power losses due to friction between the teeth of the gear wheels and in internal bearings not related to the external shafts. Since the shaft speeds are a function of ω_{AB} , ω_{CB} and $\omega_{CB} = \omega_{AB}/i_0$ due to (20), losses only occur, if $\omega_{AB} \neq 0$.

In order to achieve further equations energy flow conservation is considered according to figure 9 involving

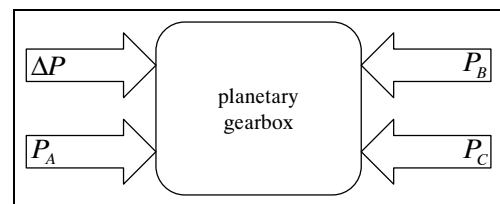


Figure 9: Energy flow

the energy flows in the shafts, P_A, P_B, P_C , and the friction losses ΔP which dissipate to heat:

$$P = P_A + P_B + P_C + \Delta P = 0 \quad (24)$$

with

$$P_A = \tau_A (\omega_{B0} + \omega_{AB}) \quad (25)$$

$$P_B = \tau_B \omega_{B0} \quad (26)$$

$$P_C = \tau_C (\omega_{B0} + \omega_{CB}) . \quad (27)$$

$$\Delta P = -\Delta \tau \omega_{AB} . \quad (28)$$

Since losses can only occur if $\omega_{AB} \neq 0$, the power loss ΔP has been formulated as the product of ω_{AB} and a, yet unknown, virtual loss torque $-\Delta \tau$. Using (24)-(27) and (20) results in

$$P = \underbrace{\omega_{B0} (\tau_A + \tau_B + \tau_C)}_{P_1} + \underbrace{\omega_{AB} (\tau_A + \tau_C / i_0 - \Delta \tau)}_{P_2} \quad (29)$$

Since a planetary gearbox has two degrees of freedom (see definition 2), the two speeds ω_{B0} and ω_{AB} can have arbitrary values which are independent from each other. Therefore the speed factors must vanish

$$0 = \tau_A + \tau_B + \tau_C \quad (30)$$

$$0 = \tau_A + \tau_C / i_0 - \Delta \tau \quad (31)$$

By solving (31) for τ_C and (30) for τ_B , the two equations can be alternatively formulated as

$$\tau_C = i_0 (-\tau_A + \Delta \tau) \quad (32)$$

$$\tau_B = (i_0 - 1) \tau_A - i_0 \Delta \tau \quad (33)$$

When stationary gear movement occurs, i.e., $\omega_{B0} = 0$, the planetary gear reduces to a standard gear with two external shafts where the losses are described according to equation (9)

$$\tau_C = i_0 (-\hat{\eta}_{mf} \tau_A + \hat{\tau}_{bf}) \quad (34)$$

where $\hat{\eta}_{mf}(\omega_{AB})$ describes mesh friction

$$\hat{\eta}_{mf} := \begin{cases} \eta_{mf1}(|\omega_{AB}|) & : \begin{cases} \tau_A \omega_{AB} > 0 \text{ or} \\ \tau_A = 0 \text{ and } \omega_{AB} > 0 \end{cases} \\ 1/\eta_{mf2}(|\omega_{AB}|) & : \begin{cases} \tau_A \omega_{AB} < 0 \text{ or} \\ \tau_A = 0 \text{ and } \omega_{AB} < 0 \end{cases} \\ \text{so that } \dot{\omega}_{AB} = 0: & \omega_{AB} = 0 \end{cases} \quad (35)$$

with $\eta_{mf1}, \eta_{mf2} \in [0; 1]$ and $\hat{\tau}_{bf}$ describes friction in the internal bearings of the planetary gearbox (e. g., for a standard planetary gearbox with sun, planet and ring

wheel, $\hat{\tau}_{bf}(\omega_{AB})$ is the bearing friction torque in the planet bearings)

$$\hat{\tau}_{bf} := \begin{cases} \tau_{bf1}(\omega_{AB}) & : \begin{cases} \tau_A \omega_{AB} > 0 \text{ or} \\ \tau_A = 0 \text{ and } \omega_{AB} > 0 \end{cases} \\ \tau_{bf2}(\omega_{AB}) & : \begin{cases} \tau_A \omega_{AB} < 0 \text{ or} \\ \tau_A = 0 \text{ and } \omega_{AB} < 0 \end{cases} \\ \text{so that } \dot{\omega}_{AB} = 0: & \omega_{AB} = 0 \end{cases} \quad (36)$$

with

$$\hat{\tau}_{bf}(\omega_{AB}) = \begin{cases} \geq 0 & : \omega_{AB} > 0 \\ \leq 0 & : \omega_{AB} < 0 \end{cases} . \quad (37)$$

No losses will be additionally introduced when a block movement is superpositioned, as discussed previously. Therefore, (34) is also valid for a general movement. Comparison of (34) with (32) results in

$$i_0 (-\tau_A + \Delta \tau) = i_0 (-\hat{\eta}_{mf} \tau_A + \hat{\tau}_{bf})$$

and therefore

$$\Delta \tau = (1 - \hat{\eta}_{mf}) \tau_A + \hat{\tau}_{bf} . \quad (38)$$

As energy can dissipate only,

$$\Delta P = -\omega_{AB} \Delta \tau \stackrel{!}{\leq} 0 \quad (39)$$

$$= -\omega_{AB} ((1 - \hat{\eta}_{mf}) \tau_A + \hat{\tau}_{bf}) \quad (40)$$

$$= -(1 - \hat{\eta}_{mf}) \tau_A \omega_{AB} - \omega_{AB} \hat{\tau}_{bf} \quad (41)$$

Since

$$1 - \hat{\eta}_{mf} \geq 0 \quad \text{for} \quad \tau_A \omega_{AB} \geq 0$$

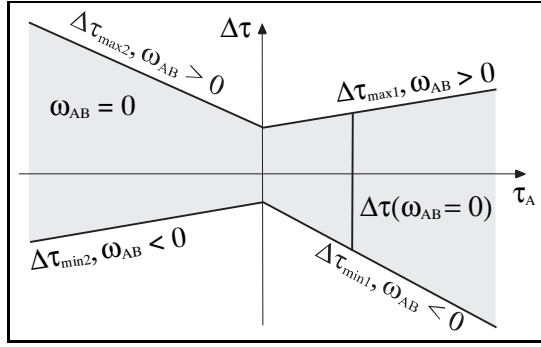
$$1 - \hat{\eta}_{mf} \leq 0 \quad \text{for} \quad \tau_A \omega_{AB} \leq 0$$

according to (35), the first term in (41) is never positive. With (37) the same also holds for the second term and therefore ΔP is in fact never positive. Utilizing (35)-(37) and (38) for the sliding case, results in the equations of table 3 to actually calculate $\Delta \tau$.

In the stuck mode the planetary gear rotates without any losses as a block. Similar to Sec. 2 the torque loss $\Delta \tau$ is then defined implicitly by the constraint equation $\dot{\omega}_{AB} = 0$. The gear remains in sliding mode until ω_{AB} becomes zero. It remains in stuck mode as long as the calculated torque loss $\Delta \tau$ is lying in the stuck region according to figure 10.

ω_{AB}	τ_A	$\Delta \tau =$
> 0	≥ 0	$(1 - \eta_{mf}) \tau_A + \tau_{bf1} \quad (= \Delta \tau_{\max 1} \geq 0)$
> 0	< 0	$(1 - 1/\eta_{mf}) \tau_A + \tau_{bf2} \quad (= \Delta \tau_{\max 2} \geq 0)$
< 0	≥ 0	$(1 - 1/\eta_{mf}) \tau_A - \tau_{bf2} \quad (= \Delta \tau_{\min 1} \leq 0)$
< 0	< 0	$(1 - \eta_{mf}) \tau_A - \tau_{bf1} \quad (= \Delta \tau_{\min 2} \leq 0)$

Table 3: $\Delta \tau = \Delta \tau(\omega_{AB}, \tau_A)$ in sliding mode

Figure 10: $\Delta\tau$ in sliding and stuck mode

$ \omega_{AB} $	η_{mf1}	η_{mf2}	$ \tau_{bf1} $	$ \tau_{bf2} $
\vdots	\vdots	\vdots	\vdots	\vdots

Table 4: Format of table `lossTable`

3.2 Modelica Model

The planetary gear loss model derived in this section can be implemented as a *Modelica* model in a similar way as described in Sec. 2. The parameters to be provided are the stationary gear ratio i and table, `lossTable` to define the gear losses, see table 4.

Whenever η_{mf1} , η_{mf2} , τ_{bf1} or τ_{bf2} are needed, they are determined by interpolation in `lossTable`. The interface of this *Modelica* model is therefore defined as

```
parameter Real i = 1;
parameter Real lossTable[:,5]
           = [0, 1, 1, 0, 0];
```

using the unit gear ratio and no losses as a default. The comments about model interna given in Sec. 2 are valid similarly for the planetary gear model.

This *Modelica* model can be connected with component `Modelica.Mechanics.Rotational.BearingFriction` at each shaft to model additionally the friction influence of bearings related to the external shafts. As a consequence multiple friction phases arise with the phenomena explained, e. g., in [8].

4 Simulation Results

In this section simulation results are presented for models containing the standard and planetary gear models with losses developed in the last sections, using the *Modelica* modeling and simulation environment Dymola, version 4.2a [1].

4.1 Standard gear with mesh friction

Figure 11 contains a Dymola screenshot of the model under consideration. It is a standard gear with mesh friction that is driven by a sinusoidal torque and has a load torque which is linearly increasing.

In figure 12 and 13 results of a simulation are shown for $\eta_{mf} = 0.5$ and $\eta_{mf} = 0.9$: The thicker lines are the loss torques $\Delta\tau$ whereas the thinner lines characterize whether mesh friction is in forward sliding (mode=+1), backward sliding (mode=-1) or stuck (mode=0) mode.

As can be seen in the upper part of figure 12 from the two stuck modes, the maximum loss torque is not constant (as it is for bearing friction) but depends on the driving torque. Additionally, figure 13 contains the speed of inertia 2 for $\eta_{mf} = 0.5$. During stuck mode, the velocity vanishes.

4.2 Gear shift dynamics of automatic gear

The mesh friction model for planetary gears as well as the clutch friction model already available in the Mod-

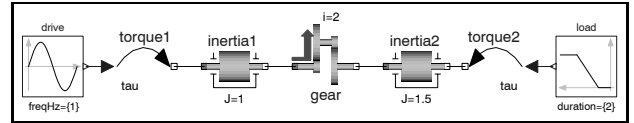
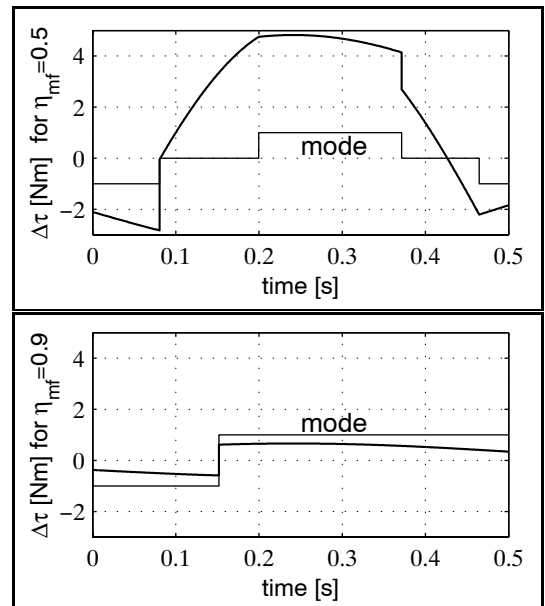
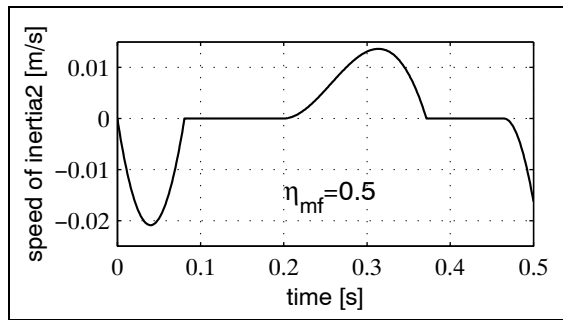


Figure 11: Modelica composition diagram of gear with mesh friction.

Figure 12: Loss torque $\Delta\tau$ and mode in gear with mesh friction $\eta_{mf} = 0.5$ and $\eta_{mf} = 0.9$.

Figure 13: Speed of inertia 2 for $\eta_{mf} = 0.5$.

elica standard library are very well suited to simulate the shift dynamics of automatic gearboxes reliably and efficiently. As an example, the shift dynamics of the automatic gearbox ZF 4HP22 is examined in more detail. A schematic together with the gear shift table is given in figure 14 (from [2]).

From this schematic it is straightforward to build up the Modelica composition diagram from figure 18 containing the clutches, combined clutches and free wheels, and the three planetary gears with mesh friction ($\eta_{mf} = 0.975$).

In order that simulations can be performed, a model of the environment in which the automatic gear operates is needed. A typical example of a longitudinal dynamics model of a vehicle is shown in figure 19. It consists of a driver, an engine, an automatic gearbox with transmission control unit, an axle and a simple 1-dimensional vehicle model containing the most important drive resistances.

Signals, such as desired vehicle velocity or throttle position, are transported between the components by a signal bus. Via `send` and `receive` blocks, signals can be send to or received from the bus connector bus

Gear	C4	C5	C6	C7	C8	C11	C12
1	x					x	
2	x		x	x		x	
3	x	x		x		x	
4	x	x		x			x
R		x			x	x	

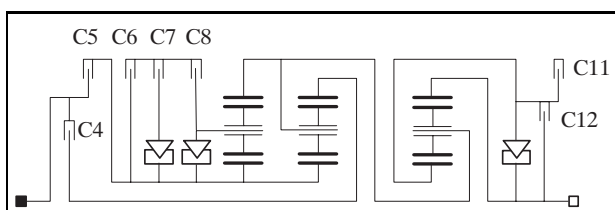


Figure 14: Gear shift table for gearbox ZF 4HP22.

which is a Modelica connector containing declarations of all variables present in the bus.

Typical simulation results of the model are shown in figure 15-17, especially in figure 15 the desired and actual velocity of the vehicle in km/h, in figure 16 the vehicle acceleration and the actual gear determined by the simple transmission control unit, and in figure 17 the torque loss $\Delta\tau$ of the right most planetary gearbox p3.

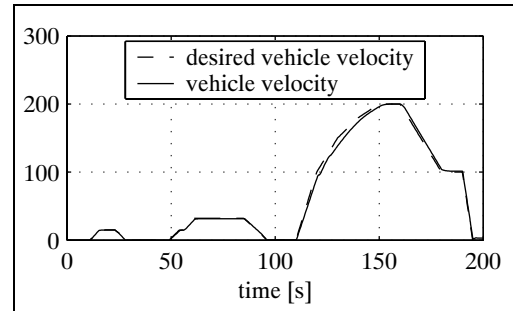


Figure 15: Desired and actual velocity of vehicle.

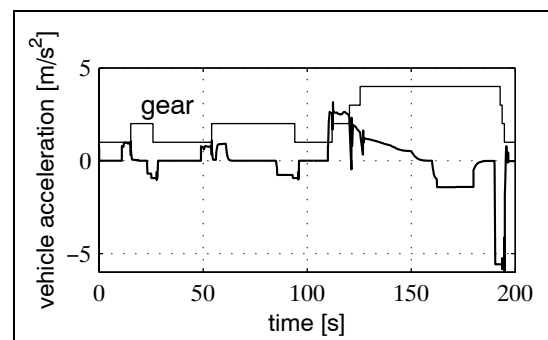
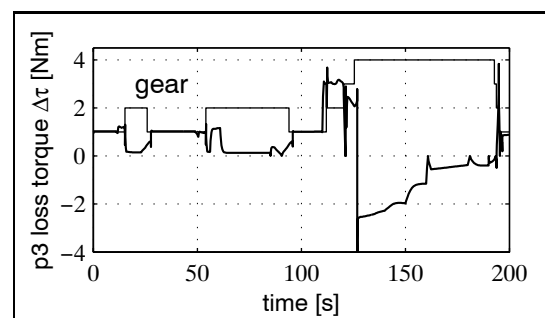


Figure 16: Vehicle acceleration and actual gear.

Figure 17: Loss torque $\Delta\tau$ in planetary gear p3.

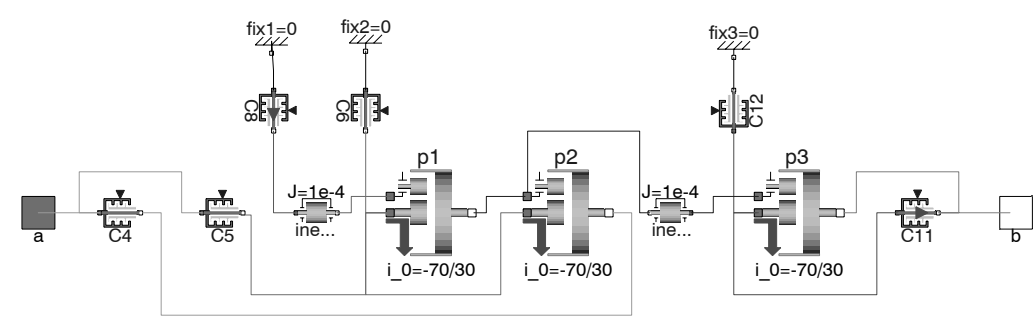


Figure 18: Modelica composition diagram of automatic gearbox ZF 4HP22.

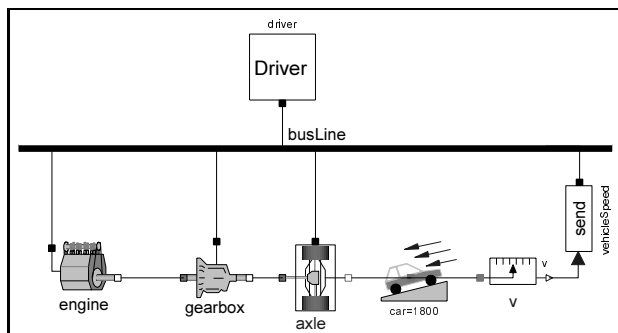


Figure 19: Modelica composition diagram of vehicle longitudinal dynamics.

5 Conclusions and Outlook

A loss model for a broad class of standard gears and planetary gears has been presented which includes Coulomb friction in the gearbox bearings and Coulomb friction between the gear teeth. Most important, the locking and unlocking of the friction elements are handled, including the friction between the gear teeth. This allows to model and simulate the stick-slip effect of standard and planetary gears as function of the shaft speeds and the driving or load torque which is essential for the design of servo drives.

The usual approach to model mesh friction as an element which switches between two different efficiencies, leads in such situations to *chattering*, i. e., very fast switching between the two possible modes which in turn results in very small step sizes and practically stops the simulation. The new approach described here will lead to much more reliable and more efficient simulations.

The gear losses in standard gears have been implemented in a new model `LossyGear` which will be available in the next version of the `Modelica.Mechanics.Rotational` library. The gear loss

model for planetary gears has been implemented in a new model `LossyPlanetary` which will be available in the next version of the `PowerTrain` library.

6 Acknowledgements

This work was in parts supported by "Bayerisches Staatsministerium für Wirtschaft, Verkehr und Technologie" under contract AZ300-3245.2-3/01 for the project "Test und Optimierung elektronischer Steuergeräte mit Hardware-in-the-Loop Simulation", and by the European Commission under contract IST-199-11979 for the project "Real-time simulation for design of multi-physics systems".

References

- [1] Dymola: "Homepage: <http://www.dynasim.se/>."
- [2] Förster H.: *Automatische Fahrzeuggetriebe*. Springer, 1991.
- [3] Köhler G., and Rögnitz G.: *Maschinenteile, Teil 2*. Teubner, 8th ed., 1992.
- [4] Loomann J.: *Zahnradgetriebe*. Springer, 3rd ed., 1996.
- [5] Modelica libraries: "Homepage: <http://www.modelica.org/libraries.shtml>."
- [6] Niemann G., Winter H., and Höhn B.: *Maschinenelemente, Band 1*. Springer 3rd ed., 2001.
- [7] Niemann G., and Winter H.: *Maschinenelemente, Band 2*. Springer, 2nd ed., 1989.
- [8] Otter M., Elmqvist H., and Mattsson S.-E.: "Hybrid Modeling in Modelica based on the Synchronous Data Flow Principle," in *CACSD'99, Hawaii, USA*, August 22–26, 1999.
- [9] Pelchen C.: "Wirkungsgradbehandlung bei Planetensatz und Standübersetzung." Personal communication to M. Otter and C. Schweiger, Nov. 2001.

Modularised Tyre Modelling in Modelica

Johan Andreasson and Jonas Jarlmark

Div. of Vehicle Dynamics

Dept. of Vehicle Engineering

Royal Institute of Technology

SWEDEN

E-mail: {johan,jonask}@fkt.kth.se

Abstract

Good tyre models are essential for driving simulation of all ground vehicles using pneumatic tyres. However, tyre behaviour is extremely complex, often requiring different models for the various behaviours. This paper presents an implemented tyre model library that takes advantage of the modular modelling possibilities in Modelica to combine different models. For example, the different sub models representing tyre-to-road contact can be combined with various models for tyre belt behaviour. The library can be used together with vehicle models in one, two and three dimensions.



Figure 1: *The tyre construction of a normal passenger car tyre. 1 - tread, 2 - sidewall, 3 - radial cord, 7 - belt, 9 - bead.*

1 Introduction

Complex tyre behaviour is a direct result of the tyre construction. While looking much like a simple rubber doughnut attached to the rim, the tyre construction is vastly more complex, figure 1.

The two main functions of the tyre, is force generation in the road plane and suspension of the vehicle mass. The force generation is made possible by the rubber tread, causing a high friction coefficient with the road surface. The suspension of the vehicle mass is managed by the carcass, consisting of the belt, radial cords and beads. The radial cords work like the spokes of a bicycle rim; a pre-tension must be exerted by pressurised air inside the tyre to carry the loads. Because of the two main functions of the tyre, modelling of the behaviour can be split into modelling the functions of the tyre, rather than modelling the tyre itself.

Naturally, there are more areas to take into account when the combination of components for the tyre is to be selected and calculated. Areas such as comfort, traction and cornering are covered by

the main functions. There are also issues such as rolling radius, steering behaviour, force feed back, vibrations, sound radiation, water dispersion, directional stability to be considered when designing a tyre.

Depending on the type of modelling required, single, several or all areas involved need to have exclusive models allocated. The selection of the areas of interest are made by the modeller. Therefore, it can be of great help to possess a flexible way to add and subtract function models of the tyre.

When examining the main functions, the demand made on precision, processing time and comprehension, generates a need for different models for the same function. The suspension function is mostly modelled dynamically and is used for comfort purposes. A simple model could

be a spring - mass - damper system with one degree of freedom. If more complexity is required, a rigid or flexible belt model connected with pre-tensioned springs and dampers to the rim with correct geometry as presented by Böhm [1] will give a fairly good representation, figure 2. The most complex models are the full FE-models with anisotropic cord, internal rubber damping, complex friction behaviour and temperature calculation.

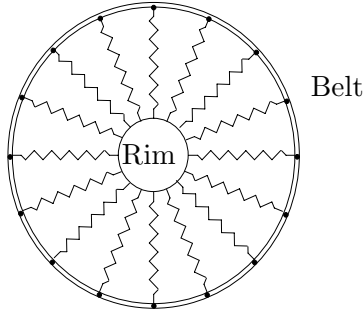


Figure 2: *Rigid belt model with pre-tensioned springs.*

For lateral and longitudinal direction, force generation models for cornering and traction are often empirical models, where measured data are fitted to polynomials to give an analytical function, figure 3, however, physical models also exist. A simple modelling approach is the use of a cornering stiffness; a linear relation between force and slip, valid only in a very limited area of handling. A simple but wider model would be a curve fit of the forces as a function of slip angle and slip ratio with saturation and load dependency. A more complex models is the Magic Formula [2], developed by TU Delft and Volvo Cars Corporation. This model also includes force due to camber, rolling radius changes and rolling resistance. The Magic Formula will give lateral and longitudinal force, aligning moment and driving moment from various analytical functions. All these models assume point contact and do not take into account the width or length of the contact patch. This assumption simplifies the calculations and reduces the need for accurate road data.

2 Definitions

Because of the tyre complexity, several reference frames are necessary to model its behaviour. To

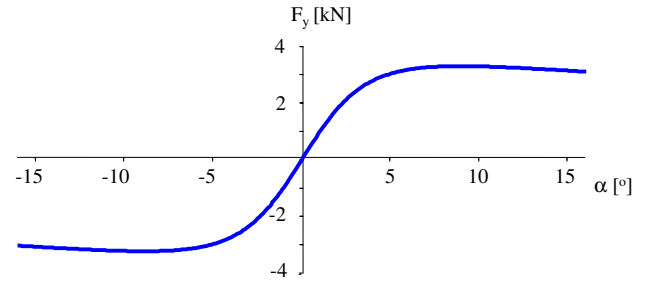


Figure 3: *Typical appearance for the lateral force, F_y , as function of slip angle, α , in this case generated with the Magic Formula, [3].*

ensure that the model structure allows the simple addition and reuse of components within new models, the modelling is based on two international standards; DIN and TYDEX. According to the DIN coordinate representation, illustrated in figure 4, x points forward, y to the left and z right up. The TYDEX definition of the carrier frame, C , and contact frame, W , is shown in figure 5. The carrier frame is fixed at the car's suspension and the contact frame is located at the intersection of the carrier frame's z -axis and the road plane. Generally, the contact point is not located at the origin of frame W and therefore a frame W' is introduced. Frame R , located at the rim, is used since the forces and torques generated by the tyre cannot act directly at the carrier.

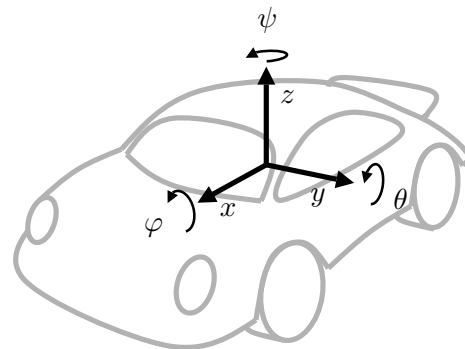


Figure 4: *The coordinates according to DIN. Yaw, pitch and roll are indicated with ψ , θ and φ , respectively.*

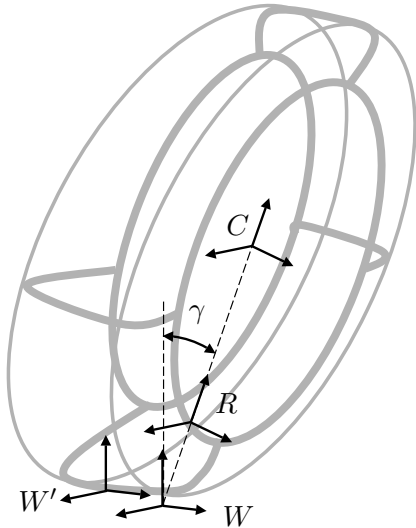


Figure 5: The frames used in the modelling. The deformation of the tyre belt results in a displacement of the contact point from W to W' . The carrier frame, C , and the rim frame, R , do not rotate with the wheel.

3 Model structure

The models that this modularisation is based on all assumes that the tyre-road contact patch can be approximated by a point. This assumption is used to define the cuts at the tyre-road, the tyre rim and the rim-carrier. With this modularisation, a tyre can be described with three parts, hub, tyre belt and tyre-road contact, as shown in figure 6. These components are described in more detail below, the frames referred to are indicated in figure 5.

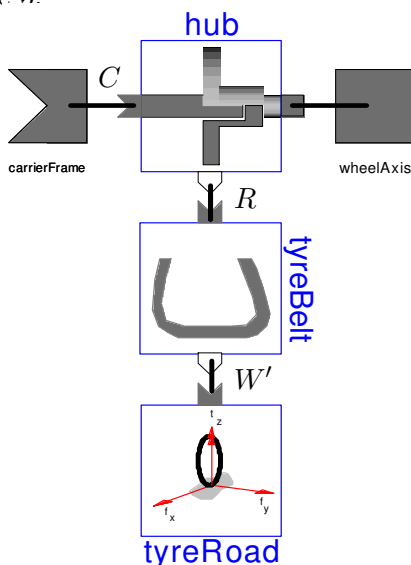


Figure 6: The tyre models consists of three main subcomponents: The hub, the tyre belt and the tyre-road contact.

3.1 Hub

The hub is mounted on the carrier and allows the wheel and drive axis to rotate around the y -axis. Since none of the frames rotates with the wheel, the Modelica-model of the hub transforms the forces and torques acting at the rim frame, R , to the carrier frame, C and the drive axis. This cannot be done by a standard component since the torque around the y -axis only should act at the drive axis.

3.2 Tyre belt

The tyre belt carries the load on the wheel distributing the forces from the tyre-road contact patch to the rim. As mentioned earlier, the patch is approximated with a point. The Modelica-model of the tyre belt describes the relation between the contact point frame, W' , and the rim frame, R . There are several ways of implementing this where the simplest rim model neglects the "horizontal" deformation of the tyre belt which gives $W' = W$.

3.3 Tyre-road contact

The tyre-road contact generates friction forces and torques that act on the tyre belt. The Modelica model calculates and applies these to the contact point frame, W' . Most of these models are empirical and generates forces as functions of slip, camber angle etc. An example is the Magic Formula model used in figure 3.

4 Utilities

In addition to the main components described in section 3.1-3.3, other components are realised to simplify the extension of the `Wheels` library. Some of which are described in more detail below.

4.1 State handler

This component is introduced since the tyre models are not strictly physical. This is due to the modularisation and to the fact that some tyre states cannot be calculated from the simpler models. For example, $x-y$ vehicle models, as the bicycle model [3], require that the normal force and the camber angle are given as parameters. More detailed models, where this information is required,

can be used in a convenient way by using this state handler.

4.2 Contact point

The **ContactPoint** component is used within the **TyreBelt** to calculate frame W . Depending on the implementation this can be implemented in several ways. As described in [4], external road information can be provided via the **inner/outer** Modelica constructs. Information needed to calculate W is the road normal and altitude as a function of longitude and latitude.

4.3 Camber angle

The camber angle is the angle between the road normal and the z -axis of the carrier frame, indicated with γ in figure 5. The implementation in Modelica is more general and calculates the angle between the z -axes of any pair of frames.

4.4 Tyre radius

There are at least three different tyre radii, undeformed radius, R , effective rolling radius, R_e and loaded radius, R_l . The loaded radius is the wheel centre's height over ground, while the effective rolling radius is a measure of how far a free rolling tyre travels per revolution. Generally, $R_l < R_e < R$. This means that the rolling tyre travels farther per revolution than determined by using the wheel's centre height as rolling radius. Mixing these definitions would thus lead to erroneous simulation results.

In the **Wheels** library, the loaded radius, can be calculated at any time, while the effective rolling radius as well as the undeformed radius must be given as parameters or functions.

5 Interfaces

As mentioned earlier, the **Wheels** library can be used in one, two and three dimensions. The interfaces for these are taken from the **Modelica.Mechanics.Translational** (1D) and **ModelicaAdditions.MultiBody** (3D), [5]. To handle the the two-dimensional models, a library **MultiBody2D** has been realised. Additionally, for example, the interfaces from **Modelica.Mechanics.Rotational** are used for the wheel

axis and thus, the wheels can be connected with available packages for power-trains.

To guarantee that the different tyre models are replaceable, they all are based on interface models, **BaseWheel[*]**, where $[*]$ is the dimensions they are meant for, e.g. **BaseWheelXY** for a wheel in the XY plane. In total, seven different interfaces are realisable, X, Y, Z, XY, XZ, YZ and XYZ.

5.1 MultiBody2D

This library is used to describe two-dimensional mechanical systems and is based on the same ideas as the three dimensional **ModelicaAdditions.MultiBody** library, [5]. Instead of three DOF each for translation and rotation as in the 3D-case, this library has two DOF for translation and one DOF for rotation, allowing only planar motion. The Modelica definition of the interface is:

```
connector Frame_a
  Modelica.SIunits.Position r0[2];
  Real S[2, 2];
  Modelica.SIunits.Velocity v[2];
  Modelica.SIunits.AngularVelocity w;
  Modelica.SIunits.Acceleration a[2];
  Modelica.SIunits.AngularAcceleration z;
  flow Modelica.SIunits.Force f[2];
  flow Modelica.SIunits.Torque t;
end Frame_a;
```

In figure 7, parts of the **MultiBody2D**-library are shown.

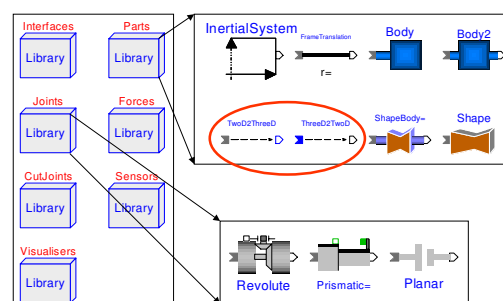


Figure 7: The *MultiBody2D* library. The parts indicated provide an interface to the *ModelicaAdditions.Multibody* library.

6 Usage

To get a better idea of how the `Wheels` library can be used, some examples and aspects are presented below.

6.1 Bicycle model

The bicycle model is a very simple model of a vehicle. However, it provides essential information about a vehicle's behaviour. The motion is restricted to the x - y plane and the, normally, two wheel per axis are replaced by one single wheel, i.e. a bicycle with the centre of gravity in the ground plane.

In figure 8, a bicycle model of a car with a trailer is shown. The bicycle is connected to an inertial system (1) via a planar joint (2) allowing x - y translation and rotation around z . The bicycle consist of a body (3) with mass and inertia, translations (4,5) from the centre of gravity to the front and rear wheels and a revolute joint for the steering (6). The wheels (7,8) could be of any kind as long as they use the `BaseWheelXY` interface. Additionally, a connector COG (9) makes it possible to attach further components to the model. In this case a trailer is attached, it would also be possible to add, for example, aero dynamical models and sensors.

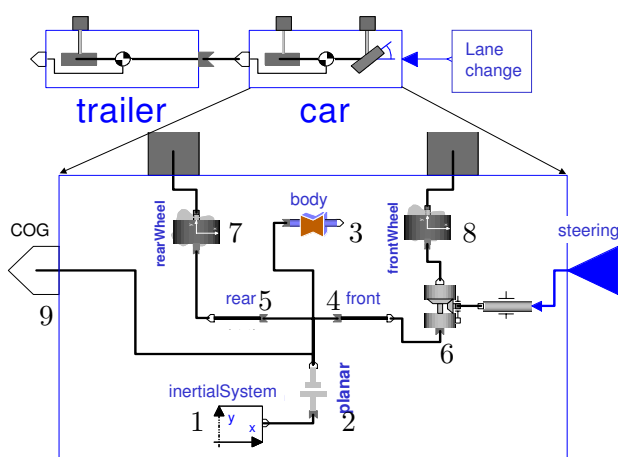


Figure 8: An implementation of a bicycle model with the `MultiBody2D` library, note how simple it is to add a trailer.

6.2 Switching tyre models

Modelica provides a convenient way of replacing models with the same interface using

`replaceable` and `redeclare`. To illustrate how this works, an example with a replaceable front wheel in a bicycle model in the x - y plane is discussed.

Since all wheel models for the x - y plane have the same interface, `BaseWheelXY`, this is used within the declaration of the bicycle model:

```
model Bicycle
  replaceable BaseWheelXY frontWheel;
  ...
end Bicycle;
```

Whenever a bicycle model is used, the tyre model can easily be replaced with any x - y model:

```
model AnyXY
  extends BaseWheelXY;
  ...
end AnyXY;
```

```
model BicycleTest
  Bicycle bicycle(redeclare
    AnyXY frontWheel);
  ...
end BicycleTest;
```

Without this feature, a new model, in this case the Bicycle model, would have to be realised for each combination of wheel models studied.

6.3 Parameterisation

The more complex a tyre model gets, the more parameters are required. As an example, the Magic Formula model to calculate the steady-state forces and torques acting at the contact point requires 73 parameters. Since the parameter sets differ from model to model, it would be lengthy to define each parameter value within a `redeclare` phrase as described in section 6.2.

Instead, the parameter sets are gathered in `records` that are used like structural variables. These records can then be treated as single parameters and be stored as models in the library.

6.4 Adding components for visualisation

Due to the modularisation, additional components can easily be attached. To illustrate this, an example where animation information is added to the tyre is shown. In figure 9, components named `contact` and `contactPatch` are added. `contact`

provides visual information about the forces as vector shapes acting at the cut. In this case, the

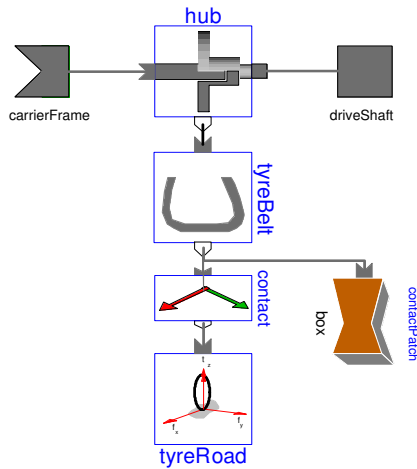


Figure 9: Since the components are separated it is easy to add additional information, in this case graphical information.

contact point forces are visualised but this component could of course be attached at any cut to illustrate forces. `contactPatch` is an ordinary box shape that visualises the contact point as a square. This is a very simple way to to illustrate the tyre belt deformation. Figure 10 shows a snapshot of the animation result.

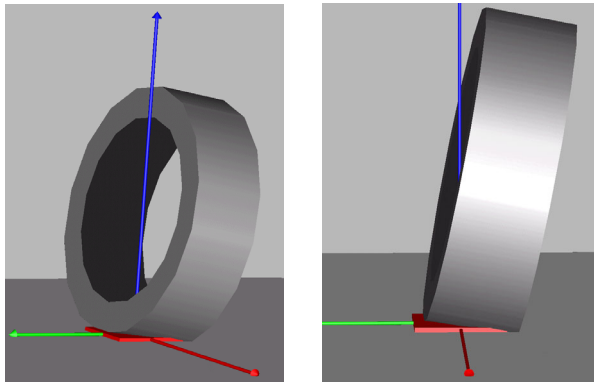


Figure 10: These animation pictures shows forces in x -, y - and z - direction of the W' frame. The tyre belt deformation is indicated by the red square.

6.5 Mixing dimensions

Up to this point it has been assumed that the tyre model used has the same dimensions as the car model that is used. However this is not a necessity as will be shown in the following example.

Assume that the y - z motion of a car is to be studied, when driving with constant speed on a highway, and that an XYZ model is to be used. With the component `TwoD2ThreeD`, a connection between the tyre model in 3D and the car in 2D can be made as illustrated in figure 11. The additional information needed for the 3D frame, such as longitudinal speed, is given as parameters.

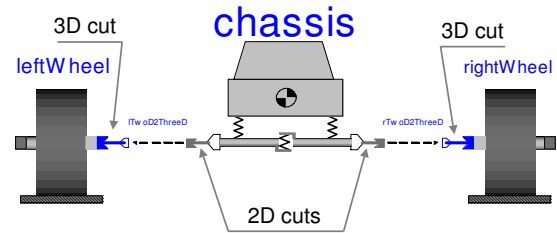


Figure 11: An example of how dimensions can be mixed. The `TwoD2ThreeD` component, allows a 3D wheel to be connected to a 2D car model.

6.6 Choosing accurate models

Even if the modularisation eases the modelling work, the need to understand the models, in order to know their limitations is still a necessity. Performing manoeuvres that takes the tyre model outside the valid range may have fatal consequences as seen in the following example:

With a vehicle model, two steering step inputs of 0.03 rad and 0.2 rad at the front wheels are simulated. First, a linear tyre-road contact model is used and then, the same simulations are performed with a Magic Formula tyre-road contact model. In the first case, the wheel slip is low and both models behave similarly, but for the second case, the linear model is no longer relevant, figure 12.

In the above example, the limitation of the linear model is deliberately made obvious, but often it is harder to detect when the range of the model's validity is exceeded. To help the user avoid these situations, the plan is to add an alert procedure that checks crucial variables against model limitations. This function is not yet implemented.

6.7 Case study

To give an example of the usage of the `Wheels` library and further illustrate the necessity of cor-

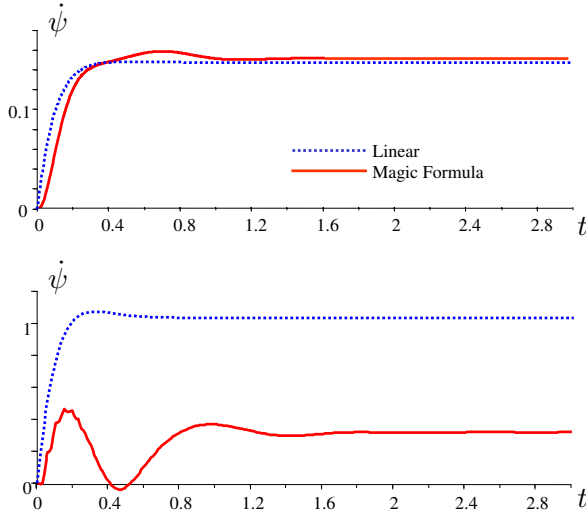


Figure 12: The plots show the yaw speed response. The upper plot shows similar results for both models while the lower plot shows how the linear model lacks ability to predict skidding. Obviously, this manoeuvre is outside the model's range of validity.

rect tyre models, a parameter study is performed. The concepts *understeer* and *oversteer* are familiar to many and a widely spread opinion is that a car with more "weight on the front wheels" will understeer while more "weight on the rear wheels" will make the car oversteer.

To show the limitation of this theory a variation of the location of the centre of gravity is compared to a variation of the tyres' maximum side force. The results are shown with the *understeer gradient*, K_{us} , which is a quantification of *how much more one has to turn the steering wheel to increase the lateral acceleration, compared to a neutral car*. Mathematically this is defined as

$$K_{us} = \frac{1}{i_s} \frac{\partial \delta_H}{\partial a_y} - \frac{\partial \delta_A}{\partial a_y} \quad (1)$$

where δ_H is the steering wheel angle and i_s the ratio between the steering wheel angle and the wheel angle. δ_A , called the *Ackermann angle*, is the wheel angle required to turn the car at low speed with no slip and a_y is the lateral acceleration.

Positive K_{us} thus means that the driver has to turn the steering wheel more to get the desired motion compared to driving a neutral car, i.e. the car understeers. If K_{us} is negative, less input has to be given and the car oversteers.

To perform the variation, the variables λ and μ

are introduced. λ indicates the distance from the front wheels to the centre of gravity. μ scales the height of the lateral force peak, shown at about $\alpha = 5^\circ$ in figure 3. Thus $\mu = 1$ does not affect the tyre and $\lambda = 0.5$ means that the centre of gravity is located half way between the wheel axes. The variation is done according to:

λ^0	$\lambda = 0.50$
λ^-	$\lambda = 0.45$
λ^+	$\lambda = 0.55$
μ^0	$\mu_{front} = 1.0, \mu_{rear} = 1.0$
μ^-	$\mu_{front} = 0.9, \mu_{rear} = 1.1$
μ^+	$\mu_{front} = 1.1, \mu_{rear} = 0.9$

The simulation is made using a bicycle model at constant speed. The steering wheel input is given as series of step increase. Between each step, the car stabilises and the lateral acceleration is logged. This information is then used to calculate K_{us} .

The result of the simulations is shown in figure 13. At low lateral accelerations, K_{us} depends solely on λ and the theory presented above holds. As a_y increases, μ gets more important and for high a_y , when the car skids, the tyre choice dominates. Neglecting this effect when designing a car would thus lead to unknown behaviour in critical situations.

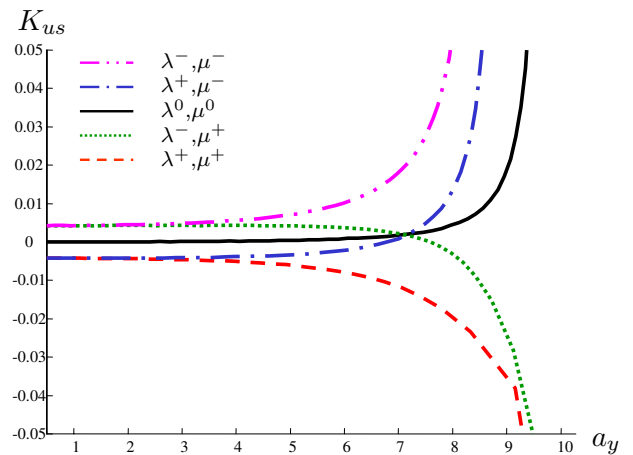


Figure 13: Result of the parameter variation.

7 Limitations

The largest limitation of the modularisation is that the selection of model accuracy must be problem based. This will require complete understanding of the problem by the user to be able to use the appropriate models. Another limit of the modularisation presented, is that the contact patch is assumed to be a point. This assumption is not valid for situations such as; driving on uneven surfaces, hitting pavements or crossing potholes. Additionally, when the steering input changes rapidly compared to the speed, for example when parking, the slip cannot be assumed to be constant over the whole patch which makes it much harder to use a contact point approximation.

8 Conclusions

The suggested division of the tyre model into hub, belt and road contact has simplified the reuse of model components and thus, made it easy to extend the `Wheels` library and to add new functionality. At the same time the coupling of behaviour that are functions of two or more parts is more complicated than without the separation.

The possibility to quickly replace the tyre models or sub-components when switching model accuracy will be of great advantage and this possibility will most certainly be used more frequently as the selection of components grows.

9 Acknowledgements

The authors would like to thank Sven-Erik Mattsson and Hans Olsson at Dynasim AB for their support, Martin Otter at DLR for support and code sharing and Boris Thorvald at Scania CV AB for constructive comments.

References

- [1] Böhm, *Mechanik des Gürtelreifens* Ingenieur-Archiv XXXV 1966, pp 82-101.
- [2] Bakker, Pacejka, Lidner. *A New Tire Model with an Application in Vehicle Dynamics Studies* SAE paper 890087
- [3] Gillespie. *Fundamentals of Vehicle Dynamics* ISBN 1-56091-199-9.
- [4] Andreasson, Möller, Otter. *Modeling of a Racing Car with Modelica's Multi-Body Library*. Paper presented at the Modelica Workshop 2000, Oct. 23-24, 2000, Lund, Sweden.
- [5] Modelica Association *ModelicaAdditions. MultiBody* <http://www.Modelica.org/library/library.html>
- [6] The Modelica Association *Modelica specification, v1.4* <http://www.modelica.org/>

The Modelica Flight Dynamics Library

D. Moormann and G. Looye
German Aerospace Center (DLR)
Institute of Robotics and Mechatronics
Oberpfaffenhofen, D-82234 Wessling, Germany
phone: +49 8153 28 1068 / fax: +49 8153 28 1441
E-Mail: Dieter.Moormann@dlr.de, Gertjan.Looye@dlr.de

Abstract

The Modelica Flight Dynamics Library has been developed to model 6-degrees-of-freedom, nonlinear flight dynamics and flight systems. Using this library the multidisciplinary interaction between flight dynamics and systems can easily be understood and analyzed. In this contribution the main benefits of the Flight Dynamics Library, concerning model building and efficient code generation – in particular for nonlinear parametric simulations and trim computations – are discussed. The library has been successfully applied to the development of aircraft models for several flight control system design projects.

1 Introduction

The design of aircraft requires contributions from different disciplines that are usually represented by different specialized groups within the aircraft development process. In design and evaluation of controlled flight system dynamics this is obvious.

In particular, the basic **flight dynamics** model consists of a description of aircraft geometry and mass together with equations of motion and of environmental influences such as gravity, atmosphere, and wind/gust. Basic flight dynamics are affected by aerodynamics and propulsion, two other distinct disciplines involved. The flight dynamics interact with the onboard **systems**, which can be grouped into motivators, sensors, and controls. Note that motivators consist of control surfaces such as elevators, and actuators which drive them.

Optimizing the interaction between flight dynamics and systems is an important area of investigation to improve efficiency of operation. For example, control surfaces can be designed to be 'just-right' in size and dynamic performance in order to minimize mass

and drag of the aircraft, while still guaranteeing the required overall aircraft flight characteristics in case of failures.

Traditional aircraft models are built using domain specific software packages that best solve their specific task with respect to the different disciplines involved, e.g., flight mechanics, propulsion, controls and hydraulics. As a drawback, those packages usually have very limited capabilities with respect to other domains and thereby it is quite cumbersome to link the different model components together. Hence, to develop a comprehensive aircraft dynamics model with low engineering effort, it is necessary to apply a model description form that is well suited for all domains involved and meets the requirements for multidisciplinary aircraft model integration. This description form has to be equally expressive for flight dynamics and for systems, which includes mechanical, electrical, hydraulic, and discrete digital control elements.

For this purpose we propose an object-oriented modeling approach, developed as a general tool for a wide variety of systems described by differential and algebraic equations. The advantage of such an approach is that it is easy to understand and that it can be used to visualize the hierarchical decomposition of a complete system. For each discipline, reusable domain specific model libraries can be built to encapsulate pertinent knowledge. The ability to work with submodels of different granularity is helpful for the design of flight control systems, where it is necessary to work with system models **and** flight dynamics models concurrently. During the design iteration process it should be possible to adjust both the refinement of the system model and the complexity of the flight dynamics model.

An important feature of object diagrams is that they are not limited to block diagrams with signal-directed input/output behavior. In an object diagram, for ex-

ample, the constituents of flight dynamics can be connected naturally according to their physical energy flow interaction and it is not required to transform all objects into a mathematical block diagram form as it has to be done for block oriented control modeling environments such as MATLAB-SIMULINK.

This paper describes how nonlinear aircraft dynamics models can be composed using the MODELICA-Flight Dynamics Library. Its main benefits concerning model composition using object-oriented structuring principles are presented in section 2. Its benefits resulting from an efficient mathematical code generation are discussed in section 3, where special emphasis is put on code generation for efficient parametric simulation and on highly accurate and efficient trim procedures.

2 Interactive multi-point model composition via hierarchical object-diagrams

An aircraft consists of a variety of different systems, which represent the interacting disciplines involved in aircraft engineering (e.g. flight mechanics, aerodynamics, engine dynamics).

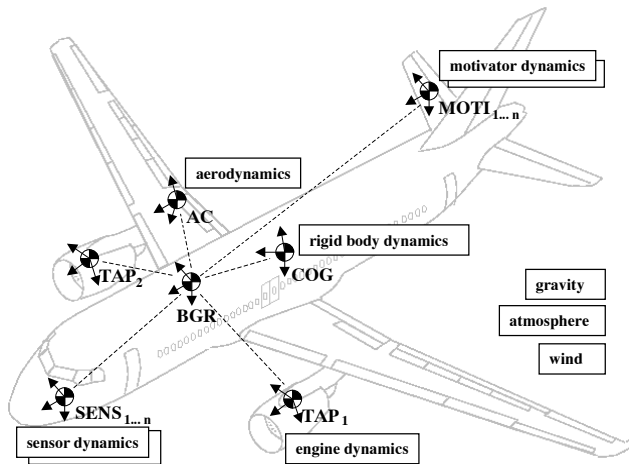


Figure 1: Domain specific reference points of flight dynamics and system models

Models of aircraft dynamics should be described in a notation close to the aircraft physics. The most natural way of modeling physical systems is as physical objects and phenomena, which are connected according to their physical energy flow interaction and kinematic constraints. This is different from modeling via signal flows or input-output block diagrams as traditionally used for controller modeling.

The 'local' description of each aircraft component (see Fig. 1) with respect to its intrinsic reference points (e.g., Center of Gravity COG, Body Geometric Reference BGR, Thrust Application Point TAP, Aerodynamic Center AC) in its domain specific coordinate system supports 'multi-point' modeling. The multi-point modeling approach allows, e.g., the proper handling of center of gravity variations and sensor positioning without any additional modeling effort, which is usually a very time-consuming and error-prone process.

A multi-point model also becomes necessary, when, e.g., the coupling effects between 'aircraft' and 'air flow' need to be modeled with higher accuracy than can be obtained by using a ordinary one-point model, where all the force, moment and velocity vectors are referred to the aircraft's center of gravity only [2].

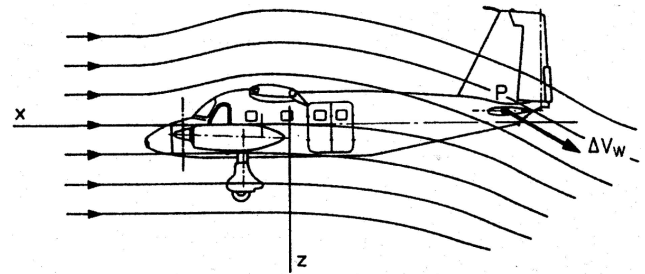


Figure 2: Local differential air velocity due to wing downwash and engine flow [2]

It is obvious from Fig. 2, that the local airflow is different for different points at the airframe due to aircraft rotation, changing wind fields, wing downwash and engine flow. The complete local airspeed \vec{V}_a for each point P can be calculated from the local inertial speed $\vec{V}(P)$ and the local speed of the airflow $\vec{V}_w(P)$:

$$\vec{V}_a(P) = \vec{V}(P) - \vec{V}_w(P).$$

Above equation can be expanded to identify all partial velocity vectors to give the complete airspeed at the point considered:

$$\begin{aligned} \vec{V}_a(P) = & \vec{V}(\text{COG}) - \vec{V}_w(\text{COG}) + \Delta\vec{V}(P, \vec{\omega}) \\ & - \Delta\vec{V}_w(P) - \Delta\vec{V}_{w_{dw}}(P). \end{aligned}$$

The total local airspeed is summed up by the inertial velocity of the center of gravity $\vec{V}(\text{COG})$, the speed of wind at COG $\vec{V}_w(\text{COG})$, the additional airspeed due to aircraft rotation ω at P about COG $\Delta\vec{V}(P)$, the effect of wind gradients due to its offset from COG $\Delta\vec{V}_w(P)$ and the effect of wing downwash and engine flow at P $\Delta\vec{V}_{w_{dw}}(P)$.

Using a 2-point-aerodynamics approach it is quite convenient to properly model the influence of aircraft rotation, wing downwash and wind gradients. According

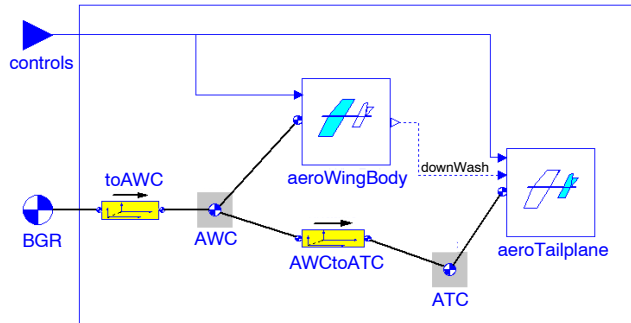


Figure 3: 2-point-aerodynamics object diagram with respect to the aircraft geometric reference BGR

to the object model of Fig. 3 this is done by separately describing the aerodynamics of wing/body (with respect to the Aerodynamic Wing Center AWC) and of the tailplane (with respect to the Aerodynamic Tail Center ATC). Between the aircraft body geometric reference BGR and these two aerodynamic reference points there are geometric offsets, which are explicitly made visible by the instances 'toAWC' and 'AWCtoATC' of a validated coordinate transformation class. The advantage of this approach is that the influence of aircraft rotation $\Delta \vec{V}(P)$ and the effect of wind gradients $\Delta \vec{V}_w(P)$ of above equation are automatically correctly handled by generic transformation objects.

In the same way, using the Flight Dynamics Library, all interactions between components of Fig. 1 can be properly formulated. In order to make the understanding of all submodels easy, each component of the library is described in its own coordinate system. Gravity, wind, and atmosphere are conveniently described in an earth related coordinate system, aerodynamics in a wind coordinate system, and engines in a system which is related to the body-fixed coordinate system. Therefore, in addition to the basic aircraft components, coordinate transformations are also detailed and handled as objects in the Flight Dynamics Library (see upper part of Fig. 4). Except for aerodynamics and engine objects all other objects are independent of a specific aircraft type.

The objects that constitute the rigid-body flight dynamics are interconnected according the object diagram of the bottom part of Fig. 4. Center point of the flight dynamics object model are the body geometric reference BGR and the center of gravity COG together with the body-object, which describes the mass properties and equations of motion of an aircraft. The

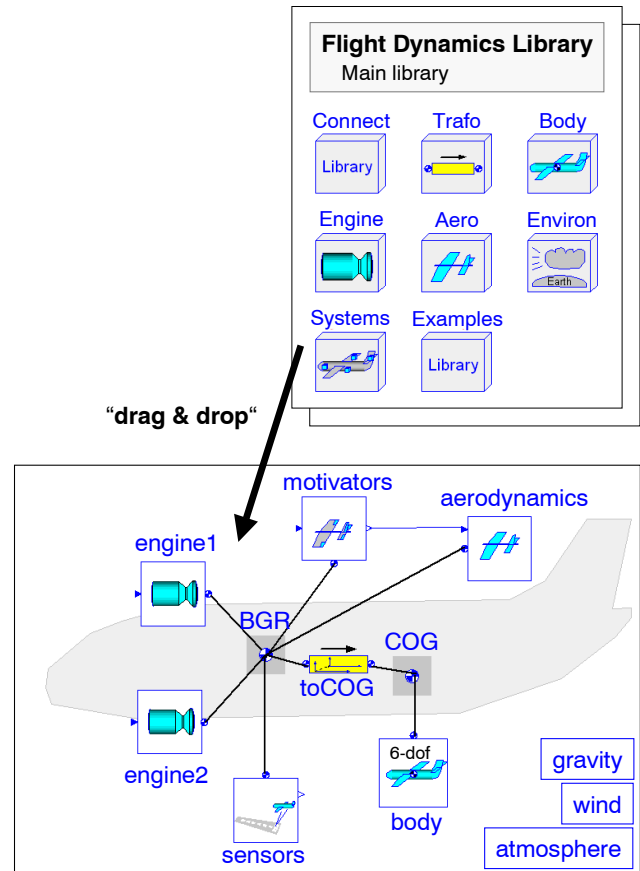


Figure 4: Interactive “drag & drop”-model building, top: flight dynamics class library, bottom: flight dynamics object-diagram

connections between objects represent their interaction. The complete aircraft consists of a **body** (fuselage and wing), which is powered by one or more **engines**. The **aerodynamics** describes the effects of the airflow over the aircraft. The aircraft is influenced by **gravity** and the surrounding **atmosphere** and winds. Additional dynamics is resulting from models of **motivators** and of **sensors**.

The connectors used to describe the interaction between flight dynamics objects, as specified by bold solid lines in the object diagram of Fig. 4, are the same as those used within the MODELICA-Multibody Library¹. The connector contains all variables which specifies the orientation, position and the corresponding speeds and accelerations with respect to some inertia. For aircraft usually some point at the earth's surface together with a 'north-east-down' coordinate system is defined as inertial reference. Additional connector variables are the force and moment vector, acting

¹URL: http://www.modelica.org/library/ModelicaAdditions/docu/ModelicaAdditions_MultiBody.html

at the origin of the point defined by the connector and solved in the coordinate system of the connector.

Specific for aircraft are the models of gravity, atmosphere and wind/gust. For multi-point models it is essential to properly formulate these models as fields, which usually vary with inertial position. For this purpose MODELICA offers the concept of 'dynamic scoping' [8]. Using this concept gravity fields, wind/gust fields and atmospheric data depending on the inertial position of individual aircraft components can be specified. Without any user effort the 2-point-aerodynamics of Fig. 3 is automatically handled correctly, because e.g. the wind field (and its gradients) are inherited position dependent to the aerodynamics models of wing/body and of the tailplane.

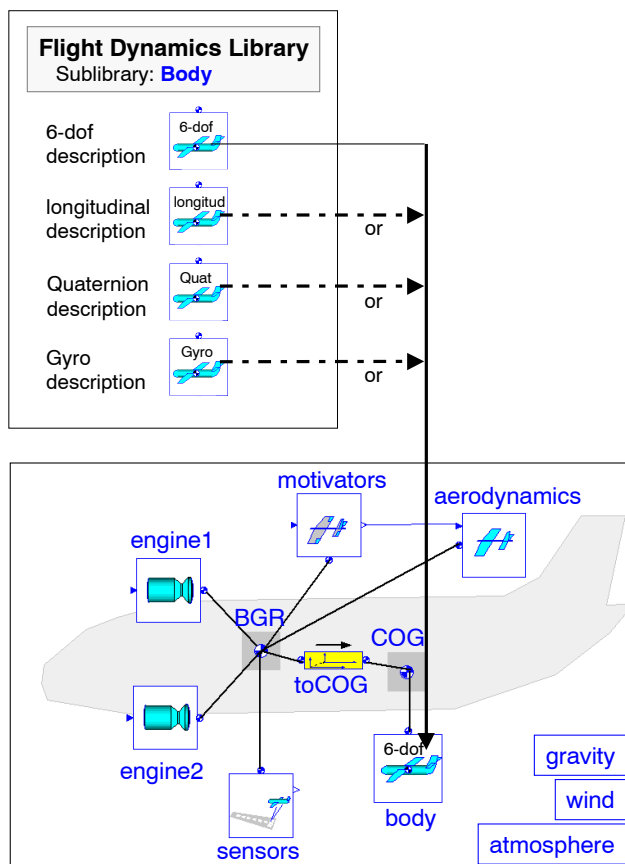


Figure 5: Local exchange of sub-components

In the Flight Dynamics Library different representations of one specific component can be found (see Fig. 5). There is a class with six degrees of freedom **body**, and another class with three degrees of freedom **bodyLong** that can be used to generate a nonlinear simulation model for the longitudinal motion only. The ordinary formulation of the equations of motion can be exchanged by a representation using quater-

nions. There are also wind, atmosphere and gravity models of different complexity. Existing codes for aerodynamics and engine models can easily be reused by using templates.

An important aspect of object-oriented modeling is the hierarchical structure and local encapsulation of objects. With the graphical user interface of DYMOLA, it can be zoomed into objects to display their internal structure. Zooming into the **engine**-object in the lower left part of Fig. 6 results in the engine model as detailed in the top left of this figure. It shows that the engine dynamics are described with respect to the thrust application point **TAP**. This is the most natural point at the airframe to formulate propulsion effects. The thrust application point is connected to the aircraft reference **BGR** via a transformation object **toTAP**, which details the offset in position and orientation from **TAP** with respect to **BGR**. Depending on this offset the generic transformation object is instantiated with the particular parameter values (top right of Fig. 6). As a result, the local engine forces and moments with respect to **BGR** are automatically computed.

Zooming into the **body**-object in the lower left part of Fig. 6 shows the equation layer of this component as displayed in the lower right part of this figure. Objects, which form the physical model, contain declarative mathematical equations, not assignments as is common in simulation languages. This makes the understanding and engineering reuse much easier as opposed to simulation code put in a form mainly for computational execution. A generic object with declarative equations can fulfill different application tasks. For example, the object **toCOG** which does the transformations between **COG** and **BGR** is used for transforming velocities with respect to **COG** to velocities with respect to **BGR**, as required as an interim step for aerodynamics and thrust calculations. The same object is used for the transformation of forces and moments from the **BGR** reference to the **COG**-reference, as required for solving the equations of motion within the **body**-object.

When connecting objects, only the relation between them is defined, and not the order in which the object equations are finally solved. Here the computer is used to sort the object equations automatically by a symbolic equation handler rather than performing this process manually. This aspect is handled in the following section.

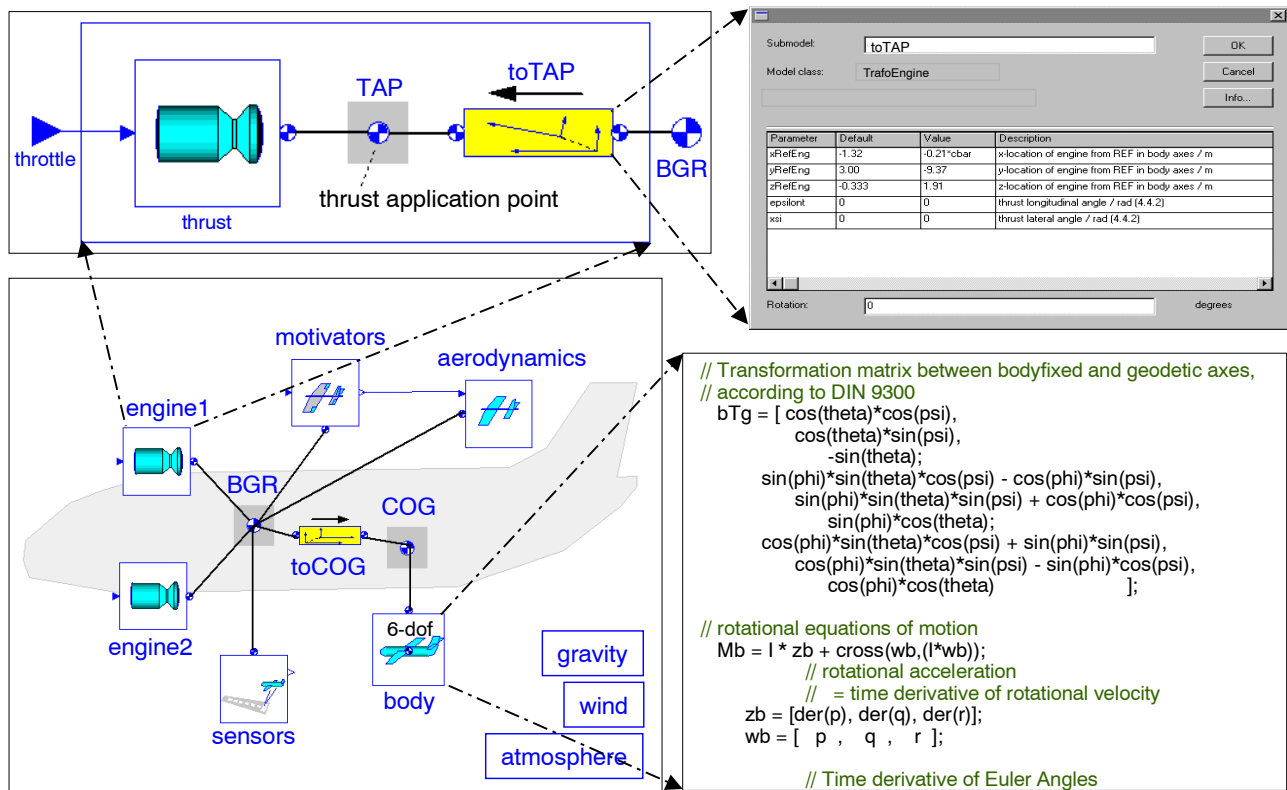


Figure 6: Zoom into an hierarchical structured object-diagram

3 Equation-based model building and efficient simulation code generation

From the model, that is graphically specified by a set of object diagrams, simulation models and documentation of the flight dynamics and systems can be generated automatically.

In the modeling process the object model is composed using different libraries and aircraft specific parameter data (see Fig. 7). The equation handler of DYMOLA solves the equations according to inputs and outputs of the complete aircraft model for a particular task. Equations, that are formulated in an object, but not needed for the specified configuration, are automatically removed in the following model building process. The result is a nonlinear symbolic state space description with a minimum number of equations for this task. Models for efficient parametric nonlinear simulations (section 3.1) can be automatically generated from object models of the Flight Dynamics Library, using the left branch of Fig. 7.

Due to MODELICA's equation-based approach it is possible to invert the interacting flight dynamics and flight systems model symbolically to the highest possible

extend. This allows to generate so called 'inverse models' which can be used for trim computations or which become nonlinear Dynamic Inversion (DI) control code [5] within a flight computer. The inversion according to the middle and right branch of Fig. 7 is mainly done by exchanging the external inputs and outputs while still using the same object model which has already been used to generate code for the simulation model. The equation handler of DYMOLA is used to solve all equations according to one of the above specified tasks. The derivation of an highly accurate and very efficient trim code is discussed in section 3.2, the DI-code generation is detailed in [5] and not presented here.

All generated models can be simulated with DYMOLA or with SIMULINK, using DYMOLA's S-function model generator. Additionally, automatic code generation is possible for the real-time engineering flight simulator AVDS (section 3.3).

3.1 Efficient parameterized simulation models

Generally a simulator requires that system models are transferred to a state space description:

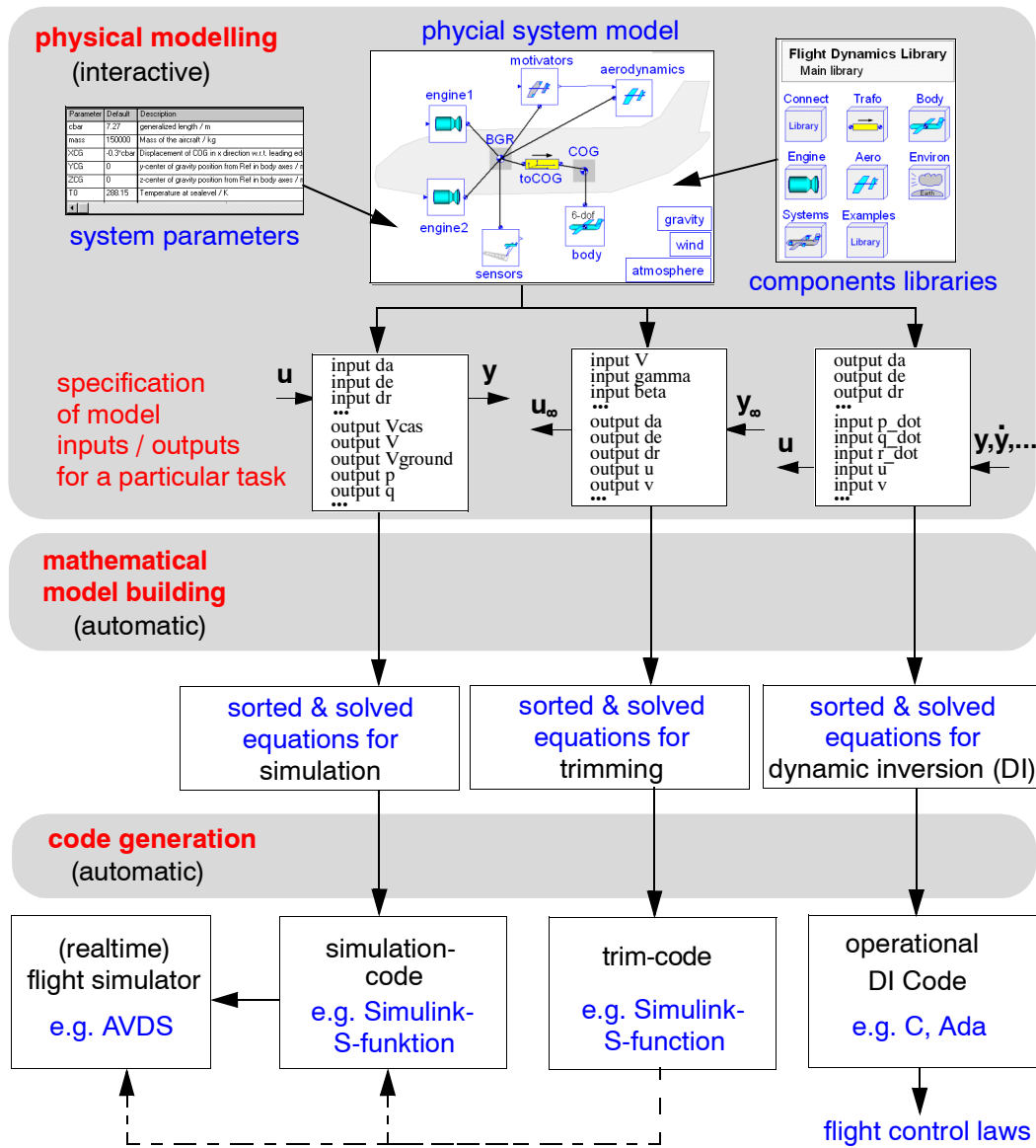


Figure 7: Equation based model building process

$$\begin{aligned}\dot{\mathbf{x}} &= \mathbf{f}(\mathbf{x}, \mathbf{u}_s, \mathbf{par}, t) \\ \mathbf{y}_s &= \mathbf{g}(\mathbf{x}, \mathbf{u}_s, \mathbf{par}, t).\end{aligned}$$

To achieve such a standard description from an object model it is only necessary to assign the aerodynamic and engine controls as simulation inputs \mathbf{u}_s and the measurement and evaluation signals as simulation outputs \mathbf{y}_s . For the generation of simulation models the state vector \mathbf{x} , which consists of the 12 basic flight dynamics states and of additional states of engine dynamics, actuator- and sensor dynamics, is automatically considered as known. Additional inputs are the simulation time t and the parameter vector \mathbf{par} (e.g. mass, center of gravity position, wing area). Depend-

ing on all inputs time derivative of the state vector $\dot{\mathbf{x}}$ and the output vector \mathbf{y}_s is computed. Fig. 8 defines a typical set of inputs, outputs, and states of a flight dynamics model. A typical set of flight dynamics states consist of the velocity (V), the angle of attack (α), the angle of sideslip (β), the angular velocities (p, q, r), the attitude (ϕ, θ, ψ) and the inertial position (x, y, z). Simulation model inputs are the aerodynamic control surface deflections of tailplane (dt), elevator (de), aileron (da) and rudder (dr), the engine controls ($throttle1, throttle2$) and, for example, additional gust inputs ($u_{gust}, v_{gust}, w_{gust}$). Typical simulation model outputs are the measurements signals such as *height*, V , α , β and the roll angle ϕ and the evaluation signal such as the flight path angle γ and the load

states	der(states)	control inputs	outputs
V	der(V) = 0	dt	height
α	der(α) = 0	de = const	V
β	der(β) = 0	da	$\gamma = 0$
p	der(p) = 0	dr	α
q	der(q) = 0	throttle1 = const	nz
r	der(r) = 0	throttle2 = const	β
ϕ	der(ϕ) = 0	u_gust = const	ϕ
θ	der(θ)	v_gust = const	
$\psi = 0$	der(ψ)	w_gust = const	
x = 0	der(x)		
y = 0	der(y)		
z	der(z)		

Figure 8: Inputs, outputs, and states of a flight dynamics model (actuator, sensor models are omitted here)

factor nz . For a detailed definition of these variables see, e.g. [1].

Automatic code generation, for example for SIMULINK, is possible separately for subcomponents as specified in Fig. 9 as well as for the complete aircraft model. The latter approach has the advantage that the transformation equations, which are in particular necessary for multi-point models, can be sorted (and eliminated) according the specified task. Algebraic loops, which occur if, e.g., the aerodynamic forces depend on accelerations, can be solved automatically using *tearing* [3].

Before starting a simulation, the initial, stationary inputs and states for a desired point of the flight envelope have to be calculated by a trim procedure. This aspect is dealt with in the following section.

3.2 Accurate trim computation

Trim calculations of complex flight system dynamics models are a very challenging computational task, involving the numerical solution of a system of nonlinear equations to calculate the stationary values of state and control variables. The difficulties mainly arise because of the lack of differentiability in aerodynamic and engine models due to the presence of various lookup tables used for linear interpolations. These severe nonlinearities as well as the presence of, e.g., control surface deflection limiters make the numerical solution of this high order system of equations very challenging.

Simulation environments, such as MATLAB/SIMULINK, offer trim routines for this task that use the simulation model to perform trim calculations, which are driven by a numerical optimization algorithm. For this purpose the state derivatives $\dot{\mathbf{x}}$ and outputs of the of the simulation model \mathbf{y}_s are set equal to their desired trim values $\dot{\mathbf{x}}_{tr}$ and \mathbf{y}_{tr} :

$$\begin{aligned}\dot{\mathbf{x}} &= \dot{\mathbf{x}}_{tr} \\ \mathbf{y}_s &= \mathbf{y}_{tr}\end{aligned}$$

The trim values of states \mathbf{x} and simulation inputs \mathbf{u}_s are calculated using the following constraint equation:

$$\begin{aligned}\dot{\mathbf{x}}_{tr} - \mathbf{f}(\mathbf{x}, \mathbf{u}_s, \mathbf{par}, t) &= 0 \\ \mathbf{y}_{tr} - \mathbf{g}(\mathbf{x}, \mathbf{u}_s, \mathbf{par}, t) &= 0\end{aligned}$$

The advantage of this numerical approach, using the complete simulation model, is that consistency between simulation model and trim computation is automatically guaranteed by using the same model. The disadvantage of this approach, which is achieved by a very high number of model evaluation of the simulation model, is its rather high calculation time and its comparatively low accuracy ('miss-trim'). The inaccuracy, which increases with the complexity and non-linearity of the model, results from this procedure neglecting the fact that some of the states directly depend on each other. For example, actuator and sensor states are treated as independent from the flight dynamics states and inputs even though they are directly related to them.

An alternative is to trim the subcomponents of the aircraft model (see Fig. 9) separately. First the flight dynamics submodel is trimmed. The trim values of this submodel are taken to trim in three additional steps the actuator, engine and sensor dynamics models, again using the same numerical trim approach. The trim computation is more accurate compared to the above one-step-approach, but usually more time consuming. Computation time and accuracy can be improved, if actuator and sensor model variables are not trimmed by an optimizer but directly set by the user. For example, actuator states and inputs can often be directly related to flight dynamics inputs and sensor states and outputs to flight dynamics outputs. The disadvantage of this approach is, that the procedure takes more engineer interaction and therefore increases the likeliness of errors, if submodels change during design.

A third option for trim computations, which is based on model inversion, is proposed here. The symbolic

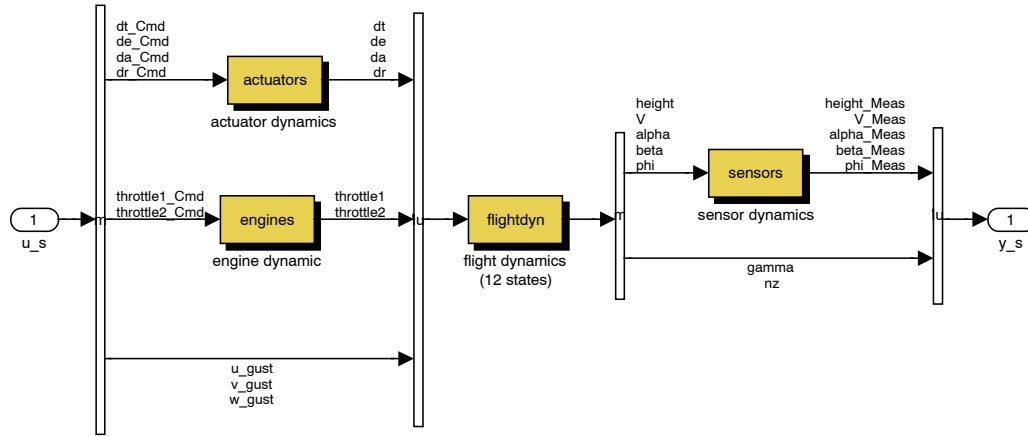


Figure 9: SIMULINK-block diagram of the flight system dynamics

engine of DYMOLA allows to generate C-code for an 'inverse model' to serve for trimming. To serve this purpose the inputs and outputs of the simulation model are inverted. The states derivatives $\dot{\mathbf{x}}$ become known, also the outputs of the simulation model \mathbf{y}_s , which are now the inputs of the trim model \mathbf{u}_t . The unknown variables are the the inputs of the simulation model $\mathbf{u}_s = \mathbf{y}_t$ and the states \mathbf{x} :

$$\begin{aligned} \mathbf{y}_t = \mathbf{u}_s &= \mathbf{h}(\mathbf{u}_t, \dot{\mathbf{x}}, \mathbf{par}, t) \\ \mathbf{x} &= \mathbf{j}(\mathbf{u}_t, \dot{\mathbf{x}}, \mathbf{par}, t) . \end{aligned}$$

One trim condition is specified in Fig. 8. In contrast to the simulation model code generation of section 3.1, where the variables are column-wise known or unknown, the known variables of the trim model (trim inputs) are shaded grey, whereas the unknown variables (trim outputs) are not. For each variable of the simulation model changed from known to unknown, one other variable is changed from unknown to known. The balance of known to unknown variables is kept equal. The inputs of the trim model are the desired trim conditions (such as velocity V and angle of attack α) and the outputs are the corresponding equilibrium values of trimmed state and aerodynamic and engine control vectors. DYMOLA generates essentially explicit equations for the inverse model by solving the high order nonlinear equation symbolically to the highest possible extend. Even if it is not possible to determine a symbolic solution, DYMOLA is still able to reduce the burden of solving numerically a high order system of nonlinear equations to the solution of a small core system of nonlinear equations which ultimately must be solved numerically.

The proposed trim approach based on model inversion was compared to the traditional approach in the HIRM

benchmark flight dynamics model of the GARTEUR Flight Mechanics Action Group on "New Analysis Techniques for Clearance of Flight Control Laws" [4]. An optimization based clearance process for flight control systems requires highly precise computations of trim values, because these values are the base for the following nonlinear or linear analysis. Even very small inaccuracies in trim values can corrupt the optimization progress. Here the trim computation based on an inverse model has proven its advantage compared to a standard optimization based trim approach. Another advantage of the inverse model trim approach is its computational time efficiency. Trimming the same, highly nonlinear flight dynamics model took, depending on the trim point, between 15 and 65 seconds using conventional trimming, just 50 to 70 milliseconds using the inverse model. Both trim computations were done within MATLAB/SIMULINK on a 400MHz personal computer [6].

3.3 Interactive Real-time Simulation using the Engineering Flight Simulator AVDS

For the evaluation of critical flight conditions and for the validation of flight control systems an aircraft animation tool can help the design engineer to analyze the aircraft performance. The Aircraft Visual Design Simulator (AVDS [7]) is such a tool to fill the gap between batch and motion-based simulation by allowing the flight control design engineer to quickly test and re-test response in a real-time environment on a low-cost PC (Fig. 10).

Fig. 11 illustrates the data flow within the interactive mode of AVDS, which allows the design engineer to virtually 'fly' the aircraft. Using, e.g. the cockpit-view together with the head-up display (HUD) of Fig. 10,

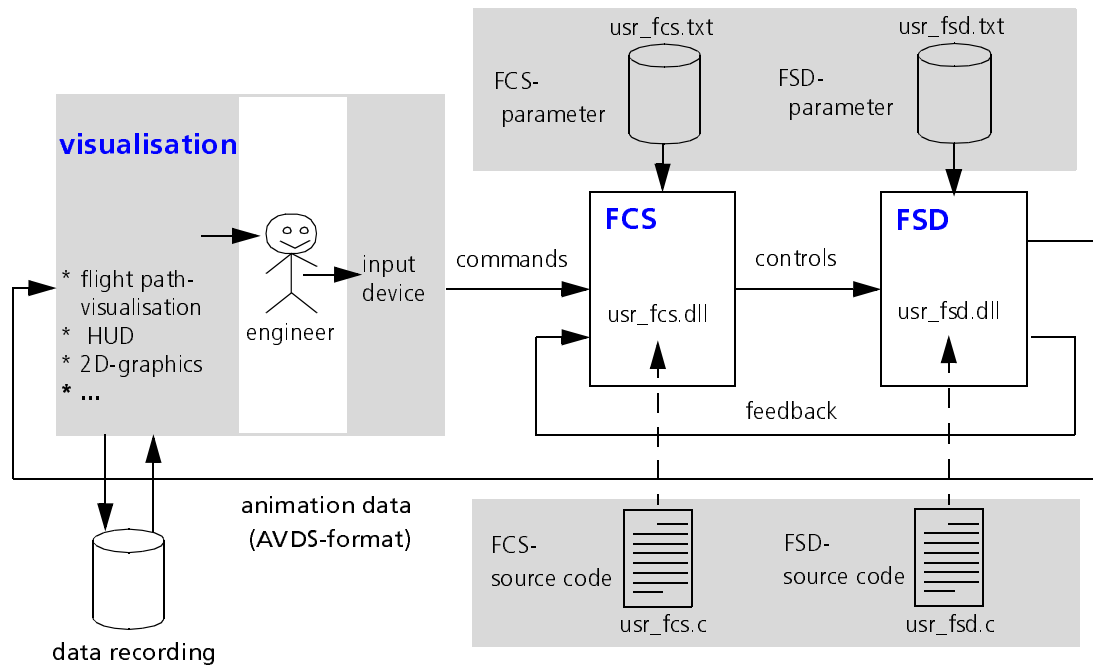


Figure 11: AVDS data flow in interactive simulation

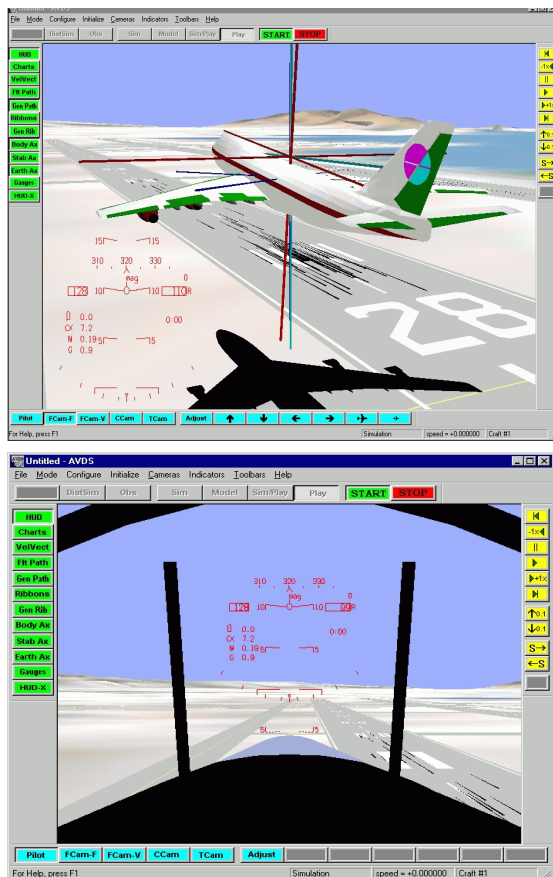


Figure 10: AVDS interactive real-time flight simulation

the engineer controls the aircraft via input devices like mouse, keyboards and other control instruments. These **commands** are transferred to the flight control system (FCS) and result in the **controls** of the flight system dynamics model (FSD). For the implementation of flight dynamics models AVDS offers an interface which consists of C-subroutines for controller and flight dynamics with systems with its corresponding set of parameters.

To avoid a manual re-implementation of these model codes, we propose to automatically generate the AVDS-codes and their parameter sets starting from the same object model as used for the parametric simulation (section 3.1). The big advantage using the MODELICA-AVDS interface is the complete automation of the code generation. The trim computation (as detailed in section 3.2) is already included in the code. This means that neither initial states of flight dynamics or systems have to be directly assigned by the user nor that an external, separate trim tool has to be used to specify flight conditions in AVDS.

The strategy of interfacing MODELICA flight dynamics models to AVDS can be transferred to any other flight simulator. The symbolic equation handler of DYMOLA guarantees a highly efficient model code. Different levels of detail do not have any influence on the interface structure. The only limit is the computational power of the platform which is used to run the flight simulator.

4 Conclusion

Complex aircraft models including actuator and sensor dynamics in addition to electronic flight control systems, are aggregated from contributions of many different disciplines involved. This paper shows that complex models are best comprehended if each disciplinary contribution is described in its own specific domain. For flight dynamics, the MODELICA Flight Dynamics Library serves this purpose.

For systematic and transparent modeling, it turned out to be important to describe all aircraft components and physical phenomena locally with respect to their intrinsic reference points, which usually have an offset in position and orientation from the aircraft's body geometric reference BGR.

The computer aided model building technique allows the modeling of engineering systems such as flight dynamics on a physical level in the form of declarative mathematical equations specifying energy exchange and kinematic constraints.

The equation-based modeling language of MODELICA allows the generation of codes for an *inverse model* to serve for trimming. Such a model has as inputs the desired trim conditions and as outputs the corresponding equilibrium values of trimmed state and controls vector. The equation handler of DYMOLA generates essentially explicit equations for the inverse model by solving the high order nonlinear equation to the highest possible extend symbolically. Thus, the trimming procedure based on such an inverse model has proven to be very accurate and fast compared to conventional optimization based trim procedure.

The code generation facility of DYMOLA allows the use of different simulators, (e.g., MATLAB/SIMULINK, DYMOLA's own simulation environment, the flight simulator AVDS) as a run-time environment for model execution. Using the Flight Dynamics Library offers the opportunity that trim code is automatically included into the simulation model and executed at simulation start. This means that no separate trim tool has to be used.

References

- [1] R. Brockhaus. *Flugregelung*. Springer Verlag, Berlin, 1994.
- [2] R. Brockhaus. A Mathematical Multi-Point Model for Aircraft Motion in Moving Air. *Zeitschrift für Flugwissenschaften und Weltraumforschung*, pages. 187-184, 1987.
- [3] H. Elmqvist und M. Otter. Methods for Tearing Systems of Equations in Object-Oriented Modeling. In *Proceedings ESM'94 European Simulation Multiconference*, pp. 326-332, Barcelona, Spain, 1994.
- [4] GARTEUR FM(AG11). Scope of a new GARTEUR Flight Mechanics Action Group on "New Analysis Techniques for Clearance of Flight Control Laws". Group for Aeronautical Research and Technology in Europe (GARTEUR), Technical Report GARTEUR TP-119-1, 1999.
- [5] G. Looye. Design of Autopilot Control Laws with Nonlinear Dynamic Inversion at *Automatisierungstechnik*, pp. 523-531, No. 12, 2001.
- [6] D. Moormann. *Automatisierte Modellbildung der Flugsystemdynamik*. Dissertation, RWTH Aachen. VDI Fortschrittsberichte, Mess-, Steuerungs- und Regelungstechnik, Reihe 8, No. 931, 2002.
- [7] S.J. Rasmussen and S.G. Breslin. AVDS: A Flight Systems Design Tool for Visualization and Engineer-in-the-Loop Simulation. *AIAA Modeling and Simulation Technologies Conference*, No. AIAA-3467-97, 1997.
- [8] M. Tiller. *Introduction to Physical Modeling with Modelica*. The Kluwer International Series in Engineering and Computer Sciences. ISBN 0792373677, 2001.

Modeling Friction in Modelica with the Lund-Grenoble Friction Model

Martin Aberger

Johannes Kepler University Linz,
Department for Design and Control of
Mechatronic Systems
A-4040 Linz, Austria
martin.aberger@students.jku.at

Martin Otter

DLR Oberpfaffenhofen,
Institute of Robotics and Mechatronics,
D-82230 Wessling, Germany
Martin.Otter@dlr.de

Abstract

The properties of the Lund-Grenoble friction model are summarized and different types of friction elements - bearing friction, clutch, one-way clutch, are implemented in Modelica using this friction formulation. The dynamic properties of these components are determined in simulations and compared with the friction models available in the Modelica standard library. This includes also an automatic gearbox model where 6 friction elements are coupled dynamically.

1 The rotational LuGre Model

1.1 Model Derivation

The LuGre (Lund-Grenoble) model [2] is a dynamic friction model with the relative angular velocity between the two surfaces in contact ω as input, and the friction torque τ as output. It approximates friction as a phenomenon caused by bristles in contact. The model can be seen as an extension of the simplified Dahl model. The LuGre model is described in standard form by a first-order nonlinear differential equation, see [6],

$$\frac{dz}{dt} = \omega - \frac{|\omega|}{g(\omega)} z \quad (1)$$

$$g(\omega) = \frac{1}{\sigma_0} \left(\tau_C + (\tau_S - \tau_C) e^{-(\omega/\omega_s)^2} \right) \quad (2)$$

$$\tau = \sigma_0 z + \sigma_1(\omega) \frac{dz}{dt} + \tau_v \omega \quad (3)$$

$$\sigma_1(\omega) = \sigma_1 e^{-(\omega/\omega_d)^2} \quad (4)$$

Where z denotes the average bristle deflection, σ_0 is the stiffness of the bristles, $\sigma_1(\omega)$ is the damping coefficient, ω_d describes the velocity interval around zero for which the damping is active, τ_v is the linear viscous friction coefficient, τ_C is the Coulomb friction level, τ_S is the static friction level, $(\tau_S \geq \tau_C)$, and ω_s is the Stribeck velocity. The function $g(\omega)$ defines how the average deflection depends on the relative velocity between the two surfaces. The simplified form of the LuGre model is given by a constant damping coefficient

$$\sigma_1(\omega) = \sigma_1 \quad (5)$$

With a constant damping coefficient and $(\tau_S > \tau_C)$ the model is dissipative, see [1], if and only if

$$\sigma_1 \leq \frac{\tau_C \tau_v}{\tau_S - \tau_C} \quad (6)$$

The velocity dependent damping coefficient $\sigma_1(\omega)$ was not implemented in the Modelica model because no identified or measured parameters for ω_d have been found in the literature. It is already rather difficult to identify the dynamic parameters σ_0 and σ_1 , which is also stated in [5]. With an increasing number of parameters the complexity of the identification process rises.

For steady-state motion $dz/dt = 0$ the average bristle deflection is given by, see (1)

$$z_{ss} = \text{sgn}(\omega) g(\omega) \quad (7)$$

Hence the relation between angular velocity and friction torque for steady state motion is

$$\begin{aligned} \tau_{ss} &= \sigma_0 \text{sgn}(\omega) g(\omega) + \tau_v \omega \\ &= \left(\tau_C + (\tau_S - \tau_C) e^{-(\omega/\omega_s)^2} \right) \text{sgn}(\omega) + \tau_v \omega \end{aligned} \quad (8)$$

If the angular velocity is not zero when integration starts, the initial value of z , see (1), shall be computed such that $dz/dt = 0$ for $t = 0$ in order to avoid (non-physical) peaks in the friction torque at the start of the simulation. Simulation experiments with stick-slip motion show that integration methods with variable step-size may have difficulties to compute the break away torque in certain cases, especially if the relative tolerance is not set strictly enough. To improve the reliability and accuracy of the simulation, an auxiliary Boolean equation is introduced

$$zs_{neg} = \begin{cases} \text{true} & \text{if } \frac{dz}{dt} < 0 \\ \text{false} & \text{otherwise} \end{cases} \quad (9)$$

that triggers a state event when dz/dt changes sign (in Modelica, every value change of a relation triggers an

event). Additionally, it is useful to scale the average bristle deflection z , as it is small compared to other state variables. Both effects are discussed in more detail below. A reasonable choice for the scaling parameter z_N is

$$z_N \approx \frac{1}{\sigma_0}. \quad (10)$$

The parameters of the LuGre friction model depend mostly on the application, especially the friction torque levels τ_c , τ_s , and τ_v may vary widely. Reasonable choices for the other parameters are given in [6]

$$\begin{aligned} \omega_s &= 0.01 \\ \sigma_0 &= 10^3 - 10^5 \\ \sigma_1 &= 2\sqrt{\sigma_0 J} \end{aligned}$$

where J is the inertia of the body subject to friction. The time for integration depends heavily on the choice of the dynamic parameters σ_0 and σ_1 because they determine essentially the stiffness of the differential equation.

1.2 Dynamic Model Behaviour

To verify the dynamic behaviour of the simplified LuGre model, the same simulations as in [2] are performed using Dymola as simulation engine [4] and a Modelica coded LuGre model. The *final* results agree qualitatively with the results in [2]. Stick-slip motion is a typical behaviour of systems with friction. It is caused by the fact that friction is larger at rest than during start of sliding.

The experiment is shown in Figure 1. An inertia with $J=1 \text{ kgm}^2$ with friction to the ground is connected to a spring with stiffness $k=2 \text{ Nm/rad}$. The end of the spring is rotating with a small constant velocity of $\omega=0.1 \text{ rad/s}$. The inertia is originally at rest and the torque from the spring increases linearly. The friction torque counteracts the spring torque, and a small displacement follows. When the applied torque reaches the break away torque, in this case approximately $g(0)\sigma_0$, the inertia starts to rotate and the friction decreases rapidly due to the Stribeck effect. The spring contracts, and the spring torque decreases. The inertia slows down and the friction torque increases due to the Stribeck effect and the rotation stops. The phenomenon repeats itself. The parameters of the friction model are shown in Table 1. Unfortunately, the passivity inequality (6) is not satisfied with these data.

Simulation of a direct implementation of the LuGre model using the integration algorithm DASSL with a relative tolerance¹ $Tol=10^{-4}$ leads to wrong results: The break away torque is too high and there are non-physical oscillations in the computed friction torque. This result is understandable, because the step-size

control of variable step integrators treat variables as zero, when they are below a predefined absolute tolerance. If no other information is available, this tolerance is usually selected as a multiple of the relative tolerance. Since z is in the order of 10^{-5} , the step-size control on z is practically switched off most of the time.

As to be expected, the simulation result is improved when a scaling for state variable z is introduced: The oscillations of the friction torque disappear while the inertia is rotating. The friction torque of the model with and without scaling using integration algorithm DASSL and a relative tolerance of $Tol=10^{-4}$ is shown in Figure 2.

In Figure 5, the scaled derivative of the bristle deflection, dz/dt , is present. As can be seen, very sharp changes of this variable appear when changing from the sliding to the stuck region and vice versa. An integrator has to detect this sharp change to compute a correct solution. In order to give the integrator a hint to this situation, a state event is triggered in the Modelica model, whenever dz/dt changes sign, by introducing the mentioned auxiliary Boolean equation. This technique improves the quality of the LuGre model simulation further, although still a considerable difference in the behavior of the friction model is present when simulating with different tolerances. In Figure 3 the friction torque is shown for relative tolerances $Tol=10^{-4}$ and $Tol=10^{-6}$. The difference between these two simulations is caused by different break away torques.

The break away torque is related to the dwell-time and the rate of increase of the applied torque. The dwell-time is the time between sticking and break away. Since the LuGre model is a dynamic model, a varying break away torque can be expected. The simulated break away torque for a relative tolerance $Tol=10^{-4}$ is $\tau_B \approx 1.5 \text{ Nm}$, for a tolerance $Tol=10^{-6}$ it is $\tau_B \approx 1.48 \text{ Nm}$. This result is also achieved with integration algorithms with fixed step-size.

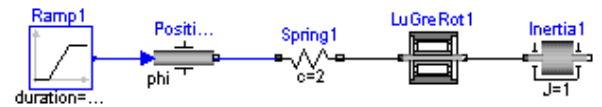


Figure 1: Simulation setup for stick-slip motion with the LuGre friction model.

σ_0	10^5	[Nm/rad]
σ_1	316.23	[Nms/rad]
τ_v	0.4	[Nms/rad]
τ_c	1	[Nm]
τ_s	1.5	[Nm]
ω_s	0.001	[rad/s]

Table 1: Parameter values of the LuGre friction model.

¹ In Dymosim the absolute and relative tolerance of the state vector are equal [4].

In Figure 4 the angular velocity, the rotation angle of the inertia and the rotation angle of the spring are shown. The scaled average bristle deflection z and the derivative of the bristle deflection are shown in Figure 5. The scaling factor is $z_n=10^5$. These results were obtained with the integration algorithm DASSL and a tolerance $Tol=10^{-6}$. They agree qualitatively with the results in [2].

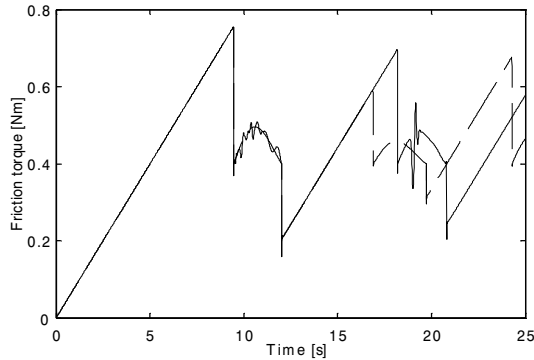


Figure 2: Friction torque of the LuGre model without scaling (solid line) and with scaling (dashed line).

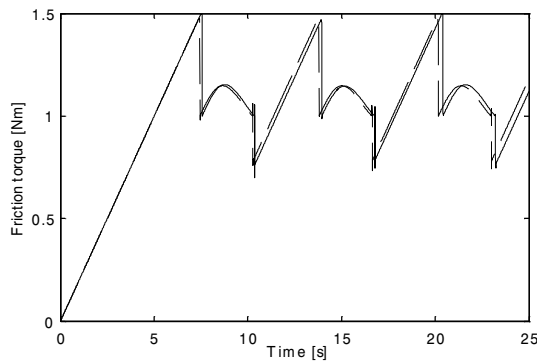


Figure 3: Friction torque of the LuGre model with DASSL and tolerance $Tol=10^{-4}$ (solid line) and $Tol=10^{-6}$ (dashed line).

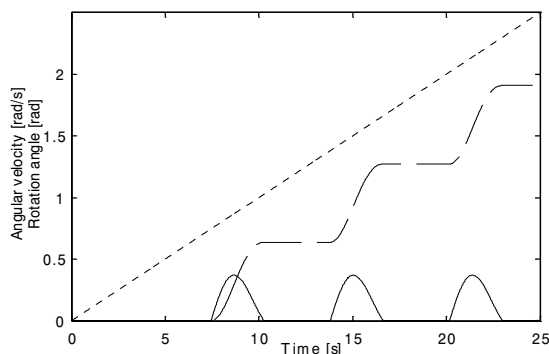


Figure 4: Angular velocity (solid line), rotation angle of the inertia (dashed line) and rotation angle of the spring (dotted line).

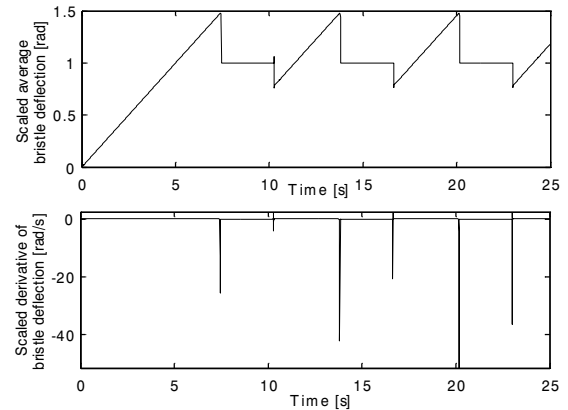


Figure 5: Scaled average bristle deflection (top) and scaled derivative of average bristle deflection (bottom).

1.3 Comparison of LuGre and Modelica BearingFriction Model

In the Modelica standard library, friction element BearingFriction is present which constrains the relative angular acceleration $d\omega/dt$ to zero, when the relative speed ω vanishes, i.e., the relative movement is forced to stay at rest. This is in contrast to the more detailed LuGre model which describes also the small relative movements in the stuck region.

The model of Figure 1 was simulated by replacing the LuGre model with component BearingFriction from library Modelica.Mechanics.Rotational. The parameters of the LuGre model are shown in Table 1. The Stribeck velocity is rather small which corresponds to a sharp decrease in the friction torque when the inertia starts to rotate. Therefore a simplified parameterization for the component BearingFriction is used. The Coulomb friction is set to $\tau_c=1$ Nm and the stiction torque is set to $\tau_s=1.5$ Nm. Therefore the parameter setting in the BearingFriction model is: $\tau_{pos}=[0, 1; 1, 1.4]$ and $peak=1.5$.

To achieve accurate results for simulations with the LuGre model, a relative tolerance of $Tol=10^{-6}$ is needed. The number of output intervals was set to $N=2500$ for a simulation time of $T_s=25$ s. For all simulations this parameter setup was kept constant. The model was built up with Dymola version 4.1d, 2001-07-11. To get similar integration times for two simulations the executable Dymosim file was executed in the DOS window. The simulations were performed on a 2xPentium III with 600 MHz and 768 MB RAM.

The model was compiled with the Microsoft Visual C++ compiler (version 4.1). Compiling the model with the GNU compiler (version egcs 2.91.66) and the standard option setting in Dymola requires a very small integration step size of $T_i=50$ μ s for the LuGre model when using the integration algorithm RKF45 (= Runge-Kutta method of order 4 with fixed step size), to obtain a converging result. A much larger step size is

possible by the GNU compiler options -O3 (max. optimization) or -O0 (no optimization). The very small integration step size is needed for compiler options -O1 (Dymola default), -O2, and -Os. The reason for this strange behaviour could not be determined.

Using the Microsoft Visual C++ compiler with Dymola defaults, a converging result with the LuGre model and integrator RKFIX4 requires a maximum integrator step size of $T_i=1$ ms and for the explicit Euler method with fixed step size (EULER) $T_i=0.9$ ms. To achieve a converging result for mixed mode integration, a maximum integrator step size of $T_i=2$ ms is required. Mixed mode integration is a special Dymola technique for real-time simulation where fast states are discretized with the implicit and slow states with the explicit Euler method. State variable z from the LuGre model was used as “fastState”. Simulation statistics are summarized in Table 2.

For the model with BearingFriction, the maximum integrator step size is $T_i=7$ ms for RKFIX4 and $T_i=4$ ms for EULER, respectively. Variations in the maximum possible integration step size for various GNU compiler options was not observed. For the comparison, the same step sizes as in the LuGre model were used for the integrators with fixed step size. The simulation statistics are summarized in Table 3.

The difference in the CPU-time for integration of the LuGre model and the BearingFriction model is not significant due to the simple model. For more complex models, the CPU-time for the variable step-size integrator DASSL is related to the number of function and Jacobian evaluations which are about 5 to 6 times higher for the LuGre model as with the BearingFriction model. The differences in the number of Jacobian evaluations and in the minimum step sizes can be traced back to the stiffness of the differential equations of the LuGre model.

A comparison of the friction torque of the BearingFriction model and of the LuGre model is shown in Figure 6. The difference of these two simulated friction torques is a result of different break away torques of the models. For the LuGre model the break away torque is varying and it is approximately $\tau_B \approx 1.48$ Nm, for the BearingFriction model the break away torque is constantly $\tau_B = 1.5$ Nm. The peaks in the simulated friction torque when the inertia stops do not appear in the BearingFriction model because of the simplified parameterization and neglect of the Stribeck effect.

Integration algorithm	DASSL	RKFIX4	EULER	MIXED MODE
CPU-time for integration [s]	0.909	1.23	1.09	0.875
CPU-time for one GRID interval [ms]	0.363	0.494	0.436	0.360
No. of result points	2587	2537	2586	2531
No. of steps	1732	25000	30000	12500
No. of F-evaluations	5626	100000	30000	12500
No. of H-evaluations	4962	25020	30044	12518
No. of Jacobian-evaluations	595	-	-	-
No. of state events	43	19	43	17
Min. integration step-size [s]	$1.58 \cdot 10^{-7}$	10^{-3}	$0.1 \cdot 10^{-3}$	$2 \cdot 10^{-3}$
Max. integration step-size [s]	1.19	10^{-3}	$0.9 \cdot 10^{-3}$	$2 \cdot 10^{-3}$
Max. integration order	5	4	1	1

Table 2: Simulation statistics of the LuGre model.

Integration algorithm	DASSL	RKFIX4	EULER
CPU-time for integration [s]	0.847	1.24	1.22
CPU-time for one GRID interval [ms]	0.339	0.498	0.489
No. of result points	2519	2517	2519
No. of steps	303	25000	30000
No. of F-evaluations	905	100000	30000
No. of H-evaluations	2864	25010	30010
No. of Jacobian-evaluations	120	-	-
No. of state events	9	9	9
Min. integration step-size [s]	$1.79 \cdot 10^{-6}$	10^{-3}	$0.1 \cdot 10^{-3}$
Max. integration step-size [s]	4.1	10^{-3}	$0.9 \cdot 10^{-3}$
Max. integration order	5	4	1

Table 3: Simulation statistics of the BearingFriction model.

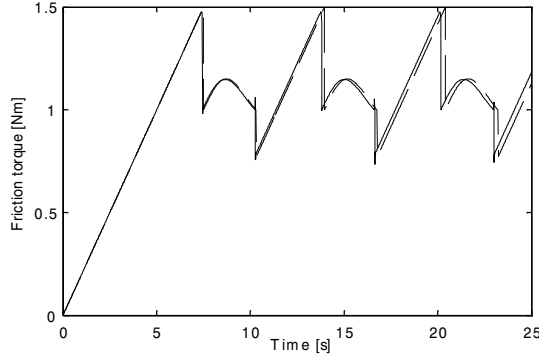


Figure 6: Friction torque of the LuGre model (solid line) and of the BearingFriction model (dashed line).

2 LuGre Clutch Models

2.1 Model Derivation

2.1.1 Clutch with LuGre Friction

Friction torque τ in clutches is usually described as a function of the friction coefficient $\mu(\omega_r)$, which is in turn a function of the relative angular velocity ω_r , of the normal force F_n , and of a geometry constant c_{geo} which takes into account the geometry of the device and the assumptions on the friction distributions:

$$\tau = \mu(\omega_r) c_{geo} F_n \quad (11)$$

The LuGre model [2] can be adapted to such a clutch description:

$$\frac{dz}{dt} = \omega_r - \frac{|\omega_r|}{g(\omega_r)} z \quad (12)$$

$$g(\omega_r) = \frac{1}{\sigma_0} \left(\mu_C + (\mu_S - \mu_C) e^{-(\omega_r / \omega_s)^2} \right) \quad (13)$$

$$\tau = \left(\sigma_0 z + \sigma_1 \frac{dz}{dt} + \sigma_2 \omega_r \right) c_{geo} F_n \quad (14)$$

where μ_C is the Coulomb friction coefficient and μ_S is the static friction coefficient. This model is related to the lumped dynamic tire model in [3].

In the Modelica model, the normal force F_n is provided as input signal u in a normalized form,

$$F_n = F_{n \max} \cdot u \quad (15)$$

where the maximum normal force $F_{n \max}$ is provided as parameter.

If the relative angular velocity does not vanish at simulation start, the initial value of z , see (12), should be computed such that $dz/dt = 0$ for $t = 0$ to avoid peaks in the friction torque at the start of the simulation. Again, a Boolean auxiliary equation is introduced to trigger an event at sharp changes of z , i.e., when dz/dt changes sign. As the average bristle deflection is small

compared to other state variables, also a scaling is introduced for z .

2.1.2 One-Way-Clutch with LuGre Friction

A one-way-clutch is an element where a clutch is connected in parallel to a free wheel. This special element is needed to resolve the ambiguity of the friction torque which would be present if a free wheel would be explicitly connected in parallel to a clutch component. If the clutch is locked, the friction torque is computed by

$$\tau = c_{\max} \left(\sigma_0 z + \sigma_1 \frac{dz}{dt} + \sigma_2 \omega_r \right) F_{n \max} \quad (16)$$

where c_{\max} has to be provided as parameter. With this parameter the maximum friction torque is defined, when the clutch is locked, a reasonable choice is

$$c_{\max} = 100.$$

The clutch is locked when the average bristle deflection z is negative. All other equations are similar to the clutch shown in the section above.

2.2 Comparison of Clutch Models

The behaviour of the LuGre models for clutches is compared to the clutch models of the Modelica standard library.

2.2.1 Simplified automatic Gearbox

A simplified model of a 3-gear automatic gearbox is simulated. The model with components from the standard Modelica rotational library is shown in Figure 7. The parameters of the clutches and brakes are shown in Table 4 and the gear shift table in Table 5. A switching sequence from first to third gear within $T_S = 4$ s is simulated using $N = 4000$ output intervals and integrator DASSL with a relative tolerance $Tol = 10^{-6}$.

The same model with the LuGre clutch is shown in Figure 8. Brakes are replaced by series connection of a LuGre clutch with a fixed flange. The parameters of the LuGre clutches are shown in Table 6. The model with LuGre clutches is unstable when using integration algorithms with a fixed step-size, even if the step-size is very small.

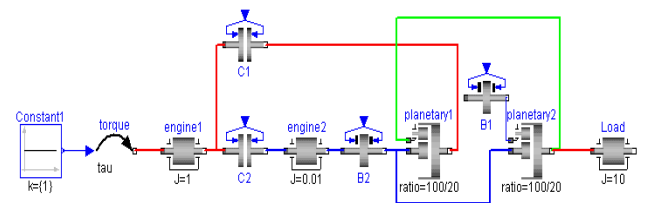


Figure 7: 3-gear automatic gearbox model.

mue_pos	[0, 0.5]	-
peak	1	-
cgeo	1	-
fn_max	10	[N]

Table 4: Parameters of the standard clutches and brakes.

gear	C1	C2	B1	B2
0				
1	on		on	
2	on			on
3	on	on		

Table 5: Gear shift table of clutches and brakes.

σ_0	10^5	[m/rad]
σ_1	300	[ms/rad]
σ_2	0	[ms/rad]
μ_C	0.5	[Nm/N]
μ_S	0.5	[Nm/N]
c_{geo}	1	-
$F_{n\ max}$	10	[N]
ω_S	0.001	[rad/s]

Table 6: Parameters of the LuGre clutches.

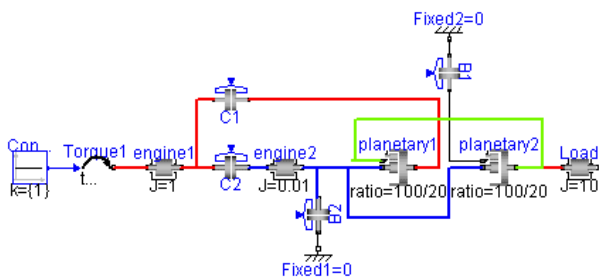


Figure 8: 3-gear automatic gearbox model with LuGre clutches.

The two different implementations of the clutches show a similar behaviour. The angular velocity of the load with standard clutches is shown in Figure 9, in comparison to the model with the LuGre clutch the switching from one gear into the next is slightly slower. The friction torques of the standard and the LuGre clutches are shown in Figure 10. There is no difference in the friction torque for clutch C2. For the LuGre clutch C1 peaks appear at the switching points. The friction torques of the brakes and the LuGre clutches are shown in Figure 11. The friction torque of the LuGre clutch B1 shows peaks at the switching points and when the clutch gets stuck. The friction torque of the LuGre clutch B2 shows some small oscillations after the switching point. With (11) and the actual parameterization for the standard clutches the maximum friction torque of the clutch is $\tau_{max}=5$ Nm. This is not always fulfilled with the LuGre clutches.

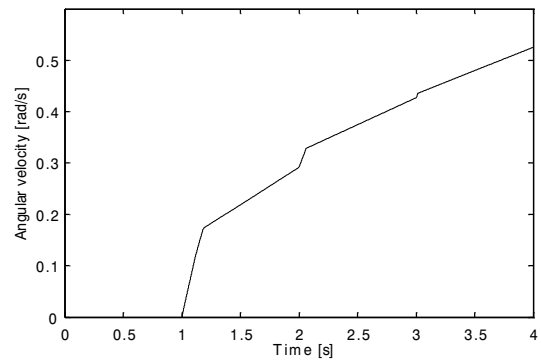


Figure 9: Angular velocity of load.

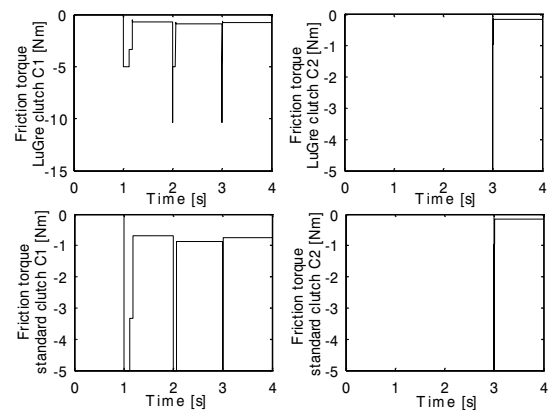


Figure 10: Friction torques of LuGre and standard clutches.

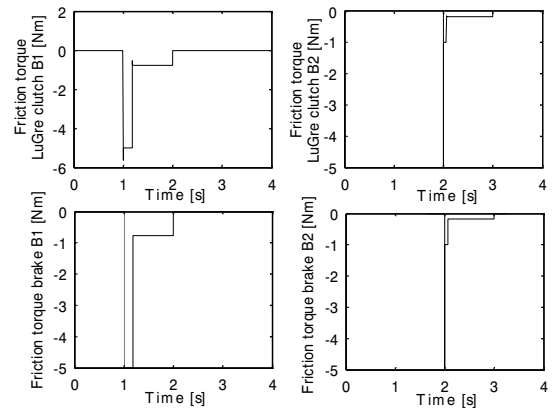


Figure 11: Friction torques of LuGre clutches and brakes.

The effect on the friction torque of a decreased stiffness of the bristles of $\sigma_0=10^4$ is shown in Figure 12 (the damping of the bristles is kept constant at $\sigma_1=300$). The amplitude of the peaks of the LuGre clutch C1 is bigger. The peak when clutch B1 gets stuck is smaller. The comparison of the angular velocity of the load with the standard model shows that the time for switching is decreased.

The effect on the friction torque of a decreased damping of the bristles $\sigma_I=30$ is shown in Figure 13 (the stiffness is kept constant $\sigma_\theta=10^5$). The height of the peaks of the LuGre clutch C1 is smaller when the clutch is activated. However, the amplitudes of the peaks when the clutches get stuck are increased and there are small oscillations. The comparison of the angular velocity of the load with the standard model shows that the time for switching is almost identical, but there are small oscillations in the angular velocity after the switching.

The angular acceleration of the load with standard clutches and brakes and with LuGre clutches with different dynamic parameters is shown in Figure 14. The maximum acceleration with LuGre clutches is higher than with standard clutches and brakes. Increasing the stiffness of the bristles in the LuGre clutches results in higher peaks at the switching points. When the damping of the bristles is reduced, oscillations occur and the acceleration is negative.

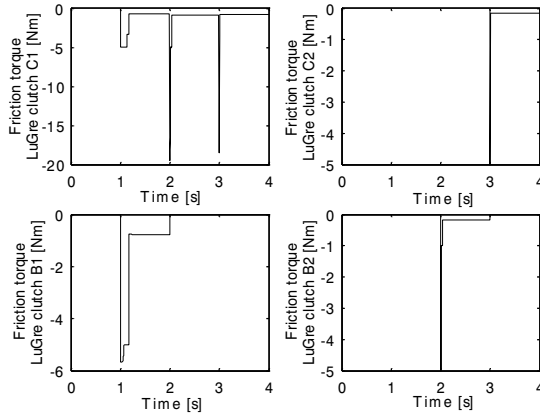


Figure 12: Friction torque of LuGre clutches C1, C2, B1, and B2 with dynamic friction parameters $\sigma_\theta=10^4$ and $\sigma_I=300$.

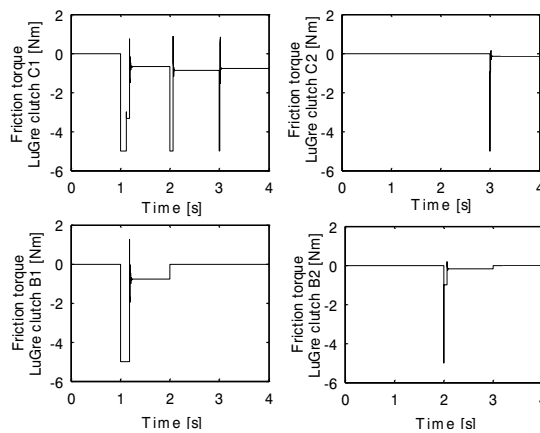


Figure 13: Friction torque of LuGre clutches C1, C2, B1, and B2 with dynamic friction parameters $\sigma_\theta=10^5$ and $\sigma_I=30$.

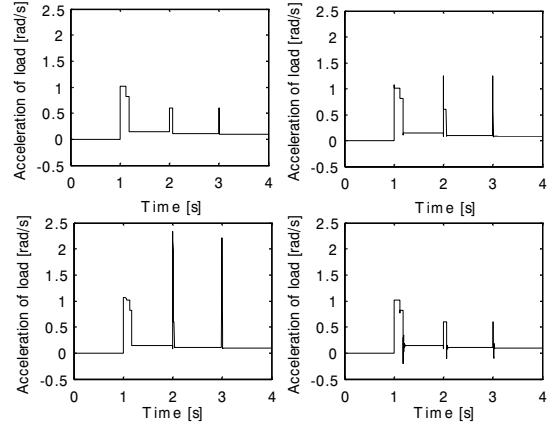


Figure 14: Acceleration of load with standard clutches and brakes (top left), LuGre clutches and dynamic parameters $\sigma_\theta=10^5$ and $\sigma_I=300$ (top right), dynamic parameters $\sigma_\theta=10^4$ and $\sigma_I=300$ (bottom left), and dynamic parameters $\sigma_\theta=10^5$ and $\sigma_I=30$ (bottom right).

2.2.2 Car Model with Automatic Gear

A complete model of a car power train with an automatic gearbox was simulated. The model of the automatic gearbox is shown in Figure 15, for details see [7]. All clutches are based on the LuGre friction model. The parameters of the LuGre clutches are shown in Table 7. The maximum normal force is not shown, because it is different for each clutch. For the simulation of $T_S=200$ s, integrator DASSL was used with a relative tolerance $Tol=10^{-4}$ and $N=1000$ output intervals. The model with LuGre clutches is unstable when using integration algorithms with a fixed step-size, e.g. RKFIX4, even with very small step sizes. The model is also unstable with integration algorithm DASSL when the stiffness of the bristles is reduced to $\sigma_\theta=10^4$, and the damping of the bristles is in the range $0.003 \leq \sigma_I \leq 3$. When the stiffness of the bristles is increased, the simulation time also increases, as to be expected.

The *velocity* of the car with standard clutches and with LuGre clutches is shown in Figure 16. There is hardly any difference in the velocity, except at the gear shift at $t \approx 122$ s, where the velocity is decreasing with the LuGre clutches (which is qualitatively wrong) in contrast to the model with standard clutches where the velocity is not decreasing. The *acceleration* of the car with standard clutches and with LuGre clutches is shown in Figure 17. There are small differences between the two curves, especially at $t \approx 122$ s where the velocity of the car is decreasing. The maximum acceleration with the LuGre clutches is $a_{max} \approx 10^5$ m/s², a completely unrealistic value. The amplitude of these peaks depends on the dynamic parameters. When the damping of the bristles is increased or the stiffness of the bristles is decreased the amplitude of the peaks increases.

σ_0	10^5	[m/rad]
σ_1	0.03	[ms/rad]
σ_2	0	[ms/rad]
μ_C	0.12	[Nm/N]
μ_S	0.144	[Nm/N]
c_{geo}	1	-
ω_S	0.5	[rad/s]

Table 7: Parameters of the LuGre clutches.

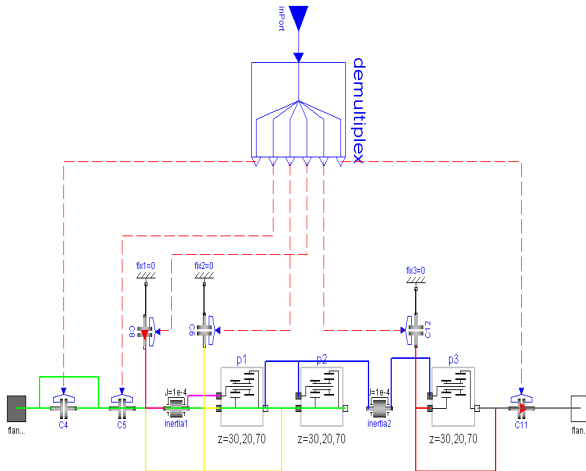


Figure 15: Automatic gearbox model.

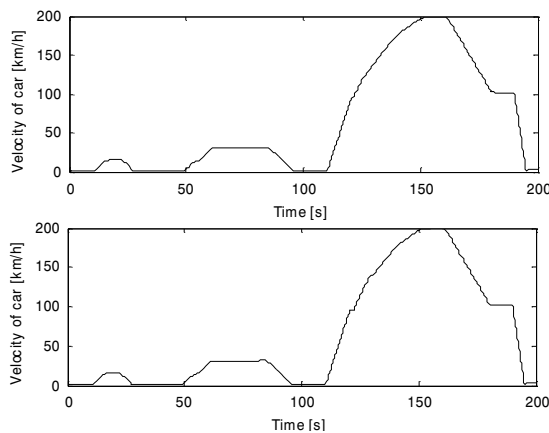


Figure 16: Velocity of car with standard clutches (top) and LuGre clutches (bottom).

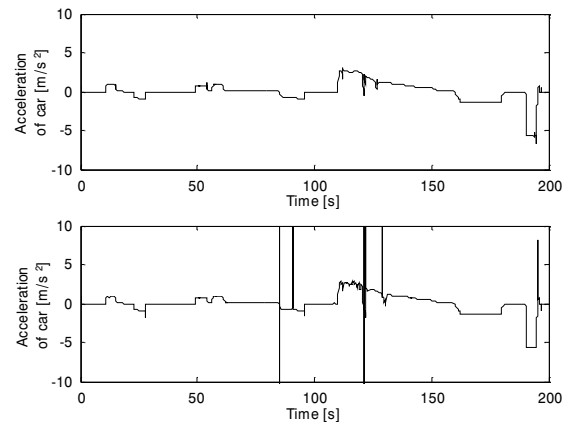


Figure 17: Acceleration of car with standard clutches (top) and LuGre clutches (bottom).

Conclusions

Modelica models for bearing friction, clutches and one-way clutches have been implemented based on the recently developed new LuGre friction model [2]. This model has been used successfully for controller design to compensate friction effects [6]. The friction model seems to be well suited for this purpose, because the (stiff) differential equation description allows an easier application of standard theory.

In this article it was investigated whether the LuGre friction model has also advantages when used in simulations. Especially, the potential for real-time simulation of the shift dynamics of automatic gearboxes was evaluated. It turns out that reasonable results can be achieved if one friction element is contained in a model. As to be expected, the simulation time is higher, since a stiff differential equation is solved. A potential advantage seemed to be that no events occur in the model because handling of state events is always problematic in real-time simulation. However, it seems to be that state events (or something similar) have to be artificially introduced to detect the sharp changes in z , in order to arrive at reliable simulations.

When friction elements are dynamically coupled, as it is the case in automatic gearbox models, LuGre based clutch models seem to be not suited: Fixed step size integrators, as needed for real-time simulation, could not be applied in the two test cases, because it was not possible to get stability (even for very small step sizes). Using the usually very robust and reliable variable step-size integrator DASSL, resulted in simulations which are quite sensitive on the choice of the dynamic LuGre friction parameters, and even led to instabilities in certain parameter ranges.

Bibliography

- [1] N. Barabanov and R. Ortega, “Necessary and Sufficient Conditions for Passivity of the LuGre Friction Model”, *IEEE Transactions on Automatic Control*, Vol. 45, 2000.
- [2] C. Canudas de Wit, H. Olsson, K. J. Åström and P. Lischinski, “A New Model for Control of Systems with Friction”, *IEEE Transactions on Automatic Control*, Vol. 40, 1995.
- [3] C. Canudas de Wit, P. Tsiotras, “Dynamic Tire Friction Models for Vehicle Traction Control”, *Conference on Decision and Control*, Phoenix, Arizona, Dec. 1999.
- [4] Dymola, *Dynasim AB*, Lund, Sweden, version 4.1, <http://www.dynasim.se>.
- [5] M. Gäfvert, “Comparison of two Friction Models”, *Master thesis, Lund Institute of Technology*, University of Lund, 1996.
- [6] H. Olsson, “Control of Systems with Friction”, *Phd. thesis, Lund Institute of Technology*, University of Lund, 1996.
- [7] M. Otter, C. Schlegel and H. Elmqvist, “Modeling and Realtime Simulation of an Automatic Gearbox using Modelica”, *Proc. ESS’97 European Simulation Symposium*, Passau, Germany, pp. 115-121, 1997.

Appendix

Simplified rotational LuGre Model

```

model LuGreRot "LuGre bearing friction"
  import R=Modelica.Mechanics.Rotational;
  import SI = Modelica.SIunits;
  extends R.Interfaces.Rigid;

  parameter SI.Torque tau_s=1.5;
  parameter SI.Torque tau_c=1;
  parameter Real tau_v=0.4 "Nms/rad";
  parameter SI.AngularVelocity ws=0.001;
  parameter Real sigma0=1e5 "Nm/rad";
  parameter Real sigma1=316.22 "Nms/rad";
  parameter Real zN=1.e-5 "Nom. value";

  SI.AngularVelocity w "Abs. speed";
  SI.AngularAcceleration a "dw/dt";
  SI.Angle z(start=0) "Bristle defl.";
  SI.AngularVelocity zs(start=0) "dz/dt";
  SI.Torque tau "Friction torque";

  Real g "see [2]";
  SI.Angle zStart "Start value of z";
  Boolean zsneg "Trigger events";

equation
  // Initial conditions
  when initial() then
    zStart = sign(w)*g/zN;
    reinit(z, zStart);
  end when;

```

```

// Speeds and accelerations
w = der(phi);
a = der(w);

// Deflection of bristles
zs = der(z);
zs = (w/zN - abs(w)*z/g);

// Friction torque
g = 1/sigma0*(tau_c + (tau_s - tau_c)*
  exp(-(w/ws)^2));
tau = sigma0*z*zN+sigma1*zs*zN+tau_v*w;

// Equilibrium of torques
flange_a.tau + flange_b.tau - tau = 0;

// Trigger events when dz/dt=0
zsneg = zs < 0;
end LuGreRot;

```

Clutch with LuGre Friction

```

model ClutchLuGre "LuGre Clutch model"
  import R=Modelica.Mechanics.Rotational;
  import SI = Modelica.SIunits;
  import B = Modelica.Blocks.Interfaces;
  extends R.Interfaces.Compliant;

  parameter Real mue_s=1.5;
  parameter Real mue_c=1;
  parameter SI.AngularVelocity ws=0.001;
  parameter Real sigma2=0.1;
  parameter Real sigma0=1e5;
  parameter Real sigma1=316.22;
  parameter Real sigma2=0.1;
  parameter Real zN=1.e-5 "Nom. value";
  parameter Real cgeo = 1 "Geom. const.";
  parameter SI.Force fn_max = 1;

  B.InPort inPort(final n=1);

  SI.AngularVelocity w_rel "Rel. speed";
  SI.AngularAcceleration a_rel;
  SI.Angle z(start=0) "Bristle defl.";
  SI.AngularVelocity zs(start=0) "dz/dt";
  SI.Torque tau "Friction torque";
  SI.Force fn "Normal force(=fn_max*u)";

  Real g "see [2]";
  SI.Angle zStart "Start value of z";
  Boolean zsneg "Trigger events";
  Boolean free;
  Real u "normalized force input [0..1]";

equation
  // Initial conditions
  when initial() then
    zStart = sign(w_rel)*g/zN;
    reinit(z, zStart);
  end when;

  // Relative quantities
  w_rel = der(phi_rel);
  a_rel = der(w_rel);

```

```

// Normal force and frict. for fn <= 0
u = inPort.signal[1];
free = u <= 0;
fn = if free then 0 else fn_max*u;

// Deflection of bristles
zs = der(z);
zs = if free then 0 else
  (w_rel/zN - abs(w_rel)*z/g);
g = 1/sigma0*(mue_c + (mue_s - mue_c)*
  exp(-(w_rel/ws)^2));
// Friction torque
tau = if free then 0 else cgeo*fn*
  (sigma0*z*zN + sigma1*zs*zN +
  sigma2*w_rel);

// Trigger events when dz/dt=0
zsneg = zs < 0;
end ClutchLuGre;

```

One-Way-Clutch with LuGre Friction

```

model OneWayClutchLuGre
  "Freewheel and clutch"
  import R=Modelica.Mechanics.Rotational;
  import SI = Modelica.SIunits;
  import B = Modelica.Blocks.Interfaces;
  extends R.Interfaces.Compliant;

  parameter Real mue_s=1.5;
  parameter Real mue_c=1;
  parameter SI.AngularVelocity ws=0.001;
  parameter Real sigma2=0.1;
  parameter Real sigma0=1e5;
  parameter Real sigma1=316.22;
  parameter Real sigma2=0.1;
  parameter Real zN=1.e-5 "Nom. value";
  parameter Real cgeo = 1 "Geo. const.";
  parameter SI.Force fn_max = 1;

  B.InPort inPort(final n=1);

  SI.AngularVelocity w_rel "Rel. speed";
  SI.AngularAcceleration a_rel;
  SI.Angle z(start=0) "Bristle defl.";
  SI.AngularVelocity zs(start=0) "dz/dt";
  SI.Torque tau "Friction torque";
  SI.Force fn "Normal force(fn_max*u)";

  Real g "see [2]";
  SI.Angle zStart "Start value of z";
  Boolean zsneg "Trigger events";
  Boolean free;
  Boolean locked;
  Real u "normalized force input [0..1]";
protected
  constant Real cmax=100;
equation
  // Initial conditions
  when initial() then
    zStart = sign(w_rel)*g/zN;
    reinit(z, zStart);
  end when;

  // Relative quantities
  w_rel = der(phi_rel);
  a_rel = der(w_rel);

```

```

// Normal force and frict. for fn <= 0
u = inPort.signal[1];
free = u <= 0;
fn = if free then 0 else fn_max*u;
locked = z < 0;

// Deflection of bristles
zs = der(z);
zs = (w_rel/zN - abs(w_rel)*z/g);

// Friction torque
g = 1/sigma0*(mue_c + (mue_s - mue_c)*
  exp(-(w_rel/ws)^2));
tau = if locked then
  fn_max*cmax*(sigma0*z*zN
  + sigma1*zs*zN + sigma2*w_rel)
else (if free then 0 else
  cgeo*fn*
  (sigma0*z*zN + sigma1*zs*zN +
  sigma2*w_rel));
end OneWayClutchLuGre;

```

Session 9b

Special Methods and Tools

The Open Source Modelica Project

Peter Fritzson, Peter Aronsson, Peter Bunus, Vadim Engelson, Levon Saldamli,
Henrik Johansson¹, Andreas Karström¹

PELAB, Programming Environment Laboratory, Department of Computer and Information
Science, Linköping University, SE-581 83, Linköping, Sweden
{petbu,petfr}@ida.liu.se

¹MathCore AB
Wallenbergs gata 4
SE-583 35 Linköping, Sweden
{henrik,andreas}@mathcore.com

Abstract

The open source software movement has received enormous attention in recent years. It is often characterized as a fundamentally new way to develop software. This paper describes an effort to develop an open source Modelica environment to a large extent based on a formal specification of Modelica, coordinated by PELAB, Department of Computer and Information Science, Linköping University, Sweden. The current version of the system provides an efficient interactive computational environment for most of the expression, algorithm, and function parts of the Modelica language as well as an almost complete static semantics for Modelica 2.0.

The longer-term goal is to provide reasonable simulation execution support, at least for less complex models, also for the equation part of Modelica which is the real essence of the language. People are invited to contribute to this open source project, e.g. to provide implementations of numerical algorithms as Modelica functions, add-on tools to the environment, or contributions to compiler itself. The source code of the tool components of the open source Modelica environment is available under the Gnu Public License, GPL. The library components are available under the same conditions as the standard Modelica library. The system currently runs under Microsoft Windows, Linux, and Sun Sparc Solaris. A benchmark example of running a simplex algorithm shows that the performance of the current system is close to the performance of handwritten C code for the same algorithm.

1 Introduction and Project Goals

The open source Modelica effort described in this paper has both short-term and long-term goals:

- The short-term goal is to develop an efficient interactive computational environment for most of the expression, algorithm, and function parts of the Modelica language, as well as a complete formal

static semantic specification of the language. It turns out that with support of appropriate tools and libraries, Modelica is very well suited as a computational language for development and execution of both low level and high level numerical algorithms, e.g. for control system design, solving nonlinear equation systems, or to develop optimization algorithms that are applied to complex applications.

- The long-term goal is to have a rather complete implementation of the Modelica language, including simulation of equation based models and additional facilities in the programming environment, as well as convenient facilities for research and experimentation in language design or other research activities. However, our goal is not to reach the level of performance and quality provided by current commercial Modelica environments that can handle large models requiring advanced analysis and optimization by the Modelica compiler.

The long-term *research* related goals and issues of the open source implementation of a Modelica environment include but are not limited to the following:

- Development of a *complete formal specification* of Modelica, including both static and dynamic semantics. Such a specification can be used to assist current and future Modelica implementers by providing a semantic reference, as a kind of reference implementation.
- *Language design*, e.g. to further *extend the scope* of the language, e.g. for use in diagnosis, structural analysis, system identification, etc., as well as modeling problems that require partial differential equations.
- *Language design* to *improve abstract properties* such as expressiveness, orthogonality, declarativity, reuse, configurability, architectural properties, etc.
- *Improved implementation techniques*, e.g. to enhance the performance of compiled Modelica code by generating code for parallel hardware.

- *Improved debugging* support for equation based languages such as Modelica, to make them even easier to use.
- *Easy-to-use* specialized high-level (graphical) *user interfaces* for certain application domains.
- *Visualization* and animation techniques for interpretation and presentation of results.

The complete formal specification for Modelica is developed in *Natural Semantics*, which is currently the most popular and widely used semantics specification formalism. This specification is used as input for automatic generation of Modelica translator implementations being part of the open source Modelica environment, using the RML compiler generation tool developed at PELAB. The availability of a formal specification facilitates language design research on new language constructs to widen the scope of the language, as well as improving its abstract properties.

The open source Modelica environment thus provides a test bench for language design ideas that, if successful, can be submitted to the Modelica Association for consideration regarding possible inclusion in the official Modelica standard.

The current version of the open source Modelica environment allows most of the expression, algorithm, and function parts of Modelica to be executed interactively, and Modelica functions to be compiled into efficient C code. The generated C code is combined with a library of utility functions and a run-time library. An external function library interfacing a LAPACK subset and other basic algorithms is also under development.

2 The Open Source Modelica Environment

The interactive open source Modelica environment currently consists of the following components:

- *An interactive session handler*, that parses and interprets commands and Modelica expressions for evaluation. The session handler also contains simple history facilities, and completion of file names and certain identifiers in commands. It is based on the ReadLine library which is available in most Linux and Unix distributions.
- *A Modelica compiler*, translating Modelica to C code, with a symbol table containing definitions of classes, functions, and variables. Such definitions can be predefined, user-defined, or obtained from libraries.
- *An execution and run-time module*. This module currently executes compiled binary code from translated expressions and functions. In the future it will also support simulation of equation based models, requiring numerical solvers as well as event handling facilities for the discrete and hybrid parts of the Modelica language.
- *A textual model editor*. Any text editor can be used. We have so far primarily employed Gnu Emacs,

which has the advantage of being programmable for future extensions. A Gnu Emacs mode for Modelica and a Modelica syntax highlighting for the UltraEdit text editor has previously been developed. There is also a special Modelica editor that recognizes the syntax to some extent, and marks keywords and comments using different colors [Modelica]. Both the Emacs mode and the special editor hides Modelica graphical annotations during editing, which otherwise clutters the code and makes it hard to read.

- *A graphical model editor*. This is a graphical connection editor, for component based model design by connecting instances of Modelica classes. This part of the system is not yet implemented. A Java based prototype is however under development.

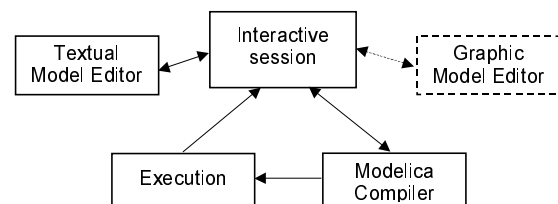


Figure 1. The architecture of the open source Modelica environment. Arrows denote data and control flow. The interactive session handler receives commands and shows results from evaluating expressions that are translated and executed. The graphic model editor is not yet implemented.

2.1 Interactive Session with Examples

The following is an interactive session with the open source Modelica environment including some commands and examples. First we start the system, which responds with a header line:

```
Open Source Modelica 0.1
```

We enter an assignment of a vector expression, created by the range construction expression 1:12, to be stored in the variable x. The type and the value of the expression is returned.

```
>> x := 1:12
Integer[12]: {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}
```

The function `bubblesort` is called to sort this vector in descending order. The sorted result is returned together with its type. Note that the result vector is of type `Real[:]`, instantiated as `Real[12]`, since this is the declared type of the function result. The input `Integer` vector was automatically converted to a `Real` vector according to the Modelica type coercion rules.

```
>> bubblesort(x)
```

```
Real[12]: {12, 11, 10, 9, 8, 7, 6, 5,
4, 3, 2, 1}
```

Now we want to try another small application, a simplex algorithm for optimization. First read in a small matrix containing coefficients that define a simplex problem to be solved:

```
>> a := read("simplex_in.txt")
Real[6, 9]:
{{-1,-1,-1, 0, 0, 0, 0, 0, 0},
{-1, 1, 0, 1, 0, 0, 0, 0, 5},
{ 1, 4, 0, 0, 1, 0, 0, 0, 45},
{ 2, 1, 0, 0, 0, 1, 0, 0, 27},
{ 3,-4, 0, 0, 0, 0, 1, 0, 24},
{ 0, 0, 1, 0, 0, 0, 0, 1, 4}}
```

Then call the simplex algorithm implemented as the Modelica function `simplex1`. This function returns four results, which are represented as a tuple of four return values:

```
>> simplex1(a)
Tuple: 4
Real[8]: {9, 9, 4, 5, 0, 0, 33, 0}
Real: 22
Integer: 9
Integer: 3
```

It is possible to compute an expression, e.g. `12:-1:1`, and store the result in a file using the `write` command:

```
>> write(12:-1:1,"test.dat")
```

We can read back the stored result from the file into a variable `y`:

```
>> y := read("test.dat")
Integer[12]: {12, 11, 10, 9, 8, 7, 6,
5, 4, 3, 2, 1}
```

It is also possible to give operating system commands via the `system` utility function. A command is provided as a string argument. The example below shows `system` applied to the UNIX command `cat`, which here outputs the contents of the file `bubblesort.mo` to the output stream.

```
>> system("cat bubblesort.mo")
function bubblesort
  input Real[:] x;
  output Real[size(x,1)] y;
protected
  Real t;
algorithm
  y := x;
  for i in 1:size(x,1) loop
    for j in 1:size(x,1) loop
      if y[i] > y[j] then
        t := y[i];
        y[i] := y[j];
        y[j] := t;
      end if;
    end for;
  end for;
end bubblesort;
```

Another built-in command is `cd`, the *change current directory* command. The resulting current directory is returned as a string.

```
>> cd(".")
String: "/home/petfr/modelica"
```

3 The Modelica Translation Process

The Modelica translation process is depicted in Figure 2 below. The Modelica source code is first translated to a so-called flat model. This phase includes type checking, performing all object-oriented operations such as inheritance, modifications etc., and fixing package inclusion and lookup as well as import statements. The flat model includes a set of equations declarations and functions, with all object-oriented structure removed apart from dot notation within names. This process is a *partial instantiation* of the model, called *elaboration* in subsequent sections.

The next two phases, the equation analyzer and equation optimizer, are necessary for compiling models containing equations. These phases are currently missing, but will be supplied in some future version of the open source Modelica system. Finally, C code is generated which is fed through a C compiler to produce executable code. In the current preliminary version of the system this code cannot be any equation based simulation code, only code for computing expressions, algorithms, and functions.

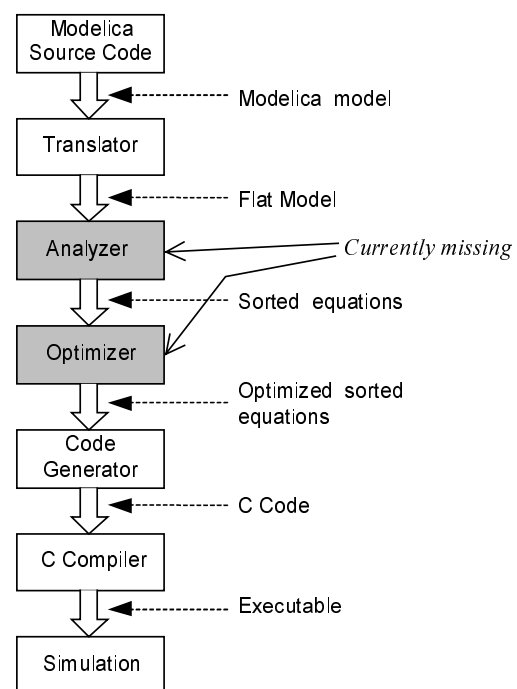


Figure 2. Translation stages from Modelica code to executing simulation. The current version of the open source Modelica compiler generates executable code only for functions and expressions since the equation analyzer and optimizer are still missing.

4 Modelica Static and Dynamic Semantics

The complete semantics, i.e. meaning, of the constructs in any programming language, also including the Modelica language, is usually divided into two major parts:

- static semantics
- dynamic semantics

The *static semantics* specifies compile time issues such as type checking, equivalence between different representations of the program such as a form with inheritance operations present, and the corresponding form with inheritance and modifications expanded, elaboration of the object-oriented model representation, conversion between different levels of the intermediate form, etc. Such a static semantics is currently given by our formal specification of Modelica.

It should be noted that the semantics specification for an equation based language such as Modelica differs from semantics specifications of ordinary programming languages, since Modelica equation based models are not programs in the usual sense. Instead Modelica is a modeling language used to specify relations between different objects in the modeled system.

The *dynamic semantics* specifies the run-time behavior including the equation solving process during the simulation, execution of algorithmic code, and additional run-time aspects of the execution process. We do not currently have a formal specification of the Modelica dynamic semantics, but intend to develop such a specification in the future. The current dynamic run-time behavior is implemented by hand in the C programming language

4.1 An Example Translation of Models into Equations.

As an example, the Modelica model B below is translated into equations.

```
model A
  Real x,y;
equation
  x = 2 * y;
end A;
```

```
model B
  A a;
  Real x[10];
equation
  x[5] = a.y;
end B;
```

The resulting equations appear as follows:

```
B.a.x = 2 * B.a.y
B.x[5] = B.a.y
```

The object-oriented structure is mostly lost which is why we talk about a flat set of equations, but the variable names in the equations still give a hint about their origin.

What the translator actually does is translating from the Modelica source code to an internal representation of the resulting set of equations. This internal representation is then further translated to C code. For functions, declarations, and interactively entered expressions, the C code can be generated fairly directly, whereas for equations several optimization stages must be applied before generating the C code.

4.2 Model Parameterization and Structural Parameters

In many cases model parameters in a Modelica model are unproblematic, and the translator can simply emit a flat model with the parameters still available as parameters that can be given different values during equation solution. But in some cases this is not so. We differentiate between two types of parameters, structural parameters, which affect the number and contents of the equations, and value parameters, that do not.

One simple example of a structural parameter is when a parameter is used in the size of an array. Consider the following model:

```
model ArrayEx
  parameter Real N = 3;
  Real a[N];
equation
  a[1] = 1;
  for i in 2:N loop
    a[i] = a[i-1] * 2;
  end for;
end ArrayEx;
```

This will, with the parameter N unmodified, produce the following equations:

```
a[1] = 1
a[2] = 2
a[3] = 4
```

However, if the parameter N is modified to be 5 when the model is elaborated, i.e. symbolically expanded, the set of equations will be different:

```
a[1] = 1
a[2] = 2
a[3] = 4
a[4] = 8
a[5] = 16
```

As the semantic specification specifies the semantics in terms of the generated equations, regarding algorithms and functions as special cases of equations, this means that the values of the parameters need to be determined at compile time to make it possible for the translator to do the translation.

Another, even more serious, complication with parameters is the combined use of parameters and connect statements. If a model contains e.g. a connect statements that looks like `connect (a[N], c)`, where a is an array of connectors, and N is a parameter, then the generated equations may look very different depending

on the value of N . Not only the number of equations may change, but the equations themselves can be altered.

5 Automatic Generation and Formal Specification of Translators

The implementation of compilers and interpreters for non-trivial languages is a complex and error prone process, if done by hand. Therefore, formalisms and generator tools have been developed that allow automatic generation of compilers and interpreters from formal specifications. This offers two major advantages:

- *High level descriptions* of language properties, rather than detailed programming of the translation process
- *High degree of correctness* of generated implementations. The high level specifications are more concise and easier to read than a detailed implementation in some programming language

The *syntax* of a programming language is conveniently described by a *grammar* in a common format, such as BNF, and the structure of the lexical entities, usually called *tokens*, are easily described by *regular expressions*. However, when it comes to the semantics of a programming language, things are not so straightforward.

The *semantics* of the language describes what any particular program written in the language “means”, i.e. what should happen when the program is executed, or evaluated in some form.

Many programming language semantic definitions are given by a standard text written in English which tries to give a complete and unambiguous specification of the language. Unfortunately, this is often not enough. Natural language is inherently not very exact, and it is hard to make sure that all special cases are covered. One of the most important requirements of a language specification is that two different language implementers should be able to read the specification and make implementations that interpret the specification in the same way. This implies that the specification must be exact and unambiguous.

For this reason formalisms for writing formal specifications of languages have been invented (Pagan 1981). These formalisms allow for a mathematical semantic description, which as a consequence is exact, unambiguous and easier to verify. One of the most widely used formalisms for specifying semantics is called Natural Semantics (Kahn, 1987). A computer processable Natural Semantic specification language called RML, for Relational Meta Language, with a compiler generation tool *rml2c* (Pettersson, 1995), has previously been developed at PELAB, and is used for the formal specification of Modelica in the open source Modelica project.

5.1 Compiler Generation

Writing a good compiler is not easy. This makes automatic compiler generation from language specifications very interesting. If a language specification is written in a formal manner, all the information needed to build the compiler is already available in a machine-understandable form. Writing the formal semantics for a language can even be regarded as the high-level programming of a compiler.

In Figure 3 below, the different phases of the compilation process are shown. More specifically, it shows the translation process of a Modelica translator, which compiles, or rather translates, the Modelica source file. All the parts of the compiler can be specified in a formal manner, using different formalisms such as *regular expressions*, *BNF grammars*, and *Natural Semantics* specifications. At all stages, there is a tool to convert the formalism into an executable compiler module.

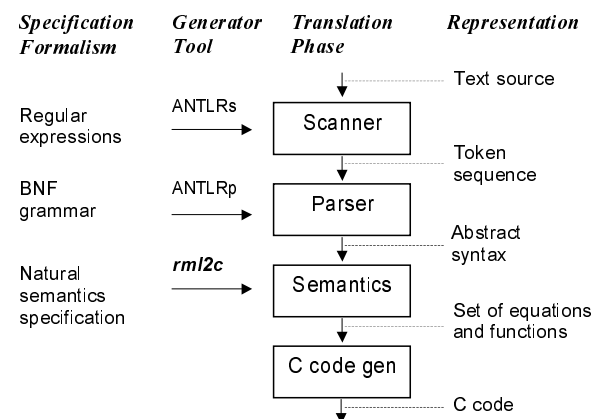


Figure 3. Phases in generating Modelica compiler modules from different kinds of specifications. The semantics module performs elaboration of the models including type checking and expansion of class structures, resulting in a set of equations, algorithms and functions.

6 Natural Semantics and RML

We have previously mentioned that a compiler generation system called RML is used to produce the Modelica translator in the open source project, from a Natural Semantics language specification. The generated translator is produced in ANSI C with a performance comparable to hand-written translators.

Below we give a short overview of the Natural Semantics formalism, and the corresponding RML specification language for expressing Natural Semantics in a computer processable way.

6.1 Natural Semantics

A Natural Semantics specification consists of a number of rules, similar to inference rules in formal logic. A rule has a structure consisting of clauses above a horizontal line followed by a clause under the line. A clause such as $a \Rightarrow b$ means a gives b .

$$\frac{c \Rightarrow b \wedge e \Rightarrow f}{a \Rightarrow b}$$

All clauses above the line are *premises*, and the clause below the line is the *consequence*. To prove the consequence, all the premises need to be proved. A small example of how rules typically appear is shown below, where a rule stating that the addition of a *Real* expression with an *Integer* expression gives a *Real* result.

$$\frac{\text{typeof}(e1) \Rightarrow \text{Real} \wedge \text{typeof}(e2) \Rightarrow \text{Integer}}{\text{elaborate}(e1 + e2) \Rightarrow \text{Real}}$$

6.2 RML

Modelica semantics is specified using the RML specification language (Pettersson 1995, 1999). The RML language is based on Natural Semantics, and uses features from languages like SML (Miller et. al. 1991) to allow for strong typing and datatype constructors. The RML source, e.g. a Modelica specifications, is compiled by an RML compiler (rml2c) to produce a translator for the described language. The RML compiler generates an efficient ANSI C program that is subsequently compiled by an ordinary C compiler, e.g. producing an executable Modelica translator. The RML tool has also been used to produce compilers for Java, Pascal and a few other languages.

6.2.1 Correspondence between Natural Semantics and RML.

The correspondence between Natural Semantics and RML is perhaps easiest to understand using an example. The following three Natural Semantics rules specifies the types of expressions containing addition operators and *Boolean* constants:

$$\frac{\text{typeof}(e1) \Rightarrow \text{Real} \wedge \text{typeof}(e2) \Rightarrow \text{Real}}{\text{typeof}(e1 + e2) \Rightarrow \text{Real}}$$

$$\frac{\text{typeof}(e1) \Rightarrow \text{Integer} \wedge \text{typeof}(e2) \Rightarrow \text{Integer}}{\text{typeof}(e1 + e2) \Rightarrow \text{Integer}}$$

$$\frac{}{\text{typeof}(\text{FALSE}) \Rightarrow \text{Boolean}}$$

The corresponding three RML rules are collected in a relation called `typeof`, which maps expressions to types:

```
relation typeof :: Exp => Type =
...
rule  typeof(e1) => Real & typeof(e2) => Real
-----
      typeof(ADD(e1,e2)) => Real

rule  typeof(e1) => Integer & typeof(e2) => Integer
-----
      typeof(ADD(e1,e2)) => Integer

axiom  typeof(FALSE) => Boolean
end
```

6.2.2 A Simple Interpretive Semantics

As a simple example of the RML syntax, we show a small expression evaluator, which “translates” a mathematical expression to the value of the expression. First, an expression data type is declared:

```
datatype Exp = NUMBER of real
             | ADD of Exp * Exp
             | MUL of Exp * Exp
```

An expression is either a number, a sum of two other expressions, or a product of two other expressions. As an example, the expression is represented as the value.

```
MUL(NUMBER(3), ADD(NUMBER(4), NUMBER(5)))
```

Then we describe the relation `eval`. The `eval` relation relates expressions to their evaluated values, so that `eval(x) => y` always holds if `y` is the value resulting from the evaluation of `x`.

```
relation eval =
  axiom eval(NUMBER(x)) => x

  rule  eval(x) => x2 & eval(y) => y2 &
        real_add(x2,y2) => sum
        -----
        eval(ADD(x,y)) => sum

  rule  eval(x) => x2 & eval(y) => y2 &
        real_mul(x2,y2) => prod
        -----
        eval(MUL(x,y)) => prod
end
```

The first rule is an axiom, which means that the set of premises for the rule is empty. It could also have been written as a rule with nothing above the line:

```
rule -----
      eval(NUMBER(x)) => x
```

The second rule tells how to evaluate sums of two expressions. What the rule says is “if `x` evaluates to `x2`, `y` evaluates to `y2`, and the sum of `x2` and `y2` is `sum`, then the result of evaluating `ADD(x,y)` is `sum`.” The relations `real_add` and `real_mul` are predefined in RML.

When an RML specification is executed, a relation named `main` is evaluated at the top level. If our program has a main relation that looks like the following, the program would simply print the number 27 and exit.

```
relation main =
  rule  eval(MUL(NUMBER(3),
                ADD(NUMBER(4),
                    NUMBER(5)))) => x &
    real_string(x) => xs &
    print(cs)
    -----
  main(_)
end
```

The RML language has some obvious similarities with functional programming languages and logic programming languages. While resolving, or “proving” the premises in a rule, a simple resolution process is carried out which tries to find a rule in the relation which matches the arguments, and if it fails, a simple retry mechanism is used to find other possible solutions.

If none of the rules in a relation is possible to prove, the relation fails, but there is also the possibility to introduce a rule that explicitly fails, by using the keyword `fail` in the corresponding clause.

There are a few other syntactic features of RML that should be known to the reader. If an argument is not used in a rule, it can be written as an underscore, which means that it matches anything, but is not used. If a relation has the right-hand side (result) type `()`, it can be omitted together with the `=>` symbol. The `main` relation above shows an example of both of these features.

7 The Formal Specification of Modelica

The specification is separated into a number of modules, to separate different stages of the translation, and to make it more manageable. This section will briefly cover some of the most important parts of the specification. In all, the specification contains several thousand lines of RML, but it should be kept in mind that the RML code is rather sparse, with many empty or short lines.

The top level relation in the semantics is called `main`, and appears as follows:

```
relation main =

  rule  Parser.parse f => p &
        SCode.elaborate(p) => p2 &
        Inst.elaborate(p2) => d &
        DAE.dump d &
        -----
        main([f])

end
```

7.1 Parsing and Abstract Syntax

The relation `Parser.parse` is actually written in C, and calls the parser generated from a grammar by the ANTLR parser generator tool (ANTLR 1998). This parser builds an abstract syntax tree (AST) from the source file, using the AST data types in a RML module called `Absyn`. The parsing stage is not really part of the semantic description, but is of course necessary to build a real translator.

7.2 Rewriting the AST

The AST closely corresponds to the parse tree and keeps the structure of the source file. This has several disadvantages when it comes to translating the program, and especially if the translation rules should be easy to read for a human. For this reason a preparatory translation pass is introduced which translates the AST into an intermediate form, called `SCode`. Besides some minor simplifications the `SCode` structure differs from the AST in the following respects:

- All variables are described separately. In the source and in the AST several variables in a class definition can be declared at once, as in `Real x, y[17];`. In the `SCode` this is represented as two unrelated declarations, as if it had been written `Real x; Real y[17];`.
- Class declaration sections. In a Modelica class declaration the `public`, `protected`, `equation` and `algorithm` sections may be included in any number and in any order, with an implicit `public` section first. In the `SCode` these sections are collected so that all `public` and `protected` sections are combined into one section, while keeping the order of the elements. The information about which elements were in a `protected` section is stored with the element itself.

One might have thought that more work could be done at this stage, like analyzing expression types and resolving names. But due to the nature of the Modelica language, the only way to know anything about how the names will be resolved during elaboration is to do a more or less full elaboration. It is possible to analyze a class declaration and find out what the parts of the declaration would mean if the class was to be elaborated as-is, but since it is possible to modify much of the class while elaborating it that analysis would not be of much use.

7.3 Elaboration and Instantiation

- To be executed, classes in a model need to be instantiated, i.e. data objects are created according to the class declaration. There are two phases of instantiation:
- The symbolic, or compile time, phase of instantiation is usually called *elaboration*. No data objects are created during this phase. Instead the

symbolic internal representation of the model to be executed/simulated is transformed, by performing inheritance operations, modification operations, aggregation operations, etc.

- The creation of the data object, usually called *instantiation* in ordinary object-oriented terminology. This can be done either at compile time or at run-time depending on the circumstances and choice of implementation.

The central part of the translation is the *elaboration* of the model. The convention is that the last model in the source file is elaborated, which means that the equations in that model declaration, and all its subcomponents, are computed and collected.

The elaboration of a class is done by looking at the class definition, elaborating all subcomponents and collecting all equations, functions, and algorithms. To accomplish this, the translator needs to keep track of the class context. The context includes the lexical scope of the class definition. This constitutes the *environment* which includes the variables and classes declared previously in the same scope as the current class, and its parent scope, and all enclosing scopes. The other part of the context is the current set of modifiers which modify things like parameter values or redeclare subcomponents.

```
model M
  constant Real c = 5;
  model Foo
    parameter Real p = 3;
    Real x;
    equation
      x = p * sin(time) + c;
  end Foo;

  Foo f(p = 17);
end M;
```

In the example above, elaborating the model *M* means elaborating its subcomponent *f*, which is of type *Foo*. While elaborating *f* the current environment is the parent environment, which includes the constant *c*. The current set of modifications is (*p* = 17), which means that the parameter *p* in the component *f* will be 17 rather than 3.

There are many semantic rules that takes care of this, but only a few are shown below. They are also somewhat simplified to focus on the central aspects.

7.4 The `elab_class` and `elab_element` Relations

The relation `elab_class` elaborates a class. It takes five arguments, the environment *env*, the set of modifications *mod*, the prefix *pre* which is used to build a globally unique name of the component in a hierarchical fashion, a collection of connection sets *csets*, and the class definition *c*. It opens a new scope in the environment where all the names in this class will be stored, and then uses a relation called `elab_class_in` to do most of the work. Finally it generates equations from the connection sets collected while elaborating this class. The “result” of the relation are the *elaborated* equations and some information about what was in the class. In the case of a function, regarded as a restricted class, the result is an algorithm section.

One of the most important relations is `elab_element`, that elaborates an element of a class. An element can typically be a class definition, a variable or constant declaration, or an extends clause. Below is shown *only* the rule for elaborating variable declarations

```
relation elab_class: (Env, Mod, Prefix, Connect.Sets, SCode.Class) =>
  (DAE.Element list, Connect.Sets, Types.Type) =

  rule Env.open_scope(env) => env' &
  elab_class_in(env', mod, pre, csets, c) => (dae1, _, csets', ci_state', tys) &
  Connect.equations csets' => dae2 & list_append(dae1, dae2) => dae &
  mktype(ci_state', tys) => ty
  -----
  elab_class(env, mod, pre, csets, c as SCode.CLASS(n, _, r, _)) => (dae, [], ty)
  end
```

```

relation elab_element: (Env, Mod, Prefix, Connect.Sets, Scode.Element) =>
(DAE.Element list, Env, Connect.Sets, Types.Var list) =
...
rule Prefix.prefix_cref(pre, Exp.CREF_IDENT(n, [])) => vn &
  Lookup.lookup_class(env, t) => (cl, classmod) & Find the class definition
  Mod.lookup_modification(mods, n) => mm &
  Mod.merge(classmod, mm) => mod & Merge the modifications
  Mod.merge(mod, m) => mod' &
  Prefix.prefix_add(n, [], pre) => pre' & Extend the prefix
  elab_class(env, mod', pre', csets, cl) Elaborate the variable
    => (dael, csets', ty, st) &
  Mod.mod_equation mod' => eq & If the variable is declared with a default equation,
  make_binding (env, attr, eq, cl) add it to the environment with the variable.
    => binding &
  Env.extend_frame_v(env, Add the variable binding to the environment
    Env.FRAMEVAR(n, attr, ty, binding))
    => env' &
  elab_mod_equation(env, pre, n, mod') Fetch the equation, if supplied
    => dae2 &
  list_append(dael, dae2) => dae Concatenate the equation lists
  -----
  elab_element(env, mods, pre, csets,
    SCode.COMPONENT(n, final, prot, attr, t, m))
    => (dae, env', csets', [(n, attr, ty)])
...
end

```

7.5 Output

The equations, functions, and variables found during elaboration are collected in a list of objects of type DAEcomp:

```

datatype DAEcomp = VAR of
Exp.ComponentRef * VarKind
| EQUATION of Exp * Exp
...

```

As the final stage of translation, in the current version of the translator, functions and expressions in this list are converted to C code.

8 A Small Benchmark

We have evaluated the current implementation of the open source Modelica compiler on a small benchmark consisting of solving a simplex optimization problem consisting of 25 variables and 5 conditions. The measured execution time was averaged over 100 executions on a Sun UltraSparcstation 10. The Modelica code of the simplex algorithm is available in Appendix A. We found that the execution time for the Modelica code was a factor of 1.54 slower than handwritten C code for the same algorithm. This is a preliminary result obtained at the time of writing this paper, and we expect to be able to get even closer to C code execution performance by tuning the implementation.

9 Conclusions

We have developed the first version of an open source Modelica environment, to a large extent based on a Modelica compiler automatically generated from a formal Natural Semantics specification of Modelica. This formal specification is intended to become a reference specification for research purposes and for future Modelica implementers.

An important short-term goal for this open source project is to provide an interactive and efficient computational environment for using Modelica as a high level strongly typed programming language for computational applications.

In the longer-term perspective we expect to extend the system to also provide simulation of equation based models in Modelica, however with lower performance and handling models of less complexity than what is currently managed by commercial implementation such as Dymola and MathModelica.

Appendix – The Simplex Algorithm

```

function pivot1
  input Real b[:, :];
  input Integer p;
  input Integer q;
  output Real a[size(b,1),size(b,2)];
protected
  Integer M;
  Integer N;
algorithm
  a := b;
  N := size(a,1)-1;
  M := size(a,2)-1;
  for j in 0:N loop
    for k in 0:M loop
      if j<>p and k<>q then
        a[j+1,k+1] := a[j+1,k+1]-
a[p+1,k+1]*a[j+1,q+1]/a[p+1,q+1];
      end if;
    end for;
  end for;
  for j in 0:N loop
    if j<>p then
      a[j+1,q+1] := 0;
    end if;
  end for;
  for k in 0:M loop
    if k<>q then
      a[p+1,k+1]:=a[p+1,k+1]/a[p+1,q+1];
    end if;
  end for;
  a[p+1,q+1] := 1;
end pivot1;

function simplex1
  input Real matr[:, :];
  output Real x[size(matr,2)-1];
  output Real z;
  output Integer q;
  output Integer p;
protected
  Real a[size(matr,1),size(matr,2)];
  Integer M;
  Integer N;
algorithm
  N := size(a,1)-1;
  M := size(a,2)-1;
  a := matr;
  p:=0; q:=0;
  while not (q==(M+1) or p==(N+1)) loop
    q := 0;
    while not (q == (M+1) or
a[0+1,q+1]<0) loop
      q:=q+1;
    end while;
    p := 0;
    while not (p == (N+1) or
a[p+1,q+1]>0) loop

```

```

      p:=p+1;
    end while;
    for i in p+1:N loop
      if a[i+1,q+1] > 0 then
        if (a[i+1,M+1]/a[i+1,q+1]) <
(a[p+1,M+1]/a[p+1,q+1]) then
          p := i;
        end if;
      end if;
    end for;
    if (q < M+1) and (p < N+1) then
      a := pivot1(a,p,q);
    end if;
  end while;
  for i in 1:M loop
    x[i] := -1;
    for j in 1:N+1 loop
      if (x[i] < 0) and ((a[j,i] >=
1.0) and (a[j,i] <= 1.0)) then
        x[i] := a[j,M+1];
      elseif ((a[j,i] < 0) or (a[j,i] >
0)) then
        x[i] := 0;
      end if;
    end for;
  end for;
  z := a[1,M+1];
end simplex1;

```

References

- Abadi Martin and Cardelli Luca. A Theory of Objects. Springer Verlag, ISBN 0-387-94775-2, 1996.
- Elmqvist Hilding, Dag Brück, and Martin Otter, Dymola - User's Manual. Dynasim AB, Research Park Ideon, Lund, Sweden, 1996
- Kågedal, D., Fritzson, P. Generating a Modelica Compiler from Natural Semantics Specifications. Summer Computer Simulation Conference '98, Reno, Nevada, USA, July 19-22, 1998.
- Modelica Home Page <http://www.Modelica.org>
- Dymola Home Page: <http://www.Dynasim.se>
- MathModelica Home Page: <http://www.MathCore.com>
- Mikael Pettersson. Compiling Natural Semantics, Linköping Studies in Science and Technology. Dissertation No. 413, 1995.
- Mikael Pettersson. Compiling Natural Semantics. LNCS 1549, Springer-Verlag, 1999.
- Miller Robin, Tofte Mads, Harper Robert. *Commentary on Standard ML*. The MIT Press, 1991.
- ANTLR home page: <http://www.ANTLR.org/>
- Frank Pagan. *Formal Specification of Programming Languages: A Panoramic Primer*, Prentice-Hall, ISBN 0-13-329052-2, 1981.
- Gilles Kahn. Natural Semantics. In *Proc. of the Symposium on Theoretical Aspects on Computer Science, STACS'87*, LNCS 247, pp 22-39. Springer-Verlag, 1987.

Extending Modelica for Partial Differential Equations

Levon Saldamli^{*}, Peter Fritzson^{*} and Bernhard Bachmann[†]

Abstract

Currently, Modelica only supports models containing constants, time-dependent variables, and time-derivatives of variables, i.e. ordinary differential and algebraic equations. In this article, we present how the Modelica language can be extended to support object-oriented modeling with partial differential equations (PDEs), in order to solve initial and boundary value problems. The techniques we present have fairly general applicability to 1D, 2D or 3D domains, although we focus mostly on 2D domains in this paper. We also describe the architecture of a prototype implementation where the PDE problem is translated and passed to an external mesh generator and a PDE solver for solution using the finite element method. An example of a stationary heat conduction problem is included together with execution results.

1 Introduction

The modeling language Modelica [4, 5, 7, 10] is currently used for modeling and simulation of systems with ordinary differential equations containing time-dependent variables and derivatives of such variables with respect to time. It is desirable to also specify models where variables vary with position in space and where partial differential equations (PDEs) occur. Therefore, there is a need to extend Modelica to support such models.

A PDE problem is solved in order to find an unknown, spatially distributed function u , that is implicitly defined by a partial differential equation. For a unique solution boundary conditions at the boundary of the geometric region of the problem is needed, and also the initial conditions if the problem is time-dependent. There can be

different boundary conditions for different parts of the boundary, and the conditions can be known values of the unknown function or its derivatives. Initial conditions can consist of values of the unknown function or its derivatives.

Modelica is an object-oriented language, supporting inheritance and component-based modeling. Extensions to support PDEs should be done with these concepts in mind in order for a PDE problem to be specified in a convenient way similar to other models written in Modelica. Previously, some basic extensions needed in Modelica were presented [14]. The domains were described by defining the limits of the space variables using constants or expressions containing other space variables in order to allow fairly general domains. In this paper, we support a more convenient domain definition, using parametric expressions for describing the boundary of the domain. We also describe how a problem can be specified with the PDE, the boundary conditions, the domain and its boundary defined as components.

This paper is organized as follows: Section 2 contains an overview of related work, Section 3 presents the problem specification and new language syntax, Section 4 describes the implementation environments, Section 5 illustrates an example problem and its solution, and Section 6 contains some conclusions and future work.

2 Related Work

There are different categories of packages for solving PDEs. Some of them are code libraries, where the PDE is not separately specified but a numerical solver is written using a programming language and components from these libraries in order to solve the specific PDE problem. PETSc [2], Diffpack [3] and Overture [12] are some packages in this category. Compose [17] is a similar package, written as a framework built upon Overture, with an object-oriented design that separates the equation definitions and numerical solver implemen-

^{*}Department of Computer and Information Science, Linköpings universitet, Linköping, Sweden. {levsa, petfr}@ida.liu.se

[†]Fachbereich Mathematik und Technik, Fachhochschule Bielefeld, Bielefeld, Germany. bernhard.bachmann@fh-bielefeld.de

tation. Equations in Compose are defined using the C++ classes in the framework or by adding new classes to define new equations and numerical solvers.

There are also problem solving environments, that contain integrated tools for the different steps of the modeling and simulation process, such as graphical tools for defining the domain, tools for specifying or selecting a numerical solver among several solvers, and tools for visualization of the simulation results. PELLPACK [9] is such a problem solving environment that contains several PDE solvers and has a high level language for the PDE problem definition. FEMLAB [6] which is a package for MATLAB, is another simulation tool, with graphical user interface where the user can choose a model among many predefined PDE models, modify its parameters, graphically define the problem domain and assign boundary conditions, simulate the model and visualize the results.

An environment that is more language oriented, analogous to Modelica, is gPROMS [11]. This environment has a high level language for specifying PDE models on rectangular domains, where complex partial differential and algebraic equations and mixed systems of integral, partial and ordinary differential and algebraic equations can be solved.

The approach taken in our present work to extend Modelica with PDEs, called PDE-Modelica, combines the usage of a high level language, object-oriented and component-based modeling, and the possibility to use different solvers and automatic solver generation for a given PDE problem.

3 Domain and PDE definition

In this section, we describe how to define the problem domain using lines and parametric curves. Also, a hierarchical PDE model definition using coefficient-based PDEs similar to FEMLAB's coefficient form is described.

3.1 Domain Description

The domain of the PDE problem is $D \subset \mathbb{R}^n$. In this paper we consider the two-dimensional case, $n = 2$. In most practical cases it is sufficient to define the domain by a parametric curve $\{(x_s, y_s) \mid s \in [s_{start}, s_{end}]\}$ describing the bound-

ary of the region, which is a sufficiently general way of stating the geometry of the domain. The curve should be closed and non self-intersecting for the parameter range specified. In the two-dimensional case, the XY-plane is divided into two regions by the curve, with the intended domain being the region at the left side of the curve.

The boundary defined in this way is used to generate a mesh for the numerical PDE solver. An external mesh generator is used to generate the mesh.

From the boundary definition, an external mesh generator is called to generate a triangular mesh which is passed to the numerical solver.

A domain class is defined by introducing a new kind of restricted class in Modelica called **domain**, where the independent space variables to be used are declared using the **space** keyword, and the boundary is described in a special section called **boundary**. The **boundary** section can contain three different constructs that define the boundary: **lines()**, **curve()** or **composite()**, described in the following sections.

3.1.1 The lines() Boundary Construct

In case of single lines or a number of connected lines, a special construct **lines()** is used, for efficiency reasons. A line segment is defined as follows:

```
domain Line2D "A line segment"
  extends Cartesian2D;
  parameter Real x0=0, y0=0, x1=1, y1=1;
  boundary
    lines({{x0,y0},{x1,y1}});
end Line2D;
```

The **lines()** construct contains an expression which is an array of points, defining the starting point, the intermediate points and the end point of the connected lines (see Figure 1).

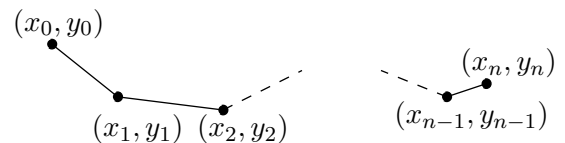


Figure 1: Connected lines that is described by the construct $\text{lines}(\{\{x_0, y_0\}, \{x_1, y_1\}, \dots, \{x_n, y_n\}\})$

3.1.2 The curve() Boundary Construct

There are several alternative ways to specify the parametric expression that defines the boundary as a curve. Using the `where...in...` construct which already has been used to specify domains for expressions [14], the curve can be defined as follows:

```
domain Cartesian2D "For all 2D-domains"
  space Real x,y;
end Cartesian2D;

domain Circle2D "Circular with r=1"
  extends Cartesian2D;
boundary
  curve(cos(2*PI*u),
        sin(2*PI*u)) where u in (0,1);
end Circle2D;
```

The boundary of this domain is defined by the curve generated by varying the value of the temporary variable u from 0 to 1. The comma-separated list of expressions in the `curve()`-construct are used to calculate the Cartesian coordinates of the points on the curve. In order for the curve to be closed, the resulting points in the XY-plane at $u = 0$ and $u = 1$ should have the same coordinates. Other requirements might be needed for the curve depending on the mesh generator used by the numerical solution stage.

3.1.3 The composite() Boundary Construct

In many cases, the boundary of the problem domain is difficult to define as a single parametric curve, but is rather defined by a number of connected lines and curves. Also, the boundary conditions for the PDE problem are often different on different parts of the boundary. Therefore, when the boundary curve is specified, there must be a way to refer to different parts of the curve when assigning boundary conditions. One solution to these problems is to have a boundary description that consists of several components, each of which are curves. The boundary components can be declared in the declaration part of the domain description. For example, a rectangular boundary can be defined using four line segments `right`, `top`, `bottom`, and `left` (see Figure 2). These parts of the boundary can be instantiated in the declaration part of the domain class `Rectangle2D` as follows:

```
domain Rectangle2D "A 6 by 4 rectangle"
  extends Cartesian2D;
  parameter Real cx=0, cy=0, w=3, h=2;
  Line2D right(x0=cx+w, y0=cy-h,
              x1=cx+w, y1=cy+h);
  Line2D top(x0=cx+w, y0=cy+h,
            x1=cx-w, y1=cy+h);
  Line2D left(x0=cx-w, y0=cy+h,
             x1=cx-w, y1=cy-h);
  Line2D bottom(x0=cx-w, y0=cy-h,
               x1=cx+w, y1=cy-h);
boundary
  composite(right, top, left, bottom);
end Rectangle2D;
```

The domain `Rectangle2D` can be seen in Figure 2. The `composite` operator is used to combine several curve segments into a complete boundary. The setting of the start and end points of the line segments and the order of the arguments to the `composite` operator must be consistent, and the direction of the resulting curve must be correct in order that the correct region is defined. Some of these requirements can be automatically fulfilled if the `composite` operator is allowed to translate each given curve segment so that the starting point of that curve matches the end point of the previous curve segment.

Although both `Line2D` and `Rectangle2D` are defined as domains, they represent different kinds of objects. The `Line2D` domain is not intended to be used as a domain by itself, but rather as a boundary component of another domain. This difference could be expressed in the definition by for example using the `partial` keyword in the definition of `Line2D`:

```
partial domain Line2D
  "Defines a part of a boundary"
  ...
```

Another alternative is to use a different keyword than `domain` for classes that represent only parts of a boundary.

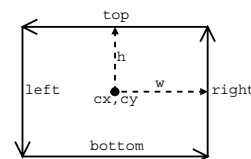


Figure 2: A rectangular domain `Rectangle2D`, defined using line segments. Note that the direction of the lines must be consistent.

3.2 Hierarchical Definition of PDEs and Boundary Conditions

In order to simplify PDE model definition, a general PDE model can be written as a base model in PDE-Modelica with the coefficients as parameters. This model can either be instantiated directly with appropriate modifications to the parameters or used as a base class to define a more specific PDE model with some parameters set which subsequently can be instantiated and used when needed. Similarly, boundary conditions can be defined using base models and inheritance. A coefficient-based PDE base model can be defined as follows:

```
model PDE2D
  space Real x,y;
  Real u(x,y);
end PDE2D;

model PDECoeff2D
  extends PDE2D;
  parameter Real da = 0;
  parameter Real c = 0;
  parameter Real a = 0;
  parameter Real f = 0;
equation
  da*der(u) - div(c*grad(u)) + a*u = f;
end PDECoeff2D;
```

The variable u represents the unknown variable, a function of time and the space variables. All parameters can be constants or functions of the space variables. However, in this example, the coefficients da , c , a and f are restricted to be constants only, for clarity. The **der** operator is an operator in Modelica and defines the first time-derivative of a variable. The **div** and **grad** operators can be additional operators in PDE-Modelica corresponding to the partial differential operators **divergence** and **gradient** that are often used in mathematical literature. The equation above written with mathematical notation follows:

$$da \frac{\partial u}{\partial t} - \nabla \cdot (c \nabla u) + au = f$$

Using **PDECoeff2D** as the base model, a simple, steady-state heat transfer model can now be written as:

```
model HeatTransfer
  extends PDECoeff2D(c=1);
end HeatTransfer;
```

A Robin boundary condition, used in a heat problem to describe a boundary that is neither a

perfect conductor nor a perfect insulator, can be written by first writing a general Robin boundary condition:

```
model Robin "Robin boundary condition"
  extends PDE2D;
  parameter Real c = 1;
  parameter Real q = 1;
  parameter Real g = 0;
equation
  nder((c*grad(u))) + q*u = g;
end Robin;
```

In mathematical notation, this equation is written as follows:

$$\frac{\partial}{\partial n}(c \nabla u) + qu = g$$

The operator **nder()** is a special operator that represents the derivative in the outward normal direction with respect to the associated domain boundary.

Other types of boundary conditions, e.g. Dirichlet and Neumann conditions, describing a perfect heat conductor and a perfect insulator, respectively, can be defined by extending the **Robin** class and setting the appropriate parameters to zero, as follows:

```
model Neumann
  extends Robin(q=0);
end Neumann;

model Dirichlet
  extends Robin(c=0);
end Dirichlet;
```

For heat transfer problems, a more specific version of the Robin boundary condition can be defined by inheriting the **Robin** class and adding application specific parameters and mapping them to the general parameters:

```
model HeatRobin "For heat transfer"
  extends Robin(c=k,
               q=hh,
               g = qh+hh*Tinf);
  parameter Real k=1;
  parameter Real qh=0;
  parameter Real hh=1;
  parameter Real Tinf=25;
end HeatRobin;
```

The corresponding mathematical equation with these parameters is as follows:

$$\frac{\partial}{\partial n}(k \nabla u) = qh + hh(T_{inf} - u)$$

where qh is the source term, hh is the heat transfer coefficient and T_{inf} is the external temperature.

3.3 Problem definition

Once the models for the PDE and the boundary conditions are stated and the domain is defined, the problem can be put together by instantiating the PDE model, the boundary conditions, and the domain and associating the boundary conditions with the boundary parts. In order to associate boundary conditions and boundary elements, an implicit variable `bc` (short for boundary condition) is introduced in the restricted class `domain`. For each domain instance this variable is assigned the desired boundary condition. Similarly, a PDE is associated with a domain by instantiating the PDE model and assigning the instance to the variable `eq` (short for equation), also a builtin variable in the restricted class `domain`. The complete problem statement is then:

```
model PDEModel
  Neumann h_iso;
  Dirichlet h_heated(g=50);
  HeatRobin h_glass(hh=30000);
  HeatTransfer ht;
  Rectangle2D dom;
equation
  dom.eq = ht;
  dom.left.bc = h_glass;
  dom.right.bc = h_heated;
  dom.top.bc = h_iso;
  dom.bottom.bc = h_iso;
end PDEModel;
```

Here, a Dirichlet condition with a constant value of 50° for u is used to emulate a heat source on the right side of the domain, Robin condition is used for a non-isolating glass layer on the left side, and Neumann condition is used for the isolated top and bottom sides. The PDE model `HeatTransfer` is instantiated as `ht`, and used in the interior of the domain `dom`, which is an instance of the `Rectangle2D` class.

4 Results

The PDE extensions discussed in Section 3 were implemented in the prototype Modelica translator generated from a Natural Semantics specification of Modelica (see Section 4.2). A heat transfer example is solved in the following section in order to demonstrate the PDE extensions and the prototype. In this example, a stationary problem is solved, because the PDE solver currently used with the prototype does not handle time-dependent problems.

4.1 Example

A stationary heat conduction problem is considered. The problem is described by Poisson's equation:

$$-\nabla \cdot (c \nabla u) = g$$

where c is the heat conductivity coefficient, and g is the source term. In this example, c is set to 1 and g is set to 0.

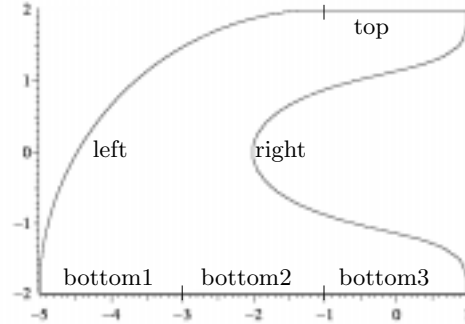


Figure 3: The problem domain with its different boundary sections.

The equation is solved on the domain shown in Figure 3. The left side of the domain is defined by a 90° arc, using an instance of a more general version of the domain class `Circle2D` defined in Section 3.1.2. The right side is defined by a Bezier curve with six control points, as the instance `right` of type `Bezier2D` defined below. The PDE-Modelica code for defining Bezier curves using De Casteljau's Algorithm follows:

```
function bezier
  constant Integer n=6;
  input Real px[n];
  input Real u;
  output Real res;
  Real qx[n];
algorithm
  for i in 1:n loop
    qx[i] := px[i];
  end for;
  for k in 1:n-1 loop
    for i in 1:(n-k) loop
      qx[i] := (1-u)*qx[i] + u*qx[i+1];
    end for;
  end for;
  res := qx[1];
end bezier;

domain Bezier2D
  constant Integer n=6;
  parameter Real px[n];
  parameter Real py[n];
  space Real u;
```

```

    BoundaryCondition bc;
    boundary
      curve(bezier(px,u), bezier(py,u));
    end Bezier2D;

```

The complete description of the domain for the heat transfer example in PDE-Modelica is:

```

domain HeatExampleDomain
  extends Domain2D;
  Circle2D left(x0=-1, y0=-2, ra=4,
    a=PI/2, b=PI/2);
  Bezier2D right(px={1.0, 1.3, -4.0,
    -4.0, 1.3, 1.0},
    py={-2.0, 0.0, -3.0,
    3.0, 0.0, 2.0});
  Line2D top(x0=1, y0=2, x1=-1, y1=2);
  Line2D bottom1(x0=-5, y0=-2,
    x1=-3, y1=-2);
  Line2D bottom2(x0=-3, y0=-2,
    x1=-1, y1=-2);
  Line2D bottom3(x0=-1, y0=-2,
    x1=1, y1=-2);
  boundary
    composite(right, top, left,
      bottom1, bottom2, bottom3);
end HeatExampleDomain;

```

At the right border, the Robin boundary condition is used, in order to model heat flow through the boundary that is proportional to the temperature difference. The temperature outside the domain is set to 20° . The middle part of the bottom border is used as a heat source, with a Dirichlet boundary condition $u = 50$. The other parts of the bottom border as well as the left and the top borders are perfectly insulated, using the homogeneous Neumann boundary conditions.

A plot of the solution can be seen in Figure 4. This example was solved using the finite element solver *rheolef* [15], and *bamg* [1] was used as the mesh generator.

4.2 Implementation

We are working with two prototype environments where the ideas described in Section 3 are being tested. The prototype written in Mathematica uses MathModelica [8] as the Modelica implementation and a numerical PDE solver generator [16] for solving the PDEs. The different modules of this environment can be seen in Figure 5. Here, the models are written in a Mathematica style Modelica syntax, and the domain analyzer generates domain information that is sent to an external mesh generator. The PDE analyzer collects the PDEs and the boundary conditions and calls the solver generator that generates a finite

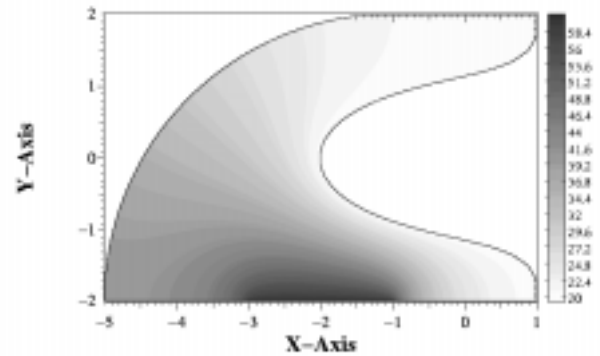


Figure 4: A stationary heat transfer example. The middle section of the bottom border is a heat source with $u = 50^\circ$, the curved right border is non-insulated with outside temperature 20° , and the other borders are insulated.

element solver in C++. The advantages of this environment is the access to symbolic manipulation in Mathematica, and the MathModelica input format that is easy to extend in order to test new language syntax extensions.

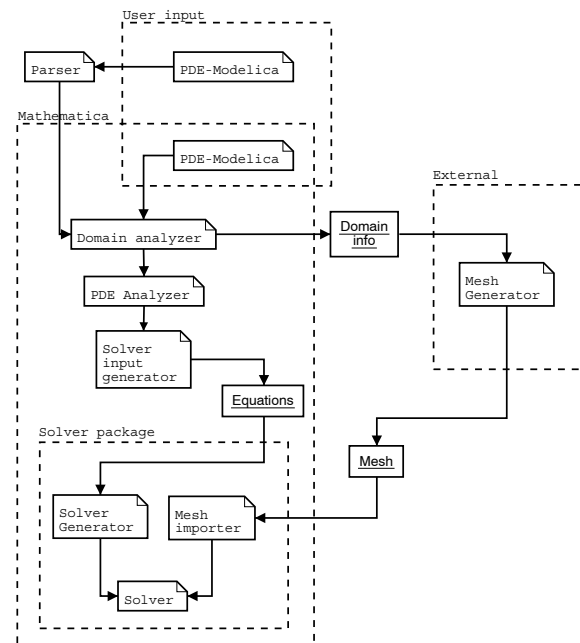


Figure 5: The PDE-Modelica prototype in the Math-Modelica environment

The other prototype environment consists of a Modelica parser, a compiler generated by the RML [13] system from a Natural Semantics description of Modelica, an external mesh generator and a PDE solver. The structure of this environ-

ment can be seen in Figure 6. The compiler generates C++ code from the PDE-Modelica description. The resulting code generates the discretized boundary at runtime, calls the mesh generator to triangulate the domain and finally calls the finite element solver.

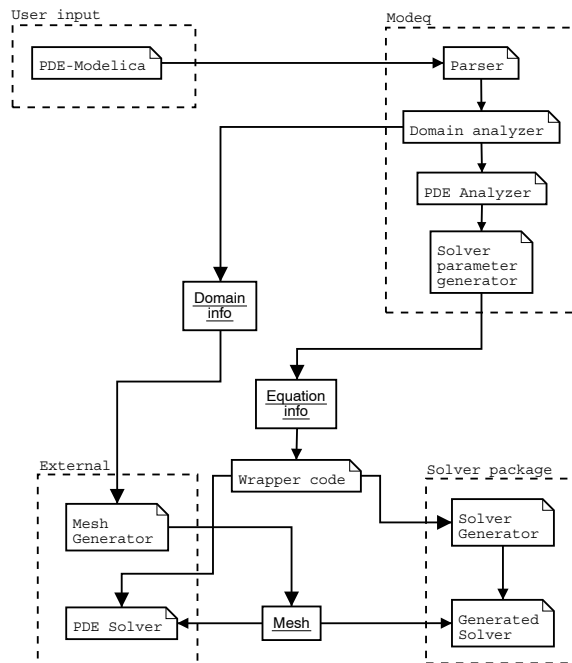


Figure 6: The PDE-Modelica prototype using the Modelica translator generated from Natural Semantics specification of Modelica in RML.

The current version of the prototype ignores the equation parts of the PDE and boundary condition models and assumes a certain structure of the PDE. A specific solver adapted to the problem is called automatically with the parameters extracted from the models. This can be done because the base model approach is used when writing the PDE models, i.e. the solver needs only to be associated with the base model, and parameters of the base model are transferred to the solver.

5 Conclusions and Future Work

We have presented a design for specifying PDE problem domains in Modelica by expressing the boundary of the domain using lines and parametric curves. We have also shown a simple example of hierarchically defined PDE model and boundary conditions and how these can be used in a problem specification together with a domain definition.

Our future work will consist of adding support

for the equation parts of the PDE and boundary condition models, instead of having predefined equations. Also, modeling with both PDE models and the current Modelica models with DAEs and the interaction between these different kinds of models needs to be considered. Support for combination of domains using set operations such as union, intersection, etc., and composition of domains into bigger domains using connect statements with different PDE models on each partial domain is another possible future extension.

References

- [1] BAMG home page. <http://www-rocq.inria.fr/gamma/cdrom/www/bamg/eng.htm>.
- [2] Satish Balay, Kris Buschelman, William D. Gropp, Dinesh Kaushik, Lois Curfman McInnes, and Barry F. Smith. PETSc home page. <http://www.mcs.anl.gov/petsc/>, 2001.
- [3] Diffpack home page. <http://www.diffpack.com/>.
- [4] H. Elmqvist, S. E. Mattsson, and M. Otter. A language for physical system modeling, visualization and interaction. In *Proceedings of the 1999 IEEE Symposium on Computer-Aided Control System Design*, Hawaii, Aug. 1999.
- [5] H. Elmqvist and S.E. Mattsson. Modelica – the next generation modeling language – an international design effort. In *Proceedings of the First World Congress on System Simulation*, Singapore, Sept. 1–3 1997.
- [6] FEMLAB home page. <http://www.femlab.com/>.
- [7] P. Fritzson and V. Engelson. Modelica—A unified object-oriented language for system modeling and simulation. In Eric Jul, editor, *ECOOP '98—Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 67–90. Springer, 1998.
- [8] P. Fritzson, J. Gunnarsson, and M. Jirstrand. MathModelica - An Extensible Modeling and Simulation Environment with Integrated Graphics and Literate Programming. In *Proc.*

of the 2nd International Modelica Conference, Munich, March 2002.

- [9] E. N. Houstis, J. R. Rice, S. Weerawarana, A. C. Catlin, P. Papachiou, K.-Y. Wang, and M. Gaitatzes. PELLPACK: a problem-solving environment for PDE-based applications on multicomputer platforms. *ACM Transactions on Mathematical Software*, 24(1):30–73, March 1998.
- [10] Modelica Association. *Modelica – A Unified Object-Oriented Language for Physical Systems Modeling - Language Specification Version 1.4*, Dec 2000. <http://www.modelica.org>.
- [11] M. Oh. *Modelling and Simulation of Combined Lumped and Distributed Processes*. PhD thesis, University of London, 1995.
- [12] Overture home page. <http://www.llnl.gov/CASC/Overture/>.
- [13] M. Pettersson. *Compiling Natural Semantics*. volume 1549 of *LNCS*. Springer-Verlag, 1999.
- [14] L. Saldamli and P. Fritzson. A Modelica-based Language for Object-Oriented Modeling with Partial Differential Equations. In A. Heemink, L. Dekker, H. de Swaan Arons, I. Smith, and T. van Stijn, editors, *Proc. of the 4th International EUROSIM Congress*, Delft, The Netherlands, June 2001.
- [15] Pierre Saramito and Nicolas Roquet. Rheolef home page. <http://www-lmc.imag.fr/lmc-edp/Pierre.Saramito/rheolef/>, 2002.
- [16] K. Sheshadri and P. Fritzson. A General Symbolic PDE-Solver Generator: Explicit Schemes. *Accepted for publication in Scientific Programming*, 2001.
- [17] K. Åhlander. *An Object-Oriented Framework for PDE Solvers*. PhD thesis, Uppsala University, 1999.

Formulation of dynamic optimization problems using Modelica and their efficient solution

Rüdiger Franke
ABB Corporate Research
Wallstadter Str. 59
68526 Ladenburg, Germany
E-Mail: Ruediger.Franke@de.abb.com

Abstract

Dynamic optimization problems often arise in advanced model based control. For example in model based predictive control and in the estimation of process parameters or not measured process signals, the underlying problems can be treated with optimization.

A process model formulated in Modelica [10] can be used as a core part in the formulation of dynamic optimization problems. This allows an efficient engineering of advanced control applications as simulation models are reused for optimization.

The paper discusses, how different types of dynamic optimization problems can be formulated based on a nonlinear dynamic system model. Furthermore, the efficient numerical solution of dynamic optimization problems as large-scale nonlinear programming problems is outlined. The treatment of state constraints is emphasized in this context. Possibilities for obtaining model sensitivities as required by an optimization solver are discussed.

However, the class of models that can be used for optimization in this way is limited, compared to all models that can be formulated in Modelica and used for initial-value simulation. Specific requirements by optimization solvers are discussed together with features of the Modelica language supporting their consideration in model formulations.

The optimal startup of a power plant serves as a practical example.

1 Introduction

Dynamic optimization problems occur if parameters and control inputs of a dynamic system shall be in-

fluenced so that a cost criterion is minimized subject to constraints. They are playing an increasingly important role in control engineering and in process engineering. Typical applications involving dynamic optimization are e.g. nonlinear model predictive control (NMPC), data reconciliation, and integrated design and control of technical processes.

Higher requirements on the efficiency of industrial processes, together with the availability of new modeling and solution technologies, are causing a trend towards the treatment of dynamic optimization problems for rigorous physical models. Unfortunately a substantial effort is generally needed to formulate an optimization model fulfilling both: high model accuracy and high solution efficiency.

This paper discusses the use of Modelica to formulate dynamic system models for optimization. A substantial reduction of the effort for model building is achieved by reusing available simulation models for optimization and by exploiting features of Modelica for application specific model adaptation. The solution of dynamic optimization problems applying large-scale nonlinear programming is outlined and requirements of state-of-the-art optimization solvers on the model are discussed.

2 Dynamic optimization problems

2.1 Nonlinear Dynamic System Model

Modelica allows the object oriented modeling of dynamic systems by differential and algebraic equations. The object oriented Modelica model is typically translated to a mathematical system of differential and algebraic equations prior to its treatment with numerical solvers. Here it is assumed that the result of the model

translation is a system of ordinary differential equations of the form

$$\dot{\mathbf{x}}(t) = \mathbf{f}[\mathbf{x}(t), \mathbf{u}(t), \mathbf{z}(t), \mathbf{p}, t], \quad (1)$$

$$\mathbf{f}: \mathbb{R}^{n_x} \times \mathbb{R}^m \times \mathbb{R}^{n_z} \times \mathbb{R}^{n_p} \mapsto \mathbb{R}^{n_x}$$

$$\mathbf{y}(t) = \mathbf{g}[\mathbf{x}(t), \mathbf{u}(t), \mathbf{z}(t), \mathbf{p}, t], \quad (2)$$

$$\mathbf{g}: \mathbb{R}^{n_x} \times \mathbb{R}^m \times \mathbb{R}^{n_z} \times \mathbb{R}^{n_p} \mapsto \mathbb{R}^{n_y}$$

Model variables are internal continuous-time states $\mathbf{x} \in \mathbb{R}^{n_x}$, control inputs $\mathbf{u} \in \mathbb{R}^m$, disturbance inputs $\mathbf{z} \in \mathbb{R}^{n_z}$, constant parameters $\mathbf{p} \in \mathbb{R}^{n_p}$, and model outputs $\mathbf{y} \in \mathbb{R}^{n_y}$.

The model behavior is completely determined by the system equations \mathbf{f} and the output equations \mathbf{g} , if initial states $\mathbf{x}_0 = \mathbf{x}(t_0)$, external inputs $\mathbf{u}(t), \mathbf{z}(t), t \in [t_0, t_f]$, and parameters \mathbf{p} are given. The outputs $\mathbf{y}(t), t \in [t_0, t_f]$ can then be obtained by solving the system of differential equations using initial-value simulation.

However, often some of the required information is not explicitly known, but can be obtained by minimizing a cost function. In many of those cases, a feasible solution can be further specified by constraining model variables. Optimization is a universal tool for treating those inverse problems.

2.2 Estimation Problem

An example for an inverse problem is the estimation of unknown parameters \mathbf{p} and/or initial states \mathbf{x}_0 based on measured inputs and outputs. The estimation problem can be solved by minimizing a least squares criterion

$$\sum_{i=1}^{n_{\bar{\mathbf{y}}}} \|\mathbf{y}(t_i) - \bar{\mathbf{y}}(t_i)\|^2 \rightarrow \min_{\mathbf{x}_0, \mathbf{p}} \quad (3)$$

for the set of measurement data $\{\bar{\mathbf{y}}(t_i), t_i \in [t_0, t_f], i = 1, \dots, n_{\bar{\mathbf{y}}}\}$.

2.3 Design Parameter Optimization Problem

Some model parameters might be free or given within useful ranges, instead of with fixed values. Optimization can be used to determine values for those unknown parameters that minimize a criterion $F(\mathbf{p}) : \mathbb{R}^{n_p} \mapsto \mathbb{R}^1$

$$F(\mathbf{p}) \rightarrow \min_{\mathbf{p}} \quad (4)$$

subject to parameter bounds $\mathbf{p}_{\min} \leq \mathbf{p} \leq \mathbf{p}_{\max}$ and required system outputs, e.g. $\mathbf{y}(t) \geq \mathbf{y}_{\min}(t), t \in [t_0, t_f]$.

2.4 Optimal Control Problem

The control inputs $\mathbf{u}(t), t \in [t_0, t_f]$ might be free to be chosen so that a criterion

$$F_0[t_f, \mathbf{x}(t_f)] + \int_{t_0}^{t_f} f_0[t, \mathbf{x}(t), \mathbf{u}(t)] dt \rightarrow \min_{\mathbf{x}_0, \mathbf{u}(t)}, \quad (5)$$

$$F_0: \mathbb{R} \times \mathbb{R}^{n_x} \mapsto \mathbb{R},$$

$$f_0: \mathbb{R} \times \mathbb{R}^{n_x} \times \mathbb{R}^{n_u} \mapsto \mathbb{R}.$$

is minimized subject to constraints on model inputs $\mathbf{u}_{\min}(t) \leq \mathbf{u}(t) \leq \mathbf{u}_{\max}(t)$ and outputs $\mathbf{y}_{\min}(t) \leq \mathbf{y}(t) \leq \mathbf{y}_{\max}(t), t \in [t_0, t_f]$.

2.5 Discrete-Time Optimal Control Problem

In order to use a digital computer to solve dynamic optimization problems, continuous-time functions have to be discretized. Here multistage control parameterization is applied to formulate dynamic optimization problems as discrete-time optimal control problems.

The time horizon $[t_0, t_f]$ is divided into K stages with $t_0 = t^0 < t^1 < \dots < t^K = t_f$. The controls $\mathbf{u}(t)$ are described in each interval $[t^k, t^{k+1}], k = 0, \dots, K-1$ as function of the discrete-time input variables $\mathbf{u}^k \in \mathbb{R}^m$. The unknown parameters \mathbf{p} are converted to state variables with the state equation $\dot{\mathbf{p}} = \mathbf{0}$ and with unknown initial values $\mathbf{p}_0 = \mathbf{p}(t_0)$. They are described together with the continuous-time model states $\mathbf{x}(t)$ with the discrete-time state variables $\mathbf{x}^k \in \mathbb{R}^n, n = n_x + n_p$. The state equation (1) is solved for the stage k with the initial values \mathbf{x}^k and the controls \mathbf{u}^k using a numerical integration formula.

This results in the multistage optimization problem:

$$F^K(\mathbf{x}^K) + \sum_k f_0^k(\mathbf{x}^k, \mathbf{u}^k) \rightarrow \min_{\mathbf{u}^k, \mathbf{x}_0}, \quad (6)$$

$$F^K: \mathbb{R}^n \mapsto \mathbb{R}^1, f_0^k: \mathbb{R}^n \times \mathbb{R}^m \mapsto \mathbb{R}^1$$

with respect to the discrete-time system equations

$$\mathbf{x}^{k+1} = \mathbf{f}^k(\mathbf{x}^k, \mathbf{u}^k), \quad (7)$$

$$\mathbf{f}^k: \mathbb{R}^n \times \mathbb{R}^m \mapsto \mathbb{R}^n$$

and the additional constraints

$$\begin{aligned} \mathbf{c}_{\min}^k &\leq \mathbf{c}^k(\mathbf{x}^k, \mathbf{u}^k) \leq \mathbf{c}_{\max}^k, \\ \mathbf{c}_{\min}^K &\leq \mathbf{c}^K(\mathbf{x}^K) \leq \mathbf{c}_{\max}^K, \\ \mathbf{c}^k: \mathbb{R}^n \times \mathbb{R}^m &\mapsto \mathbb{R}^{m_k}, \mathbf{c}^K: \mathbb{R}^n \mapsto \mathbb{R}^{m_K}. \end{aligned} \quad (8)$$

Note that initial conditions of the system model are formulated as general constraints (8) as well. Discretization formulae, known parameter values, and predetermined disturbances are included into the discrete-time functions F^K , f_0^k , \mathbf{f}^k , \mathbf{c}^k , and \mathbf{c}^K . The discrete-time functions are assumed to be two times continuously differentiable with respect to their variables.

2.6 Large-Scale Nonlinear Programming Problem

Discrete-time optimal control problems can be treated as structured large-scale nonlinear optimization problems. This has the main advantage that recently developed methods for large-scale nonlinear optimization can be applied to their efficient solution [11, 4].

The discrete-time control and state variables for all stages k are collected to one large vector of optimization variables

$$\mathbf{v} = \begin{pmatrix} \mathbf{x}^0 \\ \mathbf{u}^0 \\ \mathbf{x}^1 \\ \mathbf{u}^1 \\ \vdots \\ \mathbf{x}^{K-1} \\ \mathbf{u}^{K-1} \\ \mathbf{x}^K \end{pmatrix}. \quad (9)$$

One specific feature of the optimization approach discussed here is that the discrete-time state variables at all stages are treated as optimization variables as well, even though they are determined by initial conditions and the control parameters. This leads to a significant increase of the size of the optimization problem. However, the consideration of states as constrained optimization variables generally improves robustness and efficiency of the solution. For instance trajectory constraints can be formulated directly on the discrete-time state variables. Furthermore the separation of the overall problem into multiple stages often leads to a reduction of the required number of nonlinear iterations. The computational overhead is relatively low if the number of state variables n_x is not too high, compared to the number of control variables n_u and if the sparse multistage structure of the large-scale nonlinear optimization problem is exploited appropriately.

3 Solving nonlinear dynamic system models for optimization

Sequential Quadratic Programming (SQP) is generally considered as the most efficient numerical method available nowadays to solve nonlinear optimization problems [12]. This quasi Newton method treats nonlinear optimization problems by solving a sequence of local linear-quadratic approximations. The Lagrangian of the optimization problem is approximated quadratically, typically by applying a numerical update formula. Constraints are approximated linearly.

The differential equations (1) used to model a dynamic system together with the integration formulae determine the equality constraints (7) of the discrete-time optimal control problem. Accordingly the initial value problem

$$\begin{aligned} \dot{\mathbf{x}}(t) &= \mathbf{f}[\mathbf{x}(t), \mathbf{u}(\mathbf{u}^k, t), \mathbf{z}(t), \mathbf{p}(\mathbf{x}^k), t], \\ t &\in [t^k, t^{k+1}], \quad \mathbf{x}(t^k) = \mathbf{I}^k(\mathbf{x}^k), \end{aligned} \quad (10)$$

has to be solved for each stage $k = 0, \dots, K-1$ in each nonlinear optimization iteration to evaluate the discrete-time system functions $\mathbf{f}^k(\mathbf{x}^k, \mathbf{u}^k)$, $k = 0, \dots, K-1$.

Furthermore the discrete-time sensitivities

$$\frac{d\mathbf{f}^k(\mathbf{x}^k, \mathbf{u}^k)}{d(\mathbf{x}^k, \mathbf{u}^k)} \quad (11)$$

are needed to obtain local linear approximations of the nonlinear system model. Often it turns out that the determination of these sensitivities is the most time consuming part when solving dynamic optimization problems.

A straightforward approach for obtaining the sensitivities is to numerically differentiate the system model together with the integration formula. This is normally done by performing multiple initial value simulations for perturbed control variables \mathbf{u}^k and discrete-time states \mathbf{x}^k (e.g. when using Matlab optimization routines together with a Simulink model). However, major drawbacks of this approach are low numerical efficiency and accuracy.

More robust and efficient results can be obtained when solving continuous-time sensitivity equations together with the differential model equations. In approach discussed here the continuous-time sensitivities are needed with respect to the optimization variables

$$\mathbf{q} = \begin{pmatrix} \mathbf{x}^k \\ \mathbf{u}^k \end{pmatrix}. \quad (12)$$

The required sensitivities are

$$\mathbf{s}_i(t) = \frac{d\mathbf{x}(t)}{dq_i}, \quad i = 1, \dots, n + m. \quad (13)$$

They are defined by the sensitivity equations

$$\dot{\mathbf{s}}_i(t) = \frac{\partial \mathbf{f}}{\partial \mathbf{x}(t)} \mathbf{s}_i(t) + \frac{\partial \mathbf{f}}{\partial q_i}, \quad t \in [t^k, t^{k+1}] \quad (14)$$

with the initial conditions

$$\mathbf{s}_i(t^k) = \frac{d\mathbf{I}^k(\mathbf{x}^k)}{dq_i}. \quad (15)$$

See e.g. [9] for an extension of the famous DASSL integration algorithm with sensitivities.

The remaining task is to provide the partial derivatives of the model equations (10) as required by the sensitivity equations (14). They can be obtained with the help of algorithmic, or automatic, differentiation of the model equations [6]. Alternatively the model equations can also be differentiated numerically. This leads to a comparable simpler implementation at the cost of less accuracy and robustness. Good experiences have been made with both: application of algorithmic differentiation using ADOL-C and numerical differentiation of a model implemented as Simulink S-function. It turns out that numeric differentiation of the model equations alone gives more robust results than differentiating the model together with the integration formula numerically. This is especially true for a variable step size integration algorithm that takes different steps in subsequent runs when differentiating model equations and integration formula together numerically.

As the simulation code is generated by a model translation tool from a Modelica specification, one would wish for the future that a Modelica translator like Dymola generates required sensitivity equations together with the model equations. This would considerably simplify the treatment of dynamic optimization problems.

4 Requirements on dynamic system models used for optimization

Especially the exploitation of model sensitivities and the treatment as multistage problem are important for

an efficient solution of dynamic optimization problems. However, both techniques do also imply requirements on the optimized model.

The main advantage of the exploitation of sensitivities is that the superior performance of state-of-the-art nonlinear optimization algorithms can be utilized. This is especially important for problems with a high number of unknown parameters, e.g. to describe a complex control trajectory. However, the model must be smooth with respect to the optimization variables. This means that the values of model variables or their derivatives may not jump (e.g. caused by a state event or by discontinuous functions like absolute value, respectively). From the point of view of optimization, state events have to be formulated as integer variables. This leads to mixed integer nonlinear optimization problems that require a significantly higher solution effort than smooth nonlinear optimization problems. Fortunately in many cases discrete events can be circumvented, e.g. a diode can be modeled ideally utilizing a state event or approximately with a smooth non-linear function. Furthermore it might be sufficient to formulate an optimization problem for a restricted range of the validity of the overall model by introducing constraints on optimization variables. For instance a flow model expressing flow reversal with a state event might be restricted to only exhibit flow into one direction when used in a dynamic optimization.

It is important to note that the model must not be smooth with respect to time. This means that time events, or more generally speaking a sequence of events with fixed switching structure, can easily be incorporated into the dynamic optimization problem. In fact mixed integer nonlinear optimization solvers often exploit this feature and treat a problem with state events on two levels: integer variables are modified on an upper level, while for each set of fixed integer variables the resulting nonlinear optimization problem with fixed switching structure is solved on a lower level.

Besides the exploitation of sensitivities, the treatment as multistage problem offers following advantages:

- improved treatment of state trajectory constraints, because sampled values of the state variables are optimization variables,
- non-linearities do only occur within stages involving only discrete-time variables at specific discrete time points (often leading to a reduction of non-linear iterations),

- the time consuming sensitivity analysis can be performed in parallel for all stages because the initial states for each stage are optimization variables.

The price that has to be paid for these features is that not only sensitivities with respect to the free parameters are required, but also with respect to the initial states of each stage. That is why the number of unconstrained state variables should not be too high, compared to the number of optimized control inputs or model parameters, as otherwise the expensive calculation of sensitivities for these states does not pay off. Fortunately this practical requirement of low model complexity is not specific to dynamic optimization, but is generally known from control applications. If for instance the dynamic optimization shall be performed on-line starting at a transient initial state, the availability of measurement data for estimating the initial state often restricts the allowed model complexity too.

5 Modelica features supporting the formulation of optimization models

One mathematical model can hardly fulfill all requirements that are caused by different applications. That is why it is considered important that a modeling language supports a flexible model management allowing to build different mathematical models describing the same dynamic system depending on requirements by specific applications.

5.1 Separation of model interface and model implementation

A well known object-oriented technique is to separate interface definition and implementation. This technique is also well supported by the object oriented modeling language Modelica. An interface can be defined as partial model:

```
partial model ShellModel
  // interface definitions
end ShellModel;
```

Different implementations can be based on the same interface, e.g. an ideal model with exact switching behavior:

```
model IdealModel
  extends ShellModel;
  // implementation using
  // state events
end IdealModel;
```

and alternatively a smooth model:

```
model SmoothModel
  extends ShellModel;
  // alternative
  // implementation using
  // smooth non-linear function
end SmoothModel;
```

Further implementations can for instance provide models of different complexity, e.g. introducing different numbers of state variables.

Modelica supports the redeclaration of submodels. Exploiting this features, a system model defined for one application, say a real-time simulation, can be adapted to fulfill the requirements of an other application, say a dynamic optimization.

5.2 Model containing multiple implementations

Alternatively to defining different models for different formulations, one model can also provide multiple implementations. One possibility is to use the Modelica built-in operator **analysisType()**:

```
model UniversalModel
  // interface definitions
equation
  if analysisType() == "dynamic"
    // implementation using
    // state events
  else if analysisType() == "linear"
    // implementation using
    // smooth non-linear function
  end;
end UniversalModel;
```

The model translation tool picks out the appropriate implementation depending on the analysis type. Analysis type linear means that the continuous part of the model shall be transformed in a linear system. This implies that the model should be formulated in an appropriate way allowing linearization at given operating points.

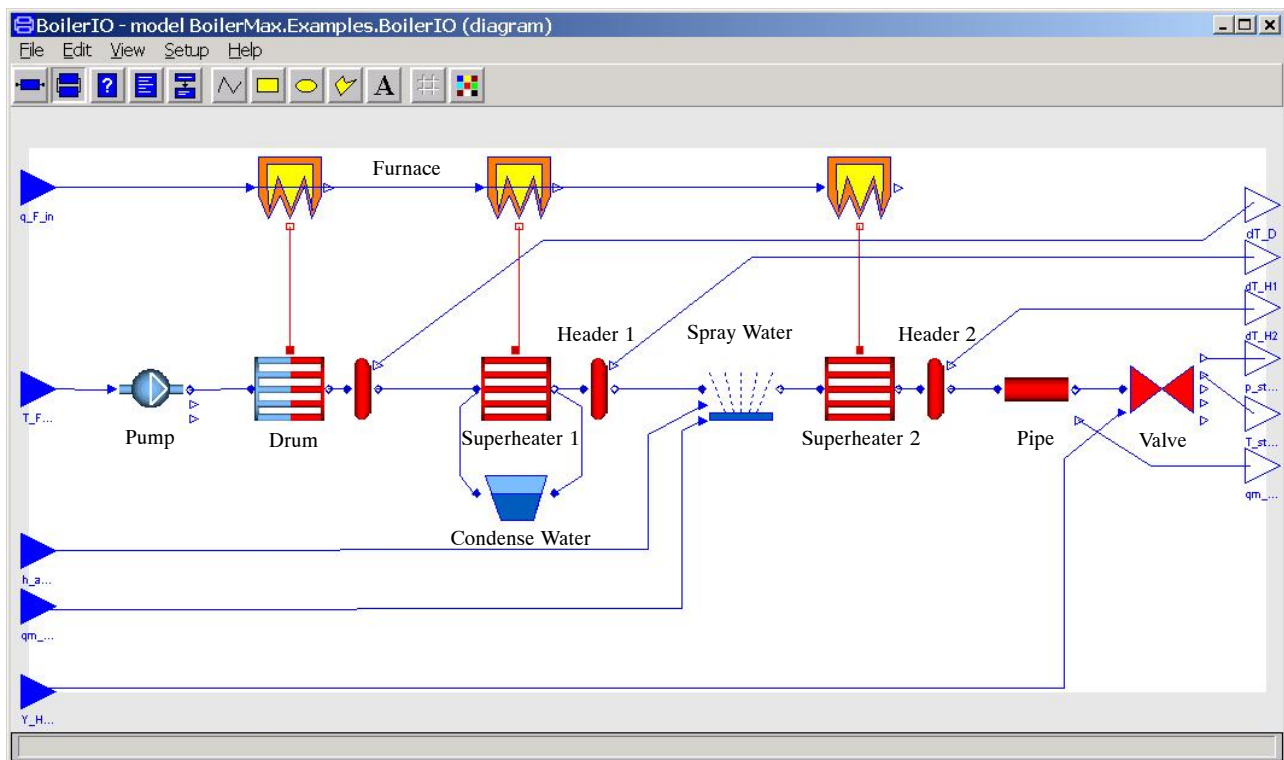


Figure 1: Flowsheet of a boiler model describing the generation of superheated steam.

5.3 Attributes of predefined types

The predefined Modelica type **Real** defines several attributes that are important for the formulation of optimization models. These are:

nominal The nominal attribute should be used to scale optimization variables (control inputs, unknown parameters, model states).

stateSelect This attribute is useful to guide the model translator to select specific states that will become optimization variables.

Furthermore the attributes **min** and **max** could be utilized to formulate bounds on model variables and constraints. However, it should be noted that Modelica is not intended to be an optimization modeling language. The primary intention of the attributes **min** and **max** is to restrict the range a model is valid for, not to define constraints like operational bounds.

Generally Modelica should not be seen as a modeling language to define a whole optimization problem, including optimization criterion and constraints. Instead Modelica is considered a powerful language to define the dynamic system model in a dynamic optimization problem.

6 Example

The optimal startup of a boiler for the generation of superheated steam in a coal fired power plant is discussed as example. The optimal control problem is to obtain a new operating point as fast and efficient as possible considering constraints on the thermal stress on thick walled parts, see [8]. Main new challenges, compared to approaches known so far, e.g. [7], are to formulate a nonlinear dynamic process model that is capable to accurately predict the behaviour over a wide range of operation, including cold start, to be open for flexible adaptation of the model to specific power plants, and to solve the optimal control problem considering constraints on multiple thermal stresses that may become active in different situations.

The process model is formulated in the object-oriented modeling language Modelica. This allows the flexible composition of a process model from sub-models for typical components. Figure 1 shows an example flowsheet. Submodels are a feedwater pump, an evaporator, two superheaters, a long pipe, and a high pressure bypass valve. Further submodels cover the furnace. The phenomenon of condense water is modeled in a separate submodel that is attached to the first superheater. A spray water inlet is placed between the two

superheaters. Thick walled parts are outlet headers of the superheaters and the boiler drum. The model components are based on the ThermoFluid model library [15]. The ThermoFluid library implements, besides others, the IAPWS Industrial Formulation IF 97 standard for the thermodynamic properties of water and steam, enabling accurate and efficient models. The reuse of this model library is considered crucial for an effortable model development concentrating on application specific phenomena.

The implementation of water and steam properties in the ThermoFluid library is accomplished by partial derivatives allowing the flexible selection of state variables, see also the model development in [1]. In the example discussed here, mainly temperatures are selected as state variables, besides pressures and mass flow rates. This simplifies the treatment of constraints on thermal stresses.

Controlled inputs are the fuel flow rate, the amount of spray water, and the position of the outlet valve. Model outputs are pressure, temperature and mass flow rate of generated steam as well as three observed thermal stresses.

The Dymola tool is applied to generate a mathematical system of differential and algebraic equations as required for an efficient numerical solution. After collecting all submodels from the used model libraries, the overall differential-algebraic equation system (DAE) contains 636 variables and equations. This DAE is converted to a system of ordinary differential equations (ODE) with 11 dynamic state variables and is compiled to a Simulink S-function.

Note that the dynamic optimization method discussed here requires the mathematical model in the same form as simulation solvers do. This means that no optimization specific extensions are required to the Dymola model translator. The S-function is directly used to treat dynamic optimization problems, in our case the estimation of model parameters and the optimal boiler startup. Sensitivities are obtained by numerical differentiation of the model.

The optimal boiler startup problem is formulated for 60 time intervals. The control trajectories are parameterized piecewise linear. The resulting large-scale nonlinear optimization problem has 1034 optimization variables, 854 equality constraints, and 1212 inequality constraints. Its solution with the HQP solver takes about 3 minutes on a PC with Pentium III 850 MHz processor.

Figure 2 shows optimization results. The optimization solver has to obtain three trajectories for the controlled inputs so that the optimization criterion is minimized subject to the constraints on thermal stresses and the required new operating point. It can be seen that first the constraints on thermal stress of superheater 2 (dT_{SH2}) and drum (dT_D) are active. Later on, when the condense water has been evaporated, the thermal stress of superheater 1 (dT_{SH1}) is becoming active between 750 s and 1900 s. Generally the constraints are limiting the amount of fuel ($q_{m,F}$) that can be fed into the boiler. Starting from 1500 s, spray water ($q_{m,AW}$) is utilized to reduce the thermal stress on superheater 2. The thermal stress of the drum is becoming active again. The high pressure bypass valve (Y_{HPB}) is primarily used to control the steam flow rate ($q_{m,Steam}$), but it influences other process variables like steam pressure (p_{Steam}) and steam temperatures (T_{Steam}) as well. The required new operating point is reached after about 2500 s.

Such an optimization can be used as core routine of a nonlinear model based controller (NMPC). In this way startup cost savings of about 10% can be reached, compared to a traditional control strategy.

7 Conclusions

The general principle of Modelica of separating the model specification from the numerical solution method allows the reuse of simulation models for optimization. Furthermore, the object-oriented features of the Modelica language and the availability of model libraries greatly simplify the development of rigorous physical models for complex dynamic systems.

Nonlinear dynamic optimization problems can be treated efficiently as discrete-time optimal control problems and solved numerically by applying large-scale nonlinear optimization methods, see also [3]. This is especially true for problems with state constraints. The HQP dynamic optimization solver has been integrated with the Dymola modeling and simulation software using Matlab and Simulink as integration platform [13, 14, 2, 5].

The optimal startup of a power plant is discussed as example. The system model is formulated based on the ThermoFluid model library [15]. The reuse of model libraries is considered crucial for an effortable model development concentrating on specific phenomena of an application.

The example demonstrates the main strengths of model based predictive control: the treatment of multi-input multi-output problems and the consideration of state constraints. For reasons of efficiency, it is important to carefully select state variables during the modeling process. The treatment of state variables as optimization variables simplifies the consideration of state trajectory constraints and allows a more robust and efficient solution of the dynamic optimization problem, even though the problem size increases.

For the future it appears desirable that a model translation tool generates required sensitivity equations in addition to the model differential equations. Model libraries might provide alternative sub-models for specific phenomena, e.g. description of sudden changes with discrete events or with an approximate non-linear function. These sub-models could then be exchanged with each other depending on the intended application and requirements by the solution method.

References

- [1] K.J. Åström and R.D. Bell. Drum-boiler dynamics. *Automatica*, 36:363–378, 2000.
- [2] Dynasim AB. Dymola: Dynamic Modeling Laboratory. <http://www.dynasim.se>.
- [3] R. Franke. *Integrated dynamic modeling and optimization of systems with seasonal heat storage*, volume 394 of *Fortschritt-Berichte VDI, Reihe 6 (in German)*. VDI-Verlag, Düsseldorf, 1998.
- [4] R. Franke and E. Arnold. Applying new numerical algorithms to the solution of discrete-time optimal control problems. In K. Warwick and M. Kárný, editors, *Computer-Intensive Methods in Control and Signal Processing: The Curse of Dimensionality*, pages 105–118. Birkhäuser Verlag, Basel, 1997.
- [5] R. Franke, E. Arnold, and H. Linke. HQP: a solver for nonlinearly constrained large-scale optimization. <http://hqp.sourceforge.net>.
- [6] A. Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, volume 19 of *Frontiers in Applied Mathematics*. SIAM, Philadelphia, 1992.
- [7] P. Kallappa, Michael S. Holmes, and Asok Ray. Life-extending control of fossil fuel power plants. *Automatica*, 33(6):1101–1118, 1997.
- [8] Klaus Krüger, Manfred Rode, and Rüdiger Franke. Optimal control for fast boiler start-up based on a nonlinear model and considering the thermal stress on thick-walled components. In *Proceedings of the IEEE Conference on Control Applications*. Mexico City, September 2001.
- [9] T. Maly and L.R. Petzold. Numerical methods and software for sensitivity analysis of differential-algebraic systems. *Applied Numerical Mathematics*, 20:57–79, 1996.
- [10] Modelica Association. Modelica: Modeling of Complex Physical Systems. <http://www.modelica.org>.
- [11] Walter Murray. Sequential quadratic programming methods for large-scale problems. *Computational Optimization and Applications*, 7(1):127–142, 1997.
- [12] P. Spellucci. *Numerische Verfahren der nicht-linearen Optimierung*. Birkhäuser Verlag, Basel, 1993.
- [13] The MathWorks, Inc. MATLAB: the language of technical computing. <http://www.mathworks.com>.
- [14] The MathWorks, Inc. Simulink: for model-based and system level design. <http://www.mathworks.com>.
- [15] Hubertus Tummescheit, Jonas Eborn, and Falko Jens Wagner. Development of a Modelica base library for modeling of thermo-hydraulic systems. In *Proceedings of the 1st Modelica Workshop 2000*. Lund, Sweden, 2000.

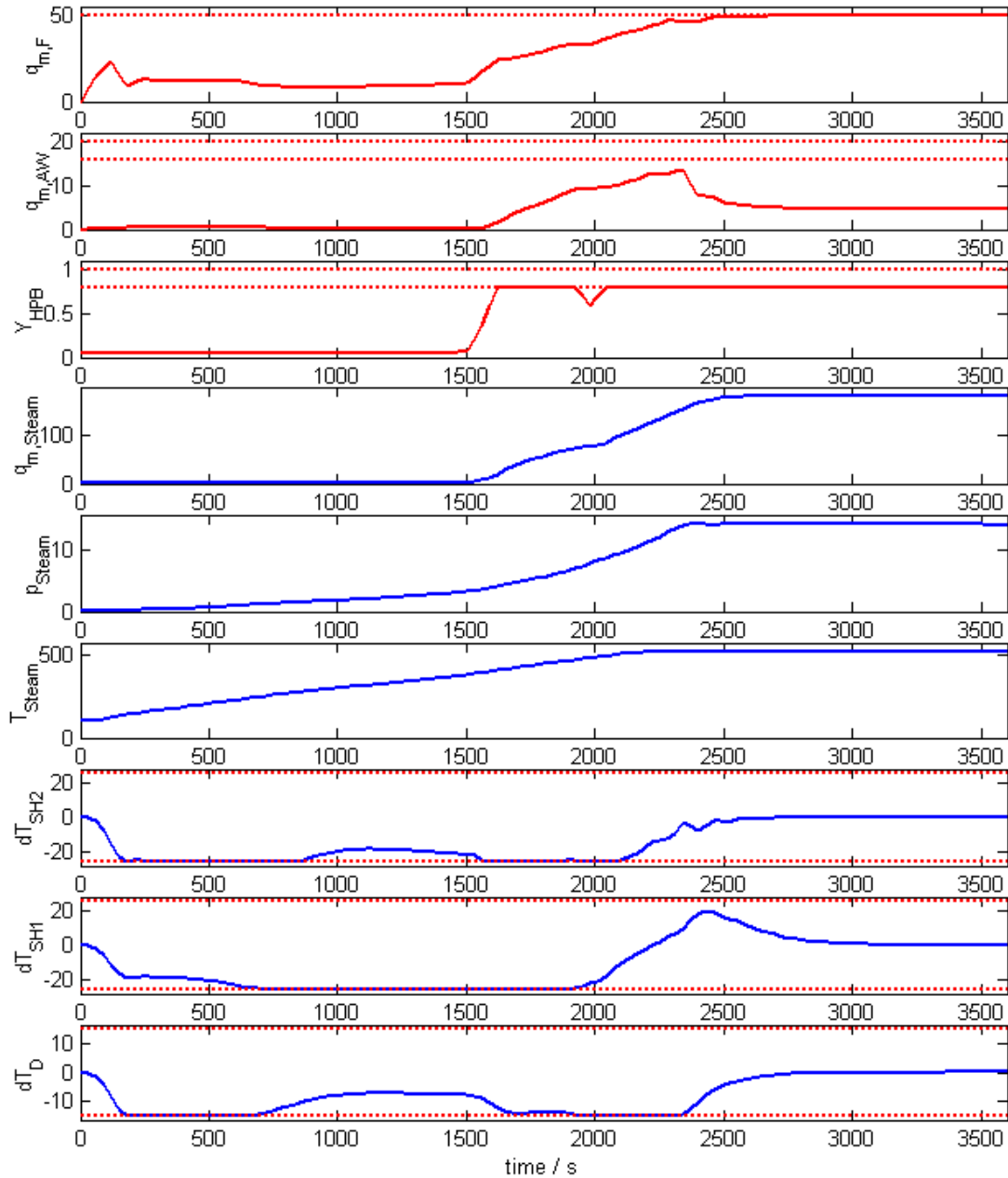


Figure 2: Results for the optimal boiler startup problem. The upper plots show the controlled inputs fuel flow rate $q_{m,F}/\%$, flow rate of spray water $q_{m,AW}/(kg/s)$, and position of high pressure bypass valve Y_{HPB} . Below process variables characterizing the generated steam are plotted: $q_{m,Steam}/kg/s$, p_{Steam}/MPa , $T_{m,Steam}/^{\circ}C$. Furthermore three thermal stresses dT/K are shown.

Estimating parameters in physical models using MathModelica*

Jonas Sjöberg, Fredrik Fyhr and Tomas Grönstedt
 Department of Machine- and Vehicle Systems
 Chalmers University of Technology
 412 96 Gothenburg, Sweden
 Phone +46-31-772 1855, Fax: +46-31-772 3690
 Email: jonas.sjoberg@me.chalmers.se

Abstract

This paper describes a program which extends *MathModelica* so that model parameters can be estimated using measured data. Given initial values of the parameters, the parameter estimates are iteratively changed so that the sum of squared errors of the difference between the model output and the data is minimized. In each iteration an extended differential equation has to be simulated. The developed program imports the Modelica model into *Mathematica* and derives a symbolic expression for this extended differential equation. The extended model is converted to Modelica format and MathModelica is used to simulate it efficiently.

1 Introduction

A mathematical model of a plant can be based on well-known physical laws. These physical laws often contains parameters which numerical values might not be exactly known. There might be, for example, spring constants, masses, resistances and other basic parameters. The program described here has been developed to estimate such parameters using measured signals from the system.

Estimating models of dynamic systems is called system identification. It is a well established engineering research field. Introductory books are, for examples, [5, 3], and more advanced ones [4, 6]. These books, and available software tools, deal with either linear models or discrete time models. For many real world problems there is a need of nonlinear continuous-time models.

There are many reasons to estimate parameters in models built on physical principles. Some examples

follows.

- Some parameters are maybe only approximately known.
- The change of the parameters in the estimation can be used as a way to validate the original model.
- The system might need to be re-tuned after age and wear.
- An online version of the program could be used for monitoring and failure detection of plants in continuous use.

The current program builds on *MathModelica*. The rationale for this is that when the user has obtained a model for simulation, no extra effort is needed to estimate its parameters. This is in opposite to most existing identification tools of today where you have to transform the model into their special format. It is also a question of flexibility, thanks to the great generality of Modelica you can specify almost any type of model. This can either be done by using the Modelica syntax in a *Mathematica* notebook or by using the graphical user interface, Figure 1, where you build a model by combining sub models from different libraries.

The following example illustrates the idea of the program.

Example: Consider the electric circuit in Figure 2. It is easy to build this model with the model editor. The only non-standard part is the resistor which is nonlinear and described by

$$u_R = R_1 i + R_2 i^5$$

where u_R is the voltage, i the current, and R_1 and R_2 are parameters. There are also parameters describing the inductance, L and the capacitor, C .

*Financial support from Volvo Aero Corporation AB is gratefully acknowledged.

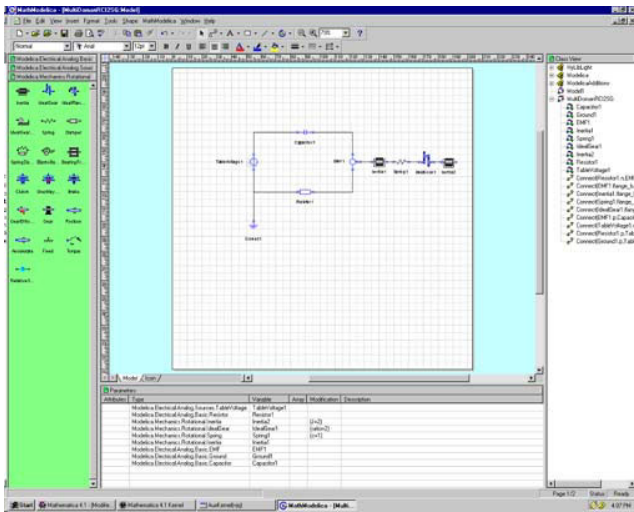


Figure 1: MathModelica model building editor.

Data was obtained by simulating the model with “true” parameter values and with a step voltage of 10 Volt at $t = 0$. The obtained current is displayed in Figure 3. The sampled data values used in the parameter estimation are indicated with dots. The model was initialized

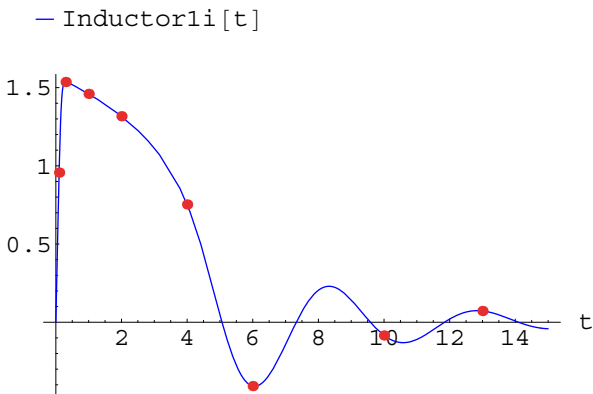


Figure 3: Current through the inductor of the circuit in Figure (2). The marked values are data samples used to estimate the parameters.

with parameter values different from the ones used to obtain the data, as indicated in Table 1. From the initialization the parameters were iteratively improved using the developed program. In Figure 4 the simulated current is shown after each iteration. As seen in the figure, the simulated values coincide with the data after some 6-7 iterations.

Parameter	True value	Initial estimates
R_1	0.5	2
R_2	1	2
C	0.5	2
L	1	2

Table 1: True parameters used to obtain the data and initial parameter values used in the optimization.

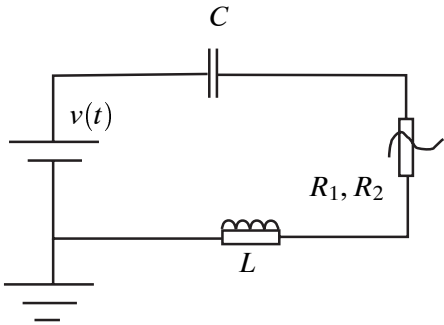


Figure 2: A nonlinear circuit with unknown parameters C , L , and R_1 and R_2 . Voltage over the resistor is described by $R_1i + R_2i^5$

The rest of the paper is organized in the following way. Section 2 gives the mathematical description of the considered system identification problem. How this theory is solved by the program is described in Section 3. Another example is given in Section 4 which is followed by a discussion on possibilities and problems with the current approach in Section 5. The paper is then concluded in Section 6.

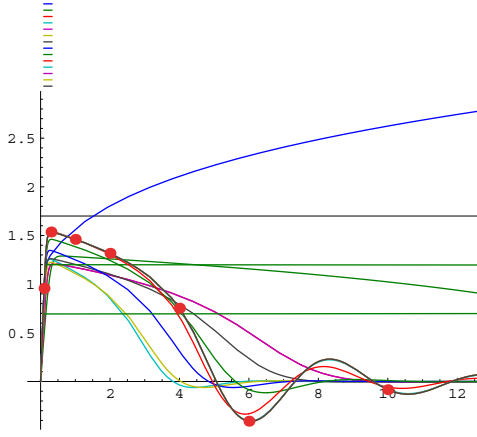


Figure 4: Simulated current after consecutive parameter estimate updates.

2 The equations describing the calculations

Assume that we are interested of a specific system and consider a model of it, described by a differential equation, DAE or ODE

$$f(\dot{x}(t), x(t), u(t), \theta) = 0 \quad (1)$$

$$\hat{y}(t, \theta) = h(x(t), \theta) \quad (2)$$

where $x(t)$ are the states of the model, $u(t)$ is the input signal, $y(t)$ is the output signal. The differential equation is then specified by the functions f and h which also depend on the parameters which are stored in a parameter vector θ .

Assume further that a data set of N samples has been collected from the system, $\{y(t), u(t)\}_{t=1}^N$. The goal is then to tune θ so that the simulated output $\hat{y}(t)$ resembles $y(t)$ when the model is simulated with the input $\{u(t)\}_{t=1}^N$. This is obtained by introducing a criterion of fit. It can be almost any arbitrary differential function, but to keep things easy we choose the mean squared error

$$V_N(\theta) = \frac{1}{N} \sum_{t=1}^N (y(t) - \hat{y}(t, \theta))^2 \quad (3)$$

Then the estimate is defined as

$$\hat{\theta} = \arg \min_{\theta} V_N(\theta) \quad (4)$$

It is generally not possible to find a closed form expression for $\hat{\theta}$. Instead, starting with an initial parameter guess $\hat{\theta}^{(0)}$ the estimate is iteratively computed by

a gradient based algorithm

$$\hat{\theta}^{(i+1)} = \hat{\theta}^{(i)} - \mu R^{-1} \left. \frac{dV_N(\theta)}{d\theta} \right|_{\theta=\hat{\theta}^{(i)}} \quad (5)$$

where R is a positive definite matrix approximating the Hessian, and μ is a step length to assure descent steps. Different standard minimization algorithms, for example Gauss-Newton, Levenberg-Marquardt and steepest-descent, are covered by (5) and they differ on the choice of R . See, eg, [1, 2].

A key part in the iterative minimization (5) is the computation of the derivative of the criterion. It becomes

$$\frac{dV_N(\theta)}{d\theta} = -\frac{2}{N} \sum_{t=1}^N (y(t) - \hat{y}(t, \theta)) \frac{d\hat{y}(t, \theta)}{d\theta} \quad (6)$$

This leads us to the derivative of the model output

$$\frac{d\hat{y}(t, \theta)}{d\theta} = \frac{\partial h(x(t), \theta)}{\partial \theta} + \frac{\partial h(x(t), \theta)}{\partial x(t)} \frac{dx(t)}{d\theta} \quad (7)$$

which cannot be obtained without the signal

$$\tilde{x}(t) = \frac{dx(t)}{d\theta} \quad (8)$$

To obtain this signal we have to take the derivative of the original state space equation in (1). This gives us

$$\begin{aligned} & \frac{\partial f(\dot{x}(t), x(t), u(t), \theta)}{\partial \dot{x}(t)} \frac{d\dot{x}(t)}{d\theta} + \\ & \frac{\partial f(\dot{x}(t), x(t), u(t), \theta)}{\partial x(t)} \frac{dx(t)}{d\theta} + \frac{\partial f(\dot{x}(t), x(t), u(t), \theta)}{\partial \theta} = \\ & \frac{\partial f(\dot{x}(t), x(t), u(t), \theta)}{\partial \dot{x}(t)} \dot{\tilde{x}}(t) + \frac{\partial f(\dot{x}(t), x(t), u(t), \theta)}{\partial x(t)} \tilde{x}(t) + \\ & \frac{\partial f(\dot{x}(t), x(t), u(t), \theta)}{\partial \theta} = 0 \end{aligned} \quad (9)$$

which is a new differential equation. Since it contains $x(t)$ it is coupled with the original differential equation (1) describing the model.

By introducing

$$z(t) = \begin{bmatrix} x(t) \\ \tilde{x}(t) \end{bmatrix} \quad (10)$$

the two coupled differential equations (1) and (9) can be described as

$$F(\dot{z}(t), z(t), u(t), \theta) = 0 \quad (11)$$

where the definition of F follows from (1) and (9).

Hence, to perform the iterative minimization (5) the differential equation (11) has to be simulated in each iteration using the current value of θ .

3 The program

The different parts of the *Mathematica* program can now be described in more detail:

1. Given a Modelica model, describing (1), with initial parameter values and data from the true system.
2. A subset of the parameters are selected for estimation.
3. The extended differential equation (11) is symbolically computed from the original model (1) and transformed into Modelica standard.
4. The extended differential equation (11) is simulated with the current parameter values.
5. The selected parameters are updated (5).
6. Until convergence, go to 4.

The main part of the program is the derivation of the extended model. The other steps consists of interface issues or well-known algorithms which have to be included into the program.

4 Example

A first example was given already in the introduction. Here follows a second one where we have a different type of nonlinear resistor. The system is described in Figure 5. The input to the system is the voltage at the voltage source and the output is the current. The

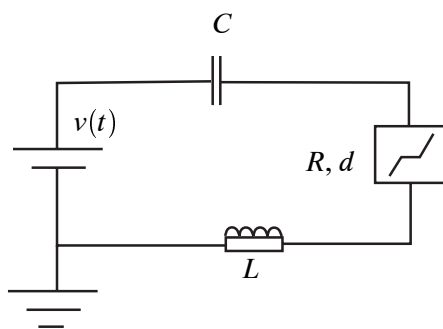


Figure 5: Nonlinear circuit with a parameterized dead zone.

resistor is described by an unknown resistance, R , and a dead zone, d , see Figure 6.

Estimation data was obtained by selecting a set of “true” parameters, given in Table 2 and simulating the model with a step input of 5 Volt. Figure 7 depicts the

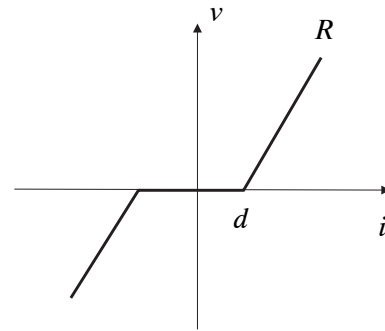


Figure 6: Description of the dead zone parameterization.

voltage over the resistor and one can clearly see the cut-off due to the dead zone.

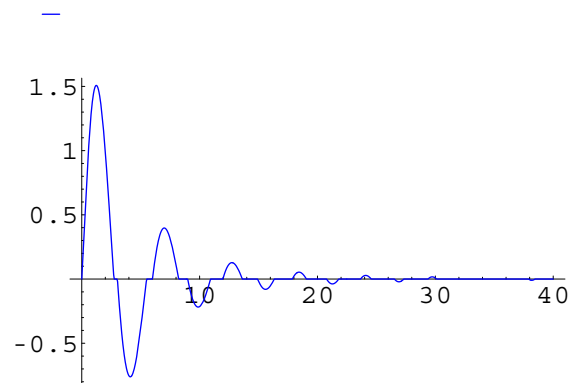


Figure 7: Simulation of the true system, the voltage over the resistor versus time.

The model was initialized with parameter values as indicated in Table 2. The simulation of the initial model together with the estimation data are depicted in Figure 8.

Parameter	True value	Initial value	Final value
R	0.5	0.7	0.503
d	0.4	0.35	0.402
C	0.8	1.1	0.805
L	1	1.1	0.993

Table 2: Parameter values for the circuit with a dead zone in the resistance.

The result of the tuning is illustrated in Figure 9 where the simulated current is depicted after each iteration together with the estimation data. Table 2 gives the final parameter values. From the figure it is clear that

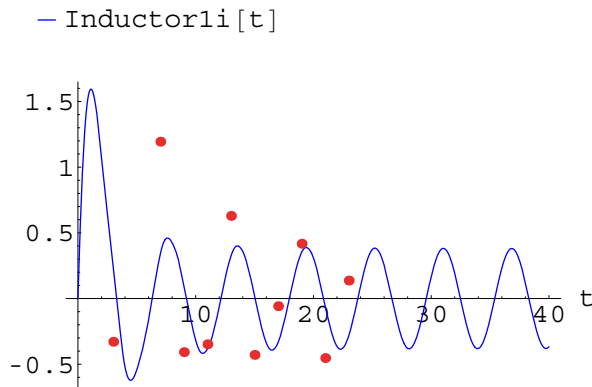


Figure 8: Simulation of the model with the initial parameters together with estimation data.

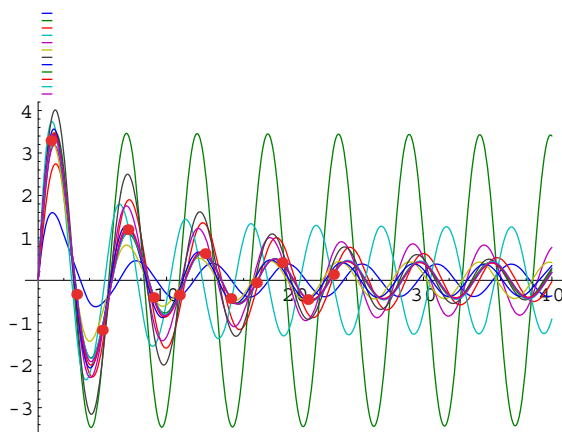


Figure 9: Simulated current after consecutive parameter estimate updates.

the parameters converge within some 10 iterations.

5 Discussion

Although the program is far from ready the following functionality is supported, or can easily be supported by making smaller changes.

- Data can be sampled at irregular sampling instances. Different signals can be sampled individually.
- Systems described by DAE can be handled in the same (automatic) way as ODE systems. The example in the introduction was actually a DAE example.
- Discontinuous (but piecewise smooth) differential equations can be handled, at least a formal result can be obtained.
- Multiple input multiple output systems are handled.
- Criterion of fit can be changed.

There are also potential problems.

- The gradient search can only guarantee convergence to a local minimum. Hence, a good initial parameter guess is necessary to obtain convergence to the global minimum.
- The order of the extended differential equation is often high, it becomes the number of parameters times the number of states in the original differential equation. This gives a high computational burden which might limit the applicability of the program.
- Stability problems may occur. Depending on the parameter values the differential equations might be stable or unstable. In the general case, where the model is nonlinear, it is not possible to monitor stability.

6 Conclusions

A *Mathematica* program has been developed which extends MathModelica so that parameters can be estimated using measured data. The program builds on the following principles.

- An existing modeling tool, the MathModelica graphical user interface, is used to describe the model.
- Mathematica is used to create the to the model specific extended differential equation which needs to be simulated in the estimation process.
- Existing, efficient numerical differential equation solver is used to simulate the extended differential equation.

So far only preliminary studies have been carried out. More experience is needed and the program has to be developed further before it becomes as easy to use, as it is supposed to be.

References

- [1] J.E. Dennis and R.B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Prentice-Hall, Englewood Cliffs, New Jersey, 1983.
- [2] R. Fletcher. *Practical Methods of Optimization*. John Wiley & Sons, 1987.
- [3] R. Johansson. *System Modeling & Identification*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632, 1993.
- [4] L. Ljung. *System Identification: Theory for the User*. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition, 1999.
- [5] L. Ljung and T. Glad. *Modeling of Dynamic Systems*. Prentice Hall, Englewood Cliffs, N.J., 1994.
- [6] T. Söderström and P. Stoica. *System Identification*. Prentice-Hall International, Hemel Hempstead, Hertfordshire, 1989.

Multiprocessor Scheduling of Simulation Code From Modelica Models

Peter Aronsson, Peter Fritzson
PELAB, The Programming Environment lab
Department of Computer and Information Science
Linköping University
SE-581 83 Linköping, Sweden
{petar,petfr}@ida.liu.se

Abstract

Modern object oriented modeling techniques, such as the Modelica modeling language, are increasing the capability to model and simulate systems of large size and complexity. Simulation of such large and complex systems is computationally very expensive. The use of parallel computers for simulation of Modelica models is one approach of handling simulation of such large and complex systems within reasonable time limits. This paper presents an automatic parallelization tool that translates the sequential simulation code generated from a Modelica compiler, Dymola, into a parallel version that can be executed on a parallel computer. The paper also presents several scheduling and clustering techniques used by the tool to partition the simulation code onto several processors. One of these techniques, called FTDT-Full Task Duplication Technique, gives a measured speedup of 2.5 on an 8 processor PC-cluster. However, future work includes developing better scheduling and clustering algorithms to further improve the results.

1 Summary

Object oriented equation based modeling languages such as Modelica enable simulation of large complex systems. However, with growing complexity of modeled systems, the need for parallelization becomes increasingly important in order to keep simulation time within reasonable limits.

The first step in a Modelica compilation results in a system of differential and algebraic equations. The Modelica compiler typically performs optimizations on this system of equations to reduce its size. Other optimizations on the equation system are also performed to for instance reduce the index of the system

to make it easier to solve numerically, and break algebraic loops to enable generation of more efficient code. Finally, sequential C code is generated from the optimized set of equations, containing assignment statements with arithmetic expressions, function calls, and subsystems of equations that are solved using a variety of solution techniques. This simulation code is then combined with a numerical solver to simulate the model.

This paper presents an automatic parallelization method and tool that builds a task graph from the optimized sequential code produced by the Dymola commercial Modelica compiler. Earlier work indicated that the task graph should be built at the expression level, resulting in a large fine grained task graph. The reason for building the task graph at the lowest level is to reveal all possible parallelism in the task graph. The fine granularity of the task graph means that the communication costs between tasks in the graph are typically much larger than the execution costs of the tasks. Hence, the scheduling algorithms need to take this into consideration to be able to produce an efficient parallel schedule of the task graph.

Several scheduling algorithms have been studied and implemented for this problem, like the TDS algorithm, which is a task duplication based scheduling algorithm. We have also investigated clustering algorithms which have the goal of clustering nodes for better computation/communication tradeoff. However the standard algorithms found in the literature gave poor result due to the special properties of the tasks graphs generated from the optimized equations converted to C-code emitted by the Dymola Modelica compiler.

There are some scheduling algorithms, specially designed for targeting simulation code, like for instance the algorithm presented in [21]. However, that algorithm is not suitable for our purposes since it is mainly

designed for coarse grained task graphs and can not handle fine grained task graphs well. The reason for obtaining good speedup in that case is that the used architecture had a reasonable fast communication network compared to the slow processor speed. However, this relation between communication speed and processor speed is not valid today, and is in fact degrading in the future, since the processor speed is increasing much faster than the communication speed.

Yet another approach, where task duplication is always used, FTDT - Full Task Duplication Technique, shows speedup results for some examples, including a thermofluid pipe model which gives a speedup of about 2.5 on 8 processors running on a PC-cluster.

Future work include designing and developing better clustering and scheduling algorithms well suited for the simulation code generated from optimized systems of equations.

2 Introduction

Modelica is an acausal, object-oriented, equation based modeling language for modeling and simulation of large and complex multi-domain systems [14, 8]. Modelica was designed by an international team of researchers, whose joint effort has resulted in a general language for design of models of physical multi-domain systems. Modelica has influences from a number of earlier object oriented modeling languages, for instance Dymola [7] and ObjectMath [9].

A Modelica compiler flattens the object oriented structure of the model into a system of differential algebraic equations (DAE) which during simulation is solved using a standard DAE solver. This code is often very time consuming to execute, and there is a great need for parallel execution, especially for demanding applications like hardware-in-the-loop simulation.

The flat set of equations produced by a Modelica compiler is typically sparse, and there is a large opportunity for optimization. A simulation tool with support for the Modelica language would typically perform optimizations on the equation set to reduce the number of equations. One such tool is Dymola [6], another is MathModelica [13].

The problem presented in this paper is to parallelize the calculation of the states (the state variables and their derivatives) in each time step of the solver. The code for this calculation consists of assignments of numerical expressions, e.g. addition or multiplication operations, to different variables. But it can also contain function calls, for instance to solve an equation system

or to compute $\sin(x)$ for a value x , which are computationally more heavy tasks. The Dymola simulation tool produces this kind of code. Hence we can use Dymola as a front end for our automatic parallelization tool.

To parallelize the simulation we first build a task graph, $G = (V, E)$ where each task $v \in V$ corresponds to a simple binary operation, or a function call. A data dependency edge ($e \in E$) is present between two task nodes v_1, v_2 iff v_2 uses the result from v_1 . This is represented in the task graph by the edge $e = (v_1, v_2)$. Each task is assigned an execution cost which corresponds to a normalized execution time of the task, and each edge is assigned a communication cost corresponding to a normalized communication time between the tasks if they execute on different processors. Figure 1 illustrates how a task graph can be represented graphically. Each node is divided by a horizontal line. Above the line a unique task label/number is given and below the line is the execution cost of the task. Near each edge is the communication cost for the edge given. The goal of a scheduling or clustering algorithm is to minimize the execution time of the parallel program. This often means that the communication between processors must be kept low, since interprocessor communication is very expensive. When two tasks execute on the same processor, the communication cost between them is reduced to zero.

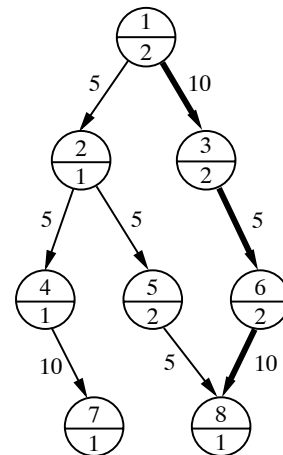


Figure 1: Task graph with communication and execution costs.

Scheduling and partitioning of task graphs of the kind described above have been studied thoroughly in the past three decades. There exist a plethora of different scheduling and partitioning algorithms in the literature for different kinds of task graphs, considering different aspects of the scheduling problem. The

general problem of scheduling task graphs for a multiprocessor system has been proven to be NP complete [15].

The rest of the paper is organized as follows: Section 3 gives a short summary of related work. Section 4 presents our contribution to parallelizing simulation code. In section 5 we give some results of our contribution, followed by a discussion and future work in section 6.

3 Related Work

A large number of scheduling and partitioning algorithms have been presented in the literature. Some of them use list scheduling techniques and heuristics [3, 11, 5, 2, 10, 22]. A list scheduler keeps a list of tasks that are ready to be scheduled, i.e. all its predecessors have already been scheduled. In each step it selects one of the tasks in the list, by some heuristic, and assigns it to a suitable processor, and updates the list.

Another technique is called critical path scheduling [17]. The critical path of a task graph (DAG) is the path having the largest sum of communication and execution cost. The algorithm calculates the critical path, extracts it from the task graph and assign it to a processor. After this operation, a new critical path is found in the remaining task graph, which is then scheduled to the next processor, and so on. One property of critical path scheduling algorithms is that the number of available processors is assumed to be unbounded, because of the nature of the algorithm.

Yet another approach to scheduling of task graphs is to first apply a task clustering algorithm and thereafter schedule the clusters for a fixed number of processors. A task clustering algorithm results in a cluster partition of the task graph. A cluster is a set of nodes designated to execute on the same processor. Thus, the communication costs for edges between nodes belonging to the same cluster are reduced to zero. A low complexity task clustering algorithm is the DSC algorithm [19]. It has a complexity of $O(n \cdot \log(n))$.

An alternative approach to task clustering is to apply task merging algorithms [12]. The difference between a task clustering algorithm and a task merging algorithm is that in the task clustering case, tasks are not merged, i.e. the communication of data is still performed for each individual task in the cluster. But for the task merging case, the tasks are merged such that the new task resulting from the merge receives all its data *before* the computational work of the task, and

sends all the resulting data to other tasks *after* the computational work has been performed.

Due to the merging property of a task merging algorithm, the resulting task graph will have a higher granularity value, i.e. the communication to computation ratio will increase. Thus, after a task merging algorithm has been applied any standard scheduling algorithm that works better for coarse grained task graphs can successfully be applied.

An orthogonal feature in scheduling algorithms is task duplication. Task duplication scheduling algorithms rely on task duplication as a mean of reducing communication cost. However, the decision if a task should be duplicated or not introduces additional complexity to the algorithm, pushing the complexity up in the range $O(n^3)$ to $O(n^4)$ for task graphs with n nodes.

4 Scheduling of Simulation Code

An overview of the automatic parallelization tool presented in this paper is given in Figure 2. The figure illustrates both how the sequential executable and the parallel executable that are built.

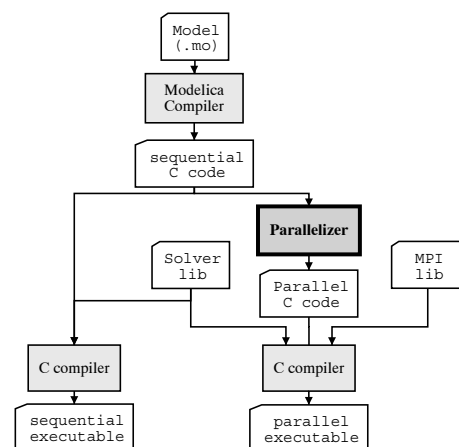


Figure 2: An overview of the parallelization tool and its environment.

Simulation code generated from Modelica mostly consists of a large number of assignments of expressions with arithmetic operations to variables. Some of the variables are needed by the DAE solver to calculate the next state, hence they must be sent to the processor running the solver. Other variables are merely temporary variables whose value can be discarded after the final use.

The simulation code is parsed, and a fine grained task graph is built, see the structure of the tool in Figure 3. The generated graph, which has the properties

of a DAG (Directed Acyclic Graph), can be very large. A typical application (e.g. a thermo-fluid model of a pipe, discretized to 100 pieces), with an integration time of around 10 milli seconds, has a task graph with 30000 nodes. The size of each node can also vary a lot. For instance, when the simulation code originates from a DAE, certain nodes represent an equation system that have to be solved in each iteration if they can not be solved statically at compile time. These equation systems can be linear or non-linear. In the linear case, any standard equation solver could be used, even parallel solvers. In the non-linear case, fixed point iteration is used. In both cases, the solving of each equation system is represented as a single node in the task graph. Such a node can have a large execution time in comparison to other nodes (like an addition or a multiplication of two scalar floating point values).

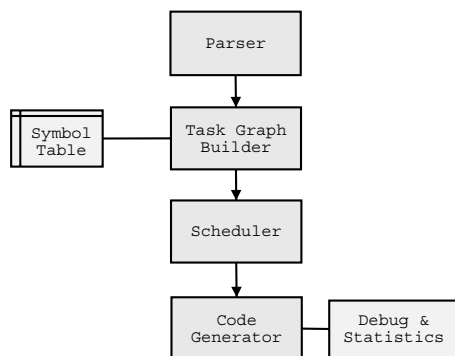


Figure 3: The internal architecture of the parallelization tool.

The task graph generated directly from the simulation code is not suitable for scheduling to multiprocessors. There are several reasons for this, the major reason is that the task graph is too fine grained for applying a standard scheduling algorithm. Many scheduling algorithms are designed for coarse grained task graphs. The granularity of a task graph is the relation between the communication cost between tasks and the execution cost of the tasks. There are several scheduling algorithms that can handle fine grained task graphs as well as coarse grained task graphs. One such category of algorithms is non-linear clustering algorithms [19, 20]. These algorithms consider putting siblings¹ into the same cluster to reduce communication cost. A problem with some of these algorithms is that the complexity is too high for the large task graphs generated by our tool.

¹A sibling s , to a task n is defined as a node where n and s has a common predecessor.

A second problem with the task graphs generated is that in order to keep the task graph small, the implementation does not allow a task to contain several operations. For instance, a task can not contain both a multiplication and a function call. The simulation code can also contain Modelica when statements, which are equivalent to a special form of `if` statements without `else` branch. These need to be considered as one task, since if the condition of the `when` statement is true, all statements included in the `when` statement should be executed. An alternative would be to replicate the guard for each statement in the `when` statement. This is however not implemented yet, since usually the `when` statements are small in size and the need of splitting them up is low.

To solve the problems above, a second task graph is built, with references into the original task graph. The implementation of the second task graph makes it possible to cluster tasks into larger ones, thus increasing the granularity of the task graph. The first task graph is kept, since it is needed later for generating code. The two task graphs are illustrated in Figure 4.

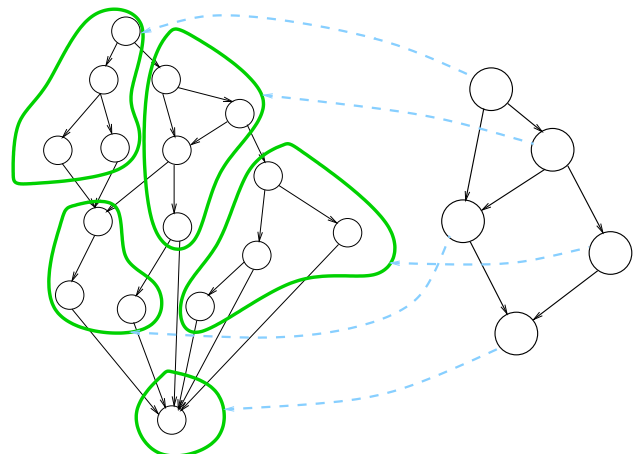


Figure 4: The two task graphs built from the simulation code.

The second level task graph can also be used to cluster tasks together using a task clustering or task merging algorithm. An algorithm for merging task similar to the algorithm found in [16] has been implemented in our tool, except that our algorithm deals with removing cycles. The algorithm constructs a cluster incrementally, starting with a single node n , taken from a list I of all nodes sorted by level in descending order.

The algorithms first examine the children of the node n . When all children have been clustered, the algorithm continues searching for a node that has a child (not in the cluster) with in-degree one and in-

cludes them as well, see figure 5. The next choice of including nodes is the set of parent nodes to the node n . After that, siblings to n are chosen, followed by an arbitrary node in the list of nodes.

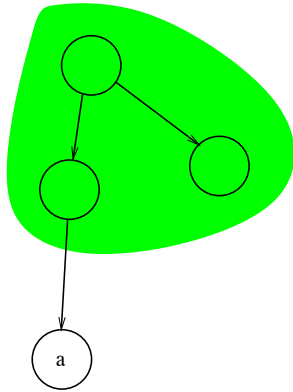


Figure 5: Node a with in-degree one is included in the cluster

To prevent the algorithm from producing cycles in the clustered task graph, a function that detects cycles is called, which includes all nodes forming a cycle from a cluster. Such a cycle is illustrated in Figure 6. The figure shows that by clustering nodes a and b together and not including c in the same task, the resulting task graph will be cyclic, thus removing the property of a DAG making it impossible to schedule.

When a cycle is detected several approaches can be taken. Either the complete cycle is added to the cluster, which can in the worst case make a cluster too large. Another alternative is to remove the node from the cluster causing the cycle, and begin a new cluster. This on the other hand might cause some clusters to be too small.

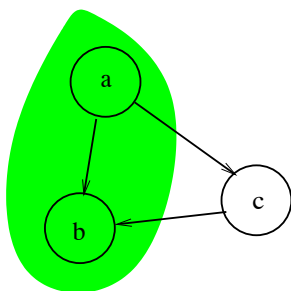


Figure 6: A cluster that forms a cycle in the resulting task graph.

Once the coarse grained task graph is built we can use standard scheduling algorithms found in the literature. In our implementation we have used a schedul-

ing algorithm called TDS [1], which is a critical path scheduling algorithm with task duplication. For coarse grained graphs it produces the optimal solution. However, the number of processors needed by the algorithm is not fixed. Thus, to use the algorithm in practice a phase following the TDS algorithm has to be introduced. This phase limits the number of processors by merging task lists of different processors.

Finally, a simple method called Full Task Duplication Technique (FTDT) is implemented in the parallelization tool. It collects clusters by collecting all parents for each exit node (i.e. a node without any successor) into clusters. These clusters are then merged in a load balancing manner until the number of clusters match the required number of processors. This method is only useful if the task granularity (communication to execution time ratio) is very high, i.e. the average cost of communication is much larger than the average cost of execution in the task graph. However, since it applies *full* duplication it represents an upper limit on the possible speedup for task graph with very high communication costs and can thus be useful in some specific cases.

5 Results

The first results without the pre-clustering (or task merging) phase implemented showed that the TDS algorithm did not work well on fine grained tasks, even with an unlimited number of processors. Most examples did not produce speedups at all. The major reason was that the early implementation did not optimize the sending of messages between tasks, by sending and receiving larger chunks of data. In practice, each scalar produced its own MPI send and receive call.

But also the limitations of the TDS algorithm on fine grained task graphs (i.e. task graphs with high communication costs) had an effect on the result. The TDS algorithm is a linear critical path scheduling algorithm, i.e. it never schedules two siblings onto the same processor. This means that the TDS algorithm exploits all available parallelism, even if the communication cost is very high. Thus it does not work well on graphs with high granularity.

When using the modified task merging algorithm described above, the results were also not showing a speedup > 1 . The main reason for not giving good results in this case was that the task merging algorithm did not succeed in both merging the tasks to increase the granularity and still reveal enough parallelism in the task graph such that speedup > 1 could be ob-

tained. Thus, the resulting task graph contained too many dependencies such that the only possible partition was a sequential partition for one processor.

The Full Task Duplication Technique did however produce reasonable results. Figure 7 gives computed speedup values from the task graphs generated from the simulation code for a thermofluid pipe model with three different discretization values. Figure 8 gives the measured speedup results for the same models when the parallelized simulation code is executed on a parallel PC-cluster with a SCI communication network [18].

Figure 9 presents computed speedup values from the task graph for the robot example (the r3 robot) found in the Modelica Standard Library.

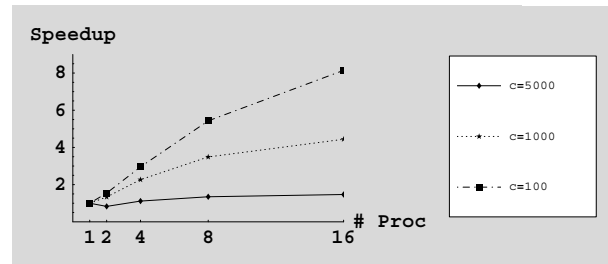
6 Discussion and Future Work

It is clear that the simulation code emitted from compilers of equation based simulation languages can be highly optimized and very irregular code. Hence, this code is not trivial to parallelize. The scheduling algorithms found in the literature are not suitable for fine grained task graphs of the magnitude produced by our tool. Therefore, a pre-clustering phase is needed. Also, the increasingly gap between processor speed and communication speed will demand better clustering and task merging algorithms in order to provide good speedup results in the future.

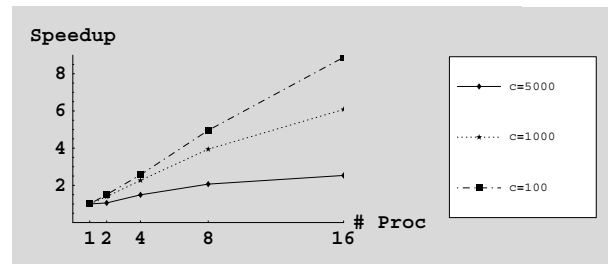
Due to the large task graphs, caused by the large simulation code files, the clustering algorithm must be of low complexity but still use for instance task duplication to reduce communication cost.

The results could be further improved by applying task duplication to the pre-cluster algorithm. Since each task can be very small, extensive task duplication could be used to reduce the communication in the clustered task graph. However, the demand for a low complexity algorithm does not allow an advanced task duplication scheme. Future work is to investigate what kind of task duplication could be considered in the pre-clustering phase. Clearly, it could improve the results significantly.

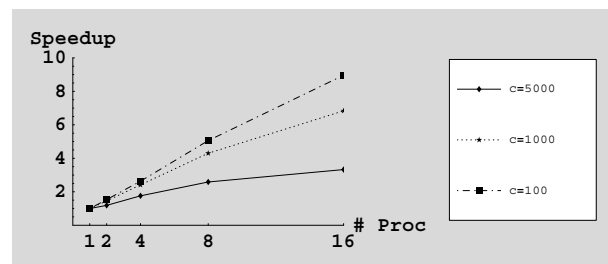
The results from the robot example did not produce good speedup. However, when using mixed mode and inline integration the amount of parallelism in the task graph increased. Therefore, future work includes a deeper investigation on larger models both using mixed mode and inline integration and without those optimizations. Future work also includes an investigation on the effects of different optimizations performed



(a) Thermofluid pipe with 50 discretization points.



(b) Thermofluid pipe with 100 discretization points.

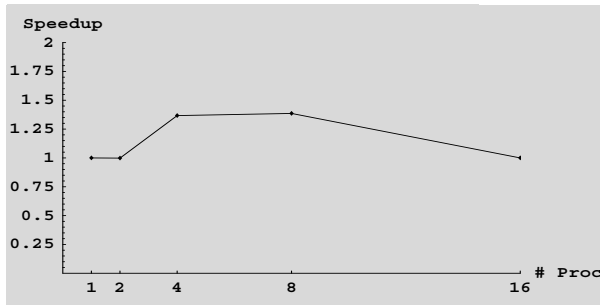


(c) Thermofluid pipe with 150 discretization points.

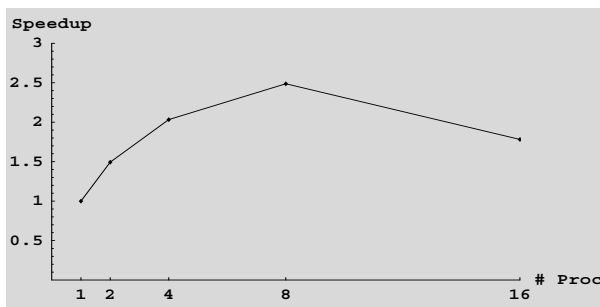
Figure 7: Computed speedup figures for different communication costs c using the FTD method on the Thermofluid pipe.

on the system of equations regarding the amount of parallelism in the simulation code.

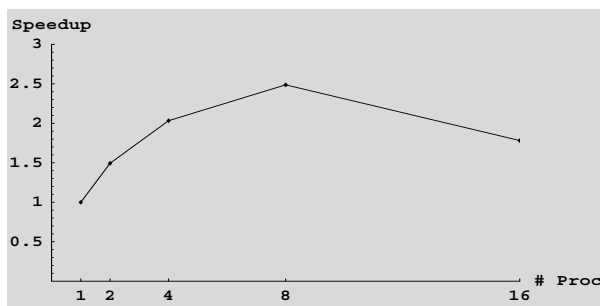
Future work on the scheduling and clustering problem for fine grained task graphs is also needed. Perhaps a more accurate parallel machine model is needed, like for instance the Logp parallel programming model [4]. This would make the model more accurate, giving better estimates of the gained speedup. However, such an extension of the computational model would also increase the complexity of the scheduling and clustering algorithms.



(a) Thermofluid pipe with 50 discretization points.



(b) Thermofluid pipe with 100 discretization points.

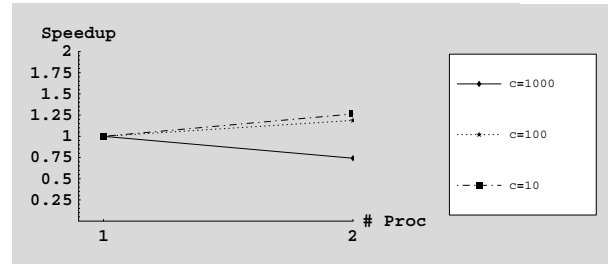


(c) Thermofluid pipe with 150 discretization points.

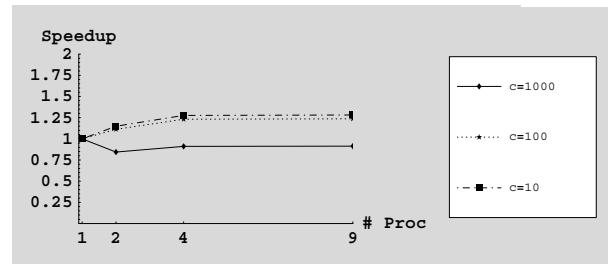
Figure 8: Measured speedup figures when executing on a PC-cluster with SCI network interface using the FTD method on Thermofluid pipe.

7 Acknowledgments

This has been supported by the Modelica Tools project in the Complex Systems framework supported by NUTEK (Swedish Board for Technical Development), and the EU-IST project RealSim.



(a) Mechanical robot model



(b) Mechanical robot model with mixed mode and inline integration

Figure 9: Computed speedup figures for different communication costs, c , using the FTD method on the robot example.

References

- [1] S. Darbha, D. P. Agrawal. Optimal Scheduling Algorithm for Distributed-Memory Machines. *IEEE Transactions on Parallel and Distributed Systems*, vol. 9(no. 1):87–94, January 1998.
- [2] Andrei Radulescu, A. J.C. van Gemund. FLB:Fast Load Balancing for Distributed-Memory Machines. Technical report, Faculty of Information Technology and Systems, Delft University of Technology, 1999.
- [3] C. Hanen, A. Munier. An approximation algorithm for scheduling dependent tasks on m processors with small communication delays. Technical report, Laboratoire Informatique Theorique Et Programmation, Institut Blaise Pascal, Universite P.et M. Curie, 1999.
- [4] David E. Culler, Richard M. Karp, David A. Patterson, Abhijit Sahay, Klaus E. Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. Logp: Towards a realistic model of par-

- allel computation. In *Principles Practice of Parallel Programming*, pages 1–12, 1993.
- [5] C.Y. Lee, J.J. Hwang, Y.C. Chow, F.D. Anger. Multiprocessor Scheduling with Interprocessor Communication Delays. *Operations Research Letters*, vol.7(no. 3), 1988.
- [6] Dymola, <http://www.dynasim.se>.
- [7] H. Elmqvist. *A Structured Model Language for Large Continuous Systems*. PhD thesis, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, 1978.
- [8] P. Fritzson, V. Engelson. Modelica - A Unified Object-Oriented Language for System Modeling and Simulation. In *Proceedings of the 12th European conference on Object-Oriented Programming, LNCS*. Springer Verlag, 1998.
- [9] P. Fritzson, L. Viklund, J. Herber, and D. Fritzson. High-level mathematical modeling and programming. *IEEE Software*, vol. 12(no. 4):77–87, July 1995.
- [10] G. Sih and E. Lee. A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures. *IEEE Transactions on Parallel and Distributed Systems*, vol. 4(no. 2), 1993.
- [11] J.J. Hwang, Y.C. Chow, F.D. Anger, C.Y. Lee. Scheduling Precedence Graphs in Systems with Interprocessor Communication Times. *Journal on Computing*, vol. 18(vol. 2), 1989.
- [12] M. Ayed, J-L Gaudiot. An efficient heuristic for code partitioning. *Parallel Computing*, 26:399–426, 2000.
- [13] MathModelica, <http://www.mathcore.com>.
- [14] *The Modelica Language*, <http://www.modelica.org>.
- [15] R.L. Graham, L.E. Lawler, J.K. Lenstra and A.H. Kan. Optimization an Approximation in Deterministic Sequencing and Scheduling: A Survey. *Annals of Discrete Mathematics*, pages 287–326, 1979.
- [16] S. Chingchit, M. Kumar, L.N. Bhuyan. A flexible Clustering and Scheduling Scheme for Efficient Parallel Computation. In *Proceedings, Parallel and Distributed Processing*, pages 500–505. IEEE Computer, 1999. 12-16 April, San Juan, Puerto Rico.
- [17] S. Darbha, D. P. Agrawal. Optimal Scheduling Algorithm for Distributed-Memory Machines. *Transactions on Parallel and Distributed Systems*, vol. 9(no. 1), 1998.
- [18] Scali - Scalable Linux Systems, <http://www.scali.com>.
- [19] T. Yang, A. Gerasoulis. DSC: Scheduling Parallel Tasks on an Unbounded Number of Processors. *Transactions on Parallel and Distributed Systems*, vol. 5(no. 9), 1994.
- [20] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. MIT Press, Cambridge, MA, 1989.
- [21] B. E. Wells. A Hard Real-Time Static Task Allocation Methodology for Highly-Constrained Message-Passing Environments. *The International Journal of Computers and Their Applications*, II(3), December 1995.
- [22] Wu, M. Y. and Gajski, D. D. Hypertool: A Programming Aid for Message-Passing Systems. *Transactions on Parallel and Distributed Systems*, vol. 1(no. 3), 1990.