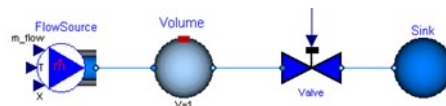
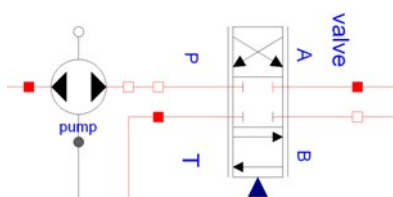
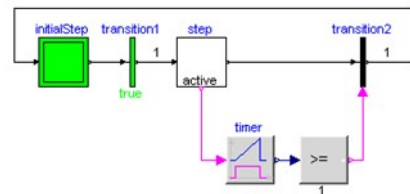
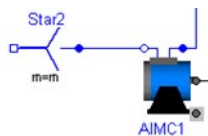
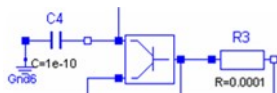
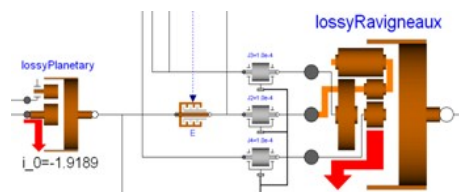
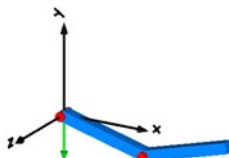
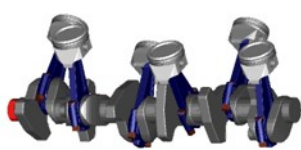


# Modeling, Simulation and Control with Modelica 3.0 and Dymola 7

Preliminary Draft, January 21, 2009

Martin Otter





Copyright © 2006-2009, Martin Otter, all rights reserved.

You are allowed to download the pdf-version of this document, print it for your use, or view it on your computer. All other usages are explicitly forbidden without written prior permission of Martin Otter. Especially, it is forbidden to distribute this document, to copy or print this document for others, to make it publicly available (e.g., on the Web, on a CD-ROM, as printed book), to sell it and/or charge a fee for it.

This document is provided “as is” without warranty of any kind, either expressed or implied. The copyright holder assumes no responsibility for its contents what so ever. Please, send typos and other error reports, as well as improvement suggestions to [Martin.Otter@DLR.de](mailto:Martin.Otter@DLR.de).

Preliminary Draft, January 21, 2009.

Business Address:

Prof. Dr.-Ing. Martin Otter

Deutsches Zentrum für Luft- und Raumfahrt e.V. (DLR)

Institut für Robotik und Mechatronik

D-82234 Wessling Germany

Email: [Martin.Otter@DLR.de](mailto:Martin.Otter@DLR.de)

Web: <http://www.robotic.dlr.de/Martin.Otter>

<http://www.Modelica.org>

In this document the following trademarks are referenced:

- Modelica is a registered trademark of the Modelica Association.
- MATLAB, Simulink, Stateflow and Real-Time Workshop are registered trademarks of The MathWorks Inc.
- CATIA is a registered trademark of Dassault Systèmes.

# Table of Contents

Preface.....	7
<b>Part A Modeling of Continuous Systems.....</b>	<b>11</b>
<b>Chapter 1 Introduction.....</b>	<b>12</b>
1.1 Composition Diagrams.....	12
1.2 Modelica Standard Library.....	14
<b>Chapter 2 Basic Modelica Language Constructs.....</b>	<b>17</b>
2.1 Simple Declarations and Equations.....	17
2.2 Basic Models.....	23
2.3 Inheritance.....	26
2.4 Conditional Equations.....	28
2.5 Hierarchical Models.....	30
2.6 Model Libraries.....	33
2.7 Balanced Models .....	38
<b>Chapter 3 Component Coupling by Ports.....</b>	<b>42</b>
3.1 Connector Design .....	42
3.2 Connectors for Drive Trains (Rotational library) .....	46
3.3 Connectors for Heat Transfer (HeatTransfer library) .....	48
3.4 Connectors for Block Diagrams (Blocks library) .....	51
3.5 Connectors for Signal Buses.....	53
3.6 Hierarchical Connectors.....	57
<b>Chapter 4 Initialization.....</b>	<b>61</b>
4.1 Mathematically defining the initialization problem.....	61
4.2 Initialization with attributes.....	62
4.3 Initialization with initial equations.....	65
4.4 Too many or not enough initial conditions.....	67
4.5 Initialization fails.....	70
<b>Chapter 5 Advanced Modelica Language Constructs.....</b>	<b>75</b>
5.1 Arrays.....	75
5.2 For Loops.....	83
5.3 Algorithm Sections.....	85
5.4 Functions.....	88
5.5 Records.....	100
5.6 External Functions (partially available) .....	105
5.7 Modelica as Scripting Language (not yet available).....	106
5.8 Component Arrays (not yet available).....	106
5.9 Component Coupling by Fields (not yet available).....	106
5.10 Replaceable Models (not yet available).....	106
<b>Part B Modeling of Discontinuous and Variable Structure Systems.....</b>	<b>107</b>
<b>Chapter 6 Discontinuous Equations.....</b>	<b>108</b>
6.1 Relation triggered events.....	109
6.2 Discrete equations and the pre() operator.....	110
<b>Chapter 7 Instantaneous Equations.....</b>	<b>114</b>

7.1 When clauses and mapping to instantaneous equations.....	114
7.2 Sampled data systems and initialization.....	117
7.3 Prioritizing event actions.....	119
7.4 Time synchronization (multi-rate, delays, range-limited sampling).....	121
7.5 Algebraic loops between continuous and instantaneous equations.....	125
<b>Chapter 8 The Synchronous Principle.....</b>	<b>128</b>
<b>Chapter 9 Finite State Machines.....</b>	<b>132</b>
9.1 Implementation of Finite State Machines.....	132
9.2 Event iteration.....	134
<b>Chapter 10 Variable Structure Systems.....</b>	<b>136</b>
10.1 Parameterized Curve Descriptions.....	136
10.2 Coulomb Friction (not yet complete).....	139
<b>Chapter 11 Re-initialization of Continuous Variables.....</b>	<b>143</b>
11.1 The reinit() operator.....	143
11.2 Experimental operators to describe impulsive behavior.....	146
<b>Part C Simulation.....</b>	<b>149</b>
<b>Chapter 12 Symbolic Transformation Algorithms.....</b>	<b>150</b>
12.1 Transformation to State Space Form.....	150
12.2 Sorting (BLT).....	156
12.3 Solving Algebraic Equations (Tearing).....	161
12.4 Singular Systems (Pantelides, Dummy Derivative, not yet available).....	164
12.5 Hybrid DAEs need Symbolic Preprocessing.....	165
<b>Chapter 13 Integration Algorithms.....</b>	<b>169</b>
13.1 Integration Algorithms.....	169
13.2 Method Order (not yet available).....	169
13.3 Stability Region (not yet available).....	169
13.4 Inline Integration (not yet available).....	169
13.5 Event Handling and Chattering (not yet available).....	169
13.6 Practical Considerations (not yet available).....	169
<b>Part D Libraries for Physical Systems.....</b>	<b>171</b>
<b>Chapter 14 Drive Trains (Rotational library).....</b>	<b>172</b>
<b>Chapter 15 Mechanical Systems (MultiBody library).....</b>	<b>173</b>
<b>Chapter 16 Thermo-Fluid Systems (Media/Fluid libraries).....</b>	<b>174</b>
<b>Part E Libraries for Control Systems.....</b>	<b>175</b>
<b>Chapter 17 Continuous and Digital Control Systems (LinearSystems library).....</b>	<b>176</b>
17.1 Basic Mathematical Data Structures.....	176
17.2 Linear System Descriptions.....	178
17.3 Continuous/Discrete Control Blocks.....	182
<b>Chapter 18 Hierarchical State Machines (StateGraph Library).....</b>	<b>190</b>
18.1 Steps and Transitions.....	190
18.2 Conditions and Actions.....	191

18.3 Parallel and Alternative Execution.....	195
18.4 Composite Steps.....	197
18.5 Execution Model.....	199
18.6 Infinite, Unsafe and Unreachable StateGraphs.....	200
<b>Chapter 19 Nonlinear Inverse Models for Advanced Controllers.....</b>	<b>202</b>
19.1 Inversion of Linear SISO Models.....	202
19.2 Inversion of Nonlinear Models.....	207
19.3 Inverse Model in Feedforward Path of Controller.....	210
19.4 Inverse Model in Feedback Path of Controller.....	211
19.5 Inverse Model in Feedback Linearization Controller.....	212
19.6 Inverse Model in Robust Controller (Disturbance Observer).....	213
19.7 Example Application.....	214
19.8 Difficulties with Inverse Models.....	219
<b>Appendix.....</b>	<b>225</b>
<b>Chapter 20 Contributors to Modelica.....</b>	<b>226</b>
<b>Chapter 21 Modelica Built-In Functions and Operators.....</b>	<b>227</b>
21.1 Elementary Operators.....	227
21.2 Mathematical Functions.....	229
21.3 Conversion Functions.....	230
21.4 Derivative and Special Purpose Operators with Function Syntax.....	232
21.5 Event-Related Operators with Function Syntax.....	232
21.6 Array Operators.....	233
<b>Chapter 22 Literature.....</b>	<b>236</b>

## Preface

This document gives an introduction into object-oriented modeling using Modelica, performing simulations of Modelica models with the Modelica simulation environment Dymola (Dynamic Modeling Laboratory), and using Modelica and Dymola for control applications. Modelica is a freely available language to model the dynamic behavior of technical systems based on schematics. The most important language features, Modelica libraries, as well as symbolic algorithms needed to transform the high level description of Modelica into a form that is suited for a numerical integration algorithm are discussed.

Modelica is suited for multi-domain modeling of large, complex, and heterogeneous technical systems, for example,

- mechatronic models in robotics, automotive and aerospace applications involving mechanical, electrical, hydraulic, thermal, and control subsystems,
- process oriented models with multi-phase and multi-substance fluids in pipe networks such as air conditioning systems, batch processes or machine cooling,
- generation, distribution and consumption of electric power.

Modelica is also very well suited for advanced non-linear control systems based on non-linear inverse plant models. Details are discussed in Part E of this document.

Modelica is designed such that it can be utilized in a similar way as an engineer builds a real system: First trying to find standard components like motors, pumps and valves from manufacturers' catalogs with appropriate specifications and interfaces and only if there does not exist a particular subsystem, a component model would be newly constructed based on standardized interfaces.

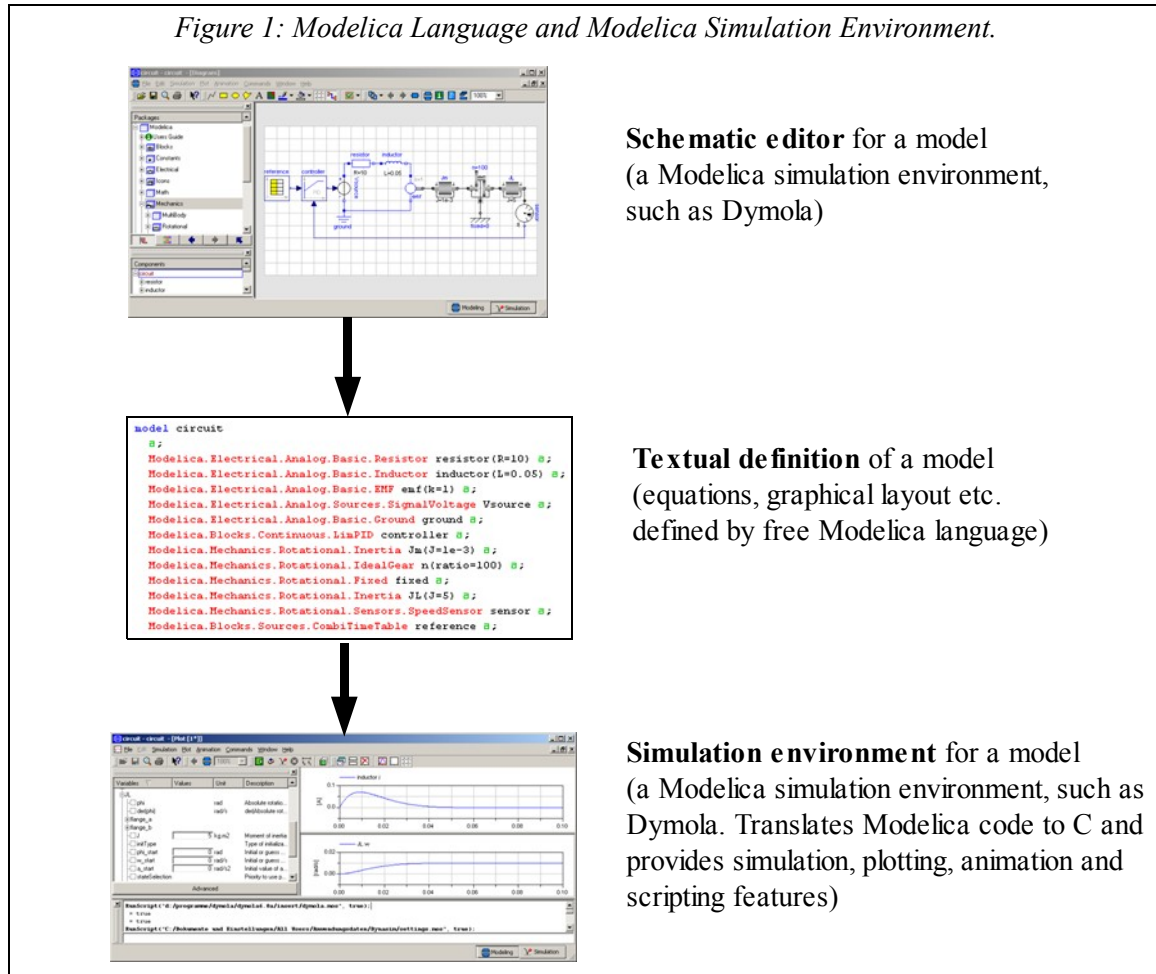
Models in Modelica are mathematically described by *differential*, *algebraic* and *discrete equations*<sup>1</sup> Modelica is designed such that available, specialized algorithms can be utilized to enable efficient handling of large system models having more than hundred thousand equations. It is suited and used for hardware-in-the-loop simulations and for embedded control systems. Modelica is not designed for the direct description of partial differential equations, as, e.g., performed by Finite Element (FE) or Computational Fluid Dynamics (CFD) programs. However, results of finite element computations are utilized in Modelica, e.g., for the description of flexible bodies.

The Modelica language is a *textual specification* that describes details of a model on a high level. In order to be useful, a Modelica modeling and simulation environment is needed, see Figure 1 below, to graphically edit and browse a textually defined Modelica schematic, to transform the model in a form that is better suited for reliable integration, to simulate the model, to visualize the simulation results and to import Modelica models in other simulation environments such as Simulink. A growing number of such Modelica environments are available commercially and also in the "public domain". For an actual overview, see <http://www.-Modelica.org/tools>. In this document, the commercial Modelica environment Dymola is used for demonstration purposes.

Reuse is a key issue for handling complexity. There have been several attempts to define object-oriented languages for modeling of technical systems. However, the ability to reuse and exchange models relies on a standardized format. It was thus important to bring this expertise together to unify concepts and notations: Hilding Elmqvist from Dynasim initiated the Modelica design effort in September 1996, headed it until the Modelica Association was founded in 2000, and is the key architect of the language until today. Still many other people have contributed (see Chapter 20 on page 226 for details): The language and the free Modelica Standard Library have been designed by the developers of the object-oriented modeling languages Allan, Dymola, NMF, ObjectMath, Omola, SIDOPS+, Smile, by computer scientists, and by modeling specialists from the mechanical, electrical, electronic, hydraulic, pneumatic, fluid, and control domain.

<sup>1</sup> "Discrete" equations are either equations that are active at distinct points in time only (e.g. at a sampling instant) or equations that are used to compute Integer or Boolean variables.

Figure 1: Modelica Language and Modelica Simulation Environment.



The non-profit Modelica Association was formed in 2000 with Martin Otter as chairman, to manage the continually evolving Modelica language and the development of the free Modelica Standard Library. In the same year, Modelica was used the first time in actual applications. As of 2009, there are more than 1000 users of Modelica. In June 2006, Dassault Systèmes, a world leader in CAD<sup>2</sup> and PLM<sup>3</sup> announced its new product line CATIA Systems, see <http://www.3ds.com/news-events/press-room/release/1220/1/>. A central part will be behavioral modeling and simulation with Modelica and Dymola. It is to be expected that Dassault Systèmes' strategic decision will further accelerate the growth of the Modelica user community.

More details, especially the actual language specification, free Modelica libraries, downloadable publications, links to Modelica modeling and simulation environments as well as to Modelica consultants can be found at the Modelica homepage <http://www.Modelica.org/>. Also books about Modelica are available, such as (Tiller 2001) and (Fritzson 2003).

This document is written for the following purposes:

- It shall serve as a script for the students of my course "Simulation von elektromechanischen Systemen / Objektorientierte Modellierung mechatronischer Systeme" at the Technical University of Munich. I give this course since 1997 once a year. The content of the course is continually changing from year to year to take into account the latest developments. In the past, about 80 students per year passed the examination for this course.
- It shall provide an in-depth description of the powerful features of Modelica to model discontinuous and variable structure systems. This part of Modelica is described in a condensed form in the Modelica Language Specification and overviews are provided in articles and books. However, a user-oriented description of the details is missing and is provided with this document.
- It shall provide an in-depth description to use Modelica in control applications, such as digital control systems, safe state machines and advanced controllers based on non-linear inverse plant models.

<sup>2</sup> CAD = Computer Aided Design

<sup>3</sup> PLM = Product Life cycle Management



This document does not explain all details of the Modelica language or of the Dymola software. The Modelica language is formally defined in the Modelica Language Specification available at <http://www.modelica.org/documents/ModelicaSpec30.pdf> or at Dymola\Modelica\Documentation\ModelicaSpec30.pdf. Furthermore, the book of Peter Fritzson (Fritzson 2003) gives a very good description and explanation of all the details of the Modelica language. The details of the Dymola software are described in the Dymola manuals available in a Dymola distribution at Dymola\Documentation\Dymola User Manual Volume 1.pdf and Dymola User Manual Volume 2.pdf.

The emphasis of this document is on using Modelica to model and simulate technical systems on the basis of many examples. All examples are provided in Modelica version 3.0 and the Dymola screen shots have been produced with Dymola 7.1 (on Windows XP with the “classic style” option).

Gilching, January 2009

Martin Otter



# Part A Modeling of Continuous Systems

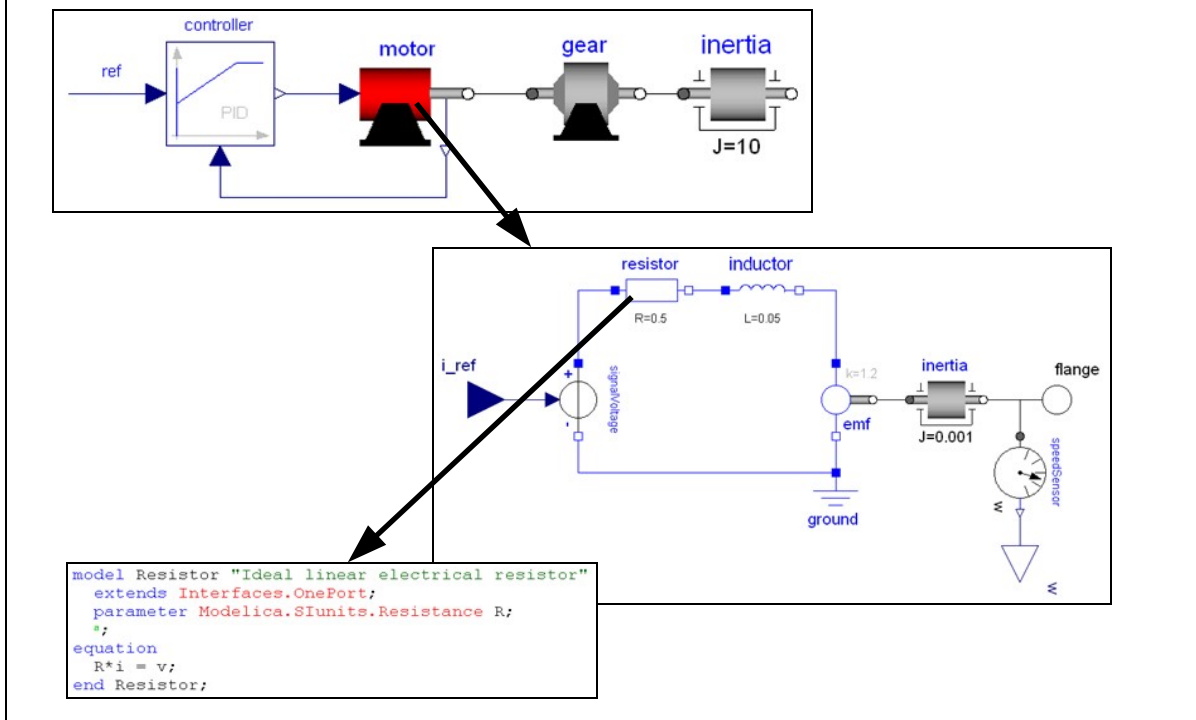
In this part, the most important Modelica language elements are introduced and explained to model continuous physical systems.

# Chapter 1 Introduction

## 1.1 Composition Diagrams

Modelica supports both high level modeling by composition of components and low level modeling by implementing basic components with equations. Models of standard components are typically available in model libraries. Using a graphical model editor from a Modelica environment, a model can be defined by drawing a composition diagram (also called schematics) by positioning icons that represent the models of the components, drawing connections and giving parameter values in dialog boxes. Constructs for including graphical annotations in the Modelica language make icons and composition diagrams portable between different tools. An example of a composition diagram of a simple drive train is shown in Figure 1.1 (the screen shots are from Dymola (Dymola 2008)).

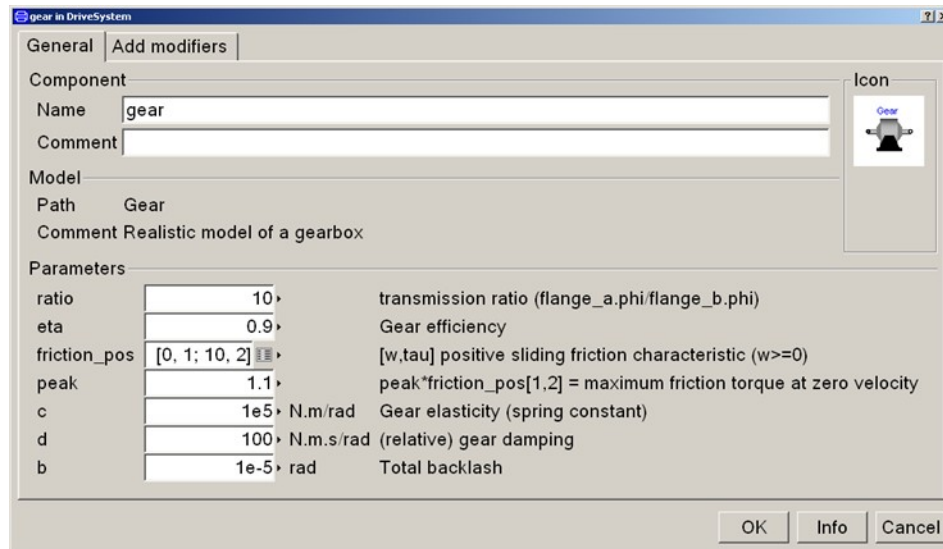
Figure 1.1: Modelica schematic of a simple drive train consisting of a controlled motor, a gear box and a load inertia.



A composition diagram consists of the following elements:

- An **iconic** graphical representation of a physical component, such as an electrical motor, a gear box, a pump or a resistor.
- **Interfaces** that allow the coupling between components. Modelica interfaces are called “connectors”. Variables associated with a connector define the possible interaction with other components.
- Directed or undirected **connection lines** between connectors represent the actual physical connection, e.g., an electrical wire, a rigid mechanical connection, heat transfer, pipe fluid flow, or a signal connection.
- A component, such as the motor or the gear box in the figure above, is either described by another composition diagram (see the electrical circuit diagram of the motor above) or is a basic component that is described by equations in the Modelica language (see the Resistor model above).
- When double clicking on a component, a parameter menu opens and displays the parameter data that can be modified for this component. For example, in Figure 1.2 below the parameter menu of the gear box

Figure 1.2: Parameter menu of gearbox component.

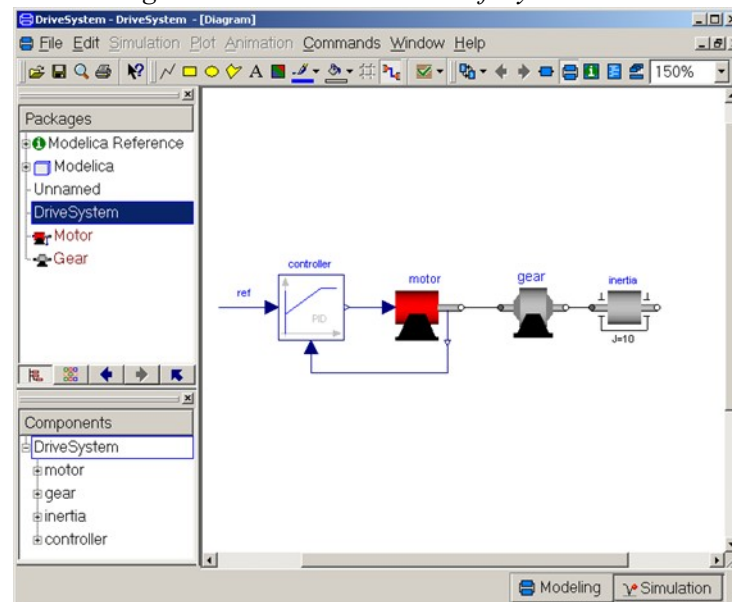


from Figure 1.1 is shown. The left column contains the names of the parameters, the middle column the actual values and the right columns the units and description texts.

- With symbolic algorithms, the high-level Modelica description is transformed into a form that is much better suited for a numerical evaluation.

Based on the natural description of composition diagrams, large, hierarchical systems can be constructed and simulated. In Figure 1.3 below, a typical screen shot of Dymola's schematic editor is shown

Figure 1.3: Schematic Editor of Dymola.



to build up a Modelica model. The editor consists basically of 3 windows: The upper left window is the package browser that shows the loaded Modelica libraries and models. The large right window is the drawing area in which component icons from the package browser are dragged with the mouse to construct the Modelica model. The lower left window (called "Components" above) is the instance browser that shows the hierarchical structure of the model from the main window.

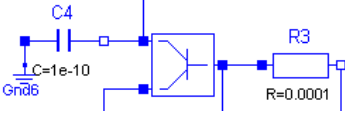
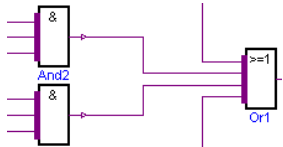
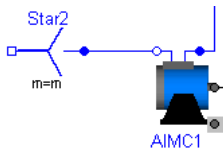
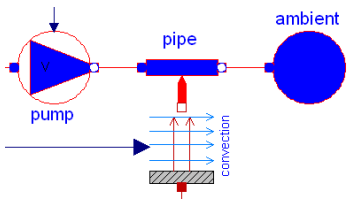
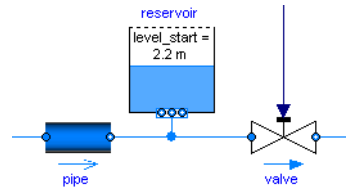
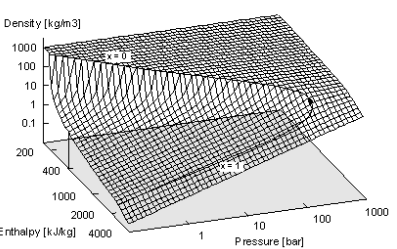
## 1.2 Modelica Standard Library

In order that a modeler can quickly build up system models, it is important that libraries of the most com-

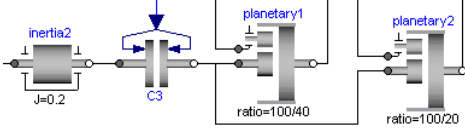
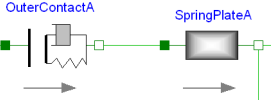
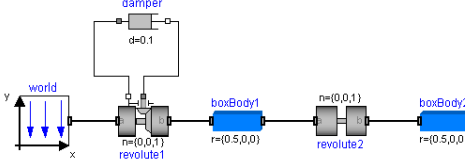
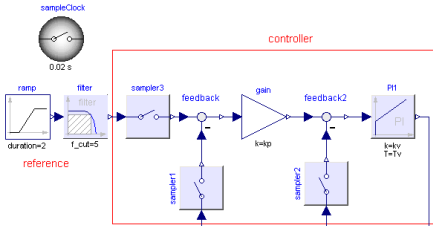
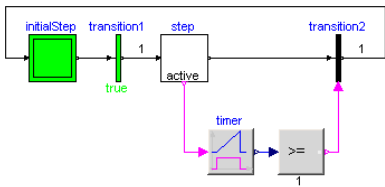
monly used components are available, ready to use, and sharable between applications. For this reason, the Modelica Association develops and maintains a growing *Modelica Standard Library* called *package Modelica*<sup>4</sup>. This is a free library that can be used in open and in commercial Modelica simulation environments. Furthermore, other people and organizations are developing free and commercial Modelica libraries. For information about these libraries and for downloading the free libraries see <http://www.modelica.org/library>.

Version 3.0 of the Modelica Standard Library from January 2008 contains about 780 models, 550 functions and 1200 media descriptions (in form of packages) in the following sublibraries:

Table 1.1: Structure of Modelica Standard Library.

Example system	Name of sublibrary
	<b>Modelica.Electrical.Analog</b> Analog electrical and electronic components such as resistor, capacitor, transformers, diodes, transistors, transmission lines, switches, sources, sensors
	<b>Modelica.Electrical.Digital</b> Digital electrical components based on VHDL with nine valued logic. Contains delays, gates, sources, and converters between 2-, 3-, 4-, and 9-valued logic.
	<b>Modelica.Electrical.Machines</b> Uncontrolled, electrical machines, such as asynchronous, synchronous and direct current motors and generators.
	<b>Modelica.Thermal</b> Simple thermo-fluid pipe flow, especially for machine cooling systems with water or air fluid. Contains pipes, pumps, valves, sensors, sources etc. Furthermore, lumped heat transfer components are present, such as heat capacitor, thermal conductor, convection, body radiation, etc.
	<b>Modelica_Fluid</b> 1-dim. thermo-fluid flow in networks of pipes. A unique feature is that the component equations and the media models are decoupled. Contains, pipes, pressure loss components, valves, pumps, sensors, sources, etc. (will be included in the next main version 3.1 of the Modelica Standard Library).
 E.g. density as a function of pressure and enthalpy	<b>Modelica.Media</b> Large media library for single and multiple substance fluids with one and multiple phases: <ul style="list-style-type: none"> <li>• 1241 high precision gas models + mixtures between these gas models.</li> <li>• Simple and high precision water models (IAPWS/IF97)</li> <li>• Dry and moist air models</li> <li>• Table based incompressible media.</li> </ul>

<sup>4</sup> A library in Modelica is defined with the language keyword “**package**”. It is common in the Modelica community to use the words “library” and “package” interchangeably.

<i>Example system</i>	<i>Name of sublibrary</i>
	<p><b>Modelica.Mechanics.Rotational</b> 1-dim. rotational mechanical systems, such as drive trains, planetary gear. Contains inertia, spring, gear box, bearing friction, clutch, brake, backlash, torque, etc.</p>
	<p><b>Modelica.Mechanics.Translational</b> 1-dim. translational mechanical systems, such as mass, stop, spring, backlash, force</p>
	<p><b>Modelica.Mechanics.MultiBody</b> 3-dim. mechanical systems consisting of joints, bodies, force and sensor elements. Joints can be driven by elements of the Rotational library. Every element has a default animation.</p>
	<p><b>Modelica.Blocks / Modelica_LinearSystems</b> Continuous and discrete input/output blocks. Contains transfer functions, linear state space systems, filter, mathematical, non-linear, logical, table, source blocks. The Modelica_LinearSystems library allows to quickly change between a continuous (= fast controller model) and a sampled description (= detailed controller model).</p>
	<p><b>Modelica.StateGraph</b> Hierarchical state machines with similar modeling power as Statecharts. Modelica is used as synchronous “action” language. Deterministic behavior is guaranteed.</p>
<pre> import Modelica.Math.Matrices; A = [1,2,3;      3,4,5;      2,1,4]; b = {10,22,12}; x = Matrices.solve(A,b); Matrices.eigenValues(A); </pre>	<p><b>Modelica.Math.Matrices / Modelica.Utilities</b> Functions operating on matrices, e.g. to solve linear systems and to compute eigen and singular values. Also functions are provided to operate on strings, streams and files.</p>

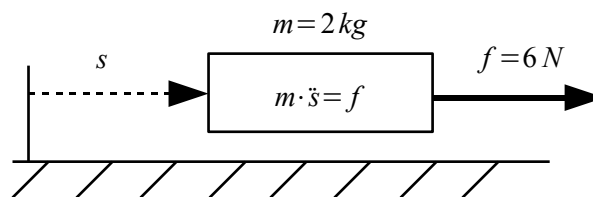
Sublibraries “Modelica\_Fluid” and “Modelica\_LinearSystems” mentioned above will be included in the Modelica Standard Library in one of the next releases. Beta releases are already shipped with Dymola and are available via the “File / Libraries” menu in Dymola.

## Chapter 2 Basic Modelica Language Constructs

### 2.1 Simple Declarations and Equations

Some basic language elements of Modelica are introduced on the basis of the very simple model of a mass that is pulled with a constant force, see Figure 2.1:

Figure 2.1: Mass that is pulled with a constant force.



It would be very easy to model this system using components of the Modelica.Mechanics.Translational library. However, modeling this system directly with Modelica language elements can be done with the following model code:

```
model MovingMass1           "Mass pulled with a constant force";
  parameter Real m=2        "Mass of block";
  parameter Real f=6        "Pulling force";
  Real s                    "Position of block";
  Real v                    "Speed of block";
  annotation (Diagram(Rectangle(extent=<other definitions>)));
equation
    v = der(s);
    m*der(v) = f;
end MovingMass1;
```

Modelica keywords are written in “bold”. A model component is defined within “**model** <Name> ... **end** <Name>”, where <Name> is the user defined model name. A Modelica model consists of sections that are separated by section keywords, such as “**equation**” above. Every variable or component needs to be declared when it is utilized in a model. In the model above, the needed variables  $m$ ,  $f$ ,  $s$ ,  $v$  are declared as real numbers due to the `Real` definition. The keyword **parameter** defines that (a) the corresponding variable is constant and does not change its value during simulation and that (b) this declaration is included in the parameter menu of the model in order that a new value can be provided.

The value of a parameter can be either defined directly at its definition (e.g. “ $m = 2$ ” above), or when using the model through the parameter menu. The above model therefore defines two parameters “ $m$ ” and “ $f$ ” with pre-defined values and two time-varying variables, “ $s$ ” and “ $v$ ”. The order of the declarations does not matter. Especially, a variable can be used before it is declared. The reason is that Modelica models are usually built-up graphically by dragging components and then it would be a large burden, if it would be important in which order elements have to be dragged.

Every Modelica element may have an optional description text enclosed in apostrophes, such as “Mass of block”. Since this description text is provided before the “;” that terminates an element, the text is uniquely associated with the corresponding element. Description texts are used by a Modelica tool for various purposes, e.g., as description text in a parameter menu or as label in a plot window. It is also possible to use comments in C++/Java notation, i.e., either

```
// <until end of line>
```



or

```
/* <text over
   several lines> */
```

Comments are just ignored during parsing of the model.

The “**equation**” keyword separates the declaration section from the equation section. In the equation section, equations between the declared variables are present. Note, these are not assignment statements, as in procedural languages, such as C, C++, or FORTRAN. Instead the general form

```
expression1 = expression2;
```

defines that the expressions on the left and right hand side of the equal sign are mathematically identical. Every equation has to be terminated with a semicolon “;”. The built-in “**der**(...)” operator of Modelica defines the time derivative of the enclosed variable, e.g., **der**(v) = dv/dt. Modelica has only an operator for the first time derivative. In order to define derivatives of higher order, auxiliary variables have to be introduced, such as “v = **der**(s)” in the model above. The notation of expressions is a direct mapping of the “usual” mathematical notation with standard precedence of operators and with parenthesis. The basic operators are “+ - \* / ^ ( )”. For example, the following two descriptions of a polynomial are identical:

```
y1 = (x - 2)*(x + 4);
y2 = x^2 + 2*x - 8;
```

Every model can have optionally “**annotation**(...)” definitions. These definitions do not influence the simulation result and are used, e.g., to define the graphical positioning of the element, the layout of the menu or the documentation.

Modelica supports the following four basic data types:

*Table 2.1: Basic data types of Modelica*

<i>Modelica</i>	<i>Description</i>	<i>Example for literal element</i>
Real	Floating point number (usually about 16 significant digits)	1.0, -2.1234, 1e-14, 1.4e3
Integer	Integer number	1, 2, -4
Boolean	Logical number	<b>false</b> , <b>true</b>
String	String (any number of characters)	"File name"
enumeration	Enumeration	<b>type</b> Extrapolation = <b>enumeration</b> (HoldLastPoint, LastTwoPoints, Periodic);  // Usage: Extrapolation.LastTwoPoints;

We have seen the usage of the “Real” type in the example above. A variable declared as Boolean can have the values “**false**” or “**true**”. A “String” is defined as in “C” by enclosing the string text in double apostrophes. An enumeration is a type with an ordered set of names. An enumeration literal value consists of the name of the enumeration type appended with a dot and the element name.

For engineering applications it is important to support units of the physical quantities. In Modelica, attributes can be defined in a declaration, including the actually used units. The example above is then refined to:

```
model MovingMass2      "Mass pulled with a constant force";
  parameter Real m(unit="kg", min=0) = 2    "Mass of block";
  parameter Real f(unit="N")      = 6    "Pulling force";
  Real s(unit="m")    "Position of block";
  Real v(unit="m/s") "Speed of block";
  annotation(Diagram(Rectangle(extent=<other definitions>)));
equation
```

```

        v = der(s);
    m*der(v) = f;
end MovingMass2;

```

The “unit” attribute defines that the equations are only correct, if the variable values are with respect to the given unit. Additionally, with attribute “min” it is defined, that the variable value cannot be negative. A “unit” is not sufficient to define a physical quantity. This can be easily seen by expressing the units of torque and of energy in the SI base units:

Table 2.2: Torque and energy in SI base units

quantity	unit	in SI base units
Torque	Nm	$\text{kgm}^2/\text{s}^2$
Energy	J	$\text{kgm}^2/\text{s}^2$

Although “Torque” and “Energy” are different physical quantities, they have identical units. In Modelica, several unit-related attributes can be defined:

Table 2.3: Unit related Modelica attributes

quantity	Type of physical quantity
unit	Unit, in which the equations are defined
displayUnit	Unit, that is by default used for input and output

Quantities and units are defined as a Modelica string. The syntax for units is defined by a grammar and follows an ISO recommendation. Examples:

Table 2.4: Modelica syntax for units

Unit	Modelica syntax
$\text{kgm}^2/\text{s}^2$	"kg.m2/s2" or "kg.m.m/(s.s)"
rad/s	"rad/s"
1/s	"1/s" or "s-1"

Besides units, the following attributes can be defined in a declaration:

Table 2.5: Attributes of a Real variable

Attribute	Meaning
min	Minimum value of variable
max	Maximum value of variable
start	Initial value of variable
fixed	= <b>true</b> : Value of “start” is a fixed initial condition = <b>false</b> : Value of “start” is a “guess” value (e.g. for iteration variable)
nominal	Nominal value for scaling of numerical algorithms (e.g., nominal = $1\text{e}5$ means that the “usual” value of the variable is around $10^5$ ).
stateSelect	Influences the choice of the state variable of the differential equations (details are given later). Possible values: StateSelect.never, .avoid, .default, .prefer, .always

A declaration of a Modelica model can therefore look like:

```

parameter Real m(min=0, quantity="mass", unit="kg") = 2;

```

```
Real v(quantity="velocity", unit="m/s", start=3, fixed=true);
```

If this would be the only way to declare attributes such as units, this feature would be not often used, since it is too much work to build such a declaration manually. For this reason, Modelica provides “**type**” classes in order that a modeler can define own types that are derived from the 4 basic types. Examples:

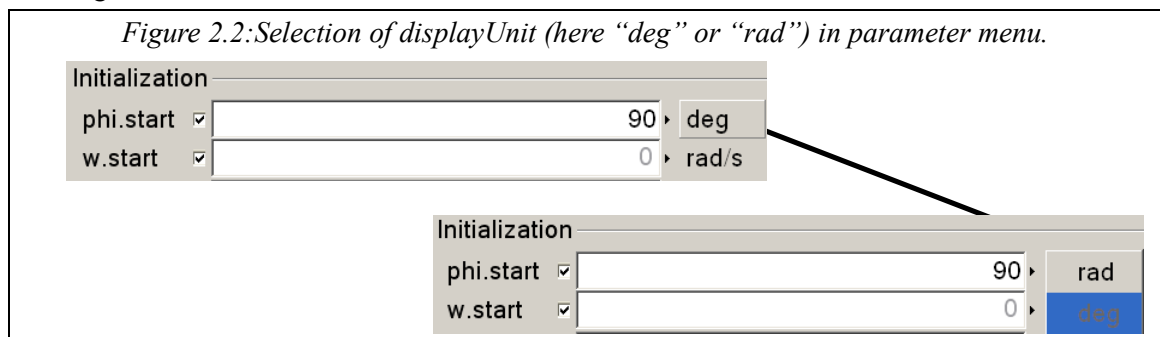
```
type Angle      = Real(final quantity = "Angle", final unit = "rad",
                      displayUnit = "deg");
type Torque     = Real(final quantity = "Torque", final unit="N.m");
type Mass       = Real(final quantity = "Mass" , final unit="kg",
                      min=0);
type Pressure   = Real(final quantity = "Pressure", final unit="Pa",
                      displayUnit = "bar", nominal = 1e5);
```

In parenthesis, every attribute can be defined. If the attribute is prefixed with “**final**”, it cannot be modified anymore. Without the “**final**” attribute, an attribute can be modified when using this type in a declaration. A user defined type class is utilized in the following form:

```
parameter Mass m = 2;
parameter Angle phi = 0.1;
Pressure p (start = 1e6, fixed = true, displayUnit = "mPa");
```

Note, that the “displayUnit” attribute of “Pressure” is modified from “bar” to “mPa”. This is possible since “displayUnit” in the type definition of “Pressure” was *not* prefixed with “**final**”. A Modelica simulation environment may or may not support all of the attributes. In Dymola the unit attributes are utilized in the following way:

- The “displayUnit” attribute is displayed in the parameter menu, the plot-variable list and is used as part of a label in a plot window, if it is defined. Otherwise, the “unit” attribute is displayed. In a parameter menu, the desired “displayUnit” can be selected from a scroll down list. An example is shown in Figure 2.2:



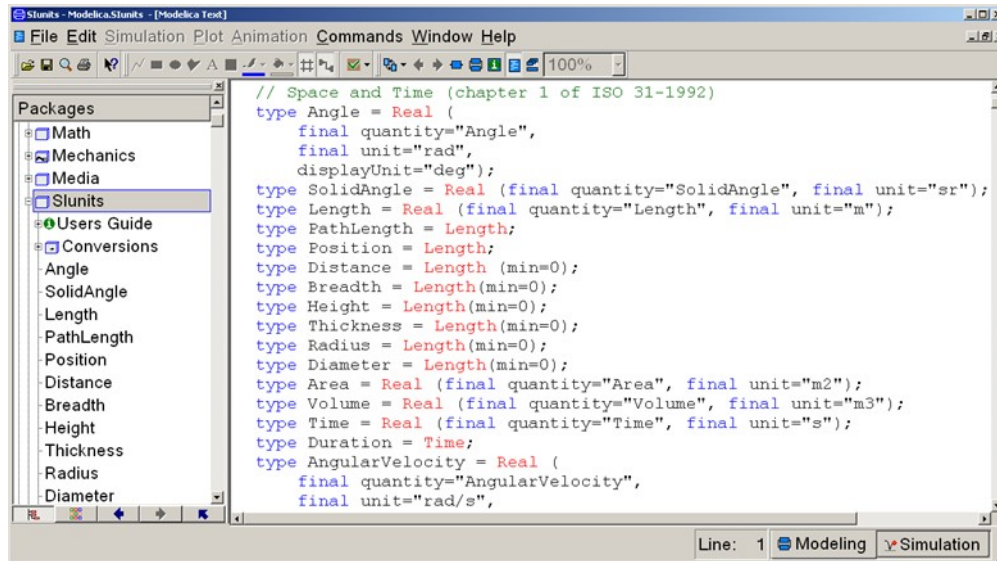
The displayUnit attribute can only be selected, if a literal value (e.g., “90”) is given in the input field. If an expression is given, e.g., a parameter name, the unit defined in the “unit” attribute is used as unit. If this unit is not already shown, it will be shown when closing the parameter menu and opening it again. The possible display units shown in the scroll down list are configured when starting Dymola by executing the script file “Dymola\insert\displayUnit.mos”. The user can modify this file to change the unit scroll down lists.

- When connecting components together, the corresponding connector variables must have the same “quantity” and “unit” definitions provided they are not empty (e.g., a connection between variables is allowed, if one variable has a unit defined and the corresponding one has an empty unit).
- Dymola checks that the variable units are compatible with the usage of the variables in equations. For example, if variables are used in an equation of the form “ $m \cdot a = f$ ”, then the units on the left hand side ( $m \cdot a$ :  $\text{kg} \cdot \text{m} / \text{s}^2$ ) must be identical to the units on the right hand side ( $f$ :  $\text{N} \rightarrow \text{kg} \cdot \text{m} / \text{s}^2$ ) when representing all units with the 7 SI base units (kg, m, s, A, K, mol, cd). If the equation would be written in the form “ $m \cdot v = f$ ”, a warning message reports that the units on the left and right hand side are not compatible to each other and that either the equation or the unit definitions of the variables are erroneous. If units are not defined, Dymola deduces the units from the context and propagates this information. For

example, if a connect equation defines the equation “ $a = b$ ” and “ $a$ ” has a unit, whereas “ $b$ ” does not have a unit defined, then Dymola assumes for the dimension analysis that “ $b$ ” has the unit of “ $a$ ” (= *unit propagation*). The same holds in expressions. For example, “ $c = a + b$ ” and “ $a$ ” has unit “kg”, whereas “ $c$ ” and “ $b$ ” do not have a unit defined, then “ $c$ ” and “ $b$ ” are assumed to have also unit “kg”.

In order to avoid that every modeler defines different sets of derived types, in the Modelica Standard Library about 450 units from ISO 31-1992 “General principles concerning quantities, units and symbols” and ISO 1000-1992 “SI units and recommendations for the use of their multiples and of certain other units” are defined, see Figure 2.3.

Figure 2.3: Predefined unit definitions in package Modelica.SIunits.



There are several possibilities to use the types from the Modelica Standard Library:

With full name:

```
parameter Modelica.SIunits.Mass m = 2;
Modelica.SIunits.Velocity v(start = 3);
```

With shortened name:

```
import Modelica.SIunits;
parameter SIunits.Mass m = 2; // reuse last name of import statement
SIunits.Velocity v(start = 3);
```

With user-defined name:

```
import SI = Modelica.SIunits;
parameter SI.Mass m = 2;
SI.Velocity v(start = 3);
```

With the shortest name by using an unqualified import statement:

```
import Modelica.SIunits.*; // Name before "." must be a package
parameter Mass m = 2; // Mass is searched in all "." imports
Velocity v(start = 3);
```

It is a matter of taste, which of the variants to use. In the Modelica Standard Library, often the third variant with a user-defined name is present.

We are now in the position to show the recommended implementation of the simple example from above:

```
model MovingMass3 "Mass pulled with force (recommended implement.)"
import SI = Modelica.SIunits;
parameter SI.Mass m = 2 "Mass of block";
```

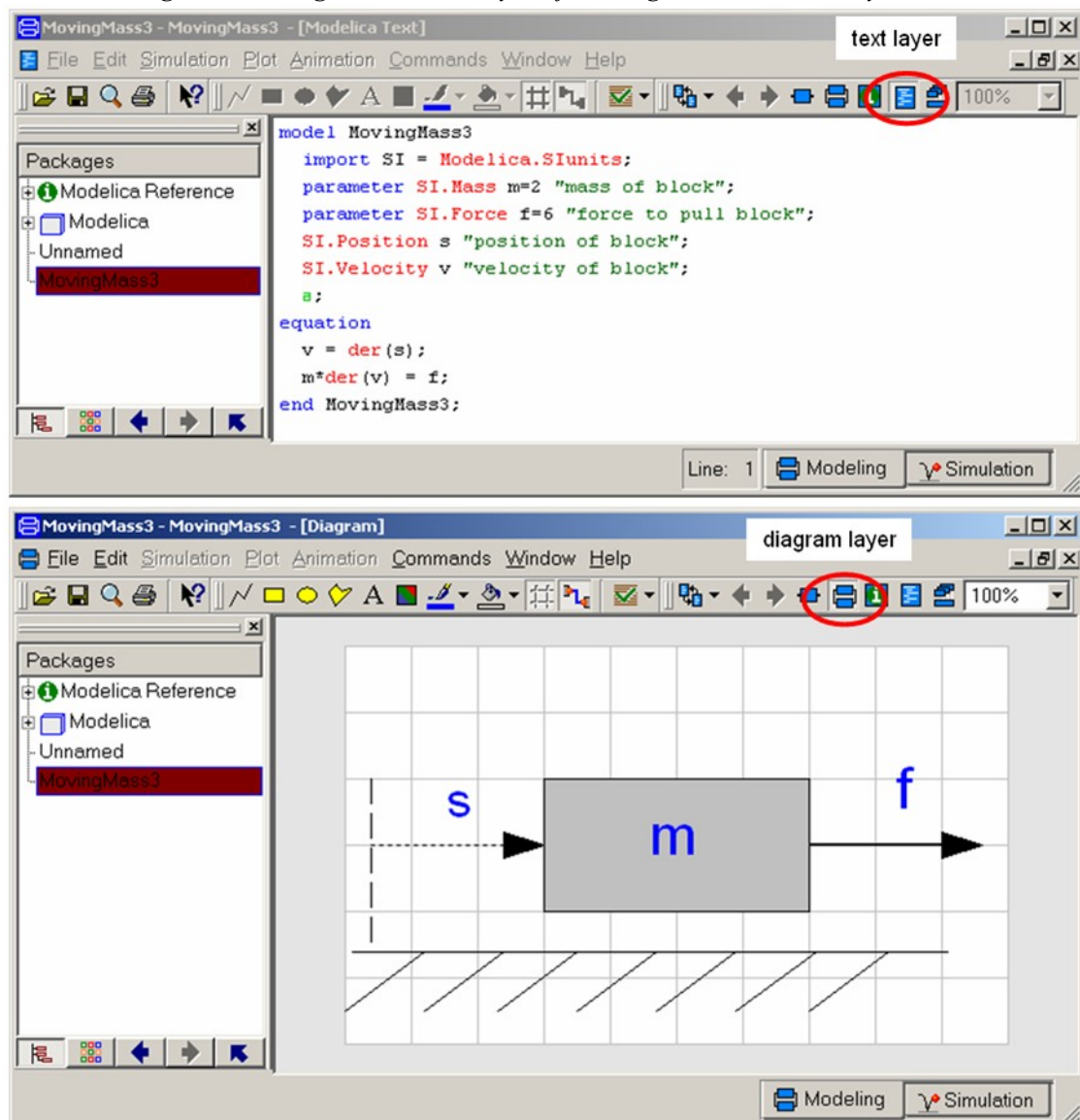
```

parameter SI.Force f = 6    "Pulling force";
SI.Position s    "Position of block";
SI.Velocity v    "Speed of block";
annotation(Diagram(Rectangle(extent=<other definitions>)));
equation
    v = der(s);
    m*der(v) = f;
end MovingMass3;

```

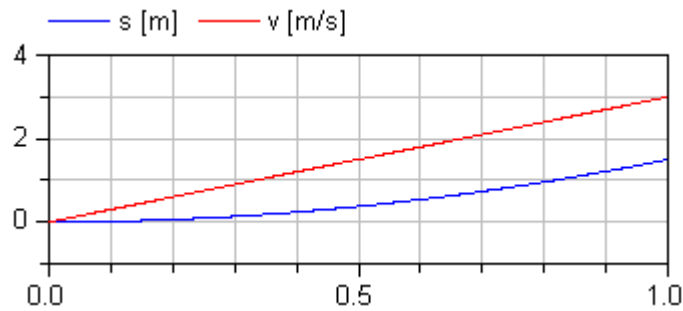
Besides the equations of a model, also additional information can be stored together with the model within the **annotation**(..) definition. In the Dymola GUI this is performed by clicking on button “text” layer (see Figure 2.4 below), to input the declarations and equations in the text layer and by clicking on button “diagram” layer (see Figure 2.4 below) to draw a picture of the system for documentation purposes.

Figure 2.4: Diagram and text layer of MovingMass3 model in Dymola.



The information about the diagram layer is stored in “**annotation**(Diagram(...))”. Simulating this model with Dymola leads to the following results where the speed  $v$  grows linearly and the position  $s$  quadratically. *By default*, Dymola shows the variable name and its displayUnit or unit as label for the respective curve in the plot:

Figure 2.5: Simulation results of model MovingMass3.



## 2.2 Basic Models

Connections specify interactions between components and are represented graphically as lines between connectors. A connector should contain all quantities needed to describe the interaction. For example, electrical potential "v" and electrical current "i" are needed for electrical component interfaces. Angle "phi" and cut-torque "tau" are needed for drive train elements:

```
connector Pin
  import SI=Modelica.SIunits;
  SI.Voltage      v;
  flow SI.Current i;
end Pin;
```

```
connector Flange
  import SI=Modelica.SIunits;
  Angle      phi;
  flow Torque tau;
end Flange;
```

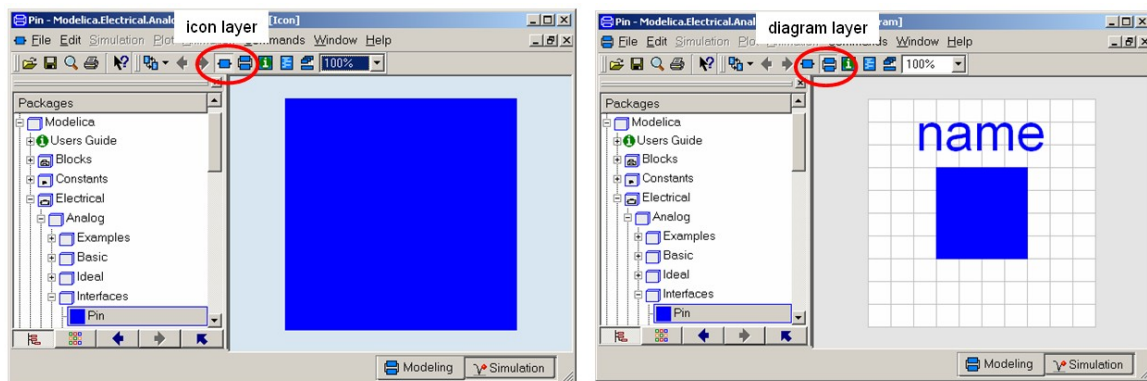
A "connector" definition contains a set of declared variables. The syntax of the declarations is identical to the declaration of variables in a model.

The "flow" prefix in front of a declaration defines that this variable is a *flow variable*. When connecting flow variables together, the sum of the flow variables is zero (a more detailed explanation follows). "Flow" variables have always an associated "direction". In order that correct equations are generated, a convention has to be established how a "positive" flow variable has to be interpreted in a component. Usually, if the corresponding physical quantity flows from the outside into the component, a "flow" variable has a positive value (and a negative value when it flows out of a component). For the "Pin" connector above this means that a current flowing in to an electrical component has a positive value (the definition of the "positive direction" of a "mechanical" connector is more complicated and will be explained later).

Variables without the flow prefix are sometimes called *potential variables*. Connected potential variables are identical.

The visual appearance of a connector is defined in the diagram and the icon layer, see Figure 2.6:

Figure 2.6: Visual appearance of connector Pin in icon and diagram layer.



The picture in the icon layer (here: a filled square, see left part of figure above) is shown at the place where the connector is placed in the icon of a component. The picture in the diagram layer (see right part of figure above) is used when dragging the connector in the diagram layer of a component, i.e., the "internal" part of a component. It is usually best to have a *smaller* picture here in order that the default size is reasonable after

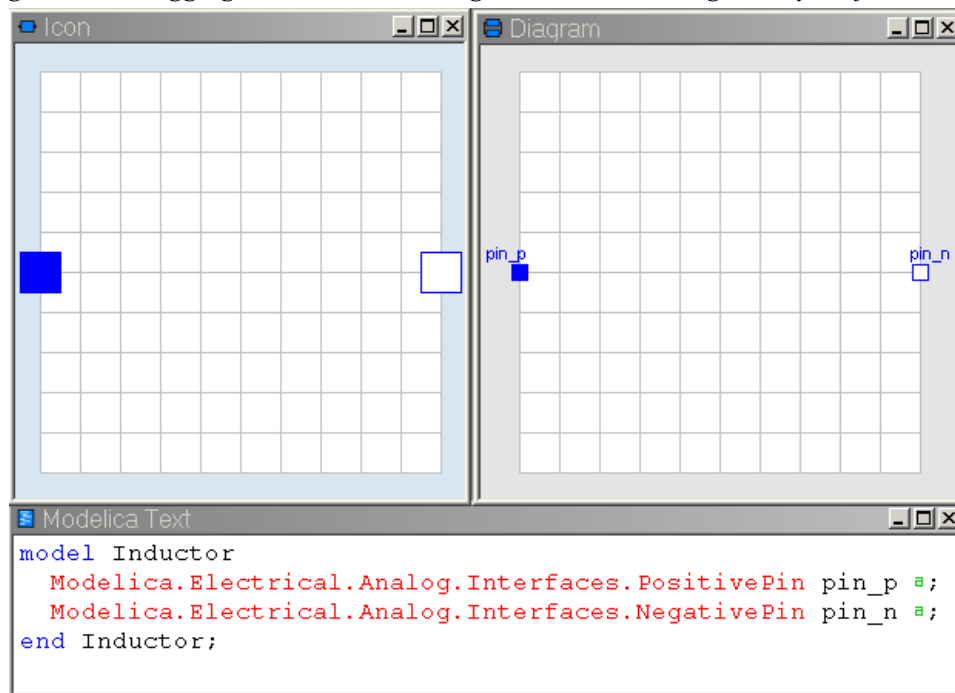


dragging it (otherwise it is too large). Furthermore, the meta-tag “%name” (this is displayed in the figure above as “name”) should be included as a text string, in order to display the name of the connector instance (see right part of Figure 2.7 below).

In the Modelica Standard Library the convention is used, to have always two identical connector definitions that have, however, different icons. It is then easier to identify different connector instances of a component. For example, there are two pin definitions, “PositivePin” and “NegativePin”, where the first one has a “filled” and the second one has a “non-filled” square icon in the Modelica Standard Library.

We will use these two pin definitions to build up a Modelica model of an inductor (the Modelica declarations are identical to connector “Pin” above). Dragging the PositivePin and NegativePin of the Modelica.Electrical.Analog.Interfaces library from the package browser into the “Diagram” layer of a new model results in:

Figure 2.7: Dragging PositivePin and NegativePin in the Diagram layer of a model.

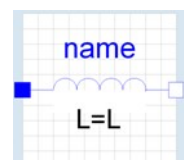


Dragging the two pins in to the “Diagram” layer, will automatically also introduce them in the “Icon” and the “Text” layer of the Inductor (see figure above). The “Text” layer of the inductor can then be edited to add the remaining declarations and equations. Note, when the connector definition would be just manually defined in the “Text” layer, then the graphical information about the connectors would not be present and then it would not be possible to connect graphically to this component in a schematic. Therefore, connectors should always be dragged in to the Diagram layer. The complete definition of the inductor model is:

```
model Inductor "Ideal linear electrical inductor model"
  parameter Modelica.SIunits.Inductance L;
  Modelica.Electrical.Analog.Interfaces.PositivePin pin_p
    annotation(Placement(transformation(extent={{-110,-10},{-90,10}})));
  Modelica.Electrical.Analog.Interfaces.NegativePin pin_n
    annotation(Placement(transformation(extent={{90,-10},{110,10}})));
  equation
    0 = pin_p.i + pin_n.i;
    L*der(pin_p.i) = pin_p.v - pin_n.v;
  end Inductor;
```

“pin\_p” and “pin\_n” are instances of the connector classes “PositivePin” and “NegativePin” respectively. Variables in these instances are accessed by “dot-notation”, e.g., “pin\_p.i” is the current “i” in “pin\_p”. The inductor is defined by two equations:

The first equation states that the sum of the currents flowing in to the inductor via the two ports is zero. Since by convention an inflowing current is positive, this means that if a

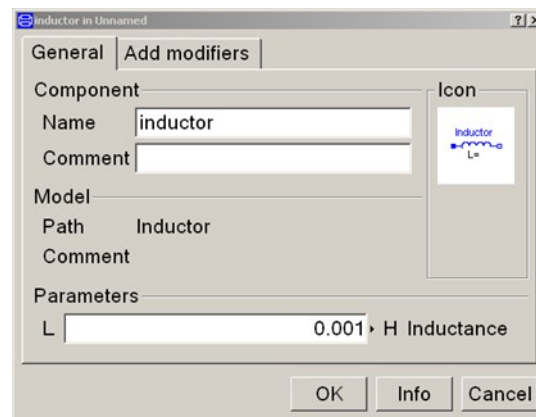


current is flowing in to `pin_p`, then the current at `pin_n` has the same absolute value, but is negative and therefore the current flowing in to `pin_p` is flowing out of `pin_n`.

The second equation states that the time derivative of the current flowing into the component via `pin_p` is proportional to the voltage drop over the two pins. As a last action, a suitable icon is drawn in the “Icon” layer of the inductor model (see figure on the right). Again the meta-tag “`%name`” (displayed as “name” in the figure) is introduced, in order that the instance name of the inductor is displayed when dragging it. Additionally, the actual value of the inductance is displayed by adding the meta-tag “`L=%L`” (displayed as “`L=L`” in the figure). This tag is displayed as a string in the icon. “`%L`” means that the actual value of the parameter with the name “`L`” shall be displayed.

When dragging the just defined model into another model, the inductor icon is shown. Double clicking on the icon opens the parameter menu. This menu is automatically constructed from the parameter declarations in the inductor model and shows the parameter name, its unit/displayUnit and description text, as well as an input field in which the actual value of the inductance can be defined:

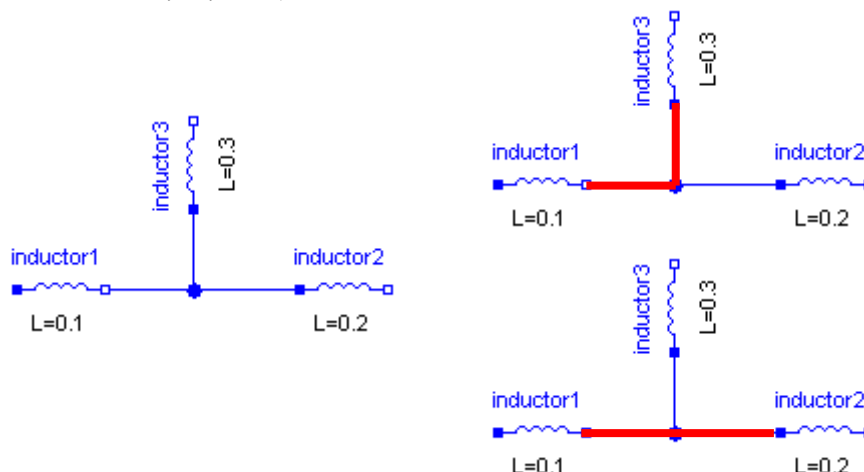
Figure 2.8: Automatically generated parameter menu from Inductor definition.



Note, that the unit “H” is displayed in the parameter menu above. The reason is that variable “`L`” is declared as “`Modelica.SIunits.Inductance`” and this pre-defined type has unit “H”.

After dragging components (such as the inductor model from above) into a diagram layer, connection lines can be drawn between connector instances. Modelica supports only “two-point” connections, i.e., connection lines that start at one connector instance and end at another connector instance. It is *not* possible to connect, say, into a middle of a line. Let us connect three inductors together in a “Diagram” layer of a new model:

Figure 2.9: Connecting three inductors together (`inductor1` and `inductor2`, and `inductor1` and `inductor3` are connected. The circle in the connection point is automatically introduced by Dymola).





Although the diagram (left part of Figure 2.9) looks as if “inductor3” would be connected to the line connecting “inductor1” with “inductor2”, this is actually not the case. In the right part of the figure the two “real” connection lines from inductor1 to inductor2 and from inductor1 to inductor3 are shown. The circle in the middle of the three connected lines (which looks graphically like an electrical node) is automatically placed by Dymola to graphically visualize that the three connected lines form one connection set. Note, this does not mean that it is possible to connect to this “node”. Instead a connection line must always start at one connector and end at another connector. The graphically drawn connection lines are interpreted as the following Modelica equations (they are shown in the “Text” layer of the model):

```
equation
  connect(inductor1.pin_n, inductor2.pin_p);
  connect(inductor1.pin_n, inductor3.pin_n);
```

The “**connect(..)**” construct is interpreted as a built-in operator that introduces equations and has therefore to be present in the “**equation**” section of a model. When processing a model, a Modelica translator replaces a set of “**connect(..)**” statements with a set of equations by the following rules:

1. Corresponding connector variables in a **connect(..)** statement are set equal, if they *do not* have the flow prefix.
2. Variables that have the **flow** prefix and belong to the same “connection set” are combined together in one equation stating that the sum of these flow variables is zero.

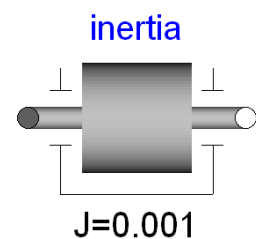
Applying these rules to the above two **connect(..)** statements above, results in the following equations (which are therefore completely equivalent to these statements):

```
inductor1.pin_n.v = inductor2.pin_p.v;
inductor1.pin_n.v = inductor3.pin_n.v;
0 = inductor1.pin_n.i + inductor2.pin_p.i + inductor3.pin_n.i;
```

The three equations state that the electrical potentials at the three pins are identical and that the sum of the electrical currents is zero. These are exactly Kirchhoff’s voltage and current laws and therefore describe correctly this electrical circuit. In section 3.1 on page 41 it will be explained in more detail, how and why the two Modelica connection rules are able to describe physical component coupling phenomena.

In a similar way, also other basic components can be constructed. For example, a rotational inertia is described by the following model.

```
model Inertia "1-dim. rotational inertia"
  import SI = Modelica.SIunits;
  import Modelica.Mechanics.Rotational.Interfaces;
  Interfaces.Flange_a flange_a;
  Interfaces.Flange_b flange_b;
  parameter SI.Inertia J;
  SI.AngularVelocity w "speed of inertia";
equation
  flange_a.phi = flange_b.phi;
  w = der(flange_a.phi);
  J*der(w) = flange_a.tau + flange_b.tau;
end Inertia;
```



It consists of a shaft with two flanges, `flange_a` and `flange_b`. Since the two flanges are rigidly connected, the angles `flange_a.phi` and `flange_b.phi` are identical. Furthermore, the momentum balance along the axis of rotation states that the time derivative of the momentum (i.e.  $d(J \cdot \omega)/dt = J \cdot \dot{\omega}$ ) is proportional to the sum of the cut-torques.

## 2.3 Inheritance

It is often the case that different models share the same code fragment. It is then convenient and less error prone to define these common code fragments only once and reference them correspondingly. In Modelica this is performed with the “**extends**” clause. For example, several electrical base components consist of two electrical pins and the voltage drop “`u`” between the two pins is needed to formulate the “material law”

of the component:

```
partial model OnePort
  Modelica.Electrical.Analog.Interfaces.PositivePin pin_p;
  Modelica.Electrical.Analog.Interfaces.NegativePin pin_n;
  Modelica.SIunits.Voltage u "Voltage drop between pin_p and pin_n";
  Modelica.SIunits.Voltage i "Current flowing from pin_p to pin_n"
equation
  0 = pin_p.i + pin_n.i;
  u = pin_p.v - pin_n.v;
  i = pin_p.i;
end OnePort;
```

Additionally, model “OnePort” defines that the in-flowing current is identical to the out-flowing current. The model is declared as “**partial**”, meaning that it is incomplete and that it cannot be “dragged” into another model. For example, the following usage of “OnePort” is not allowed:

```
OnePort onePort; // wrong Modelica code fragement
                // since OnePort is "partial".
```

“**partial**” models can only be used via inheritance with the language construct “**extends**”:

```
model Inductor
  extends OnePort;
  parameter Modelica.SIunits.Inductance L;
equation
  L*der(i) = u;
end Inductor;
```

The meaning is that the whole definition of “**OnePort**” is “conceptually” included in model Inductor, that is, the model is completely equivalent to:

```
model Inductor
  Modelica.Electrical.Analog.Interfaces.PositivePin pin_p;
  Modelica.Electrical.Analog.Interfaces.NegativePin pin_n;
  Modelica.SIunits.Voltage u "Voltage drop between pin_p and pin_n";
  Modelica.SIunits.Voltage i "Current flowing from pin_p to pin_n"
  parameter Modelica.SIunits.Inductance L;
equation
  0 = pin_p.i + pin_n.i;
  u = pin_p.v - pin_n.v;
  i = pin_p.i;
  L*der(i) = u;
end Inductor;
```

With the “**OnePort**” model, also other electrical components can be easily defined, such as:

```
model Resistor
  extends OnePort;
  parameter Modelica.SIunits.Resistance R;
equation
  u = R*i;
end Resistor;

model Capacitor
  extends OnePort;
  parameter Modelica.SIunits.Capacitance C;
equation
  C*der(u) = i;
end Capacitor;
```

As with all declarations in Modelica, there is no particular order required and the “**extends**” keyword can be placed where ever appropriate. For example, if the partial model contains parameter declarations, one has to decide whether these should appear in the parameter menu before or after the parameter declarations of the

model class using the partial model and place then the **extends** clause correspondingly.

Several **extends** clauses in a declaration are possible (= multiple inheritance is supported). The meaning is that all declarations and equations of the model classes referred in extends clauses are conceptually inserted. If two different model classes have the same declarations and both models are inherited, the declarations must be (syntactically) identical and only one of them is inserted. For example, model C in

```

model A
  parameter Real v;
  ...
end A;

model B
  parameter Real v;
  ...
end B;

model C
  extends A;
  extends B;
  ...
end C;

```

is conceptually equivalent to:

```

model C
  parameter Real v;
  ...
end C;

```

If **model** B would define a default value for  $v$  ( $v = 1$ ), whereas this would not be the case in **model** A, then **model** C would be no longer a valid Modelica model and the Modelica translator will complain.

## 2.4 Conditional Equations

A Modelica model can utilize similar control structures as in a programming language like C. Due to the equation-based nature and since Modelica is designed to define simulation problems, there are some differences. The most important control structures – if expressions and if clauses – are discussed below. They are used to describe conditional equations. More details are given in Chapter 6 page 108. Loops in **equation** sections are discussed in section 5.2, whereas control structures in **algorithm** sections are discussed in section 5.3.

An *expression* in Modelica, e.g., in an equation or in a declaration, may contain an “*if-expression*”, i.e., an if-clause that is written directly in the expression, e.g., as shown for the following limiter:

```

equation
  y = if u > 1 then 1 else if u < -1 then -1 else u;

```

As usual, the first branch “ $y = 1$ ” is selected if  $u > 1$ , otherwise the second branch “ $y = -1$ ” is selected provided  $u < -1$  and if also this does not hold, the **else** branch is selected “ $y = u$ ”. Every if-expression must have an **else** branch, in order that an equation is always well-defined.

Alternatively, conditional expressions can be defined with “*if-clauses*”, such as:

```

if u > 1 then
  y = 1;
elseif u < -1 then // "elseif" not "else if" as for if-expressions
  y = -1;
else
  y = u;
end if;

```

This if-clause has the same effect to compute  $y$  as the previous if-expression. Every **if**-clause must have an

**else**-clause and *every branch* must have exactly the *same number of equations*. Again, the reason is that in every situation the number of equations must be the same, in order that the number of unknowns and the number of equations is always identical.

In fact, since Modelica is an equation based system, every language construct is finally mapped to equations, in order that the symbolic transformation algorithms, see Chapter 12, can be applied. Typically, a Modelica translator (and especially Dymola) transforms an if-clause in to a set of equations with if-expressions. For example, the if-clause below has two equations in every branch:

<pre> <b>if</b> condition <b>then</b>   expr1a = expr1b;   expr2a = expr2b <b>else</b>   expr3a = expr3b;   expr4a = expr4b; <b>end if</b>; </pre>	$\longrightarrow$	<pre> 0 = <b>if</b> condition <b>then</b> expr1a - expr1b     <b>else</b> expr3a - expr3b;  0 = <b>if</b> condition <b>then</b> expr2a - expr2b;     <b>else</b> expr4a - expr4b; </pre>
--	-------------------	--

It is mapped to two equations having if-expressions. After this mapping, the two equations might be “sorted” independently to each other, i.e., in the generated code, the two equations need not be present one after the other.

A more general if-clause is the following:

```

parameter Integer levelOfDetail(min=1,max=3) "Modeling level";
equation
  if levelOfDetail == 1 then
    0 = u2 - u1;
  else
    L*der(i) = u2 - u1;
  end if;

```

Here, different equations are selected, depending on **parameter** “levelOfDetail”. Modelica has the special semantics, that the branch of an if-clause must be selected *during translation*, if *all conditions* of an if-clause depend on *parameter expressions* (i.e., expressions with parameters, constants, and/or literals). In such a case, it is no longer possible to set new values to the parameters used in the if-conditions after translation. The reason for this behavior is that Modelica simulation environments are currently not able to handle situations as above where either an algebraic or a differential equation shall be selected. A Modelica simulation environment is typically not able to handle this selection (a switch between an algebraic and a differential equation) during simulation, since the symbolic pre-processing is different.

In case an if-branch is selected during translation (i.e., when all conditions are parameter expressions), the restrictions that every if-clause must have an else clause and that the number of equations in every branch must be the same, do no longer hold, since only the selected branch is used for the symbolic pre-processing.

If a basic model has *different modeling levels*, it is often implemented with if-clauses where the conditions are parameters, as in the previous example with **parameter** “levelOfDetail”, see, e.g., (Kuhn et. al. 2008). Since Dymola selects the branch during translation and then performs symbolic preprocessing on this branch, the resulting code is as efficient as it can be for every modeling level.

If the sketched special semantics is not desired, an if-expression can be used. For example, Dymola will *not* select the branch during translation, if the above example is written as:

```

0 = if levelOfDetail == 1 then u2 - u1 else u2 - u1 - L*der(i);

```

Here, the branch is selected during simulation and “levelOfDetail” might be a variable that changes during time. In this particular example, the simulation will fail, because Dymola is currently not able to handle such a drastic change of a model during simulation (to switch between an algebraic and a differential equation).

The condition of an if-expression/-clause consists of relations that are build with

“>, >=, <, <=” (“greater”, “greater or equal”, “less”, “less or equal”),

e.g. “x1 >= x2” or “u1 < u2”, and that are combined by the standard Boolean operators

“and, or, not”.

Hereby, the standard precedence holds, i.e., the highest precedence has a relation followed by “**not**”, followed by “**and**” and finally by “**or**”. Example:

```
if u > 0 or not y < 1 and u > 1 then
  // identical to "if (u > 0) or ( (not (y < 1)) and (u > 1) ) then"
  ...
end if;
```

The “**==**” operator can be used to check equality of Integer, Boolean and String variables, e.g.:

```
Integer gear;
equation
  if gear == 1 then
    ...
  end if;
```

For Real variables, the “**==**” operator is only allowed in Modelica functions, see section 5.4 (even in a function, it should be avoided since the result is usually very sensitive). Outside of functions, e.g., in an **equation** section, this operator is forbidden. The reason for this restriction is that every relation in Modelica triggers an event, see section 6.1, in order that the model is guaranteed to be continuous during integration, which is a pre-requisite of every numerical integration algorithm. In order that event handling is possible, a relation is transformed to a zero crossing of a variable together with a hysteresis around zero. As a result, a relation of the form “ $x == 0$ ” is transformed to a zero crossing function “ $z = x - 0$ ”, i.e., whenever  $x$  crosses zero, an event is triggered. Due to the hysteresis around zero (which has to be introduced for numerical reasons), in nearly all cases  $x$  is either greater or less than zero when the event is triggered. The situation that  $x$  is identical to zero can therefore nearly never occur.

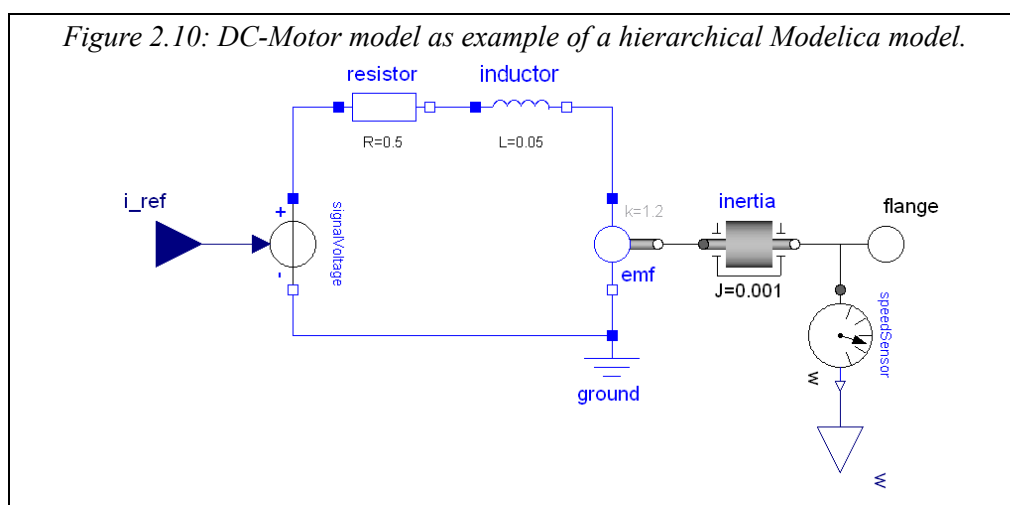
If it is really necessary to check a Real variable for equality, the recommended way in Modelica is to check for a small range of values, e.g.,

```
// Not allowed outside of functions
y1 = if x == 0 then 0 else 1;

// Use instead a small range (or if any possible, avoid it)
eps = 1000*Modelica.Constants.eps;
y2 = if abs(x) <= eps then 0 else 1;
```

## 2.5 Hierarchical Models

We will now use the basic electrical components defined in library Modelica.Electrical.Analog.Basic to build up the electrical circuit diagram of the direct current motor in the next figure (see also Figure 1.1):



The DC motor is modeled with a voltage source, the resistance and inductance of the armature and the idealized “emf” (electro-motoric-force) component to transform electrical into mechanical energy without losses.

The mechanical flange of the “emf” component drives the inertia of the motor. In the text layer the above model is described by the following Modelica model:

```

model Motor
  Modelica.Mechanics.Rotational.Interfaces.Flange_b flange;
  Modelica.Blocks.Interfaces.RealInput          i_ref;
  Modelica.Mechanics.Rotational.Inertia inertia  (J = 0.001);
  Modelica.Electrical.Analog.Basic.Resistor resistor(R = 0.5);
  Modelica.Electrical.Analog.Basic.Resistor inductor(L = 0.05);
  ...
equation
  connect(inertia.flange_b, flange);
  connect(inertia.flange_a, emf.flange_b);
  connect(resistor.n      , inductor.p);
  ...
end Motor;

```

The first two declarations define the two interfaces `flange` and `i_ref`. Then, all used components are declared in the form:

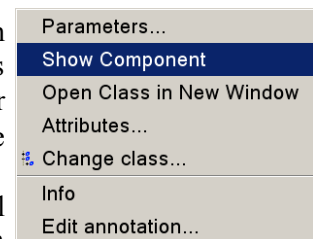
`<model-class> <object>(<modifier>);`

where `<model-class>` is the name of the model to use, e.g., “Resistor” or “Inertia”, `<object>` is the name of the local instance of this component, such as “resistor” or “inertia”, and `<modifier>` is a “modification” of the model-class. In the simplest form, new values for parameters are provided. For example “inertia(J = 0.001)” means that parameter “J” of object “inertia” gets the value “0.001”. In the equation section, every connection line in the diagram layer is represented by a corresponding “**connect**(..)” statement, defining the connectors that are connected together. In a similar way, further hierarchies can be constructed.

Assume that the Motor model above is used as object in another model, e.g., in the drive train of Figure 1.1. When clicking on the Motor icon, the parameter menu is opened. The menu will be empty, because no parameter is defined in the Motor model. It is always possible via a hierarchical modifier to change parameter values at every level of a hierarchy, e.g.,

```
Motor motor(inertia(J = 0.002));
```

In the graphical user interface of Dymola this is performed by right clicking on “motor” and selecting “Show Component” (see figure to the right). This shows the diagram level of “motor”. Clicking on “inertia” opens the parameter menu of “inertia” and a value of “J” can be given. After clicking “OK”, the above modification is introduced at the highest model level.



For important parameters, it is more convenient for a user to “propagate” all or the most important parameters from the lower to the highest hierarchy when defining the Motor model. For example, the Motor model above could be changed to:

```

model Motor
  import SI = Modelica.SIunits;
  parameter SI.Inertia      J = 0.001 "Motor inertia";
  parameter SI.Resistance R = 0.5   "Armature resistance";
  parameter SI.Inductance L = 0.05  "Armature inductance";
  parameter Real           k = 1.05  "Torque = k*current";

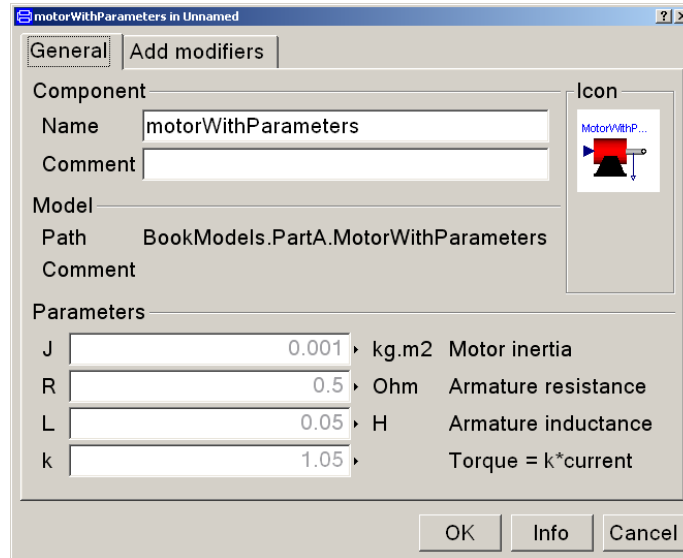
  Modelica.Mechanics.Rotational.Interfaces.Flange_b flange;
  Modelica.Blocks.Interfaces.RealInput          i_ref;
  Modelica.Mechanics.Rotational.Inertia inertia  (J = J);
  Modelica.Electrical.Analog.Basic.Resistor resistor(R = R);
  Modelica.Electrical.Analog.Basic.Resistor inductor(L = L);
  // rest of the model as before
end Motor;

```

Note, that a modifier such as “inertia(J1 = J2)” means that variable “J1” inside “inertia” (iner-

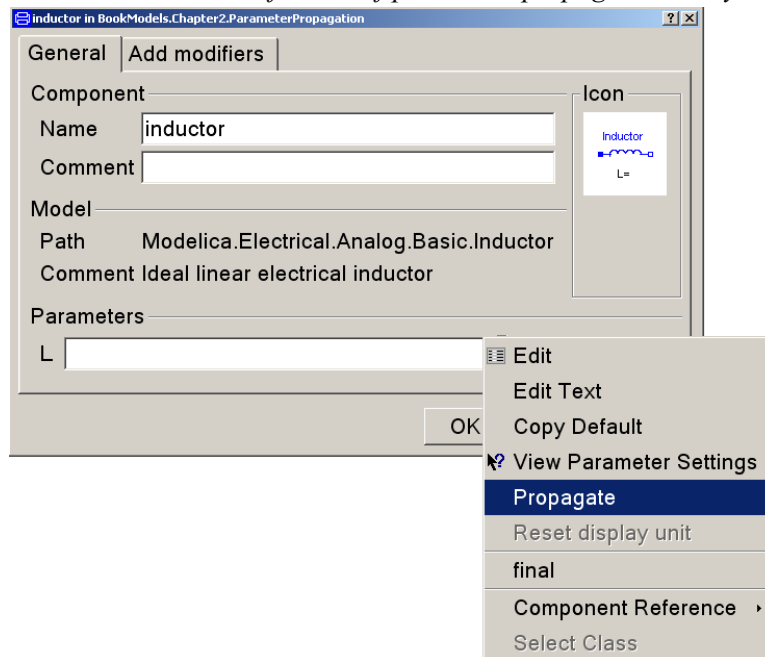
tia.J1) is identical to variable “J2” which is defined outside of “inertia”. Clicking now on an instance of model `Motor` will open the following parameter menu, in which the user can directly provide the motor data:

Figure 2.11: Motor model with propagated component parameters.



Parameter propagation can be performed conveniently in Dymola by opening the parameter menu of the desired component, clicking on the small arrow at the right side of the input field and selecting “Propagate”:

Figure 2.12: Convenient definition of parameter propagation in Dymola.



This pops up a window showing the declaration of the selected parameter (here: `inductor.L`):

Figure 2.13: Declaration of parameter via “Propagate” selection.

In the input fields, all parts of the declaration can be modified. The result is shown in the bottom field named “Modelica:”. For example, the “Description” text might be changed from “Inductance” to “Armature inductance”. Clicking on “OK” introduces (1) the shown declaration in to the model and (2) the corresponding modifier. In the example above the result is (the underlined text is newly introduced after selecting “Propagate”):

```

model Motor
  parameter Modelica.SIunits.Inductance L "Armature inductance";
  Modelica.Electrical.Analog.Basic.Inductor inductor(L=L);
  ...
end Motor;

```

## 2.6 Model Libraries

To ease usage, models are collected together in libraries. A library is called “**package**” in Modelica. For example the Modelica Standard Library is defined as:

```

package Modelica
  package Mechanics
    package Rotational
      model Inertia
        ...
      end Inertia;

      model Spring;
      ...
    end Spring;
    ...
  end Rotational;
end Mechanics;
...
end Modelica;

```

A “**package**” can contain **packages**, **models**, **connectors** and other types of classes. Note, a “**class**” is the generic term for all structuring elements in Modelica, such as **packages**, **models** or **connectors**. It is not possible to store instances of a **class** in a package, with the exception of **constants**. A **class** stored in a package is accessed from the “outside” of the **package** by referring to the full name, such as:

```

model Motor
  Modelica.Mechanics.Rotational.Inertia inertia(J=0.001);
  ...
end Motor;

```

A **class** stored in a **package** is accessed from the “inside” of the **package** by searching the “first” name of the desired classes from the inside “hierarchically” up. For example,

```

package Modelica
  package Mechanics

```



```

package Rotational
  package Interfaces
    connector Flange_a
    ...
  end Flange_a;
  ...
end Interfaces;

model Inertia
  Interfaces.Flange_a flange_a;
  Modelica.Mechanics.Rotational.Interfaces.Flange_a flange_b;
  ...
end Inertia;

model Spring;
  ...
end Spring;
  ...
end Rotational;
end Mechanics;
...
end Modelica;

```

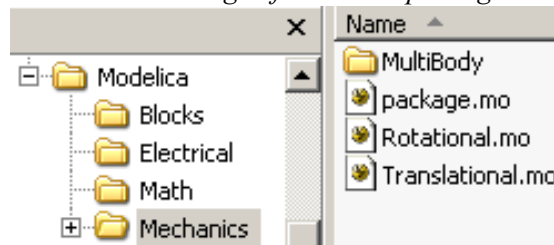
In **model** *Inertia*, two connectors from another package are accessed. The “*Interfaces.Flange\_a*” connector is determined by searching the first part of the name (here: “*Interfaces*”) outside of “*Inertia*”. In this case, the name is already present in the next hierarchical level. The rest of the name is then searched downwards, starting from *Modelica.Mechanics.Rotational.Interfaces*. An alternative is to provide the full name of the class, as if it would be accessed from the outside, such as

*Modelica.Mechanics.Rotational.Interfaces.Flange\_a*

above. Note, that the first part of the full name (here: “*Modelica*”) is still searched from the “inside” to the “outside” until it is found on the highest level. The remaining part of the name (*Mechanics.Rotational.Interfaces.Flange\_a*) is then looked up inside **package** *Modelica*. Due to this search rule, the name of a root level package (such as “*Modelica*”) should not be used as name of a subpackage.

Classes are stored in the file system to keep them persistent. Modelica has a simple mapping rule to map Modelica names to names in the file system, in order that a Modelica translator can automatically detect and load packages that are referred in a model. In the simplest case, the content of a package is stored in one file and the name of the file is the package name with the extension “.mo”. For example, **package** *Modelica* would be stored in a file called “*Modelica.mo*”. Alternatively, a package can be stored in several directories and files. In this case, every package hierarchy is mapped on a corresponding directory hierarchy. For example, **package** *Modelica* could also be stored in a Windows file system as:

Figure 2.14: Hierarchical storage of Modelica package in the file system.



Here, “*Modelica*”, “*Mechanics*” and “*MultiBody*” are stored as directories with the same name. Package *Modelica.Mechanics.Rotational* is stored in file “*Rotational.mo*” within directory *Modelica\Mechanics*. In every directory a file with the name “*package.mo*” must be present. This file contains the definition of the corresponding package class. For example, file “*package.mo*” in directory “*MultiBody*” might be defined as:

```
// File: Modelica\Mechanics\MultiBody\package.mo
within Modelica.Mechanics; // Full name = Modelica.Mechanics.MultiBody
package MultiBody "Library to model 3-dim. mechanical systems"
  annotation (Documentation (info="<html>
    // Overview of package MultiBody
    </html>"));
end MultiBody;
```

The first line in a file consists of the “**within**” clause consisting of the full name of the package in which the sublibrary is contained. For example, if “Modelica.Mechanics” is referenced in the **within**-clause and the package is called “MultiBody”, then the full name of this package is “Modelica.Mechanics.MultiBody”. The root level package uses the statement “**within** ;”, i.e., an empty **within**-clause.

As a last ingredient, directory names can be stored in the environment variable “MODELICAPATH”, for example:

```
MODELICAPATH="C:\user\StandardLibraries;C:\user\project\vehicles;"
```

We are now in the position to describe the search mechanism in Modelica: When a Modelica translator needs the definition of class “A.B.C”, and this class is not yet loaded into the Modelica environment the following search is performed:

- Is file A.mo present in the current directory?
- Is directory “A” and file “A\package.mo” present in the current directory?
- Is file “A.mo”, or otherwise file “A\package.mo”, present in one of the directories defined in the environment variable MODELICAPATH?

When class “A” is found in one of the locations above, the rest of the name, i.e., “B.C” is searched under the location of “A”. If A.B.C is found in the file system, it is automatically loaded into the Modelica environment, otherwise an error occurs that the Modelica class could not be located.

Every Modelica environment has Modelica packages in its distribution. It is tool specific, how these libraries are located. In Dymola, the first part of a class name is first searched in the Dymola distribution directory under “Dymola\Modelica” and “Dymola\Modelica\Library”. Only if the name is not found, the search strategy from above is used (searching in the current directory and then in the directories of MODELICAPATH). The advantage is that a user cannot accidentally “shadow” classes in the Dymola distribution which can lead to unexpected and wrong behavior. Sometimes it is desired to search the Dymola library location not in the first place. Then the special string:

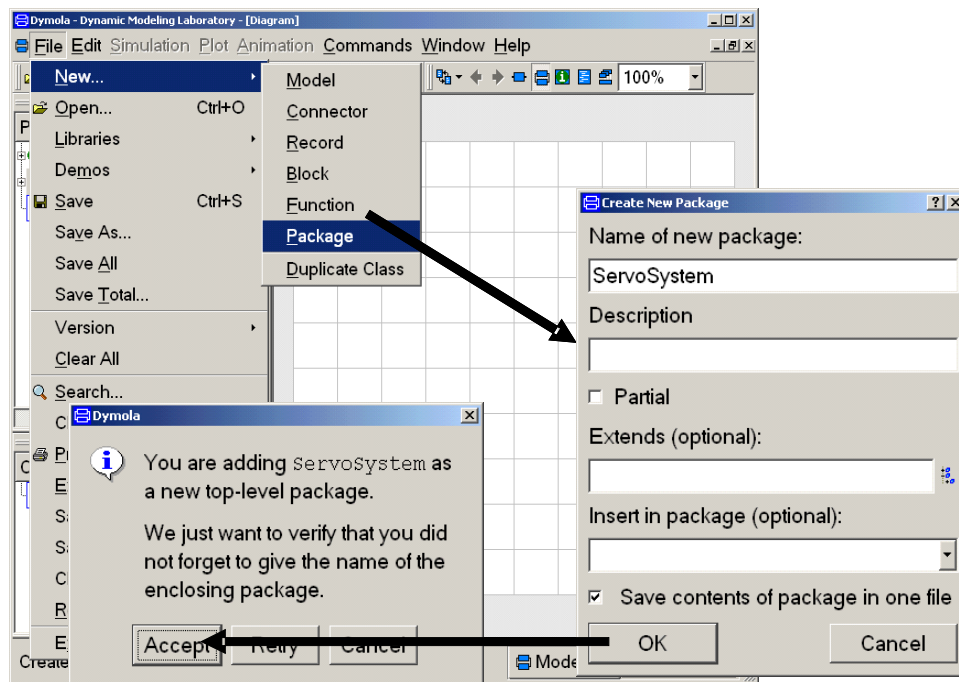
```
$DYMOLA/Modelica/Library
```

has to be included in MODELICAPATH at the desired place exactly as it is given above. Note, that the Unix directory separator, “/”, has to be used, even if MODELICAPATH is defined on a Windows system.

A package could be manually constructed with a text editor in the file system. It is, however, much more convenient to define it with a Modelica tool. In Figure 2.15 it is sketched, how a new package can be defined in Dymola.

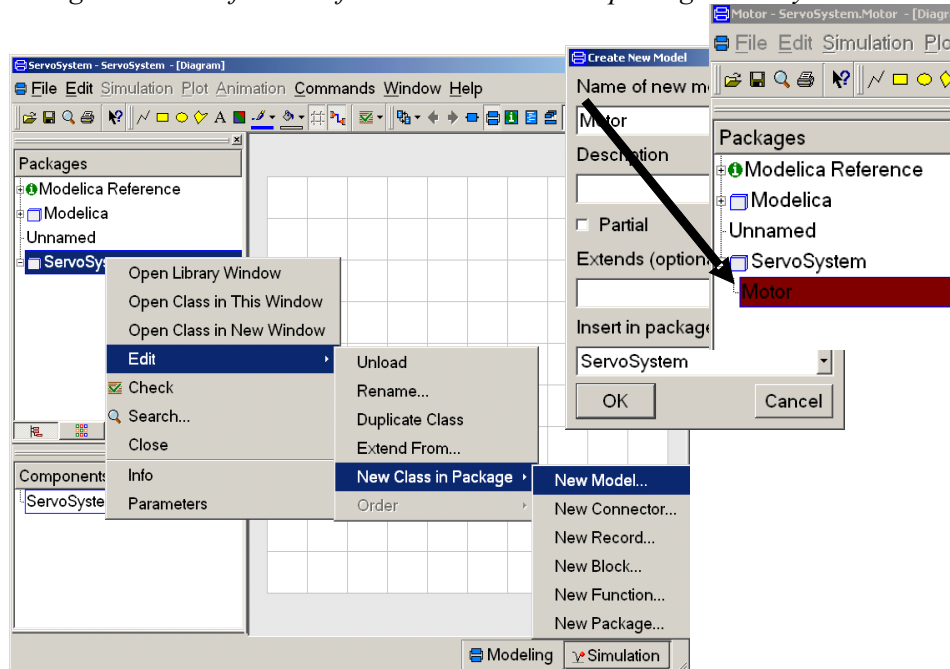
Selecting “File / New / Package” opens the menu “Create New Package”. Here, basically, the name of the desired package can be defined and whether the package shall be stored in one file or as a directory (in the latter case, deselect the check box “Save contents of package in one file”). A final menu pops up, where the user has to select “Accept”, if the package shall be created as root level package (= not inside another package). If the package shall be included as sub-library in another package, the desired package has to be selected from the pull down list “Insert in package (optional)”.

Figure 2.15: Definition of a new package with Dymola.



If a sub-library shall be created, or a model or connector should be created in a library, it is usually most convenient to first select with the right mouse button the desired package (here: ServoSystem). Then the menu entry “Edit / New Class in Package / New Model ..” has to be selected:

Figure 2.16: Definition of a new model inside a package with Dymola.



A package can be restructured, by again right clicking on the corresponding package and selecting “Edit”. The “Edit” menu contains all necessary operations to restructure a package depending on the context, especially “Rename”, “Duplicate Class” and “Remove”. Note, with “Rename” it is possible to move a class from one package to another package, or from the top level in to a package.

Classes are shown in the package browser according to the definition order. This order can be changed by clicking on a class in the package browser and moving it up or down with the arrow keys of the keyboard provided the <ctrl> key is pressed. Alternatively, the class can be moved by selecting it with the right mouse

button and selecting a command under “Edit / Order”.

If a package is stored in one file, the “definition order” is preserved by storing all classes in the “definition order” in this file. If a package is stored in several files, there is the problem that the “storage scheme” does not allow to preserve the “definition order”. As a result, if such a package is loaded in to a Modelica environment, no “definition order” is present and it is tool specific how the classes are ordered/shown in the package browser. This is a flaw in Modelica 3.0 and previous versions. Most likely this will be fixed in the next language release Modelica 3.1. In Dymola, classes stored in different files are shown in alphabetical order in such a case (classes in the same file are shown in the order in which the classes are stored in this file). Furthermore, Dymola provides the Dymola-specific **annotation** “`__Dymola_classOrder`” to explicitly define the ordering of classes in Dymola’s package browser. For example:

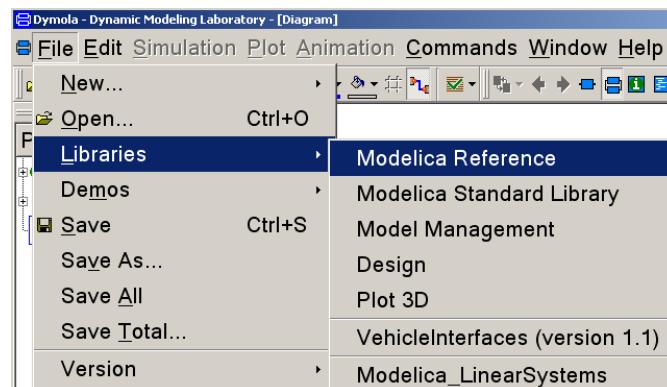
```
package Modelica_Fluid
  annotation(__Dymola_classOrder=
    {"UsersGuide", "Examples", "System", "Vessels", "Pipes",
     "Machines", "Valves", "*"});
  ...
end Modelica_Fluid;
```

defines that classes are ordered according to the given sequence (first: “UsersGuide”, then “Examples” etc.) and that classes not in the list are shown in alphabetic order at the end, due to the last “\*” entry.

If a used package is referenced in a model and it is not yet loaded in to Dymola, then it is *automatically loaded* provided Dymola can find it in its library directory, in the current directory, or in one of the directories defined in the MODELICAPATH environment variable (details see page 34). A package can be *manually loaded* in the following ways:

- “*File / Open ...*” opens a dialog box to read a file. If the complete package is stored in one file, then this file name has to be selected. If a package is stored in several files in a directory, then file “package.mo” of the root level package has to be selected. For example, the Modelica\_Fluid library is loaded manually by selecting file “Dymola\Modelica\Library\Modelica\_Fluid\package.mo” in the “File / Open ...” dialog box.
- When command “**import** <package-name>” is given in the command window in the simulation tab, then package <package-name> is loaded if it is in the search path (= Dymola library directory, current directory, and MODELICAPATH). For example, if the command “**import** Modelica\_Fluid” is given, then package Modelica\_Fluid is loaded in to Dymola and is shown in the package browser.
- “*File / Libraries*” allows to select a package from a list of libraries. The selected package is loaded (after searching it in the search path). In Figure 2.17 the packages are shown that are by default available in this menu:

Figure 2.17: Default packages in the “File / Libraries” menu.



A user can conveniently extend this menu with his/her own definitions. This is performed by storing file “libraryinfo.mos” in directory “<package-directory>\Scripts” or in “<package-directory>”. For example in the Modelica\_LinearSystems library this file is stored as “Dymola\Modelica\Library\Modelica\_LinearSystems\libraryinfo.mos”. When Dymola is started, it determines the “libraryinfo.mos” files of all packages in the search path and uses this information to build up the Library menu. As an exam-

ple, the “libraryinfo.mos” file of the (free) Modelica\_LinearSystems library is shown:

```
// File Modelica_LinearSystems\libraryinfo.mos
LibraryInfoMenuCommand(
  category    ="libraries",
  text        ="Modelica_LinearSystems",
  reference    ="Modelica_LinearSystems",
  version     ="0.95",
  isModel     =true,
  description="Modelica_LinearSystems Library (0.95)",
  ModelicaVersion="3.0",
  pos=1001)

// LibraryInfoPreload(reference="Modelica_LinearSystems")
```

The package is searched in the search path with the package name given under “reference” (here: “Modelica\_LinearSystems”). It is shown in the library menu with the string given under “text” (here: “Modelica\_LinearSystems”), and it is shown in Dymolas status bar with the string given under “description”(here: “Modelica\_LinearSystems Library (0.95)”). Entry “pos” defines the position in the library menu. Numbers 0..999 are reserved for Dynasim. If command “Library-InfoPreload” is given in the file (this command is prepared in a comment above), then the library is automatically loaded when Dymola is started and it is therefore shown in the package browser.

## 2.7 Balanced Models

In Modelica 3.0 the concept of “balanced models” is introduced, in order to significantly improve the quality of Modelica models and to allow a tool to provide very precise error messages about the source of an error. A detailed description and discussion of this concept is provided in (Olsson et.al. 2008). This section is based on this article.

Basically, restrictions are introduced in to Modelica in order that every model must be “*locally balanced*”, which means that the number of unknowns and equations must match on every hierarchical level. It is then sufficient to check every model *locally* only *once*, e.g., all models in a library. Using these models (instantiating and connecting them, redeclaring replaceable models etc.) will always lead to a model where the total number of unknowns and equations are equal. Besides this strong guarantee, it is possible to precisely pin-point which submodels have too many equations or lack equations in case of error.

Before Modelica 3.0 it was possible to connect components from a library and the resulting system could have either too many or not enough equations. Worse, the tool could not pin-point the source of the error. Dymola improved this situation some years ago, by introducing a heuristic to detect the component that leads to a mis-match of equations and unknowns. This heuristic works well in many cases. Still, it is not possible to trust the result. In fact, there are cases where this heuristic points (erroneously) to a correct component.

Since Dymola 7.0, the Modelica 3.0 restrictions are checked if the Modelica Standard Library version 3.0 and later is loaded (since this library requires Modelica 3.0). For example, this means that an *error message* is triggered for components that are not locally balanced. If a previous version of the Modelica Standard Library is loaded instead (e.g., version 2.2.2), Dymola works slightly different for backward compatibility: If the model is correct according to Modelica language version 2 but erroneous according to version 3, only a *warning message* is printed. In order that you are able to find an error in your model easily, it is highly advisable to first correct all components where such warnings are reported, before using them. Otherwise, the strong guarantee about “balanced models” does not hold (because one or more used components are not “locally balanced”).

The basic rules for “balanced models” are rather simple and are introduced together with examples to demonstrate the issues:

### Rule 1: Restrictions on Connectors

The number of *flow* variables in a *connector* must be *equal* to the number of *non-flow* variables that do not have an **input**, **output**, **parameter**, **constant** prefix.

The following connector is correct according to this rule:

```
connector Pin // correct Modelica 3.0 connector
  Real u;
  flow Real i;
end Pin;
```

because the connector has *one flow* (“i”) and *one non-flow* (“u”) variable. The following connector is not allowed in Modelica 3.0 (but was fine in previous Modelica versions):

```
connector Flange // wrong Modelica 3.0 connector
  Real angle;
  Real speed;
  flow Real torque;
end Pin;
```

because this connectors has *one flow* (“torque”) and *two non-flow* (“angle”, “speed”) variables. If the number of flow and non-flow variables has to be different for a particular application, the modeler has to use the **input/output** prefix (see also section 3.4). Example:

```
connector FluidPortA // correct Modelica 3.0 connector
  Modelica.SIunits.Pressure p;
  flow Modelica.SIunits.MassFlowRate m_flow;
  input Modelica.SIunits.SpecificEnthalpy h_inflow;
  output Modelica.SIunits.SpecificEnthalpy h_outflow;
end FluidPortA;

connector FluidPortB // correct Modelica 3.0 connector
  Modelica.SIunits.Pressure p;
  flow Modelica.SIunits.MassFlowRate m_flow;
  output Modelica.SIunits.SpecificEnthalpy h_inflow;
  input Modelica.SIunits.SpecificEnthalpy h_outflow;
end FluidPortB;
```

The two connectors are correct, because both have *one flow* (“m\_flow”) and *one non-flow* (“p”) variable and the remaining variables have an **input** or an **output** prefix. Whenever a connector has variables with the **input/output** prefix, restrictions are present how connectors can be connected together, because connected components must follow the block-diagram semantics (e.g., two input variables cannot be connected together). For the two connectors above this means, that two instances of FluidPortA or two instances of FluidPortB cannot be connected together. However, an instance of FluidPortA can be connected together with an instance of FluidPortB:

```
FluidPortA A1, A2;
FluidPortB B1, B2;
equation
  connect(A1, A2); // error, since two inputs are connected
  connect(B1, B2); // error, since two inputs are connected
  connect(A1, B1); // fine.
```

Based on the restrictions of connectors with “rule 1”, it is now possible to precisely define how many equations a component must have in order to be “locally balanced”:

#### Rule 2: Restrictions on Number of Equations (“locally balanced”)

The number of equations in a (non-partial) **model** or **block** must be:

$$\text{number of unknowns} - \text{number of inputs} - \text{number of flow variables}$$

Note: Variables with the **constant** and **parameter** prefix are treated as known variables. **model** and **block** instances are *not* taken into account for the equation count, due to rule 3 below (but variables inside a **connector** or **record** instance are taken into account).

The equation count is demonstrated on the basis of a few examples:

```

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

model Capacitor
  parameter Modelica.SIunits.Capacitance C;
  Modelica.SIunits.Voltage u;
  Pin p, n;
equation
  0 = p.i + n.i;
  u = p.v - n.v;
  C*der(u) = p.i;
end Capacitor;

```

The Capacitor model has 5 unknowns ( $u$ ,  $p.v$ ,  $p.i$ ,  $n.v$ ,  $n.i$ ) and 2 flow variables ( $p.i$ ,  $n.i$ ). It is therefore required that this model has  $5 - 2 = 3$  equations, and the model fulfills this requirement.

```

model VoltageSource
  input Modelica.SIunits.Voltage u;
  Pin p, n;
equation
  u = p.v - n.v;
  0 = p.i + n.i;
end VoltageSource;

```

The VoltageSource model has 5 unknowns ( $u$ ,  $p.v$ ,  $p.i$ ,  $n.v$ ,  $n.i$ ), 2 flow variables ( $p.i$ ,  $n.i$ ) and 1 input variable ( $u$ ). It is therefore required that this model has  $5 - 2 - 1 = 2$  equations and the model fulfills this requirement.

With the above two rules it is now possible to understand the basic philosophy of “balanced models”: For every unknown variable in a **model** or **block**, exactly one equation must be provided in the class, with the following exceptions:

- (1) If a variable has an “**input**” prefix, the equation for this variable must be provided when “using” this component (therefore, no equation has to be provided in the **model** or **block** class where this variable is defined).
- (2) If a connector has  $N$  flow variables, then it must have also  $N$  non-flow variables (without a prefix) according to rule 1. Therefore, such a connector introduces  $2N$  unknowns. Exactly  $N$  equations must be provided in the component class where the connector instance is defined, and  $N$  equations must be provided when “using” the component.

Note, that the latter requirement is automatically fulfilled when connecting connectors together: If  $M$  connectors are connected together via **connect** (..) equations and every connector has  $N$  **flow** and  $N$  non-**flow** variables, then the Modelica connecting semantics provides  $(M-1) \cdot N$  equality equations and  $N$  zero-sum equations, i.e., the missing  $M \cdot N$  equations for  $M$  connectors. For example, if 3 Capacitors are connected together:

```

  Capacitor C1, C2, C3;
equation
  connect(C1.p, C2.p);
  connect(C1.p, C3.p);

```

the following 3 equations are generated:

```

C1.p.v = C2.p.v;
C1.p.v = C3.p.v;
0 = C1.p.i + C2.p.i + C3.p.i;

```

and these are the three missing equations for the connectors  $C1.p$ ,  $C2.p$ ,  $C3.p$ . (of course, also 3 equations have to be provided for  $C1.n$ ,  $C2.n$ ,  $C3.n$ , either by connections, or by the automatically introduced default equation that flow variables are zero, if the connector is not connected).

With this explanation, the final rule follows naturally:

**Rule 3: Restrictions on Modifiers and Connectors**

- (a) *Modifiers* are only allowed on variables declared with the **constant**, **parameter**, **input** prefix or a declaration equation.
- (b) *Modifiers must* be provided for variables with the **input** prefix outside of a connector.
- (c) For every *connector* of a component that is *not connected* and has N variables with the **input** prefix, exactly *N equations* must be provided (no modifiers!).

The basic idea is that whenever a component is used, *all missing* equations must be provided, either

- (a) by appropriate *connections* (for connectors),
- (b) by providing appropriate *modifiers* (for variables defined with the **input**-prefix outside of a connector),
- (c) by providing appropriate *equations* (for variables defined with the **input**-prefix inside a connector which is not connected).

A tool can easily check these rules and can therefore easily pin-point precisely which component declaration does not follow these rules, and therefore leads to a model that is not balanced. It is now also clear, that a model is automatically balanced, if all subcomponents follow this rule, because all missing equations of a component must be provided when it is instantiated.

Before Modelica 3.0, modifiers have been allowed for all variables in a component. The above rule significantly restricts modifiers. The big advantage is, however, that a tool is able to precisely detect the source of an error. Examples:

```

model Test // wrong Modelica 3.0 model (was previously correct)
  Capacitor C1(C = 1e-6);           // fine, since modifier on parameter
  Capacitor C2(u = sin(time));      // wrong, since modifier on
                                     // variable without allowed prefix
  VoltageSource V1(u = sin(time)); // fine, since modifier on input
  VoltageSource V2;                 // wrong, since no modifier on u
  ...
end Test;

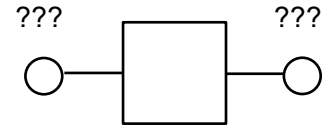
```

There are some additional rules to define “balanced models”. Some of them will be discussed when introducing the corresponding new language elements. All these rules are in the same spirit as above: Whenever a component is “used” *all missing* equations must be defined at this place.



## Chapter 3 Component Coupling by Ports

An important and difficult design decision is the definition of component interfaces, called “connectors” in Modelica. For signal blocks the interface design is simple: The modeler defines input and output signals and that’s it. For physical components that interact with other components, the interface design is more difficult. In this chapter, a systematic approach to design connectors is presented and some important connectors are discussed in detail.



### 3.1 Connector Design

The goal is to provide a set of useful connector definitions to describe physical interfaces, such as electrical pins, mechanical flanges, hydraulic or pneumatic ports, heat transfer through a wall. As a guide line, we start with the following rule:

General Rule for Interface Design:

The abstracted interface of a component should contain all independent variables that are necessary to describe the desired effects in the real interface.

Although this rule is quite obvious, it is surprisingly difficult to apply it in an actual situation. This rule is refined in a more concrete way based on the following requirements for port-based component coupling:

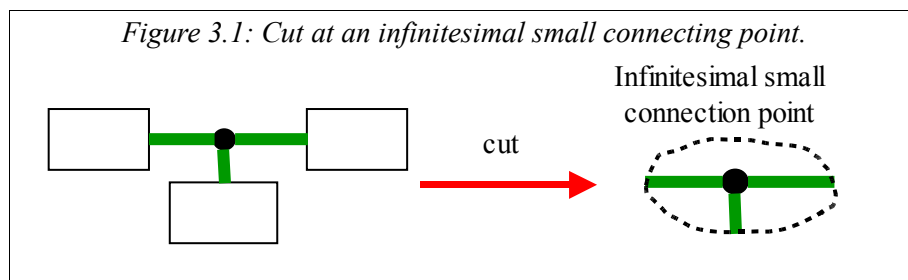
#### 1. Independent Interface Variables:

A connector should contain an independent set of variables that describes the desired physical effects in the interface. Usually, mean values over the interface area are used, e.g., a resultant cut-torque.

Variables  $c_i$  in a connector are not independent, if at least one variable  $c_j$  can be computed by algebraic equations, by differentiation and/or by integration from the other variables  $c_k$  in the connector. In Modelica it is possible to have a redundant set of variables in a connector. In such a case, there are unnecessary restrictions how components can be connected together. The basic reason for this is “rule 1” of balanced models, see page 37 and the example with FluidPortA and FluidPortB (since input/output prefixes must be used in the connector, only 1:1 connections are possible. If a loop structure is present, a special loop-breaker element is needed in order that two FluidPortA or FluidPortB connectors are not connected together).

#### 2. Relevant boundary conditions and balance equations:

When connecting components together the relevant *balance equations* and the relevant *boundary conditions* for the infinitesimal small connection point (see Figure 3.1 below) must be generated by the Modelica connection semantics of the corresponding **connect**(..) equations (variables without the **flow** prefix are equal and the sum of the variables with the **flow** prefix is zero). This is a fundamental requirement in order that the balance equations and boundary conditions are fulfilled both in a component and when connecting a component together with other components.



Examples for *boundary conditions* at an infinitesimal small connection point are given in the next table:

Table 3.1: Examples of connector boundary conditions at an infinitesimal small connection point

$v_i = v_j$	Equality of electrical potentials
$V_{mag,i} = V_{mag,j}$	Equality of magnetic potentials ( $V_{mag} = \int H \cdot ds$ ); H: magnetic field strength)
$s_i = s_j$	Equality of positions
$\phi_i = \phi_j$	Equality of angles
$T_i = T_j$	Equality of temperatures
$p_i = p_j$	Equality of pressures

In order that the above boundary conditions are generated from **connect(..)** equations, the corresponding variables have to be defined as *potential variables* in the connector. Examples:

```

connector XXX
  import SI = Modelica.SIunits;
  SI.Voltage          v;
  SI.MagneticPotential V_mag;
  SI.Position          s;
  SI.Angle             phi;
  SI.Temperature       T;
  SI.Pressure          p;
  ...
end XXX;
```

Examples for *balance equations* at an infinitesimal small connection point are given in the next table:

Table 3.2: Examples of connector balance equations at an infinitesimal small connection point

$0 = \sum i_i$	Kirchhoff's current law (sum of the currents is zero)
$0 = \sum \Phi_i$	Sum of the magnetic fluxes is zero
$0 = \sum f_i$	Force balance (sum of the cut-forces is zero)
$0 = \sum \tau_i$	Torque balance (sum of the cut-torques is zero)
$0 = \sum Q_{flow,i}$	Energy balance of heat transfer (sum of the heat flow rates is zero)
$0 = \sum m_{flow,i}$	Mass balance of a fluid (sum of the mass flow rates is zero)
$0 = \sum G_{flow,i}$	Momentum balance of a fluid (sum of the momentum flow rates " $G_{flow,i} = m_{flow,i} \cdot v_i$ " is zero)

The above equations can easily be derived by formulating the corresponding balance equation for a finite volume and then making the limit for an infinitesimal small volume. For example, Newton's momentum balance in one dimension states that:

$$m \cdot \ddot{s} = \sum f_i \xrightarrow{m \rightarrow 0} 0 = \sum f_i \quad (3.1)$$

In order that the above balance equations are generated from **connect(..)** equations, the corresponding variables have to be defined as *flow variables* in the connector. Examples:

```

connector XXX
  import SI = Modelica.SIunits;
  flow SI.Current          i;
  flow SI.MagneticFlux     Phi;
  flow SI.Force            f;
  flow SI.Torque           tau;
```

```

flow SI.MassFlowRate      m_flow;
flow SI.HeatFlowRate      Q_flow;
flow SI.MomentumFlux     G_flow;
...
end XXX;

```

When designing a component interface, an abstraction of the real physical interface has to take place. Especially it has to be decided which physical effects shall be mathematically described. From this decision follow the boundary and balance equations that shall be fulfilled.

The most fundamental balance equation is the energy balance. Therefore, the energy balance must always be fulfilled, directly or indirectly. The energy balance is fulfilled when the energy flow rate is already used as flow variable, or when “corresponding” potential- and flow variables are used in the connector. For example, a suitable connector for 1-dimensional, translational mechanical systems is:

```

connector TranslationalFlange
  Modelica.SIunits.Position s;
  flow Modelica.SIunits.Force f;
end TranslationalFlange;

```

Assume that 3 components c1, c2, c3 with this interface are connected together. This leads to the following connection equations:

$$\begin{aligned}
 c1.s &= c2.s \\
 c1.s &= c3.s \\
 0 &= c1.f + c2.f + c3.f
 \end{aligned} \tag{3.2}$$



Therefore, the force balance and the boundary conditions of equal positions are fulfilled. The energy flow from the three components into the infinitesimal connection point is computed as (under the assumption that no energy is stored in the connection point):











$$\begin{aligned}
 P &= \frac{dE}{dt} \\
 &= \frac{d c1.s}{dt} \cdot c1.f + \frac{d c2.s}{dt} \cdot c2.f + \frac{d c3.s}{dt} \cdot c3.f \\
 &= \frac{d c1.s}{dt} \cdot (c1.f + c2.f + c3.f) \\
 &= \frac{d c1.s}{dt} \cdot 0 \\
 &= 0
 \end{aligned} \tag{3.3}$$

Since  $c1.s = c2.s = c3.s$ , also the derivatives of the positions are identical and it is possible to move the position derivative out of the parenthesis. The term in parenthesis vanishes due to the zero-sum equation of the forces and therefore the sum of the energy flows vanishes as well and the energy balance is fulfilled identically.

In the Modelica Standard Library (see below), elementary connectors for all important technical domains are provided based on the general design principle explained above. The definition of these connectors is summarized in the next table:

Table 3.3: Connector definitions in the Modelica Standard Library.

type	potential variables	flow variables	connector name (Modelica library)	icon
<b>physical connectors</b>				
<i>electric analog</i>	electric pot.	electric current	Modelica.Electrical.Analog.Interfaces.PositivePin	
<i>el. multi-phase</i>	vector of electrical pins		Modelica.Electrical.MultiPhase.Interfaces.PositivePlug	

type	potential variables	flow variables	connector name (Modelica library)	icon
<u>magnetic</u>	magnetic pot.	magnetic flux	Modelica_Magnetic.Interfaces.PositiveMagneticPort	
<u>translational</u>	distance	force	Modelica.Mechanics.Translational.Interfaces.Flange_a	
<u>rotational</u>	angle	torque	Modelica.Mechanics.Rotational.Interfaces.Flange_a	
<u>3-dim. mechanics</u>	r[3], T[3,3]	f[3], t[3]	Modelica.Mechanics.MultiBody.Interfaces.Frame_a	
<u>thermal</u>	temperature	heat flow rate	Modelica.Thermal.HeatTransfer.Interfaces.HeatPort_a	
<u>1-dim. thermo-fluid flow</u>	pressure	mass flow rate	Modelica_Fluid.Interfaces.FluidPort_a	
	stream variables: h,Xi,C			
signal connectors				
<u>signal</u>	Real Integer Boolean	none	Modelica.Blocks.Interfaces.RealInput/Output Modelica.Blocks.Interfaces.IntegerInput/Output Modelica.Blocks.Interfaces.BooleanInput/Output	
<u>electric digital</u>	Integer (1..9)	none	Modelica.Electrical.Digital.Interfaces.DigitalSignal	
<u>state machine</u>	Booleans	none	Modelica.StateGraph.Interfaces.Step_in/Transition_in	
<u>signal bus</u>	configurable	none	Icon: Modelica.Icons.SignalBus	

Column “type” is the physical domain, column “potential variables” defines the variables in the connector that have no “**flow**” prefix, column “flow variables” defines the variables with a “**flow**” prefix, column “connector name” defines the connector class name where the definition is present and column “icon” is a screenshot of the connector icon. In Modelica, array dimensions are declared with square brackets, e.g.,  $A[3, 4]$  is a two-dimensional array where the first dimension has size 3 and the second size 4, see also section 5.1. Every physical connector is available with two different icons, one with a “filled” and one with a “non-filled” icon. Otherwise the Modelica definition is identical. In the table above, only the “filled” icon connectors are shown.

For 3-dim. mechanical systems, the position vector  $r[3]$  from the origin of the world frame to the origin of the connector frame, the transformation matrix  $T[3, 3]$  from the world to the connector frame, as well as the cut-force vector  $f[3]$  and the cut-torque vector  $t[3]$  are utilized. The transformation matrix  $T[3, 3]$  contains a redundant set of variables. In Chapter 15 it is explained how the information about the constraint equations of  $T$  is defined in Modelica (“overdetermined connector”), and how a tool can automatically remove connection restrictions due to this redundancy based on this additional information.

For 1-dim. thermo-fluid flow systems a third type of connector variables is needed, called “*stream variables*”. This variable type is used to describe properties that are transported with a flow of matter. Details are given in Chapter 16. The stream variables of the FluidPort are specific enthalpy ( $h$ ), independent mass fractions of multi-substance fluids ( $X_i$ ), and trace substances ( $C$ ).

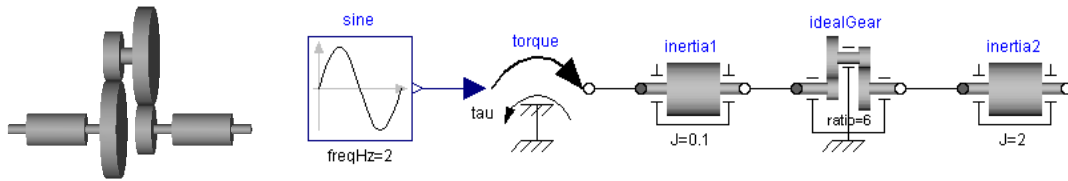
Besides physical connectors, also a set of signal connectors are provided, especially for block diagrams and for hierarchical state machines. Type “signal bus” characterizes an empty connector that has the additional prefix “**expandable**”. This connector type is used to define a hierarchical collection of named signals where the full connector definition (containing all signal definitions) is defined implicitly by the set of variables occurring in all connections of this type. This connector is used to communicate a large amount of signals in a convenient way between components. Details are given in section 3.5.

In the following sections, some of the connector definitions above are explained in more detail. Furthermore, connectors for 3-dim. mechanical systems are discussed in Chapter 15 and connectors for 1-dim. thermo-fluid flow are discussed in Chapter 16.

### 3.2 Connectors for Drive Trains (Rotational library)

Drive trains are mechanical systems where only the rotation around a fixed axis is under consideration. This abstraction allows to model many technical systems with electrical drives and mechanical transmission elements in a very simple and efficient way. Drive trains are modeled with the `Modelica.Mechanics.Rotational` library. A simple drive train build with the Rotational library is shown in Figure 3.2:

Figure 3.2: Simple drive train consisting of motor and load inertia as well as of a gear box.



In the right part of the figure the Modelica drive train is shown consisting of a motor inertia (`inertia1`), a load inertia (`inertia2`), a gear box (`idealGear`) and an applied external torque (`torque`, `sine`). The load inertia is driven with the external torque and drives via the idealized gear box with a transmission ratio of 6 the load inertia. Therefore the speed of the load inertia is 6-times slower as the speed of the motor inertia. In the left part of the figure a sketch of the 3-dim. view of this system is shown. The drive train is fixed on ground. How to model drive trains that are moving in space is discussed in Chapter 14.

Drive train components are connected by shafts and the shaft ends are the drive train interfaces. In the Rotational library they are called “Flanges”. The definition of these flanges is shown in Table 3.4:

Table 3.4: Interface of drive train components of the Rotational library.

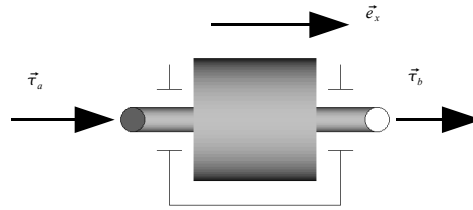
	<pre> <b>connector</b> Flange_b   Modelica.SIunits.Angle      phi; // [rad]   <b>flow</b> Modelica.SIunits.Torque tau; // [N.m] <b>end</b> Flange_b; </pre>
--	---

There is a coordinate system along the axis of rotation which is fixed in the ground (the “red” coordinate system in the figure above). Furthermore, a coordinate system is attached rigidly with every shaft, i.e., it rotates with the shaft. The absolute angle  $\phi$  (phi) between the x-axis of the coordinate system fixed in ground and between the x-axis of the coordinate system fixed in the shaft is the *potential variable* of the flange connector. The resultant cut-torque  $\tau$  (tau) in the cut-plane of the shaft is the *flow variable*. This means that at a connection point the *boundary condition of identical angles* as well as the *torque balance* is fulfilled. A similar analysis as for the connector of the Translational library, see equation (3.3), shows that additionally also the *energy balance* is fulfilled at a connection point.

When needed in a component, derivatives of the flange angle can be easily derived by applying the “`der(. . .)`” operator. For example, the angular velocity (speed) of the shaft is: “`w = der(phi)`”. Two connector definitions are available in the Rotational library: “Flange\_a” has a “filled” circle icon and is usually used for the “left” end of a shaft. “Flange\_b” has a “non-filled” circle icon and is usually used for the “right” end of a shaft. Besides the slightly different icons, the remaining parts of the Modelica connector definitions are identical.

Angular velocity and torque are vector-valued quantities. Operations on vectors from different components can only be carried out if the coordinates of the vectors are resolved in the same coordinate system. Therefore, conceptually every Rotational component has an associated coordinate system which is rigidly attached to the component. A typical component is shown in Figure 3.3:

Figure 3.3: Coordinate system and vectors of an inertia component



The x-axis of the coordinate system fixed to the inertia is denoted as unit vector  $\vec{e}_x$ . All vectors are expressed with this unit vector. This gives the following mathematical description:

Table 3.5: Mathematical and Modelica description of inertia component

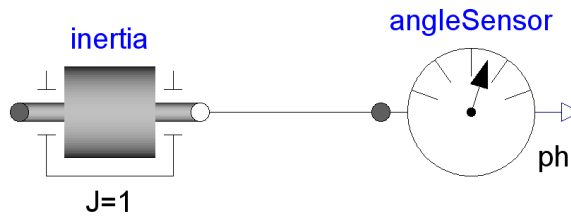
Mathematical description	Modelica definition
$\omega \cdot \vec{e}_x = \dot{\phi} \cdot \vec{e}_x$ $J_{xx} \cdot \dot{\omega} \cdot \vec{e}_x = (\tau_a + \tau_b) \cdot \vec{e}_x$ $\downarrow$ $\omega = \dot{\phi}$ $J_{xx} \cdot \dot{\omega} = \tau_a + \tau_b$	<pre> phi = flange_a.phi; phi = flange_b.phi; w = <b>der</b>(phi); J*<b>der</b>(w) = flange_a.tau + flange_b.tau; </pre>

Since all vectors are expressed with the same unit vector, the vector equations can be replaced by equations of the vector coordinates, as shown in the lower left part of the table above. This is the basis of the Modelica implementation shown in the right part of the table.

When two components are connected together, conceptually the coordinate systems fixed to every component are fixed to each other, i.e., the unit vectors  $\vec{e}_x$  in the connected connectors are identical. Since the unit vectors are identical, it is allowed to operate on the coordinates of the vectors, especially to sum up the cut-torques.

The question arises why the absolute angle is used in the connector and not the absolute angular velocity, as it is often suggested in literature. The reason is that there are applications where the angle of a drive train shall be controlled, e.g., the angle of a robot joint or the angle of a motor that drives an elevator. In these applications the angle must be measured and is then used in the control system. A typical, simple configuration is shown in Figure 3.4:

Figure 3.4: Simple drive train to measure the angle of an inertia.



Basically, this simple system is described by the following Modelica models:

```

model Servo
  Inertia    inertia;
  AngleSensor sensor;
  ...
equation
  connect(inertia.flange_b,
           sensor.flange);
  ...
end Servo;

model Inertia
  Flange_b flange_b;
  ...
end Inertia;

model AngleSensor
  Flange_a flange;
  ...
end AngleSensor;

```

We will now discuss two implementation possibilities:

*Variant 1: Angle in Flange Connector*

```

model Inertia
  parameter Modelica.SIunits.Inertia J;
  Flange_a flange_a;
  Flange_b flange_b;
  Modelica.SIunits.Angle phi;
  Modelica.SIunits.AngularVelocity w;
equation
  phi = flange_a.phi;
  phi = flange_b.phi;
  w = der(phi);
  J*der(w) = flange_a.tau + flange_b.tau;
end Inertia

model AngleSensor
  Flange_a flange;
  RealOutput phi;
equation
  phi = flange.phi;
  0 = flange.tau;
end AngleSensor

```

When the `inertia` and the `angleSensor` components are connected together, this results in the following equation:

```
inertial.flange_b.phi = angleSensor.flange.phi;
```

This formulation is unproblematic. Especially, since both angles are identical, also all higher derivatives are identical as well, especially the shaft speeds of the two components. Alternatively, the angular velocity shall be in the connector, instead of the angle:

*Variant 2: Angular Velocity in Flange Connector*

```

model Inertia
  parameter Modelica.SIunits.Inertia J;
  Flange_a flange_a;
  Flange_b flange_b;
  Modelica.SIunits.Angle phi;
  Modelica.SIunits.AngularVelocity w;
equation
  w = flange_a.w;
  w = flange_b.w;
  w = der(phi);
  J*der(w) = flange_a.tau + flange_b.tau;
end Inertia

model AngleSensor
  Flange_a flange;
  RealOutput phi;
equation
  der(phi) = flange.w;
  0 = flange.tau;
end AngleSensor

```

The main difference to the previous formulation is that the `AngleSensor` contains now a differential equation in order to compute the angle. Especially, this means that initial conditions for this angle can be provided. When the `inertia` and the `angleSensor` components are connected together, this results in the following equation:

```
inertial.flange_b.w = angleSensor.flange.w;
```

This means that the angular velocities of the two components are identical. However, this does not mean that the angles of the two components are identical. Especially, if the initial conditions given for `angleSensor.phi` and for `inertia.phi` are different, the angles are different. To summarize, variant 2 has the disadvantages that (a) an additional initial condition has to be given for `angleSensor.phi` and that (b) the two (independent) initial conditions must agree, in order that the angles of the `inertia` and of the `angleSensor` are identical.

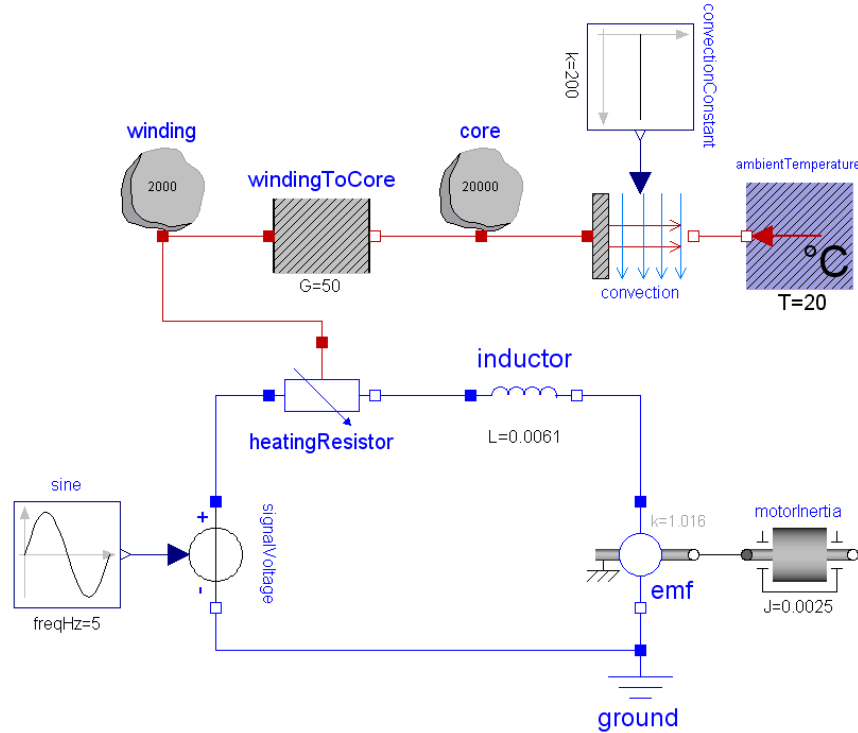
Therefore, variant 2 is not suited for component oriented modeling because connecting components together may easily lead to errors, since the states are independent from each other but not the initial conditions.

### 3.3 Connectors for Heat Transfer (HeatTransfer library)

Library `Modelica.Thermal.HeatTransfer` has components to model heat transfer with “lumped elements” by conduction, convection and radiation. These elements are derived from the integral form of the partial differential equation of heat transfer by using a simple, one-dimensional discretization formula, see, e.g., (Holman

2001). In many cases this allows to model heat transfer in a rough way, especially if the parameters of the lumped elements (such as the heat capacity of a part) can be calibrated with measurements, see e.g. (Kral et. al. 2005). It is often not practical to calculate these parameters from basic analytic formulas if complex geometries with different materials are present, as it is, e.g., the case for electrical motors. More detailed models, especially for complex geometries, require usually to use a finite element program and to utilize the result of a FE-calculation in the Modelica model. A simple example is shown in Figure 3.5.

Figure 3.5: DC-motor model with heat transfer to the environment

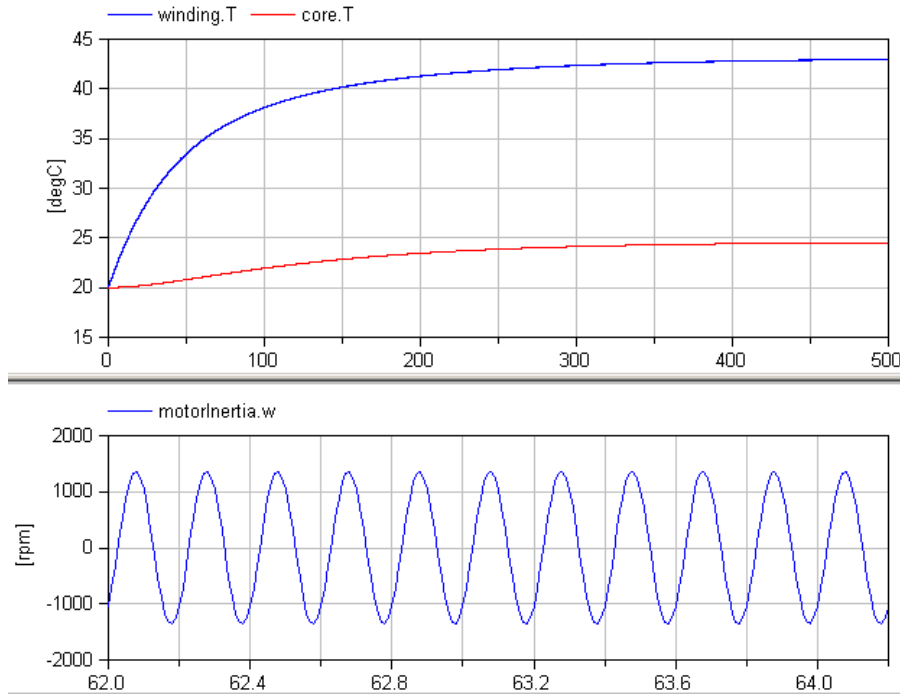


It consists of an uncontrolled DC-motor where the voltage source is driven by a sine wave. The heat transfer in the winding resistor is modeled by using a resistor with a heat port (`Modelica.Electrical.Analog.Basic.HeatingResistor`). The heat generated in the resistor ( $= \text{port.i} * (\text{port\_a.v} - \text{port\_b.v})$ ) is transferred to the heat capacitance of the winding (named: `winding`), is further transferred via conductance to the heat capacitance of the core (named: `core`) and is finally transferred to the environment via convection. The heat flow between these lumped elements is characterized by the “red” connection lines in the figure above.

Typical simulation results are shown in Figure 3.6: The temperatures of the winding and core reach a steady-state value after some time. Since the excitation is a sine wave of the source voltage, the speed of the motor inertia is also a sine wave.



Figure 3.6: Simulation results of simple electrical circuit with heat transfer (temperature in windings and core + speed of motor inertia)



The heat flow connector of the HeatTransfer library has the following definition:

Table 3.6: Interface of heat transfer components of the HeatTransfer library.

	<pre> <b>connector</b> HeatPort_a   Modelica.SIunits.Temperature T;      // [K]   <b>flow</b> Modelica.SIunits.HeatFlowRate Q_flow; // [W] <b>end</b> HeatPort_a; </pre>
--	--

The temperature  $T$  in the connection point is the *potential variable* of this connector. The resultant heat flow rate  $Q_{\text{flow}}$  flowing from the outside into the component is the *flow variable*. This means that at a connection point the *boundary condition of identical temperatures* as well as the *energy balance* (= sum of the heat flow rates) is fulfilled.

An alternative to the above interface variables would be to use entropy flow rate ( $S_{\text{flow}}$ ) instead of heat flow rate ( $Q_{\text{flow}}$ ). The reason is that temperature and entropy flow rate are sometimes used as interface variables in thermodynamics and bond graph literature since their product is energy flow rate ( $E_{\text{flow}} = T \cdot S_{\text{flow}}$ ). However, this would not be a good choice here: For the lumped heat method as used in the Modelica.Thermal.Lumped.HeatTransfer library, the two most important elements are the thermal conductor and the heat capacitor, see Table 3.7:

Table 3.7: Lumped heat transfer elements in different description forms ( $G, C$  are constants).

ThermalConductor		$0 = Q_{\text{flow},1} + Q_{\text{flow},2}$ $Q_{\text{flow},1} = G \cdot (T_1 + T_2)$ $0 = T_1 \cdot S_{\text{flow},1} + T_2 \cdot S_{\text{flow},2}$ $S_{\text{flow},1} = G \cdot (T_1 + T_2) / T_1$
HeatCapacitor		$Q_{\text{flow}} = C \cdot \frac{dT}{dt}$ $S_{\text{flow}} = C \cdot \frac{1}{T} \cdot \frac{dT}{dt}$

The equations of the ThermalConductor and of the HeatCapacitor are both *linear* in  $T$  and in  $Q_{flow}$ , but are *non-linear* in  $T$  and in  $S_{flow}$ , which means that the latter are more difficult to solve numerically. Furthermore, the common boundary condition of perfect insulation can be more easily expressed as “ $Q_{flow}=0$ ” instead of “ $T \cdot S_{flow}=0$ ” where the latter is again a non-linear relationship. Finally, most engineers have a better understanding of  $Q_{flow}$  as of  $S_{flow}$ .

### 3.4 Connectors for Block Diagrams (Blocks library)

Block diagrams consist of a connection of blocks where the interface of every block is defined by input and output signals. In principal, it would be possible to define these interfaces with the already introduced language elements, e.g.,

```
// Do not use (not a good design for block diagram interfaces)
connector RealInput1      connector RealOutput1
  Real signal;             Real signal;
end RealInput1;           end RealOutput1;
```

The disadvantage is that blocks with such interfaces can be connected in an arbitrary way, e.g., two RealOutputs could be connected, although this is not allowed in a block diagram. A better solution is to use the prefixes **input** and **output**:

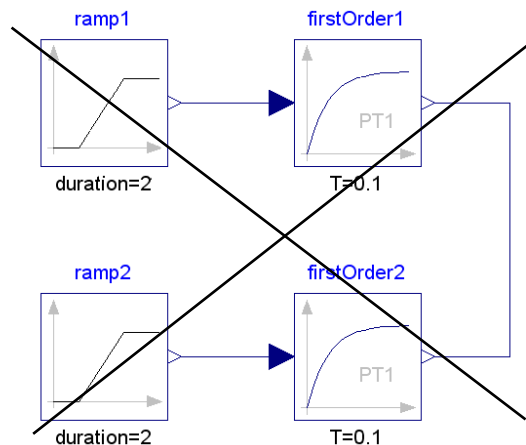
```
// Do not use (not a good design for block diagram interfaces)
connector RealInput2      connector RealOutput2
  input Real signal;       output Real signal;
end RealInput2;           end RealOutput2;
```

Connectors that have **input/output** prefixes can only be connected according to block diagram semantics. This means that the following *restrictions* hold:

1. Two or more inputs cannot be connected together.
2. Two or more outputs cannot be connected together.
3. An input must be connected to exactly one non-input (which is usually an output variable; however, a prefix need not necessarily be present), or an equation must be provided for the input.

This allows, e.g., that errors can be earlier detected with better diagnostics, if the modeler wants to model block diagrams. In Dymola an error is triggered immediately when a connection is drawn and one of the connection restrictions above is violated. For example, in the next Figure 3.7, four input/output blocks are connected together using the above connector definitions. Since the outputs of `firstOrder1` and `firstOrder2` are connected, the model is wrong.

Figure 3.7: Wrong connection of input/output blocks.



The **input/output** prefixes do *not* define “computational causality”, i.e., that an input variable needs to be provided from the outside and that the output variables are computed. The reason is that the most basic prin-

ciple of Modelica is that only equations are defined and that the Modelica tool is responsible to provide an efficient and correct evaluation order of the equations. Therefore, **input/output** prefixes in connectors only *restrict* how connectors can be connected together. This is, e.g., utilized for inverse models, see Chapter 19, where the outputs of blocks are treated as known and the inputs are computed.

An exception are connectors on the top level of a model: Here these prefixes define the *interaction* with the environment where the model is used. Especially, the **input** prefix defines that values for such a variable have to be provided from the simulation environment and the **output** prefix defines that the values of the corresponding variable can be directly utilized in the environment (e.g., when exporting a model to Simulink, the top level prefixes define the inputs and outputs of the corresponding Simulink model).

If these prefixes are not used in a connector (but still in a model), the interpretation is different: The **output** prefix does not have a particular effect and is *ignored*. The **input** prefix defines that a *modifier equation* has to be provided for the corresponding variable when the component is utilized in order to guarantee a *locally balanced* model (i.e., the number of local equations is identical to the local number of unknowns), see section 2.7. Example:

```

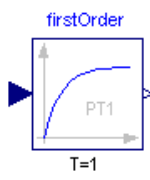
model VoltageSource
  input Modelica.SIunits.Voltage u; // modifier required, since input
  Pin p, n;
  equation
    u = p.v - n.v;
    0 = p.i + n.i;
  end VoltageSource;

model ElectricalCircuit
  VoltageSource source(u = sin(time)); // modifier required for u
  ...
end ElectricalCircuit;

```

Finally, in functions, see section 5.4, the **input/output** prefixes define the computational causality of the function body, i.e., given the variables declared as **input**, the variables declared as **output** are computed in the function body.

Let us now return to the definition of the input/output connectors for components of a block diagram. The introduction of the **input/output** prefixes in connectors RealInput2 and RealOutput2 above is the most important step. However, such a connector is inconvenient to use, due to the unnecessary hierarchy. For example:



```

model FirstOrder_NotNice
  RealInput2 inPort;
  RealOutput2 outPort;
  parameter Modelica.SIunits.Time T "Time constant";
  equation
    T*der(outPort.signal) + outPort.signal = inPort.signal;
  end FirstOrder_NotNice;

```

As can be seen, it is tedious to use a signal name, such as “outPort.signal”, in an equation.

To improve this situation, in Modelica version 2.0 so called “*short class*” definitions have been introduced to avoid a superfluous naming hierarchy if a connector consists only of one variable. Input/output signal interfaces can now be conveniently defined as:

```

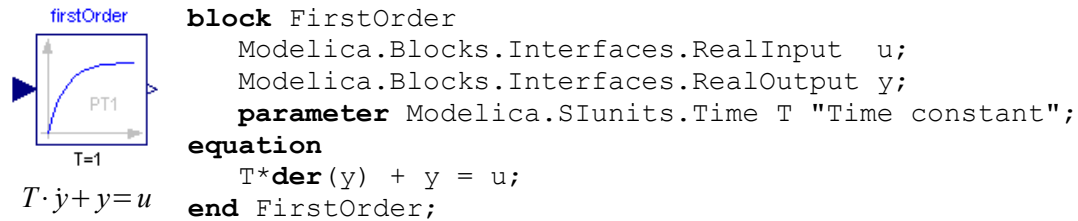
connector RealInput = input Real;
connector RealOutput = output Real;

```

In fact, these definitions are present in sublibrary Modelica.Blocks.Interfaces and are used in the components of the Modelica.Blocks library. These definitions imply that a connector instance is both a connector and a variable (i.e., has a “value”). The additional advantage is that arrays of such connectors can be easily defined, for example:

```
RealInput u[:]; // A vector of real input signals
```

The example above can now be implemented as:

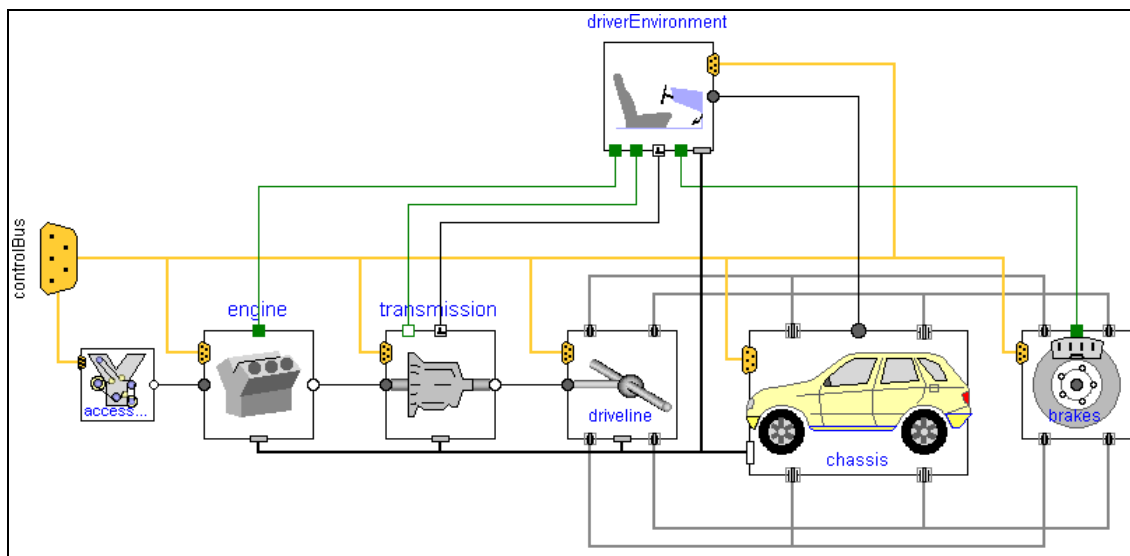


In order to still improve the handling of block diagrams, a new specialized class “**block**” is provided in Modelica (and is used in the “**FirstOrder**” component above): A **block** is a **model** with the restriction that each public variable must have one of the prefixes **constant**, **parameter**, **input** or **output**. By using the “**block**” keyword instead of “**model**”, the developer of this component gets early diagnostics, when implementing the block accidentally not as component that has only input and output variables. For a user of such a block, it is sufficient to inspect that the “**block**” keyword is used, in order to be sure that the Modelica translator guarantees that this component behaves as an input/output block of a block diagram.

### 3.5 Connectors for Signal Buses

Signal interfaces become quickly unmanageable if many signals have to be communicated between components. Therefore, buses are used in control systems of vehicles, aircrafts, drive trains, power plants etc. to communicate a large amount of data between components. This behavior can be mimicked in an idealized way in Modelica. A typical example is shown in Figure 3.8 (from the free *VehicleInterfaces* library; this library is available in Dymola under File / Libraries):

Figure 3.8: Communication of signals in a vehicle system model of the *VehicleInterfaces* library.



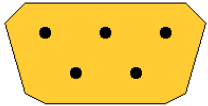
This is a system model of a vehicle with engine, transmission, drive line, chassis, brake, accessory and driver models. The many signals between these components are communicated via the connector “controlBus”. In principal, it would be possible to just define one large connector in which all desired signals are present. However, this gives several practical problems:

- If only some of the components are used (e.g. only a test model to test the “engine” model above), then dummy values have to be provided for not used bus signals in order that every declared variable can be computed. For large signal sets this becomes very inconvenient.
- Different model levels and/or model variants for a component might be present (e.g., in Figure 3.8 component “chassis” might be a simple 1-dim. model or a complicated 3-dim. model) and they might provide different signals to the bus or might need different signals from the bus. The effect is that whenever simplified models are used, then again dummy values have to be provided for not used bus signals. Worse, if a new variant is introduced that requires more signals from the bus, then existing models have to be adapted by introducing dummy values for the newly needed bus variables.

- If the component models are developed by different people or organizations and/or are even in different libraries, then one has to organize the definition of the single `controlBus` connector used by all components. This is especially cumbersome if new signals are introduced or deleted, since all libraries must be updated at the same time.

Due to the problems sketched above, “**expandable**” connectors have been introduced in Modelica 2.2. This connector type does not have a pre-defined set of declarations, but allows to automatically introduce new variables on the connector when connecting to it. An example is shown in Table 3.8.

Table 3.8: Example of bus definition with an expandable connector.

 <p>Icon provided in Modelica.Icons.SignalBus</p>	<pre> <b>expandable connector</b> ControlBus   // no declarations;   // copy the content of   // Modelica.Icons.SignalBus here. <b>end</b> ControlBus; </pre>
--	---

The connector usually does not contain any variable declarations and has only the prefix “**expandable**”. If variable declarations are present, they are handled exactly in the same way as in other connectors. However, there is the restriction that an expandable connector may not have **flow** variables. The actual content of an expandable connector is built up by connecting input and/or output signals to it. Example:

```

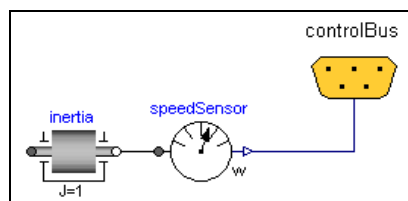
model TestBus
  Modelica.Blocks.Continuous.FirstOrder firstOrder;
  ControlBus controlBus;
  ...
equation
  connect(firstOrder.y, controlBus.filteredSpeed);
  ...
end TestBus;

```

Due to the `connect(...)` statement, variable “**filteredSpeed**” is implicitly introduced on the “`controlBus`”. This means that the new variable `controlBus.filteredSpeed` is implicitly declared. Only signals declared with the **input** or **output** prefix can be connected to an expandable connector. Dymola supports expandable connectors in a nice way in the diagram layer. Example:

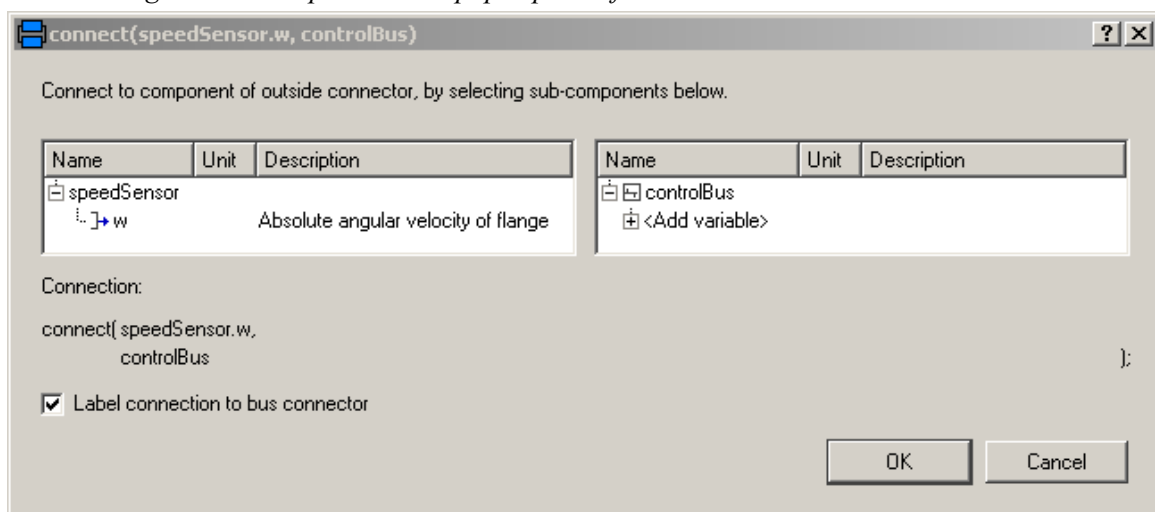
1. The output signal of a `speedSensor` component is connected to an instance of connector `ControlBus`:

Figure 3.9: Step 1: Connecting the output of the `speedSensor` component with the `controlBus`.



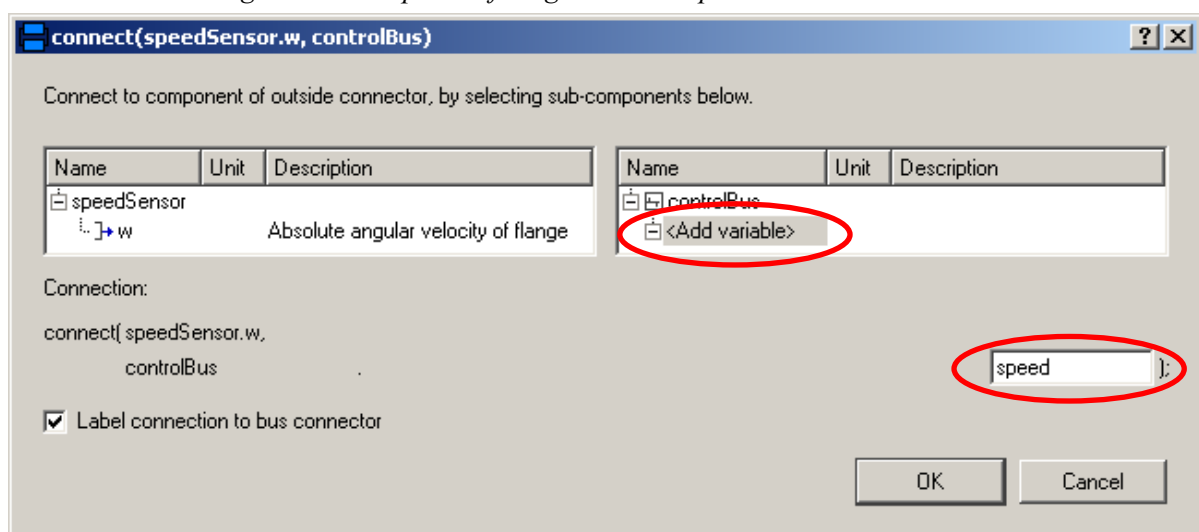
2. After drawing the connection line, the menu from Figure 3.10 pops up to define the variable on the bus for this connection. In the left column the definition of connector `speedSensor.w` is shown. The right column is used to define the variable on the `controlBus` to which `speedSensor.w` is connected.

Figure 3.10: Step 2. A menu pops up to define the variable on the controlBus.



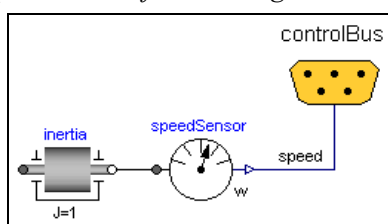
3. Clicking on “<Add variable>” opens an input field in which the name of the variable on the bus can be defined (alternatively, clicking on the “+” sign at the left side of “<Add variable>” shows a list of variables that have been defined in other classes where connections to a ControlBus connector have been made. Clicking on one of them selects the signal for the connection):

Figure 3.11: Step 3: Defining variable “speed” on the controlBus.



4. Clicking on “OK” introduces the connection “**connect**(speedSensor.w, controlBus.speed)” and adds the bus variable as label on the connection line (since “Label connection to bus connector” is enabled in the menu above):

Figure 3.12: Step 4: Result after clicking on “OK” in the menu.

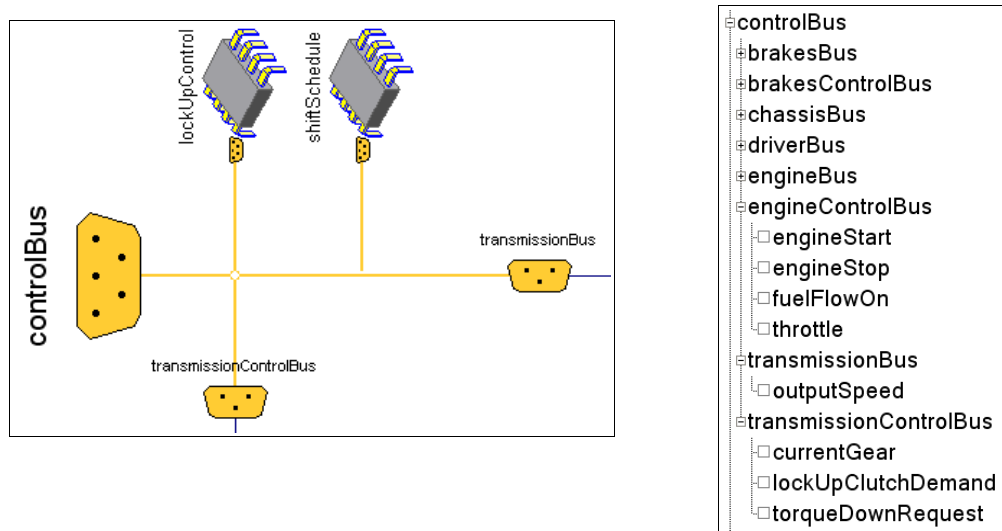


The label on the connection line is in principal a nice feature to see at once to which variable speedSensor.w is connected. The current handling in Dymola is, however, limited: If the connection line is drawn from a connector *to the bus*, then the label is placed *horizontally* to the *right* side of this connector (see Fig-

ure 3.12). If the connection line is drawn *from the bus* to a connector, then the label is placed *horizontally* to the *left* side of this connector. Therefore, the components that shall be connected to the bus must be appropriately placed, in order that the label is near to the connection line. If this is not possible, it is usually best to disable “Label connection to bus connector” in the menu above.

It is also possible to introduce sub-buses especially to structure large sets of signals. For sub-buses the slightly different icon `Modelica.Icons.SignalSubBus` should be used. An example from the `VehicleInterfaces` library is shown in Figure 3.13:

Figure 3.13: Sub-buses used in the `VehicleInterfaces` library (left: diagram layer, right: plot window).



A sub-bus is also an **expandable connector**. When connecting two expandable connectors together, the above menu also pops up. If the “transmissionBus” shall be a sub-bus inside the “controlBus”, then in the menu the “transmissionBus” has to be defined in the controlBus input field of “<Add variable>” leading to the statement “**connect**(controlBus.transmissionBus, transmissionBus)” and this statement introduces the sub-bus “controlBus.transmissionBus”. In Figure 3.13 above, on the left side the two sub-buses “transmissionBus” and “transmissionControlBus” are shown that are connected to the “controlBus” in the diagram layer. When translating the model, and selecting the controlBus in the variable list of the plot browser, the list on the right side of the figure appears which shows the hierarchical structure of the bus.

Without special action, the three buses above are all visible in the icon layer, because connectors are always visible in the diagram *and* in the icon layer. Usually, this is not desired for sub-buses because only the top-most bus connector shall be visible in the icon and connections shall be made from the outside only to this bus. This can be achieved by right-clicking on a sub-bus and selecting in the pop-down menu “Attributes ...”. A menu opens where attributes of variables and component instances can be defined. When selecting “Protected”, then the corresponding connector is no longer visible in the icon layer and it is not possible to connect to it from the outside. Since this is a bit tedious to perform for every used sub-bus, a better alternative is to define the sub-bus in the following way:

```
expandable connector ControlSubBus
  annotation (defaultComponentPrefixes="protected");
end ControlSubBus;
```

As a result, whenever an instance of this connector is dragged, the “**protected**” prefix is automatically introduced and therefore the connector is not visible in the icon layer. Dragging introduces the following code:

```
protected
  ControlSubBus controlSubBus;
```

When connecting two buses, the connecting line has usually not the desired “dark yellow” color as shown in the figures above. The reason is that every connection line (of any connector) is drawn with a default color and default thickness. This default is determined from the line attributes of the first graphical object defined in the icon layer of the connector at which the connection line starts. For this reason, the `SignalBus` and



SignalSubBus connectors in library `Modelica.Icons` have a small “dark yellow” rectangle defined as first graphical object in the icon layer. This rectangle is not directly visible because the “bus” icon is placed on top of it. When drawing a connection line from the bus to another connector, the color and thickness of this rectangle is used as color and thickness of the connection line. Due to this feature, it is recommended to copy the annotation of the `Modelica.Icons.SignalBus/SignalSubBus` component in your own control bus connector. Note, in this special case you should not use inheritance (using “**extends** `Modelica.Icons.SignalBus`”, see section 2.3), because the connection line attributes are not influenced by superclasses and they are only determined from the first graphical object directly defined in the icon layer of the corresponding class.

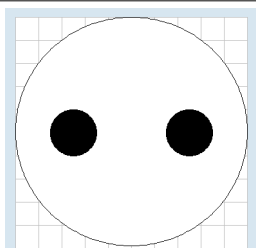
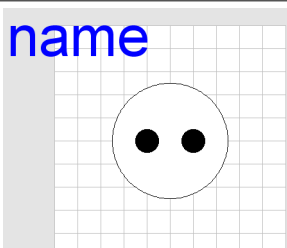
### 3.6 Hierarchical Connectors

It is quite difficult to design physical connectors. Fortunately, for the most important domains these connectors are available in the Modelica Standard Library as summarized in Table 3.3 on page 43. In Modelica it is possible to build up hierarchical connectors, i.e., connectors consisting of other connectors. It is usually best to use the available connectors in the Modelica Standard Library as basis for such hierarchical connectors. For example, a user might define the following connector “**Plug**” to describe cables with two electrical lines by selecting “File / New ... / Connector” and defining the connector in the text layer by dragging from the package browser the corresponding physical connectors in to the *text layer* resulting in:

```
connector Plug
  Modelica.Electrical.Analog.Interfaces.PositivePin phase;
  Modelica.Electrical.Analog.Interfaces.NegativePin ground;
end Plug;
```

Finally, an appropriate icon is drawn in the *icon* and *diagram layer* and using the text “%name” (displayed as “name”) in the diagram layer in order that the connector instance name is displayed when dragging the connector in a diagram layer:

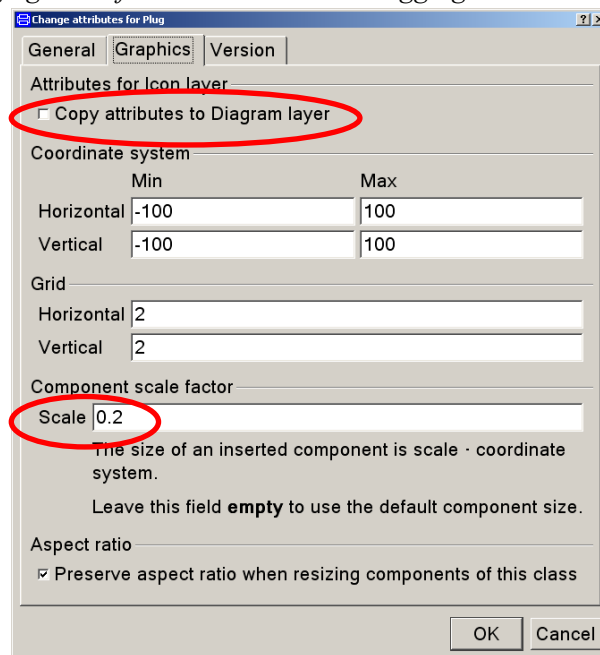
Table 3.9: Icon and diagram layer of Plug connector.

<i>icon layer</i>	<i>diagram layer</i>
	

As usual, the icon in the diagram layer should be about 40 % - 50 % of the size of the icon in the icon layer, in order that the connector looks “nice” when dragging it in to the diagram layer of a model. When dragging this connector in to a model, then the *diagram layer* of the *Plug* connector is shown in the *diagram layer* of the *model* and the *icon layer* of the *Plug* connector is shown in the *icon layer* of the *model*. Usually, the icon in the icon layer of a connector is too small if *more details* then just a “square” or a “circle” are shown. The connector icon can be enlarged by selecting the icon layer of the Plug connector and then choose “Edit / Attributes ... / Graphics”, see Figure 3.14:



Figure 3.14: Enlarging the default icon size when dragging a connector (or model).



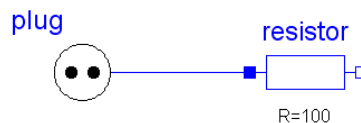
The default “Scale” factor is “0.1”. This means that when dragging a component then the icon size is “0.1” times the size of the “Coordinate system”. In order to enlarge the default icon size by a factor of two, “Scale” is set to “0.2” above. Additionally, only the icon in the icon layer shall be enlarged, not the icon in the diagram layer. For this reason the “Copy attributes to Diagram layer” is disabled.

Connecting two instances A and B of the Plug connector results in a set of equations where corresponding potential variables are set equal and the sum of corresponding flow variables sums up to zero:

$$\begin{array}{ll}
 \text{Plug } A, B; & A.\text{phase.v} = B.\text{phase.v}; \\
 \text{equation} & A.\text{ground.v} = B.\text{ground.v}; \\
 \text{connect } (A, B); & 0 = A.\text{phase.i} + B.\text{phase.i}; \\
 & 0 = A.\text{ground.i} + B.\text{ground.i};
 \end{array}
 \rightarrow$$

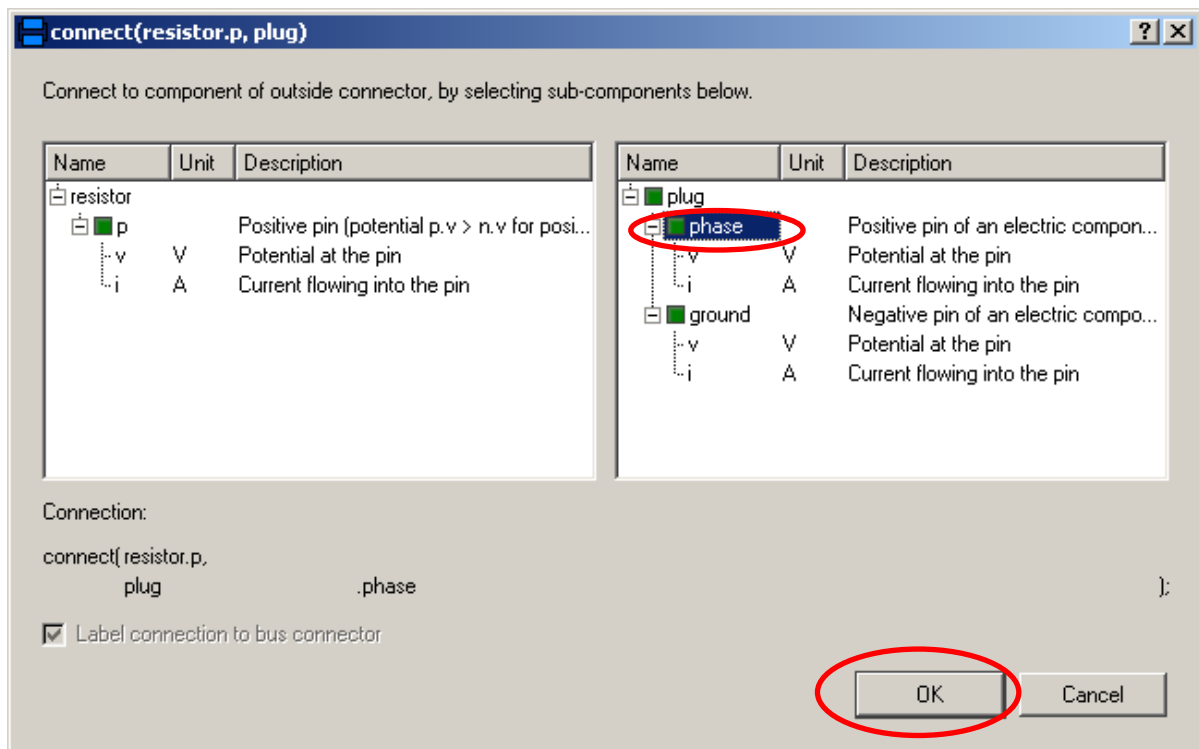
Assume that a resistor shall be connected to a pin in a Plug connector. One solution is to directly connect from the pin of the resistor to the plug, see Figure 3.15:

Figure 3.15: Connecting the pin of a resistor to a Plug connector.



Then the menu of Figure 3.16 pops up and shows in two columns the definitions of the connected connectors. In the “right” column a sub-connector in the plug must be selected that has the same definition as the connector in the “left” column of the menu. In this case, sub-connector “phase” is selected:

Figure 3.16: Menu to select the connection in to a Plug connector.



After pressing “OK” the connection definition is finalized and results in the following Modelica code in the text layer of the test model:

Figure 3.17: Text layer of model in which the pin of a resistor is connected to a plug.

```

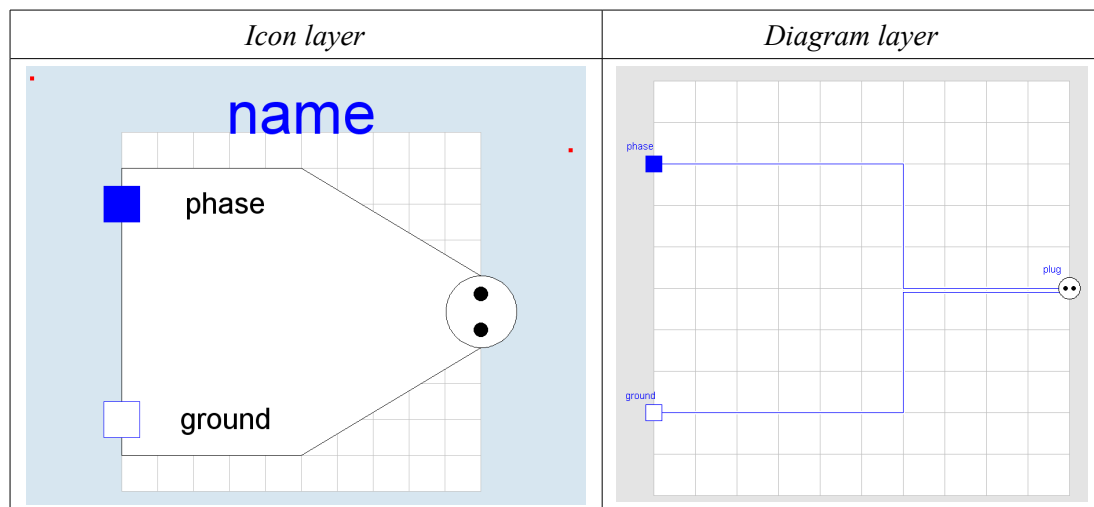
model PinToPlug
  Modelica.Electrical.Analog.Basic.Resistor resistor(R=100) ;
  Plug plug ;
  ;
equation
  connect(resistor.p, plug.phase) ;
end PinToPlug;

```

In a similar way a hierarchical connection from a connector to any hierarchy inside another connector is possible. The only restriction is that the connected sub-connectors must have identical definitions.

For reasonable simple hierarchical connectors an alternative is to provide an adapter model where the hierarchical connector is present on one side and all sub-connectors (to which connections are possible) are provided on the other side of the adapter model. For example, an adapter model for the Plug connector might be defined as shown in Table 3.10:

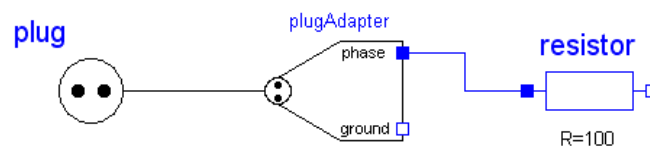
Table 3.10: Adapter model for Plug connector.



This adapter model consists just of a connection from “phase” to “plug.phase” and from “ground” to “plug.ground”. Note, that the plug connector in the “icon layer” has the double size of a pin icon, since the Plug connector was defined with “Scale = 0.2” in the icon layer.

The small demo model from Figure 3.15 can be implemented with the adapter model in the following way:

Figure 3.18: Connecting the pin of a resistor to a Plug connector with an adapter model.



## Chapter 4 Initialization

In this chapter it is discussed how models are initialized in Modelica and Dymola before a simulation is started.

### 4.1 Mathematically defining the initialization problem

A Modelica model is a declarative description of the desired abstraction of the physical world. Such a model can be utilized for various purposes. The most important usage (but not the only one) is the solution of an initial value problem. For an ordinary differential equation in state space form (abbreviated as ODE), an initial value problem is mathematically defined as:

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t), t), \quad \mathbf{x}(t=t_0) = \mathbf{x}_0, \quad t \in \mathbb{R}, \quad \mathbf{x} \in \mathbb{R}^{nx} \quad (4.1)$$

This means that the ODE  $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, t)$  is numerically solved, starting at the initial time  $t_0$  and from the given initial value of the state  $\mathbf{x}_0$ . If no initial values  $\mathbf{x}_0$  would be defined, the ODE has an infinite number of solutions and therefore the problem description is not well defined. Consequently, initial values  $\mathbf{x}_0$  must be defined (directly or indirectly) before a simulation can be carried out.

As will be explained in more detail in section 12.1, a Modelica model containing only continuous equations is mapped to the following differential algebraic equation system (abbreviated as DAE):

$$\mathbf{0} = \mathbf{f}(\dot{\mathbf{x}}(t), \mathbf{x}(t), \mathbf{y}(t), t), \quad t \in \mathbb{R}, \quad \mathbf{x} \in \mathbb{R}^{nx}, \quad \mathbf{y} \in \mathbb{R}^{ny}, \quad \mathbf{f} \in \mathbb{R}^{nx+ny} \quad (4.2)$$

where  $\mathbf{x}$  are “nx” variables that appear differentiated (i.e., the “**der**(..)” operator is applied on them) and  $\mathbf{y}$  are all other “ny” unknown variables. “Initialization” of this DAE means to compute initial values of all variables so that the DAE is fulfilled at the initial time:

$$\mathbf{0} = \mathbf{f}(\dot{\mathbf{x}}_0, \mathbf{x}_0, \mathbf{y}_0, t_0) \quad (4.3)$$

with  $\dot{\mathbf{x}}_0 = \dot{\mathbf{x}}(t=t_0)$ ,  $\mathbf{x}_0 = \mathbf{x}(t=t_0)$ ,  $\mathbf{y}_0 = \mathbf{y}(t=t_0)$ . Besides of an initial value problem (also called simulation problem), also other operations on a model require first appropriate initialization, for example, the linearization around a stationary point in order to compute eigenvalues or frequency responses. Therefore, initialization is usually the first operation to be carried out on a Modelica model, before any other action is performed.

Since  $\mathbf{f}(\cdot)$  are  $nx+ny$  equations and there are  $2*nx + ny$  unknowns  $\dot{\mathbf{x}}_0$ ,  $\mathbf{x}_0$ ,  $\mathbf{y}_0$ , “usually”  $nx$  additional conditions are needed to uniquely compute all the unknowns at the predefined initial time  $t_0$ . If the Jacobian matrix  $\mathbf{J}$  of  $\mathbf{f}(\cdot)$ :

$$\mathbf{J}(t_0) = \left[ \frac{\partial \mathbf{f}}{\partial \dot{\mathbf{x}}} : \frac{\partial \mathbf{f}}{\partial \mathbf{y}} \right] \quad (4.4)$$

is regular at the initial time, exactly “nx” (independent) additional conditions are needed. If the Jacobian matrix  $\mathbf{J}$  is singular, less than “nx” (independent) additional conditions are required. The details why this is the case and how this is algorithmically handled by a tool is explained in Chapter 12. For the moment, we assume that the Jacobian is regular and that “nx” additional conditions are needed.

The most simple initialization problem is defined by providing initial values for the variables  $\mathbf{x}$ . The remaining unknowns are then computed from the model equations:

$$\begin{aligned} &\text{known: } t_0, \mathbf{x}_0 \\ &\text{solve } \mathbf{0} = \mathbf{f}(\dot{\mathbf{x}}_0, \mathbf{x}_0, \mathbf{y}_0, t_0) \text{ for } \dot{\mathbf{x}}_0, \mathbf{y}_0 \end{aligned} \quad (4.5)$$

This means that a non-linear system of  $nx + ny$  equations has to be solved to compute the unknowns  $\dot{\mathbf{x}}_0$ ,  $\mathbf{y}_0$ .

In most practical cases, many equations can be explicitly solved and only a rather small subset of the equations will be “really” non-linear, requiring a numerical non-linear equation solver. As will be explained in Chapter 12, this is exactly the same equation system that has to be solved at every time instant, too. Therefore, this is a rather unproblematic situation for a Modelica tool.

It is often not practical to require from a modeler to define initial values for all variables on which the **der**(..) operator is applied. Instead, it is often easier to provide initial values on other variables. It might be even necessary to provide additional initial equations. In the most general case, “nx” additional (initial) equations **g**(...) have to be provided to formulate the “initialization problem”:

$$\begin{aligned} &\text{known: } t_0 \\ &\text{solve } \mathbf{0} = \begin{bmatrix} \mathbf{f}(\dot{\mathbf{x}}_0, \mathbf{x}_0, \mathbf{y}_0, t_0) \\ \mathbf{g}(\dot{\mathbf{x}}_0, \mathbf{x}_0, \mathbf{y}_0, t_0) \end{bmatrix} \text{ for } \dot{\mathbf{x}}_0, \mathbf{x}_0, \mathbf{y}_0 \end{aligned} \quad (4.6)$$

This means that a non-linear system of  $2 \cdot nx + ny$  equations has to be solved to compute the unknowns  $\dot{\mathbf{x}}_0, \mathbf{x}_0, \mathbf{y}_0$  at the initial time  $t_0$ . This means that a completely different equation system has to be solved at the initial time as at all other time instants. In Dymola, this is handled by generating special (symbolically pre-processed) code for the initialization problem.

An important special case of this general initialization formulation is “stationary initialization”, also called “steady-state initialization”. Intuitively, this means that the system shall start “in rest”. Mathematically, this means that all derivatives are zero, i. e., the additional equations **g**(...) are simply:

$$\mathbf{0} = \dot{\mathbf{x}}_0 \quad (4.7)$$

In many cases this is what the modeler would like to have. Unfortunately, it is sometimes not possible to set all state derivatives to zero. Reasons are:

- A PI-controller is present. This controller is described by the equation  $\dot{x} = u/T$ ,  $y = k \cdot (x + u)$ . If during initialization  $\dot{x} = 0$  is required, then this implies that the input signal  $u = 0$ . However, “ $u$ ” is defined somewhere else, and then during initialization there are two equations for “ $u$ ”. In such a case, it is not possible to include the equation  $\dot{x} = 0$  as initialization equation.
- An aircraft flying in the air is basically described by the equations  $\dot{h} = v$ ,  $\dot{v} = f(h, v, \dots)$  where  $h$  is the height over ground and  $v$  is the forward velocity. Additionally, there are other equations describing e.g., the aerodynamics and the rotation of the aircraft. Stationary initialization for such an aircraft model would require that  $\dot{h} = 0$ ,  $\dot{v} = 0$ , i. e., the forward velocity of the aircraft is zero. However, this is not possible, because an aircraft can only fly if the forward velocity is sufficiently large. The only meaningful way of initialization is here to define the initial height  $h = h_0$  (instead of setting the derivative of the height to zero).
- A drive system driven by an electrical motor shall usually not be initialized in steady-state (which would mean that the drive system is at rest), but in such a way that the load drives with a pre-defined speed. For synchronous or asynchronous electrical motors this means that voltages and currents in the motor have a sine-wave behavior which is obviously far away from “steady-state”. Changing the model variables by a non-linear state transformation (the so called “Park transformation”) so that the transformed voltages and currents are defined with respect to the rotating coordinate system of the motor inertia, gives the desired initialization by setting the derivatives of the transformed quantities to zero.

Whenever a non-linear system of equations is solved, “guess” values have to be defined for the “iteration” variables. The reason is that non-linear equation solvers are only able to compute a solution, if start values for the “iteration” variables are provided that are “sufficiently” close to the solution.

In Modelica, it is possible to formulate the above initialization problems by providing the missing information in the models. The two basic approaches are discussed in the next sections.

## 4.2 Initialization with attributes

With the attribute “start” an initial value of an arbitrary variable can be defined. The start value must be a literal number or an expression that contains (directly or indirectly) only references of parameters and constants. With the additional Boolean attribute “fixed” it is defined, whether the corresponding start value is the initial value of the corresponding variable (**fixed = true**), or whether the start value is just used as a

guess value (`fixed = false`), i. e., as first value of the variable, before the initialization problem is solved. The default value of `fixed` is `false`, i. e., it has to be explicitly set to `true`, if an initial value for a variable shall be defined<sup>5</sup>. As an example we analyze a simple first order system with the transfer function

$$Y(s) = \frac{k}{T \cdot s + 1} \cdot U(s) \quad (4.8)$$

Transforming this transfer function into a differential equation results in

$$T \cdot \dot{y} + y = k \cdot u \quad (4.9)$$

Since this differential equation has one variable that appears differentiated ( $y$ ), we need to provide one initial condition. The goal is here to provide an initial value  $y_0=1$ . The appropriate model can be defined with Modelica in the following way:

```
model FirstOrder1 // initialize via the start attribute
  parameter Real k "gain";
  parameter Real T "time constant";
  RealInput u "input signal";
  RealOutput y(start = 1, fixed = true) "output signal";
equation
  T*der(y) + y = k*u;
end FirstOrder1;
```

This model defines the following two equations for the initialization phase:

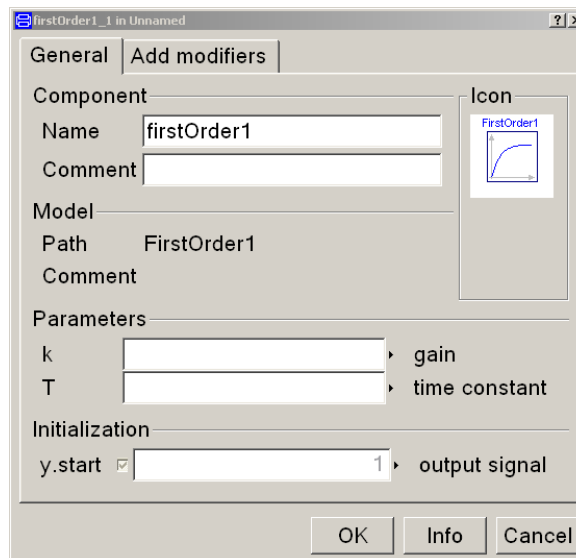
```
y = 1;
T*der(y) + y = k*u;
```

Under the assumption that the initial value of the input variable  $u$  is computed somewhere else, the two equations above have the solution:

```
y := 1;
der(y) := (k*u - y)/T;
```

and this is the solution of the initialization problem. Dragging the above model “FirstOrder1” in a model as “firstOrder1” and double clicking on the model icon in Dymola, opens the following parameter menu that is automatically constructed from the definition above:

Figure 4.1: Automatically constructed parameter menu with start and fixed attribute for  $y$



As can be seen, not only “parameters” but also start values are automatically introduced into the parameter

<sup>5</sup> To be precise, for parameters the default value of “`fixed = true`”.

menu. Note, that only start values are introduced that are explicitly defined within the model by using the “start” attribute. All start values are collected together under a “group” with the heading “Initialization”. In the input field, the desired start value can be given. At the left side of the input field, a small “checkbox” is present that defines how the start value shall be interpreted. Possible values are:

Table 4.1: Interpretation of check box for setting of attribute fixed.

<input checked="" type="checkbox"/> y.start	fixed = <b>true</b> , i. e., $y = y.start$ is used as initial equation
<input type="checkbox"/> y.start	fixed = <b>false</b> , i. e., $y$ is assigned the value $y.start$ before the initialization computation is started ( $y.start$ is a guess value)
<input checked="" type="checkbox"/> y.start	the “fixed” attribute is not set as a <i>modifier</i> (neither fixed = <b>true</b> nor fixed = <b>false</b> ). In “grey” color the current value of the fixed attribute is shown as defined in the model class. Here, fixed = <b>true</b> , because in model firstOrder1 fixed = <b>true</b> is set.

It is also possible to include different initial conditions in the same model by setting the fixed attribute correspondingly. For example, in the following model the user can either use the initial value of  $y$  as initial condition, or he can initialize in steady-state by setting the start value of **der** ( $y$ ) to zero:

```

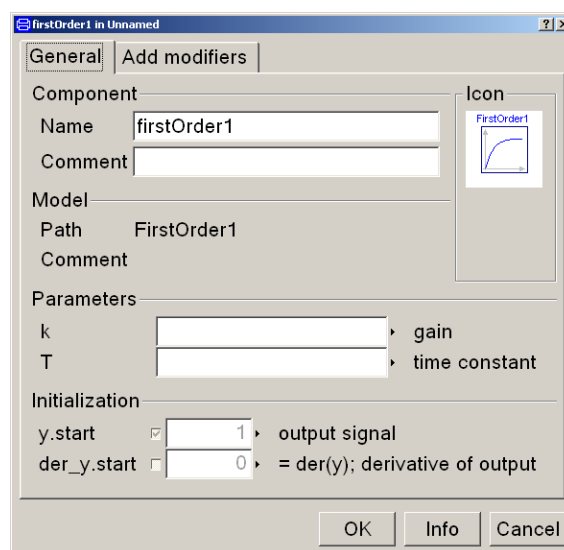
model FirstOrder2 // initialize with y.start or in steady-state
  parameter Real k "gain";
  parameter Real T "time constant";
  RealInput  u "input signal";
  RealOutput y (start = 1, fixed = true) "output signal";
  Real      der_y(start = 0, fixed = false)
                                     "= der(y); derivative of output"

equation
  der_y = der(y);
  T*der_y + y = k*u;
end FirstOrder2;

```

When using the default setting of this model, the initial equation “ $y = 1$ ” is used during initialization. When clicking on the icon of an instance of this model, the following parameter menu opens:

Figure 4.2: Parameter menu with start and fixed attributes for  $y$  and  $der(y)$ .



When marking the check box of `der_y.start` and removing the mark for `y.start`, then the model initializes in steady-state:

Since the feature to include the “start” value definition automatically into the parameter menu is simple to

Figure 4.3: Steady-state initialization with *der\_y.fixed* attribute.

y.start	<input type="checkbox"/>	1	output signal
der_y.start	<input checked="" type="checkbox"/>	0	= der(y); derivative of output

define and the user interface is very clear for someone using such a model, this is the *recommended* way to define initialization in Modelica models. Defining initialization via the “initial equation” section approach as discussed in section 4.3 below, should only be performed in special cases.

### 4.3 Initialization with initial equations

Arbitrary initial equations **g(...)** can be defined in an “**initial equation**” section. All equations present in such a section are only evaluated during initialization and are ignored otherwise. This approach is demonstrated with the first order system from above. This time, the goal is steady-state initialization of the system:

```

model FirstOrder3 // initialize always in steady state
  parameter Real k "gain";
  parameter Real T "time constant";
  RealInput u "input signal";
  RealOutput y "output signal";
equation
  T*der(y) + y = k*u;
initial equation
  der(y) = 0;
end FirstOrder3;

```

This model defines the following two equations for the initialization phase:

$$\begin{aligned} \text{der}(y) &= 0; \\ T \cdot \text{der}(y) + y &= k \cdot u; \end{aligned}$$

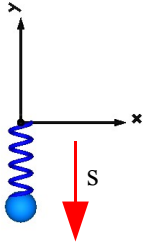
Under the assumption that the initial value of the input variable  $u$  is computed somewhere else, the two equations above have the solution:

$$\begin{aligned} \text{der}(y) &:= 0; \\ y &:= k \cdot u; \end{aligned}$$

and this is the solution of the initialization problem.

Let us analyze a slightly more complicated system: a mass point in a gravity field that is attached with a spring to the environment. The goal is to initialize the spring-mass system in “steady-state”, i.e., after initialization the system shall be in rest. The Modelica model of this model is shown below together with an animation that has been constructed with the Modelica.Mechanics.MultiBody library:

Table 4.2: Steady state initialization of spring-mass system.

	<pre> <b>model</b> SpringMassSystem // steady state initialization   <b>import</b> SI = Modelica.SIunits;   <b>parameter</b> SI.Mass m "mass of mass point";   <b>parameter</b> SI.Position s0 "unstretched spring length";   <b>parameter</b> SI.Acceleration g = 9.81 "gravity acceleration";   <b>parameter</b> SI.TranslationalSpringConstant c;   SI.Position s "distance of mass from attachment point";   SI.Velocity v "= der(s); speed of mass"; <b>equation</b>   <b>der</b>(s) = v;   m*<b>der</b>(v) = m*g - c*(s - s0); <b>initial equation</b>   <b>der</b>(s) = 0;   <b>der</b>(v) = 0; <b>end</b> SpringMassSystem; </pre>
---	--

During the initialization phase, this model is described by the 4 initialization equations:



```

der(s) = v;
m*der(v) = m*g - c*(s - s0);
der(s) = 0;
der(v) = 0;

```

Solving this system of 4 equations for the 4 unknowns' results in the following sequence of assignment statements to compute the initial values of these variables:

```

s := s0 + m*g/c;
v := 0;
der(s) := 0;
der(v) := 0;

```

It is also possible to include different initial conditions in the same model, by using an if-clause in the initial equation section. Example:

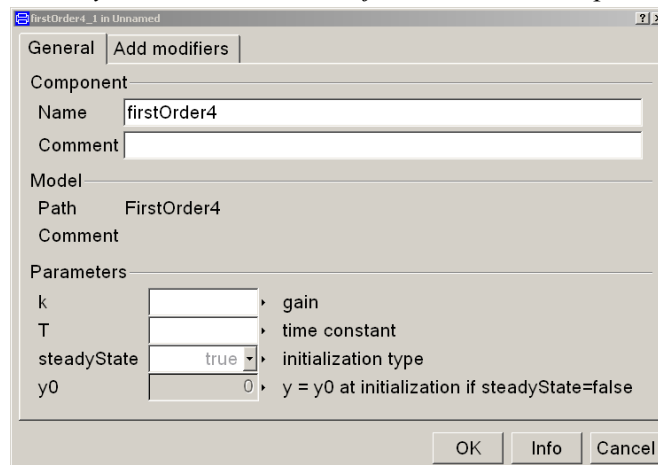
```

model FirstOrder4 // initialize with y0 or in steady state
  parameter Real k "gain";
  parameter Real T "time constant";
  parameter Boolean steadyState = true "initialization type";
  parameter Real y0 = 0
    "y = y0 at initialization if steadyState=false"
    annotation(Dialog(enable=not steadyState));
  RealInput u "input signal";
  RealOutput y "output signal";
equation
  T*der(y) + y = k*u;
initial equation
  if steadyState then
    der(y) = 0;
  else
    y = y0;
  end if;
end FirstOrder4;

```

Via parameter “steadyState” the type of initialization is defined. Depending on this condition, a value for parameter y0 is or is not needed. In order to make this more clear in the graphical user interface, the annotation “Dialog(enable=**not** steadyState)” is included for y0. This is a standard Modelica annotation and defines that the input field for parameter y0 is only enabled when the “enable” attribute is **true**. In other words, it is not possible to define a value for y0 in the parameter menu, if parameter steadyState has value **true**. The parameter menu of the above model has the following layout in Dymola:

Figure 4.4: Steady-state initialization defined with Boolean parameter.



#### 4.4 Too many or not enough initial conditions

It is quite a lot of work to define *exactly* the *required initial conditions* for a larger model. If too many initial conditions are provided, there exists usually no solution of the initialization problem (because there are more equations as unknown quantities) and therefore a Modelica tool has to reject such a model. For example, we might add the equation “ $s = s_0$ ” to the initial equation section of the SpringMassSystem model above. During translation, Dymola prints then the following error message:

Figure 4.5: Error message of Dymola if too many initial conditions are defined.

Translation of [SpringMassSystem](#):  
 DAE having 2 scalar unknowns and 2 scalar equations.  
 Error: The initial conditions for variables of type Real are overspecified.  
 There are 1 too many scalar conditions.  
 To correct it you can  
 remove Initial equations:  
 equation  
 der(v) = 0;  
 s = s0;

In the error message it is stated that there is one initial condition too much and that therefore one initial condition has to be removed. Dymola tries to identify which initial conditions are the critical ones. Therefore, in the above message only 2 of the 3 initial conditions are listed and one of the listed initial equations has to be removed.

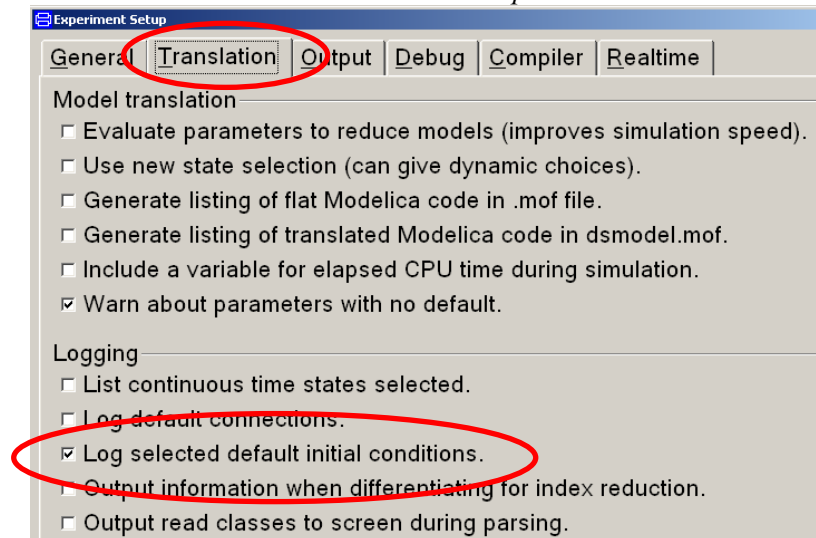
Much more often, the modeler defines none or not enough initial conditions. In such a case, there are usually an infinite number of solutions of the initialization problem. Since Dymola 7.1 a warnings message is printed in the “Messages” window in such a case:

Figure 4.6: Warning message of Dymola if not enough initial conditions are defined.

Translation of [SpringMassSystem](#):  
 The DAE has 2 scalar unknowns and 2 scalar equations.  
 Warning: The initial conditions are not fully specified.  
 Dymola has selected default initial conditions.  
 LogDefaultInitialConditions = true; gives more information.

Additionally, Dymola applies some suitable heuristics to construct a well-defined initialization problem. In many cases, this is the problem that the user would like to solve. However, it might also be that the initialization problem constructed by Dymola is far away from the expectations of the modeler. Furthermore, the model is no longer portable between different Modelica tools, because in the Modelica Language Specification it is not defined in which way a tool should act, if not enough initial conditions are provided. It is very likely that different Modelica tools use different heuristics or different warning/error messages strategies. For these reasons, it is highly recommended to always enable the option “Log selected default initial conditions” in the “Simulation / Setup / Translation” menu of Dymola, see Figure 4.7.

Figure 4.7: Dymola menu to enable “Log selected default initial conditions” in “Simulation / Setup / Translation”



If enabled (which is not the default setting), a warning message is printed *listing the initial conditions* that have been automatically added by Dymola, if the user did not provide enough initial conditions. Assume for example, that no initial conditions are defined for the “SpringMassSystem” above and that Dymola translates this modified system with the enabled “Log selected default initial conditions” option. In the “Messages” window the warning message of Figure 4.8 is printed.

Figure 4.8: Warning message from Dymola if not enough initial conditions are defined.

```
Translation of SpringMassSystem:
DAE having 2 scalar unknowns and 2 scalar equations.
Warning: The initial conditions for variables of type Real are not fully specified.
Assuming fixed default start value for the continuous states:
s(start = 0)
v(start = 0)
```

In the message it is stated that 2 initial conditions are missing and that Dymola added the initial equations “ $s = 0$ ” and “ $v = 0$ ”.

A useful strategy for initialization is:

1. Define either no initial conditions or define initial conditions where it is clear that they have to be applied.
2. Translate the model with the “Log selected default initial conditions” option enabled. Dymola will print the number of missing initial conditions and will provide suitable proposals for the missing initial conditions.
3. Include the initial condition proposals of Dymola in the model, or if necessary modified ones. A new translation of the model should then no longer result in missing initial conditions.

It might be that some of the provided initial equations are “redundant”, i.e., include exactly the same information. If this is a “structural” property of the initialization problem, Dymola prints an error message due to a “*singular*” system or due to an “*overspecified*” system and rejects the model. For example, if the initial equations “ $\text{der}(s) = 0$ ” and “ $v = 0$ ” are provided for the “SpringMassSystem” above, the error message of Figure 4.9 is printed:

Figure 4.9: Error Message in Dymola for a redundant set of initial conditions.

Translation of [SpringMassSystem](#):  
 DAE having 2 scalar unknowns and 2 scalar equations.  
 Error: The initial conditions for variables of type Real  
 are overspecified.  
 There are 1 too many scalar conditions.  
 To correct it you can  
 remove Initial equations:  
   equation  
   der(s) = 0;  
   v = 0;

The reason is that “ $v = \text{der}(s)$ ” and therefore the same initial condition is given twice. After removing one of the two equations, a warning message will be printed that one initial condition is missing. Therefore, another round is needed to supply the last missing initial condition.

If the “redundancy” of the provided initial conditions is not a structural property, it is not possible to detect this defect during translation. Note, if redundant equations are present, in most cases either an infinite number of solutions exists or no solution at all. It might be that the numerical initial equation solver will still find a solution. It is, however, likely that the numerical solver will fail because no unique solution exists.

The heuristics of Dymola to construct missing initial conditions operates roughly in the following way:

1. If  $m$  initial conditions are provided, Dymola adds  $nx - m$  additional initial conditions by providing start values for a suitable subset of variables  $\mathbf{x}$  (i.e., the variables on which the `der(.)` operator is applied). Since there are  $nx$  variables  $\mathbf{x}$ , this selection is usually not unique. In any case, the selection is made such that the resulting initial equation system is structurally non-singular (this concept is explained in detail in Chapter 12). This is a necessary condition in order that a *unique* solution of the initialization problem exists.
2. If there are several choices, Dymola gives the highest priority for a selection, if a “start” value is set for the corresponding  $\mathbf{x}$  variable. This is a good heuristic, because modelers often “forget” to define “fixed = true” after setting a start value. If there are not enough start-values where the fixed attribute is not set or is set with `fixed = false`, Dymola uses an arbitrary selection for the remaining subset of  $\mathbf{x}$  and initializes these variables with a start value of zero.
3. If there are more subset variables of  $\mathbf{x}$  available that have a “start” value (but *not* with `fixed = true`), priorities are used for the selection that depend on the “model level” where the start variables are set. “Start” values set on a “high” level of a model have a higher priority over “start” values set on a “lower” level of a model.

The heuristics of Dymola works quite well, if zero start values are “good” initial conditions for a subset of  $\mathbf{x}$ . This is for example the case for drive trains, where the absolute angle and the absolute angular velocity of drive train elements are used as variables  $x_i$ . The reason is that zero angles and zero speed for all shafts of a drive train, are meaningful initial conditions (= the drive train is fully at rest).

This works also quite well for electrical circuits, because voltage drops or currents flowing through electrical components are usually utilized as variables  $x_i$ . Zero voltage drop and/or zero current flow are meaningful initial conditions because no currents are flowing and therefore the circuit is at rest.

For thermal or fluid systems, the Dymola heuristics would be completely useless, without taking appropriate actions. The reason is that absolute pressure, temperature (in Kelvin), or density are used as variables  $x_i$  and zero values of these variables are completely outside of the operating range of “usual” models. For this reason, in every medium model “Modelica types” with medium-specific start values are defined for the relevant variable categories and these types are used in the declaration section of a model. For example, the following default types for media are defined in `Modelica.Media.Interfaces.PartialMedium`:

```
type AbsolutePressure = Modelica.SIunits.AbsolutePressure (
  min      =0,
  max      =1.0e8,
  nominal=1.0e5,
  start    =1.0e5);
```

```

type Temperature = Modelica.SIunits.Temperature (
  min      =1,
  max      =1.0e4,
  nominal=300,
  start    =300);

```

As a result, all relevant variables have start values that are meaningful for the corresponding medium and therefore there is a chance that the heuristics of Dymola or of another Modelica tool works.

The heuristics of Dymola can be changed by providing the following setting in the command window of Dymola:

```
Advanced.DefaultSteadyStateInitialization = true
```

If this option is selected, missing initial conditions are selected so that a subset of the equations  $\text{der}(x) = 0$  is added to the initialization equation system. Again, the equation subset is selected, so that the initialization problem is not structurally singular. This means that a model is initialized in “steady-state”, if no initial conditions are provided and if this option is enabled. Example:

```

model FirstOrderInSeries
  parameter Real k;
  parameter Real T;
  RealInput  u;
  RealOutput y1(start=1, fixed=true);
  RealOutput y2;
equation
  T*der(y1) + y1 = k*u;
  T*der(y2) + y2 = y1;
end FirstOrderInSeries;

```

This model has 2 variables appearing differentiated ( $y1$ ,  $y2$ ) and therefore 2 initial conditions have to be provided (we assume that the initial value of the input signal  $u$  is defined somewhere else). In the model, only 1 initial condition is defined and therefore 1 initial condition is missing. If the above option is set, Dymola will add the initial equation “ $\text{der}(y2) = 0$ ”. Note, it is not possible to select the other potential initial equation “ $\text{der}(y1) = 0$ ” because otherwise the second differential equation cannot be properly initialized and since this is a structural property, it can be detected during the translation phase. As a result, the following 4 equations define the initialization problem:

```

      y1 = 1;
      der(y2) = 0;
T*der(y1) + y1 = k*u;
T*der(y2) + y2 = y1;

```

This system of equations has the following solution:

```

      y1 := 1;
      y2 := k*y1;
der(y1) := k1*u;
der(y2) := 0;

```

## 4.5 Initialization fails

In the previous sections it was explained how to formulate well-defined initialization problems in Modelica using Dymola. This is, unfortunately, sometimes not enough, because the numerical solution of the initialization equations may fail. In this section we will analyze reasons for the failure and will discuss what to do in order that initialization becomes successful. Modelers will usually be upset if a tool claims that it cannot solve the initialization problem. In many cases the tool is right, that a numerical solution of the initialization problem is either very difficult or is impossible to obtain. Therefore, the basic remedy is to change or to improve the definition of the initialization problem (changing the tool will not help in such a case).

### 4.5.1 Guess values for iteration variables missing

Problems often occur because *non-linear algebraic* equations are solved during initialization. Especially for

steady-state initialization, large non-linear algebraic equation systems appear. For example, if the ODE (4.1) is initialized in steady state, we have a system of  $n_x$  coupled non-linear algebraic equations with  $\mathbf{x}_0$  as unknowns (= iteration variables):

$$\mathbf{0} = \mathbf{f}(\mathbf{x}_0, t_0) \quad (4.10)$$

Every numerical solver for non-linear algebraic equations requires *guess* values for the used *iteration* variables. In Dymola the number of iteration variables is nearly always much smaller as the total number of unknown variables (for details, see the tearing method in section 12.3). A modeler has to provide suitable guess values for these iteration variables otherwise it is highly likely that the initialization problem will fail with any numerical solver. Guess values are provided with the `start` attribute (and either not providing a value for attribute `fixed` or setting `fixed = false`). If a guess value is not provided, the corresponding iteration variable starts with zero. Initialization may fail because, e.g., the solver converges to a “minimum” from a given guess value, or converges to the wrong solution, if the system of equations has several solutions.

It is not known in advanced which iteration variables Dymola will select during code generation and it might even be that a small change of a model will result in different sets of iteration variables and therefore guess values for other variables need to be provided. If *no guess value* is provided for an *iteration variable*, Dymola prints a warning message during code generation. Example:

Figure 4.10: Warning message from Dymola if no guess values are provided for iteration variables.

Warning: The following variables are iteration variables of the initialization problem:  
 y  
 but they are not given any explicit start values. Zero will be used.  
 Finished  
 WARNING: 1 warning was issued

This is a serious issue and modelers should always provide *start* values for the variables shown in such warning messages, since in most cases the default start value of zero is not a good choice as a guess value.

Whenever a start value of a variable is set in a model, it appears in the parameter menu of this model and can be easily set in the parameter menu, see section 4.2. Dymola tries to select variables as iteration variables for which start values are provided. If there are several choices, Dymola selects the variable for which the start value is given on the “highest” level of the model hierarchy.

Still, there are cases where Dymola selects an iteration variable for which no start value is defined. In such a case, the start value does *not* appear in the *parameter menu* of the corresponding model. If the user has access to the underlying Modelica library and providing a start value for the corresponding variable makes sense in general, then the best action is to include a suitable start value in the library (in order that a different start value can be conveniently set in the parameter menu). If it is not possible to provide a start value in general, it is best to just add a *start-value dialog* to the *parameter menu* with the `__Dymola_initialDialog` annotation. For example, the Modelica.Mechanics.Rotational.Components.Inertia component is defined as:

```
model Inertia
  import SI = Modelica.SIunits;
  parameter SI.Inertia J(min=0);
  SI.Angle phi annotation(Dialog(group="Initialization",
    __Dymola_initialDialog=true));
  SI.AngularVelocity w annotation(Dialog(group="Initialization",
    __Dymola_initialDialog=true));
  SI.AngularAcceleration a annotation(Dialog(group="Initialization",
    __Dymola_initialDialog=true));
  ...
end Inertia;
```

This definition leads to the following parameter menu:

Figure 4.11: Parameter menu of model *Inertia* using the `__Dymola_initialDialog` annotation.

The Modelica standard annotation `Dialog(group="Initialization")` defines that an input field of the corresponding variable is placed in a group with the name “Initialization” in the parameter menu. The Dymola-specific annotation `__Dymola_initialDialog = true` defines that this input field is for the start attribute of the variable and that additionally a check-box for the “fixed”-attribute shall be added.

Note, the start attribute is still undefined with these definitions and therefore if either `phi`, `w`, or `a` is selected as iteration variable and no start value is provided when using model *Inertia*, then Dymola prints a warning message and the user gets diagnostics for this case. If an explicit start value would be provided in model *Inertia*, then *no warning message* would be printed if the variable is selected as iteration variable. Most likely, this default start value would not be suited as guess value for the iteration variable and initialization might fail.

If it is not possible to change the library (to introduce a start value dialog of the iteration variable in the parameter menu) or if the selected iteration variable is too special for the problem at hand, the start value has to be provided as modifier in the model to be simulated. Such a modifier can be provided in the “Add modifiers” tab of the parameter menu of the corresponding model instance, see the example in Figure 4.12.

Figure 4.12: Adding a start value as modifier in the “Add modifiers” tab in a parameter menu.

This introduces an input field for the start value of variable “y” in this parameter menu and gives the start value a default of 2 (the value of 2 will then be used as guess value for the iteration variable). The next time when this parameter menu is opened, it is possible to directly change the start value in the parameter menu.

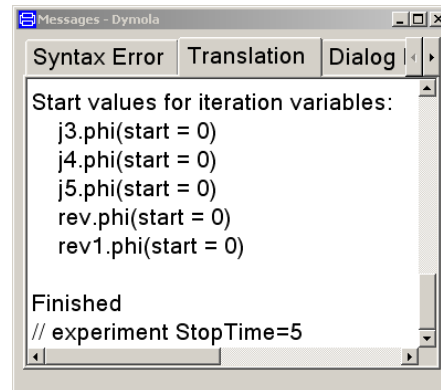
Above we have discussed how to provide guess values of iteration variables in order to define a reasonable setup of the initialization problem. If initialization still fails, the first action is to inspect the actual values of all guess values of the iteration variables. This is performed by setting the command

```
Advanced.LogStartValuesForIterationVariables = true
```

in the input field of the command window (at the bottom of the simulation window) and translating the model. Dymola reports the iteration variables and their start values in the message window. For example, when translating the example model `Modelica.Mechanics.MultiBody.Examples.Loops.Fourbar1` and the above option has been set, then the following output is presented in the message window:



Figure 4.13: Iteration variables and their start values in the message window.



Therefore, this model has the iteration variables `j3.phi`, `j4.phi`, `j5.phi`, `rev.phi`, `rev1.phi` and as guess value for every iteration variable zero is used.

The list of start values for the iteration variables should be carefully inspected. It suffices that one start value is not meaningful in order that initialization fails.

#### 4.5.2 Difficult nonlinear equations

There are nonlinear algebraic equation systems that are difficult to solve. Such systems can occur, e.g., if components have very steep characteristics such as diodes in electrical circuits or pressure drop components for turbulent flow in fluid networks. The problem is that in some region very small changes of the iteration variables lead to very large changes of other variables. During initialization this is quite critical, because the guess values of the iteration variable are not yet good enough and they might be too far away from the solution. During simulation this is less critical because good initial guesses for the iteration variables are available from the solution of the last integration step and the integrator has just to decrease the step size in order to find a solution.

One solution strategy in such a case is to start the model to be simulated in an operating point where initialization is possible. After initialization, the sources are controlled such that they drive the system to the desired operating point. For example, electrical circuits may be initialized with zero voltage sources and after initialization the voltage sources are ramped up to the desired voltages. Fluid systems might be initialized with zero mass flow rates and after initialization mass flow sources or pumps are ramped up to the desired mass flow rates.

#### 4.5.3 Inconsistent redundant equations

The algebraic equations appearing during initialization might be redundant without having a solution. A very simple (constructed) example of this type is shown in the next example:

```
model InconsistentRedundantEquations
  parameter Boolean steadyStateInit=false;
  parameter Real a11=2, a12=3;
  parameter Real a21=2, a22=3;
  parameter Real b1=1;
  parameter Real b2=3;
  Real x1;
  Real x2;
initial equation
  if steadyStateInit then
    der(x1) = 0;
    der(x2) = 0;
  else
    x1 = 1;
    x2 = 2;
  end if;
equation
  der(x1) = -(a11*x1 + a12*x2 - b1);
```



```

    der(x2) = -(a21*x1 + a22*x2 - b2);
end InconsistentRedundantEquations;

```

This model can be simulated without problems if “steadyStateInit = **false**”. If “steadyStateInit = **true**”, the model shall be initialized in steady state, but fails during initialization with the error message:

```

Residual component 1 of system 1 is -1.000000e+000
Residual component 2 of system 1 is 1.000000e+000
LINEAR SYSTEM OF EQUATIONS IS SINGULAR AT TIME = 0
...Linear system of equations number = 1
...Variables which cannot be uniquely computed:
x1 = 0.307692
... NOT ACCEPTING SINCE TOO LARGE RESIDUAL

```

The message indicates that a linear system of equations is solved but this system does not have a solution. The listed variables which give rise to the problem (here: “x1”) may already indicate the source of the problem. If this is not the case, one should have a look at the equations. This is performed by selecting in the simulation tab: “Simulation / Setup ... / Translation / Generate listing of translated Modelica code in dsmodel.mof” and translating the model. Dymola generates a file “dsmodel.mof” where the symbolically processed equations are present in a Modelica-like syntax. These equations are also stored in file dsmodel.c, but the C-file is by far not as readable as the dsmodel.mof file. The dsmodel.mof file for the model above is:

```

// dsmodel.mof file for model InconsistentRedundantEquations
...
// Linear system of equations
// Matrix solution:
//   Original equations:
//   0 = b1-(a11*x1+a12*x2);
//   0 = b2-(a21*x1+a22*x2);
//
// Calculation of the J matrix and the b vector,
// but these calculations are not listed here.
// To have them listed, set
//   Advanced.OutputModelicaCodeWithJacobians = true
// before translation. May give much output,
// because common subexpression elimination is not activated.

x := Solve(J, b); // J*x = b
{x1, x2} := x;

// Torn part
// End of linear system of equations
...

```

From the description it is apparent that Variable “x1” is computed from a linear system of equations. Taking in to account the actual values of the parameters gives the following equation system:

$$\begin{aligned}
 2 \cdot x_1 + 3 \cdot x_2 &= 1; \\
 2 \cdot x_1 + 3 \cdot x_2 &= 3;
 \end{aligned}$$

It is now obvious that no value of x1 or x2 can fulfill this equation system and therefore a modeling error is present and the modeler has to re-formulate the initialization problem.

In a concrete application the equations in the dsmodel.mof file might be too large in order to analyze them manually as shown above. In such a case one should assume that Dymola is right and that the initialization problem does not have a solution and should at once try to re-formulate the initialization problem.

## Chapter 5 Advanced Modelica Language Constructs

In this chapter more advanced Modelica language constructs are discussed.

### 5.1 Arrays

Modelica supports arrays with any number of dimensions. In this section, the most important properties of arrays are introduced on the basis of examples.

#### 5.1.1 Array declaration and construction

Arrays are defined by declaring an array name with the dimensions appended in square brackets, for example:

```
Real v[3];           // Vector with 3 elements
Real M[3,4];         // Matrix with 3 rows and 4 columns
Real A[2,3,4,5];     // 4-dimensional array
```

Alternatively, the dimensions can be defined with the type, for example:

```
Real[3]   v1, v2;    // Two vectors with 3 elements each
Real[3,4] M;         // Matrix with 3 rows and 4 columns
```

Unknown dimensions are declared with “:”, for example:

```
parameter Real denominator[:]; // dimension is not yet known
parameter Real M[:, :];       // size of matrix M is not yet known
```

As will be explained in section 12.5, a Modelica environment that shall be able to process the Modelica Standard Library has to preprocess the model equations symbolically during translation. This is only possible if at translation time all equations are known, which in turn means that the number of variables and the sizes of dimensions need to be known. This implies that Modelica tools have currently the restriction that array sizes in models need to be known implicitly or explicitly during translation. An exception are functions that might be translated without knowing the array sizes of the input or output arguments. A typical scenario for a model is that array dimensions are defined with the “:” operator and when using the model, the actual array dimensions are provided.

An array element is accessed by using “[...]” as the access operator. For example “M[2,3]” is the element in the second row and the third column of matrix M. The first index of an array dimension is *one* (as in FORTRAN or Matlab), for example:

```
Real v[3]; // defines the three elements v[1], v[2], v[3]
equation
  v[1] = 1;
  v[2] = 2;
  v[3] = 3;
```

Modelica makes a clear difference between the access of an array element, such as “v[2]”, and a function call, such as “v(2)” to call function “v” with argument “2”. In FORTRAN or Matlab the two operations are not distinguishable which means that the possibilities for error diagnostics are better in Modelica for this case.

A *vector* can be constructed with the *array construction* operator “{..}”:

```
parameter Real v1[3] = {1,2,3};
Real v2[2];
equation
  v2 = {time, sin(v1[3]*time)};
```

A matrix can be constructed with the matrix constructor “[...]”:

```

parameter Real M1[2,3] = [11, 12, 13; 21, 22, 23]; // M1= $\begin{bmatrix} 11 & 12 & 13 \\ 21 & 22 & 23 \end{bmatrix}$ 
Real M2[1,3]; // row matrix
equation
M2 = [time, 2*time, 3*time];

```

which means that elements in a row are separated by a (required) comma and different rows are separated by a (required) semicolon.

An array of any dimension, including vectors and matrices, can be constructed with the array constructor “{...}”. This is achieved by assembling arrays of dimension “n” along a new first dimension and constructing thereby a new array with “n+1” dimensions. Examples:

```

// use 2 scalars to build a vector:
parameter Real v[2] = {1, 2};

// use 3 vectors to build a matrix
parameter Real M[3,2] = {v, 2*v, 3*v};

// use 4 matrices to build an array with 3 dimensions
parameter Real A[4,3,2] = {M, 2*M, 3*M, 4*M};

```

In the example, it is also shown that an array can be multiplied with a scalar according to the usual mathematical “scalar\*array” operation where all elements of the “array” are multiplied with the “scalar”. The array constructor can also be nested, e.g.

```

parameter Real M[3,2] = {{1,2}, {2,4}, {3,6}}; // = [1,2; 2,4; 3,6]

```

In Modelica, scalars, vectors, matrices, arrays of 3 dimensions, arrays of 4 dimensions etc. are all different data types. Arrays with different dimensions can only be used in an operation together, if a cast operator generates the same number of dimensions, or if there is a special built-in rule for the respective operation. Examples with cast operators:

```

Real v[3]; // Vector with 3 elements
Real M[3,1]; // Column matrix with 3 elements
equation
v = M; // error, since different number of dimensions
v = vector(M); // fine, since matrix M is transformed to a vector
M = matrix(v); // fine, since matrix(v) generates a column matrix
M = [v]; // fine, since [v] generates a column matrix

```

With the cast operator “**vector** (A)” a vector with all elements of A is generated provided there is at most one dimension size of A > 1. For a scalar s, **vector** (s) is a vector with s as element.

With the cast operator “**matrix** (A)”, a scalar A is transformed to a matrix with 1 row and 1 column, a vector A is transformed to a column matrix and an array with more than two dimensions is transformed to a matrix using the first two dimensions (all other dimensions must have size one). When a scalar or vector is used inside the matrix constructor “[...]” it is interpreted as a column matrix.

With the cast operator “**scalar** (A)”, an array is transformed to a scalar provided it consists only of one element.

The matrix constructor “[...]” can be used to construct a matrix from other scalars, vectors, matrices or arrays. Example:

```

Real v1[2] = {1, 2};
Real v2[3] = {4, 5, 6};
Real M1[3,2] = [11, 12;
                21, 22;
                31, 32];
Real M2[3,4] = [M1, [v1;3], v2];

```

The definitions on the left side construct array M2 as:

$$M2 = \begin{bmatrix} 11 & 12 & 1 & 4 \\ 21 & 22 & 2 & 5 \\ 31 & 32 & 3 & 6 \end{bmatrix}$$

The “[...]” operator is a bit more general as presented here. In the Modelica specification it is precisely

defined to concatenate arrays along the first and second dimension. The general case is not often needed and therefore we restrict our description to the straightforward applications shown above.

The *colon operator* “:” is provided to construct *regular vectors*:

```
Integer v1[:] = 1:4;    // same as {1,2,3,4}
Integer v2[:] = 1:2:7;  // same as {1,3,5,7}
```

If the operands of the colon operator are all of type `Integer`, the result is an `Integer` vector. If one of the operands is of type `Real`, the result is a `Real` vector. The colon operator is often used to define an index range, e.g., to extract an array out of another array:

```
Real M3[2,2] = M2[2:3,3:4]; // M2 from above, i.e., M3 = [2, 5; 3, 6]
Real v3[3]   = M2[2,1:4];   // M2 from above, i.e., v3 = {21, 22, 2, 5}
```

As demonstrated, a matrix is generated if the colon operator is applied to 2 index ranges (here: to construct M3) and a vector is generated, if the colon operator is only applied to one index range (here: to construct v3).

The `fill(...)` operator is provided to construct arrays of the desired type and dimensions and to fill the array with one value for each element:

```
Real    M4[:, :] = fill(2.0,2,3); // M4=[2.0, 2.0, 2.0; 2.0, 2.0, 2.0]
Boolean active[:] = fill(true,3); // active = {true, true, true}
```

The first argument defines the value that should be used for all elements and the type of the array is identical to the type of this first argument. The remaining arguments define the desired dimensions of the array to be constructed.

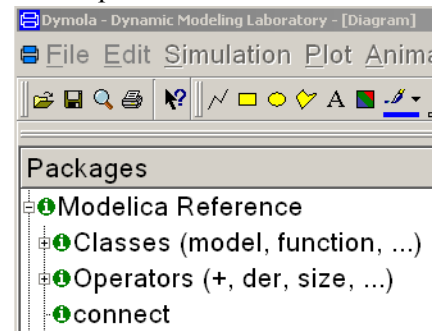
For two often occurring cases, special operators are introduced for convenience: `zeros(...)` is the same as `fill(0,...)`, and `ones(...)` is the same as `fill(1,...)`. So, in the first case an array with “zero” elements and in the second case an array with “one” elements is generated.

With the inquiry operators “`ndims(A)`”, the number of dimensions of array A are returned as `Integer`, and with “`size(A,i)`” the size of dimension i is returned (again as `Integer`). The keyword “`end`” as index of an array element characterizes the last element of a dimension. Example:

```
Real v[:];
equation
  i1 = v[size(v,1)];
  i2 = v[end]; // i2 is identical to i1
```

In the package browser of Dymola, the library

“*ModelicaReference*” is present as shown in the right figure. This is a free library from the Modelica Association that contains a short reference to all Modelica language elements. In particular, sublibrary “*Operators*” contains a description of all built-in operators, such as the operators sketched above.



### 5.1.2 Array operations

Most standard mathematical operations on arrays are available in Modelica. They are demonstrated in the following examples:

```
Real[3,2,4] A1, A2, A3;
Real        s1, s2;
equation
  // Addition/Subtraction of arrays
  A1 = A2 + A3;

  // Scalar multiplication
  A1 = s1*A2 + s2*A3;
```

The “+” and “-” operator on arrays with the same sizes have the standard interpretation to add or subtract corresponding array elements. Multiplication of a scalar with an array results in the multiplication of every element of the array with this scalar.

```

Real A[3,4], B[3,5], C[5,4];
Real v1[3], v2[4], s;
equation
  // Scalar product (vector*vector = scalar)
  s = v1*v1;

  // Matrix multiplication (matrix*matrix = matrix)
  A = B*C;

  // Matrix vector multiplication (matrix*vector = vector)
  v1 = A*v2;

```

Multiplication of two vectors is a scalar product and results in a scalar. Multiplication of two matrices is interpreted as the mathematical matrix multiplication. Multiplication of a matrix with a vector is defined as matrix multiplication where the vector is interpreted as a column matrix. The resulting column matrix is implicitly casted to a vector.

### 5.1.3 Empty Arrays

It is allowed to define arrays with zero dimensions and to use such arrays in equations, provided the equation has “no effect” because the “zero” dimensions effectively “remove” the equation. Example:

```

Real A[3,0], B[3,0], C[0,0];          // Arrays have zero dimension
parameter Real D[:,:] = fill(0.0, 0, 0); // D is empty by default
equation
  A + B = 2*A;    // fine; the equation is not present
  A + C = 2*A;    // error; arrays with different dimensions are added

```

It is not allowed to access elements of an empty array because an empty array does not have elements:

```

Real x[0];          // empty vector
equation
  x[0]*x[0] = 2;    // error, since an element x[0] does not exist

```

Empty arrays are very useful to handle limiting cases without extra effort. For example, the connector of a Fluid library might be designed for single and multiple substance fluids:

```

connector FluidPort "connector for single + multiple substance fluids"
  import SI = Modelica.SIunits;
  parameter Integer ns "number of substances";

  flow    SI.MassFlowRate    m_flow "mass flow rate";
          SI.AbsolutePressure p      "pressure";
  stream SI.SpecificEnthalpy h_outflow;
  stream Real                Xi_outflow[ns-1]; //Real[0], if ns = 1
end FluidPort;

```

For single substance fluids “ns = 1”. Therefore, array Xi\_outflow has dimension zero and is actually not present. For multiple substance fluids, the array is present. The connector is, for example, used to model a fluid volume:

```

model Volume
  import SI = Modelica.SIunits;
  FluidPort port;
  SI.Mass ms[ns-1] "substance masses" // Real[0], if ns = 1
  ...
  equation
    der(ms) = port.m_flow*actualStream(port.Xi_outflow);
                                // not present, if ns = 1

```

The volume contains especially the *mass balance* for the substance mass flow rates. If ns = 1, both the vector of substance masses “ms” and the vector of independent mass fractions port.Xi\_outflow have zero dimensions and therefore the balance equation “der(ms) = ...” is not present.

### 5.1.4 Example: General transfer function model

The introduced operations on arrays shall be demonstrated on the basis of a more complicated example: A single-input, single-output transfer function is defined as:

$$y = \frac{b_1 s^m + b_2 s^{m-1} + \dots + b_m s + b_{m+1}}{a_1 s^n + a_2 s^{n-1} + \dots + a_n s + a_{n+1}} \cdot u \quad (5.1)$$

where  $u$  is the input,  $y$  is the output, vector  $b[:]$  contains the coefficients of the numerator polynomial and vector  $a[:]$  contains the coefficients of the denominator polynomial. In order to simulate this system, it has to be transformed to a differential equation system. There is no unique transformation. We will use the controller canonical form. For  $n = m = 5$  it has the following structure:

$$\begin{aligned} \dot{\mathbf{x}} &= \begin{pmatrix} -\frac{a_2}{a_1} & -\frac{a_3}{a_1} & -\frac{a_4}{a_1} & -\frac{a_5}{a_1} \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \cdot \mathbf{x} + \begin{pmatrix} \frac{1}{a_1} \\ 0 \\ 0 \\ 0 \end{pmatrix} \cdot u \\ y &= \left( b_2 - a_2 \frac{b_1}{a_1} : b_3 - a_3 \frac{b_1}{a_1} : b_4 - a_4 \frac{b_1}{a_1} : b_5 - a_5 \frac{b_1}{a_1} \right) \cdot \mathbf{x} + \frac{b_1}{a_1} \cdot u \end{aligned} \quad (5.2)$$

For  $n = 0$  we have the special case of a pure gain:  $y = (b_1/a_1) \cdot u$ . Since several single-input, single-output blocks are present, it is useful to define super class SISO that defines the common properties (in this case the declaration of an input and an output signal). In fact such a super class is available as `Modelica.Blocks.Interfaces.SISO`:

```
partial block SISO "Single Input/Single Output block"
  Modelica.Blocks.Interfaces.RealInput u "input";
  Modelica.Blocks.Interfaces.RealOutput y "output";
end SISO;
```

The transfer function is then implemented as:

```
block TransferFunction
  extends Modelica.Blocks.Interfaces.SISO;
  parameter Real b[:]={1} "Numerator coefficients";
  parameter Real a[:]={1} "Denominator coefficients";
  output Real x[size(a, 1) - 1] "State of transfer function";
protected
  parameter Integer na = size(a, 1);
  parameter Integer nb = size(b, 1);
  parameter Integer nx = na-1;
  Real bb[:] = vector( [zeros(max(0, na-nb), 1); b] );
  Real d = bb[1]/a[1];
initial equation
  der(x) = zeros(nx);
equation
  assert(nb <= na, "Transfer function is not proper " +
    "(size(b,1) <= size(a,1) required)");
  if nx == 0 then
    y = d*u;
  else
    der(x[1]) = (-a[2:na]*x + u)/a[1];
    der(x[2:nx]) = x[1:nx-1];
    y = (bb[2:na] - d*a[2:na])*x + d*u;
  end if;
end TransferFunction;
```

The model is defined as “block” in order that it can only be used in a connection structure according to

block diagram semantic. The coefficients of the numerator and denominator polynomial are defined with parameter vectors “b” and “a” that have suitable defaults of “1”. The state “x” of the state space form shall be accessible from the outside, e.g. to set initial conditions. It has to be declared as “output” because it is computed inside the block and every public declaration in a block must have the **parameter**, **constant**, **input** or **output** prefix. The dimension of vector “x” is zero, when the denominator coefficient vector “a” has only one element.

In the “**protected**” section the needed dimensions are defined and especially the utility vector “bb”. This vector has at least the same size as the denominator coefficient vector “a” and is constructed by concatenating a zero vector with vector “b”. The basic goal is that “a” and “bb” have the same size in order that the two vectors can be used in the same operation. Therefore, the desired definition of “bb” is:

```
Real bb[na] = vector( [zeros(na-nb,1);b] );
```

Note, that [**zeros**(na-nb,1);b] is a column matrix that is transformed to a vector with the **vector**(..) cast operator. If “a” is, e.g., defined as “{1,2,3}” and “b” is defined as “{4}”, then “bb = {0,0,4}”. A user might define vector “b” with more elements as vector “a” and then the above construction gives a translation error (because the built-in function **zeros**(..) has a negative input argument which is not allowed), and this error is difficult to understand by the user. One might declare the dimension of vector “b” as

```
parameter Integer nb(max=na) = size(b, 1);
```

in order that vector “b” cannot be defined larger as “a”. Unfortunately, by default Dymola only prints a *warning message* if the “**min**” and/or “**max**” attributes are violated. Only if the non-default option “Simulation / Setup / Debug / Min/Max assertions / All variables” is selected, then an *error* occurs if the bounds defined by the “**min**” and/or “**max**” attributes are not fulfilled. Additionally, the error or warning message in the above context is not so easy to interpret because “nb” is not a variable in the public interface.

For these reasons, in the model above another approach is used: Vector “bb” is defined so that the declaration is correct, even if “b” has more elements as “a”. Furthermore, an “**assert**(...)” statement is added that triggers an error during translation (because the assert condition can be evaluated during translation since “na” and “nb” are dimensions that are fixed during translation) and provides a meaningful error message in its second argument.

The essential part of the controller canonical form is implemented with an if-clause since the standard case (here: the “**else**” branch) is not correct if the block has no states, i.e., if the dimension of vector “x” is zero. Dymola evaluates an if-clause during translation, provided the “if-condition” is a parameter expression. Only in this special case, the number of equations in the different branches of an if-clause might be different and therefore the if-clause above is correct.

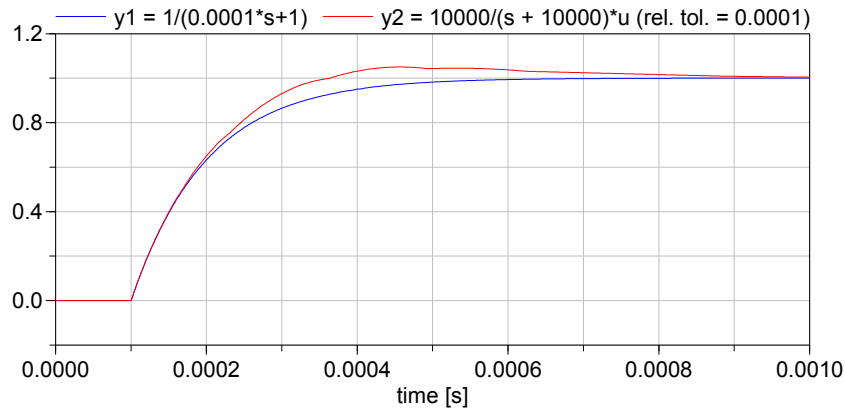
It is a bit involved to provide different useful options for the initialization of the block. Above, only the default initialization is defined that initializes the block in “steady state”. This initialization is also correct if the dimension of vector “x” is zero, because then the initialization equation is not present.

The above model works usually fine. Let us simulate the transfer functions:

$$y_1 = \frac{1}{0.0001 \cdot s + 1} \cdot u, \quad y_2 = \frac{10000}{s + 10000} \cdot u \quad (5.3)$$

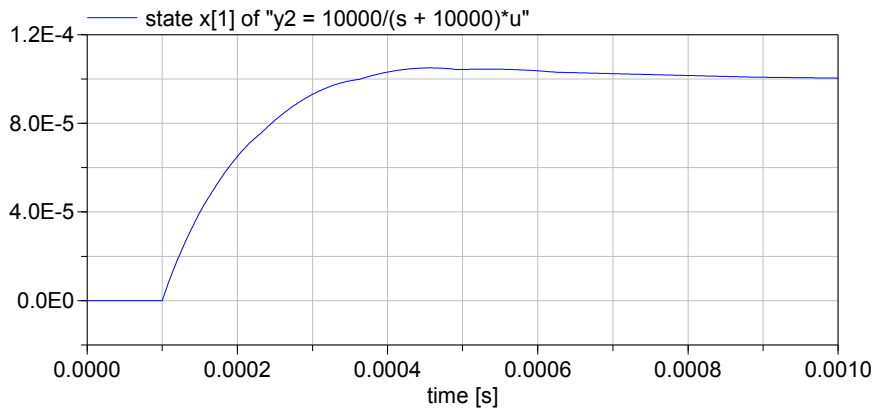
The two transfer functions are mathematically equivalent and should give the same simulation result. When simulating the step response of these systems using the default options of Dymola (integrator = “DASSL”, relative tolerance = 1e-4), different simulation results are obtained as shown in Figure 5.3 below (note, the step has to start at time > start time, because the transfer function initializes in steady state; furthermore different simulation runs have to be performed for the two cases, since otherwise the step size of y1 is also used for y2 and the problem disappears):

Figure 5.1: The same transfer function gives different results (for a relative tolerance of  $1e-4$ )



The “upper, red” curve in Figure 5.3 belongs to  $y_2$  and is wrong. The first action in such a case is to simulate with a higher tolerance (i.e., the number of correct digits is higher). Simulating  $y_2$  with a tolerance of  $1e-6$  gives the correct result of the “lower, blue” curve in the figure. The reason of this behavior can be seen when plotting the state of the  $y_2$  transfer function, see Figure 5.1:

Figure 5.2: The state  $x[1]$  of the  $y_2$  transfer function is in the order of  $10^{-4}$ .



The state  $x[1]$  is in the order of  $1e-4$ . As will be explained in Chapter 13, the step size control of a variable step size integrator is controlled by the “relative tolerance” which defines the desired number of correct digits and the “absolute tolerance” which defines when a variable is so small that the integrator can consider it as negligible. When a variable is smaller as its “absolute tolerance”, the step size control for this variable is switched off. The “relative tolerance” is defined in the “Simulation / Setup / General / Tolerance” menu of Dymola. The “absolute tolerance” is defined implicitly by the product of the “nominal” value, the “relative tolerance” and a simulator specific value, e.g.:

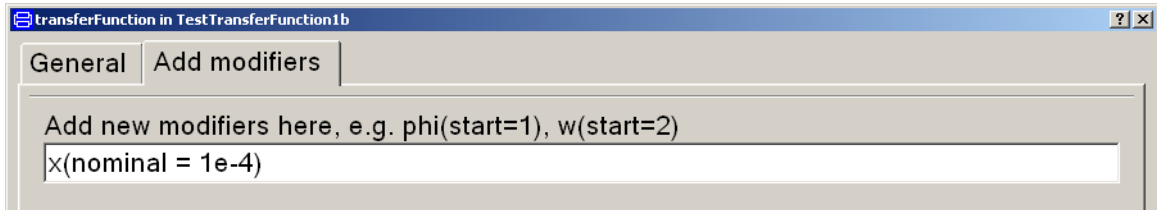
$$\text{absoluteTolerance} = \text{relativeTolerance} * \text{nominalValue} * 0.01$$

where the “nominal” value is defined by attribute “nominal” in a variable declaration. Since no “nominal” attribute is set for  $x[1]$  (i.e., the nominal value is “1”), the absolute tolerance is in the order of the relative tolerance, i.e., around  $1e-4$ . Since  $x[1]$  is permanently quite small, the step size control has not much effect and therefore the result is not reliably computed.

A quick fix of this situation is to provide a nominal value for the state vector  $x$ . In our example, this can be done by opening the parameter menu of the transfer function, selecting the “Add modifiers” tab and declaring a nominal value of  $1e-4$  for  $x$  by the statement:



Figure 5.3: The “Add modifiers” menu used to define the nominal value of a variable.



The defined nominal value is used for all elements of vector “x”. According to the Modelica specification it would be possible to define a nominal value for every element of an array. However, Dymola supports currently only one nominal value for all array elements. Simulating again with this change of the model gives the correct result, even for the default relative tolerance of  $1e-4$ .

It is, of course, desirable to treat such a situation automatically without bothering the user to provide nominal values. Unfortunately, it is always possible to construct a model where the default behavior of a library component will fail due to inappropriate nominal values, since the library has not enough information about the simulation problem to be solved. However, it is possible to enlarge the number of models that result in a reliable simulation. Basically, one has to utilize information from the numerator and/or denominator polynomial coefficients to set appropriate nominal values, e.g., in the form:

```
output Real x[size(a, 1) - 1] (each nominal = 1/a[end]);
```

The “each” prefix is explained in section 5.8. It defines that the provided value is used for all array elements. In principal, the above declaration would be a good solution. Unfortunately, Dymola has currently the significant restriction that the parameter vector “a” cannot be modified after translation any more, if an element of “a” is used to define an attribute such as “nominal”, “min” or “max”. This means that the model has to be retranslated whenever an element of “a” shall be changed.

Due to this drawback, another approach is used which is based on the scaling of the state vector so that the “critical” state is usually in the order of “1” and no longer in the order of “ $1e-4$ ” as in the example above. For this, the heuristics is used that the “magnitude of “x” is in the order of its steady state value under the assumption that the input  $u = 1$ . To get a feeling for the situation, let us compute the steady state value of the 4th order transfer function :

$$\mathbf{0} = \dot{\mathbf{x}} = \begin{pmatrix} -\frac{a_2}{a_1} & -\frac{a_3}{a_1} & -\frac{a_4}{a_1} & -\frac{a_5}{a_1} \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \cdot \mathbf{x} + \begin{pmatrix} \frac{1}{a_1} \\ 0 \\ 0 \\ 0 \end{pmatrix} \cdot u \Rightarrow \mathbf{x} = \begin{pmatrix} \frac{1}{a_5} \cdot u \\ 0 \\ 0 \\ 0 \end{pmatrix} \quad (5.4)$$

The steady state value is computed by setting the derivatives to zero. The result implies that we should use a state transformation

$$\mathbf{x} = \frac{1}{a_5} \cdot \bar{\mathbf{x}} \quad (5.5)$$

leading to the slightly changed state space form:

$$\begin{aligned} \dot{\bar{\mathbf{x}}} &= \begin{pmatrix} -\frac{a_2}{a_1} & -\frac{a_3}{a_1} & -\frac{a_4}{a_1} & -\frac{a_5}{a_1} \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \cdot \bar{\mathbf{x}} + \begin{pmatrix} \frac{a_5}{a_1} \\ 0 \\ 0 \\ 0 \end{pmatrix} \cdot u \\ y &= \left( b_2 - a_2 \frac{b_1}{a_1} \quad b_3 - a_3 \frac{b_1}{a_1} \quad b_4 - a_4 \frac{b_1}{a_1} \quad b_5 - a_5 \frac{b_1}{a_1} \right) \cdot \frac{1}{a_5} \cdot \bar{\mathbf{x}} + \frac{b_1}{a_1} \cdot u \end{aligned} \quad (5.6)$$

Setting again the state derivatives to zero results in a steady state value of:  $x_1 = u$ . Changing the Modelica

model above according to this scaling leads to a model where the discussed problem is again resolved and the simulation result is correct:

```

block TransferFunctionWithScaling
  extends Modelica.Blocks.Interfaces.SISO;
  parameter Real b[:]={1} "Numerator coefficients";
  parameter Real a[:]={1} "Denominator coefficients";
  output Real x[size(a, 1) - 1] "State of transfer function";
protected
  parameter Integer na = size(a, 1);
  parameter Integer nb = size(b, 1);
  parameter Integer nx = na-1;
  parameter Real x_scale = if abs(a[end]) > 1000*
                                     Modelica.Constants.eps*
                                     Modelica.Math.Vectors.length(a)
                                     then abs(a[end]) else 1;
  Real bb[:] = vector( [zeros(max(0,na-nb),1);b] );
  Real d      = bb[1]/a[1];
initial equation
  der(x) = zeros(nx);
equation
  assert(nb <= na, "Transfer function is not proper " +
        "(size(b,1) <= size(a,1) required)");
  if nx == 0 then
    y = d*u;
  else
    der(x[1])    = (-a[2:na]*x + x_scale*u)/a[1];
    der(x[2:nx]) = x[1:nx-1];
    y = (bb[2:na] - d*a[2:na])*x/x_scale + d*u;
  end if;
end TransferFunctionWithScaling;

```

Note, since a[**end**] might be zero, the scaling is disabled if a[**end**] is too small.

## 5.2 For Loops

In equation sections, **for** loops are used to iterate over equation sets. In the following example, a **for** loop is used to build up a filter block where the order of the filter is defined by a parameter:

```

block CriticalDampingFilter
  Modelica.Blocks.Interfaces.RealInput  u;
  Modelica.Blocks.Interfaces.RealOutput y;
  parameter Integer n(min=1)           "Order of filter";
  parameter Modelica.SIunits.Frequency f "Cut-off frequency";
  output Real x[n]                     "Filter states";
protected
  parameter Real w=2*Modelica.Constants.pi*f;
initial equation
  der(x) = zeros(n); // Steady state initialization
equation
  der(x[1]) = (u - x[1])*w;

  for i in 2:n loop
    der(x[i]) = (x[i-1] - x[i])*w;
  end for;

  y = x[n];
end CriticalDampingFilter;

```

This block has an input signal “u” and an output signal “y”. A set of differential equations is present that define a (unnormalized) critical damping filter. For a filter of order “n”, there are “n” first order differential

equations which are constructed using a **for**-loop.

Every **for**-loop in an equation section is expanded during translation in to a set of equations by iterating over the loop variable, in order that symbolic processing, see Chapter 12, is possible. In the example above the loop variable is “i” and it takes the values defined by the vector after the “**in**” keyword. Above, this vector is defined via the “colon” operator: “2:n”, i.e., “2, 3, 4, . . . , n”. Below the expanded code is shown for “n=3”:

```
// n = 3
der(x[1]) = (u - x[1])*w;
der(x[2]) = (x[1] - x[2])*w;
der(x[3]) = (x[2] - x[3])*w;
y = x[3];
```

All types of vector expressions can be used to define the values of the loop variable, as long as it is a parameter expression (in order that the **for** clause can be expanded into equations during translation). This vector expression is evaluated once for each **for** clause, and is evaluated in the scope immediately enclosing the **for** clause. Example:

```
parameter Real x = {1.1, 2.2, 3.3, 4.4};
equation
  for i1 in 1:10 loop          // i1 takes values 1,2,3,...,10
    ...
  end for;

  for i2 in {1,3,6,7} loop     // i2 takes values 1, 3, 6, 7
    ...
  end for;

  for r1 in 1.0:1.5:4.5 loop    // r1 takes values 1.0, 2.5, 4.0, 5.5
    ...
  end for;

  for r2 in 2*x loop           // r2 takes values 2.2, 4.4, 6.6, 8.8
    ...
  end for;
```

The loop-variable is implicitly introduced in the scope inside the loop-construct and shall not be assigned to. If a variable with the same name is declared, the loop-variable “hides” the declared variable in the **for**-loop. For example:

```
// Modelica code that is correct, but is not recommended. Use
// different names for constant integer j and for loop variable j!
constant Integer j=4;
Real x[j];
equation
  for j in 1:j loop // The loop-variable j takes the values 1,2,3,4
    x[j]=j;         // Uses the loop-variable j, so x={1,2,3,4}
  end for;
```

This also means that if the loop variable is declared explicitly, then a translation error occurs, because an equation for the declared variable is missing. Example:

```
// Wrong Modelica code (an equation is missing for j)
parameter Integer nx;
Real x[nx];
Integer j; // No equation for j present
equation
  for j in 1:nx loop
    x[j]=j;
  end for;
```

To summarize: Loop variables are never declared!

For-loops in equation sections are not often used, since “loops” are traditionally written with assignment statements, see section 5.3. In equation sections, for loops are usually used to build up regular differential equation structures as shown for the filter above and for building up regular connection structures of components, see section 5.8.

### 5.3 Algorithm Sections

It is possible in Modelica to program in a procedural style, similar as in a programming language. This is performed with an **algorithm** section. Example:

```

block Limiter;
  Modelica.Blocks.Interfaces.RealInput  u;
  Modelica.Blocks.Interfaces.RealOutput y;
  parameter Real uMax= 1  "Maximum value";
  parameter Real uMin=-1  "Minimum value";
algorithm
  y := u;
  if u > uMax then
    y := uMax;
  elseif u < uMin then
    y := uMin;
  end if;
end Limiter;

```

Block “Limiter” provides the limited input signal *u* as output signal *y*. Contrary to an **equation** section, in an **algorithm** section only assignment statements are allowed. They have the form:

```
variable := expression;
```

The operator “:=” is used to define that the right hand side (here: “expression”) is assigned to the variable defined on the left hand side (here: “variable”). This is performed by first evaluating the expression on the right hand side and then assigning the result to the variable on the left hand side. All statements in an **algorithm** section are evaluated in the defined order.

In an **algorithm** section **if**-expressions, **if**-clauses and **for**-loops can be used in a similar way as in an **equation** section. The only difference is that the body of **if**-clauses and of **for**-loops must have assignment statements and no longer equations and that the restrictions from the **equation** sections do no longer hold. For example, an **if**-clause needs not to have an **else** clause and the number of statements in the different branches of an **if**-clause, need not to be the same. Furthermore, there might be several statements to compute the same variable (which is not possible in an **equation** section). This is demonstrated in the “Limiter” example above, where first an assignment to “*y*” is performed and then a second assignment to *y* is present, to reassign a new value if *u* is outside of the defined limits.

Another example with nested for-loops is shown in the next code fragment, to determine whether two arrays are identical (up to a given precision):

```

// Determine whether matrices A and B are equal
input Real A[:, :];
input Real B[size(A,1), size(A,2)]; // B has same sizes as A
parameter Real eps(min=0) = 0.0
  "Two numbers r1, r2 are identical if abs(r1-r2) <= eps";
output Boolean identical;
algorithm
  identical := true;
  for i in 1:size(A, 1) loop
    for j in 1:size(A, 2) loop
      if abs(A[i, j] - B[i, j]) > eps then
        identical := false;
        break;
      end if;
    end for;
  end for;

```

```

    if not identical then
        break;
    end if;
end for;

```

With the two nested **for**-loops, all elements of matrices A and B are tested pairwise whether they are identical. If one pair of elements is found that are not identical, the flag `identical` is set to **false** and the **for**-loops are at once terminated, without checking all the remaining elements. The termination is performed with the language keyword “**break**” which terminates the inner most **for** or **while** loop. The “**break**” keyword is only available in an **algorithm** section. It *cannot* be used in an **equation** section.

In an **algorithm** section, it is also possible to use a “**while**” loop (it cannot be used in an **equation** section). The example above can be implemented with **while** loops in the following way:

```

// Determine whether matrices A and B are equal
input Real A[:, :];
input Real B[size(A,1), size(A,2)]; // B has same sizes as A
parameter Real eps(min=0)=0.0
    "Two numbers r1, r2 are identical if abs(r1-r2) <= eps";
output Boolean identical;
Integer i, j;
algorithm
    identical := true;
    i := 1;
    while i <= size(A,1) loop
        j := 1;
        while j <= size(A,2) loop
            if abs(A[i, j] - B[i, j]) > eps then
                identical := false;
                i := size(A,1);
                j := size(A,2);
            end if;
            j := j + 1;
        end while;
        i := i + 1;
    end while;

```

The **while**-clause corresponds to while-statements in programming languages, and is formally defined as follows:

1. The expression of a **while**-clause shall be a scalar boolean expression.
2. When entering a **while**-clause, the expression of the **while**-clause is evaluated.
3. If the expression of the **while**-clause is **false**, the execution continues after the **while**-clause.
4. If the expression of the **while**-clause is **true**, the entire body of the **while**-clause is executed (except if a **break** statement or **return** statement is executed), and then execution proceeds at step 2.

As with all Modelica language elements, also an **algorithm** section is mapped to a set of equations, in order that the symbolic transformation algorithms, see Chapter 12, can be applied. This is performed by treating an **algorithm** section as a set of N equations, where every variable in the **algorithm** section is part of the equation system and all variables appearing at the left hand side of an assignment statement are treated as variables at the left hand side of the equation. All variables that appear at the left hand side of an assignment statement are initialized to their start values (for non-discrete variables) or to their **pre(.)** values (for discrete variables, see Chapter 7), whenever the **algorithm** is invoked. Due to this feature it is impossible for an **algorithm** section (or a **function**) to have a memory. Furthermore, it is guaranteed that the left hand side variables always have well-defined values.

For example, the algorithm section of block “Limiter” above has only one variable, *y*, appearing at the left hand side of the assignment statements. Therefore, the algorithm section of the Limiter block is treated as one equation of the following form for the symbolic processing:

```
y = f(u, uMax, uMin, y.start);
```

Practically, this means that this equation is “sorted” together with all other equations. After the evaluation order of the equations has been determined, the original **algorithm** section can be re-introduced at the appropriate place, together with the initialization of the left-hand side variables when invoking the **algorithm** section.

As another example, the **algorithm** section above with the two **while**-clauses has three variables (**identical**, **i**, **j**) that appear on the left hand side of the assignment statements. Therefore, this **algorithm** section is treated as a set of three coupled equations of the form (for simplicity the dependence on the **pre(..)** values of the output variables is not shown):

```
(identical, i, j) = f(A,B);
```

where the “**(..)**” part informally means the set of variables that is identical to the vector expression on the right hand side. During sorting (see Chapter 12) this set of equations is treated as three independent equations of the form:

```
0 = f1(identical, i, j, A, B);
0 = f2(identical, i, j, A, B);
0 = f3(identical, i, j, A, B);
```

This means, that these three equations will appear as part of an algebraic loop and therefore remain “together” during sorting.

If the evaluation of an algorithm section depends on the “**start**” value or on the “**pre(..)**” value of a variable, this usually indicates that the user has made a mistake, or at least, that it is possible to formulate this algorithm section in a “cleaner” and easier to understand way. It would be helpful, if a Modelica tool would indicate this in an information message, if this situation can be detected during translation (currently, neither Dymola, nor other tools provide diagnostics here). As an example, consider the following model:

```
Real x(start=1), y(start=2);
algorithm
  x := 2 + y;
  y := 2 + x;
```

During translation, such a code fragment is transformed to the following evaluation sequence:

```
input      : x.start, y.start
output     : x, y
algorithm: x := x.start
           y := y.start
           x := 2 + y;
           y := 2 + x;
```

With the given start values (**x.start**=1, **y.start**=2), the result of this algorithm section is therefore **x** = 4, **y** = 6. Since the result depends on **y.start**, the user probably made a mistake and a rewriting of the algorithm section would be useful.

A model may have any number of **equation** and **algorithm** sections, e.g.,

```
model Test
...
equation // equation section 1
...
algorithm // algorithm section 1
...
algorithm // algorithm section 2
...
equation // equation section 2
...
algorithm // algorithm section 3
...
end Test;
```

The semantics is, that every **algorithm** section is mapped to a set of equations and all equations are sorted together. Practically, this means that all statements within an **algorithm** section always remain together. However, different **algorithm** sections may be sorted relative to each other. This means, that in the “Test” example above, the **algorithm** sections might have a different evaluation order with respect to each other after sorting, e.g., **algorithm** section 3 is evaluated, say, before **algorithm** section 2, which is evaluated before **algorithm** section 1.

The question arises, when **algorithm** sections should be utilized instead of **equation** sections. The simple answer is: *Only use algorithm sections, if absolutely necessary*. The reason is that the symbolic pre-processing is considerably reduced in an **algorithm** section (a Modelica translator can only symbolically process an **algorithm** section, if the result is identical, as if the **algorithm** section would have been evaluated without any symbolic pre-processing. Since this is difficult, a Modelica translator might even decide to not perform any symbolic pre-processing in an **algorithm** section; this is currently the case for Dymola). As a consequence, the code is usually more reliable and more efficient if **equation** sections are used.

Usually, **algorithm** sections are utilized if a **while**-loop is needed, if complex, nested **if**-clauses become too cumbersome to define them with equations, or if a discrete algorithm is implemented (a controller) where the evaluation order is defined by the implementor. Finally, **algorithm** sections are always used in Modelica functions, see next section.

## 5.4 Functions

### 5.4.1 Basic properties of functions

Modelica supports functions that are similar to functions in programming languages like C. For example, the Limiter block from the previous section could also be implemented as the following function:

```
function Limiter
  input Real u;
  input Real uMax = 2;
  input Real uMin = -2;
  output Real y;
algorithm
  if u > uMax then
    y := uMax;
  elseif u < uMin then
    y := uMin;
  else
    y := u;
  end if;
end Limiter;
```

A function is defined in a similar way as a block, i.e., it has **public** and/or **protected** sections to declare all used variables. All variables in the **public** sections, must have the prefixes **input**, **output**, **parameter** or **constant**. A function has exactly one **algorithm** section. In this **algorithm** section nearly all operations are allowed that are also allowed in an **algorithm** section of a **model** or **block**, e.g., it is possible to use array operations, **if**-clauses, **for** and **while** loops. It is, however, not allowed to use the **der**(..) operator and operators associated with event handling (such as the **pre**(..) operator).

It is not allowed to modify the input arguments inside a function. In the Limiter example above this means, that variables “u, uMax, uMin” are not allowed to be used as left hand side variables of an assignment operator. As a result of this restriction, a Modelica tool is free to decide whether the actual calling mechanism of a function is performed with “call by value” (i.e., actual values are copied) or “call by reference” (i.e., the addresses of the actual values are copied). For efficiency reasons, it is important that arrays are passed with “call by reference” (which is the case for Dymola), in order that only the addresses of arrays are copied and not all array elements.

A function is executed by calling the function with actual values for the input arguments. If a function has only one variable declared with the **output** prefix, the function can be called at all places where an expression is allowed. The actual values for the input arguments can be either given in the order of the declarations (= “positional” arguments) or the actual values can be provided with binding equations (= “named” ar-

guments). Therefore, all of the following function calls are equivalent:

```
y1 = Limiter(u, 2, -2);
y2 = Limiter(u, uMax=2, uMin=-2);
y3 = Limiter(u, uMin=-2, uMax=2);
y4 = Limiter(uMin=-2, uMax=2, u=u);
y5 = Limiter(u); // uMax=2, uMin=-2
```

Actual values of input arguments need not to be provided, as demonstrated with “`y5 = Limiter(u)`”, if the input arguments have default values, as for `uMax` and `uMin` above.

If a function has several output arguments, the actual output arguments must be provided in parenthesis in the same order as the corresponding variables are declared in the function. Here, only positional arguments are allowed. Example: A function “`f`” with 2 input and 3 output arguments is called as:

```
(o1, o2, o3) = f(i1, i2);
```

If one or more of the output arguments are not needed when calling the function, they need not to be provided. Therefore, the function “`f`” could also be called in the following forms:

```
(o1,o2) = f(i1,i2);
(,o2,o3) = f(i1,i2);
o1 = f(i1,i2);
```

It is not possible to inquire in a function which of the input and/or output arguments are not provided from the outside. Therefore, always all output arguments must be computed inside the function assuming that all input arguments are provided. If it is “computationally expensive” to calculate one or more of the output arguments, a function could have an additional option argument that allows the caller of a function to decide whether output arguments shall actually be computed in the function, or whether only a default value shall be returned. For example, the function “`f`” above, could be called as:

```
(o1,o2) = f(i1,i2, computeOutput3 = false);
```

and the function could be implemented as:

```
function f
  input Real i1, i2;
  input Boolean computeOutput3=true;
  output Real o1, o2, o3=0;
algorithm
  o1 := ..;
  o2 := ..;
  if computeOutput3 then
    o3 := ...;
  end if
end f;
```

As a demonstration for a function implementation, in the following example the function returns the product of two polynomials. A polynomial is described by a vector of its coefficients (e.g. polynomial “ $p = 2x^2 + 3x + 4$ ” is defined by its coefficient vector “`a = {2, 3, 4}`”):

```
function polynomialMultiply
  input Real a[:], b[:]; // vectors of polynomial coefficients
  output Real c[size(a,1) + size(b, 1) - 1];
protected
  Integer na = size(a,1);
  Integer nb = size(b,1);
  Integer nc = n1+n2-1;
algorithm
  for i in 1:nc loop
    c[i] := 0.0;
    for j in max(1,i+1-nb) : min(i,na) loop
      c[i] := c[i] + a[j]*b[i+1-j];
    end for;
```



```

    end for;
end polynomialMultiply;

```

The dimensions of vectors *a* and *b* are arbitrary and are defined when calling the function. Output vector *c* is the coefficient vector of the product of the polynomials represented by the coefficients vectors *a* and *b*. The dimension of this vector must be defined before terminating the function. In the example above, the actual dimension of *c* is directly given when *c* is declared (as function of the dimensions of *a* and *b*). This is the usual way to dimension output arrays. Afterwards, in the algorithm section the output array is calculated.

Functions are terminated when the last statement of the respective function is reached. Alternatively, a function can be terminated after every desired statement by using the language keyword “**return**”. This is demonstrated in the following example (yet another variant to test whether two matrices *A* and *B* are identical):

```

function isEqual "Determine whether matrices A and B are equal"
  input Real A[:, :];
  input Real B[size(A,1), size(A,2)]; // B has same sizes as A
  input Real eps(min=0) = 0.0
    "Two numbers r1, r2 are identical if abs(r1-r2) <= eps";
  output Boolean identical;
algorithm
  identical := true;
  for i in 1:size(A, 1) loop
    for j in 1:size(A, 2) loop
      if abs(A[i, j] - B[i, j]) > eps then
        identical := false;
        return;
      end if;
    end for;
  end for;
end isEqual;

```

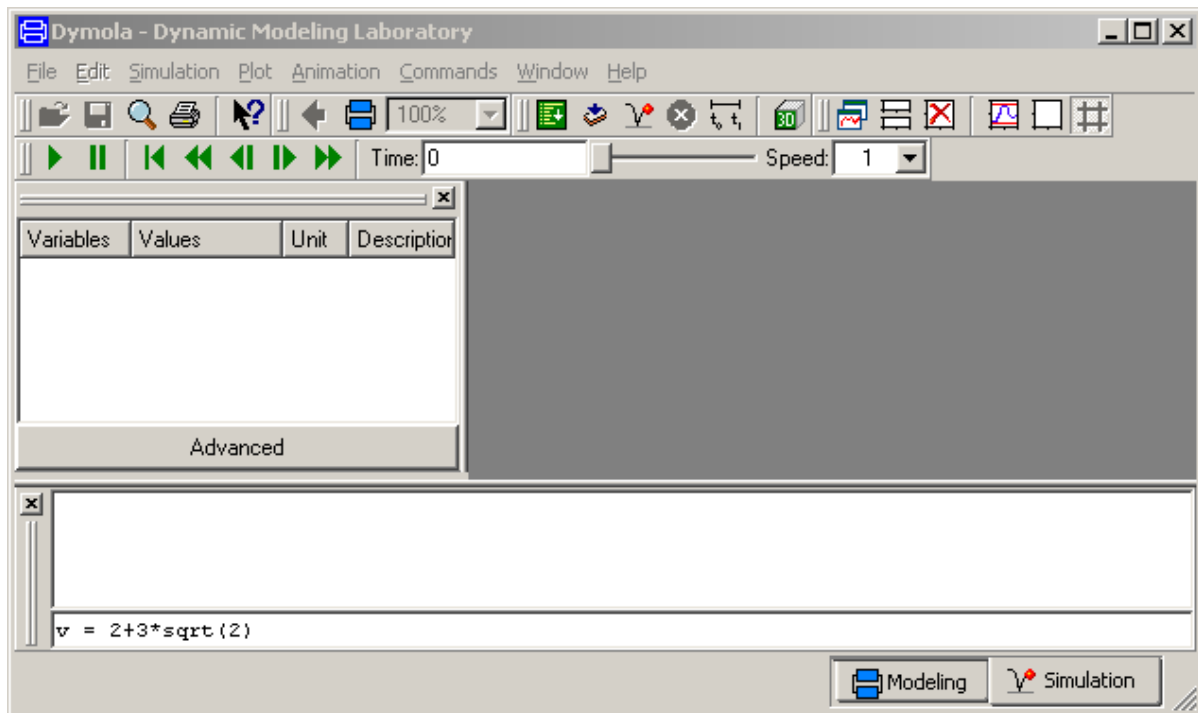
Here the function is at once terminated once an element of matrix *A* is not identical to the corresponding element of matrix *B*. This also means that the `for`-loops are at once terminated.

### 5.4.2 Calling functions interactively

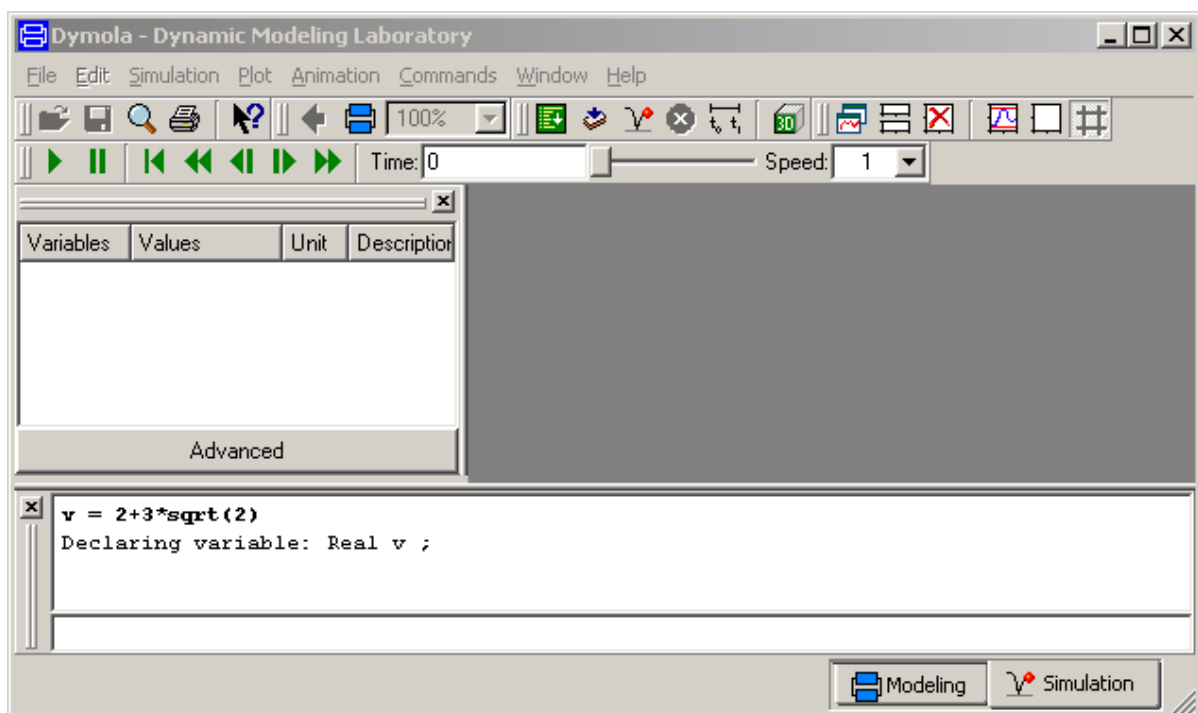
Modelica functions can be called in a model or a block (and of course in another function), so they are executed during simulation. Another possibility is to call them interactively. In fact, the command window in the simulation tab of Dymola is an *interpreter* for the algorithmic part of the Modelica language. To be precise: all Modelica language elements can be used that are also allowed in an **algorithm** section of a function. In order to ease an interactive use, there are a few simplifications:

- Variables in the interpreter are not declared. Instead, they are automatically introduced as the result of an operation (this also means, that no declaration is allowed in the interpreter; e.g., “`Real x;`” is not allowed).
- The interpreter supports only assignment statements, since it is always an **algorithm** section. To ease the usage, the “`=`” operator can be used, but is interpreted as “`:=`”.

As a very simple example, an arithmetic operation shall just be carried out. Typing the assignment statement “`v = 2+3*sqrt(2)`” in to the command window results in the situation displayed in Figure 5.4:

*Figure 5.4: Defining an arithmetic expression in Dymolas command window.*

When pressing the return key, the expression on the right hand side is evaluated, assigned to the variable at the left hand side of the “=” sign, and the operation and its result are logged in the window above the command window, see Figure 5.5:

*Figure 5.5: Result of executing a statement in Dymolas command window.*

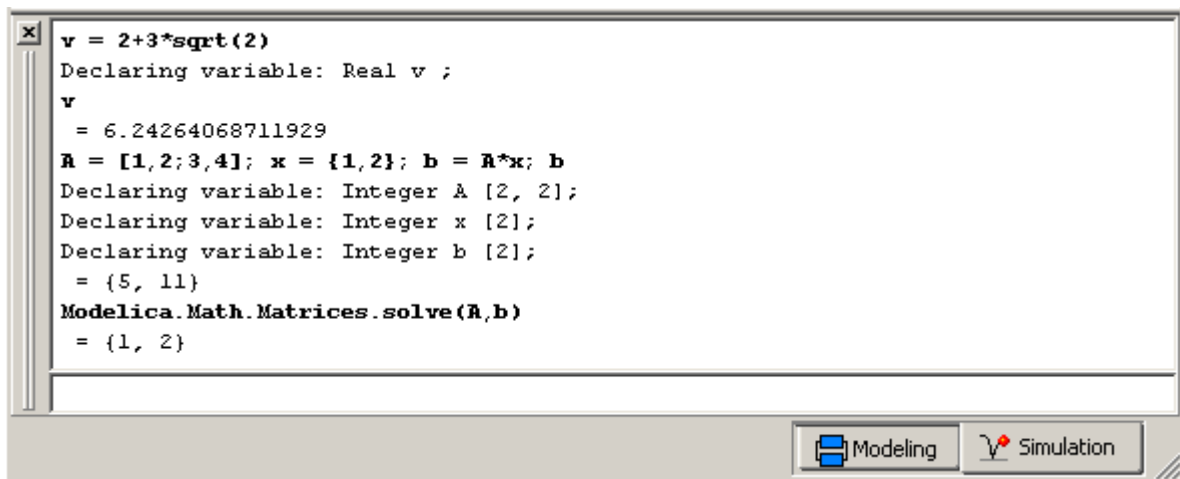
When typing the variable name and pressing “return”, the value of the variable is displayed, as shown in Figure 5.6

Figure 5.6: The value of a variable is displayed in the command window, when typing its name.



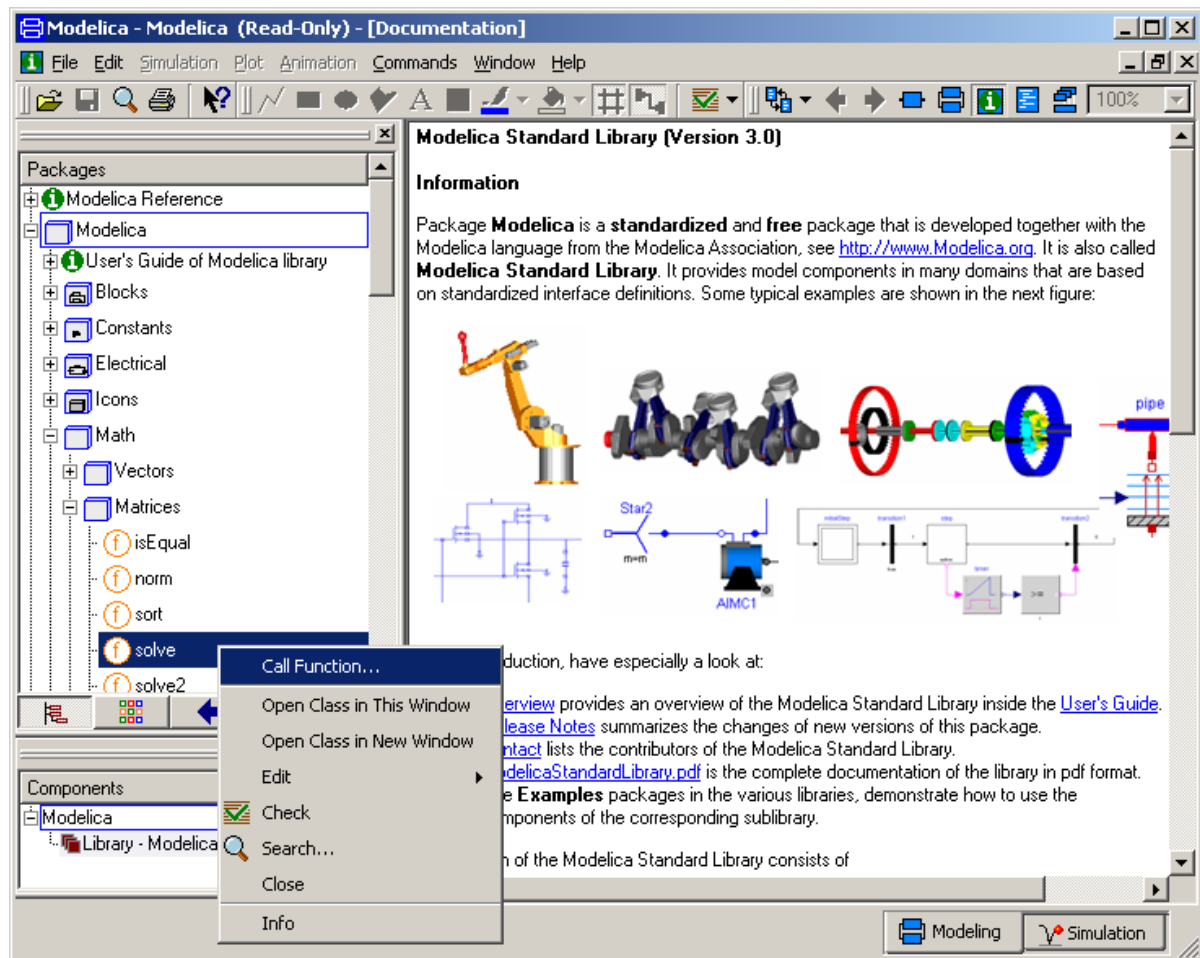
It is also possible to call any Modelica function in the command window. For example, a linear system  $\mathbf{Ax}=\mathbf{b}$  shall be solved with function `Modelica.Math.Matrices.solve`. In order to verify the result, vector  $\mathbf{b}$  is first constructed by a given  $\mathbf{A}$  matrix and  $\mathbf{x}$  vector and then the function is called with matrix  $\mathbf{A}$  and vector  $\mathbf{b}$  to compute  $\mathbf{x}$ . The result of typing these commands in the command window are shown in Figure 5.7.

Figure 5.7: Solving a linear system of equations in the command window.



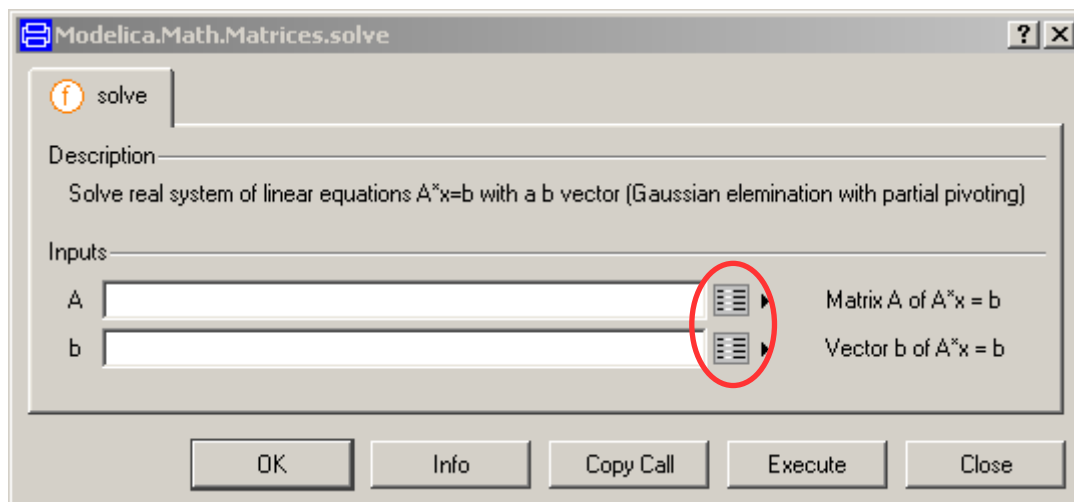
An alternative, more convenient way, is to select the corresponding function in the package browser, right click with the mouse and select "Call Function ..." in the context sensitive menu, as shown in Figure 5.8:

Figure 5.8: Calling a Modelica function in the package browser by right clicking on the function.



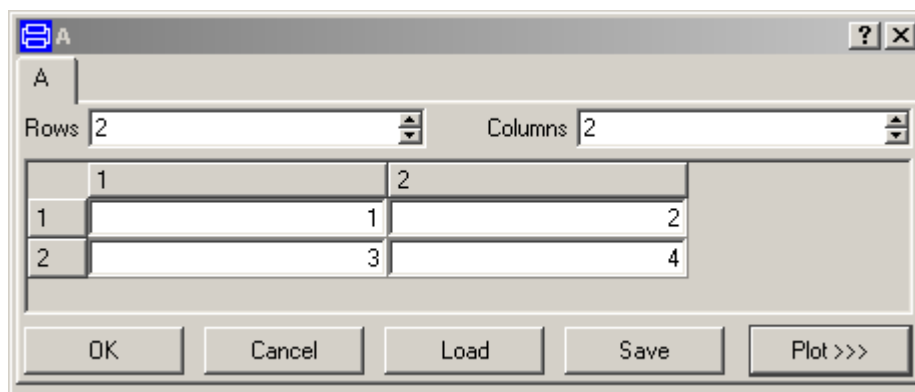
Dymola opens then a menu in which all input arguments of the function together with the description texts are shown. The actual arguments to the function can be provided in the input fields, e.g.,  $A = [1, 2; 3, 4]$ , see Figure 5.9:

Figure 5.9: Automatically constructed menu when calling function “Modelica.Math.Matrices.solve”



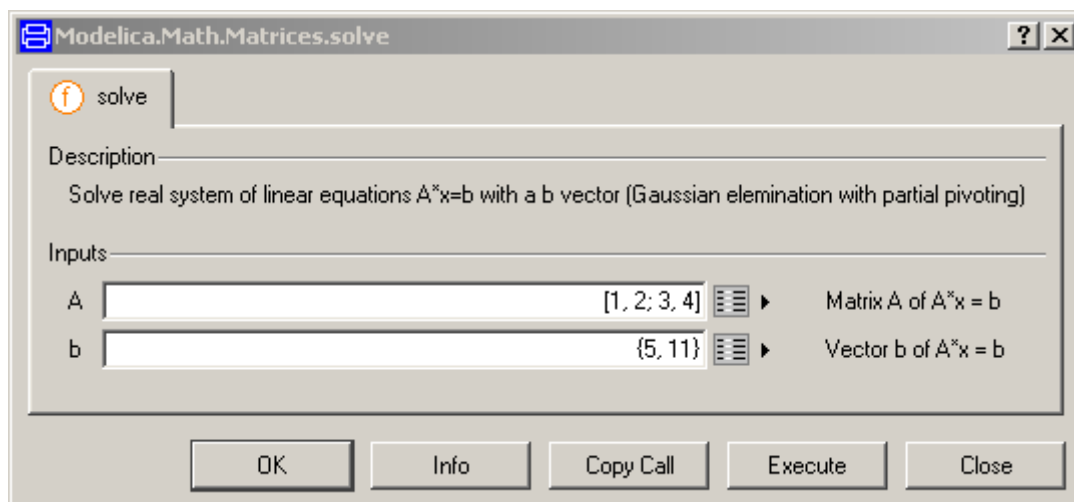
It is more convenient to use instead the vector/matrix editor, by clicking on the “table” symbol to the right of the input field (marked with a red ellipse in Figure 5.9). This opens the menu in Figure 5.10:

Figure 5.10: Array editor of input field of a function or parameter menu in Dymola.



First the number of rows and number of columns are defined and then all elements of a vector or matrix can be easily provided. After clicking “OK”, the vector or matrix is copied in Modelica notation in to the corresponding input field. The result is shown in Figure 5.11, after both matrix “A” and vector “b” are defined in this way:

Figure 5.11: Resulting menu after defining the input arguments in the array editor.



After clicking “OK”, the function call is copied in to the command window of Dymola and is executed. Remember, that the construction of the menu for the input arguments of a function call is performed automatically by Dymola, based on the declarations of the input variables of the function. Therefore, also *user defined functions* can be *conveniently called* in this way.

### 5.4.3 Modelica built-in functions and functions from the Modelica Standard Library

Modelica has a large set of built-in functions and of built-in operators with function syntax. A summary of these functions is provided in the appendix on page 227, and in package `ModelicaReference.Operators`. Built-in functions and operators are provided, if they cannot be expressed as standard Modelica functions, e.g., because the number of input arguments is varying, and/or the input/output arguments are of varying types, and/or the operator communicates with the simulation engine. For example, the built-in function “**String**(...)” transforms variables to a string representation. The first input argument can be of type Real, Integer or Boolean. With a standard Modelica function, this cannot be defined and therefore **String**(...) has to be a built-in function. Although built-in functions cannot be defined with Modelica, the calling mechanism of such functions is identical to calling Modelica functions. There are some built-in functions (such as “**sin**(...)” or “**cross**(...)”) that could be defined as Modelica function, but are provided as built-in function mostly for historical reasons.

Additionally, about 550 Modelica functions are provided in the Modelica Standard Library. A short overview is given in the following table:

Table 5.1: Functions in the Modelica Standard Library

Modelica	
Math	Mathematical functions (e.g., “sin”, “cos”)
Vectors	Functions operating on vectors (e.g. “isEqual”, “length”, “sort”)
Matrices	Functions operating on matrices (e.g., “norm”, “solve”, “eigenValues”)
Media	Functions computing media properties
Utilities	
Files	Functions to work with files and directories (e.g. “remove”)
Streams	Functions to read from file and write to file (e.g. “print”)
Strings	Functions operating on strings (e.g. “sort”, “scanReal”)
System	Functions for environment (e.g. “getWorkDirectory”)

Some of these functions will be discussed in more detail in the following chapters.

#### 5.4.4 Functions with dynamically sized arrays

Arrays inside functions can be dynamically sized as function of input arguments. For example, in function “**polynomialMultiply**(...)” on page 88, the dimension of output vector *c* is defined as function of the dimension of input vectors *a* and *b* respectively, using the **size**(...) operator:

```
input Real a[:], b[:];
output Real c[size(a,1) + size(b, 1) - 1];
```

This is the standard way to dimension arrays in functions.

It is also possible to dimension arrays in a more flexible way after the array has been declared, and/or to resize arrays dynamically. This is performed by assigning a *complete* array to the corresponding variable. As an example, the following function has a vector of complex numbers as input argument *c* (represented as a two-column matrix, where the first column are the real and the second column are the imaginary parts of the complex numbers) and returns the real values *r* that do not have a complex part (therefore the dimension of “*r*” depends on the element values of *c*, but *not* on the dimension of *c*):

```
function realNumbers1 "Return real numbers of a complex vector"
  input Real c[:,2] "Complex numbers; col. 1: real, col. 2: imag.";
  output Real r[:] "The real numbers of c";
protected
  Integer nr=0 "Dimension of r";
  Integer j =1;
algorithm
  // Determine the size of vector r
  for i in 1:size(c,1) loop
    if c[i,2] == 0.0 then
      nr := nr + 1;
    end if;
  end for;

  // Generate vector "r" and copy the numbers
  r := zeros(nr);
  for i in 1:size(c,1) loop
    if c[i,2] == 0.0 then
      r[j] := c[i,1];
      j := j + 1;
    end if;
  end for;
end realNumbers1;
```

Note, before assigning an element to array “*r*”, this array needs to be defined by assigning an array to it (here: by assigning a vector of zeros with “*r* := **zeros**(nr)”). It is an error if a value is assigned to an element of an array, if this element does not yet exist.

Alternatively, a function can be provided to determine the dimension of a vector and this function is

used in the declaration of this vector. With this strategy, the function above could be implemented in the following way:

```

function numberOfReals "Return the number of real numbers"
  input Real c[:,2] "Complex numbers; col. 1: real, col. 2: imag.";
  output Integer nr=0 "Number of Reals in c";
algorithm
  for i in 1:size(c,1) loop
    if c[i,2] == 0.0 then
      nr := nr + 1;
    end if;
  end for;
end numberOfReals;

function realNumbers2 "Return real numbers of a complex vector"
  input Real c[:,2] "Complex numbers; col. 1: real, col. 2: imag.";
  output Real r[numberOfReals(c)] "The real numbers of c";
protected
  Integer j=1;
algorithm
  for i in 1:size(c,1) loop
    if c[i,2] == 0.0 then
      r[j] := c[i,1];
      j := j + 1;
    end if;
  end for;
end realNumbers2;

```

The difference to the previous implementation is that the dimension of vector “r” is computed with the function call “**numberOfReals**(c)” and the returned result is directly used to declare the dimension of “r”.

Functions **realNumbers1**(...) and **realNumbers2**(...) can be called, directly or indirectly, in the interactive environment of Dymola without any problems.

Calling these functions *directly* in a *model* is *not* possible, because the dimensions of any array in a model must be known during translation (in order that symbolic processing is possible) and the two functions return arrays where the dimensions are only known after the actual values of the input vector “c” is known.

Calling these functions *indirectly* in a *model* should be possible as long as all array dimensions in the model can be determined during translation. For example, if the following function is called in a model, everything should be fine, since array dimensions in the model are not influenced by this function call:

```

function testRealNumbers1
  input Real A[:,size(A,1)] = [1,2,0;4,5,6;7,8,9];
protected
  Real eigenValues[size(A,1),2];
  Real realEigenValues[:];
algorithm
  eigenValues := Modelica.Math.Matrices.eigenValues(A);
  realEigenValues := realNumbers2(eigenValues);
  for i in 1:size(rEval,1) loop
    Modelica.Utilities.Streams.print(
      "realEigenValues[" + String(i) + "] = " +
      String(realEigenValues[i]));
  end for;
end testRealNumbers1;

```

However, in Dymola 7.1 this is not (yet) supported and Dymola reports a warning during translation:

*Variable “realEigenValues” was declared with dimension “:”. That is not yet supported in dsmodel.c, and the function will fail if called in the model.*

Dymola fails when simulating this model as announced in this warning message.

It is possible to rewrite the function so that translation and simulation is successful in Dymola by calcu-

lating the dimension of “realEigenValues” to get rid of the “:” dimension declaration:

```

function testRealNumbers2
  input Real A[:,size(A,1)] = [1,2,0;4,5,6;7,8,9];
protected
  Real eigenValues[size(A,1),2] =
    Modelica.Math.Matrices.eigenValues(A);
  Real realEigenValues[numberOfReals(eigenValues)];
algorithm
  realEigenValues := realNumbers2(eigenValues);
  ...
end testRealNumbers2;

```

The drawback is the inconvenient rewriting of the code and that function “**numberOfReals**(...)” is called twice (in **testRealNumbers2**(...) and in **realNumbers2**(...)) and not once as in the first variant.

### 5.4.5 Functions with internal memory

Modelica functions are *mathematical functions*, i.e., functions that return always the same result values, when called with the same input arguments. In other words, Modelica functions do *not* have a “memory”. This is enforced by the semantics of an algorithm section, see section 5.3: All left hand side variables of the assignment operator “:=” are initialized with their start values before the algorithm section is executed (if no start value is defined, the default start value of 0.0 for Real, 0 for Integer, **false** for Boolean and an empty string for String types are used). It is therefore not possible to access the value of a variable from the last function call. If an internal memory is really needed, a Modelica **block** has to be used instead (but then a function call syntax is not available and a Modelica **block** cannot be used inside another function). Alternatively, an external C-function in form of an **ExternalObject** can be used, see section 5.6.

There are the following reasons, why functions with internal memory are not allowed in Modelica:

1. Functions are potentially called, directly or indirectly, in an equation section which is mapped to the right hand side  $\mathbf{f}(\cdot)$  of a differential equation system  $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, t)$ : Numerical integration algorithms might fail if  $\mathbf{f}(\cdot)$  is computed via a “hidden”, internal memory that accesses values of variables from previous integration steps. As usual, Modelica uses the “safe” approach and guarantees that such a potential misuse is not possible with Modelica functions and therefore has this restriction (with external C-functions, such type of misuse cannot be prevented).
2. The symbolic transformation algorithms, see Chapter 12, cannot be applied on functions that have an internal memory, since “equation sorting” requires that all variables that are relevant for the sorting process are taken into account and an internal memory of a function would consist of “hidden” variables.
3. Modelica tools, especially Dymola, enhance efficiency by “common subexpression elimination”. For example, the following code fragment

```

y = { sin(phi)*x[1] + cos(phi)*x[2],
      -cos(phi)*x[1] + sin(phi)*x[2] };

```

is translated by Dymola into the following form during code generation:

```

w1 := sin(phi);
w2 := cos(phi);
y := { w1*x[1] + w2*x[2],
      -w2*x[1] + w1*x[2] };

```

This means that the function calls of **sin**(...) and **cos**(...) are evaluated only once and not twice. In general, function calls that have identical input arguments are only evaluated once and the result values of the first call are used at all other places where this function is called with the same input arguments. Such type of code simplification is not possible, if functions with internal memory would be present.

Note, common subexpression elimination makes external C-code with internal memory unreliable, because a tool might just replace several identical calls to the same C-function by just one call and then the expected behavior is gone. The *only* reliable way to use external C-code with internal memory in Modelica are therefore **ExternalObjects**, see section 5.6.



Although Modelica functions cannot have internal memory, it is possible to emulate them by having the internal memory as input argument and the modified internal memory as output argument. This is demonstrated with the next example, a pseudo random number generator. In a programming language like “C”, a random number generator is typically utilized in the following way:

```
// Wrong Modelica code to use a random number generator
randomInit(seed={23,87,187});
...
x = random(); // x = 0.67206
...
y = random(); // y = 0.36786
```

The internal memory of this random number generator consists of an Integer vector with 3 elements. With “**randomInit**(...)” this internal memory is initialized. Every call of “**random**()” produces now another number and updates the internal memory in order to get this behavior. As previously explained, it is not possible to implement such a behavior with a Modelica function. It would be possible to implement this behavior with external C-functions. However, Dymola would transform the above code fragment into the following form:

```
// Common subexpression elimination of the above code results in
randomInit(seed={23,87,187});
...
x = random(); // x = 0.67206
...
y = x; // y = 0.67206
```

Therefore, every call to “**random**()” would produce exactly the same output. The reason is that Modelica functions generate the same output arguments if identical input arguments are provided. It is therefore fine that a tool replaces all occurrences of a function call with identical input arguments by just one function call.

In order to achieve the expected behavior, the random-number generator must be implemented in the following way, where the “internal memory” is both an input and an output argument of the Modelica function:

```
function random "Pseudo random number generator"
  input Integer seedIn[3] "Seed from last call";
  output Real x "Random number between 0 and 1";
  output Integer seedOut[3] "Modified seed for next call";
algorithm
  seedOut[1] := rem((171*seedIn[1]), 30269);
  seedOut[2] := rem((172*seedIn[2]), 30307);
  seedOut[3] := rem((170*seedIn[3]), 30323);

  // Zero is a poor seed, therefore substitute 1;
  for i in 1:3 loop
    if seedOut[i] == 0 then
      seedOut[i] := 1;
    end if;
  end for;

  x := rem((seedOut[1]/30269.0 +
           seedOut[2]/30307.0 +
           seedOut[3]/30323.0), 1.0);
end random;
```

This function is a variant of the Wichmann-Hill random number generator (Wichmann and Hill 1983). It produces a uniformly distributed output argument “x” in the range from 0.0 to 1.0. The period (how many numbers it generates before repeating the sequence exactly) of this generator is 6,953,607,871,644. The built-in function “**rem**(...)” is the Integer remainder of the division of two Real numbers, see Table 21.7 on page 230.

The function gets the “internal memory” (seedIn) from the last call as input argument, changes it and returns it as output argument (seedOut). In a function, this random number generator might be utilized in

the following way:

```

function testRandom1
  input Integer seed_start[3] = {23,87,187};
protected
  Integer seed[3] = seed_start;
  Real x;
algorithm
  for i in 1:10 loop
    (x, seed) := random(seed);
    Modelica.Utilities.Streams.print("x = " +
                                     String(x,significantDigits=16));
  end for;
end testRandom1;

```

In the local variable “seed”, the “internal memory” of the random number generator is stored. The **random**(..) function is called in a **for** loop. For convenience, the same variable is used for the input and output seed. In an algorithm section this can be performed, because the right hand side expression is evaluated and assigned to the left hand side variable. In such a situation, the same variable can be used on the left and right hand side. Function “**print**” from the Modelica Standard Library prints a string to the command window. Here, the string consists of the actual value of “x”. This function produces the following output in the command window of Dymola when called interactively:

```

x = 0.6720613541740214
x = 0.3678597646977231
x = 0.6041085613242592
x = 0.1667374195729034
x = 0.4629589578171096
x = 0.9289893342375364
x = 0.6531921316126386
x = 0.5926293930235833
x = 0.5294703487946859
x = 0.7684225743864563

```

In a model it is a bit more tricky to call this function and there are different variants. A random number generator is usually used to generate “noise” for the measurement signals of a plant model. A typical usage is therefore a sampled data system, where the random function is called at every sample time:

```

model testRandom2
  parameter Integer seed_start[3] = {23,87,187};
  Integer seed[3](start=seed_start, each fixed=true);
  Real x;
equation
  when sample(0,0.2) then
    (x, seed) = random(pre(seed));
    Modelica.Utilities.Streams.print(
      "time = " + String(time, format=".1f") +
      ", x = " + String(x, format=".6f"));
  end when;
end testRandom2;

```

The details of a **when**-clause and of the built-in operators **sample**(..) and **pre**(..) are discussed in Chapter 7. In short, the body of the **when**-clause in testRandom2 is executed every 0.2 seconds once. As input seed, the value of the seed vector from the previous sample instant is used. If noise for several measurement signals is needed, different “seed” vectors with different start values have to be defined, since otherwise the noise of different measurement signals would be correlated to each other (= would produce the same values). The above test model produces the following output to Dymola's message window when simulated for 1 second:

```

time = 0.0, x = 0.672061
time = 0.2, x = 0.367860

```

```

time = 0.4, x = 0.604109
time = 0.6, x = 0.166737
time = 0.8, x = 0.462959
time = 1.0, x = 0.928989

```

As demonstrated with the example above, functions with memory can be emulated in Modelica. Besides the slight inconvenient usage (since the “internal memory” has to be passed as input and as output argument to a function call), the major drawbacks are that the “internal memory” can be accessed in the calling environment and that the whole “internal memory” is copied for every function call. As a result, this approach is limited to cases, where the “internal memory” has a limited size, since copying large data structures for every function call would be too inefficient.

## 5.5 Records

### 5.5.1 Basic properties of records

Modelica records are used to structure data by name, similarly to a “struct” in “C” or “Matlab”. A record consists of a set of declarations (without equation or algorithm sections). For example, every medium in the `Modelica.Media` library is defined with the `Modelica.Media.Interfaces.PartialMedium.FluidConstants` record to identify the medium:

```

record FluidConstants "Standard data of fluid"
  String iupacName          "Complete IUPAC name";
  String casRegistryNumber  "Chemical abstracts sequencing number";
  String chemicalFormula    "Chemical formula (according to Hill)";
  String structureFormula   "Chemical structure formula";
  Modelica.SIunits.MolarMass molarMass "Molar mass";
end FluidConstants;

```

Records can be derived by *inheritance* from one or more other records. They can be instantiated in a similar way as a “**model**” to provide actual values for their elements via a *modifier*. If an instantiated record is prefixed with “**parameter**” it appears in the parameter menu of the corresponding class. Examples:

```

record TwoPhaseFluidConstants "Standard data for two phase media"
  import SI = Modelica.SIunits;
  extends FluidConstants;
  SI.Temperature      criticalTemperature;
  SI.AbsolutePressure criticalPressure;
  SI.MolarVolume      criticalMolarVolume;
  SI.Temperature      triplePointTemperature;
end TwoPhaseFluidConstants;

```

```

package Water
  constant TwoPhaseFluidConstants fluidConstants(
    iupacName          = "oxidane",
    casRegistryNumber  = "7732-18-5",
    chemicalFormula    = "H2O",
    structureFormula   = "H2O",
    molarMass          = 0.018015268,
    criticalTemperature = 647.096,
    criticalPressure    = 22064.0e3,
    criticalMolarVolume = 1/322.0*0.018015268,
    triplePointTemperature = 273.16);

  model MediumBaseProperties
    Modelica.SIunits.SpecificHeatCapacity R "Gas constant";
    ...
  equation
    R = Modelica.Constants.R/fluidConstants.molarMass;
    ...

```

```

    end MediumBaseProperties;>
    ...
  end Water;

```

The record instance “waterConstants” above is a constant record where every element of the record is assigned a value. The elements of a record instance can be accessed by “dot” notation, e.g., fluidConstants.molarMass. The above construction shows how to build up structured data that is defined *once* and cannot be modified afterwards since defined as “**constant**”.

### 5.5.2 Record constructor function

Every record definition has an associated function called “*record constructor function*”. It has the same name and is in the same scope as the record definition. All components of the record are input arguments of the function (with exception of components defined as “**constant**” or as “**final parameter**”) and the single output argument is an instance of the record where all input arguments are assigned to the record elements. Example:

```

record Complex "Complex number"
  extends Modelica.Icons.Record;
  Real re "Real part";
  Real im "Imaginary part";
end Complex;

function add
  input Complex u1, u2;
  output Complex y;
algorithm
  y.re = u1.re + u2.re;
  y.im = u1.im + u2.im;
end add;

model TestComplex
  Complex c1(re=2, im=1) annotation(...);
  Complex c2 annotation(...);
  Complex c3 annotation(...);
equation
  c2 = add(c1, Complex(sin(time), cos(time)));
  c3 = add(c1, Complex(re=sin(time), im=cos(time)));
end TestComplex;

```

Above a record for complex numbers is defined. The record extends from “Modelica.Icons.Record” in order that a default icon is available for the record. Function “add” is used to add two complex numbers. The input and argument arguments of this function are records of type Complex. Finally, the test model “TestComplex” shows a simple usage of the complex number record: In the declaration section three complex numbers are declared. They are constructed here by dragging them from the package browser in to the diagram layer of the model. The advantage is that a user can click on a record instance in the diagram layer and the parameter menu of the record pops up to, e.g., define the actual values of complex number c1, see Table 5.2:

Table 5.2: Graphical view of a record instance in the diagram layer.

Diagram layer of TestComplex	Parameter menu of TestComplex.c1
<div><div><div>c1</div><div><div></div><div></div><div></div><div></div></div></div><div><div>c2</div><div><div></div><div></div><div></div><div></div></div></div><div><div>c3</div><div><div></div><div></div><div></div><div></div></div></div></div>	<div><div>Parameters</div><div><div>re</div><div><div></div><div>2</div></div><div>Real part</div></div><div><div>im</div><div><div></div><div>1</div></div><div>Imaginary part</div></div></div>

The complex numbers c2 and c3 are constructed by adding c1 with another complex number that is “on-the-fly” constructed with the record constructor function “Complex” of the Complex number. The two dif-

ferent calling forms are shown: either by positional or by named arguments. This demonstrates that a record constructor function can be used at all places where a function call is allowed to construct a new instance of a record.

Record constructor functions allow to build up data libraries where the data entries can still be modified (contrary to the medium record example above). This is demonstrated on the basis of a small library of electrical motors:

```

package Motors
  record MotorData "Data sheet of a motor"
    parameter Real inertia;
    parameter Real nominalTorque;
    parameter Real maxTorque;
    parameter Real maxSpeed;
  end MotorData;

  model Motor "Motor model" // using the generic MotorData
    parameter MotorData data;
    ...
  equation
    ...
  end Motor;

  record MotorType1 = MotorData(
    inertia      = 0.001,
    nominalTorque = 10,
    maxTorque     = 20,
    maxSpeed      = 3600) "Data sheet of motor type 1";

  record MotorType2 = MotorData(
    inertia      = 0.0015,
    nominalTorque = 15,
    maxTorque     = 22,
    maxSpeed      = 3600) "Data sheet of motor type 2";
end Motors

model Robot
  import Motors.*;
  Motor motor1(data = MotorType1()); // just refer to data sheet
  Motor motor2(data = MotorType1(inertia=0.0012));
                                     // data can still be modified
  Motor motor3(data = MotorType2());
  ...
end Robot;

```

Library “Motors” has record “MotorData” defining the most important data for an electrical motor. Model “Motors.Motor” is a physical model of the motor using the `motorData` record. The `motorData` record declaration has the prefix “**parameter**”, in order that the record appears in the parameter menu of a `Motor` instance. New motor data records are defined by inheritance from the “MotorData” record and providing concrete values for the parameters. Inheritance is used here via the “*short class definition*”. E.g. `MotorType1` is completely equivalent to the following definition:

```

record MotorType1
  extends MotorData(inertia=0.0015, ...);
end MotorType1;

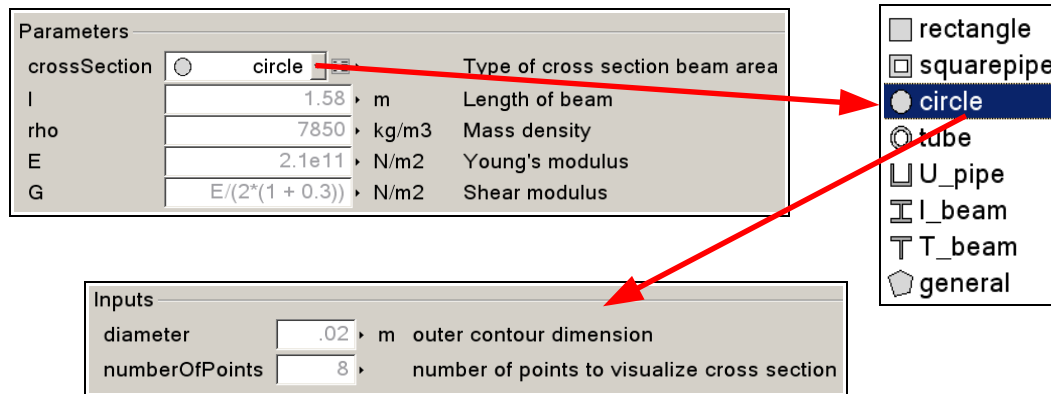
```

The defined `Motors` library is utilized to build up a `Robot` model. In the `Robot` model several motors are instantiated and the motor data is given as modifier. Note, it is not possible to directly provide “`MotorType1`” as modifier, because “`MotorType1`” is a **class** and not an instance and only instances can be provided as modifiers. Instead the record constructor function has to be used to generate on-the-fly an instance of `MotorType1`. If no arguments are given, the defaults from the `Motors` library are used. However,

it is also possible to overwrite the defaults of the `Motors` library by providing actual values to the input arguments (like changing `inertia` from 0.001 to 0.0012 for `motor2` above).

Functions that return a record can nicely be utilized to provide different pre-defined variants for an input field of a parameter menu. For example, in the commercial `FlexibleBodies` library, flexible beams are defined via a hierarchical data record. The geometry of a beam is defined by its cross section area and its length. In Figure 5.12 part of the beam data record menu is shown.

Figure 5.12: Menu to define the cross section area of a flexible beam in the `FlexibleBodies` library.



Parameter “`crossSection`” consists of a pop-down menu in which different cross section shapes can be selected such as “rectangle”, “circle”, “I\_beam” etc. When one particular shape is selected then a menu pops up to define exactly the data for the selected shape, e.g., “diameter” for a “circle” shape, or “width” and “height” for a “rectangle” shape. The above menus are defined in the following way:

```

record BeamCrossSection
  Modelica.SIunits.Area area "Cross sectional area";
  Real polyLine[:,2]      "Visual appearance (for animation)";
  Real Iyy(final unit="m4") "Geometrical moment of inertia Iyy";
  Real Izz(final unit="m4") "Geometrical moment of inertia Izz";
  Real It (final unit="m4") "Geometrical, torsional moment It";
end BeamCrossSection;

record BeamData
  parameter BeamCrossSection crossSection;
  ...
end BeamData;

```

Basically, the `BeamData` record consists of sub-record `crossSection` to define all characteristic numbers of the cross section area that are needed to describe the flexibility of the beam. To require this data directly from the user as stated above would be very inconvenient. For this reason, the actual implementation shown in Figure 5.12 adds an annotation to define a pop-down menu:

```

record BeamData
  parameter BeamCrossSection crossSection annotation(choices(
    choice = BeamChoices.rectangle(),
    choice = BeamChoices.squarepipe(),
    choice = BeamChoices.circle(), ...
  ));
  ...
end BeamData;

```

The “**choices**” annotation is a standard Modelica annotation. For every “**choice** = xxx” definition, an entry in the pop-down menu is defined. If an entry is selected interactively, the corresponding “**choice**” definition is used as modifier. For example, if the third entry is selected above interactively, then the input arguments of function “`BeamChoices.circle()`” are shown in a menu and after clicking “OK” a modifier is generated with the given inputs. For example, if a diameter of 0.04 is defined then the following code is gen-

erated:

```
Beam beam(data( crossSection = BeamChoices.circle(diameter=0.04) ));
```

The different functions to define the cross section are provided in a separate sub-library:

```
package BeamChoices
...
function circle
  input Modelica.SIunits.Length diameter = 0.02;
  input Integer numberOfPoints(min=4) = 8;
  output BeamCrossSection crossSection(
    redeclare Real polyLine[numberOfPoints,2]);
  import Modelica.Constants.pi;
protected
  Modelica.SIunits.Angle phi;
algorithm
  crossSection.area = pi*diameter^2/4;
  crossSection.Iyy = pi*diameter^4/64;
  crossSection.Izz = crossSection.Iyy;
  crossSection.It = 2*crossSection.Iyy;
  for i in 1:numberOfPoints loop
    phi = 2*pi*(i-1)/numberOfPoints;
    crossSection.polyLine[i,:] = diameter/2*{sin(phi), cos(phi)};
  end for;
end circle;
...
end BeamChoices;
```

Function “BeamChoices.circle” has `diameter` and `numberOfPoints` as input arguments and returns an instance of the `BeamCrossSection` record. In the function, all elements of this record are computed and assigned. The record contains a vector of type “Real[numberOfPoints,2]”, i.e., the dimension of this vector changes according to the input argument `numberOfPoints`. It is a bit tricky to define the actual dimensions of an array that is used in an output record of a function. Currently, the only possible way in Dymola is to use a “**redeclaration**”, see section 5.10, where the declaration of the array is given as modifier to the output argument with the actual dimensions and with the prefix “**redeclare**”.

The above approach allows to define pop-down menus in a quite convenient way. There is, however, one drawback: The modifier to the record instance is a function call, i.e., an expression. This means that the record cannot be modified in the plot variable browser, before simulation starts. If for example, the diameter in the example above is changed, Dymola has to re-translate the model. Providing the input arguments to the function as parameters, e.g., as in:

```
parameter Modelica.SIunits.Diameter D = 0.04;
Beam beam(data( crossSection = BeamChoices.circle(diameter=D) ));
```

does not help either currently in Dymola, since Dymola does not allow to change “D” before simulation starts and therefore changing “D” requires again a re-translation of the model.

### 5.5.3 Records with array elements (partially available)

Records may have arrays as elements. For example, in the `Modelica_LinearSystems` library (a free library available from the “File / Libraries” menu in Dymola) linear state space systems are available as:

$$\begin{aligned}\dot{\mathbf{x}} &= \mathbf{A} \cdot \mathbf{x} + \mathbf{B} \cdot \mathbf{u} \\ \mathbf{y} &= \mathbf{C} \cdot \mathbf{x} + \mathbf{D} \cdot \mathbf{u}\end{aligned}\tag{5.7}$$

The data of such a system is basically provided as a record:

```
record StateSpace
  Real A[:, :];
  Real B[size(A,1), :];
```

```

Real C[:, size(A,1)];
Real D[size(C,1), size(B,2)];
end StateSpace;

```

where the 4 matrices are elements of this record. The data is *not* provided as “parameter” since  
 < ... >

## 5.6 External Functions (partially available)

The Modelica language has a basic mechanism to call functions of other programming languages within a Modelica model or a Modelica function. Details are standardized for “C” and “FORTRAN 77” functions<sup>6</sup>.

Dymola supports the calling of external “C”, “FORTRAN 77” and “Java” functions. The details for calling “Java” functions in Modelica are not discussed here, see instead Chapter 8.3.2 in the document “Dymola User Manual Volume 2.pdf” in the installation directory of Dymola under “Dymola\Documentation”.

### 5.6.1 Calling C functions in Modelica (not yet available)

### 5.6.2 Calling C++ functions in Modelica (not yet available)

### 5.6.3 Calling FORTRAN 77 functions in Modelica

Although FORTRAN 77 is a stone-age programming language, it is still a major language to implement numerical algorithms. A huge collection of useful (and free) FORTRAN 77 libraries is available from <http://www.netlib.org>. One goal is that these libraries can be utilized in Modelica.

Calling FORTRAN 77 functions directly in Modelica requires to have a FORTRAN 77 compiler that generates “compatible” code to the C-compiler that is used by Dymola (for translating Modelica models to object code). If such a compiler is available, FORTRAN 77 object libraries can be directly linked to the C-code generated by Dymola.

The problem is that a *portable* mapping of Modelica strings to native FORTRAN 77 code is *not* possible, because the FORTRAN 77 data type CHARACTER is actually mapped to 2 arguments of a function call: One argument for the address of the CHARACTER variable and one (hidden) argument for the length of the CHARACTER variable. How the “hidden” argument is passed for a function call, depends on the FORTRAN 77 compiler at hand (e.g., passed as “call by value”, as “call by reference”, as INTEGER\*2, INTEGER\*4, as last argument to the function call, as argument followed directly the CHARACTER argument, ...). Dymola does not support an automatic mapping (e.g., by introducing automatically the appropriate “hidden” arguments for strings) and therefore the user has to manually introduce the “hidden” arguments for strings, depending on the used FORTRAN 77 compiler.

There is also the slight problem that the FORTRAN 77 name of a function is mapped to another name by the compiler when generating object code. Often, a FORTRAN 77 name like “SOLVE” is mapped to “\_solve” in the object code and therefore the FORTRAN 77 reference in the Modelica definition needs to be “\_solve” and not “SOLVE”.

To summarize, directly using a FORTRAN 77 compiler is complicated and not portable if strings are passed as arguments. This approach is therefore not recommended.

Instead it is recommended to use the free “f2c” translator from <http://www.netlib.org> to transform the FORTRAN code to C and use the C-libraries provided with “f2c” that emulate the FORTRAN run time environment. This is the most portable and less cumbersome way to call FORTRAN 77 functions within Modelica (in any Modelica tool). In the Modelica Standard Library this approach is used for the more advanced numerical functions that operate on matrices (Modelica.Math.Matrices). Here, the standard linear algebra FORTRAN 77 library LAPACK provides sophisticated algorithms to, e.g., solve linear systems of equations or calculate eigen values. In Dymola a C-version of LAPACK is used that has been transformed from FORTRAN 77 to C with the “f2c” tool.

After transforming the FORTRAN 77 code to C, the further usage is, of course, identical to native C-code. However, there is one remaining issue: Arrays of two or more dimensions have a different storage layout in FORTRAN (column wise storage) as in C (row wise storage) and “f2c” does not change this layout

<sup>6</sup>For notational convenience, “functions” means both “functions” and “subroutines”.



scheme. For example, the 3x2 array A

$$\mathbf{A} = \begin{pmatrix} 11 & 12 \\ 21 & 22 \\ 31 & 32 \end{pmatrix} \quad (5.8)$$

is stored in the following way in contiguous memory in the two programming languages:

Table 5.3: Storage layout of matrix A in different programming languages.

C	A = {11, 12, 21, 22, 31, 32}
FORTRAN	A = {11, 21, 31, 12, 22, 32}

Dymola uses the C storage layout for code generation. When an array is passed to a FORTRAN function, the storage layout must first be changed. Dymola performs this change automatically, if “FORTRAN 77” is selected as language. This means, every array with 2 or more dimensions is first copied before it is passed to a FORTRAN 77 function. This procedure is also needed, if the function was transformed with “f2c”.

For example, the FORTRAN 77 functions from the LAPACK library are first transformed with “f2c” to “C”, are translated with the C-compiler and are stored in the object library “Lapack.lib”. Then, function “dgesv” from LAPACK (to solve a linear system of equations) can be accessed in Modelica with the following Modelica definition:

```

function dgesv "Solve A*x=b for x"
  input Real A[:, size(A,1)];
  input Real b[size(A,1)];
  output Real x[size(A,1)] = b;
  output Integer info;
protected
  Real Work [size(A,1), size(A,1)] = A;
  Integer pivots[size(A,1)];
  external "FORTRAN 77" dgesv(size(A,1), 1, Work, size(A,1),
                             pivots, x, size(A,1), info)
  annotation (Library="Lapack");
end dgesv;

```

The “Library” annotation tells Dymola to use the object library “Lapack.lib” in the linking phase. The **external** “FORTRAN 77” definition tells Dymola to use the FORTRAN 77 storage layout when passing arrays, i.e., to copy Modelica arrays from a row wise storage in to a column wise storage scheme. So, even if “dgesv” is a “C” function (since transformed with f2c to C), the function needs to be defined as “FORTRAN 77” and not as “C” in Modelica!

If the copy operation of arrays shall be avoided to enhance efficiency, the FORTRAN function needs to be rewritten (before transformed to C) by exchanging indices of the arrays in the FORTRAN code. Usually, this will not be an option, since too much work and the risk of errors is high.

## 5.7 Modelica as Scripting Language (not yet available)

## 5.8 Component Arrays (not yet available)

## 5.9 Component Coupling by Fields (not yet available)

## 5.10 Replaceable Models (not yet available)



## Part B Modeling of Discontinuous and Variable Structure Systems

All numerical integration algorithms require that the functions to be integrated are continuous and differentiable up to a certain order. In this part it is discussed how systems are handled with the Modelica language and in a Modelica simulation environment where this fundamental prerequisite of an integrator does not hold.

If a physical system is modeled macroscopically detailed enough, all variables are continuous and the following considerations are not necessary. Discontinuities are introduced due to simplified assumptions:

- To reduce the simulation time, e.g., because the integrator does not have to follow a very rapid change of a variable, if this rapid change is approximated by a discontinuity.
- To reduce the identification effort, e.g., because the parameters that define a rapid change of a characteristic do not have to be determined by measurements, if this rapid change is approximated by a discontinuity.

Modelica is very well suited to describe discontinuous and variable structure systems in a numerically reliable way. Still, before trying to express the desired behavior with native Modelica language elements, it is easier to first try to use the model classes already available in the Modelica Standard Library, such as shown in Table 5.1:

*Table 5.4: Modelica libraries with discontinuous and/or variable structure systems*

<b>Modelica library</b>	<b>Features, components</b>
<code>Modelica.Blocks.Logical</code>	Logical blocks (and, or, etc.), timer, ...
<code>Modelica_LinearSystems.Sampled</code>	Sampled data systems, discrete controllers
<code>Modelica.StateGraph</code>	Hierarchical finite state machines
<code>Modelica.Mechanics.Rotational</code>	Coulomb friction, clutches brakes
<code>Modelica.Electrical.Analog.Ideal</code>	Ideal electrical switches, ideal diodes, ...
<code>Modelica.Electrical.Digital</code>	Logical blocks with 9-valued logic

## Chapter 6 Discontinuous Equations

Numerical integration algorithms approximate the solution of differential equation systems by  $n$ -th order polynomials that are attached continuously and smoothly together. It is therefore impossible to describe discontinuous systems precisely, because polynomials do not have discontinuities. Therefore, an integrator has to perform very small step sizes near a discontinuity in order that it is roughly approximated by polynomials. As a result, good integrators with variable step size are usually very slow near a discontinuity (provided the discontinuity has a significant effect on the simulation result).

It is well known how to handle discontinuous systems in a numerical reliable and efficient way by using the following strategy (Cellier 1979):

1. Detect the time instant of the discontinuous change, e.g., by determining the time instant when an indicator signal crosses zero.
2. Hold the integration at this time instant.
3. Perform the discontinuous change.
4. Restart the integration.

As a result of this approach, the signals during the numerical integration are always continuous and differentiable and the signals at the discontinuous point are precisely handled. The time instants where the integration is halted are called “events”. Since at an event instant, variables may be discontinuous, a simulation environment should store at least two signal values at the event time instant: the value when then integration was halted and the value when the integration was restarted (this is, e.g., the case in Dymola).

The Modelica language is designed, such that the above strategy is automatically performed, i.e., the basic prerequisite of an integrator to operate on smooth equations is fulfilled between event instances. Exceptions are Modelica functions, because function results may be discontinuous and it is not possible to trigger events in a function. It is planned to fix this defect of Modelica in a future language version. Furthermore, external functions are outside of the control of a Modelica translator and can therefore lead to any type of misbehavior of an integrator.

There are additionally restrictions in the language to prevent the implementation of models that have a memory during the continuous integration phases, since this would violate the integrator prerequisite in an even more drastic way as discontinuities are doing it. Therefore, it is for example impossible to determine the integrator step size from a Modelica model (of course in an external function with a memory, such types of properties can be determined).

Most important properties of Modelica with respect to discrete systems:

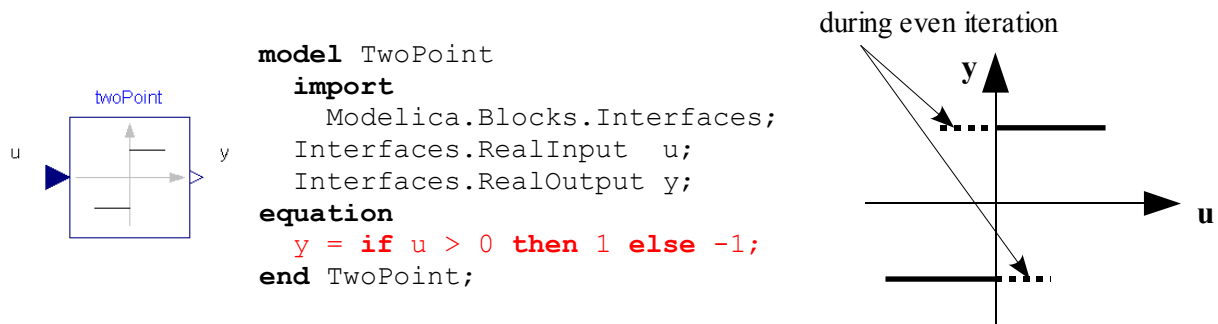
- A variable keeps its value until it is explicitly changed. It is always possible to access the value of a variable, both during continuous integration and at event instants.
- It is only possible to access the left or the right limit of a variable at the actual time instant, but not at previous time instants (to avoid memories during the continuous integration phases). During continuous integration the two values are identical. At an event instant  $t_e$ , a variable name, such as “ $v$ ” is defined to be the right limit “ $v(t_e+\epsilon)$ ” and “ $\text{pre}(v)$ ” is defined to be the left limit “ $v(t_e-\epsilon)$ ” of  $v$  where  $\epsilon$  is infinitesimally small.
- The values of “Integer”, “Boolean”, “String”, “discrete Real”, and “enumeration” variables can only be changed at event instants. It is impossible to change their values during continuous integration. Only (non-discrete) “Real” variables can change their value during continuous integration. Note, a variable declared as “Real” is treated as “discrete Real”, if it appears on the left hand side of an equation or statement in a **when**-clause.
- An event is processed in zero (simulated) time. i.e., Modelica has instantaneous communication. If this is not desired, e.g., because the actual computing time of a sampled data controller shall be taken into account, it has to be explicitly defined, e.g., by triggering two events.

- It is not defined whether two or more events occur at the same time instant. If time synchronization of events is needed, it has to be explicitly programmed, e.g., by accessing the same `Boolean` variable that signals an event, or by explicitly implementing a counter if, say, a “slow” sampling event shall occur at every 5th “fast” sampling event.
- It is not possible to determine the time instant of the “last accepted step of the integrator”. In some special cases, such a feature might be useful, but it is always a “hack” with the danger of severe errors and there are usually better implementation alternatives.

## 6.1 Relation triggered events

The Modelica language has no construct to *explicitly* trigger an event. Instead events are always *implicitly* triggered if a relation, such as “ $v1 > v2$ ”, changes its value. Since relations are the only means in Modelica (and also in programming languages) to change the value of a variable discontinuously, e.g., by an if-clause, every *potential discontinuous* change in a Modelica model or block triggers an event and halts the numerical integration. This basic mechanism is explained in more detail on the basis of the following simple block that models a two-point switch:

Figure 6.1: Two point switch block to demonstrate relation triggered events.



Assume that  $u$  is negative and becomes continuously larger until it is positive. At the beginning,  $y = -1$ , since  $u$  is negative. When the integrator performs a step so that  $u$  is positive, an iteration is started to detect the zero crossing of  $u$  within a small time interval. During this iteration,  $y$  remains still on the else branch, i.e.,  $y = -1$  independent of the value of  $u$ . Once the zero crossing time instant is found precisely enough (usually close to machine precision), the integration is halted, an event is triggered, the “if” clause is changed from the “else” to the “then” branch and the integration is restarted. This means that the above “if”-expression is not equivalent to an “if”-expression of a programming language such as Fortran, C or Java, because the “if” expression remains on the branch that it had previously until the event iteration is finalized, independently of the actual value of the “if”-condition.

The advantage of this approach is that in most cases discontinuous changes are handled automatically in a numeric reliable and efficient way.

If a relation has the form “ $\text{time} \geq \text{discrete-expression}$ ” or “ $\text{time} < \text{discrete-expression}$ ”, it is possible to compute the time instant of the event in advance. In such a case, no event iteration takes place and the integrator adapts its step size so that it steps exactly to the event point, i.e., the simulation is more efficient. Example:

```
y = if time < 1 then 0 else 1;
```

The latter case is called “*time event*” whereas an event is called “*state event*” when an event iteration has to take place.

In some cases it is unnecessary that a relation triggers an event, because the corresponding expression is smooth. The modeler can report such a property with the “`smooth(order, expr)`” built-in operator of Modelica. Example:

```
y = smooth(1, if u > 0 then u^2 else u^3);
```

The first argument to `smooth(...)` reports the differentiability order (order = 0: continuous, order = 1: continuous and first derivative is continuous, order = 2: continuous, first and second derivatives are continuous etc.). Since  $u^2 = u^3 = 0$  for  $u=0$  and  $\text{der}(u^2) = 2*u = \text{der}(u^3) = 3*u^2 = 0$  for  $u = 0$ , ar-

gument `order = 1` in the example above. The actual behavior of the Modelica simulation environment is not defined for the `smooth(..)` operator. The usual interpretation is to not generate an event if `order ≥ 0`. If a statement containing the `smooth(..)` operator is differentiated during the symbolic preprocessing, the “order” argument is decremented for every differentiation. If `order = -1`, the expression is discontinuous and an event has to be generated. If such a statement is differentiated again, a Modelica environment will usually report a translation error, because the derivative of a discontinuous expression results in a Dirac impulse that is difficult to handle.

In some exceptional cases it is necessary to force the Modelica simulation environment to never generate an event. This is performed with the `noEvent(..)` built-in operator. Example:

```
y = noEvent(if u > eps then 1/u else 1/eps);
```

This `if`-expression is taken literally, as in programming languages, i.e., no event is triggered. Without the `noEvent(..)` operator this example might fail, because if `u` is larger as `eps` and then becomes smaller as `eps`, an event iteration would take place. During the event iteration it might be that `u` becomes zero or so small that `1/u` results in an “overflow” leading to a run-time error. The “`noEvent(..)`” operator can only be applied in equations that involve “Real” variables. It is not allowed to apply it for Integer, Boolean, String, enumeration or discrete Real expressions, since otherwise it would be possible to define very drastic changes of a model during continuous integration that cannot be handled by a reliable numerical integration method. For example, during continuous integration the number of equations could be changed (in combination with “when-clauses”, see next section), or the integrator step size could be computed and used to make the model evaluation dependent on the actual integrator step size.

It is sometimes useful to combine the `smooth(..)` and the `noEvent(..)` operators:

```
y = smooth(100, noEvent(if u > eps then 1/u else 1/eps))
```

The reason is that the equation above might be differentiated and the guard to prevent a zero division must be present in a `noEvent(..)`, in order that it is guaranteed that no event is generated (remember: it is tool-dependent, whether the `smooth(..)` operator generates or does not generate events). If Dymola differentiates an expression where the `noEvent(..)` operator is applied, it checks at run-time whether the equation that was differentiated is continuous. Otherwise, the simulation result could be wrong, because differentiating a discontinuous expression leads to Dirac impulses that are not supported by Dymola in this general form. This check of Dymola is avoided by providing the `smooth(..)` operator around the `noEvent(..)` expression. Technically, this statement is wrong, because the expression above is not smooth. A better solution is to really provide a smooth transition from the “`1/u`” to the “`1/eps`” case and provide the correct differentiation order in the `smooth(..)` operator.

## 6.2 Discrete equations and the `pre()` operator

Equations are characterized according to the type of variable that are defined by the equation. Therefore, Modelica has Boolean, Integer, String and enumeration equations, besides Real equations.

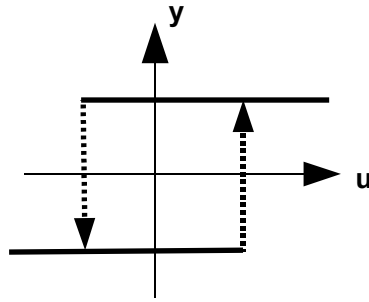
- Continuous equations:

An equation is called “continuous equation”, when the equation defines a relationship between (non-discrete) Real variables. In general, a continuous equation can have the form “`expr1 = expr2`”, where “`expr1`” and “`expr2`” are expressions that evaluate to a Real expression.

- Discrete equations:

An equation is called “discrete equation”, when it defines a discrete variable `v`, that is a Boolean, Integer, String, discrete Real or enumeration variable. It is required that a discrete equation has the form “`v = expr`”, i.e., the discrete variable `v` to be calculated has to be present on the left hand side of the equal sign and the expression on the right hand side of the equal sign evaluates to the base type of `v`.

The following example shows a combination of Real and Boolean equations to define a basic Hysteresis block:



```

model Hysteresis
  import IF=Modelica.Blocks.Interfaces;
  IF.RealInput u;
  IF.RealOutput y;
  Boolean high(start=true, fixed=true);
equation
  high = not pre(high) and u >= 1 or
         pre(high) and u > -1;
  y = if high then 1 else -1;
end Hysteresis

```

The Boolean variable “high” defines whether the output  $y$  is on the “upper” ( $\text{high} = \text{true}$ ) or on the “lower” ( $\text{high} = \text{false}$ ) branch of the characteristic. Since “high” is a Boolean variable, its value can only change at an event instant (because relations only change at event instants). The Boolean equation for “high” contains “ $\text{pre}(\text{high})$ ”, i.e., the value of “high” just before the actual event instant. This is the value of “high” from the previous event instant (because the value of “high” can change only at event instants). Events occur when one of the relations, i.e., “ $u \geq 1$ ” or “ $u > -1$ ”, changes their values, i.e., whenever there is a potential change of the corresponding branch.

It is a “quality of implementation” of the used Modelica simulation environment, whether unnecessary events are generated. For example, a simple implementation will trigger an event whenever “ $u \geq 1$ ” or “ $u > -1$ ” changes its value. However, assume that  $u > 1$  and  $u$  is becoming smaller as 1, then the relation “ $u \geq 1$ ” changes its value, but it will not influence the result (since “ $y$ ” remains on the upper branch) and therefore no event should be generated. In principal it is possible to detect such a situation since, e.g., “ $\text{not pre}(\text{high}) \text{ and } u \geq 1$ ” is defined and therefore this expression is **false**, whenever  $\text{pre}(\text{high})$  is **true**, independently of the value of the relation “ $u \geq 1$ ”. In the Modelica simulation environment Dymola (Dymola 2008), unnecessary events of this type are avoided. The equation for “high” could also be defined with the following equation:

$$\text{high} = u \geq 1 \text{ or } \text{pre}(\text{high}) \text{ and } u > -1;$$

This formulation has the (slight) drawback that an event will always be triggered when “ $u \geq 1$ ” changes its values. Therefore unnecessary events will occur since a Modelica simulation environment cannot detect during translation that some events will not have an effect on the result.

The equation for the Boolean variable “high” is not obvious. How to derive such equations systematically is explained in section XXX. To check whether the equation for “high” is correct, the different possibilities are analyzed:

- $\text{pre}(\text{high}) = \text{false}$  and  $u < -1$ :  
 $\text{high} = \text{true and false or false and false} \rightarrow \text{high} = \text{false}$ ,  
 i.e.,  $y$  remains on the lower branch.
- $\text{pre}(\text{high}) = \text{false}$  and  $-1 < u < 1$ :  
 $\text{high} = \text{true and false or false and true} \rightarrow \text{high} = \text{false}$ ,  
 i.e.,  $y$  remains on the lower branch.
- $\text{pre}(\text{high}) = \text{false}$  and  $u > 1$ :  
 $\text{high} = \text{true and true or false and true} \rightarrow \text{high} = \text{true}$ ,  
 i.e.,  $y$  changes from the lower to the upper branch.
- $\text{pre}(\text{high}) = \text{true}$  and  $u > 1$ :  
 $\text{high} = \text{false and true or true and true} \rightarrow \text{high} = \text{true}$ ,  
 i.e.,  $y$  remains on the upper branch.
- $\text{pre}(\text{high}) = \text{true}$  and  $-1 < u < 1$ :  
 $\text{high} = \text{false and false or true and true} \rightarrow \text{high} = \text{true}$ ,  
 i.e.,  $y$  remains on the upper branch.
- $\text{pre}(\text{high}) = \text{true}$  and  $u < -1$ :  
 $\text{high} = \text{false and false or true and false} \rightarrow \text{high} = \text{false}$ ,  
 i.e.,  $y$  changes from the upper to the lower branch.

Once “high” is computed, it is trivial to calculate the output  $y$  with the Real equation

`y = if high then 1 else -1`.

The declaration of “high” defines the attributes “start = true” and “fixed = true” and we will now discuss the precise meaning of this declaration: For (non-discrete) Real variables, the start attribute defines the value of a variable after initialization, provided fixed = true. This means, initialization has to be carried out so that the variable has the required “start” value after initialization. Often, this requires solving a non-linear algebraic system of equations. For Boolean, Integer, String, discrete Real and enumeration variables “v” this is different: The “start” attribute defines the value of “pre(v)” before initialization, i.e.,  $\text{pre}(v) = v.\text{start}$  before initialization starts. The alternative possibility to require “v = v.start” after initialization would usually lead to nasty initial equation systems involving pre(v) as unknowns. For example, the equations of the Hysteresis block would not have a solution, if high = true and u = -2 would define the initial value problem and it would be in general difficult to compute “pre(v)”, once “v” is given.

The question arises, in which order discrete and continuous equations are evaluated: The fundamental principal of Modelica is to map all language elements to (continuous and/or discrete) equations and to require that the unknown variables are determined such that all equations are fulfilled concurrently, i.e., a set of “n” equations in “n” unknowns has to be solved at every time instant and at every event instant. A first step is usually to sort the equations. The result is an explicit evaluation order of equations and/or of systems of algebraic equations.

During continuous integration the discrete variables cannot change their values (because relations cannot change) and therefore only the continuous equations have to be solved using the values of the discrete variables from the previous event instant. At an event instant, including the initialization that is treated as “initial event”, the continuous and the discrete equations have to be solved concurrently under the assumption that the “pre(v)” values of the variables “v” are given (= values from the previous event).

If  $\text{pre}(v_i) \neq v_i$  for at least one discrete variable  $v_i$ , after the equations have been solved, then  $\text{pre}(v_i) := v_i$  is set and the system of equations is solved again. This event iteration is terminated, once  $\text{pre}(v_i) = v_i$  for all discrete variables  $v_i$ . A solver will usually terminate with an error, if a maximum number of iterations is reached. It is an unresolved problem in Modelica to either detect during translation that event iteration will not converge (indicating an erroneous model) or to modify the semantics of Modelica so that such a situation cannot occur.

As an example, the initialization of the hysteresis block from above is analyzed.

- u > -1:

Since  $\text{pre}(\text{high}) = \text{true}$ , the following equations are solved during initialization:

`high = false and u >= 1 or true and u > -1;`

`y = if high then 1 else -1;`

This results in high = true and y = 1. Since high = pre(high), the initialization is finished.

- u < -1:

Since  $\text{pre}(\text{high}) = \text{true}$ , the following equations are solved during initialization:

`high = false and u >= 1 or true and u > -1;`

`y = if high then 1 else -1;`

This results in high = false and y = -1. Since  $\text{high} \neq \text{pre}(\text{high})$ , the initialization equations are again evaluated using  $\text{pre}(\text{high}) := \text{high} := \text{false}$ , i.e., solving the following equations:

`high = true and u >= 1 or false and u > -1;`

`y = if high then 1 else -1;`

This results in high = false and y = -1. Since high = pre(high), the initialization is finished.

Since the equations of the Hysteresis block are not obvious (at the current stage), one might think to implement the block without using a Boolean equation:

```
// Wrong Modelica model
y = if u >= 1 or (pre(y) > 0.5 and u > -1) then 1 else -1;
```

However, this is not allowed in Modelica because outside of a when-clause, the pre(.) operator can only be applied on *discrete* variables and “y” is a (non-discrete) Real variable. One reason for this restriction is that otherwise the following type of equations could be defined:



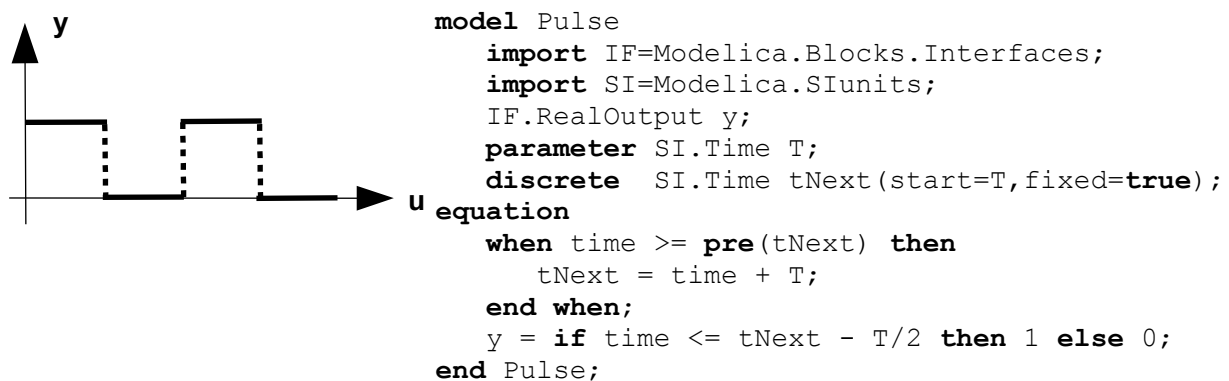
```
// Wrong Modelica model  
y = noEvent(if u >= 1 then u else pre(y)/2);
```

and then it is not clear how `pre(y)` should be computed: Due to the `noEvent(.)` operator, no event is generated when “`u >= 1`” changes its value. The problem is that “**y**” changes its value discontinuously at this time instant and the value of “`y`” before this change is not defined, because the switching time instant is not determined.

## Chapter 7 Instantaneous Equations

### 7.1 When clauses and mapping to instantaneous equations

At an event instance, additional equations can be activated with the “**when**” clause. This is demonstrated in the next example by means of a pulse generator block:



Parameter “ $T$ ” is the period of the pulse. The discrete Real variable “ $tNext$ ” defines the time instance of the next rising edge of the pulse. Due to the start attribute,  $pre(tNext) = T$ , before simulation starts. Variable “ $time$ ” is a predefined variable that is present in all models and defines the actual simulation time. When the condition “ $time \geq pre(tNext)$ ” of the **when**-clause becomes **true**, or more precisely, at the time instant when the condition changes from **false** to **true**, the body of the **when**-clause becomes active. Since the equations in a **when**-clause are only active at one particular time instant, they are called *instantaneous equations*. In the above **when**-clause, the time instant of the next rising edge of the pulse is computed, once the **when**-condition becomes **true**. Outside of the **when**-clause, the output variable  $y$  is easily calculated based on the time instant “ $tNext$ ” of the next rising edge and the desired shape of the pulse signal.

It is essential to distinguish precisely when to use  $pre(tNext)$  and when to use  $tNext$ : In the condition the  $pre(\dots)$  value has to be used, because at the event instant when the condition becomes true, the value of  $tNext$  from the last event has to be used and in the **when**-clause the value of  $tNext$  for the next event is calculated.

Variables of type “Real” that appear at the left hand side of an equal sign in a **when**-clause are defined to be “discrete Real” variables, independently whether they are declared as “Real” or as “discrete Real”. The prefix “discrete” in a declaration is therefore just a safe guard to explicitly state that the modeler expects that this variable is computed in a **when**-clause and therefore can change its value only at event instants. If this is not the case (e.g., due to a typing error), a translation error occurs.

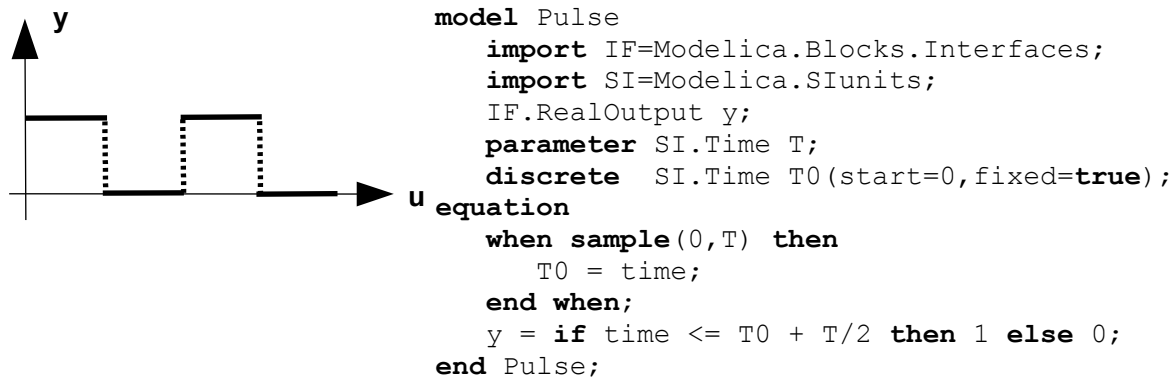
Several built-in operators are provided in Modelica for the handling of discrete equations. The most important ones are listed with a short explanation in the following table:

Table 7.1: Build-in operators for discrete equations.

<b>initial</b> ()	Returns <b>true</b> during the initialization phase and <b>false</b> otherwise.
<b>terminal</b> ()	Returns <b>true</b> at the end of a successful analysis.
<b>sample</b> (start,period)	Returns <b>true</b> and triggers time events at time instants $start + i \cdot period$ ( $i=0,1,\dots$ ). During continuous integration the operator returns always false. Arguments <i>start</i> and <i>period</i> have to be parameter expressions.

<b>pre</b> (v)	Value of variable v before the current event instant. If <b>pre</b> (v) is used outside of a <b>when</b> -clause, v must be a discrete variable. This definition implies that <b>pre</b> (v) is the “left limit” $v(t-\varepsilon)$ of variable v at event instant $t$ ( $\varepsilon \rightarrow 0$ ), if no event iteration takes place.
<b>edge</b> (b)	Returns <b>true</b> when the Boolean variable b changes from <b>false</b> to <b>true</b> , i.e., <b>edge</b> (b) = b and not <b>pre</b> (b).
<b>change</b> (v)	Returns <b>true</b> when variable v changes its value, i.e., <b>change</b> (v) = v <> <b>pre</b> (v), and v is not of base type Real.

The **sample** (..) operator triggers time events in a regular fashion. It can, e.g., be used to implement the Pulse block a little bit simpler:



The equations in a **when**-clause are “*discrete equations*” that are only evaluated at an event instant but not during continuous integration. Therefore, only the already explained restricted form of equations “v = expr” are allowed in a **when** body, where at the left hand side of the equal sign a variable reference must be present. The variable computed by a “discrete equation” has to be explicitly identified, contrary to a “continuous equation”. When the **when**-clause is not active, all discrete variables computed in the **when**-body (= all variables referenced on the left side of the equal signs) are not changed and keep their value from a previous event instant. Note, “discrete equations” are still equations and not assignment statements, e.g., they are sorted with the rest of the other equations (details will be given below). The discussed features are demonstrated by means of some examples:

Although only the restricted form of equations “v = expr” is allowed in a **when**-clause, it is still possible to define implicit algebraic equations in a **when**-clause<sup>7</sup>:

```

// Implicit algebraic equation in when-clause
parameter Real A[:, :] = [1,2;3,4]
parameter Real b[:] = {-3,-4};
Real x[2];
equation
  when sample(0,1) then
    // solve "0 = A*x - b*time" for x
    x = x + A*x - b*time;
  end when;

```

The trick is to add the unknown x on both sides of an implicit equation “0 = f(x)” in order to not change the implicit equation, but to just state that the unknown should be a discrete variable.

It is also possible to have an algebraic loop between discrete and continuous equations. Again, one has to be careful how to formulate the discrete part:

```

// Wrong Modelica code
2*x + 3*y = time;
when sample(0,1) then
  0 = x - 4*y;
end when;

```

<sup>7</sup> The following example is from Hans Olsson

The above code fragment is not allowed in Modelica, because in the **when**-clause no variable is referenced at the left side of the equal sign. As a result, it is not known, which variable ( $x$  or  $y$ ) keeps its value during event instants and is therefore a discrete variable. Slightly rewriting this equation results in correct Modelica code:

```
// Correct Modelica code
2*x + 3*y = time;
when sample(0,1) then
  x = 4*y;
end when;
```

The interpretation is that during continuous integration  $x$  is known (= the value from the previous event instant) and that  $y$  is computed from the continuous equation as:

```
y := (time - 2*x)/3; // equations solved during contin. integration
```

At an event instant, also the discrete equation is active, and we have a linear system of equations for  $x$  and  $y$ :

```
2*x + 3*y = time; // equations if when-clause is active
x - 4*y = 0;
```

Note, the current version of Dymola (Dymola 2008) cannot handle algebraic loops involving continuous and instantaneous equations, so the above example will not translate in Dymola.

On the basis of some examples the semantics of a **when**-clause has been explained. The semantics is defined precisely in the Modelica specification by the following (conceptual) mapping rule that defines how a **when**-clause is mapped to a set of equations<sup>8</sup>:

<pre><b>when</b> condition <b>then</b>   v1 = expr1;   v2 = expr2;   ...   vN = exprN; <b>end when</b>;</pre>	→	<pre>c = condition; ev = <b>edge</b>(c) <b>and not initial</b>(); v1 = <b>if</b> ev <b>then</b> expr1 <b>else pre</b>(v1); v2 = <b>if</b> ev <b>then</b> expr2 <b>else pre</b>(v2); ... vN = <b>if</b> ev <b>then</b> exprN <b>else pre</b>(vN);</pre>
---	---	--

This means that every equation in a **when**-clause is mapped to a standard equation using an if-expression, so that the “**when**-equation” is only active when the “condition” changes from **false** to **true** and in all other cases the value of the discrete variable from the previous event is used. Furthermore, it can be seen that the **when**-equations are *not active during initialization*, but become only active when the simulation starts. After the mapping, equations are present that are handled in the same way as the “continuous equations”. The only new issue is that the set of all equations are solved under the assumption that all **pre**(...) variables (besides the continuous states) are known. As usual, a first step during the solution procedure is to sort the equations. To distinguish this special type of “**when**-mapped” equations from the “continuous equations”, they are called “*instantaneous equations*” because the “**then**” branch is only active at an event instant.

Note, the above mapping can also be performed manually in a Modelica model for Boolean, Integer, String, and enumeration variables. This means that **when**-clauses are not really necessary for these variable types, although the model will be easier to read and to understand if a **when**-clause is used. On the other hand, the above mapping cannot be performed manually in a Modelica model for Real variables. The reason is that the **pre**(...) operator can only be applied on Real variables that are assigned within a **when**-clause. As previously explained, this restriction is present to prevent constructions of the form:

```
// Wrong Modelica model
v = noEvent(if u >= 1 then u else pre(v)/2);
```

because the value of “**pre**(v)” is not well defined, if no event is generated when “v” changes its value discontinuously.

<sup>8</sup> This is a „conceptual“ mapping rule, because it is a „quality of implementation“ whether a Modelica environment implements exactly this mapping rule or whether it uses a different implementation scheme that results in the same semantics.

## 7.2 Sampled data systems and initialization

The implementation of sampled data systems is basically straightforward with the currently discussed Modelica language elements. For example, the simple first order discrete system

$$y(t_i) = a*y(t_i-T) + b*u(t_i), \quad t_i = 0, T, 2*T, \dots$$

can be implemented in the following way:

```
block DiscreteFirstOrderBlockVersion1
  Modelica.Blocks.Interfaces.RealInput  u;
  Modelica.Blocks.Interfaces.RealOutput y;
  parameter Modelica.SIunits.Time T "Sample time";
  parameter Real a, b;
equation
  when sample(0,T) then
    y = a*pre(y) + b*u;
  end when;
end DiscreteFirstOrderBlockVersion1;
```

Due to the **sample**(..) operator, time events are generated at every sampling point  $i*T$ . The input signal  $u$  can be a continuous or discrete variable. In any case, the value of  $u$  at a sampling instant is utilized as discrete input to the discrete system. The output signal  $y$  is a discrete signal since it is assigned in a **when**-clause at every sampling instant. Therefore, **pre**( $y$ ) is the value of  $y$  before the current event occurred, i.e., it is the value of  $y$  from the previous sampling instant (since  $y$  does not change its value during event instants). The implementation is then straightforward in the **when**-clause.

Assume this discrete block is used as controller for a continuous plant. For simplicity we collect the equations together in one model (although in an actual model this would be a **block** and a **model** instance that are connected appropriately together):

```
// Equations of continuous plant
der(xc) = f(xc, uc)
yc = g(xc);

// Equations of discrete controller
when sample(0,T) then
  yd = a*pre(yd) + b*ud;
end when;

// Equations that connect plant and controller together
uc = yd;
ud = yc_ref - yc;
```

A Modelica simulation environment will map the **when**-clause to an instantaneous equation and will sort the equations, resulting in:

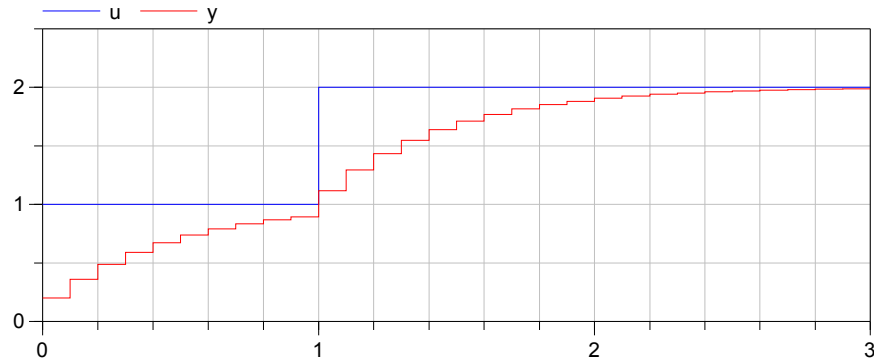
```
// Sorted equations with input: yc_ref, xc, pre(yd);
yc := g(xc);
ud := yc_ref - yc;
c  := sample(0,T);
yd := if edge(c) then a*pre(yd) + b*ud else pre(yd);
uc := yd;
der(xc) := f(xc, uc);
```

During continuous integration the plant equations are evaluated using  $y_d = \text{pre}(y_d)$ , i.e., the value of  $y_d$  from the last event instant. At a sampling instant, the equations of the discrete controller are used to compute the input to the plant based on the value of the plant output  $y_c$ .

The model `DiscreteFirstOrderBlockVersion1` is not yet satisfactory. One issue is that initialization is not explicitly defined. For example, assume that the input  $u$  is 1 at the beginning and after 1 s it jumps to 2, i.e.,  $u = \text{if time} < 1 \text{ then } 1 \text{ else } 2$ :

Figure 7.1: Simulation results of model *DiscreteFirstOrderBlock1* for a step input.

DiscreteFirstOrderBlock1 (T=0.1, a=0.8, b = 0.2)



As can be seen, there is an undesired initial vibration of the output  $y$  of the controller, due to the default initialization. Another issue is that the controller is modeled in a quite idealized form because no computation time of the control algorithm is taken into account and therefore the output  $y$  of the controller acts instantaneously on a given measurement input  $u$  of the controller which is not possible in reality. Since a controller might be constructed from a set of connected discrete blocks, it is not useful to include the computational delay in a discrete block but implement a separate delay block that is connected between the control system and the actuator input of the plant. The above considerations lead to the following implementation. For simplicity, a computational delay of one sample interval is taken into account and is implemented in the first order block and not in a separate block.

```

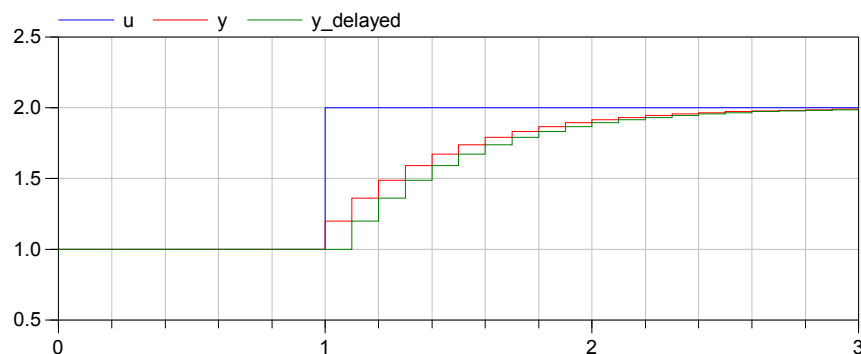
block DiscreteFirstOrderBlockVersion2
  Modelica.Blocks.Interfaces.RealInput u;
  Modelica.Blocks.Interfaces.RealOutput y;
  Modelica.Blocks.Interfaces.RealOutput y_delayed;
  parameter Modelica.SIunits.Time T "Sample time";
  parameter Real a, b;
equation
  when {initial() , sample(0,T)} then
    y = a*pre(y) + b*u;
    y_delayed = pre(y);
  end when;
initial equation
  pre(y) = y;
end DiscreteFirstOrderBlockVersion2;

```

A plot of the same situation as before is shown in the next figure:

Figure 7.2: Simulation results of model *DiscreteFirstOrderBlock2* for a step input.

DiscreteFirstOrderBlock2 (T=0.1, a=0.8, b = 0.2)



The initial vibration of  $y$  is no longer present and the computational delay of one sample instant is clearly shown in variable `y_delayed`.

A **when**-clause can be triggered by several conditions that are defined as a vector of conditions of the form:

```
when {condition1, condition2, ..., conditionN} then
    ...
end when;
```

The semantics is that whenever one of the conditions changes from **false** to **true**, the **when**-clause is activated. The definition in the discrete first order block

```
when {initial(), sample(0,T)} then
    ...;
end when;
```

states that the equations in the **when**-clause are active both at sample instants and during the initialization phase. Together with the stationary initialization equation “**pre**( $y$ ) =  $y$ ” we have therefore a linear system of equations during initialization to compute the unknowns “ $y$ ” and “**pre**( $y$ )”:

```
 $y$  =  $a \cdot \text{pre}(y) + b \cdot u$ ;
pre( $y$ ) =  $y$ ;
```

that has the solution

```
 $y := b \cdot u / (1 - a)$ ;
pre( $y$ ) :=  $y$ ;
```

This means that the output  $y$  remains constant, as long as the input  $u$  does not change, because **pre**( $y$ ) is initialized in such a way that every evaluation of “ $y = a \cdot \text{pre}(y) + b \cdot u$ ” results in **pre**( $y$ ).

### 7.3 Prioritizing event actions

We will now analyze an often occurring situation, by means of the following example:

```
// Wrong Modelica model
when  $h1 > level1$  then
    openValve = true;
end when;
...
when  $h2 < level2$  then
    openValve = false;
end when;
```

The intention of the modeler is to open a valve when “ $h1 > level1$ ” and to close the valve when “ $h2 < level2$ ”. This definition is problematic, because the value of `openValve` is not well defined, when both conditions become **true** at the same time instant (accidentally or by purpose). Mapping the above **when**-clauses to instantaneous equations leads to:

```
 $c1 = h1 > level1$ ;
 $c2 = h2 < level2$ ;
openValve = if edge( $c1$ ) then true else pre(openValve);
openValve = if edge( $c2$ ) then false else pre(openValve);
```

It is now clear that we have *two equations* for *one unknown* “openValve” and this system of equations does not have a unique solution. A Modelica translator will therefore report an error. Due to the fundamental property of Modelica that all language elements are mapped to equations and that every model evaluation requires to solve  $n$  equations in  $n$  unknowns, a Modelica translator can easily detect this type of non-determinism and therefore forces the modeler to resolve this problem by defining an explicit priority of the event actions.

One solution is to rewrite the equations as:

```
 $c1 = h1 > level1$ ;
```

```

c2 = h2 < level2;
when {c1, c2} then
  openValve = if edge(c1) then true else false;
  // or alternatively
  // openValve = edge(c1);
end when;

```

However, this is not very easy to understand. A better alternative is to use **elsewhen**-clauses:

```

equation
  when h1 > level1 then
    openValve = true;
  elsewhen h2 < level2 then
    openValve = false;
  end when;

```

The **when/elsewhen** construct is treated similarly as an “**if then/elseif**” construct, i.e., if the first condition ( $h1 > level1$ ) becomes **true**, then the first branch ( $openValve = \mathbf{true}$ ) is active and the other branches are ignored. Otherwise, the second branch ( $openValve = \mathbf{false}$ ) is active provided the second condition ( $h2 < level2$ ) becomes **true**. There might be additional “**elsewhen**” branches, but no “**else**” branch. The above implementation therefore introduces a priority, so that the relation “ $h1 > level1$ ” has a higher priority as the relation “ $h2 < level2$ ”. Similarly, as in “**if then/elseif**” clauses, the number of equations in every branch of a **when/elsewhen**-clause must be identical. Additionally, the references on the left hand side of “**=**” must be identical in all branches. If every branch has “ $n$ ” equations, such a **when**-clause is (conceptually) mapped to “ $n$ ” equations and therefore the equation system becomes well defined. For example, the **when**-clause above is basically mapped to one equation for “openValve”:

```

b1 = h1 > level1;
b2 = h2 < level2;
openValve = if edge(b1) then true else
             if edge(b2) then false else pre(openValve);

```

Currently, “**elsewhen**” branches in equation sections are not (yet) supported in Dymola and therefore the above code fragment does not work in Dymola. However, Dymola supports **elsewhen**-clauses in:

```

algorithm
  when h1 > level1 then
    openValve := true;
  elsewhen h2 < level2 then
    openValve := false;
  end when;

```

This code fragment is again (conceptually) mapped to one equation in “openValve” and therefore the equation system becomes well defined.

We will now discuss the mapping of an algorithm section to a set of equations in more detail by means of the following example:

```

Integer v1;
Real    v2;
Real    v3;
algorithm
  v1 := expr1;
  v2 := expr2;
  when condition then
    v3 := expr3;
  end when;

```

This algorithm section is (conceptually) mapped in a first phase to

```

Integer v1;
Real    v2;
Real    v3;

```



```

    Boolean c;
algorithm
    // Initialization of algorithm section
    v1 := pre(v1);    // can be always optimized away
    v2 := v2.start;   // can be usually optimized away
    v3 := pre(v3);    // can be always optimized away

    // Transformed statements
    v1 := expr1;
    v2 := expr2;
    c := condition;
    if edge(c) then
        v3 := expr3;
    end if;

```

The transformation of the **when**-clause in to an **if**-clause is similar as for a **when**-clause in an equation section. The only difference is that the assignment to “**pre**(...)” values is performed when entering the algorithm section (in order to handle all possible **if**-clauses, when the corresponding “**when**-clause” is not active). Note, that also non-discrete Real variables are initialized (by using their start value), in order that a modeler cannot introduce a hidden memory. For example, the following code fragment

```

    Real Tlast, stepSize;
algorithm
    if noEvent(time > Tlast) then
        stepSize := time - Tlast;
        Tlast := time;
    end if;

```

would allow to compute the actual integrator step size, if variable `Tlast` would not be explicitly initialized whenever the **algorithm** section is entered. It would be of course better to forbid such models, in order to give better diagnostics to the user. However, it seems difficult to figure out at translation time all such obscure models and therefore the simple approach from above is used in Modelica (since `Tlast` is initialized to its start value zero when the **algorithm** section is entered, `stepSize` is just the difference between the actual time and the initial time, i.e., it is not possible to compute the step size).

In a second (conceptual) mapping phase, the algorithm section after the first mapping phase is transformed in a function call, e.g., the algorithm section above is transformed in the function call:

```

(v1, v2, v3) = algFunction(v_expr, pre(v1), v2.start, pre(v3));

```

where the output arguments are constructed from all variables occurring on the left hand side of the equal signs and all other used variables are the input arguments to the function (“`v_expr`” symbolizes all variable references from “`expr1`”, “`expr2`”, “`expr3`”, and “`condition`”). A modeler cannot perform such a mapping manually, because Modelica functions are “*pure*” functions, e.g., it is not allowed to access any of the discrete operators, such as **pre**(.), **initial**(.) etc., in a Modelica function. Finally, this function call is treated as a set of equations that is solved together with all other equations of a model.

**We have now discussed all language elements in Modelica to model discrete systems. In the following chapters, it is shown how these language elements can be used to model different important situations, but no new language elements are introduced, with the exception of `reinit`(..) in Chapter 11.**

## 7.4 Time synchronization (multi-rate, delays, range-limited sampling)

Modelica has no language elements to synchronize “events” because the synchronization is automatically performed by data flow analysis (since all active equations are treated as a system of equations that has to be solved concurrently and in a first step the equations are sorted). Therefore, if information about a triggered event has to be passed to other parts of a model or to a different model this is simply performed by passing Boolean or Integer variables, e.g.,

```

Boolean limitReached, activateController;

```

```

equation
  limitReached = x > xLimit;  // triggers event
  ...
  activateController = edge(limitReached);  // passes event info

```

#### 7.4.1 Multi-rate controller

There is no guarantee in Modelica that two relations or operators that trigger an event, trigger it at the same time instant. For example, a multi-rate controller may have a base sampling rate for some fast parts of the controller and a multiple of this sampling rate for slow parts of the controller. The modeler would expect to define this as:

```

fastSampling = sample(0, basicSamplingPeriod);
slowSampling = sample(0, 5*basicSamplingPeriod);

```

In most cases this will work as expected. However, round-off errors and the particular epsilon/hysteresis strategy at an event instant of a Modelica simulation environment might not give the expected result that `fastSampling` and `slowSampling` are both true at every 5th instant of the basic sampling period. To get a guaranteed behavior, the time synchronization has to be explicitly programmed with a counter, as shown in the following code fragment:

```

parameter Modelica.SIunits.Time basicSamplingPeriod;
parameter Integer sampleFactor = 5;
Boolean fastSampling, slowSampling;
Integer ticks;
initial equation
  pre(ticks) = sampleFactor - 1;
equation
  fastSampling = sample(0, basicSamplingPeriod);
  when fastSampling then
    ticks = if pre(ticks) < sampleFactor then pre(ticks) + 1 else 1;
  end when;
  slowSampling = fastSampling and ticks >= sampleFactor;

  when {initial(), fastSampling} then
    // equations for fast sampling controller
  end when;

  when {initial(), slowSampling} then
    // equations for slow sampling controller
  end when;

```

Integer variable “ticks” is used to count the number of occurrences of the “fastSampling” events. At every event instant this counter is incremented by 1, until the desired “sampleFactor” is reached. Then “slowSampling” is set to **true** at this time instant and the counter is reset to 1 at the next fastSampling event. During initialization, the controller equations are activated in order to, e.g., initialize in steady state.

#### 7.4.2 Inertial delay

Another often occurring situation is the modeling of an “*inertial delay*”, especially to model digital electrical circuits. The purpose is to delay the input by a given time delay, but only, if the input holds its value for at least the delay time. This can be defined in the following way in Modelica:

```

block InertialDelay
  Modelica.Blocks.Interfaces.IntegerInput u;
  Modelica.Blocks.Interfaces.IntegerOutput y;
  parameter Modelica.SIunits.Time delayTime;
  parameter Integer y_start = 0 "Initial value of y";
protected
  Integer u_delayed(start=y_start, fixed=true);

```

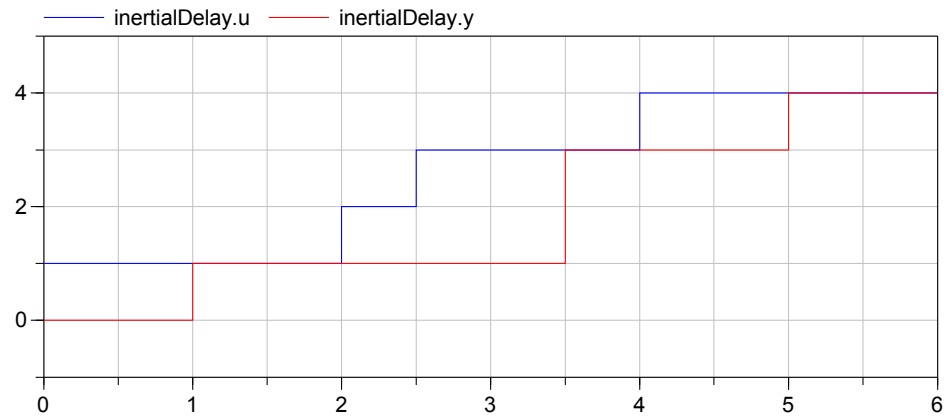
```

Integer u_old      (start=y_start, fixed=true);
discrete Modelica.SIunits.Time tNext(start=delayTime, fixed=true);
algorithm
  when change(u) then
    u_old := u;
    tNext := time + delayTime;
  elseif time >= tNext then
    u_delayed := u;
  end when;
equation
  assert(delayTime > 0, "Positive delayTime required");
  y = u_delayed;
end InertialDelay;

```

Whenever the input “u” is changing, the value of “u” is remembered and a time event is planned after “time + delayTime”. When “u” is not changing during “delayTime”, the planned time event is triggered and the value of “u” is remembered in “u\_delayed”. This value is used as output y. When “u” is changing, the previously planned time event is replaced by a new time event. A typical example of the InertialDelay block with a delay time of 1 s is shown in the next figure:

Figure 7.3: Simulation results of InertialDelay block.



### 7.4.3 On/Off-delay

Modelica has a **delay**(...) operator to model *time delays* of Real, Integer and Boolean signals. Besides the InertialDelay, also other delays need to be modeled, e.g., “**onDelay**(...)”: A rising edge of the Boolean input gives a delayed output. A falling edge of the input is immediately given to the output. This can be defined as:

```

block OnDelay
  Modelica.Blocks.Interfaces.BooleanInput u;
  Modelica.Blocks.Interfaces.BooleanOutput y;
  parameter Modelica.SIunits.Time delayTime;
protected
  Boolean delaySignal(start=false, fixed=true);
  discrete Modelica.SIunits.Time tNext(start=0, fixed=true);
algorithm
  when u then
    delaySignal := true;
    tNext := time + delayTime;
  elseif not u then
    delaySignal := false;
    tNext := time - 1;
  end when;
equation

```

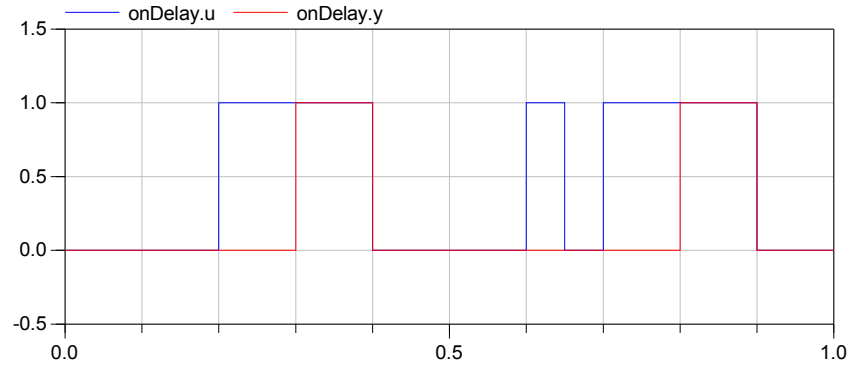
```

    y = delaySignal and time >= tNext;
end OnDelay;

```

The implementation is similar to the `InertialDelay` but with some changes. Simulation results of a typical example of the `OnDelay` block with a delay time of 0.1 s is shown in the next figure:

Figure 7.4: Simulation results of `OnDelay` block with a delay time of 0.1 s.



#### 7.4.4 Range limited sampling

Sampled data systems are often reducing the simulation speed considerably because the integrator has to be re-initialized after every sample instant. This is especially the case for small sample periods. In some applications, sampled data systems do not perform any action most of the time. For example, an ABS braking system only acts, if the brake is pressed and the car starts to slide. Still, in a Modelica simulation model the integration will be halted at every sampling instant. Manuel Remelhe from University of Dortmund has proposed a scheme that considerably enhances simulation speed in such a case:

The idea is to trigger a state event when the controller should act. This state event just sets a flag to trigger a time event at the next sample time. Only then, the sampling actually occurs. As a result, the permanent sampling is removed from the code and replaced by a sampling that only occurs after the exact matching condition is triggered with a state event. In Modelica this situation can be expressed simply as<sup>9</sup>:

```
sampleTrigger = condition and sample(..)
```

for example

```
sampleTrigger = v >= 500 and sample(0, 0.001);
```

This means that time events of the `sample(..)` operator should only be triggered if  $v \geq 500$ . Dymola does not (yet) perform this optimization and triggers time events every 0.001 s, even if  $v < 500$  (but of course “sampleTrigger” is only `true` at the specified time instants). In order to get a simulation speed-up, this situation has to be currently implemented in Dymola as:

```

if condition then
    sampleTrigger = sample(0,T);
else
    sampleTrigger = false;
end if;

```

Here no unnecessary time events are generated. Since it is inconvenient to write these statements at all the needed places, one would encapsulate it in a block. But then, it would be convenient to call this block as a function, as discussed below.

#### 7.4.5 Calling a Modelica block as a function

As shown by four examples, Modelica is suitable to model various kinds of time synchronization mechanisms. The described blocks can be easily used in a block diagram. However, in many cases, the modeler

<sup>9</sup> The proposed implementation is from Hans Olsson.

would like to apply these very basic definitions as a function call, such as,

```
// wrong Modelica code
slowSampling = Counter(fastSampling, samplingFactor);
y = InertialDelay(u, delayTime);
pressButtons = B1 and B2 and not OnDelay(B1 or B2, 0.5);
```

This is not possible, because the functionality was implemented as a block and therefore requires to first instantiate the block and then connect input and output signals:

```
// correct Modelica code
Counter counter(samplingFactor = samplingFactor);
InertialDelay inertialDelay(delayTime = delayTime);
OnDelay onDelay(delayTime = 0.5);
equation
  counter.u = fastSampling;
  counter.y = slowSampling;
  inertialDelay.u = u;
  inertialDelay.y = y;
  onDelay.u = u;
  pressButtons = B1 and B2 and not onDelay.y;
```

This might be improved in a future Modelica version by introducing syntax to call a block as a function.

## 7.5 Algebraic loops between continuous and instantaneous equations

As already mentioned, the Modelica language allows algebraic loops between continuous and instantaneous equations. However, this feature is not supported by Dymola (Dynasim 2008). Usually, this is not a critical issue because models can be rewritten so that an algebraic loop is removed and is replaced by event iteration. In many cases this is a better model of the physical world and should be performed, even if a tool would support algebraic loops. The basic idea is explained on the basis of some examples:

Let us reconsider the simple digital controller that is connected to a continuous plant. The only difference is that the output equation of the plant shall now be explicitly a function of its input signal *uc* (this was not the case previously):

```
// Equations of continuous plant
der(xc) = f(xc, uc)
yc = g(xc, uc);

// Equations of discrete controller
when sample(0,T) then
  yd = a*pre(yd) + b*ud;
end when;

// Equations that connect plant and controller together
uc = yd;
ud = yc_ref - yc;
```

It is not possible to sort the equations of this system in an explicit forward sequence as it was possible previously. In order to make this clearer, the equations that are active at a sample instant are shown and the two simple connection equations at the end are substituted:

```
// Equations at an event instant with input: yc_ref, xc, pre(uc);
der(xc) := f(xc, uc);
yc := g(xc, uc);
uc := a*pre(uc) + b*(yc_ref - yc);
```

The last two equations are a non-linear system of equations to compute “*yc*” and “*uc*” from “*xc*”, “*pre(uc)*”, “*yc\_ref*”.

This algebraic loop is a result of an over-idealization of the digital controller, because the computing time of the controller is not taken into account. Even if we want to model an idealized controller where the comput-

ing time is neglected, the above model does not express our intention. The reason is that it is impossible to measure a signal, compute the actuator signal and apply the actuator signal instantaneously. From a numerical point of view, several evaluations of the plant and the controller equations might be necessary to compute the solution of the non-linear system of equations via, say, a Newton method. However, in reality the controller equations are only evaluated once at the sample instant and not several times. Therefore, the solution of the non-linear system of equations above might not be a solution to our problem. A closer model to reality is the following implementation of the controller:

```
// Improved equations of discrete controller
when sample(0,T) then
    yd = a*pre(yd) + b*ud;
end when;
uc = pre(yd);
```

By this construction, an infinitesimal small time delay is introduced that, as a side effect, removes the algebraic loop. The basic idea is that the equations of the controller are evaluated once at the sample instant and that the computed actuator signal `yd` is afterwards applied to the plant. Event iteration takes place to restart the continuous integration, but during this iteration the controller equations are no longer evaluated. Let us analyze this situation in more detail by inspecting the sorted system of equations (eliminating again the two trivial connection equations):

```
// Sorted equations with input: yc_ref, xc, pre(yd);
uc := pre(yd);
der(xc) := f(xc, uc);
yc := g(xc, uc);
c := sample(0,T);
yd := if edge(c) then a*pre(yd) + b*(yc_ref - yc)
      else pre(yd);
```

A time event is triggered by the `sample(...)` operator. The sorted model equations from above are evaluated at the event instant: As actuator input `uc` the output `pre(yd)` of the previous sample instant is used and the continuous equations of the plant model are evaluated, especially the measurement signal `yc = g(xc, pre(yd))`. Since `sample(...)` triggered an event, “`c = true`”, `edge(c) = true` and therefore the controller equation is evaluated using the measurement signal `yc` as an input. Since `yd ≠ pre(yd)` at the end of the model evaluation, the sorted model equations are again evaluated after setting `pre(yd) := yd`. This means that the continuous equations of the plant are evaluated again, but this time using the just computed controller output as actuator input. Since the `sample(...)` operator is only active once at a time instant “`c = edge(c) = false`” and therefore the controller output just keeps its value, i.e., `yd := pre(yd)`. The event iteration is finished since `yd = pre(yd)` after the model evaluation and the continuous integration is restarted.

Note that the above procedure can be interpreted as solving the original non-linear system of equations by a fixed point iteration scheme and terminate after one iteration. It is clear, that in general this will give a different result as when using, say, a global convergent Newton method and iterate until convergence. Since the fix point iteration describes the desired solution of an infinitesimal small computational time delay, it is the “correct model” and the original model (with the algebraic loop) has to be viewed as erroneous.

Some types of controllers, especially in a car, are most of the time not acting. A typical example is an ABS (anti-blocking brake system) algorithm that is only acting when the brake is pressed and the car starts to slide. In reality, the controller is implemented as a sampled data system. Since an integrator has to be restarted after every sample instant, a simulation would be very inefficient if such a controller would be modeled as sampled data system (an exception is, if the trick from “Range limited sampling” in section 7.4 is used). It is often a good approximation to model it in such a way that it reacts only in particular situations defined by state events thereby neglecting the sampling time.

An ABS algorithm is described in terms of “when this happens → do something; when that happens → do something else” kind of procedure. A straightforward model would therefore use an `algorithm` section with `when/elsewhen`-clauses to model the controller.

A very simplified model of the sketched system is shown in the next example<sup>10</sup>:

<sup>10</sup>This example is from Michael Tiller.

```
// Wrong Modelica model
model ReactiveController1
  Real w, a, tau;
  initial equation
    tau = 10;
  equation
    a = tau - w;
    der(w) = a;
  algorithm
    when a <= 4 then
      tau := 0;
    end when;
end ReactiveController1;
```

On first view, this model is uncritical, because the discrete variable “tau” is set to zero in a **when**-clause and then applied to the continuous equation “a = tau - w”. Mapping the **when**-clause to an instantaneous equation results in:

```
c = a <= 4;
tau = if edge(c) then 0 else pre(tau);
a = tau - w;
der(w) = a;
```

However, it is not possible to sort these equations in a forward sequence, because “a” is needed to compute “tau” and “tau” is needed to compute “a”, i.e., an algebraic loop is present. Similarly to the previous example, this indicates a modeling error. The reason is that again in reality, a small time delay is present that is needed to compute “tau”. For an idealized controller this time delay can be approximated by an infinitesimal small delay, but it cannot be just completely removed. The “correct model”, which has again no longer an algebraic loop, is represented by the following model:

```
// Correct Modelica model
model ReactiveController2
  Real w, a, tau, tau_controller;
  initial equation
    tau_controller = 10;
  equation
    tau = pre(tau_controller);
    a = tau - w;
    der(w) = a;
  algorithm
    when a <= 4 then
      tau_controller := 0;
    end when;
end ReactiveController2;
```

A similar analysis as previously shows, that at the event instant when “a” became smaller as 4, the continuous equations are evaluated with the value of “tau\_controller” that was present just before the event occurred. The new value of “tau\_controller” is then set to zero and since **pre**(tau\_controller) ≠ tau\_controller at the end of the model evaluation, the model is re-evaluated after setting **pre**(tau\_controller) := tau\_controller. In this second evaluation phase, the continuous equations are evaluated with tau = 0. The equation in the **when**-clause is not active, because “a <= 4” was **true** in the previous evaluation and is **true** in this new evaluation and therefore **edge**(c) = **false** (“c = a <= 4”). Therefore, **pre**(tau\_controller) = tau\_controller, and the event iteration is finished, i.e., the integration is restarted.

## Chapter 8 The Synchronous Principle

In the previous chapters we have discussed the basic language elements of Modelica to model discrete and combined continuous/discrete systems. In this chapter the basic underlying principle of Modelica is summarized and compared with other *synchronous languages*.

Synchronous languages (Halbwachs 1993, Benveniste et. al. 2003), such as SattLine (Elmqvist 1992), Lustre (Halbwachs et.al. 1991), Signal (Gautier et.al. 1994) or Esterel (1999) are used to model discrete controllers to yield safe implementations for real-time systems and to verify important properties of the discrete controller before executing it. These languages are used in industrial applications since quite a long time. For example, SattLine has a large installation base at ABB in the process industry. Lustre is the basis of the commercial SCADE tool used, e.g., to generate certified real-time programs for aircraft controllers.

Synchronous languages have one of the implementation models shown in the next table (Benveniste et al 2003) that can be categorized as event driven (left) and sample driven (right):

Table 8.1: Implementation models of synchronous languages.

<i>event driven</i>	<i>sample drive</i>
<pre>Initialize memory <b>for</b> each input event <b>loop</b>   compute outputs   update memory <b>end for</b></pre>	<pre>Initialize memory <b>for</b> each clock tick <b>loop</b>   read inputs   compute outputs   update memory <b>end for</b></pre>

One essential issue is that there is a strict alternation between environment and program actions. Once the input signals are read, the program starts and computes its output. During that time, any change to the inputs is ignored. When all outputs have been computed or at a given time determined by a clock, the output values are fed back to the environment and the program waits for the start of the next cycle. The other essential ingredients of synchronous languages are:

- *Instantaneous communication*, i.e., in the “compute outputs” part of the program, no delay of any form is present, i.e., all variables “act instantaneously” (without any time delay or infinitesimal micro steps contrary to, e.g., in VHDL).
- *Deterministic concurrency* by requiring that the parallel composition of two program parts is the conjunction of the reactions of the two parts. This may lead to algebraic loops (also called constraints) that are handled in various ways in the different languages. For example, Lustre forbids algebraic loops.

We will now discuss the relationship of Modelica to synchronous languages. It is clear that it cannot be a one-to-one relationship, because (standard) synchronous languages operate on discrete entities whereas Modelica operates on a combination of continuous and discrete entities. The following definitions are needed:



Table 8.2: Definitions needed to define the Modelica execution semantic.

<i>discrete equation</i>	A discrete equation is either of the following two forms: 1. An equation of the form “ $v = \text{expr}$ ” or a statement of the form “ $v := \text{expr}$ ”, where, “ $v$ ” is of type <code>Boolean</code> , <code>Integer</code> , <code>String</code> or <code>enumeration</code> . If “ $v$ ” is a record or array reference it is (conceptually) mapped to a set of scalar variables/equations. 2. An equation or a statement in a <b>when</b> -clause <sup>11</sup> that is (conceptually) mapped to an equation or statement as defined in section 7.1. This type of discrete equation is also called “instantaneous equation”.
<i>differential equation</i>	An equation or a statement where the <b>der</b> ( . . ) operator is applied at least on one variable reference.
<i>algebraic equation</i>	An algebraic equation is an equation or a statement that is neither a differential equation nor a discrete equation.
<i>discrete variable</i>	A variable of type <code>Integer</code> , <code>Boolean</code> , <code>String</code> , <b>enumeration</b> or a variable of type <code>Real</code> that appears on the left hand side of an equal sign in a <b>when</b> -clause.
<i>continuous state</i>	A subset of the variables on which the “ <b>der</b> ( . . )” operator is applied or that are defined with the <code>StateSelect.prefer</code> or <code>StateSelect.always</code> attribute. The selection of the proper subset is the task of the Modelica translator.
<b>pre</b> (v) <i>operator</i>	Value of variable $v$ before the current event instant. If <b>pre</b> (v) is used outside of a <b>when</b> -clause, $v$ must be a discrete variable. This definition implies that <b>pre</b> (v) is the “left limit” $v(t-\varepsilon)$ of variable $v$ and “ $v$ ” is the “right limit” $v(t+\varepsilon)$ at event instant $t_e$ ( $\varepsilon \rightarrow 0$ ), if no event iteration takes place. Other discrete operators are expressed as function of <b>pre</b> ( . . ), e.g., <b>edge</b> (v) = $v$ <b>and not</b> <b>pre</b> (v) .

We are now in the position to state the operational model of Modelica informally:

### Operational model of Modelica

A Modelica model is mapped to a set of differential, algebraic and discrete equations. The differential and algebraic equations are solved with a numerical integration method. During this phase, the values of discrete variables do not change and the discrete equations are not taken into account. Whenever a relation (e.g.,  $x1 > x2$ ), that is not part of a **noEvent** ( . . ) operator, changes its value, the integration is halted<sup>12</sup>. The halt of the integration is called “event”.

At the event instant, the continuous states and the **pre** ( . . ) variables (and external input variables, if they appear) are treated as known, and their values do not change during the event. All other variables are computed, such that the differential, algebraic and discrete equations are fulfilled concurrently. This is usually performed by sorting the equations and by solving remaining implicit algebraic systems of equations by an appropriate numerical method.

If there is at least one discrete variable with  $v \neq \text{pre}(v)$  after the event has been processed, a new event instant is triggered, **pre** (v) :=  $v$  is set, and the variables are again computed according to the previous paragraph. This phase is called “event iteration”. If an event is finalized with  $v = \text{pre}(v)$  for all discrete variables  $v$ , the integration is restarted and the differential and algebraic equations are again solved with a numerical integration method.

The above informal description can be formally described by the following procedure:

<sup>11</sup> Only equations of the form “ $v = \text{expr}$ ” or statements of the form “ $v := \text{expr}$ ” are allowed in a **when**-clause. Variable “ $v$ ” on the left hand side of the equal sign can have a base type of `Real`, `Integer`, `Boolean`, `String`, or `enumeration`, and/or it can be a record, record element, array or array element reference of one of these base types.

<sup>12</sup> The time instant at which a relation is changing its value is determined up to a precision determined by the integration method. It is usually in the order of  $10^{-14}$  s

In a first step, a Modelica translator transforms a hierarchical Modelica model into a “flat” set of Modelica “statements”, consisting of the equation and of all used components by:

- Expanding all class definitions (flattening the inheritance tree) and adding the equations and assignment statements of the expanded classes for every instance of the model.
- Replacing all **connect**-equations by the corresponding equations of the connection set.
- Mapping all **algorithm** sections to equation sets.
- Mapping all **when**-clauses to equation sets.

As a result of this transformation process, a set of equations is obtained consisting of differential, algebraic and discrete equations of the following form where ( $\mathbf{v} := [\dot{\mathbf{x}}; \mathbf{x}; \mathbf{y}; t; \mathbf{m}; \mathbf{pre}(\mathbf{m}); \mathbf{p}]$ ):

$$\mathbf{m} = \mathbf{f}_m(\mathbf{v}, \mathbf{relation}(\mathbf{v})) \quad (8.1)$$

$$\mathbf{0} = \mathbf{f}_x(\mathbf{v}, \mathbf{relation}(\mathbf{v})) \quad (8.2)$$

and where

Table 8.3: Meaning of variables in the operational model of Modelica

<b>p</b>	Modelica variables declared as parameter or constant, i.e., variables without any time-dependency.
$t$	Modelica variable time, the independent (real) variable.
<b>x(t)</b>	Modelica variables of type <code>Real</code> , appearing differentiated.
<b>m(te)</b>	Modelica variables of type <b>discrete</b> <code>Real</code> , <code>Boolean</code> , <code>Integer</code> , <code>enumeration</code> which are unknown. These variables change their value only at event instants $te$ . <b>pre(m)</b> are the values of <b>m</b> immediately before the current event occurred.
<b>y(t)</b>	Modelica variables of type <code>Real</code> which do not fall into any other category (= algebraic variables).
<b>relation(v)</b>	A relation containing variables $v_i$ , e.g. $v1 > v2$ or $v3 \geq 0$ and the relation is not within a <b>noEvent</b> (...) operator definition.

For simplicity, the special cases of the **reinit**() operator, as well as of higher index DAEs are not contained in the equations above and are not discussed below.

The generated set of equations is used for simulation and other analysis activities. Simulation means that an initial value problem is solved, i.e., initial values have to be provided for the states **x**. The equations define a DAE (Differential Algebraic Equations) which may have discontinuities, a variable structure and/or which are controlled by a discrete-event system. Such types of systems are called hybrid DAEs. Simulation is performed in the following way:

1. The DAE (8.2) is solved by a numerical integration method under the assumption that the discrete variables **m**, **pre(m)**, and **relation(v)** are kept constant (**pre(m)** = **m**, due to the event handling). Therefore, (8.2) is a continuous function of continuous variables and the most basic requirement of numerical integrators is fulfilled.
2. During integration, the relations **relation(v)** from (8.1) and (8.2) are monitored. If one of the relations changes its value, an event is triggered, i.e., the exact time instant of the change is determined and the integration is halted.
3. At an event instant, (8.1) and (8.2) are a mixed set of algebraic equations which is solved according to the iteration procedure given below.
4. After an event is processed, the integration is restarted with.

Note, that the values of **m** (all **discrete** `Real`, `Boolean`, `Integer` and `enumeration` variables) are only changed at an event instant and that these variables remain constant during continuous integration. At every event instant, including the initial event, new values of the discrete variables **m** are determined according to the following iteration procedure:

```

known variables : x, t, p
unknown variables: dx/dt, y, m, pre(m)
// pre(m) = value of m before event occurred
loop
  solve (8.1) and (8.2) for the unknowns, with pre(m) fixed
  if m == pre(m) then break
  pre(m) := m
end loop

```

Solving (8.1) and (8.2) for the unknowns is non-trivial, because this set of equations contains not only `Real`, but also `Boolean`, `Integer` and `enumeration` unknowns. Usually, in a first step these equations are sorted. In many cases the `Boolean`, `Integer` and `enumeration` unknowns can be just computed by a forward evaluation sequence. In some cases, there remain systems of equations (e.g. for ideal diodes, Coulomb friction elements) and specialized algorithms have to be used to solve them. In other cases, algebraic loops between `Boolean`, `Integer` and `enumeration` equations are present. This is treated as an error.

Modelica is called a *synchronous language*, because it has the most important properties of (discrete) synchronous languages: First, Modelica has *instantaneous communication*, since at an event instant all equations have to be fulfilled concurrently. If needed, it is possible to model the computing time by explicitly delaying the assignment of variables. Second, unknown variables are uniquely computed from a system of equations<sup>13</sup>, i.e., Modelica has *deterministic concurrency*. This implies that the number of active equations and the number of unknown variables in the active equations at every time instant are identical. This feature is sometimes also called “*single assignment rule*”, meaning that a variable is defined by one equation (this phrase is not completely correct, because a variable might be determined from an algebraic system of equations).

Modelica does not have one of the implementation schemes of synchronous languages sketched above (event driven or sample drive model) because Modelica handles continuous and discrete equations and then such an event model is not possible, because time is advanced according to the numerical integration of the continuous system and the triggered events and not solely by a clock or by events. If needed, it would be possible to introduce in Modelica a new type of restricted model class, say “controller”, that corresponds to a data flow oriented synchronous language like SattLine or Lustre: The “controller class” shall have all the properties of a “block class” and additionally the restriction that inside the “controller” the operators “`der( . )`” and “`sample( . )`” and the predefined variable “`time`” cannot be used.

The definition of the event iteration is not yet satisfactory in Modelica, because there is no guarantee that event iteration converges. Practically, a Modelica tool will terminate a simulation with an error, if the number of event iterations becomes larger as a predefined value (say a maximum of 20 iterations). This is of course not acceptable when using a Modelica model in a real-time system. Therefore, in the current controller applications of Modelica, the real-time model is usually a sampled data system since event iteration does not occur here. It would be useful to introduce a better solution in a future Modelica release to widen the application area of Modelica in the real-time area.

Historical note: The basic feature of Modelica to solve a system of differential, algebraic and discrete equations at every time instant and at every event instant, was developed by Hilding Elmqvist based on the same principle used for continuous equations in the Dymola language (Elmqvist 1978) and for discrete equations in the SattLine language (Elmqvist 1992). This basic idea was further developed and improved in (Elmqvist et. al. 1993, Otter et. al. 1999, Elmqvist et. al. 2001).

<sup>13</sup>Non-linear equations may have multiple solutions. However, starting from appropriate defined initial conditions, the desired solution is usually uniquely identified.

## Chapter 9 Finite State Machines

In many applications a user would like to define a discrete algorithm graphically as a finite state machine, e.g., as a Petri net, a Sequential Function Chart (SFC) or a Statechart. In this chapter hints are given, how certain classes of finite state machines can be defined in Modelica. In general, only formalisms can be implemented conveniently in Modelica that are deterministic and have a black box module behavior. This means, e.g., that most Statechart types cannot be implemented in a convenient way because they may have transitions over several hierarchies and the hierarchical layers are not “hidden” in a “black box” but are visible on the highest layer.

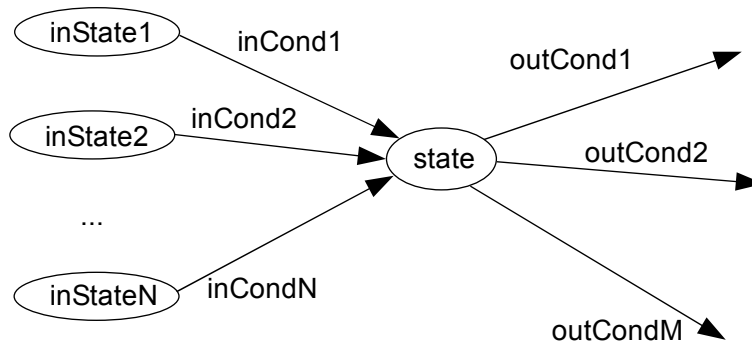
In , the Modelica.StateGraph library is described that can be directly utilized in a large class of discrete event applications. It is a Modelica based hierarchical state machine with a similar modeling power as Statecharts.

In this chapter, the basic method is explained how to implement state machines. This mechanism was utilized to implement the StateGraph library, but it can also be used, to implement other types of deterministic automata, such as prioritized Petri nets.

### 9.1 Implementation of Finite State Machines

There are several ways to implement state machines in Modelica. If components shall be implemented in a library, it is usually desired to have an “object-oriented” implementation where the equations use only information from the connected elements, and nothing else. The basic mechanism is explained by means of the following figure:

Figure 9.1: General method to implement a state machine in Modelica



State “state” of the finite state machine has N input and M output conditions. This situation can be described by the following Boolean equation:

```

state = pre(inState1) and inCond1 or
        pre(inState2) and inCond2 or
        ...
        pre(inStateN) and inCondN or
        pre(state) and not (outCond1 or outCond2 or ... or outCondM);
  
```

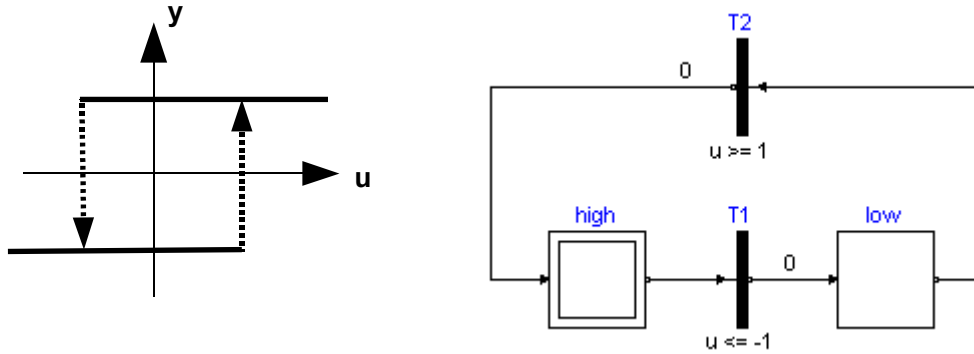
A finite state machine is now just defined by providing this type of Boolean equation for every discrete state. The Boolean equation can be adapted to the peculiarities of the desired discrete formalism. For example, in Petri nets there must be an additional part that checks whether the following step does not have a token. If the state machine is modularized in “state” and “transition” objects, the above equation must be split into parts that are partly present in the “state” and partly present in the “transition” component.

The Boolean equation above can be also utilized to implement very simple finite state machines in order

to avoid some overhead associated for example with a StateGraph model.

Let us apply this technique on the simple Hysteresis block discussed in section 6.2 on page 111, which has the characteristic shown in the left part of Figure 9.2 below, and can be described by the simple state machine visualized in the right part of the figure:

Figure 9.2: Hysteresis block defined by a simple state machine



For every branch of the Hysteresis block one discrete state (“high” and “low”) is introduced. If on the upper branch, state “high”, the state machine switches to the lower branch, state “low”, if  $u \leq -1$ . When in state “low”, it switches back to the upper branch when  $u \geq 1$ . Applying the general formula from above, leads to the following two Boolean equations:

$$\begin{aligned} \text{high} &= \text{pre}(\text{low}) \text{ and } u \geq 1 \text{ or pre}(\text{high}) \text{ and not } u \leq -1; \\ \text{low} &= \text{pre}(\text{high}) \text{ and } u \leq -1 \text{ or pre}(\text{low}) \text{ and not } u \geq 1; \end{aligned}$$

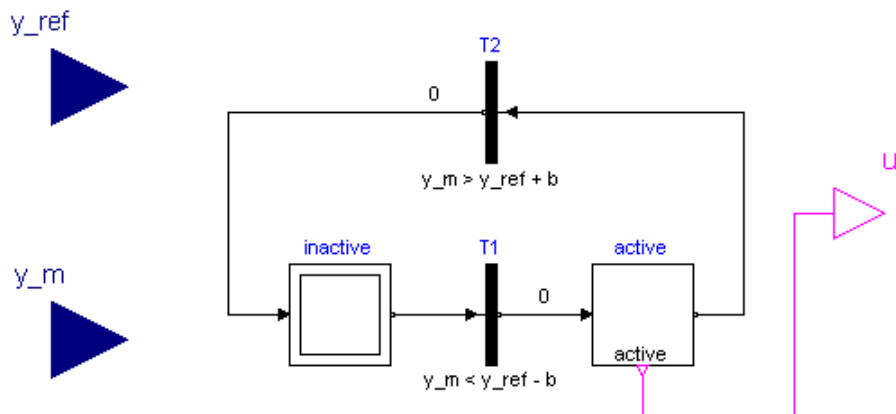
Since  $\text{low} = \text{not high}$ , the two Boolean equations can be simplified to:

$$\begin{aligned} \text{high} &= \text{not pre}(\text{high}) \text{ and } u \geq 1 \text{ or pre}(\text{high}) \text{ and not } u \leq -1; \\ \text{low} &= \text{not high}; \end{aligned}$$

For simplicity the second Boolean equation can be removed. The remaining Boolean equation was used in the Hysteresis block in section 6.2. It is now clear how this equation was derived from a finite state machine.

Another example is shown in the next figure that describes a simple two-point controller (e.g. a temperature control):

Figure 9.3: Two-point controller defined with state machine.



At the left side, “ $y_{\text{ref}}$ ” is the reference signal, and “ $y_{\text{m}}$ ” is the measurement signal that should be close to its reference. If “ $y_{\text{m}}$ ” is smaller as “ $y_{\text{ref}} - b$ ” where “ $b$ ” is a pre-defined bandwidth, the controller switches from its “inactive” in its “active” step and the Boolean output “ $u$ ” becomes true. When the measurement signal “ $y_{\text{m}}$ ” becomes larger as “ $y_{\text{ref}} + b$ ”, it switches back to “inactive” and the output “ $u$ ” becomes **false**. Again, the method of figure Figure 9.1 leads to the following implementation:

```

active    = pre(inactive) and y_m < y_ref+b or
           pre(active) and not y_m > y_ref+b;
inactive  = not active;
u         = active.active

```

and therefore

```

u = not pre(u) and y_m < y_ref+b or pre(u) and not y_m > y_ref+b;

```

## 9.2 Event iteration

Both for the StateGraph library and the manual implementation discussed in the previous section, event iteration takes place at every event instant. Basically, all equations of the respective state machine are evaluated to perform (usually) one transition. Since at least for one state, “**pre**(state)  $\neq$  state”, the equations are again evaluated after setting “**pre**(state) := state”. The event iteration terminates until no transition fires any more. This approach has several drawbacks:

1. It is not guaranteed that the event iteration terminates. This is critical (and not acceptable) for a real-time control system.
2. Even if the event iteration terminates, the number of iterations until it terminates is usually not known in advance. This is also critical for a real-time control system.
3. The complete model (i.e., also the continuous parts) is evaluated in every iteration which means that after every transition the inputs are newly measured and the outputs are newly applied on a continuous plant. In reality this is not the case, because new inputs are only measured and outputs are newly applied only if the iteration in the state machine is finished.

The first of these issues could be fixed by evaluating the state machine equations only once at every sample instant. For example, the basic equation for one state could be changed to:

```

when sampling then
  state = inState1 and inCond1 or
          inState2 and inCond2 or
          ...
          inStateN and inCondN or
          pre(state) and not (outCond1 or outCond2 or ... or
                              outCondM);
end when;

```

and “sampling” is defined at the highest hierarchical level as **sample**(0,T), i.e., generating time events every T seconds. This means that at every sample instant the equations of a state machine are evaluated just once which in turn implies that usually at most one transition fires. The severe drawback of this approach is that the sample period will significantly restrict the step size of the continuous integration although most of the time the output signals applied to the continuous plant will not change.

The efficiency of the event iteration can be improved with the following approach: For the sorting algorithm (BLT partitioning) the “**pre**(...)” values are assumed to be known. The idea is to perform the event iteration not over all model equations, but only over the minimal part. This part can be determined by assuming in a first step that **pre**(...) values are unknown and that the equation “**pre**(v) = v” is added. BLT-partitioning will then result in algebraic loops and these loops will identify the minimal parts for the event iteration. In a second step, BLT partitioning is applied again under the assumption that “**pre**(v)” is known. The event iteration has then only to be applied on the identified minimal parts. The effectiveness of this algorithm can be checked at the following code fragment where a sampled data system is implemented not with the **sample**(...) operator but by Boolean equations:

```

equation
  when time >= pre(tNext) then
    tNext = pre(tNext) + samplePeriod;
    // controller equations
  end when;

```

The above code fragment generates time events at every samplePeriod time instant. At every sample in-

stant, the whole model equations need to be evaluated at least twice, since after the first evaluation “**pre**(tNext)  $\neq$  tNext” and therefore an event iteration has to take place.

The proposed approach adds the equation “**pre**(tNext) = tNext” and identifies the mapped equation:

```
condition = time >= pre(tNext);
tNext = if edge(condition) then pre(tNext) + samplePeriod
      else pre(tNext);
pre(tNext) = tNext;
```

as algebraic loop, since tNext and **pre**(tNext) are not used at any other place. As a result, a local event iteration can be used when generating code:

```
// Pseudo code of local event iteration
if event() then
  loop
    condition := time >= pre(tNext);
    tNext := if edge(condition) then pre(tNext) + samplePeriod
          else pre(tNext);
    break if pre(tNext) == tNext;
    pre(tNext) := tNext;
  end loop;
end if;
```

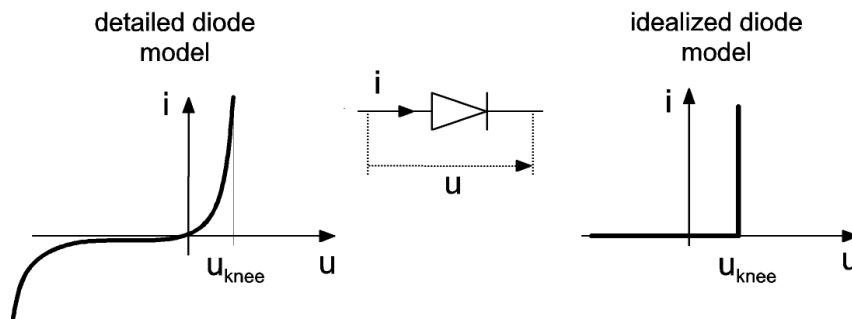
which in turn implies that the model equations need to be evaluated only once at every sample instant.

## Chapter 10 Variable Structure Systems

### 10.1 Parameterized Curve Descriptions

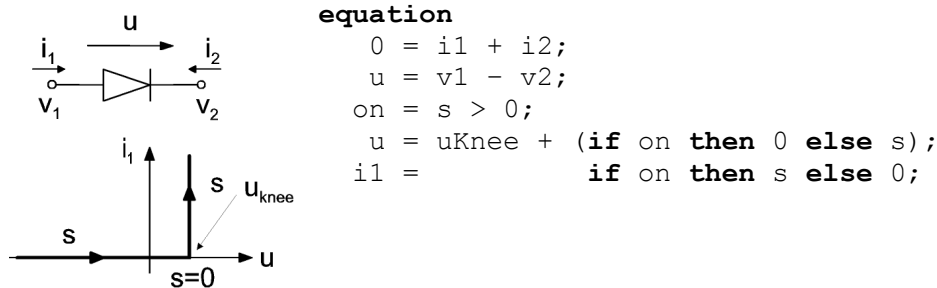
If a physical component is modeled (macroscopically) precisely enough, there are no discontinuities in a system. When neglecting some “fast” dynamics, in order to reduce simulation time and identification effort, discontinuities may appear. As a typical example, consider modeling of a diode, where  $i$  is the current through the diode and  $u$  is the voltage drop between its pins:

Figure 10.1: Detailed and idealized diode characteristic.



A detailed nonlinear characteristic is shown on the left part of the figure. If the detailed switching behavior around the origin is negligible compared to other model phenomena, it is often sufficient to approximate the characteristic by the ideal diode model on the right part of the figure. This corresponds to an ideal switch. This abstraction typically gives a simulation speedup of 1 to 2 orders of magnitude with respect to the detailed diode characteristic.

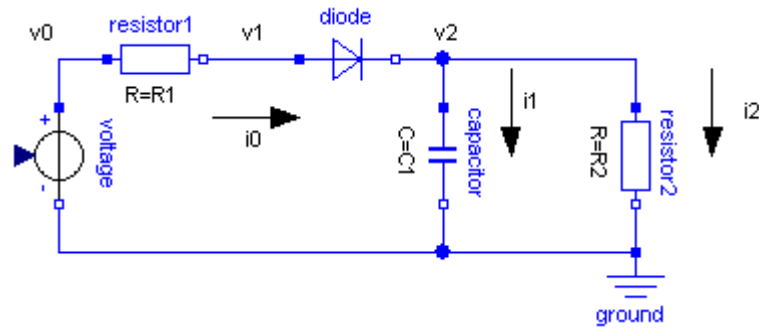
It is straightforward to model the detailed diode curve, because the current  $i$  has just to be given as (analytic or tabulated) function of the voltage drop  $u$ . It is more difficult to model the ideal diode curve in the right part of Figure 10.1 because the current  $i$  at  $u = u_{\text{knee}}$  is no longer a function of  $u$ , i.e., a mathematical description in the form  $i = i(u)$  is no longer possible. This problem can be solved by introducing a curve parameter  $s$  and describing the curve as  $i = i(s)$  and  $u = u(s)$ . This description form is more general and allows defining an ideal diode uniquely in a declarative way:



In order to understand the consequences of parameterized curve descriptions, the ideal diode is used in the simple rectifier circuit of Figure 10.2:



Figure 10.2: Simple electrical circuit to demonstrate the handling of an ideal diode model.



Collecting the equations of all components and connections, as well as sorting and simplifying the set of equations under the assumption that the input voltage  $v_0(t)$  of the voltage source is a known time function and that the states (here:  $v_2$ ) are assumed to be known and the derivatives should be computed, leads to

```

on = s > 0;
u = v1 - v2;
u = u_knee + (if on then 0 else s);
i0 =          if on then s else 0;
R1*i0 = v0 - v1;

i2:= v2/R2;
i1:= i0-i2;
der (v2) := i1/C;

```

The first 5 equations build a system of equations in the 5 unknowns  $on$ ,  $s$ ,  $u$ ,  $v_1$  and  $i_0$ . The remaining assignment statements are used to compute the state derivative **der** ( $v_2$ ). During continuous integration the Boolean variables, i.e., “on”, are fixed and the Boolean equations are not evaluated. In this situation, the first equation is not touched and the next 4 equations form a linear system of equations in the 4 unknown variables  $s$ ,  $u$ ,  $v_1$  and  $i_0$  which can be solved by Gaussian elimination. An event occurs if one of the relations (here:  $s > 0$ ) changes its value.

At an event instant, the first 5 equations are a mixed system of discrete and continuous equations which cannot be solved by, say, Gaussian elimination, since there are both **Real** and **Boolean** unknowns. However, appropriate algorithms can be constructed:

- (3) Make an assumption about the values of the relations in the system of equations.
- (4) Compute the discrete variables.
- (5) Compute the continuous variables by Gaussian elimination (discrete variables are fixed).
- (6) Compute the relations based on the solution of (2) and (3). If the relation values agree with the assumptions in (1), the iteration is finished and the mixed set of equations is solved. Otherwise, new assumptions on the relations are necessary, and the iteration continues.

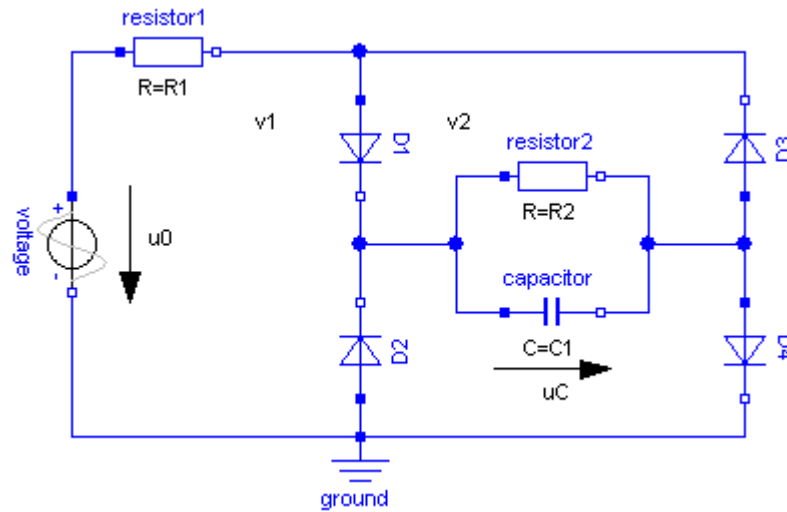
Useful assumptions on relation values are for example:

- (d) Use the relation values computed in the last iteration and perform a fixed point iteration (the convergence can be enhanced by some algorithmic improvements).
- (e) Try all possible combinations of the values of the relations systematically (exhaustive search).

In the above example, both approaches can be simply applied, because there are only two possible values ( $s > 0$  is **false** or **true**). However, if  $n$  switches are coupled, there are  $n$  relations and therefore  $2^n$  possible combinations which have to be checked in the worst case.

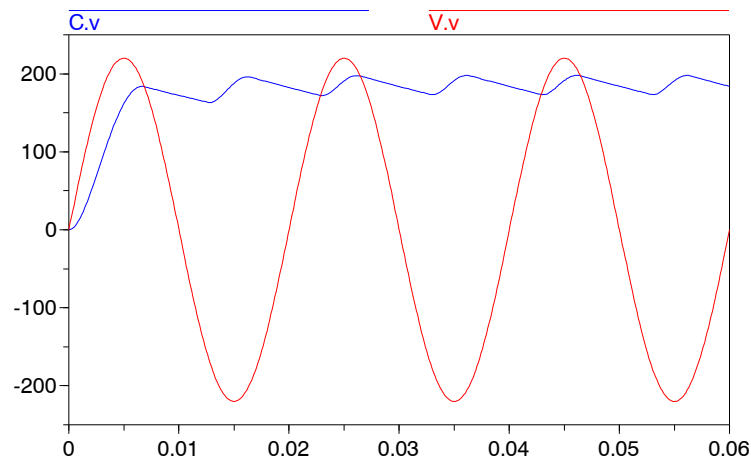
A more complicated example is shown in the next figure, consisting of a rectifier circuit with 4 idealized diode models:

Figure 10.3: Rectifier circuit with 4 ideal diode models.



The diodes are connected together, such that direct current is flowing through the load ( $R_2$ ,  $C_1$ ) driven by an AC voltage source. This circuit has essentially two operational modes: If the source current  $i_0$  is positive, diodes 1 and 4 are on (= switches are closed) and diodes 2 and 3 are off. If the source current is negative, diodes 2 and 3 are on, and diodes 1 and 4 are off. In both cases the current flows from left to right through the load ( $R_2$ ,  $C_1$ ), i.e., always in the same direction. A typical simulation result can be seen in the figure below, where the source voltage  $u_0$  and the voltage drop over the capacitance  $C$  are shown.

Figure 10.4: Simulation result of rectifier circuit with 4 ideal diode models.



Collecting the equations of all components and connections, and transforming this set of equations with the algorithms sketched above, results in the state space form:

$$\begin{aligned}
& \text{input} & : & t, u_c \\
& \text{output} & : & \dot{u}_c \\
& m_1 & = & s_1 < 0 \\
& m_2 & = & s_2 < 0 \\
& m_3 & = & s_3 < 0 \\
& m_4 & = & s_4 < 0 \\
& \left( \begin{array}{cccc} 0 & m_2 & 0 & m_4 \\ m_1 & -m_2 & m_3 & -m_4 \\ m_1/R_1 + (1-m_1) & -m_2/R_1 & m_3-1 & 0 \\ m_1-1 & m_2-1 & 1-m_3 & 1-m_4 \end{array} \right) \cdot \begin{pmatrix} s_1 \\ s_2 \\ s_3 \\ s_4 \end{pmatrix} & = & \begin{pmatrix} -u_c \\ 0 \\ u_0/R_1 \\ 0 \end{pmatrix} \\
& & & \dots
\end{aligned} \tag{10.1}$$

Note, that  $s_1, s_2, s_3, s_4$  are the curve parameters of the corresponding ideal diodes and  $m_1, m_2, m_3, m_4$  are the Boolean variables that characterize the off structures of these diodes. It is assumed that the value true of a Boolean variable is represented by “1” and the value false by “0”. This System has the structure explained above and can be solved with one of the sketched method.

The technique of parameterized curve descriptions was introduced in (Clauss et.al. 1995) and a series of related papers. However, no proposal was given how to implement such models in a numerically sound way. In Modelica, the solution method follows logically because the equation based system naturally leads to a system of mixed continuous/discrete equations which have to be solved at event instants (Otter et. al. 1999).

Alternative approaches to treat ideal switching elements are

- (a) by using variable structure equations which are controlled by state machines to describe the switching behavior, see, e.g., (Barton 1992, Elmqvist et. al. 1993, Mosterman and Biswas 1996), or
- (b) by using a complementarity formulation, see, e.g., (Lötstedt 1982, Pfeiffer and Glocker 1996, Schumacher and van der Schaft 1998).

The approach (a) has the disadvantage that the continuous part is described in a declarative way but not the part describing the switching behavior. As a result, algorithms with better convergence properties for the determination of consistent switching structure cannot be used. Furthermore, this involves a global iteration over all model equations whereas parameterized curve descriptions lead to local iterations over the equations of the involved elements. The approach (b) seems to be difficult to use in an object-oriented modeling language and seems to be applicable only in special cases (e.g., it does not seem possible to describe ideal thyristors).

Note, mixed systems of equations do not only occur if parameterized curve descriptions are used, but also in other cases, e.g., whenever an if-statement is part of an algebraic loop and the expression of the if-statement is a function of the unknown variables of the algebraic loop.

## 10.2 Coulomb Friction (not yet complete)

The simulation of components with ideal switch elements becomes difficult, if switching results in an index change of the DAE, i.e., if the number of states is changing. A typical example is Coulomb friction where this situation is present even in the simplest case. To concentrate on the essentials, first the simplified friction element in the next figure is discussed:

< ... >

The friction force “ $f$ ” acts between two surfaces, see right part of the figure, and is a linear function of the relative velocity “ $v$ ” between the friction surfaces when the surfaces are sliding relative to each other. When the relative velocity becomes zero, the two surfaces are stuck to each other and the friction force is no longer a function of “ $v$ ”. The element starts sliding again if the friction force becomes larger than the maximum static friction force “ $f_0$ ”. This element can also be described as a parameterized curve, as indicated in the left part of the figure, leading to the following equations:

Simplified Coulomb friction element

$$\mathbf{forward} = s > 1;$$

```

backward = s < -1;
v = if forward then s - 1 else
    if backward then s + 1 else 0;
f = if forward then f0+f1*(s-1) else
    if backward then -f0+f1*(s+1) else f0*s;

```

This model completely describes the simplified friction element in a declarative way. Unfortunately, currently it is not known how to transform such an element description automatically in a form which can be simulated. Let us analyze the difficulties by applying this model to the simple block on a rough surface shown in the right part of the figure above which is described by the following equation:

$$m \cdot \text{der}(v) = u - f$$

Note, that “m” is the mass of the block and “u(t)” is the given driving force. If the element is in its “forward sliding” mode, i.e.,  $s \geq 1$ , this model is described by:

$$\begin{aligned} m \cdot \text{der}(v) &= u - f \\ v &= s - 1 \\ f &= f_0 + f_1 \cdot (s-1) \end{aligned}$$

which can be easily transformed into state space form with “v” as the state. If the block becomes stuck, i.e.,  $-1 \leq s \leq 1$ , the equation “v=0” becomes active and therefore “v” can no longer be a state, i.e., an index change takes place. Besides the difficulty to handle the variable state change, there is a more serious problem: Assume that the block is stuck and that “s” becomes greater than one. Before the event occurs,  $s \leq 1$  and  $v = 0$ ; at the event instant  $s > 1$  because this relation is the event triggering condition. The element switches into the forward sliding mode where “v” is a state which is initialized with its last value “v=0”. Since “v” is a state, “s” is computed from “v” via “s := v+1”, resulting in “s=1”, i.e., the relation “s > 1” becomes **false** and the element switches back into the stuck mode. In other words, it is never possible to switch into the forward sliding mode. Taking numerical errors into account, the situation is even worse.

Figure: Friction characteristic at “v=0”

The key to the solution is the observation that “v=0” in the stuck mode and when forward sliding starts, but “der(v) > 0” when sliding starts and der(v) = 0 in the stuck mode, see above figure. Since the friction characteristic in the above figure at zero velocity is no functional relationship, again a parameterized curve description with a new curve parameter “s\_a” has to be used leading to the following equations (note: at zero velocity):

```

startFor = sa > 1;
startBack = sa < -1;
a = der(v);
a = if startFor then sa-1 else
    if startBack then sa+1 else 0;
f = if startFor then f0 else
    if startBack then -f0 else f0*sa;

```

At zero velocity, these equations and the equation of the block form again a mixed continuous/discrete set of equations which has to be solved at event instants, similarly as in the simple rectifier circuit discussed above. When switching from sliding to stuck mode, the velocity is small or zero. Since the derivative of the constraint equation  $\text{der}(v) = 0$  is fulfilled in the stuck mode, the velocity remains small even if  $v=0$  is not explicitly taken into account. By this well-know procedure, the velocity “v” remains a state in all switching configurations.

Consequently, “v” is small but may have any sign when switching from stuck to sliding mode; if the friction element starts to slide, say in the forward direction, one has to wait until the velocity is really positive, before switching to forward mode (note, that even for exact calculation without numerical errors a “waiting” phase is necessary, because “v=0” when sliding starts). Since “der(v) > 0”, this will occur after a small time period. This “waiting” procedure is most easily described by the state machine of the next figure.

Figure: Switching structure of friction element

Collecting all the pieces together, finally results in the following equations of a simple friction element:

```

// part of mixed system of equations
startFor = pre(mode) == Stuck and sa > 1;
startBack = pre(mode) == Stuck and sa < -1;
a = der(v);
a = if pre(mode) == Forward or startFor then sa - 1 else
    if pre(mode) == Backward or startBack then sa + 1 else 0;
f = if pre(mode) == Forward or startFor then f0 + f1*v else
    if pre(mode) == Backward or startBack then -f0 + f1*v else f0*sa;

// state machine to determine configuration
mode = if (pre(mode) == Forward or startFor) and v>0 then Forward
    else
    if (pre(mode) == Backward or startBack) and v<0 then Backward
    else Stuck;

```

Note, that the equations within the mixed system are evaluated based on the value of “mode” when the event occurred, i.e., on `pre(mode)`. After the new sliding or stuck mode is determined by the solution of a mixed set of continuous/discrete equations, the new value of mode is computed by the last equation which is just a direct mapping of the state machine of the figure above.

Figure: Coulomb friction characteristic

The described procedure can be easily applied also for the more general friction element in the figure above where the sliding friction force has a nonlinear characteristic and there is a jump in the friction force from  $f_{\max}$  to  $f_0$  when sliding starts. The element equations of the simple friction element need only two changes:

- (1) the linear equation of the sliding friction force has to be replaced by an appropriate nonlinear relationship (usually realized by interpolation in a table) and
- (2) the sliding conditions have to be modified to:

```

startFor = pre(mode) == Stuck and
    (sa > peak or pre(startFor) and sa > 1);
startBack = pre(mode) == Stuck and
    (sa < -peak or pre(startBack) and sa < -1);

```

where “ $\text{peak} = f_{\max}/f_0 \geq 1$ ”. All other equations are identical to the simple friction element. It is straightforward to adapt this general friction element, e.g., to model clutches or brakes.

The current implementation of friction models in Modelica allows to solved quite complex problem of coupled friction elements, e.g., models of an automatic gearbox. However, the description is still too complicated and it would desirable to find a simpler solution.

Additional Material:

Mass with friction and pulling force  $f$ :

$$m \cdot a = f - f_R \quad (10.2)$$

Consider the two cases

$a = s - s_0$  (forward sliding at  $v = 0$ ):

$$\begin{aligned} m \cdot a &= f_{R_{\max}} - f_R \\ a &= s - s_0 \end{aligned} \quad (10.3)$$

Solving for  $s$  results in:

$$\begin{aligned}
s &= s_0 + \frac{1}{m} \cdot (f_{Rmax} - f_R) \geq f_{Rmax} \\
\rightarrow s_0 &\geq f_{Rmax} - \frac{1}{m} (f_{Rmax} - f_R) \\
s_0 &\geq f_{Rmax} \quad \text{for } m \rightarrow \infty
\end{aligned} \tag{10.4}$$

On the other hand, if during the iteration it is temporarily assumed that the model is sliding backward, i.e.,  $a = s + s_0$ , we have:

$$\begin{aligned}
m \cdot a &= f_{Rmax} - f_R \\
a &= s + s_0
\end{aligned} \tag{10.5}$$

Solving for  $s$  results in:

$$\begin{aligned}
s &= -s_0 + \frac{1}{m} \cdot (f_{Rmax} - f_R) \leq -f_{Rmax} \\
\rightarrow s_0 &\geq f_{Rmax} + \frac{1}{m} (f_{Rmax} - f_R) \\
s_0 &\geq f_{Rmax} \quad \text{for } m \rightarrow \infty
\end{aligned} \tag{10.6}$$

This means that a contradiction is present, because in the first case the result is that forward sliding is true and in the second case the result is that backward sliding is true. This means that the mixed system of equations has two solutions.

The exact formulation of the friction characteristic at zero velocity should be uncritical:

$$\begin{aligned}
a &= \text{if } s \geq f_{fmax} \text{ then } s - f_{fmax} \text{ else if } s \leq -f_{fmax} \text{ then } s + f_{fmax} \text{ else } 0 \\
f_f &= \text{if } s \geq f_{fmax} \text{ then } f_{fmax} \text{ else if } s \leq -f_{fmax} \text{ then } -f_{fmax} \text{ else } s
\end{aligned} \tag{10.7}$$

## Chapter 11 Re-initialization of Continuous Variables

At an event instant, non-discrete Real variables might change discontinuously, e.g., due to an impact. To define this with a modeling language is a very difficult issue and we are not aware of a really satisfactory solution (neither in Modelica nor in another generic modeling system). The reason is that a differential-algebraic equation system (DAE) of the form

$$\mathbf{0} = \mathbf{f}(\dot{\mathbf{x}}, \mathbf{x}, \mathbf{y}, t) \quad (11.1)$$

where  $\mathbf{x}$  are variables that appear differentiated and  $\mathbf{y}$  are pure algebraic variables, must be fulfilled all the time, also at an event instant. Assume that this DAE is fulfilled at the current event instant and that one or more elements of  $\mathbf{x}$  and/or  $\mathbf{y}$  are changed discontinuously, then:

1. Other variables must be changed discontinuously too, in order that the DAE is still fulfilled. For example, if there is an equation “ $0 = y_1 + y_2$ ” and “ $y_2$ ” is changed discontinuously, then “ $y_1$ ” must be changed discontinuously too. If  $m < nx$  variables are changed discontinuously at the same event instant, there is usually no unique solution to compute all the other variables, so that the DAE is still fulfilled.
2. If the variable that is changed discontinuously appears also differentiated in the DAE (e.g. “ $v$ ” is changed discontinuously and “ $\text{der}(v)$ ” appears in the equations), then a Dirac impulse occurs which means that other continuous variables need also to be changed discontinuously due to this Dirac pulse.

In section 11.1 the current (unsatisfactory) solution in Modelica to this problem is presented together with some examples. In section 11.2 an experimental feature is described that was developed by Dynasim in the EU-project REALSIM and that is considerably better as the current Modelica solution, but there are still unresolved issues and further research is needed.

### 11.1 The `reinit()` operator

In Modelica it is possible to change non-discrete Real state variables discontinuously at an event instant with the `reinit(...)` built-in operator that is defined as:

**reinit**( $x, \text{expr}$ ) In the body of a **when**-clause, reinitializes  $x$  with  $\text{expr}$  at an event instant.  $x$  is a Real variable (resp. an array of Real variables, in which case vectorization applies) that must be selected as a state (resp., states) at least when the enclosing **when**-clause becomes active.  $\text{expr}$  needs to be type-compatible with  $x$ . The **reinit** operator can only be applied once for the same variable (resp. array of variables). It can only be applied in the body of a when clause.

The `reinit(...)` operator is processed by assuming that the previously known state variable “ $x$ ” referenced in the `reinit(...)` operator is treated (temporarily) as unknown with the equation “ $x = \text{expr}$ ”. After sorting (BLT partitioning), the `reinit`-clause is present so that all references to “ $x$ ” are “after” the “`reinit(...)`” equation and all variables referenced in “ $\text{expr}$ ” are computed “before” this equation. Note, this gives some restrictions what can be referenced in “ $\text{expr}$ ”, e.g., no variable that depends explicitly or implicitly on “ $x$ ”. Additionally, code must be introduced to inform the integrator that the corresponding state has changed at an event instant. Since state variables “ $x$ ” are assumed to be known during equation sorting, they can be changed at an event instant (in the way described above) and the DAE is still fulfilled. For other variables this property does not hold and therefore only this restricted version of the `reinit(...)` operator is supported in Modelica.

From a user point of view, this approach is problematic because it is not known in advance whether a “potential state” variable “ $x$ ” is used as “real state” from the simulation system. However, the

“**reinit**(..)” operator can only be applied on the “real states” that are selected from the modeling system during code generation. Therefore, the only way how this can work is that the Modelica translator tries to use variables “x” appearing in a “**reinit**(..)” clause as “real states”. So, this is equivalent to set attribute `stateSelect = StateSelect.always` on variable “x”. If this is not possible, a translation error will occur.

The **reinit**(..) operator is a very dangerous construct and a modeler can easily construct models that are physically not correct (but the Modelica translator cannot detect this error). Therefore, this operator should be removed from the Modelica language once a better alternative is available.

The usage of the **reinit**(..) operator is demonstrated in the next example that describes a ball (a mass point) that is vertically falling down to the ground. The ball starts at height  $h = 1$  m. When it reaches the ground at  $h = 0$  m, an impact occurs. At the impact point, the velocity of the ball changes discontinuously according to Newtons impact hypothesis:

$$v(t+\epsilon) = -e \cdot v(t-\epsilon), \quad \epsilon \rightarrow 0 \quad (11.2)$$

where “e” is the impact coefficient ( $0 \leq e \leq 1$ ):

```

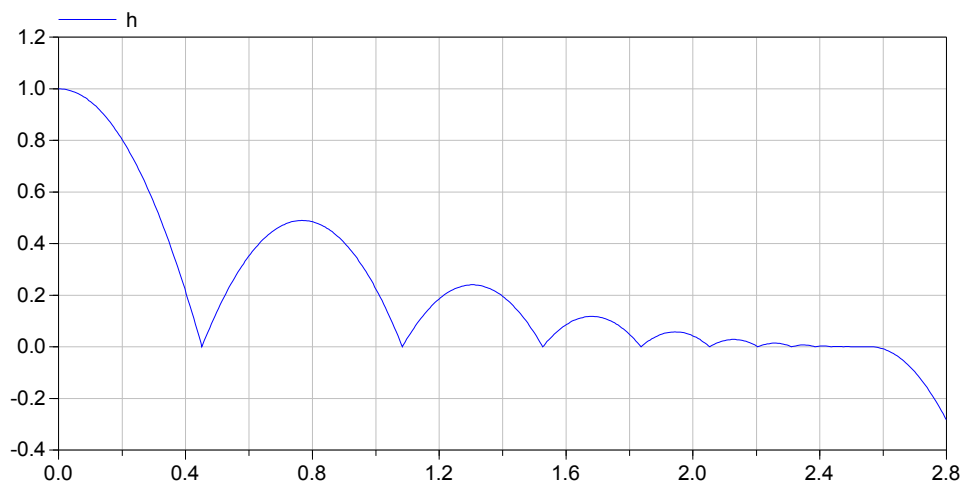
model BouncingBall1 // no proper description of bouncing ball
  import SI=Modelica.SIunits;
  parameter Real e=0.7;
  parameter SI.Acceleration g=9.81;
  SI.Height h(start=1, fixed=true);
  SI.Velocity v;
equation
  der(h) = v;
  der(v) = -g;

  when h <= 0 then
    reinit(v, -e*pre(v));
  end when;
end BouncingBall1;

```

The impact is simply modeled by a **when**-clause that triggers an event when the height becomes less than zero. Then the velocity “v” of the ball is re-initialized with “ $-e \cdot \text{pre}(v)$ ” where **pre**(v) is the velocity of the ball just before the event  $h \leq 0$  occurred, i.e., the velocity with which the ball touches the ground. The simulation result of the above model is shown in the next figure:

Figure 11.1: Simulations results of BouncingBall1 (the solution is not satisfactory)



After about 2.6 s, the model does no longer behave as expected, because the ball falls through the ground. The reason is that  $h$  is smaller as zero when the event occurs. When the re-bounce velocity  $-e \cdot \text{pre}(v)$  is not large enough, the ball will not fly above  $h=0$  and since it will then remain permanently below  $h = 0$ , the **when**-clause does not trigger a new event.



The reason for this behavior is a *wrong model*, because the constraint of the ground is not properly described. A reasonable solution requires describing the surfaces and the material properties of the two parts that collide. From this information the impact equations should be automatically derived and handled properly. This is currently far away from what can be achieved in Modelica and only certain “simple” forms of “hard” impacts (e.g., between two parts with simple surface descriptions) can be defined in a reasonable way. In the simple case of the bouncing ball, one solution is the following:

```

model BouncingBall2
  import SI=Modelica.SIunits;
  parameter Real e=0.7;
  parameter SI.Acceleration g=9.81;
  SI.Height    h(start=1, fixed=true);
  SI.Velocity  v;
  Boolean      flying;
equation
  der(h) = v;
  der(v) = if flying then -g else 0;
  flying = not (h <= 0 and v <= 0);

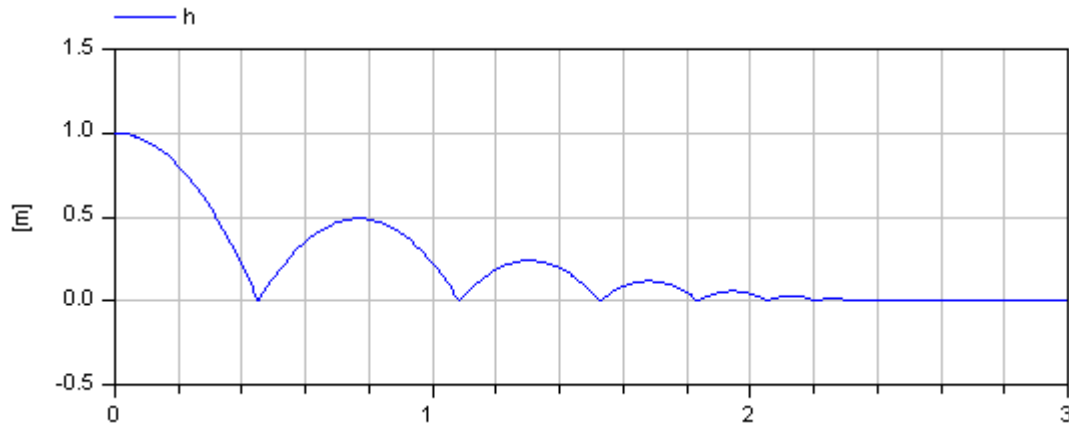
  when h <= 0 then
    reinit(v, -e*pre(v));
  end when;
end BouncingBall2;

```

The basic idea is always to switch between different model structures when two colliding parts remain on each other (here: when the ball remains on the ground). In the above model, the two model structures are defined with the Boolean variable “flying”. When “flying = **true**”, the ball is flying and is described by the equation “**der**(v) = -g”. When the ball remains on the ground, “flying = **false**” and the model equations are changed to “h = 0”. However, this would require to dynamically change the number of state variables during simulation, since “flying = **false**” would have no states any more. The current Modelica tools cannot handle such a situation and therefore the approximation is used that “**der**(v) = 0” instead. Since h is close to zero when the model structure is changed, it will remain close to zero when the second derivative **der**(v) is zero and therefore this solution is a reasonable approximation.

It remains to discuss when to switch between the defined model structures. In the general case, when impact can occur between more than two parts, the switching can be described by a finite state machine (but the number of states of the finite state machine grows quickly with the number of potential impacts, so this is not a “good” solution, but the best what can be currently implemented in Modelica). In the simple case of the bouncing ball, we have only two discrete states (flying = **false** and flying = **true**) and one transition between these two states: when the height is negative and the ball velocity is negative, then the ball would fly through the ground and therefore at this point the model structure is changed to “flying = **false**”. Therefore, it is easy to just add a Boolean equation for “flying” that describes this behavior. The result of a simulation is displayed in the next figure which shows the expected behavior.

Figure 11.2: Simulation results of BouncingBall2 (satisfactory result).



The above model has the slight problem that many events occur at the end before the ball remains on the ground. This effect is called “chattering” and could be solved using the method from “Fillipov”. Mathematically, it is well known that this problem has a different solution: An infinite number of events occur but there is a time barrier that the ball cannot cross. In reality such a behavior, of course, does not occur. The model is too idealized. In reality, the coefficient of restitution is not constant but depends on the impact velocity: If the impact velocity is becoming smaller, the coefficient of restitution also becomes smaller.

## 11.2 Experimental operators to describe impulsive behavior

In the EU project Realsim (Real-time Simulation of Multi-physics systems), Dynasim has experimented with alternatives to re-initialize DAEs. From a user point of view two built-in operators are introduced in Dymola<sup>14</sup>:

Table 11.1: Experimental operators to describe impulsive behavior.

<b>DiracImpulse</b> (condition)	Returns an infinitely large value at the time instant when “condition” becomes <b>true</b> . Otherwise, the return value is zero. The integral over this signal is 1.
<b>PositiveImpulse</b> (condition, v, expr)	Returns an infinitely positive large value at the time instant when “condition” becomes <b>true</b> . Otherwise, the return value is zero. The integral over this signal is defined, such that at the same time instant, variable “v” is re-initialized to “expr” when the <b>PositiveImpulse</b> (..) value is applied.

The major differences to the **reinit**(..) operator are that the two operators can be used to “re-initialize” any variable, not only states, and that the discontinuous change of one variable may result in the discontinuous change of other variables due to the propagation of the impulse. So this concept is more powerful and more useful as the **reinit**(..) method. On the other hand, there are several unresolved problems, e.g., a mathematically and physically well defined description of several impulses that occur at the same time instant, handling of the contact constraint in a convenient way, so that the colliding parts remain in contact to each other (e.g., that the bouncing ball stays on the ground), handling of the probably more realistic contact hypothesis of Poisson, where not the velocity after and before the impulse is related to each other with the impact coefficient, but the impulse of the decompression phase with respect to the impulse of the compression phase, see, e.g., (Pfeiffer, Glocker 1996).

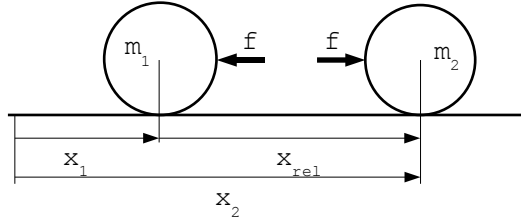
Conceptually, the **PositiveImpulse**(..) operator is mapped to the following equation (this mapping and the following example is from document hybrid2.pdf of H. Elmqvist and M. Otter, from the minutes of the 21st Modelica design meeting in Detroit):

<sup>14</sup>The two operators are available in Dymola, but require an activation flag in order that they can be used. Contact Dynasim (info@dynasim.se) for more information.

$0 = \text{if edge}(\text{condition}) \text{ then } v - \text{expr} \text{ else } y;$

This equation states that  $y$  has an infinite large value at the time instant when the “condition” becomes **true**, such that the (unknown) time-integral of  $y$  is determined so that “ $v = \text{expr}$ ” after the impulse. At other time instants,  $y = 0$ . The “**PositiveImpulse**(...)” operator is demonstrated at the following example of two colliding mass points:

Figure 11.3: Two colliding mass points.



that are described by the following equations:

```

der(x1) = v1
der(x2) = v2
m1*der(v1) = -f
m2*der(v2) = f
v_rel = v2 - v1
f = PositiveImpulse(x2 < x1, v_rel, -e*pre(v_rel));

```

The solution is performed in the following way. In principal this is just standard impact mechanics, see e.g. (Pfeiffer, Glocker 1996). However, formulated in a more general framework based on Dirac function algebra and the Pantelides algorithm.

Case 1: **edge**(x2 < x1) = **false**

```

der(x1) = v1
der(x2) = v2
m1*der(v1) = -f
m2*der(v2) = f
v_rel = v2 - v1
f = 0;

```

input: x1, x2, v1, v2



```

der(x1) := v1
der(x2) := v2
der(v1) := 0
der(v2) := 0
v_rel := v2 - v1
f := 0

```

Case 2: **edge**(x2 < x1) = **true**

```

der(x1) = v1
der(x2) = v2
m1*der(v1) = -f
m2*der(v2) = f
v_rel = v2 - v1
v_rel = -e*pre(v_rel)

```

These are 6 equations in 6 unknowns. Since the last two equations are two equations for one unknown, the Pantelides algorithm and the dummy derivative method have to be applied which requires differentiating the last two equations. However,  $v_{rel}$  is a discontinuous function. Differentiating a discontinuous function results in a Dirac impulse  $\delta(t-t_0)$  at time  $t_0$ . We use the well known formula that differentiating  $v(t)$  with discontinuity at  $t_0$ , left limit  $v_1$  and right limit  $v_2$  results in the following value of the derivative at  $t_0$ :

$$\dot{v}(t_0) = (v_2 - v_1) \cdot \delta(t_0) \quad (11.3)$$

Therefore, differentiating the two equations above results in:

```

der(v_rel) = der(v2) - der(v1);
der(v_rel) = -(1+e)*pre(v_rel)*dirac(t0);

```

Applying the dummy derivative method results in a set of 8 equations in 8 unknowns at the time instant when `edge(x2 > x1)` is true:

```

input: x1, x2, v1, pre(v_rel)

dder(v_rel) := -(1+e)*pre(v_rel)*dirac(t0);
der(v1)      := -m2*dder(v_rel)/(m1+m2);
dder(v2)     := dder(v_rel) + der(v1);
f             := m2*dder(v2);
v_rel         := -e*pre(v_rel);
v2            := v_rel + v1;
der(x1)       := v1;
der(x2)       := v2;

```

where **dder**(v) is the dummy derivative of “v” (i.e., **dder**(v) is treated as an algebraic variable). All the equations are integrated from  $t_0 - \varepsilon$  to  $t_0 + \varepsilon$  in order to get rid of the Dirac impulse. This results in equations for the new initial values of all variables. For the integration, the following standard formula is used:

$$\int_{t_0 - \varepsilon}^{t_0 + \varepsilon} v(t) \cdot \delta(t_0) \cdot dt = v(t_0) \quad (11.4)$$

which holds under the assumption that  $v(t)$  is continuous in the integration interval. The final result is:

```

input: x1, x2, v1, pre(v_rel)

v_rel_dummy := -(1+e)*pre(v_rel);
v1           := pre(v1) - m2*v_rel_dummy/(m1+m2);
v_rel        := -e*pre(v_rel);
v2           := v_rel + v1;
x1           := pre(x1);
x2           := pre(x2)

```

These are the correct equations for the impact of two mass points.

## Part C Simulation

The following chapters provide an overview about the symbolic and numeric algorithms to perform a simulation of a Modelica model, as well as what to do when a simulation fails. Algorithms are sketched, in order that a Modelica modeler understands how a Modelica translator and simulator operates. This knowledge is useful to understand error messages and to resolve problems with a Modelica translator or when a simulation fails. It is not the intention to explain the algorithms in such detail that they can be used for an implementation in a Modelica tool.

## Chapter 12 Symbolic Transformation Algorithms

The Modelica Language Specification (Modelica 2007) defines how a Modelica model shall be mapped into a mathematical description as a mixed system of differential-algebraic equations (DAE) and discrete equations with Real, Integer and Boolean variables as unknowns. This is also described in chapter 8 in this document. There are no general-purpose solvers for such problems. There are numerical DAE solvers, which could be used to solve the continuous part. However, if a DAE solver is used directly to solve the original model equations, the simulation will be very slow and initialization might be not possible for singular systems (see section 12.4). It is therefore assumed that Modelica models are first symbolically transformed into a form that is better suited for numerical solvers. In this chapter, the transformation techniques are sketched that have been initially designed for the Dymola modeling language (Elmqvist 1978), further developed in Omsim for the Omola language (Pantelides 1988, Mattsson and Söderlind 1993) and further improved in the commercial Modelica simulation environment Dymola (Mattsson et. al. 2000, Dymola 2008):

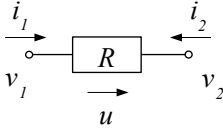
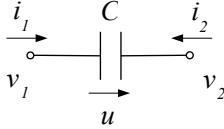
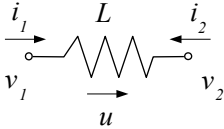
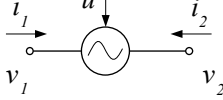

Dymola converts the differential-algebraic system of equations symbolically to ordinary differential equations in state-space form, i.e. solves for the derivatives. Efficient graph-theoretical algorithms are used to determine which variables to solve for in each equation and to find minimal systems of equations to be solved simultaneously (algebraic loops). The equations are then, if possible, solved symbolically or code for efficient numeric solution is generated.

Other Modelica simulation environments might work differently, although the basic algorithms, such as BLT (section 12.2) and the Pantelides algorithm (section 12.4) will most likely be used.

### 12.1 Transformation to State Space Form

In Chapter 8 it is sketched how a Modelica model is mapped to a set of differential, algebraic and discrete equations. We will demonstrate this mapping on the basis of a simple electrical circuit and show how this basic set of equations is further processed. In a first step, a small library of electrical components is modeled using the connector definition of an electrical pin as explained in section 2.2. The result is shown in the next table:

Table 12.1: Equations of basic electric circuit components

<b>Resistor</b>		$0 = i_1 + i_2$ $u = v_1 - v_2$ $u = R \cdot i_1$
<b>Capacitor</b>		$0 = i_1 + i_2$ $u = v_1 - v_2$ $C \cdot \frac{du}{dt} = i_1$
<b>Inductor</b>		$0 = i_1 + i_2$ $u = v_1 - v_2$ $L \cdot \frac{di_1}{dt} = u$
<b>Voltage source</b>		$0 = i_1 + i_2$ $u = v_1 - v_2$ $u = A \cdot \sin(\omega \cdot t)$
<b>Ground</b>		$v = 0$

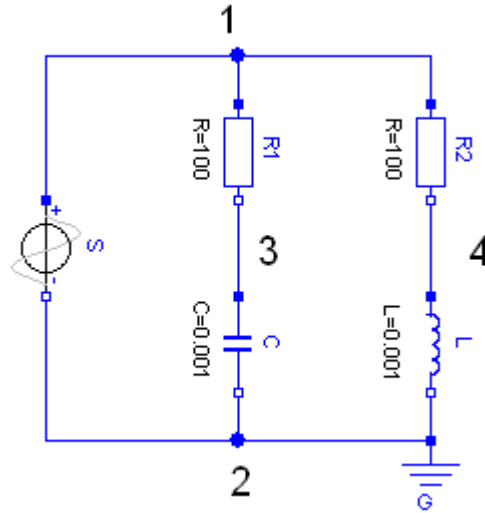
In order to simplify the notation in this example, a connector variable such as “pin\_p.i” is defined as “ $i_l$ ”. Otherwise, the same equations are used as for the electrical components in the Modelica.Electrical.Analog library. For example, a capacitor is defined by 3 equations:

$$\begin{aligned}
 0 &= i_1 + i_2 \\
 u &= v_1 - v_2 \\
 C \cdot \frac{du}{dt} &= i_1
 \end{aligned} \tag{12.1}$$

The first equation states that the current flowing into the capacitor at the left connector is identical to the current flowing out of the right connector ( $i_1$  and  $i_2$  are positive, if the current is flowing into the component). The second equation computes the voltage drop as difference of the electrical potentials  $v_1$  and  $v_2$  at the left and at the right connector. Finally, the third equation defines that the time derivative of the voltage drop is proportional to the current entering the left connector.

We will now use these elements to build the following simple electrical circuit:

Figure 12.1: Simple electrical circuit to demonstrate code generation.



This circuit has been constructed by dragging the corresponding elements into the diagram layer of a new model and connecting them appropriately together. In the text layer of Dymola, the Modelica definition of this model is automatically available (for clarity, the graphical annotations that describe the visual appearance in the diagram layer are removed below; pins with a “filled square” have the instance name “p” and pins with a “non-filled square” have the instance name “n” in the Modelica.Electrical.Analog library):

```

model SimpleCircuit
  Modelica.Electrical.Analog.Sources.SineVoltage S (V=220, freqHz=50);
  Modelica.Electrical.Analog.Basic.Resistor      R1 (R=100);
  Modelica.Electrical.Analog.Basic.Resistor      R2 (R=100);
  Modelica.Electrical.Analog.Basic.Capacitor     C (C=0.001);
  Modelica.Electrical.Analog.Basic.Inductor      L (L=0.001);
  Modelica.Electrical.Analog.Basic.Ground G;
equation
  connect (S.p, R1.p);
  connect (S.p, R2.p);
  connect (L.n, G.p);
  connect (G.p, C.n);
  connect (G.p, S.n);
  connect (R1.n, C.p);
  connect (R2.n, L.p);
end SimpleCircuit;

```

When selecting “Simulation / Simulate” in the toolbar of Dymola, the above circuit is translated into C-code, the C-code is compiled into object code, the object code is loaded into the simulation environment and then the circuit is simulated and the result can be visualized in the plotting environment of Dymola. It is now analyzed how the translation process is performed:

The first step is to map the Modelica model above into a set of equations by

- replacing component declarations with the component equations from the small library above and
- replacing all “**connect** ( . . )” statements with corresponding equations (see section 2.2).

The result is shown in the next table:

Table 12.2: Basic equations of SimpleCircuit model.

R1	$0 = R1.i_1 + R1.i_2$	R2	$0 = R2.i_1 + R2.i_2$
	$R1.u = R1.v_1 - R1.v_2$		$R2.u = R2.v_1 - R2.v_2$
	$R1.u = R1.R \cdot R1.i_1$		$R2.u = R2.R \cdot R2.i_1$



C	$0 = C.i_1 + C.i_2$ $C.u = C.v_1 - C.v_2$ $C.i_1 = C.C \cdot \frac{dC.u}{dt}$	L	$0 = L.i_1 + L.i_2$ $L.u = L.v_1 - L.v_2$ $L.u = L.L \cdot \frac{dL.i_1}{dt}$
A	$0 = S.i_1 + S.i_2$ $S.u = S.v_1 - S.v_2$ $S.u = S.i_1 + R1.i_1 + R2.i_1$	G	$0 = G.v$
1	$S.v_1 = R1.v_1$ $S.v_2 = R2.v_1$ $0 = S.i_1 + R1.i_1 + R2.i_1$	2	$G.v = C.v_2$ $G.v = L.v_2$ $G.v = S.v_2$ $0 = S.i_2 + C.i_2 + L.i_2 + G.i$
3	$R1.v_2 = C.v_1$ $0 = R1.i_2 + C.i_1$	4	$R2.v_1 = L.v_1$ $0 = R2.i_2 + L.i_1$

The result is a set of 27 equations that describe the electrical circuit above in a unique and declarative form, independently which type of analysis is performed on the circuit.

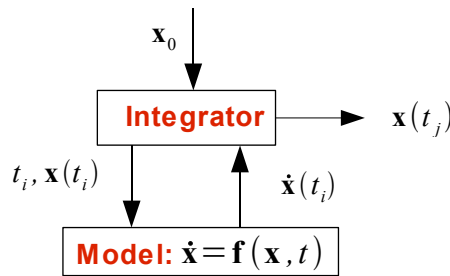
Our goal is to transform this set of equations into a set of ordinary differential equations in state space form, or shortly also called “state space form” (abbreviated as ODE):

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t), t), \quad \mathbf{x}(t=t_0) = \mathbf{x}_0, \quad t \in \mathbb{R}, \quad \mathbf{x} \in \mathbb{R}^{n_x} \quad (12.2)$$

The variables  $\mathbf{x}$  are called (continuous) states of the system. The reason is that a lot of efficient and reliable numerical integration algorithms are available to solve this initial value problem. Especially, most algorithms for real-time simulation require an ODE form. A short introduction and an overview of these algorithms are provided in 13.1. Another reason is that it is then possible to import the Modelica model also in other modeling and simulation environments (e.g., Simulink from MathWorks) because these environments provide interfaces to import ODE descriptions of models (but usually not more general descriptions, such as a DAE form).

Numerical integration algorithms for ODEs need a function that computes the state derivatives  $\dot{\mathbf{x}}(t_i)$  of the desired model at time instant  $t_i$ , given  $t_i$  and the states  $\mathbf{x}(t_i)$  at this time instant. This basic structure is shown in the next figure:

Figure 12.2: Basic structure of integration algorithms.



From the integrator point of view, the initial states  $\mathbf{x}_0$  are provided as inputs and the integration algorithm computes the states  $\mathbf{x}(t_j)$  at desired time instants  $t_j$  in the future. This means that the states  $\mathbf{x}$  are the unknown variables and they are computed from the integrator.

From the model function point of view, time instants  $t_i$  and the states  $\mathbf{x}(t_i)$  at this time instant are inputs to the model function. This means that the states  $\mathbf{x}$  are known variables! The Model function computes the state derivatives  $\dot{\mathbf{x}}(t_i)$  at this time instant.

To summarize: Depending on the view point, the states  $\mathbf{x}$  are either known or unknown variables. The main goal in this chapter is to explain how the model function can be automatically generated from a Modelica model. For this reason, in the rest of this chapter we take the view point that the states  $\mathbf{x}$  are known variables (= provided from the integrator).

The set of equations of the simple electrical circuit from above is mathematically described as a set of differential algebraic equations (abbreviated as DAE):

$$\mathbf{0} = \mathbf{f}(\dot{\mathbf{x}}(t), \mathbf{x}(t), \mathbf{y}(t), \mathbf{u}(t), \mathbf{p}, t), \quad t \in \mathbb{R}, \mathbf{x} \in \mathbb{R}^{nx}, \mathbf{y} \in \mathbb{R}^{ny}, \mathbf{u} \in \mathbb{R}^{nu}, \mathbf{p} \in \mathbb{R}^{np}, \mathbf{f} \in \mathbb{R}^{nx+ny} \quad (12.3)$$

where  $t$  is time,  $\mathbf{p}$  are known constants or parameters,  $\mathbf{u}(t)$  are known input signals,  $\mathbf{x}(t)$  are variables that appear differentiated in the model equations (i.e., the **der**(...) operator is applied on them) and  $\mathbf{y}(t)$  are all other unknown variables. Variables  $\mathbf{x}$  are called “*potential states*” because the states in the ODE form are either the full vector  $\mathbf{x}$  of the DAE form above or a subset of it. Variables  $\mathbf{y}$  are also called algebraic variables, because no derivatives of these variables appear in the model equations.

In order to simplify notation, the explicit dependency of the model equations from constants, parameters and input signals is usually not shown in the DAE form, because these are just known variables that do not pose any problems. Formally, these variables are treated as a special case of the explicit dependency of time  $t$  of the DAE form. As a result, the following shortened DAE form will be utilized in the sequel:

$$\mathbf{0} = \mathbf{f}(\dot{\mathbf{x}}(t), \mathbf{x}(t), \mathbf{y}(t), t), \quad t \in \mathbb{R}, \mathbf{x} \in \mathbb{R}^{nx}, \mathbf{y} \in \mathbb{R}^{ny}, \mathbf{f} \in \mathbb{R}^{nx+ny} \quad (12.4)$$

The primary goal is to transform this DAE into the ODE form:

$$\begin{pmatrix} \dot{\mathbf{x}}(t) \\ \mathbf{y}(t) \end{pmatrix} = \mathbf{f}_2(\mathbf{x}(t), t) \quad (12.5)$$

i.e., computing  $\dot{\mathbf{x}}$  and  $\mathbf{y}$  from the potential states  $\mathbf{x}$  and the actual time instant  $t$ . For an ODE integration algorithm it is then sufficient to just pass  $\dot{\mathbf{x}}$  to the integrator and to hide the algebraic variables  $\mathbf{y}$  from the integrator. The transformation from DAE to ODE form therefore requires solving a non-linear system of equations

$$\mathbf{0} = \mathbf{f}(\dot{\mathbf{x}}, \mathbf{y}) \quad (12.6)$$

for the unknowns  $\dot{\mathbf{x}}$  and  $\mathbf{y}$ . This is not as difficult as it looks like, because balance equations in physics are linear in the derivatives  $\dot{\mathbf{x}}$  and all connection equations are simple linear equations. This means that a large portion of  $\mathbf{f}(\cdot)$  is usually linear in the unknown variables and not non-linear which will considerably simplifies the transformation to ODE form.

In order that a unique solution of the non-linear equation exists, the Jacobian  $\mathbf{J}$  of this equation with respect to the unknowns must be regular. For the moment we will only treat models where this basic requirement is fulfilled:

$$\mathbf{J} = \left( \frac{\partial \mathbf{f}}{\partial \dot{\mathbf{x}}} : \frac{\partial \mathbf{f}}{\partial \mathbf{y}} \right) \text{ is regular everywhere} \quad (12.7)$$

In such a case, all “potential states”  $\mathbf{x}$  of the DAE are “states” of the ODE form. Models with a singular Jacobian are treated in section 12.4. Before transforming to ODE form, the “states” of the ODE form must be identified from the set of variables used in a Modelica model, because these variables are treated as “known” during the transformation. The “states” are simply all variables on which the “**der**(...)” operator is applied in the Modelica model.

In the simple electrical circuit example above, the variables  $C.u$  and  $L.i_1$  are used with the **der**(...) operator and therefore these variables are assumed to be known. Additionally, all constants and parameters, such as  $R1.R$  or  $R2.R$ , are known variables. All other, unknown, variables are summarized below:

Table 12.3: The 27 unknowns of the SimpleCircuit model.

$R1.i_1$	$R1.i_2$	$R1.v_1$	$R1.v_2$	$R1.u$
$R2.i_1$	$R2.i_2$	$R2.v_1$	$R2.v_2$	$R2.u$
$C.i_1$	$C.i_2$	$C.v_1$	$C.v_2$	$\frac{dC.u}{dt}$
$\frac{dL.i_1}{dt}$	$L.i_2$	$L.v_1$	$L.v_2$	$L.u$
$S.i_1$	$S.i_2$	$S.v_1$	$S.v_2$	$S.u$

$G.i$	$G.v$			
-------	-------	--	--	--

Since we have 27 unknowns and 27 equations it is possible to solve the equations for the unknowns. In the following sections the algorithms are sketched to achieve the following result:

**input** variables:  $\mathbf{x} = \{C.u, L.i_1\}, t$ , all parameters ( $R1.R, \dots$ ),  
onlyStates (= **true** : return only states,  
= **false**: return all variables)  
**output** variables:  $d\mathbf{x}/dt = \{dC.u/dt, dL.i_1/dt\}$ ,  
 $\mathbf{y} = \{\text{all other unknown variables}\}$  **if not** onlyStates

**algorithm**

```

R2.u := R2.R · L.i1
R1.v1 := S.A · sin(S.ω · t)
L.u := R1.v1 − R2.u
R1.u := R1.v1 − C.u
C.i1 := R1.u / R1.R
dL.i1 / dt := L.u / L.L
dC.u / dt := C.i1 / C.C

```

```

if not onlyStates then // compute all other 20 variables

```

```

    S.i2 := −C.i1 − L.i1
    G.i := S.i2 + C.i1

```

```

end if

```

From the 27 equations, a subset of 7 equations is sufficient to compute the state derivatives  $\dot{\mathbf{x}}$  in a recursive evaluation order. Whenever this model function is called, these 7 equations have to be evaluated. However, the other 20 equations are not needed for the integration algorithm. Therefore, they are only evaluated when variables shall be stored on file at a communication grid. Dymola performs such type of equation partitioning in order to improve efficiency.

One might wonder why the 7 equations are not inserted into each other in order to arrive at only 2 equations for the state derivatives? The reason is that the number of operations usually increases dramatically when expressions are substituted and therefore the efficiency of the model evaluation is considerably reduced. This can be already seen at the simple example above: There are 2 multiplications and one sin-function evaluation needed in order to compute variable  $R1.v_1$ . This variable is present at two places. It would therefore be possible to remove the assignment statement to compute  $R1.v_1$  and replace this variable at every occurrence by its definition equation. However, this would mean that the number of operations increases, because then 4 multiplications and 2 sin-function evaluations are necessary.

The symbolic transformation algorithms to be discussed are suited for very large systems of equations: E.g., in Dymola it is possible to process more than 100000 equations in a few seconds on a PC. One reason is that the effort for the symbolic transformation grows usually linearly with the number of equations.

In Dymola, the default integrator to numerically solve the ODE form of the symbolically processed equations is DASSL. DASSL is an integrator that is designed to solve DAEs. This can be easily accomplished by just re-writing the ODE into the DAE form:

$$\mathbf{0} = \dot{\mathbf{x}}(t) - \mathbf{f}(\mathbf{x}(t), t) \quad (12.8)$$

The numerical solution of (12.8) with DASSL is usually very reliable and efficient due to the symbolic pre-processing. DASSL would be not suited for many Modelica models, if it would be directly applied to the original equations, before the symbolic pre-processing takes place. The reasons for this behavior will be discussed in the next sections.

## 12.2 Sorting (BLT)

In this section the most fundamental algorithm for Modelica models called “Block Lower Triangular” transformation will be discussed (this algorithm is abbreviated as BLT). Before applying this algorithm, a trivial simplification is first carried out by substituting alias variables of the form “ $a = b$ ” or “ $a = -b$ ”. This means, equations of the form “ $a = b$ ”, “ $a = -b$ ”, “ $a + b = 0$ ” etc. are removed and variable “ $a$ ” is replaced at all occurrences by “ $b$ ” or “ $-b$ ”. Modelica models contain a lot of such alias variables by construction, because every `connect( . . )` definition leads to such equations. Therefore, the reduction of the number of equations by this trivial substitution is usually substantial. Note, that the number of operations is not increased by this type of variable substitution.

In order to simplify notation, we collect the unknown variables together in vector  $\mathbf{z}$ :

$$\mathbf{z} = \begin{pmatrix} \dot{\mathbf{x}} \\ \mathbf{y} \end{pmatrix} \quad (12.9)$$

The goal is to solve the non-linear system of equations for the unknowns  $\mathbf{z}$ , under the assumption that time  $t$  and state variables  $\mathbf{x}$  are known quantities and the Jacobian with respect to the unknowns is regular:

$$\mathbf{0} = \mathbf{f}(\mathbf{z}), \quad \frac{\partial \mathbf{f}}{\partial \mathbf{z}} \text{ is regular, } \mathbf{z} \in \mathbb{R}^{nx+ny}, \quad \mathbf{f} \in \mathbb{R}^{nx+ny} \quad (12.10)$$

The basic idea shall be sketched by means of the following simple example of 5 equations  $f_i$  in 5 unknowns  $z_i$ :

$$\begin{array}{l} 0 = f_1(z_3, z_4) \\ 0 = f_2(z_2) \\ 0 = f_3(z_2, z_3, z_5) \\ 0 = f_4(z_1, z_2) \\ 0 = f_5(z_1, z_3, z_5) \end{array} \quad \mathbf{S} = \begin{array}{c|ccccc} & z_1 & z_2 & z_3 & z_4 & z_5 \\ \hline 0 = f_1 & 0 & 0 & 1 & 1 & 0 \\ 0 = f_2 & 0 & 1 & 0 & 0 & 0 \\ 0 = f_3 & 0 & 1 & 1 & 0 & 1 \\ 0 = f_4 & 1 & 1 & 0 & 0 & 0 \\ 0 = f_5 & 1 & 0 & 1 & 0 & 1 \end{array} \quad \begin{array}{l} \text{Incidence matrix } \mathbf{S}: \\ S_{ij} = 1, \text{ if variable } j \text{ is present in equation } i. \end{array}$$

In the left column, the structure of the 5 equations is displayed, i.e., it is defined which variable appears in which equation. In the middle column, the incidence matrix  $\mathbf{S}$  of this system of equations is shown. This matrix consists of zeros and ones and defines formally, whether variable  $j$  is present in equation  $i$ . Note, the incidence matrix is just a “conceptual” matrix that is used to clarify the operations to be carried out. When actually performing the BLT algorithm, the incidence matrix is never actually constructed, because this would be a vast of memory for large systems (e.g., for  $10^5$  equations, the incidence matrix has  $10^5 \cdot 10^5 = 10^{10}$  elements, i.e., about 10 Gbyte of memory would be needed, if an element would be stored in one byte).

The BLT algorithm sorts the equations in such a form that the unknowns can be computed in a “recursive” way. Equivalently, this can be stated as: Permute columns and rows of the incidence matrix, so that it is transformed to block lower triangular form. In the simple example above, this “*equation sorting*” can be easily performed manually resulting in:

$$\begin{array}{l} 0 = f_2(\underline{\mathbf{z}_2}) \\ 0 = f_4(\underline{\mathbf{z}_1}, z_2) \\ 0 = f_3(z_2, \underline{\mathbf{z}_3}, \underline{\mathbf{z}_5}) \\ 0 = f_5(z_1, \underline{\mathbf{z}_3}, \underline{\mathbf{z}_5}) \\ 0 = f_1(z_3, \underline{\mathbf{z}_4}) \end{array} \quad \mathbf{S}_2 = \begin{array}{c|ccccc} & z_2 & z_1 & z_3 & z_5 & z_4 \\ \hline 0 = f_2 & \mathbf{1} & 0 & 0 & 0 & 0 \\ 0 = f_4 & 1 & \mathbf{1} & 0 & 0 & 0 \\ 0 = f_3 & 1 & 0 & \mathbf{1} & \mathbf{1} & 0 \\ 0 = f_5 & 0 & 1 & \mathbf{1} & \mathbf{1} & 0 \\ 0 = f_1 & 0 & 0 & 1 & 0 & \mathbf{1} \end{array} \quad \begin{array}{l} \text{The solution can be directly derived:} \\ \rightarrow \begin{array}{l} 1. \text{ Solve } f_2 \text{ for } z_2 \\ 2. \text{ Solve } f_4 \text{ for } z_1 \\ 3. \text{ Solve } f_3 \text{ and } f_5 \text{ for } z_3 \text{ and } z_5 \\ 4. \text{ Solve } f_1 \text{ for } z_4 \end{array} \end{array}$$

From the BLT-form, the solution can be directly derived by solving the functions in a meaningful order, so that one variable at a time can be computed. In the left column above, the variables that are computed from the corresponding functions are marked in bold font and are underlined. For example,  $z_1$  can be computed from  $f_4$  since  $z_2$  is already computed in the previous step from  $f_2$ . In some cases, e.g.,  $f_3$  and  $f_5$  above, two or more variables can only be computed together. In such a case, a (non-trivial) algebraic loop is present. As a result, in the example above, an algebraic loop of 5 equations is transformed in to a sequence of operations

where 3 scalar equations are solved for one variable per equation and an algebraic loop with 2 equations has to be solved. Obviously, the solution problem was simplified by this transformation.

Since a Modelica translator knows the details of every equation, further symbolic processing is possible. In many cases, there is a linear relationship of the unknown variable in the corresponding function. For example, assume that  $f_4$  is linear in  $z_1$ , which is mathematically described as:

$$0 = f_{4a}(z_2) \cdot z_1 + f_{4b}(z_2) \quad (12.11)$$

It is then easy to analytically solve for  $z_1$  during translation of the model:

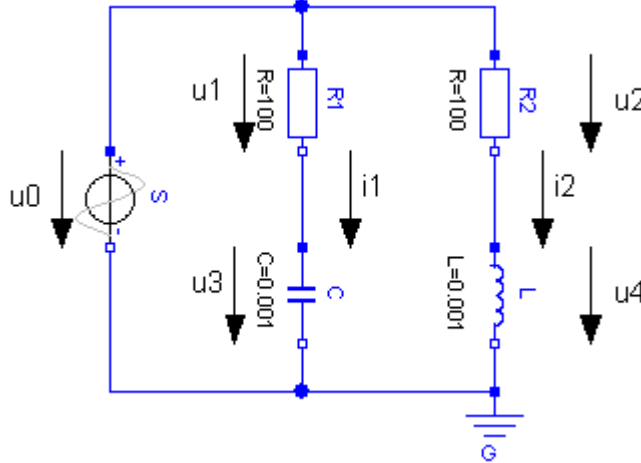
$$z_1 = \frac{-f_{4b}(z_2)}{f_{4a}(z_2)} \quad (12.12)$$

Of course, we have to assume that  $f_{4a}(z_2) \neq 0$ , in order that a division by zero does not occur. Sorting does not change the rank of the algebraic equation system (equations are just re-arranged in a different order). Since there is only one meaningful order to solve the sorted equations due to the block diagonal form of the incidence matrix<sup>15</sup>, the original system of equations would be rank deficient if  $f_{4a}(z_2)$  would be zero. As a result, if  $f_{4a}(z_2)$  becomes zero during simulation, then the original (non-transformed) system of equations does no longer has a unique solution and therefore the model is erroneous.

If a function is not linear in the variable for which it shall be solved, then a scalar non-linear algebraic equation is present and this equation has to be solved numerically during simulation for the unknown.

The sketched approach is demonstrated on the basis of the following simple electrical circuit:

Figure 12.3: Simple electrical circuit to demonstrate BLT transformation.



In order to follow more easily the transformation process, the equation system is not built from the Modelica model (which would result in 27 equations in 27 unknowns), but Kirchhoff's voltage and current laws are applied manually, resulting in the following 7 equations:

$$\begin{aligned} 0 &= u_1 - R_1 \cdot i_1 & (= f_1) \\ 0 &= u_2 - R_2 \cdot i_2 & (= f_2) \\ 0 &= C \cdot \frac{du_3}{dt} - i_1 & (= f_3) \\ 0 &= L \cdot \frac{di_2}{dt} - u_4 & (= f_4) \\ 0 &= u_2 + u_4 - u_0 & (= f_5) \\ 0 &= u_1 + u_3 - u_2 - u_4 & (= f_6) \end{aligned} \quad (12.13)$$

<sup>15</sup>For example, it is not possible to solve  $f_4$  for  $z_2$ , since  $z_2$  is already computed in the previous step from  $f_2$  and because  $f_2$  depends only on  $z_2$ , this is the only possible way to compute  $z_2$

The variables can be categorized as states  $\mathbf{x}$ , state derivatives  $d\mathbf{x}/dt$  and algebraic variables  $\mathbf{y}$ :

$$\mathbf{x} = \begin{pmatrix} u_3 \\ i_2 \end{pmatrix}, \quad \dot{\mathbf{x}} = \begin{pmatrix} \frac{du_3}{dt} \\ \frac{di_2}{dt} \end{pmatrix}, \quad \mathbf{y} = \begin{pmatrix} u_1 \\ u_2 \\ u_4 \\ i_1 \end{pmatrix} \quad (12.14)$$

Variables that appear differentiated ( $\mathbf{x}$ ) are assumed to be known and the unknowns are  $d\mathbf{x}/dt$  and  $\mathbf{y}$ . The functional dependency from the unknowns is therefore given by the left column below:

$0 = f_1(u_1, i_1)$ $0 = f_2(u_2)$ $0 = f_3(du_3/dt, i_1)$ $0 = f_4(di_2/dt, u_4)$ $0 = f_5(u_2, u_4)$ $0 = f_6(u_1, u_2, u_4)$	sorting (BLT) results in:	$0 = f_2(\underline{u}_2)$ $0 = f_5(u_2, \underline{u}_4)$ $0 = f_6(\underline{u}_1, u_2, u_4)$ $0 = f_1(u_1, \underline{i}_1)$ $0 = f_3(\dot{\underline{u}}_3, i_1)$ $0 = f_4(d\underline{i}_2/dt, u_4)$	solving for the unknowns results in:	$u_2 := R_2 \cdot i_2$ $u_4 := u_0 - u_2$ $u_1 := u_2 + u_4 - u_3$ $i_1 := u_1 / R_1$ $du_3/dt := i_1 / C$ $di_2/dt := u_4 / L$
--	------------------------------	--	--	--

Note, that  $f_2$  depends only on  $u_2$  during sorting, because  $i_2$  is a state and therefore assumed to be known. Sorting (or BLT transformation) results in the middle column. Using the actual equations, it is then possible to solve for the unknowns in a forward sequence without algebraic loops, as shown in the right column above.

Several algorithms are known to transform to BLT form, see, e.g., (Duff et.al., 1986). The mostly used approach splits the problem in two sub problems:

### 12.2.1 Step 1: Output set assignment

In a first phase, called “output set assignment”, the question is answered, for which variable an equation is solved for. Since this algorithm is also the basis to handle singular systems in section 12.4, we will discuss it in some detail. The basic idea shall be first sketched on the basis of a simple example:

original	step 1	step 2	step 5
$0 = f_1(z_3, z_4)$	$0 = f_1(\underline{z}_3, z_4) \Leftarrow$	$0 = f_1(\underline{z}_3, z_4)$	$0 = f_1(\underline{z}_3, z_4)$
$0 = f_2(z_1, z_6, z_7)$	$0 = f_2(z_1, z_6, z_7)$	$0 = f_2(\underline{z}_1, z_6, z_7) \Leftarrow$	$0 = f_2(\underline{z}_1, z_6, z_7)$
$0 = f_3(z_3, z_5)$	$0 = f_3(z_3, z_5)$	$0 = f_3(z_3, z_5)$	$0 = f_3(z_3, \underline{z}_5)$
$0 = f_4(z_1, z_2)$	$0 = f_4(z_1, z_2)$	$0 = f_4(z_1, z_2)$	$0 = f_4(z_1, \underline{z}_2)$
$0 = f_5(z_2, z_3, z_6)$	$0 = f_5(z_2, z_3, z_6)$	$0 = f_5(z_2, z_3, z_6)$	$0 = f_5(z_2, z_3, \underline{z}_6) \Leftarrow$
$0 = f_6(z_2)$	$0 = f_6(z_2)$	$0 = f_6(z_2)$	$0 = f_6(z_2)$
$0 = f_7(z_3, z_7)$	$0 = f_7(z_3, z_7)$	$0 = f_7(z_3, z_7)$	$0 = f_7(z_3, z_7)$

In the first column (called “original”), we have 7 equations  $f_i$  in 7 unknowns  $z_i$ . The goal is to uniquely assign one unknown to one equation with the intention that the assigned variable is computed from the corresponding equation. The important point is that no variable can be assigned twice, because it can only be once computed from an equation. In “step 1” we start with the first equation and assign arbitrarily one of the unknowns. Here we assign  $z_3$ . In “step 2” we continue with the second equation and assign one of the unknowns of this equation that has not yet been assigned before. We select arbitrarily  $z_1$ . In this way the algorithm continues without any problems until “step 5”. In this step, the first two unknowns,  $z_2$  and  $z_3$ , cannot be assigned, because these variables are already assigned in previous steps. So the only choice is to assign unknown  $z_6$ .

It is not possible to perform such a simple assignment in the next equation  $f_6$ . This equation has only one unknown  $z_2$  and this unknown has already been assigned. The idea is now to perform a systematic reassignment of all previous unknowns until it is possible to make an assignment to  $z_2$  in  $f_6$ . The process is sketched below:

$$\begin{aligned} 0 &= f_6(\underline{z}_2) \\ 0 &= f_4(\underline{z}_1, z_2) \\ 0 &= f_2(z_1, z_6, \underline{z}_7) \end{aligned}$$

Unknown  $z_2$  is already assigned in equation  $f_4$ . This equation has the two unknowns  $z_1$  and  $z_2$ . The assignment is changed to  $z_1$ . However,  $z_1$  is already assigned in  $f_2$ . This equation has the three unknowns  $z_1$ ,  $z_6$ , and  $z_7$ . The assignment is changed to  $z_7$ , since  $z_6$  is already assigned in  $f_5$ . The result is shown below, together with the final step 7:

<i>step 6</i>	<i>step 7</i>
$0 = f_1(\underline{z}_3, z_4)$	$0 = f_1(z_3, \underline{z}_4) \Leftarrow$
$0 = f_2(z_1, z_6, \underline{z}_7) \Leftarrow$	$0 = f_2(z_1, z_6, \underline{z}_7)$
$0 = f_3(z_3, \underline{z}_5)$	$0 = f_3(z_3, \underline{z}_5)$
$\dots \rightarrow 0 = f_4(\underline{z}_1, z_2) \Leftarrow$	$\rightarrow 0 = f_4(\underline{z}_1, z_2)$
$0 = f_5(z_2, z_3, \underline{z}_6)$	$0 = f_5(z_2, z_3, \underline{z}_6)$
$0 = f_6(\underline{z}_2) \Leftarrow$	$0 = f_6(\underline{z}_2)$
$0 = f_7(z_3, z_7)$	$0 = f_7(\underline{z}_3, z_7) \Leftarrow$

In “step 7”, again the two unknown variables  $z_3$  and  $z_7$  are already assigned. It is possible to perform a reassignment of  $f_1$  for the other unknown  $z_4$  and we have finally reached an assignment where every variable is assigned exactly once.

It might be that this is not possible. In such a case, the original equation system is structurally singular. This means that the equation system has no longer a unique solution, independently how the equations are built-up. This can also be stated the other way round: A system of equations must be not structurally singular in order that a unique solution is possible (= a necessary condition for uniqueness). For example, in the following system with 3 equations in 3 unknowns:

$$\begin{aligned}
 0 &= f_1(z_1, z_3) \\
 0 &= f_2(z_2) \\
 0 &= f_3(z_2)
 \end{aligned} \tag{12.15}$$

it is possible to assign either  $z_1$  or  $z_3$  to  $f_1$  and  $z_2$  to  $f_2$ . However, it is impossible to make an assignment for  $f_3$  because  $z_2$  is already assigned and a reassignment is not possible. If  $f_2(\cdot)$  is identical to  $f_3(\cdot)$ , we have an infinite number of solutions because either  $z_1$  or  $z_3$  can be selected arbitrarily and  $z_2$  can be computed either from  $f_2$  or from  $f_3$ . If the two functions are not identical, we have a contradiction, and there is no  $z_2$  that fulfills both equations all the time, i.e., there is no solution at all.

If the model equations are structurally singular, it is still possible that a unique solution of the underlying differential-algebraic equation exists by modifying the assumption that all variables appearing differentiated are states. The details are discussed in section 12.4.

The sketched approach of the output set assignment can be coded in a very compact form in a recursive function, as proposed in (Pantelides 1988). This algorithm is shown in form of a Modelica-like pseudo code that can be easily implemented in a desired programming language:

<pre> <b>output set assignment</b> Integer Assigned[n]; Boolean Visited [n]; Boolean success; <b>algorithm</b> Assigned := <b>zeros</b>(n);  <b>for</b> i <b>in</b> 1:n <b>loop</b>   Visited := <b>fill</b>(false,n);   success := <b>assign</b>(i);   <b>if not</b> success <b>then</b>     <b>error</b>("singular");   <b>end if</b>; <b>end for</b>; </pre>	<pre> <b>function assign</b> <b>input</b> Integer i; <b>output</b> Boolean success; <b>algorithm</b> <b>if</b> 'a variable j of equation i exists,   such that Assigned[j] = 0' <b>then</b>   success := <b>true</b>;   Assigned[j] := i; <b>else</b>   success := <b>false</b>;   <b>for</b> 'every variable j of equation i     with Visited[j] = <b>false</b>' <b>loop</b>     Visited[j] := <b>true</b>;     success := <b>assign</b>(Assigned[j]);     <b>if</b> success <b>then</b>       Assigned[j] = i;       <b>return</b>;     <b>end if</b>; </pre>
---	---

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

There are two global arrays that hold the current status and they are available in both functions:

i = Assigned[j]:	Equation i is solved for variable j.
	If i=0, there is not yet an assignment for variable j.
Visited[i]	: if <b>false</b> then variable i is not yet visited, otherwise it was already visited

The main function analyzes every equation once. Before starting the analysis of equation  $i$ , the “Visited” array is initialized to **false**. Then the central function “**assign**(.)” is called, to perform an assignment for this equation. It is first tried to perform a “simple assignment”, i.e., trying whether any of the unknown variables of the equation can be directly assigned, because at least one of them is not yet assigned. If this is possible, the function returns successfully. Otherwise, it is tried systematically, to perform a reassignment of the unknown variables. For this process, the “Visited” array is used to mark, which variable was already tried to be reassigned, in order to not repeat the same computations. Then, the “**assign**(.)” function is called recursively to perform a reassignment. If an assignment is not possible after trying all possible combinations of reassignments, the system of equations is structurally singular.

The worst case complexity of the algorithm is  $\mathcal{O}(n \cdot m)$ , where “ $n$ ” are the number of equations and “ $m$ ” is the sum of the variables appearing in all equations. In the example above,  $n = 7$ ,  $m = 15$ . This means that the number of operations to compute the complete assignment is at most “ $k \cdot n \cdot m$ ” with a suitable constant “ $k$ ”. The worst case complexity usually only appears for specially constructed examples. It turns out that in most practical cases for Modelica models the complexity is about  $\mathcal{O}(n)$ , i.e., the number of operations grows linearly with the number of equations.

### 12.2.2 Step 2: Loops of a directed graph (algorithm of Tarjan)

With the result of the previous phase, a directed graph can be constructed, containing the functions with their assigned variables as “nodes” and the dependency from the other unknowns described by appropriate “branches”. For example, the result of the “output set assignment” of the example above:

$$\begin{array}{ll} 0 = f_1(z_3, \underline{z_4}) & 0 = f_5(z_2, z_3, \underline{z_6}) \\ 0 = f_2(z_1, z_6, \underline{z_7}) & 0 = f_6(\underline{z_2}) \\ 0 = f_3(z_3, \underline{z_5}) & 0 = f_7(\underline{z_3}, z_7) \\ 0 = f_4(\underline{z_1}, z_2) & \end{array}$$

can be transformed to the following set of nodes:



Figure 12.4: Output set assignment visualized as directed graph (incomplete).

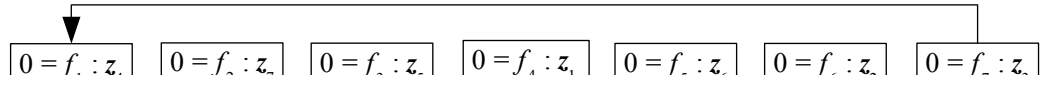
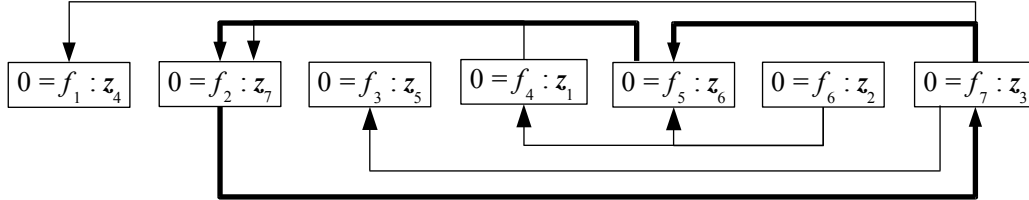


Figure 12.5: Output set assignment visualized as directed graph (complete).



where every equation  $f_i$  is a node together with its assigned variable. The dependency of the equations from the respective variables is defined by appropriate branches. For example, a branch from node  $f_7/z_3$  to node  $f_1/z_4$  signals that variable  $z_3$  is needed to compute  $z_4$ . The complete directed graph of this example is shown in the next figure:

The task is to determine the “loops” in this directed graph (also called “strong components”). For this, the very efficient graph algorithm of Tarjan (Tarjan 1972) is used that determines all “loops” in  $O(n)$  operations, where “ $n$ ” is the number of “nodes” (i.e. the number of equations). In the example, only one loop is present as visualized in the figure above by a larger line width of the corresponding branches. All variables in this loop can only be computed together and characterize the smallest algebraic loop that can be reached by sorting the equations. The other variables can be computed in a recursive sequence. For example,  $z_2$  is computed from  $f_6$ . Then,  $z_1$  can be computed from  $f_4$  using the already computed  $z_2$ , etc. The final result of this analysis is shown below, i.e., the BLT form of the example:

$$\begin{aligned}
 0 &= f_6(z_2) \\
 0 &= f_4(z_1, z_2) \\
 0 &= f_2(z_1, z_6, z_7) \\
 0 &= f_5(z_2, z_3, z_6) \\
 0 &= f_7(z_3, z_7) \\
 0 &= f_1(z_3, z_4) \\
 0 &= f_3(z_3, z_5)
 \end{aligned} \tag{12.16}$$

The worst case complexity of the overall algorithm to transform to BLT form is the sums of the two used algorithms, i.e., it is  $O(n \cdot m)$ . On a standard PC, a suitable implementation can transform 100000 equations and more in a few seconds to BLT form.

### 12.3 Solving Algebraic Equations (Tearing)

In the last section, it was discussed how the model equations are sorted and algebraic loops of minimal dimensions (with respect to sorting) are identified. Applying the BLT transformation to block diagrams gives in most cases the optimal result, since often no algebraic loops appear. Applying the transformation to physical systems, e.g., electrical circuits, mechanical or fluid systems, results nearly always in algebraic loops. Surprisingly, the “minimal dimensions” of the algebraic loops in the BLT-form are usually still quite large for physical systems and a direct solution would be still inefficient. Fortunately, it is possible to reduce the effort to solve the algebraic loops by a special variable substitution technique that is called “tearing”. The basic idea is well known under different names since the sixties. It is sketched at the following simple example:

$$\begin{aligned}
z_1 &= f_1(z_5) \\
z_2 &= f_2(z_1) \\
z_3 &= f_3(z_1, z_2) \\
z_4 &= f_4(z_2, z_3) \\
z_5 &= f_5(z_4, z_1)
\end{aligned} \tag{12.17}$$

This is a system of 5 algebraic equations in 5 unknowns  $z_i$ . It is not possible to reduce the dimension of the algebraic loop by sorting here. The numerical solution of this non-linear system of equations would be computed by providing “guess” values for the 5 unknowns and then computing the residues of every equation. The numerical solver modifies the unknown variables so that the residues become small. Therefore, the equation system must be provided to the numerical solver in form of the following function:

<b>input:</b>	$z_1, z_2, z_3, z_4, z_5$ (iteration variables )
<b>output:</b>	$r_1, r_2, r_3, r_4, r_5$ (residues)
<b>algorithm:</b>	$r_1 := z_1 - f_1(z_5)$ $r_2 := z_2 - f_2(z_1)$ $r_3 := z_3 - f_3(z_1, z_2)$ $r_4 := z_4 - f_4(z_2, z_3)$ $r_5 := z_5 - f_5(z_4, z_1)$

Due to the special structure of the equations, where some unknowns can be computed explicitly from other unknowns, it is possible to reduce the number of iteration variables from 5 to 1:

<b>input:</b>	$z_5$ (iteration variable)
<b>output:</b>	$r_5$ (residue)
<b>algorithm:</b>	$z_1 := f_1(z_5)$ $z_2 := f_2(z_1)$ $z_3 := f_3(z_1, z_2)$ $z_4 := f_4(z_2, z_3)$ $r_5 := z_5 - f_5(z_4, z_1)$

Since a non-linear numerical solver needs at least  $O(n^3)$  operations to compute the solution, where “n” is the number of iteration variables, the number of operations is reduced significantly, if the number of iteration variables can be reduced as in the example above.

The main difficulty is now to determine symbolically during translation the tearing (or iteration) variables, here:  $z_5$ , and the corresponding residue equations (here:  $r_5$ ), so that the number of iteration variables is as small as possible. In (Carpanzano and Girelli 1997) it is shown that this requires in general to try out all possible combinations and that the number of operations grows exponentially with the system size (= NP-complete problem). Therefore, it is impossible that such a nice “optimal” algorithm as for the BLT form exists and only heuristic algorithms are available. Another severe difficulty is that the rank of the system shall not be changed by the tearing transformation. For example, a system of 3 equations in 3 unknowns might be described by:

$$\begin{aligned}
z_1 &= \sin(\omega \cdot t) \\
z_1 \cdot z_2 &= \sin(z_3) \\
0 &= z_2^2 - z_3
\end{aligned} \tag{12.18}$$

By BLT transformation it is determined that  $z_1$  is computed from the first equation and that  $z_2$  and  $z_3$  have to be computed simultaneously from the last two equations. With the tearing approach one could compute  $z_2$  from the second equation and then solve 1 non-linear equation in  $z_3$ :

<b>known:</b>	$z_1$
<b>input:</b>	$z_3$ (iteration variable)
<b>output:</b>	$r_3$ (residue)
<b>algorithm:</b>	$z_2 := \sin(z_3)/z_1$ $r_3 := z_2^2 - z_3$

However, during simulation  $z_1$  will become zero and then a division by zero will take place in the tearing transformed equation. When the original equations 2 and 3 are simultaneously solved together, this problem does not occur and the equations have a unique solution. In other words, this application of the tearing transformation is erroneous. Due to this problem, nearly all known tearing algorithms introduce some kind of “pivoting” in order to guarantee that the rank of the system is not changed by the tearing transformation. However, it is then no longer possible to perform the tearing transformation *symbolically once* during translation. Instead, the numerical values of the pivots have to be watched during simulation and when the tearing transformation is no longer appropriate (in the example above, when  $z_1$  comes too close to zero), a new selection of tearing variables and residue equations has to be determined dynamically, see, e.g., (Duff et. al. 1986, Mah 1990). In that case, the tearing approach is no longer attractive and other “sparse matrix” methods are most likely to be more effective.

Fortunately, for Modelica models more information is available (as just the information whether a variable is present in an equation or not). It is possible to construct tearing algorithms where the details of the equations are taken into account in order to select tearing variables and residue equations so that it is guaranteed that the transformation does not change the rank of the system. Therefore, such a transformation can be carried out once during translation of the Modelica model. Within Dymola, a very effective (unpublished) algorithm of this kind is available that produces in many practical cases the “minimal” or “close to the minimal” number of iteration variables. For example, in the example above Dymola selects  $z_2$  as iteration variable, leading to the following tearing transformation:

<b>known:</b>	$z_1$
<b>input:</b>	$z_2$ (iteration variable)
<b>output:</b>	$r_2$ (residue)
<b>algorithm:</b>	$z_3 := z_2^2$ $r_2 := z_1 \cdot z_2 - \sin(z_3)$

As can be seen, a division by zero cannot take place in the function and the residue is always well defined. Therefore, the tearing transformed and the original system of equations have the same rank and both equation systems have the same solution.

Due to the above general requirement for any symbolic tearing algorithm, a modeling specialist can influence the tearing transformation by the way how the model equations are defined (note, this is in contrast to the BLT form that is independent of the original equation ordering or the way an equation is defined). For example, a media model might contain the idea gas law:

$$p = \rho \cdot R \cdot T \quad (12.19)$$

where  $p$  is the absolute pressure,  $\rho$  is the density,  $T$  is the thermodynamic temperature and  $R$  is the gas constant. The only way a symbolic tearing algorithm can use this equation is, to substitute the pressure  $p$  at other places, i.e., to potentially use either  $\rho$  or  $T$  or both as iteration variables or to compute the two variables in a previous step by other equations. It is not possible to use  $p$  as an iteration variable because, e.g.,  $\rho$  would then be computed as:

$$\rho = p / (R \cdot T) \quad (12.20)$$

Therefore, potentially a division by zero can occur. In reality this is not possible because a suitable media model will never be evaluated for  $T = 0$  K. However, it is impossible to deduce this property from the model equations and therefore every reliable symbolic tearing algorithm has to assume that  $T$  may become zero and that therefore  $p$  can not be selected as iteration variable. There are more suitable options for a tool if the gas

law is directly written in the form (12.20). The reason is that the `modeler` divides by  $T$ . It is therefore the modeler's responsibility that this variable never becomes zero (and for the gas law this is actually the case). A tool can now either use  $p$  or  $\rho$  as iteration variable because in both cases the rank of the Jacobian of this equation is not reduced when solving for the remaining unknown. If the modeler uses  $p$  and  $T$  as state variables (which is a natural choice for gases), then this is a good choice, because potentially  $\rho$  is computed directly from this equation. The choice of (12.19) is not as good, because a tool would have to divide by the state " $T$ " to compute  $\rho$  and a reliable tool cannot do this, if there are other choices. Therefore, if this equation appears in a larger algebraic loop, it is not possible to simply eliminate  $\rho$  and therefore the number of iteration variables will most likely be larger.

Note, the discussed property does not only hold for the tearing algorithm used in Dymola but for every suitable symbolic tearing algorithm, so it is a tool independent property (for reliable tools).

Once the tearing variables and residue equations are determined, Dymola analysis whether the equations from an algebraic loop depend only linearly on the iteration variables. If this is the case, Dymola transforms the teared non-linear system of equations symbolically in a system of linear equations. This is performed in the following (well-known) way: Assume that the equations of the algebraic loop are partitioned into tearing variables  $z_2$  and other variables  $z_1$ :

$$\begin{pmatrix} \mathbf{L} & \mathbf{J}_{12} \\ \mathbf{J}_{21} & \mathbf{J}_{22} \end{pmatrix} \cdot \begin{pmatrix} \mathbf{z}_1 \\ \mathbf{z}_2 \end{pmatrix} = \begin{pmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \end{pmatrix} \quad (12.21)$$

Since  $z_2$  are tearing variables, the remaining variables  $z_1$  can be computed in a forward sequence. This is only possible if  $L$  is a lower triangular matrix. Since a rank-preserving symbolic tearing transformation is used, the diagonal elements of the  $L$  matrix are guaranteed to be non-zero and therefore  $L$  is a regular lower triangular matrix. Due to this property, the inverse of  $L$  exists and it is possible to solve a linear system of equations in  $L$  in at most  $O(n_L^2)$  operations, where " $n_L$ " is the row and column dimension of  $L$  (without this property,  $O(n_L^3)$  operations are needed). It is then possible to transform the system above to:

$$\begin{pmatrix} \mathbf{J}_{22} - \mathbf{J}_{21} \cdot \mathbf{L}^{-1} \cdot \mathbf{J}_{12} & \mathbf{0} \\ -\mathbf{J}_{12} & \mathbf{L} \end{pmatrix} \cdot \begin{pmatrix} \mathbf{z}_2 \\ \mathbf{z}_1 \end{pmatrix} = \begin{pmatrix} \mathbf{b}_2 - \mathbf{L}^{-1} \cdot \mathbf{b}_1 \\ \mathbf{b}_1 \end{pmatrix} \quad (12.22)$$

The result is a BLT form with a (usually) much smaller diagonal block in the left upper corner. The lower right corner is matrix  $L$ , which has only trivial, non-zero diagonal elements. As a result the original algebraic loop in  $\dim(z_1) + \dim(z_2)$  equations has been transformed symbolically during translation into an algebraic loop of  $\dim(z_2)$  equations. If the number of "tearing" (also called "iteration") variables  $z_2$  is small, this reduction in the size of the algebraic loop leads usually to a much more efficient computation of the unknowns.

## 12.4 Singular Systems (Pantelides, Dummy Derivative, not yet available)

In the previous sections we have analyzed DAEs of the form:

$$\mathbf{0} = \mathbf{f}(\dot{\mathbf{x}}(t), \mathbf{x}(t), \mathbf{y}(t), t), \quad t \in \mathbb{R}, \quad \mathbf{x} \in \mathbb{R}^{n_x}, \quad \mathbf{y} \in \mathbb{R}^{n_y}, \quad \mathbf{f} \in \mathbb{R}^{n_x + n_y} \quad (12.23)$$

where  $\mathbf{x}$  are variables that appear differentiated in the model equations and  $\mathbf{y}$  are algebraic variables. Algorithms have been sketched, and demonstrated by examples, to transform this DAE into an ODE of the form:

$$\begin{pmatrix} \dot{\mathbf{x}}(t) \\ \mathbf{y}(t) \end{pmatrix} = \mathbf{f}_2(\mathbf{x}(t), t) \quad (12.24)$$

This transformation is possible if and only if the Jacobian with respect to the unknowns is regular at all time instants. In this section, systems are handled where this requirement is not fulfilled. We will call them "singular systems" because the Jacobian is singular:

$$\mathbf{J} = \left( \frac{\partial \mathbf{f}}{\partial \dot{\mathbf{x}}} : \frac{\partial \mathbf{f}}{\partial \mathbf{y}} \right) \text{ is singular} \quad (12.25)$$

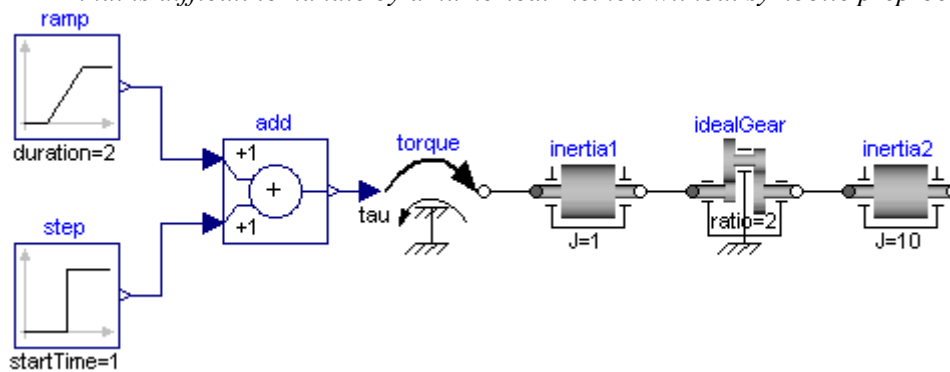
Since the basic requirement of a regular Jacobian is violated, it is no longer possible to transform to the ODE form (12.24).

XXXX

## 12.5 Hybrid DAEs need Symbolic Preprocessing

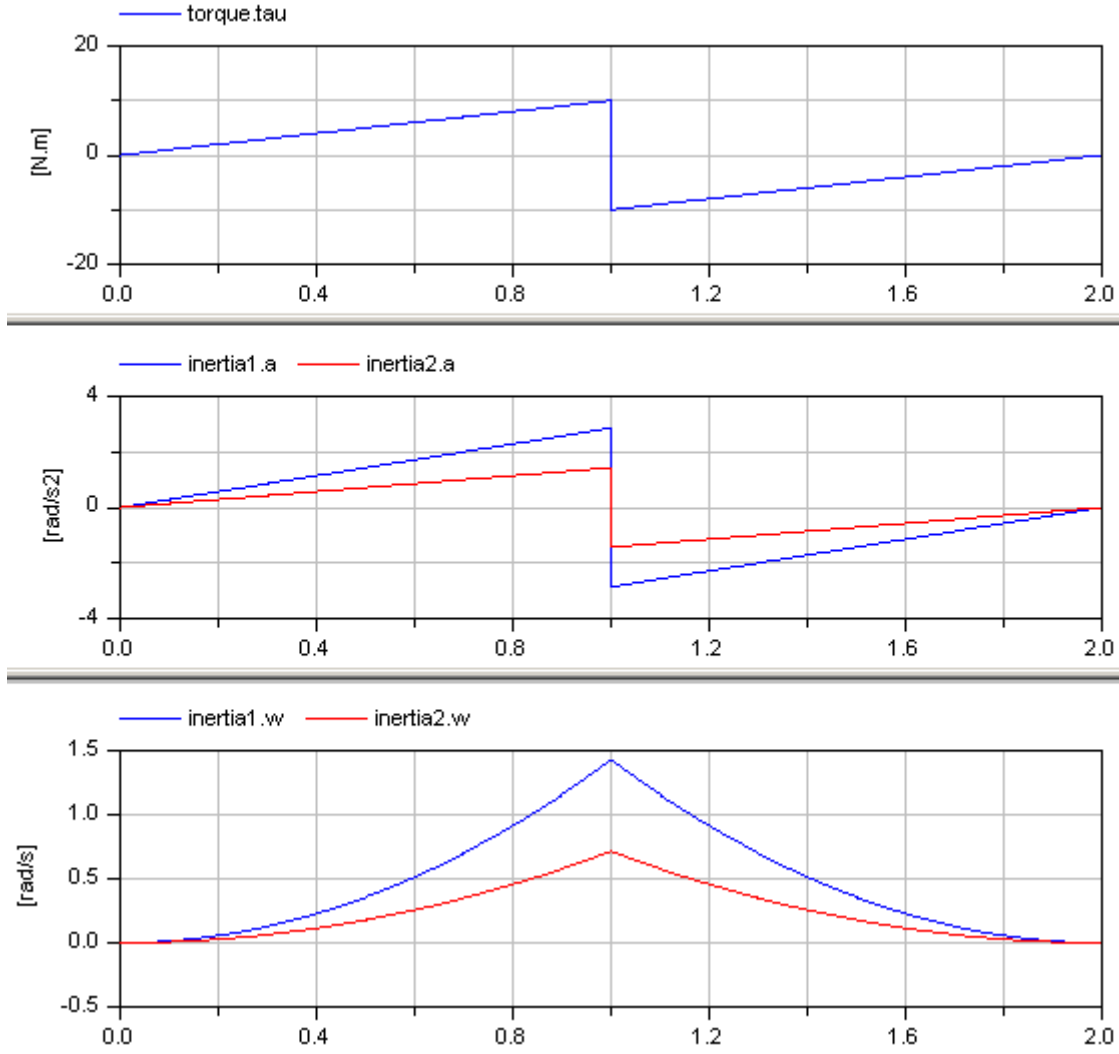
There are a large number of publications how to solve DAE systems by *numerical methods* and there are several simulation systems based on this approach. In most engineering applications “pure” DAEs do not occur, but usually in combination with discontinuities or other event-based effects as described in “Part B” of this document. These types of systems are called “*hybrid* DAEs”. In this section, the limits of numerical integration methods shall be discussed when applied to hybrid DAEs *without* symbolic preprocessing. The problematic cases are *singular* DAEs, i.e., DAEs that cannot be transformed algebraically to state space form without differentiating equations. Although certain classes of singular DAEs can be directly integrated with a numerical method, severe problems occur when variables change discontinuously. This shall be demonstrated by means of the simple example of Figure 12.6:

Figure 12.6: Simple drive train with discontinuous change of the driving torque that is difficult to handle by a numerical method without symbolic preprocessing.



The drive train consists of a rotational inertia “inertia1” that is connected via an ideal gear to another rotational inertia “inertia2”. The driving torque is a ramp that has a discontinuous change at time = 1, as shown in the figure below. Note, discontinuous changes of driving forces or torques occur naturally with sampled data systems.

Figure 12.7: Dymola simulation results of drive train of Figure 12.6.



Simulating this model with Dymola or another Modelica simulator is unproblematic and leads to the result shown in the Figure above. Note, that at time = 1 s, the accelerations of the two inertias change discontinuously due to the discontinuous change of the driving torque. Let us now analyze this system under the assumption that no symbolic preprocessing would take place. The system above is described by the following equations (the equations of the angles are ignored for simplicity):

$$\begin{aligned}
 J_1 \cdot \dot{\omega}_1 &= \tau_{drive} - \tau_1 && // \text{equation of inertia1} \\
 \omega_1 &= i \cdot \omega_2 && // \text{equation 1 of idealGear} \\
 \tau_2 &= i \cdot \tau_1 && // \text{equation 2 of idealGear} \\
 J_2 \cdot \dot{\omega}_2 &= \tau_2 && // \text{equation of inertia2}
 \end{aligned} \tag{12.26}$$

where  $\omega_1, \omega_2$  are the angular velocities of `inertia1` and `inertia2`,  $\tau_{drive}$  is the pre-defined driving torque that changes discontinuously at time = 1s, “ $i$ ” is the gear ratio, and  $\tau_1, \tau_2$  are the constraint torques in the left and right flange of the gear. It is possible to solve this system with an implicit integration algorithm, see Chapter 13, without symbolic pre-processing because equations need to be differentiated only once in order to transform algebraically to state space form (if more differentiations would be needed, numerical integration algorithms are usually no longer reliable). Under the assumption that the differentiated variables are known, i.e.,  $\omega_1, \omega_2$ , these are 4 equations in the 4 unknowns  $\dot{\omega}_1, \dot{\omega}_2, \tau_1, \tau_2$ . At time = 1s, an event occurs and  $\tau_{drive}$  changes discontinuously.

The values of all variables are known just before the event occurs. At the event instant the driving torque changes. The important question is how to initialize this DAE to restart the integration? The usual assumption is that the differentiated variables remain continuous over an event. We have therefore 4 equations in 4

unknowns  $\dot{\omega}_1$ ,  $\dot{\omega}_2$ ,  $\tau_1$ ,  $\tau_2$  at the event instant. For every *singular* DAE this system of equations is *singular* (this is the definition of a singular DAE). A well-posed, singular DAE has an infinite number of solutions. For example, in equation (12.26), the angular acceleration of `inertia2` can be selected arbitrarily and the other variables can then be computed uniquely. For example, the following values fulfill the DAE:

$$\begin{aligned}\dot{\omega}_1 &:= \tau_{drive}/J_1 \\ \dot{\omega}_2 &:= 0 \\ \tau_1 &:= 0 \\ \tau_2 &:= 0\end{aligned}\tag{12.27}$$

Although, these values fulfill the DAE (12.26), they are not correct, because the mathematical solution requires that

$$\dot{\omega}_1 = i \cdot \dot{\omega}_2\tag{12.28}$$

and this equation is not fulfilled from the values above. The trouble is that this equation is not present in the DAE, but is only implicitly defined by *differentiating* the second equation of (12.26).

To summarize, since a *singular* DAE has an *infinite* number of solutions at an *event* instant, it is *impossible* for *any numerical method* to derive the mathematically correct solution without taking into account the “hidden” constraint equations that can be derived by the Pantelides algorithm or another method. As a result, a simulation system that does not take into account the “hidden” constraint equations of a singular DAE when a variable changes discontinuously, either fails at an event instant or produces a wrong solution around the event point. As a consequence, a reliable modeler should not solve singular DAEs with such a simulation system, even if it seems to be that the solver can handle it in some cases.

When building models with components from the Modelica Standard Library, the results are often singular DAEs, because the components are designed for maximal flexibility. This is especially the case for the Modelica.Mechanics.MultiBody, Modelica.Mechanics.Rotational, Modelica.Mechanics.Translational, Modelica.Media and Modelica\_Fluid libraries, and for many inverse systems (see Chapter 19) constructed with any library. Since the Pantelides algorithm together with the “Dummy Derivative” method are the only known methods to transform generic, singular DAEs into a regular form, every Modelica simulation environment that shall be able to simulate models constructed with the Modelica Standard Library, has to support these two algorithms or a variant of them.

Let us analyze how other simulation environments treat singular DAEs:

- Modelica simulation environments, such as Dymola ([www.Dynasim.se](http://www.Dynasim.se)), MathModelica ([www.mathcore.com](http://www.mathcore.com)), OpenModelica ([www.ida.liu.se/~pelab/modelica/OpenModelica.html](http://www.ida.liu.se/~pelab/modelica/OpenModelica.html)) support the Pantelides algorithm and the dummy derivative method, i.e., they can handle singular DAEs with discontinuous variable changes.
- SIMPACK ([www.simpack.com](http://www.simpack.com)) is a software system to simulate complex multi-body systems with rigid and flexible bodies. The mechanical equations can be integrated numerically as a singular DAE (index 2 formulation). At an event instant, singular DAEs are always transformed to regular DAEs using the well-known transformation techniques of mechanical systems. Therefore, SIMPACK can handle certain classes of singular DAEs with discontinuous variable changes.
- “The MathWorks” ([www.mathworks.com](http://www.mathworks.com)) provides the block-diagram simulator Simulink and blocksets such as SimMechanics and SimPowerSystems. Simulink supports a subset of regular DAEs. The blocksets use domain-specific algorithms to transform singular DAEs (such as a mechanical system) to state space form. As a result, reliable solutions of certain classes of singular DAEs with discontinuous variable changes can be expected. It is, however, not possible to handle generic singular DAEs, as, e.g., occurring when building inverse systems, see Chapter 19.
- In the electronic field, modeling is performed with HDLs (= Hardware Description Languages) such as MAST ([www.openmast.org](http://www.openmast.org)) or VHDL-AMS ([www.eda.org/vhdl-ams](http://www.eda.org/vhdl-ams)). HDL simulators have usually numerical integration methods *without* symbolic or domain specific preprocessing to transform a singular DAE to a regular DAE and it is therefore questionable whether they can solve reliably singular DAEs with discontinuous variable changes<sup>16</sup>. This is usually not critical in the electronic field, since singular systems

<sup>16</sup>It is of course possible to model mechanical and/or fluid systems in another modeling environment (e.g., in Modelica), perform the transformation to a regular, hybrid DAE in this other environment and use the result in a HDL simulator or

do not often occur here.

---

alternatively use co-simulation.



## **Chapter 13   Integration Algorithms**

### **13.1   Integration Algorithms**

### **13.2   Method Order (not yet available)**

### **13.3   Stability Region (not yet available)**

### **13.4   Inline Integration (not yet available)**

### **13.5   Event Handling and Chattering (not yet available)**

### **13.6   Practical Considerations (not yet available)**



## Part D Libraries for Physical Systems

This part contains a description of a selected set of libraries available in the Modelica Standard Library. All libraries are first described from the view “how to use the library” and then internal details of the implementation are discussed.

## **Chapter 14 Drive Trains (Rotational library)**

## **Chapter 15 Mechanical Systems (MultiBody library)**

In this chapter an overview of the Modelica.Mechanics.MultiBody library is given. It is explained how to use the elements to build complex mechanical systems, how the library is implemented and how a modeler can introduce his/her own additional basic models, such as special joint or force elements.

## **Chapter 16 Thermo-Fluid Systems (Media/Fluid libraries)**

In this chapter an overview of the Modelica.Media and Modelica\_Fluid libraries is given to model thermo-fluid flow in pipe networks.

## Part E Libraries for Control Systems

This part provides an overview how to use Modelica in control applications, in particular:

- Simulation and design of continuous and digital control systems with the Modelica\_LinearSystems library.
- Defining state machines and logical controller functions with the Modelica.StateGraph and the Modelica.Blocks.Logical library.
- Automatically constructing inverses of non-linear Modelica models and using them in advanced control systems. This topic is closely related to the discussed symbolic transformation algorithms and is a unique feature of Modelica and of Modelica tools to design controllers of non-linear plants in a systematic fashion.

## Chapter 17 Continuous and Digital Control Systems (LinearSystems library)

The free Modelica\_LinearSystems library provides basic data structures for linear control systems as well as operations on them, and includes blocks that allow quick switching between a continuous and a discrete representation of a multi-rate controller. It is useful to have both description forms of a controller in a consistent way: The continuous controller approximation allows much faster simulations and is often sufficient for parameter design studies whereas the discrete controller model is needed for more detailed analysis. This Modelica library is shipped with Dymola since some time and is available via the library menu (File / Libraries / Modelica\_LinearSystems). It is planned to include this library in the Modelica Standard Library, once some missing features in Modelica and in Dymola are available (such as “operator overloading”) and Dymola specific build-in functions are replaced by equivalent standardized functions (e.g., for plotting and for linearization).

Library ModelicaLinearSystems provides different representations of linear, time invariant differential and difference equation systems, as well as typical operations on these system descriptions. In the right figure a screen-shot of the first hierarchical level of the library is shown. *StateSpace*, *TransferFunction*, *ZerosAndPoles*, and *DiscreteStateSpace* are records that provide basic data structures for linear control systems. Every record contains a set of utility functions that operate on the corresponding data structure. Especially, operations are provided to transform the respective data structures in to each other.

Sublibrary *Math* contains the basic data structures *Complex* and *Polynomial* that are utilized from the linear system descriptions.

Sublibrary *Sampled* contains a library of input/output blocks to conveniently model and simulate sampled data systems where it is convenient to quickly switch between a continuous and a discrete block representation.

### 17.1 Basic Mathematical Data Structures

Sublibrary *Math* provides the basic data structures *Complex* and *Polynomial* that are utilized and needed from the linear systems data structures to be discussed in the next section.

In the Modelica Association there is currently a proposal to introduce operator overloading into the Modelica language. The data structures in the Modelica\_LinearSystems library are organized so that their usage becomes convenient once operator overloading is available.

The basic approach will be explained by means of the record *Math.Complex*. Its content is displayed on the right side. The record contains the basic definition of a complex number consisting of a real (re) and an imaginary (im) part:

```
record Complex
  Real re "real part";
  Real im "imaginary part";
  // function definitions

end Complex;
```

Additionally, a set of functions is present that operates on this record. Especially, function **constructor**(...) constructs an instance of this record and all functions 'xx' provide basic operations such as addition and multiplication. Note, the apostrophe is part of the function name. In the operator overloading proposal, predefined function names are defined, such as '+' that will be automatically be invoked when a Complex number is used in an arithmetic expression.

The Complex number record can be utilized in current Modelica environments, such as Dymola (Dynasim 2008), in the following way:



```

import Modelica.Utilities.Streams;
import Modelica_LinearSystems.Math.Complex;
Complex c1 = Complex(re=2, im=3) "= 2 + 3j";
Complex c2 = Complex(3,4)       "= 3 + 4j";
algorithm
  c3 := Complex.'+'(c1, c2) "= c1 + c2";
  Streams.print("c3 = " + Complex.'String'(c3));
  Streams.print("c3 = " + Complex.'String'(c3,"i"));

```

In the declaration, the two record instances `c1` and `c2` are defined and are added in the algorithm section to construct `c3`. The value of record `c3` is printed with the `Complex.'String'` functions in different forms resulting in the following print-out:

```

c3 = 5 + 7j
c3 = 5 + 7i

```

Once operator overloading is available, the above statements can be simply written as:

```

c3 := c1 + c2;
Streams.print("c3 = " + String(c3));
Streams.print("c3 = " + String(c3,"i"));

```

It is then also possible to write:

```

Complex j = Complex.j();
Complex c4 = -2 + 5*j;
Complex c5 = c1*c2 / ( c3 + c4);

```

In record *Math.Complex* the basic operations for complex numbers are provided.

In a similar way all other data structures of the `Modelica_LinearSystems` library are defined. They can all be at once used in current Modelica environments and they will be conveniently usable once operator overloading is available in Modelica and in Modelica tools.

Record *Math.Polynomial* defines a data structure for polynomials with *real-valued* coefficients and provides the most important operations on polynomials. Its content is displayed on the right side. A Polynomial is constructed by the command

```
Polynomial(coeff.Vector)
```

where the input argument provides the polynomial coefficients in descending order. For example, the polynomial  $y = 2x^2 + 3x + 1$  is defined as

```
Polynomial({2,3,1})
```

Besides arithmetic operations, functions are provided to compute the zeros of a polynomial (via the eigen values of the companion matrix), to differentiate, to integrate and to evaluate the function value, its derivative and its integral. Find below a typical example from the scripting environment of Dymola:

```

import Modelica_LinearSystems.Math.Polynomial;
p = Polynomial({6, 4, -3})
Polynomial.'String'(p) // = "-6*x^2 + 4*x - 3"

int_p = Polynomial.integral(p)
Polynomial.'String'(int_p) // = "-2*x^3 + 2*x^2 - 3*x"

der_p = Polynomial.derivative(p)
Polynomial.'String'(der_p) // = "-12*x + 4"

Polynomial.evaluate(der_p,1) // = -8

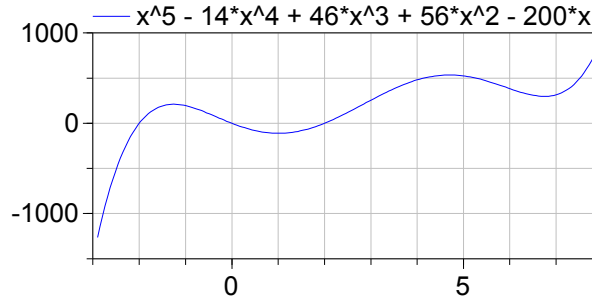
r = Polynomial.roots(p, printRoots=true)
// = 0.333333 + 0.62361j
// = 0.333333 - 0.62361j

```

With function `fitting(...)`, a polynomial can be determined that approximates given table values. Finally

with function `plot(..)`, the interesting range of  $x$  is automatically determined (via calculating the roots of the polynomial and of its derivative) and plotted. The plotting is currently performed with the `plot` function in Dymola. Once a Modelica built-in plot function is available, this will be adapted. A typical plot is shown in the next figure:

Figure 17.1: Plot of a polynomial with automatically detecting the interesting range.



## 17.2 Linear System Descriptions

At the top level of the Modelica\_LinearSystems library, data structures are provided as Modelica records defining different representations of linear, time invariant differential and difference equation systems. In the record definitions, functions are provided that operate on the corresponding data structure. Currently, the following linear system representations are available:

### 17.2.1 Record StateSpace

This record defines a multi-input, multi-output linear time-invariant differential equation system in state space form:

$$\begin{aligned}\dot{\mathbf{x}}(t) &= \mathbf{A} \cdot \mathbf{x}(t) + \mathbf{B} \cdot \mathbf{u}(t) \\ \mathbf{y}(t) &= \mathbf{C} \cdot \mathbf{x}(t) + \mathbf{D} \cdot \mathbf{u}(t)\end{aligned}\tag{17.1}$$

The data part of the record contains the constant matrices  $\mathbf{A}$ ,  $\mathbf{B}$ ,  $\mathbf{C}$ ,  $\mathbf{D}$  with the following definition:

```
record StateSpace
  Real A[:, :];
  Real B[size(A,1), :];
  Real C[:, size(A,1)];
  Real D[size(C,1), size(B,2)];
  // function definitions
end StateSpace;
```

The content of the StateSpace record definition is displayed on the right side. The *fromXXX* functions transform from another representation to a StateSpace record. Especially `fromModel(..)` linearizes a Modelica model and provides the linear system as output argument. `fromFile(..)` reads a StateSpace description from file and `fromTransferFunction(..)` transform a transfer function description into a StateSpace form.

The basic arithmetic operations are interpreted as series and parallel connection of StateSpace systems ('+' connects systems in parallel; '\*' connects systems in series). Function `invariantZeros(..)` computes the invariant zeros. For single-input, single-output systems these are the zeros of the transfer function. Function `plotEigenValues(..)` computes and plots the eigenvalues of the system.

### 17.2.2 Record TransferFunction

This record defines the transfer function between the input signal  $u$  and the output signal  $y$  by the coefficients of the numerator and denominator polynomials  $n(s)$  and  $d(s)$  respectively:

$$y = \frac{n(s)}{d(s)} \cdot u\tag{17.2}$$

The order of the numerator polynomial can be larger as the order of the denominator polynomial (in such a case, the transfer function can not be transformed to a `StateSpace` system, but other operations are possible). For example, the transfer function

$$y = \frac{2 \cdot s + 3}{4 \cdot s^2 + 5 \cdot s + 6} \cdot u \quad (17.3)$$

is defined in the following way:

```
import TF=LinearSystems.TransferFunction;
import Modelica.Utilities.Streams;

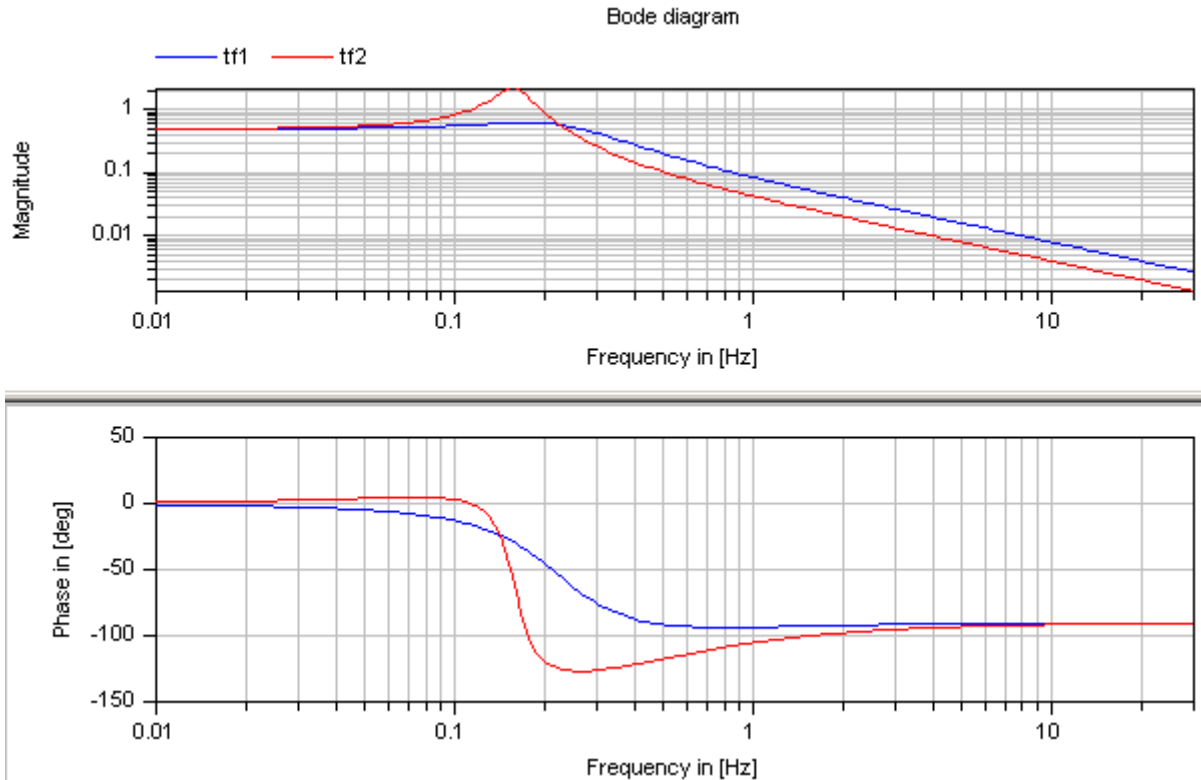
TF tf(n={2,3}, d={4,5,6});
print("y = " + TF.'String'(tf) + " * u")
```

The last statement prints the following string to the output window:

```
y = (2*s + 3) / (4*s^2 + 5*s + 6) * u
```

Besides arithmetic operations on transfer functions and constructing them optionally also from polynomials and from zeros and poles, the zeros and poles can be computed and a Bode plot can be constructed, as shown in the next figure:

Figure 17.2: Bode plot of two transfer functions generated with `Modelica_LinearSystems.TransferFunction.Examples.bodePlot2`.



The function call `plotBode(tf1, legend="tf1")` automatically selects an appropriate frequency interval and uses the selected legend. The useful frequency range is estimated such that the phase angle of the plot of one (numerator or denominator) zero is in the range:

$$\frac{\varphi_{\min}}{n} \leq |\text{phase angle}| \leq \frac{\pi}{2} - \frac{\varphi_{\min}}{n} \quad (17.4)$$

where  $n$  is the number of (numerator or denominator) zeros. Note, the phase angle of one zero for a frequency of 0 up to infinity is in the range:

$$0 \leq |\text{phase angle}| \leq \frac{\pi}{2} \quad (17.5)$$

Therefore, the frequency range is estimated such that the essential part of the phase angle (defined by  $\varphi_{\min}$ ) is present in the Bode plot (default is  $10^{-4}$  rad).

### 17.2.3 Record ZerosAndPoles

This record defines the transfer function between the input signal  $u$  and the output signal  $y$  by its zeros, poles and a gain:

$$y = k \cdot \frac{\prod (s - z_i)}{\prod (s - p_j)} \cdot u \quad (17.6)$$

where the zeros and poles are defined by two Complex vectors of coefficients  $z_i$  and  $p_j$ . The elements of the two Complex vectors must either be real numbers or conjugate complex pairs (in order that their product results in a polynomial with Real coefficients).

A description with zeros and poles is problematic: For example, a small change in the imaginary part of a conjugate complex pole pair, leads no longer to a transfer function with real coefficients. If the same zero or pole is present twice or more, then a diagonal state space form is no longer possible. This means that the structure is very sensitive if zeros or poles are close together. Performing arithmetic operations on such a description therefore leads easily to transfer functions with non-real coefficients. For this and other reasons, the constructor transforms this data structure and stores it internally in the record as first and second order polynomials with real coefficients:

$$y = k \cdot \frac{\prod (s + n_{1i}) \cdot \prod (s^2 + n_{2j} \cdot s + n_{3j})}{\prod (s + d_{1k}) \cdot \prod (s^2 + d_{2l} \cdot s + d_{3l})} \cdot u \quad (17.7)$$

All functions operate on this data structure. It is therefore guaranteed that an operation results again in a transfer function with real coefficients. It is possible to construct a ZerosAndPoles record directly with these coefficients by using function `fromFactorized(...)`.

This data structure is especially useful in applications where first and second order polynomials are naturally occurring, e.g., as for filters: With function `filter(...)` this factorized form is directly generated from the desired low and high pass filters of type *CriticalDamping*, *Bessel*, *Butterworth* or *Chebyshev*. The filter options can be seen in Figure 17.3, a screenshot of the input menu of this function.

Figure 17.3: Parameter menu of function `ZerosAndPoles.filter(...)`.

**filter**

Description  
Generate the data record of a ZerosAndPoles transfer function from a filter description

Inputs

analogFilter	AnalogFilter.CriticalDamping	Analog filter characteristics (CriticalDamping/Bessel/Butterworth/Chebyshev)
filterType	Types.FilterType.LowPass	Type of filter (LowPass/HighPass)
order	2	Order of filter
f_cut	$1/(2 \cdot \text{Modelica.Constants.pi})$ Hz	Cut-off frequency (default is $\omega_{\text{cut}} = 1$ rad/s)
gain	1.0	Gain (= amplitude of frequency response at zero frequency)
A_ripple	0.5 dB	Pass band ripple for Chebyshev filter (otherwise not used)
normalized	true	= true, if amplitude of low pass filter at $f_{\text{cut}}$ is $1/\sqrt{2}$ , otherwise unmodified filter

OK Info Copy Call Execute Close

Besides type of filter and whether it is a low or high pass filter, the order of the filter and the cut-off frequency can be defined. With option “normalized, filters can be defined in normalized (default) and non-normalized form.

In the normalized form, the amplitude of the filter transfer function at the cutoff frequency is 3 dB (= 0.7071...). When trying out different filter types with different orders, the result of a comparison makes only sense if the filter is normalized.

Note, when comparing the filters of this function with other software systems, the setting of "normalized" has to be selected appropriately. For example, the signal processing toolbox of Matlab provides the filters in non-normalized form and therefore `normalized = false` has to be set if results shall be compared.

In Figure 17.4 the magnitudes of the 4 supported low pass filters are shown in normalized form and in Figure 17.5 the corresponding step responses are given (generated with the `Modelica_LinearSystems.Sampled` library).

Figure 17.4: Magnitudes of normalized low pass filters of order 4.

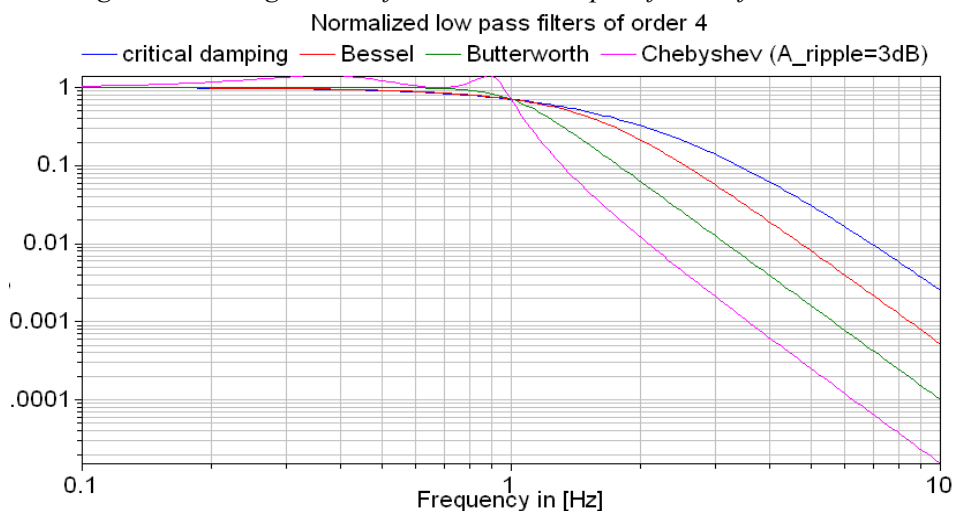
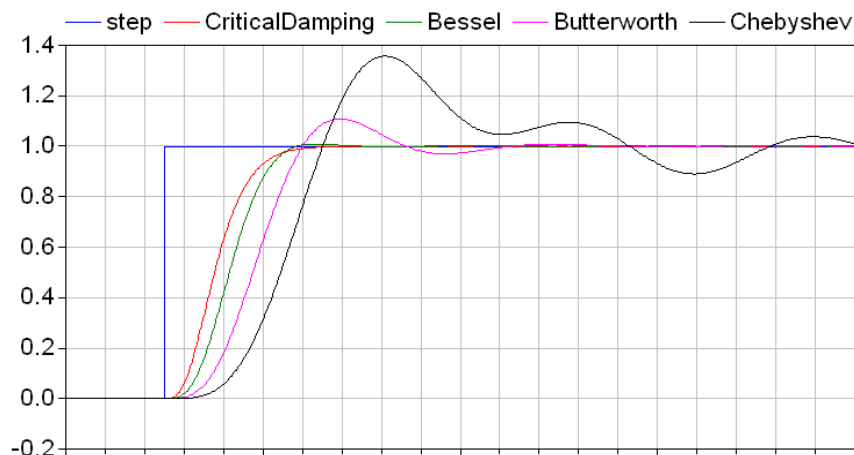


Figure 17.5: Step responses of normalized low pass filters of order 4.



Obviously, the frequency responses give a somewhat wrong impression of the filter characteristics: Although Butterworth and Chebyshev filters have a significantly steeper magnitude as the CriticalDamping and Bessel filters, the step responses of the latter ones are much better since the settling times are shorter and no overshoot occurs. This means for example, that a CriticalDamping or a Bessel filter should be selected, if a filter is mainly used to make a non-linear inverse model realizable.

#### 17.2.4 Record DiscreteStateSpace

This record defines a linear time invariant difference equation system in state space form. At the time of writing, this record contains only the core function `fromStateSpace(. .)` to transform a StateSpace description in a DiscreteStateSpace form. The details of this record and of this function are discussed in section .

### 17.3 Continuous/Discrete Control Blocks

The core of the Modelica\_LinearSystems library is sublibrary Sampled to model continuous and discrete multi-rate control systems. Experience shows that the combination of physical plant models that are controlled by digital controllers slow down the simulation speed significantly, if the sampling rates of the digital controllers are small compared to the step-size that could be used for the continuous plant. For example, at DLR detailed, calibrated models of robot systems are available. A stiff solver with variable step size integrates this system with step sizes in the order of 10 – 20 ms. When replacing the continuous approximation of the controllers by the actual digital controller implementation, the simulation time is *increased* by a factor of 20-30. The reason is that the digital controllers have a sample time in the order of 1 ms and therefore the step size of the integrator is limited by 1 ms. Additionally at every sample point the integrator has to be restarted to reliably handle the discontinuous change of the actuator signal which reduces again the simulation efficiency.

For some design phases a continuous approximation of a controller might be sufficient, e.g., when tuning the controller parameters by parameter variation or multi-criteria optimization. By reducing the simulation time with a factor of, say 20-30, will then significantly reduce the optimization time.

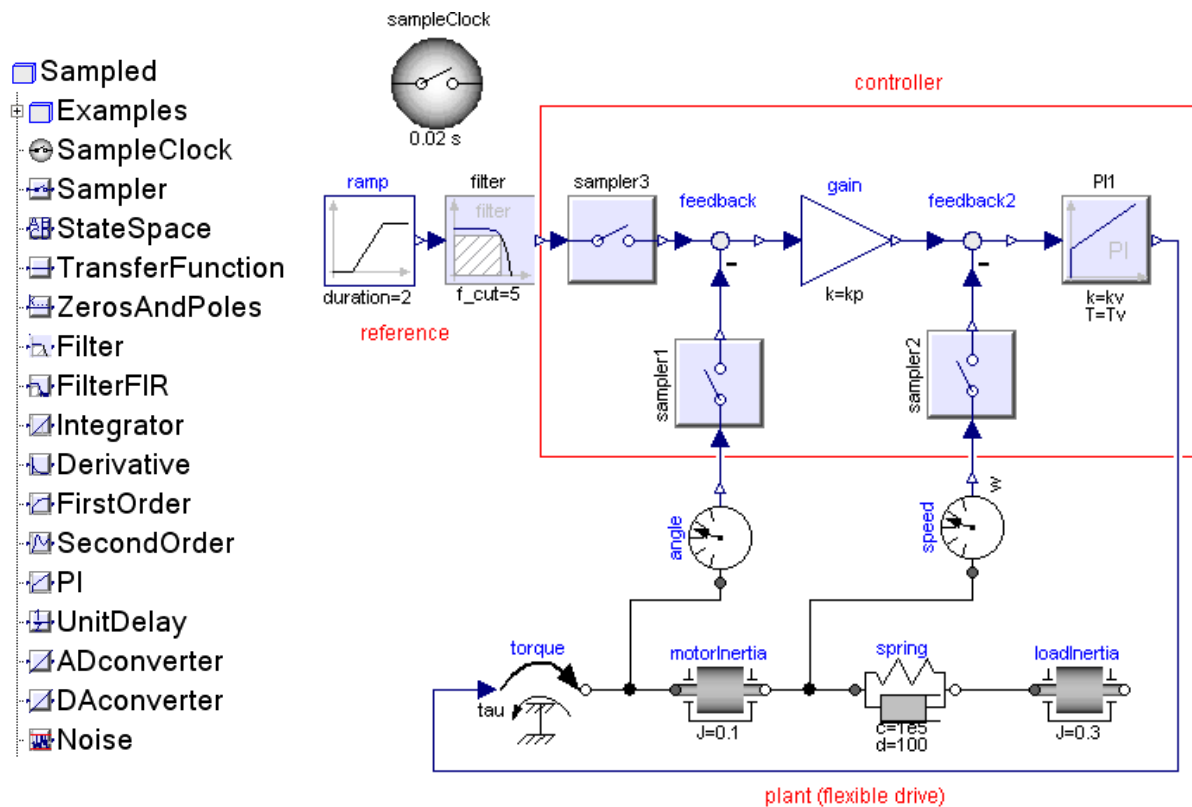
It is also necessary to validate or fine tune the controllers with the most detailed models available. Then effects such as sampling, AD-/DA-converter quantization, resolver quantization and noise, computing time of the control algorithms, signal communication times, as well as additional filters have to be taken into account.

Practical experience at DLR shows that it is difficult to maintain the consistency between a continuous and a digital representation of a control system model. For this reason, in a master thesis project (Walther 2002) a Modelica library was developed that allows to easily switch between a continuous and a discrete representation of a controller. This library has been in use at DLR for some years. Based on the gained experience in using this library, as well as new features in Modelica and in the Modelica simulation environment Dymola (Dynasim 2008), the library was considerably restructured, and completely newly implemented. Besides (Walter 2002), the books of (Aström and Wittenmark 1997) and (Tietze and Schenk 2002) have been helpful for the design and the actual implementation.

### 17.3.1 Introductory Example

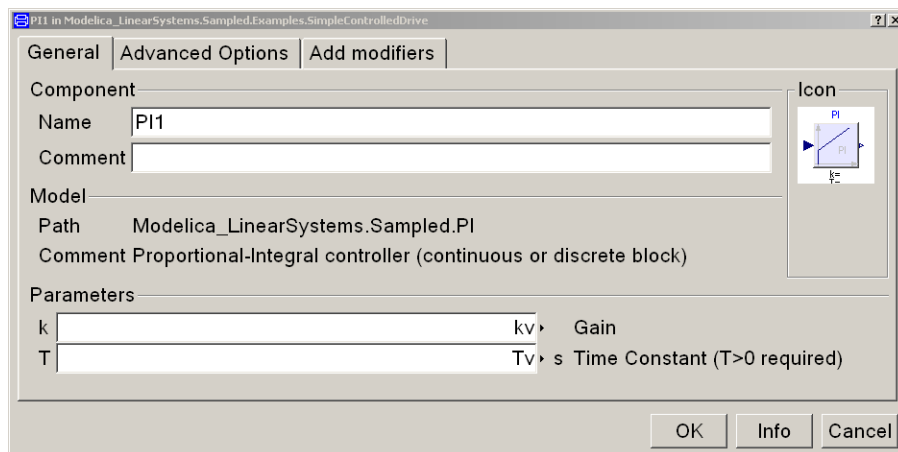
In the left part of Figure 17.6 below a screenshot of the *Sampled* library and a simple example in using it is shown in the right part of the figure. The example is a simple controlled flexible drive consisting of a motor inertia, gear elasticity and load inertia. The angle and the angular velocity of the motor inertia are measured and are used in the position and speed controller. The output of the speed controller is directly used as the torque driving the motor inertia. The multi-rate controller consists of all blocks in the red square together with component `sampleClock`. The latter defines the base sampling time together with defaults for the `blockType` (Continuous or Discrete), the `methodType` (the used discretization method for a discrete block) and the block initialization (none, initialize states, initialize in steady state). All *Sampled* blocks on the same hierarchical level as `sampleClock`, and all blocks on a lower hierarchical level, use the `sampleClock` setting as a default. Every block can override the default and can use its individual settings.

Figure 17.6: Blocks of *Sampled* library (left figure) and introductory example (right figure).



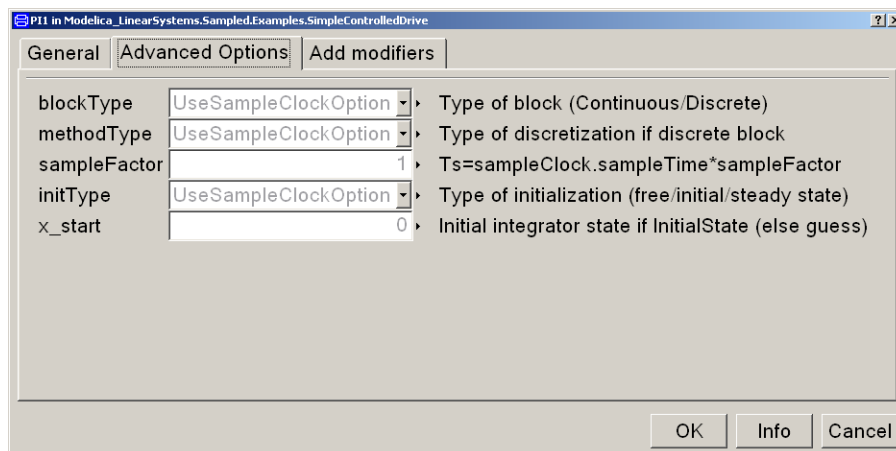
In the example of Figure 17.6, block `filter` is a continuous filter that filters the reference motion of the motor (here: ramp). For this reason, `filter` uses explicitly the `blockType` `Continuous` whereas all other blocks use the `blockType` `UseSampleClockOption`, i.e., they use the setting defined in `sampleClock`. By changing the `blockType` from `Continuous` to `Discrete` in block `sampleClock`, the controller is automatically transformed from a continuous to a discrete representation.

Every block of the *Sampled* library has a continuous input and a continuous output. Inside the respective block, the input and output signals might be sampled with the base sampling period defined in “`sampleClock`” or with an Integer multiple of it. For example, the PI controller in Figure 17.6 has the following parameter menu to define the continuous parameterization of the PI controller (i.e., gain  $k$  and time constant  $T$ ):

Figure 17.7: Parameter menu of *Sampled.PI* controller.

In the Advanced Options tab, see Figure 17.8, the remaining block settings are present, especially to define whether it is a continuous or a discrete block and in the latter case define also the discretization method. Since parameter `sampleFactor` is 1, the base sampling time of the `sampleClock` component is used. In other blocks of the controller (`Sampler1`, `Sampler2`), **parameter** `sampleFactor` = 5 which means that the inputs and outputs of these blocks are sampled by a sampling rate that is 5 times of the base sample time defined in the `sampleClock` component. Note, the `samplerX` blocks in Figure 17.6 have only an effect if the controller is discrete. For a continuous representation, these blocks just state that the output signal is identical to the input signal.

Figure 17.8: “Advanced Options” parameter menu.



Finally, with parameter `initType` the type of initialization can be defined by using either the setting from the `sampleClock` component (default) or a local definition. The default of the `sampleClock` component initialization is “steady state”. This means that continuous blocks are initialized so that the state derivatives are zero and discrete blocks are initialized so that a re-evaluation of the discrete equations gives the same discrete states (provided the input did not change). Since the equations of the blocks of the `Sampled` library are linear, the default setting leads to linear systems of equations during initialization that can be usually solved very reliably. The “steady state” initialization for a control system or a filter has the significant advantage that unnecessary settling times after simulation start are avoided.

### 17.3.2 Discretization Methods

The core of the `Sampled` library is function `DiscreteStateSpace.fromStateSpace` that transforms a linear, time invariant *differential equation* system in state space form:



$$\begin{aligned}\dot{\mathbf{x}}(t) &= \mathbf{A} \cdot \mathbf{x}(t) + \mathbf{B} \cdot \mathbf{u}(t) \\ \mathbf{y}(t) &= \mathbf{C} \cdot \mathbf{x}(t) + \mathbf{D} \cdot \mathbf{u}(t)\end{aligned}\tag{17.8}$$

into a linear, time invariant, *difference equation* system in state space form:

$$\begin{aligned}\mathbf{x}_d((k+1) \cdot T_s) &= \mathbf{A} \cdot \mathbf{x}_d(k \cdot T_s) + \mathbf{B} \cdot \mathbf{u}(k \cdot T_s) \\ \mathbf{y}(k \cdot T_s) &= \mathbf{C} \cdot \mathbf{x}_d(k \cdot T_s) + \mathbf{D} \cdot \mathbf{u}(k \cdot T_s) \\ \mathbf{x}(k \cdot T_s) &= \mathbf{x}_d(k \cdot T_s) + \mathbf{B}_2 \cdot \mathbf{u}(k \cdot T_s)\end{aligned}\tag{17.9}$$

where

- $t$  is the time
- $T_s$  is the sample time
- $k$  is the index of the actual sample instance ( $k = 0, 1, 2, \dots$ )
- $\mathbf{u}(t)$  is the input vector
- $\mathbf{y}(t)$  is the output vector
- $\mathbf{x}(t)$  is the state vector of the continuous system from which the discrete block has been derived.
- $\mathbf{x}_d(t)$  is the state vector of the discretized system

If the discretization method, e.g., the trapezoidal integration method, accesses actual and past values of the input  $\mathbf{u}$  (e.g.  $\mathbf{u}(T_s \cdot k)$ ,  $\mathbf{u}(T_s \cdot (k-1))$ ,  $\mathbf{u}(T_s \cdot (k-2))$ ), a state transformation is needed to arrive at the difference equation above where only the actual value  $\mathbf{u}(T_s \cdot k)$  is accessed.

If the original continuous state vector at the sample times shall be computed from the discrete equations, the matrices of this transformation have to be known. For simplicity and efficiency, library Modelica\_LinearSystems supports only the specific transformation as needed for the provided discretization methods: As a result, the state vector of the underlying continuous system can be calculated by adding the term  $\mathbf{B}_2 \cdot \mathbf{u}$  to the state vector of the discretized system. In fact, since the discrete state vector depends on the discretization method, it is a protected variable that cannot be accessed outside of the block. This vector is also not needed by the user of the block, because the continuous state vector is available both in the continuous and in the discrete case. In Modelica notation, the difference equation above is implemented as:

```
when {initial(), sample(Ts, Ts)} then
  new_x_d = A * x_d + B * u;
  y = C * x_d + D * u;
  x_d = pre(new_x_d);
  x = x_d + B2 * u;
end when;
```

Since no "next value" operator is available in Modelica, an auxiliary variable "new\_x\_d" stores the value of "x\_d" for the next sampling instant. The relationship between "new\_x\_d" and "x\_d" is defined via equation "x\_d = pre(new\_x\_d)".

The body of the **when**-clause is active during initialization and at the next sample instant  $t = T_s$ . Note, the **when**-equation is not active after the initialization at  $t = 0$  (due to **sample**(Ts, Ts) instead of **sample**(0, Ts)), since the state "x\_d" of the initialization has to be used also at  $t = 0$ . Additional equations are added for the initialization to uniquely compute vectors "x" and "x\_d" at the initial time:

```
initial equation
  if init == InitialState then
    x = x_start;
  elseif init == SteadyState then
    x_d = new_x_d;
  end if;
```

If option InitialState is set, the discrete state vector "x\_d" is computed such that the continuous state vector "x = x\_start", i.e., the continuous state vector is identical to the desired start value of the continuous state.

If option SteadyState is set, the discrete controller is initialized in steady state (= default setting in sampleClock). This means that the output  $\mathbf{y}(T_s \cdot k)$ ,  $k=0,1,2,\dots$ , remains constant provided the input vector  $\mathbf{u}(T_s \cdot k)$  remains constant. Most simulation systems support steady state initialization only for the continuous

part of a model. Due to the equation based nature of Modelica, where basically everything is mapped to equations, it is possible in Modelica to initialize also a mixture of continuous and discrete equations in steady state.

Via parameter `methodType` in the global block `sampleClock` or also individually for every block, the following discretization methods can be selected:

- *explicit Euler* integration method,
- *implicit Euler* integration method,
- *trapezoidal* integration method,
- *exact* solution under the assumption that the input signal is *piecewise constant*,
- *exact* solution under the assumption that the input signal is *piecewise linear*.

The last method (exact solution for piecewise linear input) gives nearly always the best approximation to the continuous solution without frequency or phase distortion. The effect of the discretization methods and of steady state initialization is demonstrated on the basis of a simple PT2 block that is initialized in steady state, where the input signal is 0.2 at the beginning and jumps to 1.2 at 0.5 s

In the next figure, the blue curve is the solution of the continuous block and the red curve is the solution of the digital block using the *implicit Euler* integration method for the block discretization. As can be seen, even for this simple block the results of the continuous and digital representation are quite different. A much smaller sample time would be needed here, in order that the solutions of the discrete and the continuous representations would be closer together.

Figure 17.9: Step response of PT2 block discretized with the implicit Euler method.

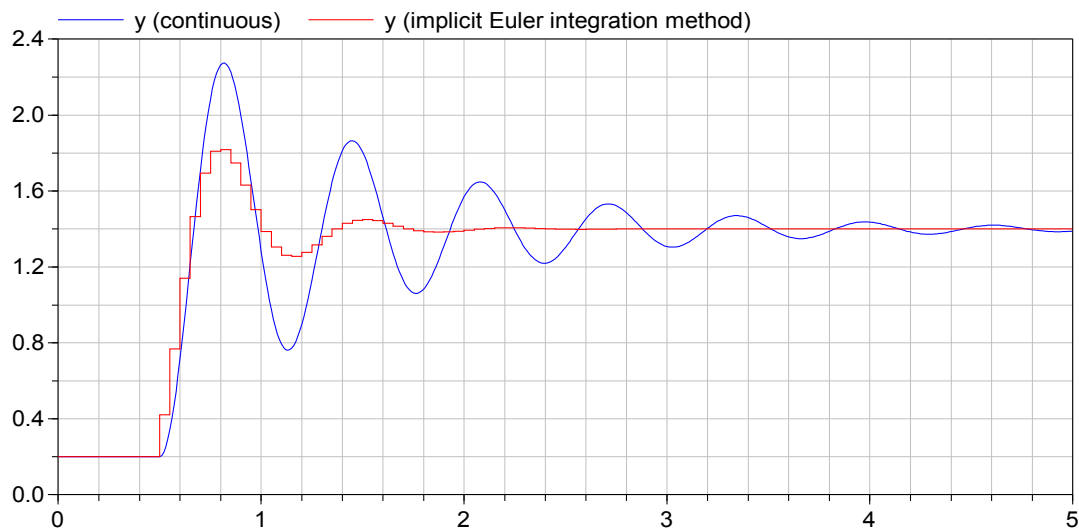
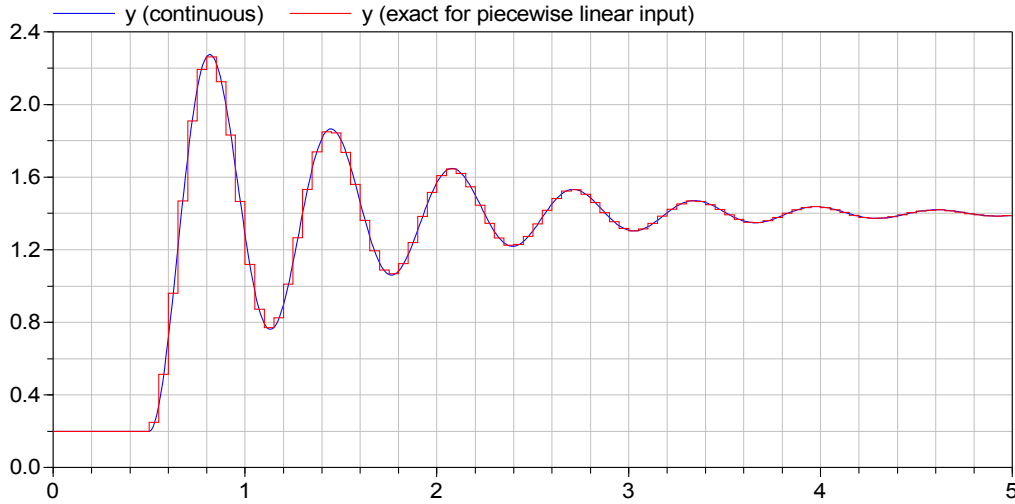


Figure 17.10: Step response of PT2 block discretized with the exact solution for piecewise linear input signals (method = *rampExact*).



In the next figure, the red curve is the solution of the digital block using the *rampExact* method (i.e. the exact solution under the assumption that the input signal is piecewise linear). The solutions of the continuous and of the discrete blocks are practically identical without any phase shift. A better solution cannot be expected.

Continuous models in `StateSpace` form are transformed into discrete systems according to the sketched discretization methods. A `TransferFunction` system description is implemented as state space system in controller canonical form.

The transformation of a `ZerosAndPoles` system is more involved in order to reduce numerical difficulties, especially for filter implementations: The basic approach is to transform the transfer function

$$y = k \cdot \frac{\prod (s + n_{1i}) \cdot \prod (s^2 + n_{2j} \cdot s + n_{3j})}{\prod (s + d_{1k}) \cdot \prod (s^2 + d_{2l} \cdot s + d_{3l})} \cdot u \quad (17.10)$$

into a connected series of first and second order systems. All these systems are implemented as small state space systems in controller canonical form that are connected together in series. The output is scaled such that the gain of every block is one (if this is possible) in order that the inputs and outputs of the blocks are in the same order of magnitude. The output of the last block is multiplied with a factor so that the original gain of the transfer function is recovered.

A simpler, alternative implementation for both the `TransferFunction` and the `ZerosAndPoles` system would have been to compute the **A**, **B**, **C**, **D** matrices of the corresponding state space system with available function calls and then use the general `StateSpace` model for their implementations. The severe disadvantage of this approach is that the structure of the state space system is lost for the symbolic preprocessing. If, e.g., index reduction has to be applied (e.g. since a filter is used to realize a non-linear inverse model), then the tool cannot perform the index reduction anymore. Example:

Assume, a generic first order state space system is present

$$\begin{aligned} \dot{x} &= a \cdot x + b \cdot u \\ y &= c \cdot x + d \cdot u \end{aligned} \quad (17.11)$$

and the values of the scalars  $a$ ,  $b$ ,  $c$ ,  $d$  are parameters that might be changed before the simulation starts. If  $y$  has to be differentiated symbolically during code generation, then

$$\begin{aligned} \dot{y} &= c \cdot \dot{x} + d \cdot \dot{u} \\ \dot{x} &= a \cdot x + b \cdot u \end{aligned} \quad (17.12)$$

As a result, the input  $u$  needs to be differentiated too, and this might not be possible and therefore translation

might fail.

On the other hand, if the first order system is defined to be a low pass filter and the state space system is generated by keeping this structure, we have:

$$\begin{aligned}\dot{x} &= -b \cdot x + u \\ y &= x\end{aligned}\tag{17.13}$$

Differentiating  $y$  symbolically leads to:

















$$\begin{aligned}\dot{y} &= \dot{x} \\ \dot{x} &= -b \cdot x + u\end{aligned}\tag{17.14}$$

Therefore, in this case, the derivative of  $u$  is not needed and the tool can continue with the symbolic processing.

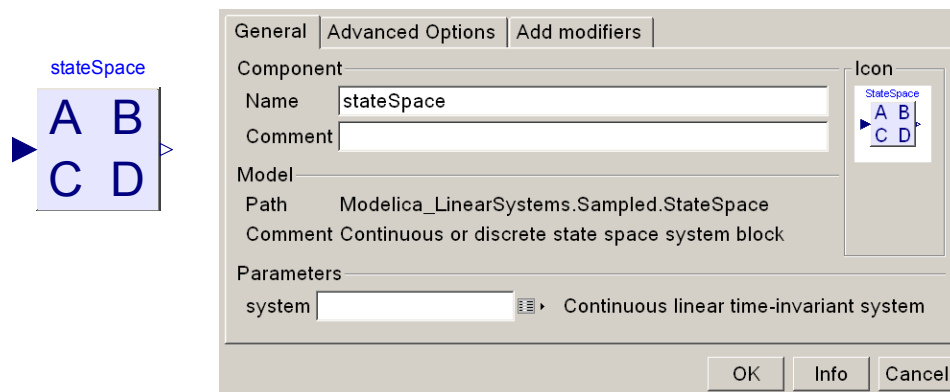
### 17.3.3 Block Components

A short description of the input/output blocks of the *Sampled* sublibrary is given in below.

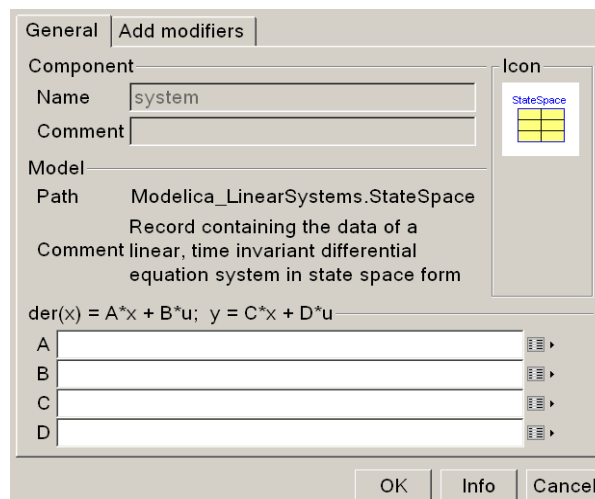
*Table 17.1: The blocks of the Modelica\_LinearSystems.Sampled library.*

Name	Description
 SampleClock	Global options for blocks of Sampled library (e.g. base sample time)
 StateSpace	Continuous or discrete state space system block
 TransferFunction	Continuous or discrete, single input single output transfer function
 ZerosAndPoles	Continuous or discrete, single input single output block described by zeros and poles
 Filter	Analog low or high pass IIR-filters (CriticaDaming/Bessel/Butterworth/ Chebyshev)
 FilterFIR	Discrete finite impulse response (low or high pass) filters
 Integrator	Output the integral of the input signal (continuous or discrete block)
 Derivative	Approximate derivative (continuous or discrete block)
 FirstOrder	First order (continuous or discrete) transfer function block (= 1 pole)
 SecondOrder	Second order (continuous or discrete) transfer function block (= 2 poles)
 PI	Proportional-Integral controller (continuous or discrete block)
 UnitDelay	Delay the input by a multiple of the base sample time if discrete block or $y = u$ if continuous block
 Sampler	Sample the input signal if discrete block or $y = u$ if continuous block
 ADconverter	Analog to digital converter (including sampler)
 DAconverter	Digital to analog converter (including zero order hold)
 Noise	Generates uniformly distributed noise in a given band at sample instants if discrete and $y = 0$ if continuous

As an example block `Sampled.StateSpace` is shortly described. Its icon is shown in the figure at the left. When double clicking on the block, the parameter menu shown in Figure 17.11 pops up.

Figure 17.11: Parameter menu of *Sampled.StateSpace* block.

Here, either the name of a StateSpace record can be given, or when clicking on the “table” symbol at the right end of the input field another menu pops up in which the 4 matrices can be defined, as shown in Figure 17.12.

Figure 17.12: Parameter menu of *StateSpace* record.

By clicking again on the “table” symbol on the right side of one of the input fields, a matrix editor appears that allows to conveniently define the corresponding matrix.

Some of the blocks of the *Sampled* library have only a pure discrete representation, such as the *FilterFIR*, *UnitDelay*, *Sampler*, *ADconverter*, *DAconverter* and *Noise* blocks. The continuous representation of these blocks is defined to be “ $y = u$ “, i.e., the output is identical to the input which means that these components are effectively removed in the continuous mode.

## Chapter 18 Hierarchical State Machines (StateGraph Library)

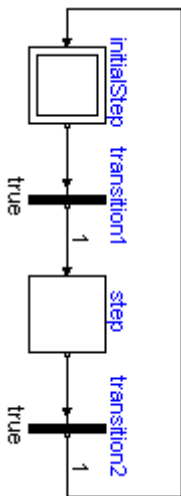
In many applications a user would like to define a discrete algorithm graphically as a finite state machine, e.g., as a Petri net, a Sequential Function Chart (SFC) or a Statechart. In this chapter it is described, how certain classes of finite state machines can be defined in Modelica. In general, only formalisms can be implemented conveniently in Modelica that are deterministic and have a “black box” module behavior. This means, e.g., that Statecharts cannot be implemented in a convenient way because Statecharts may have transitions over several hierarchies and the hierarchical layers are not “hidden” in a “black box” but are visible on the highest layer.

In the Modelica Standard Library, there is a sublibrary called `Modelica.StateGraph` that is used to model hierarchical finite state machines. It is based on SFCs but with the extensions from JGraphcharts (Arzen et.al. 2002) so that the modeling power is similar to Statecharts. This library should be used to model finite state machines in Modelica, if there are no specific reasons to do it differently.

In this section the components of the `StateGraph` library are introduced by examples to show how it can be used in applications.

Currently a new state machine library, called `ModeGraph`, is under development that shall be much better as the `StateGraph` library by including new features in Modelica and Dymola to be able to get rid of the drawbacks of current state machine handling in Modelica.

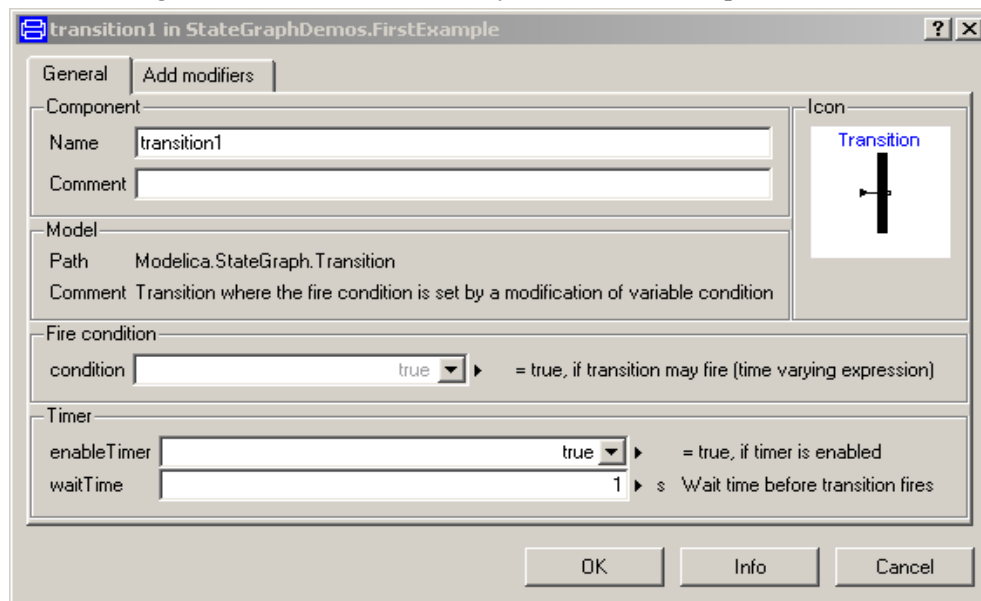
### 18.1 Steps and Transitions



The basic elements of `StateGraphs` are *steps* and transitions as shown in the figure to the left. *Steps* represent the possible states a `StateGraph` can have. If a step is *active* the Boolean variable `active` of the step is **true**. If it is deactivated, `active` = **false**. At the initial time, all ordinary steps are deactivated. The `InitialStep` objects are steps that are activated at the initial time. They are characterized by a double box (see figure at the left).

Transitions are used to change the state of a `StateGraph`. When the step connected to the input of a transition is active, and when the transition condition becomes **true**, then the transition *fires*. This means that the step connected to the input to the transition is deactivated and the step connected to the output of the transition is activated. The transition condition is defined via the parameter menu of the transition object. Clicking on object "transition1" in the figure to the left, results in the parameter menu shown in next figure:

Figure 18.1: Parameter menu of “Transition” component.

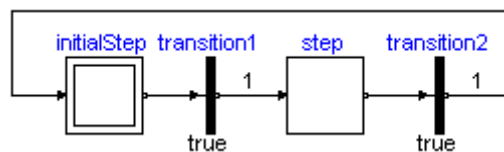


In the input field "condition", any type of time varying Boolean expression can be given (in Modelica notation, this is a modification of the time varying variable `condition`). Whenever this condition is **true**, the transition can fire. Additionally, it is possible to activate a *timer*, via `enableTimer` (see menu above) and provide a `waitTime`. In this case the firing of the transition is delayed according to the defined `waitTime`. The transition only fires if the condition remains **true** during the `waitTime`. The transition condition and the `waitTime` are displayed in the transition icon.

In the above example, the simulation starts at `initialStep`. After 1 second, `transition1` fires and `step1` becomes active. After another second `transition2` fires and `initialStep` becomes again active. After a further second `step1` becomes active, and so on.

In Grafchart, Grafcet and SFC the network of steps and transitions is drawn from top to bottom. In StateGraphs, no particular direction is defined, since Modelica models do not depend on the placement of components and connection lines. Usually, it is more practical to define the network from left to right, since it is easier to read the labels on the icons. The example from above has then the following layout:

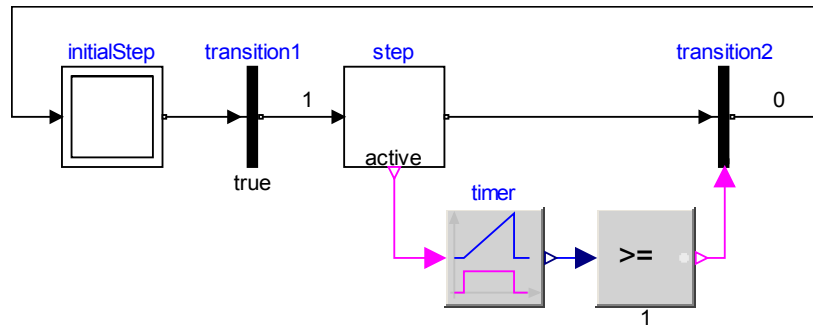
Figure 18.2: First StateGraph example drawn from “left” to “right”.



## 18.2 Conditions and Actions

With the block `TransitionWithSignal`, the firing condition can be provided as Boolean input signal, instead as entry in the menu of the transition with block `Transition`, see example in the next figure:

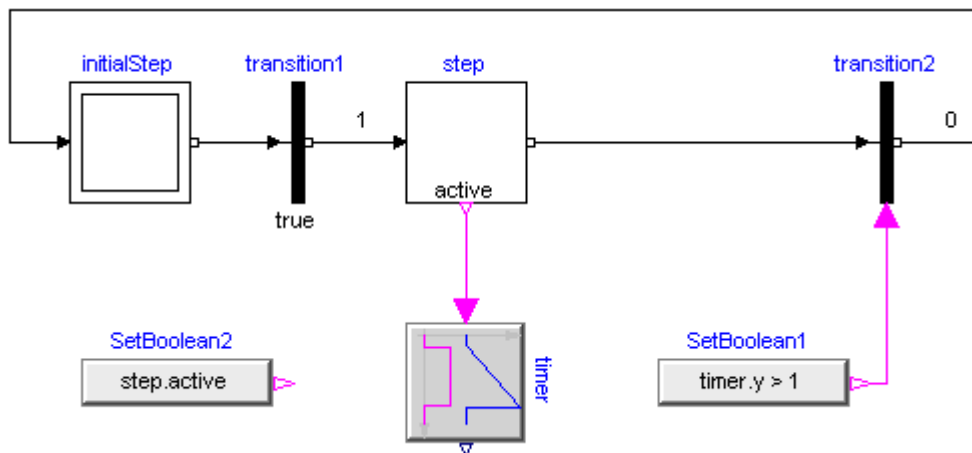
Figure 18.3: Example of a StateGraph with StepWithSignal and TransitionWithSignal components.



Component "step" is an instance of "StepWithSignal" that is a usual step where the active flag is available as Boolean output signal. To this output, component "Timer" from library "Modelica.Blocks.Logical" is connected. It measures the time from the time instant where the Boolean input (i.e., the active flag of the step) became **true** up to the current time instant. The timer is connected to a comparison component. The output is **true**, once the timer reaches 1 second. This signal is used as condition input of the transition. As a result, "transition2" fires, once step "step" has been active for 1 second. Of course, any other Modelica block with a Boolean output signal can be connected to the condition input as well, especially blocks of the Modelica.Blocks.Logical library.

Instead of using logical blocks, via the Modelica.Blocks.Sources.SetBoolean component any type of logical expression can be defined in textual form, as shown in the next figure:

Figure 18.4: Example demonstrating the use of the SetBoolean component.



With the block "SetBoolean", a time varying expression can be provided as modification to the output signal *y* (see block with icon text "timer.y > 1" in the figure above). The output signal can be in turn connected to the condition input of a TransitionWithSignal block.

The "SetBoolean" block can also be used to compute a Boolean signal depending on the active step. In the figure above, the output of the block with the icon text "step.active" is **true**, when "step" is active, otherwise it is **false** (note, the icon text of "SetBoolean" displays the modification of the output signal "y"). This signal can then be used to compute desired actions in the physical systems model. For example, if a valve shall be open, when the StateGraph is in "step1" or in "step4", a "SetBoolean" block may be connected to the valve model using the following condition:

```
step1.active or step2.active
```

Via the Modelica operators `edge(...)` and `change(...)`, conditions depending on rising and falling edges of Boolean expressions can be used when needed.

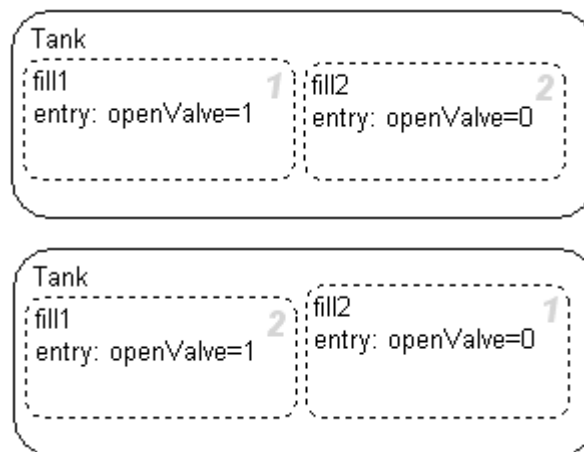
In Grafchart, Grafcet, SFC and Statecharts, actions are formulated within a step. Such actions are distinguished as entry, normal, exit and abort actions. For example, a valve might be opened by an entry action of



a step and might be closed by an exit action of the same step. In StateGraphs this is not possible due to Modelica's "single assignment rule" that requires that every variable is defined by exactly one equation. Instead, the approach explained above is used. For example, via the "SetBoolean" component, the valve variable is set to **true** when the StateGraph is in particular steps.

This feature of a StateGraph is very useful, since it allows a Modelica translator to guarantee that a given StateGraph has always deterministic behavior without conflicts. In the other methodologies this is much more cumbersome. As an example, in the next figure a critical situation in Stateflow is shown (Mathworks Stateflow is a Statechart type state machine and is integrated in Mathworks Simulink):

*Figure 18.5: Parallel execution in Stateflow, where a variable is assigned a value in parallel branches. The upper definition results in "openValve=0" whereas the lower definition results in "openValve=1".*



The two substates "fill1" and "fill2" are executed in parallel. In both states the variable "openValve" is set as entry action. The question is whether openValve will have value 0 or 1 after execution of the steps. Stateflow changes this non-deterministic behavior to a formally deterministic one by defining an execution sequence of the states that depends on the graphical position. The light number on the right of the states shows in which order the states are executed. In the upper part of Figure 18.5 this means that "openValve=0" after execution of the parallel states.

In the lower part of the figure, the second state "fill2" is changed a little bit graphically and then "openValve=1" after "fill1" and "fill2" have been executed. This is a dangerous situation because (a) slight changes in the placement of states might change the simulation result and (b) if the parallel execution of actions depends on the evaluation order, errors are difficult to detect.

Similar problems occur in other state machine formalisms, such as SFC, Grafcet and Graphcharts: Variables are changed according to an evaluation sequence of the simulator. It seems not possible to provide an easy-to-grasp rule about evaluation order of actions that are executed in parallel. Therefore, either the simulator just uses an internal evaluation order, or non-obvious rules are present as in Stateflow that do not solve the underlying problem.

In a StateGraph, such a situation is detected by the translator resulting in an error, since there are two equations to compute one variable. The user is forced to reformulate the state machine by explicitly defining priorities. For example, if "fill1" and "fill2" are steps that are executed in parallel, there might be a "Set-Boolean" block that defines:

```
openValve = if fill1.active then 1
            else (if fill2.active then 0 else 2);
```

Therefore step fill1 has a higher priority as step fill2.

In a Statechart or Graphchart it is difficult to modularize a sub chart if the used actions reference variables in an outer scope: Assume, for example, that a state machine "control" has the following hierarchy:

```
control.superstate1.step1
```

Within "step1", a Statechart would, e.g., access variable "control.openValve", say as "entry action:

`openValve = true`". This typical usage has the drawback that it is difficult to use the hierarchical state "superstate1" as component in another context, because "step1" references a particular name outside of this component.

In a StateGraph, there would be typically a "SetBoolean" component in the "control" component stating:

```
openValve = superstate1.step1.active;
```

As a result, the "superstate1" component can be used in another context, because it does not depend on the environment where it is used.

The disadvantage of the StateGraph approach is that the user might not be able to formulate the network directly as desired. For example, in order to fill a tank usually several actions are necessary, e.g., to close one valve and to open another one. In a SFC all actions to "fill a tank" would be defined as actions to a "fill\_a\_tank" step and this might be more convenient for the user. For example, copying or deleting a "fill\_a\_tank" step would require only a change at one place in a SFC whereas it would require changes at several places in a StateGraph.

### 18.2.1 Example:

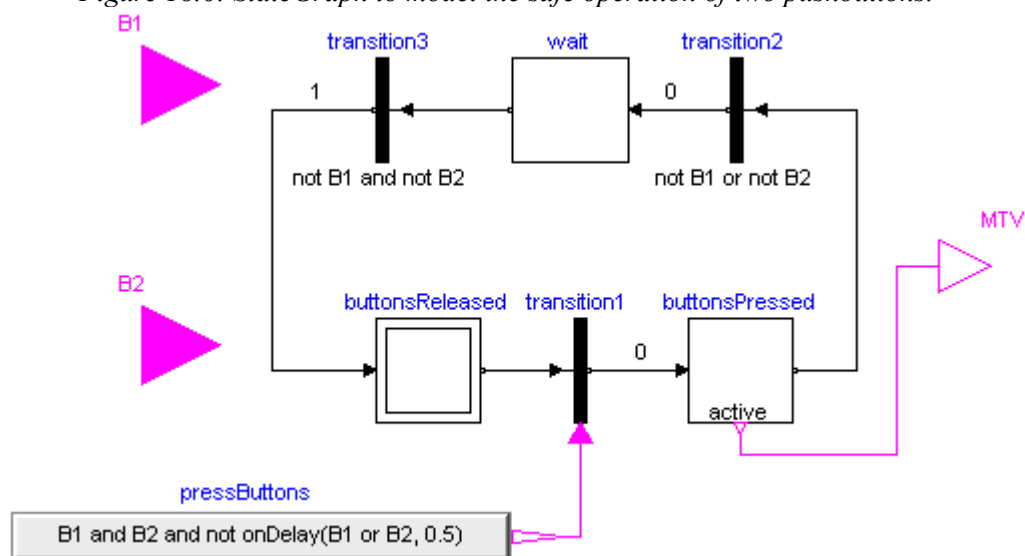
In (Roussel and Faure, 2002), an example of a simple safety-related program is given that is verified with a new method. The implementation is shown as a PLC program with function blocks from IEC 61131-3 and a newly introduced language. In both cases the implementation is difficult to understand because flip-flops are used that are very low level description elements. This example shall be modeled with a StateGraph. The specification given in the article is:

"The aim of this program is to monitor the safe operation of two pushbuttons used to operate presses and similar dangerous machinery. It ensures that both hands of an operator are kept outside the danger zone during machine operation. The required properties are:"

- (1) A cycle can only be initiated by pressing the two pushbuttons simultaneously (within 0.5 s).
- (2) A cycle is interrupted by releasing one or both buttons to stop the output.
- (3) The output signal can only be re-initiated after both inputs have been released and the pushbuttons are operated again.

The implementation with a StateGraph is shown in the next figure (under the assumption that the "onDelay(. . .)" block/operator would be available, see page 123):

Figure 18.6: StateGraph to model the safe operation of two pushbuttons.



As can be seen,

- property (1) to be proved ("A cycle can only be initiated by pressing the two pushbuttons simultaneously

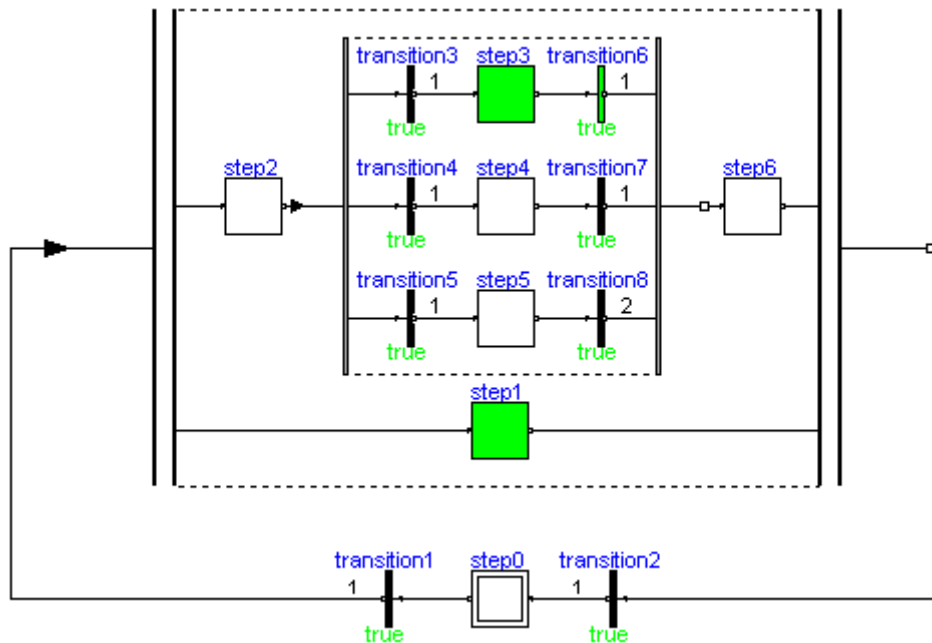
- with 0.5s”) is used as condition of “transition1”,
- property (2) (“A cycle is interrupted by releasing one or both buttons to stop the output”) is used as condition of “transition2”, and
- property (3) (“The output signal can only be re-initiated after both inputs have been released and the pushbuttons are operated again”) is used as condition of “transition3”. At transition3 also an explicit wait time of 1 s is added to enforce that the machine is really coming to a halt, before the buttons can be pressed again.

This program is so simple and obvious that verification of the properties (1), (2), (3) seems not to be necessary.

### 18.3 Parallel and Alternative Execution

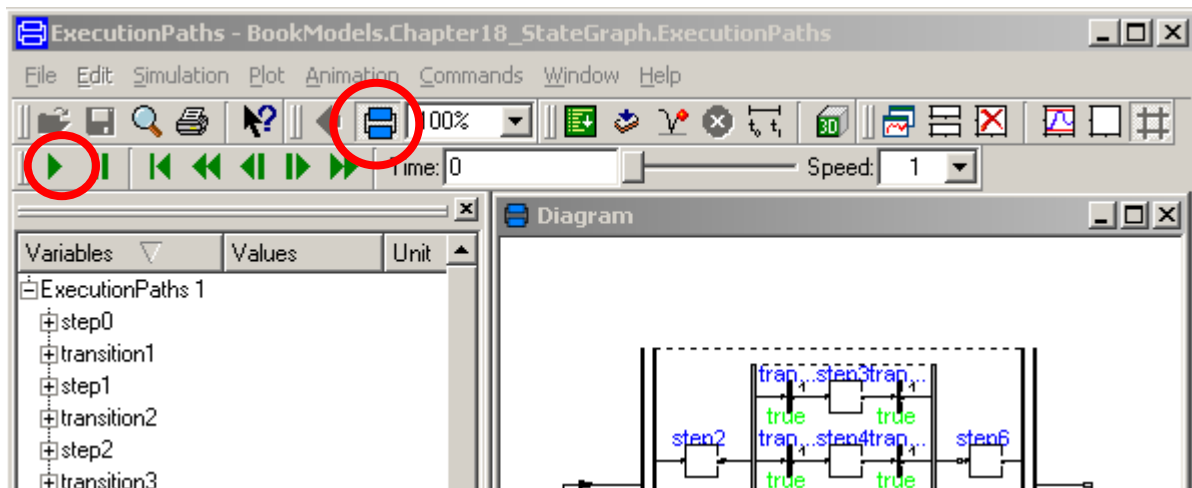
*Parallel* activities can be defined by component `StateGraph.Parallel` and *alternative* activities can be defined by component `StateGraph.Alternative`. An example for both components is given in the next figure.

Figure 18.7: Example for parallel and alternative execution paths in a StateGraph.



Here, the branch from "step2" to "step6" is executed in parallel to "step1". A transition connected to the output of a parallel branch component can only fire if the final steps in all parallel branches are active simultaneously. The figure above is a screen-shot from the diagram animation of the `StateGraph` model in `Dymola`: Whenever a step is active or a transition can fire, the corresponding component is marked in green color. This diagram animation is automatically available after a simulation has been performed, once the “Diagram” and the “Runs the animation” button in the tool bar of the simulation window have been pressed, as sketched in the next figure:

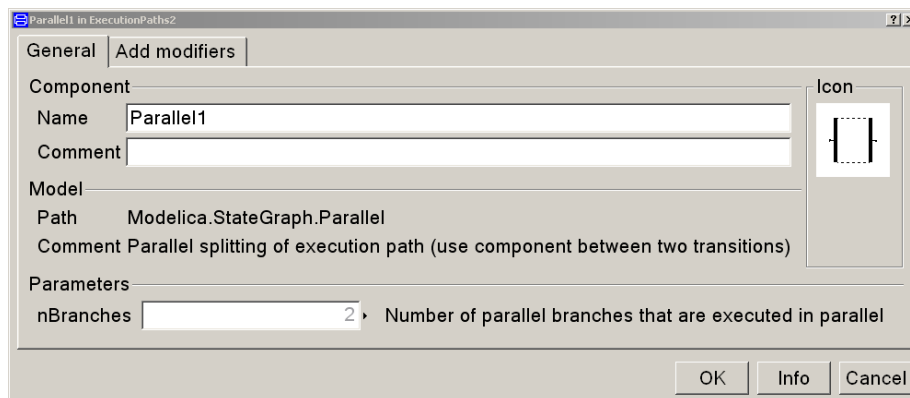
Figure 18.8: Diagram animation in Dymola by selecting the "Diagram" button.



The three branches within "step2" to "step6" are executed alternatively, depending which transition condition of "transition3", "transition4", "transition5" fires first. Since all three transitions fire after 1 second, they are all candidates for the active branch. If two or more transitions would fire at the same time instant, a priority selection is made: The alternative and parallel components have a vector of connectors. Every branch has to be connected to exactly one entry of the connector vector. The entry with the lowest index has the *highest priority*.

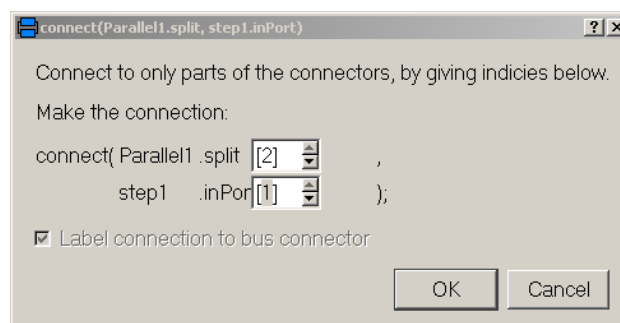
Parallel, Alternative and Step components have vectors of connectors. The dimensions of these vectors are set in the corresponding parameter menu. For example in a "Parallel" component:

Figure 18.9: Defining the number of branches in a Parallel component.



Currently, in Dymola the following menu pops up when a branch is connected to a vector of components in order to define the vector index to which the component shall be connected:

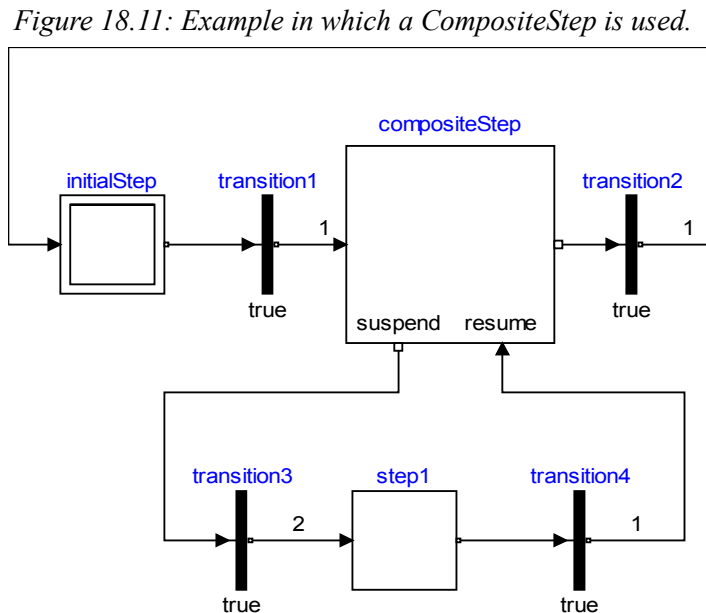
Figure 18.10: Menu to define the vector elements that are connected.



This is inconvenient. There have been discussions to improve the Modelica language to handle such situations more conveniently.

## 18.4 Composite Steps

A StateGraph can be *hierarchically* structured by using a component that *inherits* from StateGraph.: *PartialCompositeStep*. An example is given in the next figure:



The CompositeStep component contains a local StateGraph that is entered by one or more input transitions and that is left by one or more output transitions. Also, other needed signals may enter a CompositeStep. The CompositeStep has similar properties as a "usual" step: The CompositeStep is *active* once at least one step within the CompositeStep is active. Variable active defines the state of the CompositeStep.

Additionally, a CompositeStep has a suspend port. Whenever the transition connected to this port fires, the CompositeStep is left at once. When leaving the CompositeStep via the suspend port, the internal state of the CompositeStep is saved, i.e., the active flags of all steps within the CompositeStep. The CompositeStep might be entered via its resume port. In this case the internal state from the suspend transition is reconstructed and the CompositeStep continues the execution that it had before the suspend transition fired (this corresponds to the history ports of Statecharts or JGfcharts).

A CompositeStep may contain other CompositeSteps. At every level, a CompositeStep and all of its content can be left via its suspend ports (actually, there is a vector of suspend connectors, i.e., a CompositeStep might be aborted due to different transitions).

The CompositeStep can be used in the same way as a superstate in Statecharts. In a superstate it is possible to enter the state in different ways ending up in different internal states. This can be modeled in a StateGraph or a Graphchart by having multiple input transitions, each leading to a different internal step. In a superstate it is possible to exit a superstate in different ways depending on which internal state that is active. This is modeled in a StateGraph or Graphchart by associating different output transitions to the different internal steps. In a superstate it is, finally, also possible to exit the state independently from which internal state that is active. This is achieved with the suspend port here. The conditions connected to the transitions attached to the suspend port can also be conditioned by the status of the internal steps of the CompositeStep. In this way it is possible to suspend the step if a certain condition holds and unless a certain internal step is active. The history arcs in Statecharts correspond to the resume port. Superstates with parallel subparts, so called XOR superstates, can be modeled using parallel constructs inside the CompositeStep.

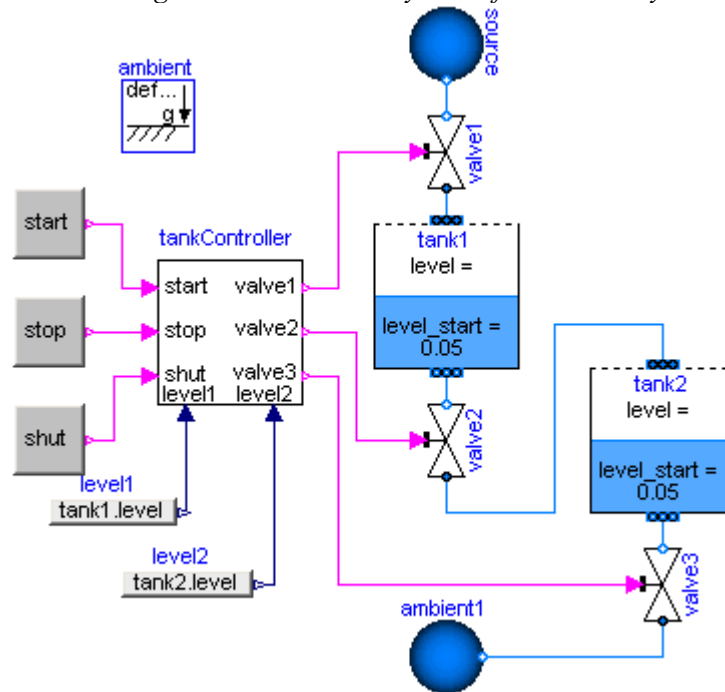
In addition to using CompositeSteps for modeling hierarchical states they can also be used to simply

aggregate a part of a larger `StateGraph`. This can be useful to improve the structure.

### 18.4.1 Example:

The figure below is a screen shot of one of the examples of the `Modelica_Fluid` library:

Figure 18.12: Control system of a two tank system.



Two tanks are present that are controlled by the “`tankController`” which in turn is controlled by 3 push buttons. The basic operation is to fill and empty the two tanks:

1. Valve 1 is opened and tank 1 is filled.
2. When tank 1 reaches its fill level limit, valve 1 is closed.
3. After a waiting time, valve 2 is opened and the fluid flows from tank 1 into tank 2.
4. When tank 1 reaches its minimum level, valve 2 is closed.
5. After a waiting time, valve 3 is opened and the fluid flows out of tank 2
6. When tank 2 reaches its minimum level, valve 3 is closed

The above "normal" process can be influenced by three buttons:

1. Button “`start`” starts the above process. When this button is pressed after a “`stop`” or “`shut`” operation, the process operation continues.
2. Button “`stop`” stops the above process by closing all valves. Then, the controller waits for further input (either “`start`” or “`shut`” operation).
3. Button “`shut`” is used to shutdown the process, by emptying at once both tanks by opening valve 2 and valve 3. When this is achieved, the process goes back to its start configuration where all 3 valves are closed. Clicking on “`start`” restarts the process.

A safe operation requires that the tanks never overflow. One approach could be to state this proof goal formally and then proof that the implementation of the above operations is always safe. An alternative approach seems to be to define how a safe operation can be guaranteed and implement this directly. The following rules guarantee that the tanks never overflow:

1. When tank1 reaches its fill limit, valve1 is closed, so that no new fluid enters tank1.

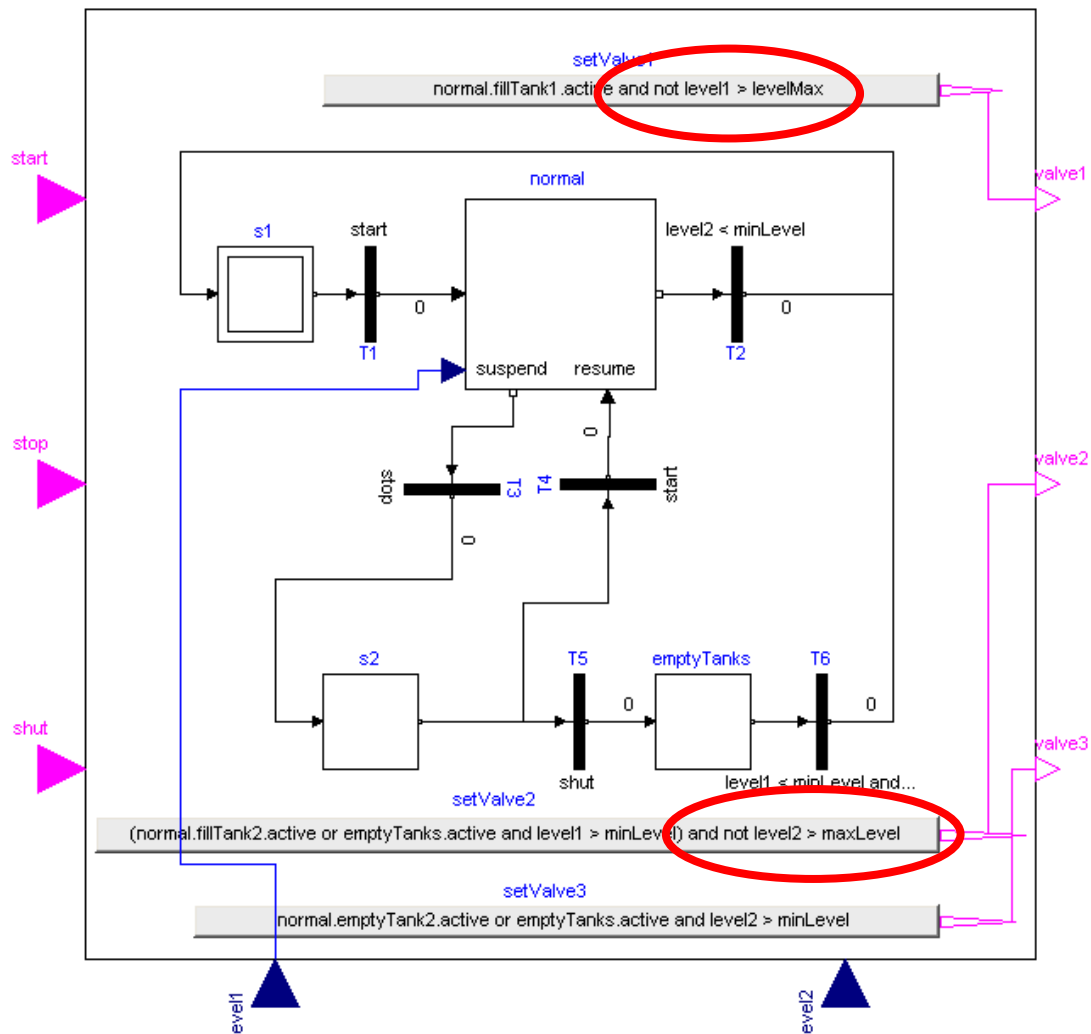
2. When tank2 reaches its fill limit, valve2 is closed, so that no new fluid enters tank2

Due to the “single assignment” rule in Modelica, there must be exactly one equation for valve1 and for valve2 respectively. It is then sufficient to define this equation, e.g., for valve1, in the following form:

valve1 = (<normal operation>) **and not** level1 > maxLevel

By this implementation it is guaranteed that the tank never overflows independently how the “normal” operation is implemented. This implementation is shown in the next figure.

Figure 18.13: Tank controller with safety rule so that tanks never overflow.



If one of the safe guards (“**and not** level2 > maxLevel”) becomes active, it could be that the state machine for the “normal” operation “hangs” at some “step”, since a transition can no longer fire: For example, tank2 might be filled until the transition “level1 < minLevel” fires (instead of “level1 < minLevel **or** level2 > maxLevel”). This transition will never fire, if valve2 is closed due to the safeguard (“valve2 = (...) **and not** level2 > maxLevel”). In the current case this is uncritical because by clicking on button “stop” or “shut” the composite step is terminated, independently which step was active. Still, it would be useful to detect such a situation that the state machine can “hang” at a step because a transition does no longer fire (this would be detected by a reachability analysis, that must be, however somehow defined which is currently outside of the scope of Modelica).

## 18.5 Execution Model

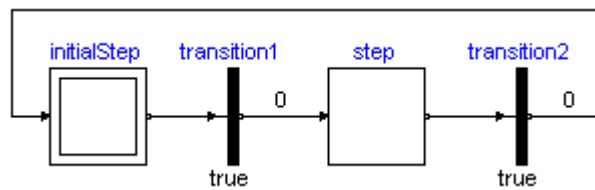
The execution model of a StateGraph follows from its Modelica implementation: Given the states of all steps, i.e., whether a step is active or not active, the equations of all steps, transitions, transition conditions,

actions etc. are sorted resulting in an execution sequence to compute essentially the new values of the steps. If conflicts occur, e.g., if there are more equations as variables, or if there are algebraic loops between Boolean variables, an error occurs. Once all equations have been processed, the active variables of all steps are updated to the newly calculated values. Afterwards, the equations are again evaluated. The iteration stops, once no step changes its state anymore, i.e., once no transition fires anymore. Then, simulation continues until a new event is triggered, i.e., when a relation changes its value.

## 18.6 Infinite, Unsafe and Unreachable StateGraphs

It is possible to define networks with the StateGraph library that should not be allowed and should lead to an error, since they give rise to infinite (instantaneous) looping, or since they are unsafe or unreachable. For example, the following StateGraph

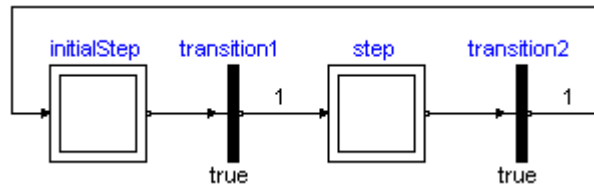
Figure 18.14: Erroneous StateGraph (infinite looping).



starts at the “initialStep”, fires at once at “transition1”, goes to “step”, fires at once at “transition2”, goes to “initialStep” and continues. Therefore, at a given time instant, the state machines “loops” forever.

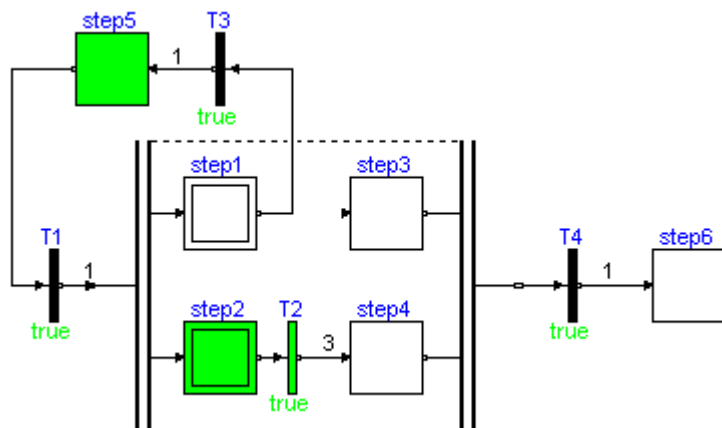
Another unwanted situation is the case, e.g., if two initial steps are introduced although only one should be present:

Figure 18.15: Erroneous StateGraph (two initial steps).



It is not predictable what will occur in such cases. Still another erroneous situation is displayed in the following figure, where the transition from step1 to step5 goes out of the parallel branches which should not be allowed.

Figure 18.16: Erroneous StateGraph: Branching out of a parallel execution component.



With the current StateGraph library, all the cases above can be modeled without getting an error message.



The behavior is mostly undefined and the StateGraph will not work as expected.

As shortly mentioned in the introduction of this chapter, a new state machine Modelica library, called ModeGraph, is currently under development. By suitable enhancements of Modelica it is possible that either by construction unsafe state machines can not be build, or a translation error occurs. The new methodology has also significant other advantages.

## Chapter 19 Nonlinear Inverse Models for Advanced Controllers

The subject of this chapter is the systematic design of controllers for non-linear systems, based on inversion of the non-linear plant model. This chapter is based on the article (Thümmel et. al. 2005). Traditional design techniques require the nonlinear plant model to be linearized around a stationary operating point. Afterwards linear methods may be applied to synthesize a controller for this operating point. In order to make this controller work over the full operating range of the plant, robust design techniques and/or gain scheduling are applied. The first approach may considerably reduce achievable performance if the plant dynamics vary strongly over the operating range, whereas the latter may involve designing many controllers at a grid of operating points and finding an interpolation scheme in between them.

In linear design, inversion of plant dynamics is sometimes used to compensate for coupled input / output responses, or as an easy way to impose specific dynamic behavior of the closed-loop system (Enns et. al. 1994). In a nonlinear context, the application of model inversion additionally provides compensation of nonlinear dynamic behavior of the plant. This is exploited in design techniques such as feedback-linearization (Slotine and Li 1991).

The design approach proposed in this chapter starts from any controller structure that is based on a linear inverse model of the plant. This model is replaced with a nonlinear inverse one, resulting in a controller that is valid for the full operating range of the plant. In case the plant model is available in Modelica, it will be demonstrated that inversion can be performed automatically, exploiting symbolic algorithms and code generation features of a Modelica simulation environment as discussed in Chapter 12. This often allows for an automated design process that directly results in nonlinear controllers that work in all operating conditions of the plant, avoiding the need for gain scheduling.

### 19.1 Inversion of Linear SISO Models

The goal is to use a nonlinear plant model in a controller in order that the nonlinearities of the plant are directly taken care of in the control system. Some basic properties to use inverse models in controllers for linear, single-input, single-output plants are recapitulated, as needed for the rest of the chapter. This section is based on the fundamental article of (Kreisselmeier 1999).

A single input, single output plant shall be described by a rational transfer function of the form

$$y = P(s) \cdot u = \frac{n_p(s)}{d_p(s)} \cdot u \quad (19.1)$$

where  $u(s)$  is the plant input or actuator signal,  $y(s)$  is the plant output,  $P(s)$  is the plant transfer function, consisting of the numerator polynomial  $n_p(s)$  and the denominator polynomial  $d_p(s)$ . The two polynomials  $n_p(s)$  and  $d_p(s)$  shall have no zero in common. All following properties hold both for continuous and discrete linear systems, i.e., the independent variable can either be the derivative operator “s” or the shift operator “z”. For simplicity, we only discuss the continuous case. The “relative degree”  $D$  is defined as:

$$D_p = \deg(d_p) - \deg(n_p) \quad (19.2)$$

where  $\deg(p)$  is the degree of polynomial  $p$ . We assume that the plant is causal, i.e.,  $D_p \geq 0$  (which means that the transfer function is proper). A polynomial  $p$  is called *stable* if all zeros of the polynomial are inside the left half part of the complex plane, i.e.,  $\text{Re}(\text{zeros}(p)) < 0$ . Otherwise the polynomial is called *unstable*. A *transfer function*  $P$  is called *stable* if all zeros of the denominator polynomial  $d_p$  are inside the left half part of the complex plane, i.e.,  $\text{Re}(\text{zeros}(d_p)) < 0$ . A polynomial  $p$  can be factorized in a product of the form

$$p = p^+ \cdot p^- \quad (19.3)$$

where  $p^+$  contains all elementary divisors that are unstable and  $p^-$  contains all elementary divisors that are stable. If a polynomial has no unstable zeros,  $p^+ = 1$ . The zeros of the numerator polynomial  $n_p(s)$  of a transfer function are also called “the zeros of the transfer function” and the zeros of the denominator polynomial  $d_p(s)$  of a transfer function are also called “the poles of the transfer function”. A transfer function is called “minimum phase”, if the numerator polynomial is stable. A transfer function is called “non-minimum phase” if the numerator polynomial is unstable.

The above well known standard definitions are demonstrated by means of the following non-minimum phase system:

$$y = \frac{(s+1) \cdot (s-2)}{s \cdot (s+3) \cdot (s+4) \cdot (s-5)} \cdot u$$

stable zeros:	$s = -1$	$n_p^+ = (s-2)$
unstable zeros:	$s = 2$	$n_p^- = (s+1)$
stable poles:	$s = -3, s = -4$	$d_p^+ = s \cdot (s-5)$
unstable poles:	$s = 0, s = 5$	$d_p^- = (s+3) \cdot (s+4)$
		$D_p = 2$

### 19.1.1 Open-loop controllers for stable, minimum phase plants

Our first goal is to design an open-loop controller  $F(s)$  for a causal, stable and minimum phase plant  $P(s)$ . This can be achieved with the simple controller structure of Figure 19.1:

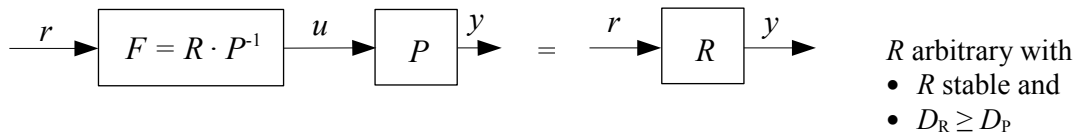


Figure 19.1: Open-loop controller  $F$  for a stable, minimum phase plant  $P$

where  $r$  is the reference signal, i.e., the desired value for output  $y$ . The open-loop controller  $F$  is connected in series to plant  $P$  and is designed by inverting the plant and multiplying it with an arbitrary transfer function  $R$  which has the only restriction that controller  $F$  must be stable and causal in order to be realizable and in order that the overall system is stable. Since  $P$  is minimum phase by assumption, plant  $P$  has only stable zeros and therefore

$$P = \frac{n_p^-}{d_p^-} \Rightarrow P^{-1} = \frac{d_p^-}{n_p^-}, \quad D_{P^{-1}} = -D_P \quad (19.4)$$

that is, the inverse plant model is stable and its relative degree is  $-D_P$ . In order that  $F$  is stable and causal, and since

$$F = \frac{n_R \cdot d_p^-}{d_R \cdot n_p^-} \quad (19.5)$$

$R$  must be stable and  $D_F = D_R + D_{P^{-1}} = D_R - D_P \geq 0$ . Therefore, the relative degree of  $R$  must be identical or larger as the relative degree of plant  $P$  resulting in  $D_R \geq D_P$ . Since  $R$  is the reference transfer function, the structure of Figure 19.1 is a very simple way to design an open-loop controller by just providing the desired reference transfer function and multiplying it with the inverse plant model. How to provide a suitable desired reference transfer function  $R$  is discussed at the end of this section on page 207.

### 19.1.2 Open loop controllers for stable, non-minimum phase plants

We will now analyze open-loop controllers  $F(s)$  for causal and stable plants  $P(s)$  that might be non-minimum phase. This can be achieved with the controller structure of Figure 19.2:

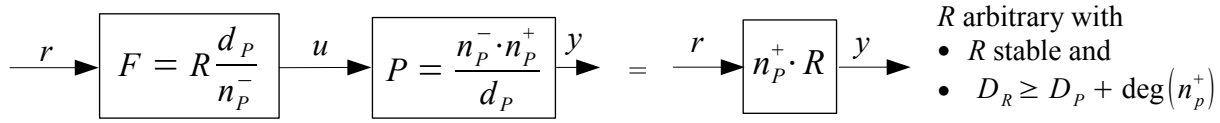


Figure 19.2: Open-loop controller  $F$  for a stable plant  $P$  ( $P$  might be non-minimum phase)

For a non-minimum phase plant  $P$ , only the stable part of  $P^{-1}$  can be utilized in the open-loop controller because otherwise the unstable zeros of  $P$  would become unstable poles of  $F$  and then  $F$  would be unstable. As a result, the unstable zeros  $n_P^+$  of  $P$  remain in the reference transfer function and therefore the restrictions on the design part  $R$  of  $F$  are larger:

$$D_F = D_R - (D_P + \deg(n_P^+)) \geq 0 \Rightarrow D_R \geq D_P + \deg(n_P^+) \quad (19.6)$$

Still, the construction of the open-loop controller is conceptually very simple, because the controller is defined by multiplying the freely designable part  $R$  of the desired reference transfer function with the stable part of the plant inverse. Example:

$$P = \frac{(s-1) \cdot (s+2)}{(s+3) \cdot (s+4) \cdot (s+5)}, \quad F = \frac{-(s+3) \cdot (s+4) \cdot (s+5)}{(s+2) \cdot (T_1 \cdot s+1) \cdot (T_2 \cdot s+1)} \Rightarrow P \cdot F = \frac{-(s-1)}{(T_1 \cdot s+1) \cdot (T_2 \cdot s+1)} \quad (19.7)$$

In this example the open-loop controller  $F$  is constructed with the inverse of  $P$ , but not using the unstable term  $(s-1)$ , and adding additional poles until the relative degree  $D_F$  is zero and therefore the controller is causal. The gain -1 is additionally introduced in order that the steady state gain of the reference transfer function is  $P(s=0) \cdot F(s=0) = 1$ . The two time constants  $T_1$  and  $T_2$  of the controller can be selected arbitrarily with the only restriction that they must be positive,  $T_1 > 0$ ,  $T_2 > 0$ , in order that the controller is stable.

### 19.1.3 Standard controller with one structural degree of freedom

If the plant is unstable and/or the controller shall compensate for unknown disturbances, a feedback controller is needed. The standard controller configuration for this case is shown in Figure 19.3:

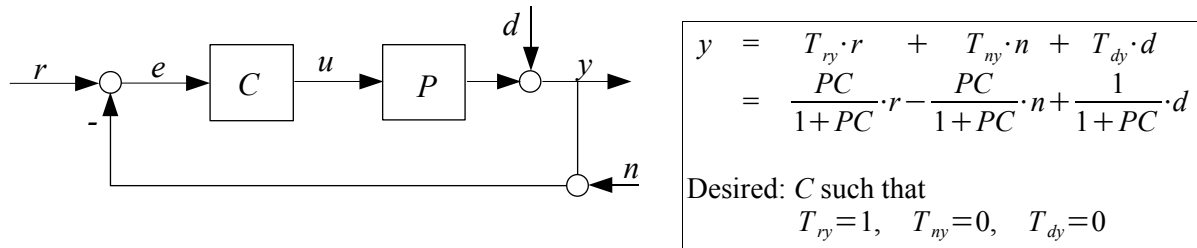


Figure 19.3: Controller  $C$  with one structural degree of freedom for a plant  $P$  ( $P$  might be unstable).

The identifiers in Figure 19.3 have the following meaning:

- $P(s)$  is the plant transfer function,
- $C(s)$  is the feedback controller transfer function,
- $y$  is the output signal of the plant,
- $r$  is the reference signal, i.e., the desired value of  $y$ ,
- $e$  is the control error,
- $d$  is the disturbance, and
- $n$  is the measurement noise.

The goal is to design the feedback controller  $C$  in such a way that the transfer function from the reference  $r$  to  $y$  is one and the transfer functions from the disturbance  $d$  and the measurement noise  $n$  to  $y$  vanish. Obviously, this is not possible. There are in fact quite severe restrictions, because the measurement noise and the

reference signal have the same transfer function and

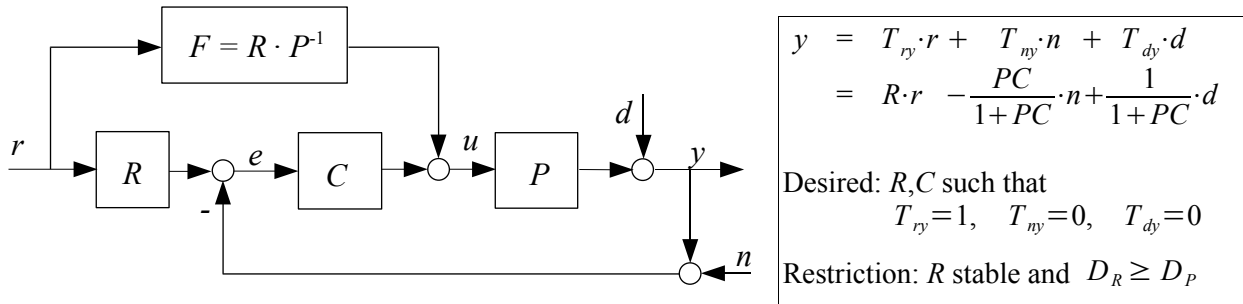
$$T_{ry} + T_{dy} = 1 \quad (19.8)$$

The sum of the two transfer functions is independently of  $C$ . Therefore, when  $C$  is constructed such that the reference transfer function  $T_{ry}$  is optimized then the disturbance transfer function  $T_{dy}$  is fixed and cannot be influenced any more. This strong coupling means that the standard controller configuration of Figure 19.3 has only one structural degree of freedom. The design of the controller with respect to the measurement noise  $n$  is not as critical because  $C$  can be constructed such that  $T_{ry}$  has a low pass behavior with unit gain and then the effect of the high frequency measurement noise is considerably reduced. Still, this leads to structural limitations on the achievable performance for the reference transfer function (independently, what type of controller  $C$  is actually used).

#### 19.1.4 Controller with two structural degrees of freedom for minimum phase plants

We will now considerably enhance the control system by a second structural degree of freedom. In a first step, only minimum phase plants are taken into account to simplify the discussion, see Figure 19.4:

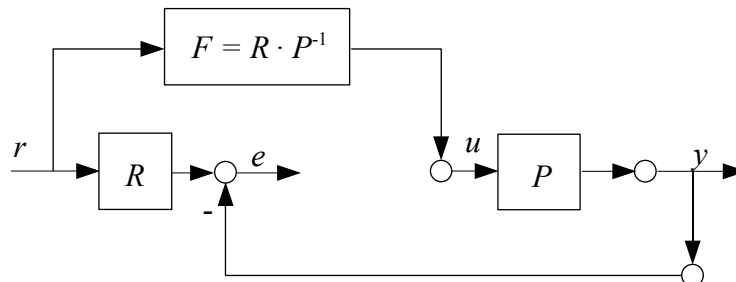
Figure 19.4: Controller  $(F, R, C)$  with two structural degrees of freedom  $(R, C)$  for a minimum phase plant  $P$  ( $P$  might be unstable).



In this case the controller consists of 3 parts: the feedback controller  $C$ , the pre-filter  $R$  and the feedforward controller  $F = RP^{-1}$  which is computed from the pre-filter and the inverse plant model. From the transfer functions in the right part of Figure 19.4, it can be seen that all of them have a different dependency on the controller parts. Especially, the reference transfer function is identical to the pre-filter  $R$  and the measurement noise and disturbance transfer functions depend only on  $C$ , but not on  $R$ . This means that the reference transfer function can be designed completely independent from the measurement noise and disturbance transfer functions. Due to this important property, the controller structure above has two structural degrees of freedom. The 3 controller parts are computed from the design transfer functions  $C$  and  $R$ , and the given plant transfer function  $P$ . The design of  $R$  is simple and straightforward, since  $R$  is the desired reference transfer function. It has only the slight restriction that it must be stable and that the relative degree cannot be smaller as the relative degree of the plant. Details how  $R$  is designed in a suitable way are given below.

It is important to understand the construction above, in order to be able to generalize it to nonlinear plant models in section 19.3: In the next figure, the feedback controller  $C$  and the disturbance and measurement noise signals are removed:

Figure 19.5: Construction of “two degree of freedom” controller for minimum phase plant.



Output  $y$  is computed by the open-loop controller from Figure 19.1. Therefore, restrictions on  $R$  apply, as de-

finied in Figure 19.1. The reference transfer function is independent of the plant:

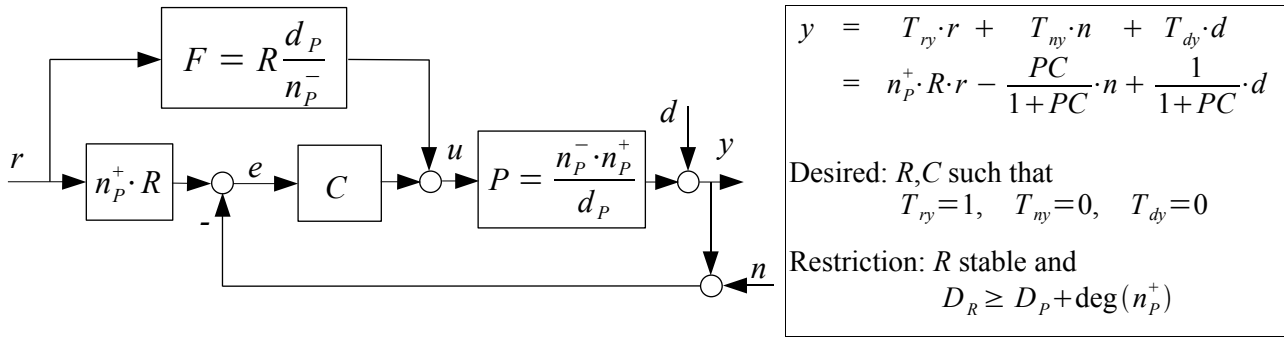
$$\begin{aligned} y &= P \cdot (P^{-1} R) \cdot r = R \cdot r \\ e &= R \cdot r - y = R \cdot r - R \cdot r = 0 \end{aligned} \quad (19.9)$$

A control error  $e$  is constructed by adding the reference transfer function also as pre-filter and subtracting it from the measurement  $y$ . Due to this construction, the control error  $e$  is identical to zero, if disturbance and measurement noise are zero and the plant inverse cancels the plant identically. In this idealized situation, the feedback controller  $C$  is not needed because its input is zero. Therefore,  $C$  is only used to act against disturbances, measurement noise and imprecise plant model, and can be optimized for these tasks, without taking into account the desired reference transfer function.

### 19.1.5 Controller with two structural degrees of freedom for non-minimum phase plants

The above approach can also be generalized to any type of SISO plant as shown in Figure 19.6:

Figure 19.6: Controller  $(F, R, C)$  with two structural degrees of freedom  $(R, C)$  for a plant  $P$  ( $P$  might be unstable and/or non-minimum phase).



With the same argument as before, but by using the more general open-loop controller from Figure 19.2 in the feedforward path and the desired reference transfer function as pre-filter, it can be easily seen that again the control error  $e$  is identical to zero, when disturbance  $d$  and measurement noise  $n$  are zero, and the exact plant model is used in the controller. Therefore, controller  $C$  is again only used to act against disturbances, measurement noise and imprecise plant model, and can be optimized for these tasks, without taking into account the desired reference transfer function.

In (Kreiselmeier 1999) it is shown that every causal controller with inputs  $r, y$  and output  $u$  that stabilizes the plant can be represented by the controller structure of Figure 19.6. Therefore, this is the most general (linear) controller structure for single-input, single-output systems. There are other controller structures with a feedforward path. However, they are not necessarily as general as the controller above. For example, in a first step a feedback controller could be used to stabilize the plant and then an open-loop controller could be designed according to Figure 19.2 for the stabilized plant system to provide a desired reference transfer function. For minimum-phase plants this would result in:

$$F = R \cdot \frac{1+P \cdot C}{P \cdot C}, \quad D_R \geq D_P + D_C \quad (19.10)$$

and therefore the class of realizable reference transfer functions  $T_{ry} = F$  is both smaller and depends on the controller  $C$ , whereas the reference transfer functions  $T_{ry} = R$  in Figure 19.4 do not depend on  $C$ . For example, if  $C$  is strictly proper, i.e.,  $D_C > 0$ , the class of realizable reference transfer functions is obviously smaller as in the controller structure in Figure 19.4, since  $D_R > D_P$ , instead of  $D_R \geq D_P$ .

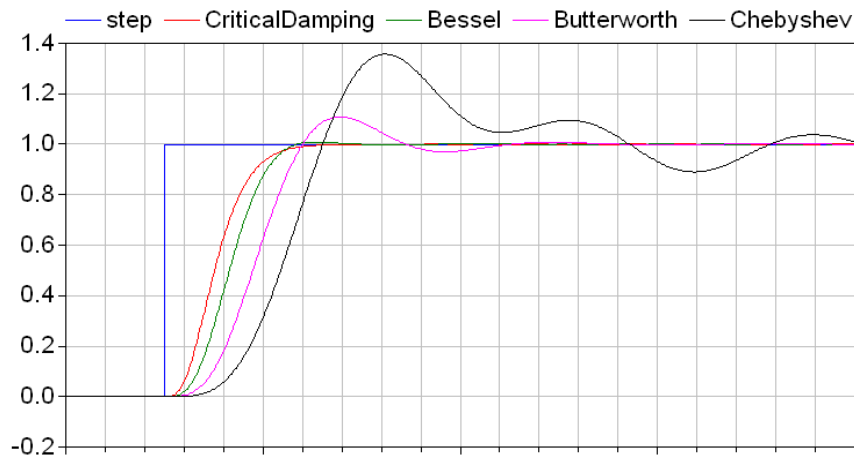
It is unlikely that the controller structure from Figure 19.6 can be generalized to nonlinear plant models, because this would require that a nonlinear model can be split up in a part that has a “stable inverse” and the “remaining part”. This might be possible for special plant models, but not in general. For this reason, the generalizations to nonlinear systems as presented in the following sections are based on Figure 19.4.

### 19.1.6 Design of Prefilter

It remains to be discussed, how the prefilter  $R$  shall be constructed: For *minimum phase* plants,  $R$  is the reference transfer function  $T_{ry}$ . The optimal result would be  $T_{ry}=1$  since then the output signal would follow exactly the reference signal. This is usually not possible, because the relative degree of  $R$  must be larger, or at least as large, as the relative degree of plant  $P$  (which is usually  $> 0$ ). A suitable choice is therefore often a *low pass filter* where the filter *order* is identical to the *relative degree of  $P$*  and the cut-off frequency of the filter is as large as possible. The plant model  $P$  used in the feedforward path is only valid up to a certain frequency. Only up to this frequency, a reasonable cancellation of the plant dynamics can be expected and therefore the cut-off frequency of the filter should be smaller. Practically, the frequency range of the plant model is often known (at least a rough order) and the cut-off frequency of the filter is adjusted around the plant frequency range so that the control behavior is improving.

In Figure 17.3 on page 181 Modelica models for filters are discussed. From the step response of the filters, see Figure 19.7, it can be deduced that only the “CriticalDamping” and the “Bessel” filter are suitable as a reference transfer function, since otherwise the reference signal would have unwanted vibrations.

Figure 19.7: Step responses of normalized low pass filters of order 4.



Since the “Bessel” filter is “faster” as the “CriticalDamping” filter, the first choice for  $R$  is the “Bessel” filter. The design of the reference transfer function is therefore rather simple:

1. Use a normalized, low-pass Bessel filter with unit gain (= `Modelica_LinearSystems.Sampled.Filter` with parameters: `analogFilter = Bessel`, `filterType = LowPass`, `gain = 1`, `normalized = true`).
2. Use the relative degree of the plant model  $P$  as filter order “`order`” (for example, if  $P$  is a PT2 block, use `order = 2`).
3. Adjust the cut-off frequency “`f_cut`” of the filter (manually or by parameter variation/optimization) until the controller performance is satisfactory. Start the tuning with a value of “`f_cut`” which is in the order of the frequency up to which the plant transfer function  $P$  is valid. Use either a more detailed plant model (e.g., a non-linear model) for the tuning, or perform the tuning directly at the real plant.

## 19.2 Inversion of Nonlinear Models

In the previous section 19.1 it was shown how inverse plant models can be utilized for an important controller structure. There are also other useful linear controller structures that are based on inverse models. In general, all controllers that are designed based on *one* linear plant model have the significant drawback that linear plant models nearly never represent the reality in the whole operating region “good enough” and therefore such controllers often operate only reasonable close to the operating point for which they have been designed. In the remaining sections of this chapter we will generalize the approach by using inverse systems of non-linear plant models in order that the controller operates satisfactorily in a much wider region. The described approach is based on a lot of practical experience in designing industrial control systems for aircrafts, robots and vehicles at the DLR Institute of Robotics and Mechatronics since the end of the nineties.

The essential idea is as follows:

- (1) Take any controller structure for linear systems that utilizes a *linear inverse* plant model.
- (2) Replace the linear inverse plant model by a more detailed *nonlinear inverse* plant model.
- (3) Determine the remaining part of the control system by appropriate techniques, e.g., by tuning controller coefficients via parameter optimization.

Several different controller structures according to this technique will be discussed below. A difficult part is issue (2): The nonlinear plant model should be constructed in a convenient way and the inverse model should be directly derived from the plant model. It turns out that Modelica is very well suited for this approach because Modelica is designed to model complex systems, and since Modelica tools, like Dymola, can generate nonlinear inverse models *automatically*:

A continuous Modelica model is primarily mapped to a DAE (= set of Differential Algebraic Equations) of the form:

$$\mathbf{0} = \mathbf{f}(\dot{\mathbf{x}}(t), \mathbf{x}(t), \mathbf{y}(t), \mathbf{u}(t), t), \quad t \in \mathbb{R}, \quad \mathbf{x} \in \mathbb{R}^{n_x}, \quad \mathbf{y} \in \mathbb{R}^{n_y}, \quad \mathbf{u} \in \mathbb{R}^{n_u}, \quad \mathbf{f} \in \mathbb{R}^{n_x + n_y} \quad (19.11)$$

where  $\mathbf{x}(t)$  are variables that appear differentiated in the model,  $\mathbf{y}(t)$  are algebraic variables and  $\mathbf{u}(t)$  are known inputs as function of time  $t$ . It is possible to transform system (19.11) to the following state space form, at least numerically, as sketched in section 12.4:

$$\begin{bmatrix} \dot{\mathbf{x}}_1 \\ \mathbf{x}_2 \\ \mathbf{y} \\ \mathbf{w} \end{bmatrix} = \mathbf{f}_3(\mathbf{x}_1, \mathbf{u}) \quad (19.12)$$

where  $\mathbf{x}_1$  and  $\mathbf{x}_2$  form vector  $\mathbf{x}$  such that the subset vector  $\mathbf{x}_1$  is the *state vector* and contains the independent variables of  $\mathbf{x}$ . The new vector  $\mathbf{w}$  contains higher order derivatives of  $\dot{\mathbf{x}}$  and of  $\mathbf{y}$  that appear when differentiating equations of  $\mathbf{f}(\cdot)$  and that are treated as algebraic variables. For the computation of  $\mathbf{f}_3(\cdot)$ , it might be necessary to solve linear and/or non-linear algebraic systems of equations. The equations to be differentiated can be determined with the algorithm of Pantelides (Pantelides 1988). The selection of the state variables  $\mathbf{x}_1$  can be performed with the “dummy derivative method” of Mattsson and Söderlind (Mattsson and Söderlind, 1991). Both algorithms are, for example, available in the Modelica simulation environment Dymola.

An *inverse* model of the DAE is constructed by *exchanging* the meaning of variables: A subset of the input vector,  $\mathbf{u}_{\text{inv}}$ , with dimension  $n_{\text{inv}}$ , is treated no longer as known but as unknown, and  $n_{\text{inv}}$  previously unknown variables from the vectors  $\mathbf{x}$  and/or  $\mathbf{y}$  are treated as known inputs. The result is still a DAE which can be handled with the same methods as any other DAE. Examples are given in the following sections. This technique of constructing non-linear inverse models has been first applied in (Mugica and Cellier 1994, Otter and Cellier 1996).

For linear systems a complete theory for inverse models can be given, as sketched in section 19.1. For nonlinear systems such a complete theory does not exist and it is very unlikely that it will ever be developed. For example, one critical issue (which seems unsolvable) is the useful definition of “stability” that can be checked by a tool. The current state in the theory of nonlinear control systems is, e.g., summarized in (Slootne and Li 1991, Isidori 1995, Isidori 2006). The practical approach taken in this chapter is *not* well known outside of the Modelica community because the very powerful symbolic algorithms to transform singular DAEs, as presented in section 12.4, seem to be not widely known.

There are several meaningful definitions of “stability” for non-linear systems, e.g., “Ljapunov stability”, “Bounded-Input, Bounded-Output stability”, “Input to State Stability”. Since it is not important for the described approach which definition is used, we will assume that the reader applies the stability definition that is most appropriate for his/her applications. In the following, we will therefore use the term “*stable*” DAE without defining it mathematically. A DAE will be called “*unstable*” if it is not “*stable*”.

In the following, only inverse models will be used in a controller if the DAE of the inverse model has a *unique solution* and if it is *stable*. As discussed in section 19.1, for linear single-input, single output systems the latter requirement means that the plant must have *stable zeros*, that is, it must be a *minimum phase* system. To simplify notation, we will generalize the term “minimum phase system” to non-linear DAEs by the following definition<sup>17</sup>:

<sup>17</sup>In (Isidori 1995) a non-linear *state space* system is defined to be “minimum phase” if the “zero dynamics” is globally



**Definition:**

A DAE is called a *minimum phase* system with respect to a subset of its inputs  $\mathbf{u}_s$  and a subset of its algebraic or differentiated variables  $\{\mathbf{x}_s, \mathbf{y}_s\}$ , with  $\dim(\mathbf{u}_s) = \dim(\{\mathbf{x}_s, \mathbf{y}_s\})$ , if the inverse system from  $\{\mathbf{x}_s, \mathbf{y}_s\}$  to  $\mathbf{u}_s$  has a *unique* solution and is *stable*.

In the following we will assume that models to be inverted are minimum phase systems according to this definition. In section 19.8 it is discussed how to proceed if this requirement does not hold.

Since the transformation from (19.11) to (19.12) might differentiate equations (see section 12.4), the *known inputs* of the *inverse* model may be differentiated too, i.e., the derivatives of these inputs must exist and must be provided analytically up to a certain order. These derivatives can be provided if, e.g., the inputs are available as analytic functions that can be differentiated sufficiently often, or by a desired reference model that in combination with the inverse DAE results in a DAE that does not require derivatives of inputs. Often, the reference model is selected as a filter such that a combination of the filter states yields the needed derivatives. For linear systems, this approach is well known, see section 19.1. Take for example the following linear system with one zero and two poles:

$$y = \frac{s+1}{(s-2)(s+3)} \cdot u \quad (19.13)$$

The inverse model together with a reference model might be constructed as

$$u = \frac{(s-2)(s+3)}{(s+1)} \cdot \frac{1}{(T \cdot s+1)} \cdot y \quad (19.14)$$

In order that the model is *causal* (i.e., can be implemented as an algorithm), additional poles have to be added until the degree of the denominator is larger or, at least, as large as the degree of the numerator. For this reason, a filter  $1/(T \cdot s+1)$  has been connected in series, making the combined transfer function causal.

Another possibility is not to control  $y$ , but one of its derivatives instead:

$$\dot{y} = s \cdot y = \frac{s(s+1)}{(s-2)(s+3)} \cdot u \quad (19.15)$$

The transfer function has now a relative degree of zero and may be inverted:

$$u = \frac{(s-2)(s+3)}{s(s+1)} \dot{y} \quad (19.16)$$

Connecting this controller with the plant in series, results in integrator behavior. A simple feedback loop may be added to place the integrator pole at a desired location.

With a Modelica simulation environment, such as Dymola, the practical derivation of inverse models is straightforward, even for complex systems:

1. Define the plant as Modelica model and include input and output signals of the plant over which the inversion shall take place.
2. If necessary, provide a reference model or input filter of appropriate relative degree. The relative degree may be known from physical knowledge of the plant dynamics, or can be automatically derived by Dymola, as described below.
3. Connect the “u1” inputs of a “Modelica.Blocks.Math.InverseBlockConstraints” block to the plant outputs, the “u2” inputs of this block to the outputs of the reference model, and the input of this model to an input signal connector (Modelica.Blocks.Interfaces.RealInput) that defines the desired plant outputs.<sup>18</sup>

---

asymptotically stable. For DAE systems, there seems to be no publication with a corresponding definition. The definition introduced here for DAEs is simple and reduces to the standard definition for linear systems if the DAE is linear.

<sup>18</sup>In the Modelica Standard Library 2.2.2 and previous versions, inversion is defined with block “Modelica.Blocks.Math.TwoInputs”. This block is not a “balanced model” and was therefore replaced in Modelica Standard Library 3.0 by “Modelica.Blocks.Math.InverseBlockConstraints”.

A typical example is shown in Figure 19.8.

Figure 19.8: Definition of inverse model in Modelica (left part: the 4 basic components; right part: the 4 basic components connected together).



On the left part of the figure, the four basic components are shown

- (1) “Modelica.Blocks.Interfaces.RealInput” is the input “u” to the inverse model.
- (2) The input signal is filtered with a corresponding filter block (e.g., “Modelica.Blocks.Continuous.CriticalDamping” or “Modelica.LinearSystems.ZerosAndPoles.filter”).
- (3) The output of the filter should be connected to the output of the plant. This is not directly possible, because signal connectors can only be connected according to block diagram semantics and in block diagrams it is not allowed to connect two output signals with each other. For this reason block “Modelica.Blocks.Math.InverseBlockConstraints” is present. It has two input connectors  $u_1, u_2$  and two output connectors  $y_1, y_2$  and is described by the trivial equations “ $u_1 = u_2$ ”, “ $y_1 = y_2$ ”. When connecting a block to the “inner” connectors  $u_2, y_2$ , the input/output signals are exchanged.
- (4) The plant model to be inverted.

On the right part of Figure 19.8, it is shown how the components are connected together. Note, that the plant model is horizontally “flipped” and connected to the “inner” input/output connectors of the “InverseBlockConstraints” block. If the filter order is too low, the DAE is not causal and Dymola prints an error message of the following form, deducing this information from the result of the Pantelides algorithm:

Error: The model requires derivatives of some inputs as listed below:

Order of input derivative

4	$u_1$
2	$u_2$
3	$u_3$

Error: Failed to reduce the DAE index

In the second column the Modelica names of the input signals are listed that need to be differentiated according to the differentiation order of the first column. The numbers in the first column are therefore the minimum order of the corresponding filters. For example, a low pass filter with a minimum order of 4 has to be used to filter the input signal  $u_1$  in order that no derivatives of  $u_1$  are needed for the inverse model. Note, in linear and nonlinear controller theory the “Order of input derivative” numbers shown in Dymola’s error message are the negative “*relative degrees*” of the respective inputs. For example,  $u_1$  has a relative degree of -4.

If the inversion is to be based on a time derivative of the measured output, a sufficient number of integrators might be added, instead of increasing the filter order.

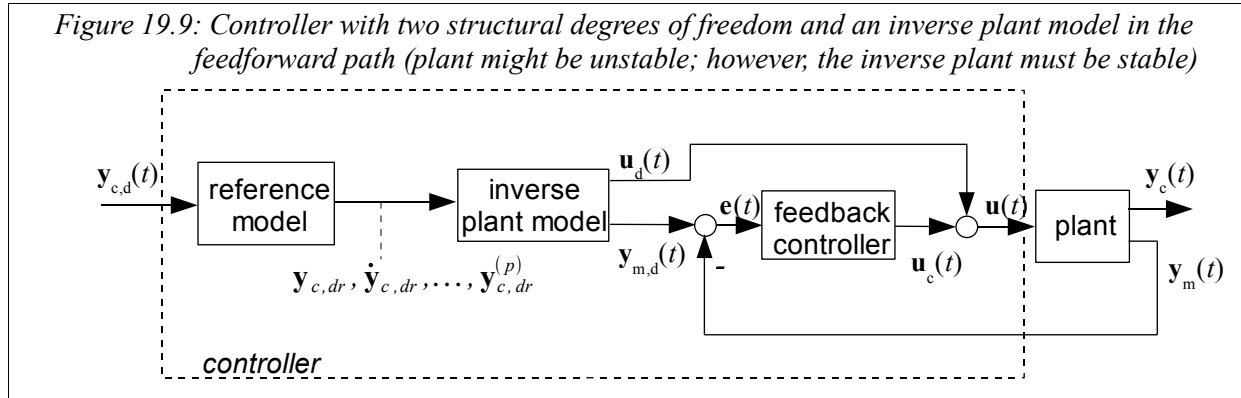
There is always a filter order and/or a number of integrators for which the system will translate. The higher the filter order, the more problems will occur when applying it in a real control system. In such cases, one might remove dynamic elements from the plant and try it again. One might even use a *stationary plant* model for the inversion.

In the following sections, different controller structures will be discussed in more detail that follow the general approach outlined above.

### 19.3 Inverse Model in Feedforward Path of Controller

The controller with two structural degrees of freedom from Figure 19.4 on page 205 is generalized according to the approach sketched in the previous section. This structure has been applied in (Thümmel et. al. 2001) to the control of robots and has been successfully validated with hardware experiments. In flight control, the “model following approach”, see for example (Adams and Banda 1993), is a special case of this structure whereby the reference model is known as the “command block” providing state references for the inverse

model as well as the feedback controller.



In Figure 19.9, the multi-input/multi-output plant has inputs  $\mathbf{u}$ , measured signals  $\mathbf{y}_m$  and controlled outputs  $\mathbf{y}_c$ . In many cases  $\mathbf{y}_c \in \mathbf{y}_m$ , i.e., the controller outputs are a subset of the measured signals. However, there are also practically important cases where the two signal sets are different. For example, a standard electrical drive system has the speed of the motor inertia as measurement signal  $\mathbf{y}_m$  and the speed of the load inertia as the signal  $\mathbf{y}_c$  to be controlled.

For this controller structure, the number of inputs must be identical to the number of controlled variables:  $\dim(\mathbf{y}_c) = \dim(\mathbf{u})$  and the plant must be minimum phase (but might be unstable). In this case the inverse plant model with (known) inputs  $\mathbf{y}_c$  and unknown outputs  $\mathbf{u}$  is used in the feedforward path of the controller to compute the desired actuator inputs  $\mathbf{u}_d$  to the plant.

A “reference model” defines the desired dynamic behavior of the closed loop system. It is often most convenient to use a filter, since the filter is parameterized by just the cut-off frequency, once the filter order and the filter type is fixed, and because a filter provides the “optimal” reference model with transfer function “1” below the cut-off frequency (for details see page 207). There are also other useful choices of the reference model.

The outputs  $\mathbf{y}_{c,dr}$  of the reference model are the inputs to the inverse plant model. By solving a DAE system or the symbolically transformed system, the inverse plant model computes the desired measurement signals  $\mathbf{y}_{m,d}$  and the desired plant inputs  $\mathbf{u}_d$ . A feedback controller is used to stabilize the overall system and to improve robustness. This might be a simple PID like controller.

It can be shown that the feedback controller has no effect, as long as the plant and the inverse plant models are identical, the plant and the inverse plant models are stable and both start at the same initial conditions. In this case the “reference model” (i.e., in many cases the filter) determines the input/output behavior, i.e., it is the transfer function of the closed loop system. If these assumptions are not fulfilled, a control error occurs and the controller has to stabilize the system and cope with the imprecise inverse plant model and its initial conditions. This structure has several advantages:

- The two controller parts (inverse plant model with reference model and feedback controller) can be designed independently from each other.
- The controller structure can be applied to unstable plants provided the inverse model is stable, see section 19.8.
- Since the inverse plant model is in the feedforward path, the calculation of  $\mathbf{u}_d$  and of  $\mathbf{y}_{m,d}$  might be performed offline if possible, so that hard real-time requirements for the solution of the inverse plant model need not to be present.

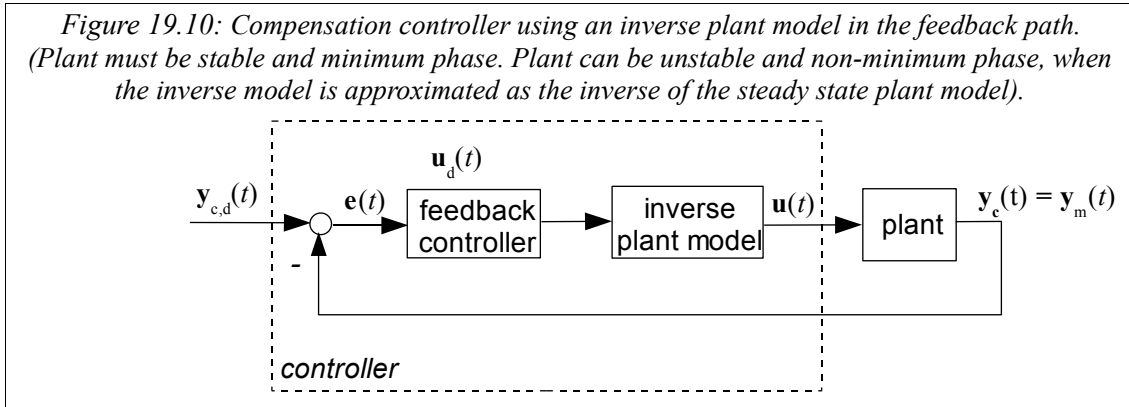
The disadvantage of this structure is that for some applications the feedback controller may still have to be scheduled as a function of the operating conditions.

Inverse model-based feedforward control will be demonstrated on the basis of an example in section 19.7.

## 19.4 Inverse Model in Feedback Path of Controller

The disadvantage of the inverse feedforward controller can be avoided by moving the inverse model into the feedback path. This is shown in Figure 19.10. The feedback controller now only “sees” the combined inverse and plant model. The structure is a generalization of the linear compensation controller described in

(Föllinger 1994, page 266).



For linear plant models the “feedback controller”, see Figure 19.10, must have a relative degree that is equal or larger than the relative degree of the plant, in order that the system is proper. For single-input/single-output systems, a useful “feedback controller” is

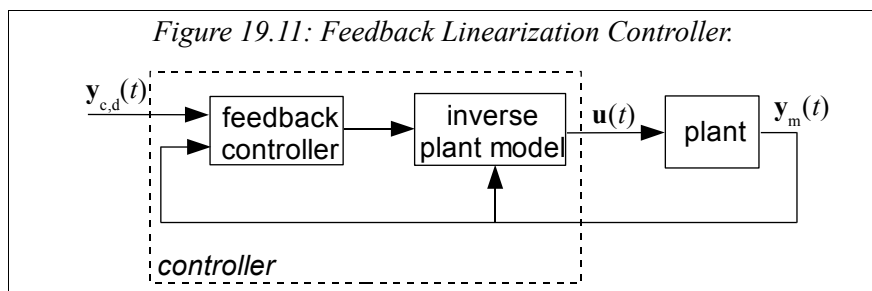
$$u_c = \frac{1}{r(s)-1} \cdot e \quad (19.17)$$

where  $1/r(s)$  is the desired reference transfer function from  $y_{c,d}$  to  $y_c$ . Under the assumption that the desired and the actual plant behavior is identical, the inverse and the actual plant model “cancel” each other and the transfer function from  $y_{c,d}$  to  $y_c$  is identical to  $1/r(s)$ , i.e.,  $r(s)$  of the feedback controller defines the “desired” closed loop behavior. Note that it is assumed that  $y_c$  is measurable (in this case,  $y_c = y_m$ ). Alternatively, the procedure as described above may be applied: integrators are added to the inverse model input before designing the feedback controller.

This structure has the disadvantage that it can be applied to stable plants only. Also the inverse plant model needs to be stable. For a linear plant model this is obvious, since otherwise an unstable pole/zero cancellation occurs, resulting in an internally unstable system. For multi-input/multi-output systems it is nearly always possible (also for *unstable* plants) to construct the *inverse of a stationary* desired plant model. Once the control error  $e$  has reached a stationary value, the inverse plant model leads to a decoupled control loop. In other words, the different outputs might be controlled independently from each other by simple PID-like single-input/single-output controllers and the stationary inverse plant model is used to decouple the control loops from each other.

## 19.5 Inverse Model in Feedback Linearization Controller

A complete theory to use nonlinear plant models as the controller kernel is “feedback linearization” (in aerospace applications also known as “Nonlinear Dynamic Inversion”, NDI), see for example (Isidori 1995) and (Enns et. al. 1994). The basic structure is given in Figure 19.11. The principal difference compared with the feedforward (Figure 19.9) and the compensation controller (Figure 19.10) is that the states in the inverse model are obtained from the actual plant, via *measurement* and *estimation*. Contrary to the compensation controller, the methodology can also be applied to *unstable* plants.



When deriving feedback linearizing control laws manually, the outputs to be controlled are differentiated un-

til an analytical relation with a control input is found. For this process, usually “Lie” algebra is used. The number of required differentiations is the so-called relative degree of the specific output.

If the system model is available in Modelica, the derivation of the control laws can be automated using a similar procedure as described above. However, instead of a filter of appropriate relative degree, a set of integrators is added (see above):

$$y_i = \frac{1}{s^{p_i}} v_i \quad (19.18)$$

where  $v_i$  is the  $i^{\text{th}}$  new model input, corresponding with the  $i^{\text{th}}$  output (with relative degree  $p_i$ ). The desired dynamic behavior of the closed-loop system is then imposed by application of an additional feedback law, like for example:

$$v_i = k_{0,i}(y_{md,i} - y_{m,i}) - k_{1,i}(\dot{y}_{m,i}) - \dots - k_{(p_i-1),i}(y_{m,i}^{(p_i-1)}) \quad (19.19)$$

Note that this feedback law requires availability of the  $(p_i-1)^{\text{th}}$  derivative of the controlled output. This derivative may be obtained from measurements or, less favorably, from the computed value in the inverse model. In aerospace applications first or no time derivatives are usually required, since relative degrees of controlled variables tend to be low (1 or 2). One reason for this is that control laws are designed in the form of multiple cascaded loops. In case the inverted model exactly represents the true system, the closed loop system becomes:

$$y_{m,i} = \frac{k_{0,i}}{s^p + k_{(p-1),i}s^{p-1} + \dots + k_{1,i}s + k_{0,i}} y_{md,i} \quad (19.20)$$

Note that the coefficients may be selected to match the reference model in Figure 19.9.

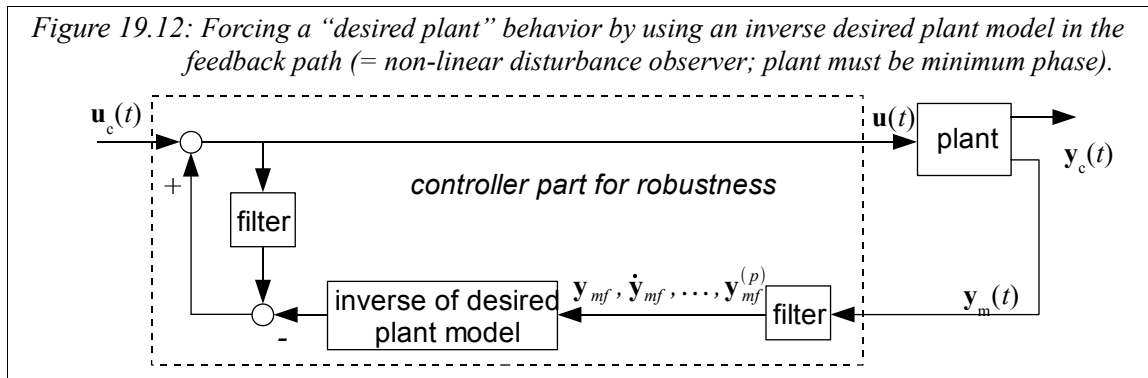
In case time derivatives of the desired output  $y_{c,d,i}$  are available, the relative degree (i.e., the phase lag of the response) of this linear closed loop system may be reduced, provided that these are not too fast as to require too large control inputs.

An important disadvantage of feedback linearization is that the state vector of the plant must be fully available from measurement and/or estimation.

Automatic generation of feedback linearization control laws in Modelica will be illustrated in section 19.7. This procedure has been applied for an automatic landing system, and manual control laws for a fighter aircraft. The software code for the automatic landing system was automatically generated with Dymola and successfully flight tested on a small passenger jet (Bauschat et. al. 2001).

## 19.6 Inverse Model in Robust Controller (Disturbance Observer)

All previous controller structures require that the plant model used as inverse system in the controller matches the real plant “sufficiently” accurate. The controller structure in Figure 19.12 uses an inverse model to achieve a more robust design. It was developed for linear systems with the goal to enhance robustness against disturbances and model errors. This structure is called “*disturbance observer*” in the literature although the name is misleading, because it is actually an additional component for a controller. It can be designed independently from the main control loop.



In Figure 19.12 the generalization for nonlinear systems is shown: One important part is an inverse model of a desired plant behavior in the feedback path. Additionally, the same filter is present at two places. The standard disturbance observer uses a linear model for the inverse plant model. A nonlinear desired plant model provides more freedom, since it might be impossible that a physical system can be forced to have the same linear behavior in its whole operating range. Note, there is the requirement that the number of measurement signals is identical to the number of plant inputs:  $\dim(\mathbf{y}_m) = \dim(\mathbf{u})$ .

For a single-input/single-output system where all parts are linear, the transfer function from  $u_c$  to  $y_m$  is given by:

$$y_m = \frac{1}{\frac{1-F(s)}{P(s)} + \frac{F(s)}{P_{des}(s)}} \cdot u_c \quad (19.21)$$

where  $F(s)$  is the filter,  $P(s)$  is the plant and  $P_{des}(s)$  is the desired plant transfer function in the feedback loop. For low frequencies,  $F(s) \approx 1$  and therefore  $y_m \approx P_{des}(s) \cdot u$ . For high frequencies,  $F(s) \approx 0$  and then  $y_m \approx P(s) \cdot u_c$ . The effect of the disturbance observer is therefore, that it enforces the desired plant behavior for low frequencies. In other words, if there are modeling errors or disturbances then the disturbance observer enforces a desired plant behavior below the cut-off frequency of the filter, i.e., the controller designed for the desired plant will usually work considerably better.

The disturbance observer is usually combined with other controller structures. For example, with the structure from section 19.3:

- An inverse plant model from  $\mathbf{y}_c$  to  $\mathbf{u}$  in the feedforward path is used for command following and for providing the desired measurements  $\mathbf{y}_{m,d}$ .
- An inverse plant model from  $\mathbf{y}_m$  to  $\mathbf{u}$  in the feedback path is used to make the closed loop system robust against model errors and disturbances.
- The feedback controller in the feedback loop is used to stabilize the system.

## 19.7 Example Application

In this section the *feedforward* and *feedback linearization* controller structures as discussed in the previous sections will be illustrated by means of a simple example. The plant description is from (Föllinger 1998, page 279):

A substance A is flowing continuously into a mixing reactor. Due to a catalyst, the substance reacts and splits into several base substances that are continuously removed. The reaction generates energy and therefore the reactor is cooled with a cooling medium. The cooling temperature  $T_c(t)$  in [K] is the primary actuation signal. Substance A is described by its concentration  $c(t)$  in [mol/l] and its temperature  $T(t)$  in [K] according to the following DAE:

$$\begin{aligned} \gamma &= c \cdot k_0 \cdot e^{-\epsilon/T} \\ \dot{c} &= -a_{11} \cdot c - a_{12} \cdot \gamma + a_{13} \\ \dot{T} &= -a_{21} \cdot T + a_{22} \cdot \gamma + a_{23} + b \cdot T_c \end{aligned} \quad (19.22)$$

with

$$\begin{aligned} k_0 &= 1.24 \cdot 10^{14}, & a_{11} &= 0.00446, & a_{21} &= 0.0303, \\ \epsilon &= 10578, & a_{12} &= 0.0141, & a_{22} &= 2.41, \\ b &= 0.0258, & a_{13} &= 0.00378, & a_{23} &= 1.37 \end{aligned} \quad (19.23)$$

For the given input  $T_c(t)$  these are 1 algebraic equation for the reaction speed  $\gamma(t)$  and two differential equations for  $c(t)$  and  $T(t)$ . The concentration  $c(t)$  is the signal to be primarily controlled ( $= \mathbf{y}_c$ ) and the temperature  $T(t)$  is the signal that is measured ( $= \mathbf{y}_m$ ).

### 19.7.1 Inverse Model in Feedforward Path

The inverse plant model is constructed from (19.12) by assuming that the variable to be controlled, i.e., the concentration  $c(t)$ , is a known time function and that the previously known input  $T_c(t)$  shall be computed

from the inverse model. By inspection or by using the Pantelides algorithm, see section 12.4, it turns out that the first two equations of (19.12) have to be differentiated:

$$\begin{aligned}\dot{y} &= \left( \dot{c} + \frac{\varepsilon \cdot c}{T^2} \dot{T} \right) \cdot k_0 \cdot e^{-\varepsilon/T} \\ \ddot{c} &= -a_{11} \cdot \dot{c} - a_{12} \cdot \dot{y}\end{aligned}\quad (19.24)$$

(19.22) and (19.24) are the inverse model of (19.22). A filter with an  $n$ th order pole on the negative real axis is used as “reference model”. Since the second derivative of the input appears ( $= \ddot{c}$ ), at least a filter of order 2 is needed, such as<sup>19</sup>:

$$c = \frac{1}{(s/\omega + 1)^2} \cdot c_{des} \quad (19.25)$$

with  $c_{des}$  the desired concentration,  $\omega = 2\pi f$  and  $f$  the cut-off frequency of the filter. A state space description of the filter is given by:

$$\begin{aligned}\dot{x} &= (c_{des} - x) \cdot \omega \\ \dot{c} &= (x - c) \cdot \omega\end{aligned}\quad (19.26)$$

The needed second derivative of  $c$  is obtained by differentiating the second equation of (19.26):

$$\ddot{c} = (\dot{x} - \dot{c}) \cdot \omega \quad (19.27)$$

Equations (19.22), (19.24), (19.26), (19.27), are the DAE of the inverse model of (19.22) with a prefilter of order 2, i.e., these are the connected blocks labeled as “inverse plant model” and as “reference model” in Figure 19.9. It turns out that this DAE has two states. One possibility is to use the filter states  $\{x, c\}$  as state vector  $\mathbf{x}_1$  of the overall system. Here, the original plant states  $\{c, T\}$  are used as state vector  $\mathbf{x}_1$ . Transforming the equations to state space form (19.12) results in the following sequence of assignment statements to compute the derivative  $\{\dot{c}, \dot{T}\}$  of the state vector and of the output  $T_c$  as function of  $\{c, T\}$ :

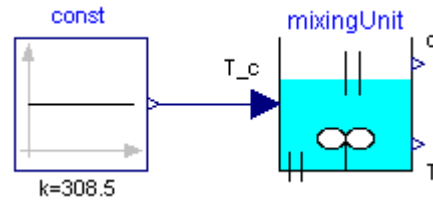
$$\begin{aligned}y &:= c \cdot k_0 \cdot e^{-\varepsilon/T} \\ \dot{c} &:= -a_{11} \cdot c - a_{12} \cdot y + a_{13} \\ x &:= c + \dot{c} / \omega \\ x' &:= (c_{des} - x) \cdot \omega \\ c'' &:= (x' - \dot{c}) \cdot \omega \\ y &:= (c'' + a_{11} \cdot \dot{c}) / a_{12} \\ \dot{T} &:= \frac{T^2}{\varepsilon \cdot c} \cdot \left( \frac{y'}{k_0 \cdot e^{-\varepsilon/T}} - \dot{c} \right) \\ T_c &:= (\dot{T} + a_{21} \cdot T - a_{22} \cdot y - a_{23}) / b\end{aligned}\quad (19.28)$$

For notational clarity, the time derivatives of variables that are treated as purely algebraic variables ( $=$  “dummy derivative method”) are denoted with an apostrophe, such as  $x'$ . Equations (19.28) are a set of differential equations in state space form: Given the desired concentrations  $c_{des}$ , it is possible by numerical integration to compute the desired cooling temperature  $T_c$  ( $= \mathbf{u}_d$  in Figure 19.9) and the desired substance temperature  $T$  ( $= \mathbf{y}_{m,d}$  in Figure 19.9). The latter is compared with the measured substance temperature forming the control error  $\mathbf{e}$  as input to the feedback controller.

Even for this rather simple system, the derivation of the nonlinear feedforward controller is not so easy. Such a manual derivation becomes impractical if the plant model consists of hundreds or of thousands of equations as it is usual in complex Modelica models. It is now demonstrated how to derive this nonlinear feedforward controller in an automatic way with Modelica and Dymola:

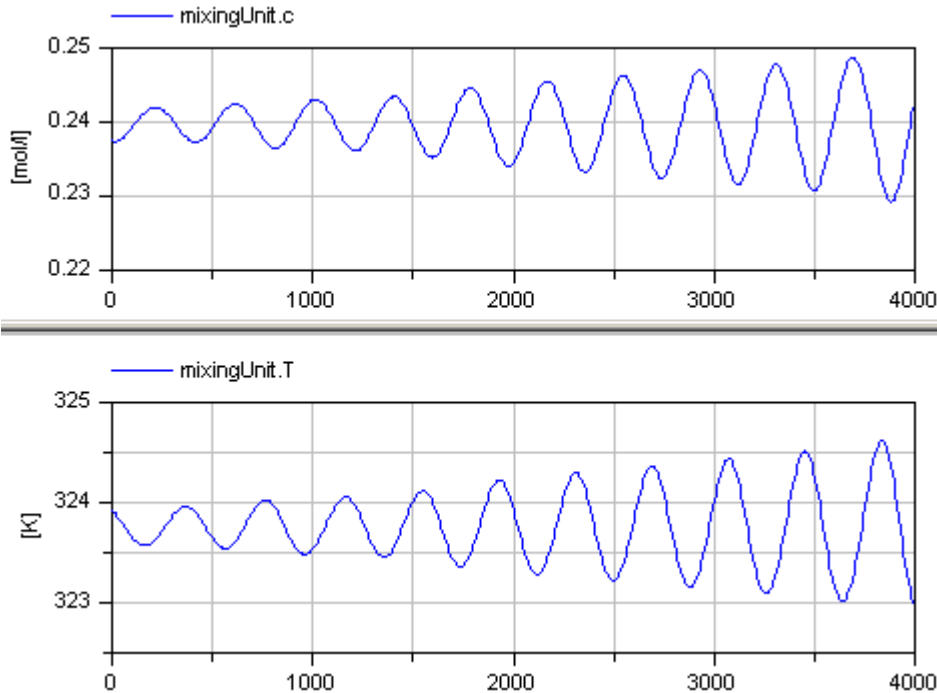
<sup>19</sup>This is a CriticalDamping filter without normalization (for simplicity).

Figure 19.13: Modelica model of mixing unit with constant cooling temperature  $T_c$ .



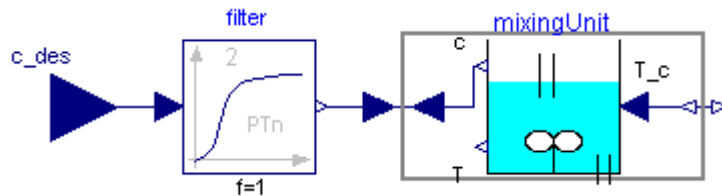
In Figure 19.13 a Modelica model of the mixing unit is shown. The constant input is the cooling temperature; the outputs are the concentration  $c$  and the temperature  $T$  of the substance. This model contains just the plant equations (19.22). Simulation results of this model are shown in Figure 19.14.

Figure 19.14: Simulation results of mixing unit for  $c(t_0) = 0.237$  mol/l,  $T(t_0) = 323.9$  K,  $T_c(t) = 308.5$  K.



As can be seen, the system is unstable at this operating point. In Figure 19.15 the inverse model of the mixing unit is constructed by connecting the input “ $c\_des$ ” via a filter to the “ $c$ ” output of the mixing unit, i.e., the concentration  $c$  is treated as known input signal.

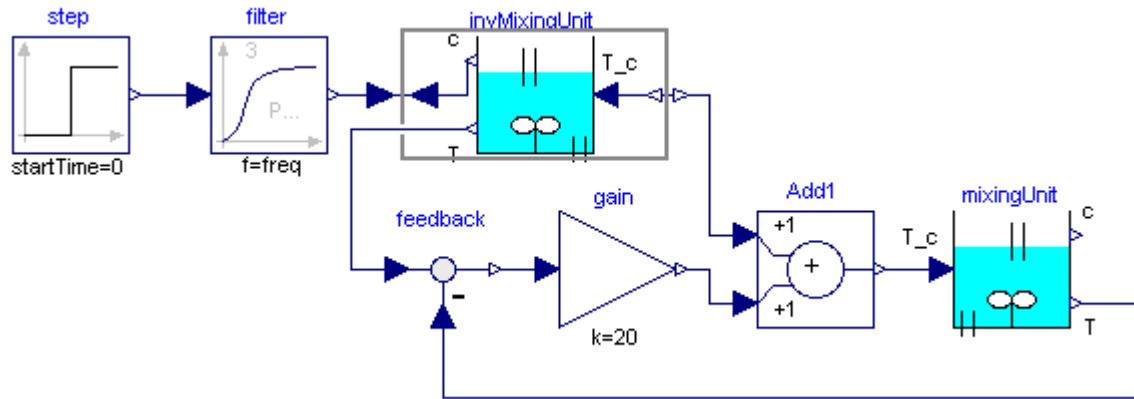
Figure 19.15: Inverse model of mixing unit with pre-filter.



When this system is translated without the filter, Dymola reports that the second derivative of  $c\_des$  is needed. In a second step, the filter is included with `order = 2` and Dymola translates without an error. Afterwards, the inverse model is connected with the plant model according to Figure 19.8. The result is shown in Figure 19.16. In order to not have a jump in the cooling temperature, a filter order of 3 instead of 2 is actually used. The cut-off frequency of the filter is set to 1/300 Hz. It turns out that a simple P controller is sufficient to stabilize the system. A controller gain of 20 is selected.

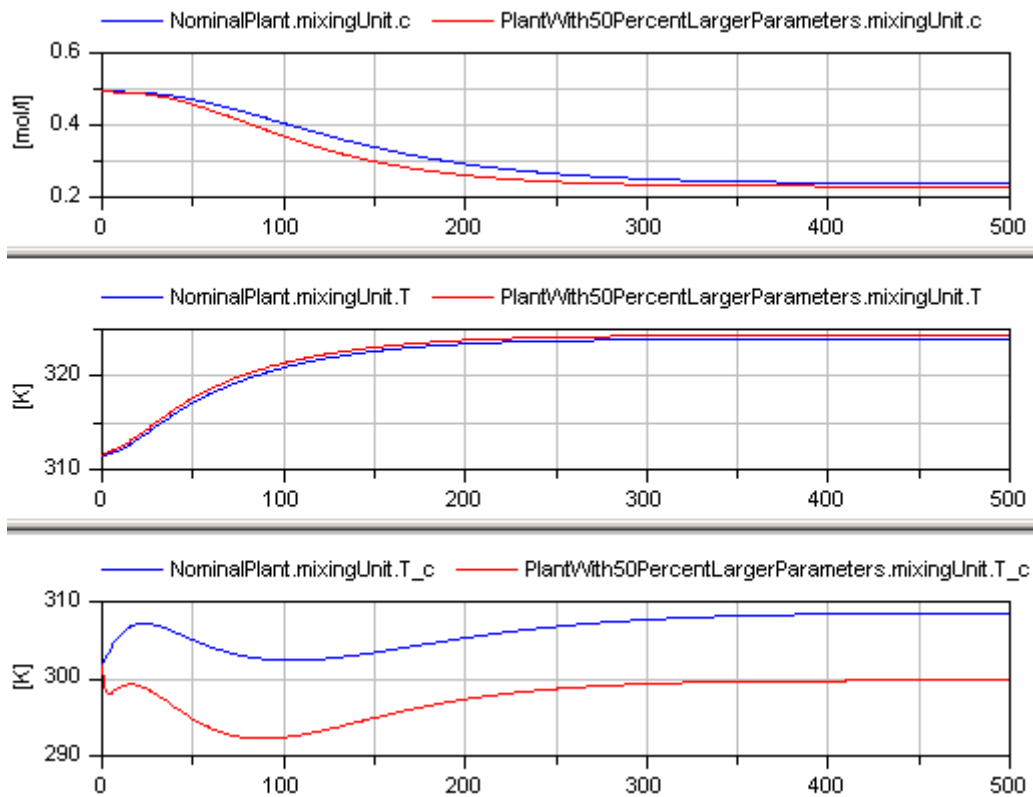


Figure 19.16: Control system with nonlinear feedforward controller for mixing unit.



Simulation results are shown in Figure 19.17 for a jump of  $c_{\text{des}} = 0.492$  to  $0.237$ . The straight lines correspond to the nominal case, where the plant and the inverse plant model have the same parameters. The result is a good control behavior. The dashed lines correspond to the case where the parameters of the plant are 50 % higher as the parameters of the inverse plant model to check the robustness of the design (only parameter  $\varepsilon$  was not changed because the result is very sensitive to it). As can be seen, the result is still satisfactorily. For an actual design, it is useful to perform Monte Carlo simulations by varying all model parameters and initial conditions of the plant systematically in order to determine the robustness of the control system.

Figure 19.17: Simulation results of mixing unit of Figure 19.16.  
 Blue lines: same model parameters for plant and inverse plant model.  
 Red lines: model parameters of plant are enlarged by 50 %.

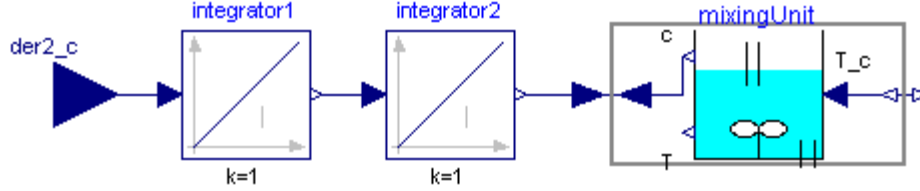


### 19.7.2 Feedback linearization

The compensation control scheme and the feedback linearization cannot be applied directly to the example plant, since the concentration  $c$  is not measurable. One possibility is to use model knowledge in combination with estimation (e.g., a Kalman filter), but this is beyond the scope of this example. For this reason it is as-

sumed that the concentration is measurable. In Modelica, design of feedback linearization and the compensation controllers start in the same way. For these controller types two integrators are added, instead of an input filter, see Figure 19.18.

Figure 19.18: Inverse model of mixing unit for feedback linearization (compare with Figure 19.15).



The input “der2\_c” of this inverse model is the second derivative of “c\_des” ( $= \ddot{c}_{des}$ ). In the case of the compensation controller, the inversion work is done. For feedback linearization, the states in the inverse plant model must be replaced with measured ones. This can be performed by setting the flag

Advanced.TurnStatesIntoInputs = **true**

in Dymola before translation to transform all states into inputs in the generated code. This code can be incorporated with the “Simulink export” feature of Dymola in another environment, such as Simulink from Mathworks. Currently, it is not possible to import this transformed system in to Modelica again. Dynasim plans to support this in the future. For the example, the differentiated equations are added manually to the model, and  $\ddot{c}_{des}$  and the plant states  $\{c, T\}$  are selected as input variables. This has been performed by

- (1) translating the model of Figure 19.18 in Dymola with option “Simulation / Setup / Translation / Generate listing of translated Modelica code in dsmodel.mof”,
- (2) copying the equations of file “dsmodel.mof” in to a new “InverseMixingUnit” model, and
- (3) changing all “der(xx)” to new variables “der\_xx”.

This derivation is, of course, no practical for complex models. The design is finished by adding the feedback controller, e.g., (19.19) in case of feedback linearization. For the model at hand, (19.19) is defined as:

$$c'' = k_1(c_{des} - c) - k_2\dot{c} \quad (19.29)$$

whereby  $c$  is available as measurement signal,  $\dot{c}$  is computed with the inverse model and  $c''$  is the input “der2\_c” to the inverse model of Figure 19.18. By choosing

$$k_1 = (2\pi f)^2, \quad k_2 = 2(2\pi f) \quad (19.30)$$

with  $f = 1/300$  Hz, the same closed loop dynamics is obtained as with the input filter of second order with the feedforward controller. Starting from the ideal response

$$\frac{1}{r(s)} = \frac{k_1}{s^2 + k_2s + k_1} \quad (19.31)$$

the feedback controller may also be shaped as:

$$T_c = \frac{1}{r(s)-1}s^2 = \frac{k_1s^2}{s^2 + k_2s} \quad (19.32)$$

whereby in the numerator  $s^2$  has been added, since the input of the inverse model is effectively the second time derivative of  $c_{des}$ . The over-all closed-loop system is depicted in Figure 19.19. Component “inverseMixingUnit” is the manually derived inverse model with 2 integrators according to Figure 19.18.

Figure 19.19: Closed loop system of mixing reactor and feedback linearization controller.

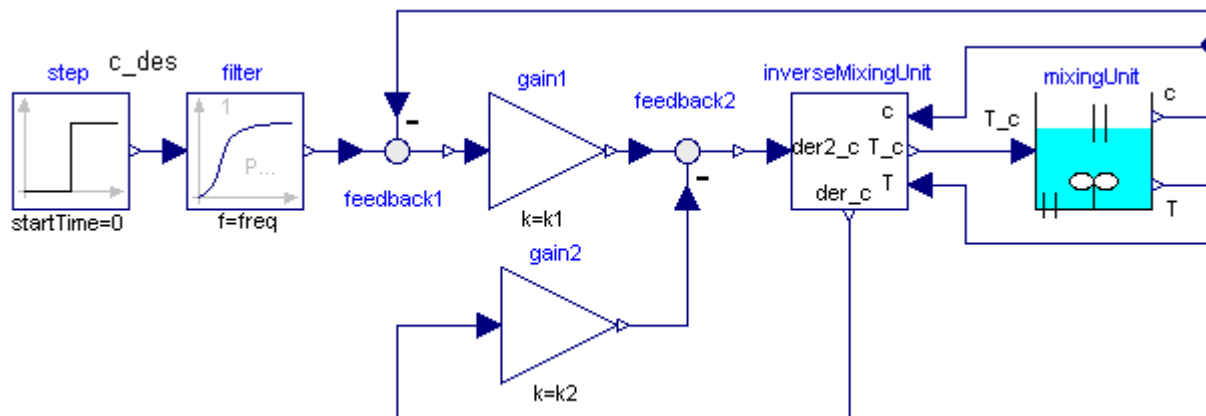
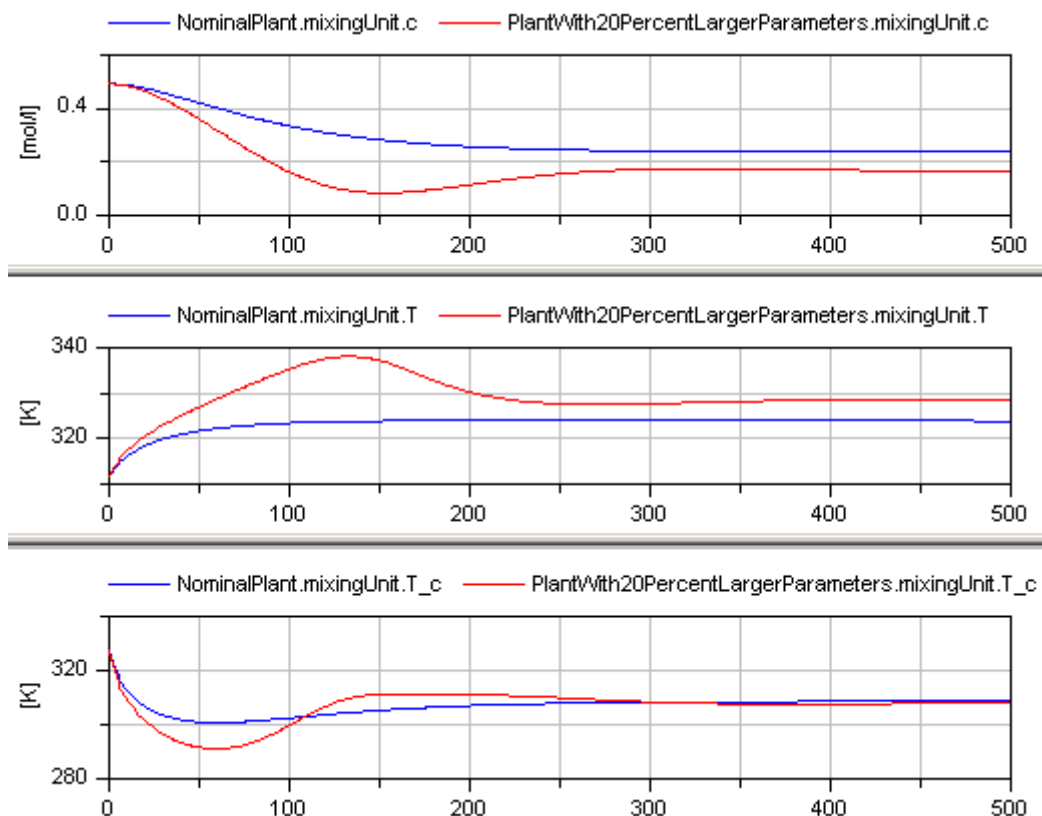


Figure 19.20 shows the response of the closed loop system to the same command as in Figure 19.17. The command input has been smoothed with a first order filter. When the parameters of the “mixingUnit” are changed by 50 % as in Figure 19.17, the closed loop system becomes unstable. Therefore, in Figure 19.20, only parameter variations of 20 % are shown.

Figure 19.20: Simulation results of mixing unit of Figure 19.19.

Blue lines: same model parameters for plant and inverse plant model.

Red lines: model parameters of plant are enlarged by 20 %.



## 19.8 Difficulties with Inverse Models

When constructing inverse models for industrial systems, it is often the case that the generated inverse models do not work as expected. In this section, the major reasons are discussed and it is explained how to circumvent such problems.

### 19.8.1 Unstable inverse models

Usually, it is required that the inverse model is a stable system. For example, in the structure of Figure 19.9 the inverse model is in the feedforward path and if it would not be stable, the overall system would be unstable as well. For linear single-input/single-output systems this situation is well known and can be easily analyzed. For example, take the following linear plant model:

$$y = \frac{s-1}{(s-2) \cdot (s+3)} \cdot u \quad (19.33)$$

The inverse model together with a reference model is

$$u = \frac{(s-2) \cdot (s+3)}{(s-1) \cdot (Ts+1)} \cdot y \quad (19.34)$$

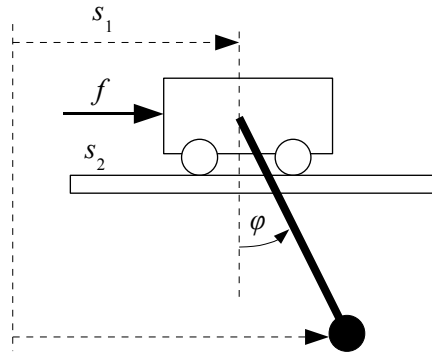
As can be seen, the inverse model is unstable, because the plant has an unstable zero. In other words, for linear systems the plant must be a minimum phase system in order that the inverse model is stable. For a general DAE no stability proof exists. Therefore many simulations have to be performed with the inverse DAE to check whether it is stable in the desired operation region. For certain classes of DAEs, it might be possible to proof that the inverse model is stable. An alternative is to linearize the plant model around several stationary operating points and check whether the transmission zeros are stable. Of course, none of these checks can guarantee that the inverse DAE is stable for simulations or stationary points that have not been analyzed.

If the inverse plant is unstable, only approximate inverse plant models can be used for the design. For linear single-input/single-output systems this can be achieved by removing unstable zeros before inverting the plant. E.g., in the example above, the approximate inverse plant model would be:

$$u = \frac{(s-2) \cdot (s+3)}{(T \cdot s + 1)^2} \cdot y \quad (19.35)$$

For a non-linear plant, one might choose other outputs of the plant as inputs to the inverse model, since this might change the stability behavior of the inverse plant, see for example (Snell 2002). Alternatively, the plant might be modified before inversion. These advices are demonstrated by the crane example in Figure 19.21:

Figure 19.21: Crane consisting of a horizontal moving crab and a load on a rope.



The crane consists of a horizontally moving crab and a rope on which the load is attached. For simplicity, the load is modeled as a mass point. The crab is driven by the external force “ $f$ ”. The horizontal position of the crab “ $s_1$ ” and its derivative “ $v_1$ ” are measured. The goal is to move the load to a specified horizontal position “ $s_2$ ”.

For a non-linear disturbance observer, the inverse model from  $s_1$  to  $f$  is needed, since  $s_1$  is measured. The system is first linearized around the stationary position where the rope hangs vertically down ( $\varphi = 0^\circ$ ). The transfer function from  $f$  to  $s_1$  has 2 conjugate complex zeros on the imaginary axis, signaling an undamped oscillation of the inverse model which is not desired. This can be improved by including linear damping ( $= d \cdot \dot{\varphi}$ ) in the revolute joint for the inverse plant used in the controller. If the damping constant  $d$  is large enough, the two zeros on the imaginary axis are moved to the negative real axis. The disturbance observer is able to force the plant (that does have low damping) moving in such a way as if there would be high damping.

The major goal is to position the load, i.e., to control the horizontal position “ $s_2$ ” of the load. Therefore, the feedforward control should use the inverse model from  $s_2$  to  $f$ . The transfer function from  $f$  to  $s_2$  of the linearized model has no zeros and a relative degree of 4. Constructing the inverse model from the non-linear plant model requires, however, a filter of order 2 instead of 4 as suggested by the linearized model. Simulating the inverse model results in a division by zero if  $\varphi = 0^\circ$  or  $\varphi = 180^\circ$ .

To summarize, the structure of the inverse model equations is different at these two points and at  $\varphi \neq 0^\circ$  and  $\varphi \neq 180^\circ$  (the DAE index is 5 for  $\varphi \neq 0^\circ$  and  $\varphi \neq 180^\circ$  and the DAE index is 3 otherwise). Since the division by zero occurs when computing  $\ddot{\varphi}$ , the plant model should be changed to compute  $\ddot{\varphi}$  in a different way. This can be accomplished by taking the inertia of the load into consideration (previously it was neglected). With a non-zero inertia, the transfer function from  $f$  to  $s_2$  of the linearized plant has 2 conjugate complex zeros on the imaginary axis and a relative degree of 2. Again, by introducing damping in the revolute joint, these two zeros are moved to the negative real axis. It turns out that the inverse model is very *insensitive* with respect to the newly introduced load inertia.

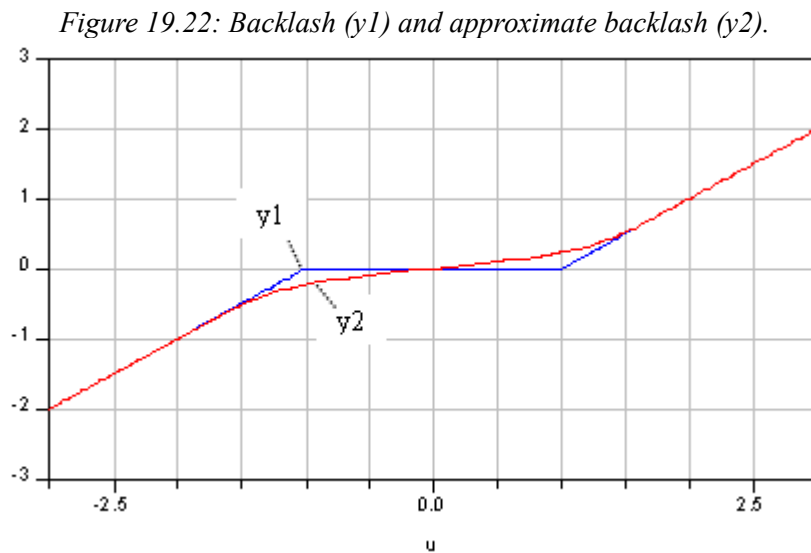
To summarize, for the crane example the inverse plant models from  $s_1$  to  $f$  and from  $s_2$  to  $f$  can be constructed by inverting a modified plant that has a load inertia and additionally damping in the revolute joint. A simpler alternative is also available: Before inversion, the angle  $\varphi$  is fixed to  $0^\circ$  and therefore  $s_1 = s_2$ , and the plant to be inverted is described by the following equation ( $m_{crab}$  is the mass of the crab and  $m_{load}$  is the mass of the load):

$$(m_{crab} + m_{load}) \cdot \ddot{s}_1 = f \quad (19.36)$$

which can be easily inverted. This example demonstrates that it might be necessary to slightly modify the original plant model in order that the inverse model of the plant can be used in a controller.

### 19.8.2 Equations that cannot be inverted

A plant may have equations that cannot be inverted. Examples are time delays, backlash, friction, hysteresis. This can be fixed by approximating the problematic elements in such a way that the resulting equation leads to a unique inverse.



A typical example is shown in Figure 19.22. The original backlash characteristic  $y_1 = f_1(u)$  is not invertible because for  $y_1 = 0$ , there are an infinite number of solutions ( $u = -1 \dots +1$ ). In Figure 19.22 an approximation  $y_2 = f_2(u)$  is shown that is strict monotonic and therefore the inverse function has a unique solution.

It might also occur that tables have to be inverted. Formally, a table in one dimension is defined as a function  $y = f(u)$ . Inversion of this function means to solve a non-linear equation. This can be often quite easily avoided by providing already the inverse tabulated values  $u = g(y)$  in the plant before inverting the plant model. The advantage is that the solution is faster and more robust. This problem was, e.g., encountered in (Steinhauser et. al. 2004), where control surface effectiveness of a military jet tended to have a local maximum as a function of the deflection. This was solved by adapting tables and internal limitations of control commands.

The inverse plant model may have also other singularities at particular operating points or regions that prevent an inversion, e.g., due to divisions by zero, singular linear or singular non-linear systems. The reason is that the corresponding inverse model has no or infinitely many solutions in particular points or in particular regions of the state space. Again, one remedy is to change the plant model before the inversion, e.g., by neglecting dynamic elements or by approximating components with functions that are less problematic to invert.

### 19.8.3 Actuator limits

Every control system is inherently limited by constraints in the actuator or other parts of the plant and therefore the question arises how to cope with these restrictions. When inverting a plant model, such constraints have to be *removed* before the inversion. Otherwise *no unique* solution of the inverse exists anymore, because there are infinitely many solutions when an actuator is in one of its limits. As a result, usually only the trivial action is possible to add appropriate limiters to the outputs of inverse models. This will only help for short-time violations of the constraints because the control system is effectively switched off when the actuators are in their limits.

The most effective way to cope with actuator constraints in any control system is to adapt the desired control signals, such as  $y_{c,d}(t)$  in Figure 19.9. In the most general case this means to solve a trajectory optimization problem, i.e., to determine actuator signals  $u(t)$  such that the plant outputs  $y_c(t)$  have a desired behavior, e.g., reaching the desired position in minimum time or with minimum energy, without violating the plant constraints. The result is used as  $y_{c,d}(t)$ . A typical example can be found in (Franke et. al. 2003). Note, if the plant is unstable and the inverse plant model is stable, it might be considerably simpler to solve the trajectory optimization problem with the inverse plant instead with the plant model. Usually, trajectory optimization problems are difficult to solve and therefore highly simplified plant models are used.

Take for example the crane model. The basic requirement is to move the crab from position  $s_1=a$  to  $s_1=b$  in a short time. The plant model is simplified by fixing the angle to  $\varphi = 0^\circ$  resulting in equation (19.36). Based on this equation, the actuator limit  $|f| \leq f_{max}$  can be directly transformed into a limit of the acceleration:

$$|\ddot{s}_1| \leq f_{max} / (m_{crab} + m_{load,max}) \quad (19.37)$$

Together with limits on the maximum speed, due to the maximum speed of the motor,  $|\dot{s}_1| \leq \dot{s}_{1,max}$ , and the requirement to move in minimum time from  $a$  to  $b$  it is straightforward to construct the analytic solution of the desired movement  $s_{1,d}(t)$ . This solution is, e.g., available via the block `Modelica.Blocks.Sources.KinematicPTP`. Note, the plant model used for the trajectory optimization problem and for the inverse plant model in the feedforward path according to Figure 19.9 are identical here. In such a case, the feedforward controller can be removed and can be replaced by the result of the trajectory optimization:  $s_{1,d}$  and  $f_{1,d} = (m_{crab} + m_{load,max}) \cdot \ddot{s}_{1,d}$ . For the trajectory optimization problem an  $f_{max}$  should be used that is, say, 10 % - 20 % smaller as the actual limit in order to provide some margin for the feedback controller.

If the desired control variables  $y_{c,d}(t)$  are not known in advance but generated online, e.g., by an operator, online optimization techniques have to be used: The operator request is reduced such that the plant constraints are fulfilled in the next sample time instant. A well known measure in flight control is the so-called daisy-chain. In case a control input saturates, a secondary, redundant control input is brought in that provides the remaining required control power. In (Steinhauser et. al. 2004) for example, lateral deflection of the thrust vector is used to yaw the aircraft in case the rudder saturates.

### 19.8.4 Real-time implementation

If inverse plant models are part of the controller, linear and non-linear systems of equations as well as non-linear differential equations might have to be solved in every sample interval of the controller. The techniques developed for hardware-in-the-loop simulations can be also applied for such an application. The methods described in section 13.4 are available in Dymola with the Dymola real-time option and can be applied by selecting the appropriate options when translating the inverse model (Simulation / Setup / Realtime / Inline integration method). Only fixed step integrators can be used for a real-time application. Via simulations, the appropriate step size of the integrator has to be determined.

### 19.8.5 Robustness

As already mentioned, the use of inverse model equations gives rise to robustness issues, since any mismatch

between the inverted model equations and the actual plant will leave part of the nonlinearities and couplings uncompensated. The usual approach is to provide robustness to model uncertainty via the (linear) feedback controller (Figure 19.9, 19.10, 19.11) or the disturbance observer (Figure 19.12). This can be done by application of a robust control synthesis technique (Adams and Banda 1993), or by robust parameter tuning in a classical structure, e.g. using multi-model techniques and enforcing sufficient stability margins (Looye 2001).

Tolerances on parameters in the model also appear in the inverse model equations. In (Looye 2001) it has been shown that these parameters may be very effectively used as additional tuning parameters in multi-objective optimization. The result is a model that is basically inverted at a location in the parameter space that provides the highest level of robustness.

### 19.8.6 Summary

Several control structures have been discussed that are based on non-linear inverse plant models. These structures are attractive since it is possible to cope directly with operating point dependencies. The difficult part to construct an inverse model can be performed automatically even for complex systems: The plant is modeled with Modelica, inputs and outputs are exchanged and a Modelica simulation environment, such as Dymola, generates automatically the appropriate C-code for the inverse plant model, including real-time integration algorithms. The generated code can be easily embedded into Simulink from Mathworks using Dymolas Simulink-export option. Via Mathworks Realtime-Workshop, the code can be finally downloaded to different target processors.

The presented controller structures can be used in all types of areas such as control of robots, vehicles, aircrafts, satellites, ships, motors, air conditioning systems. The most important requirement is that an appropriate plant model is available. Then, the inverse modeling approach is in principle fully automatic, although the practical application is usually more difficult. The essential issues have been discussed in section 19.8 and also possible remedies.





## Appendix

## Chapter 20 Contributors to Modelica

Many persons have contributed to the Modelica language definition and to the Modelica Standard Library. Find below a list of all contributors:

## Chapter 21 Modelica Built-In Functions and Operators

This chapter contains a summary of all built-in operators and built-in functions of Modelica 3.0:

- Elementary operators on page 227.
- Mathematical functions on page 229.
- Conversion functions on page 230.
- Derivative and special purpose operators on page 232.
- Event-related operators on page 232.
- Array operators on page 233.

This information is also available from the following package, which is provided in Dymola's package browser:

`ModelicaReference.Operators`

The tables in this chapter are partly a copy of the corresponding tables from the Modelica Language Specification 3.0 and/or from the `ModelicaReference` package.

### 21.1 Elementary Operators

*Table 21.1: Arithmetic Operators (operate on Real, Integer scalars and arrays)*

<i>Operator</i>	<i>Example</i>	<i>Description</i>
<code>+</code> , <code>-</code> , <code>.*</code> , <code>.-</code>	<code>a + b</code> <code>a .* b</code>	Addition and subtraction; element-wise on arrays.
<code>*</code>	<code>a * b</code>	Multiplication: scalar*array: element-wise multiplication vector*vector: vector product (element-wise multiplication and summing the result; result: scalar) matrix*matrix: matrix product vector*matrix: row-matrix*matrix (result: vector) matrix*vector: matrix*column-matrix (result: vector)
<code>/</code>	<code>a / b</code>	Division of two scalars or division of an array by a scalar is defined element-wise. The result is always of type Real, even if both arguments are of type Integer. In order to get Integer division with truncation the function <code>div()</code> .
<code>^</code>	<code>a^b</code>	Scalar Real power of a variable or scalar Integer power of a square matrix.
<code>.*</code> , <code>./</code> , <code>.^</code>	<code>a .* b</code>	Element-wise multiplication, division and exponentiation of scalars and arrays
<code>=</code>	<code>a*b = c+d</code>	Equal operator of an equation; element-wise on arrays. Assignment operator in the command window of Dymola.
<code>:=</code>	<code>a := c+d</code>	Assignment operator; element-wise on arrays.

*Table 21.2: Relational Operators (operate on Real, Integer, Boolean, String, enumeration scalars).  
 For Boolean: false < true;  
 for String: comparison based on lexicographical order where "I" < "A" < "a";  
 For enumeration: comparison according to definition order.*

Operator	Example	Description
==	a == b	True, if equal (for strings: true if identical characters). For Real, only allowed inside functions.
<>	a <> b	True, if not equal (for strings: true if not identical characters). For Real, only allowed inside functions.
<	a < b	True, if a less than b.
<=	a <= b	True, if a less or equal b.
>	a > b	True, if a greater than b.
>=	a >= b	True, if a greater or equal b.

*Table 21.3: Boolean Operators (operate on Boolean scalars or element-wise on Boolean arrays)*

Operator	Example	Description
<b>and</b>	a <b>and</b> b	logical and ( {true, false} and {true, true} = {true, false} )
<b>or</b>	a <b>or</b> b	logical or
<b>not</b>	<b>not</b> a	logical not

*Table 21.4: Other operators*

Operator	Example	Description
[..]	Real A[2,2]; A[1,2] = 3; A=[1,2;3,4];	Array dimension (in declaration) or array index or matrix constructor ("," separates rows, ";" separates columns).
{..}	A={{1,2},{3,4}}	Array constructor; every {...} adds one dimension.
"..."	"string value" "string \"value\""	String literal (\ is used inside a string for ").
+	"abc" + "def"	Concatenation of string scalars or string arrays (element-wise)

*Table 21.5: Operator precedence in expressions  
 (operators with higher precedence are evaluated first; e.g. "[" has higher precedence as "\*")*

Operator	Example	Description
(..)	2*(3+5)	Parenthesis
arrayName[..]	arr[index]	Postfix array index operator
name.name	a.b.c	Postfix access operator
funcName(function-arguments)	sin(4.36)	Postfix function call
{expr} [expr]	{2,3} [5,6]	Array construction/concatenation

<i>Operator</i>	<i>Example</i>	<i>Description</i>
<code>[expr;expr;...]</code>	<code>[2,3; 7,8]</code>	
<code>^</code>	<code>2^3</code>	Exponentiation
<code>*, /, .*, ./</code>	<code>2*3, 2/3, [1,2;3,4].*[2,3;5,6]</code>	Multiplicative and array element-wise multiplicative
<code>+, -, +expr, -expr, .+, .-</code>	<code>a+b, a-b, +a, -a [1,2;3,4].+[2,3;5,6]</code>	Additive and array element-wise additive
<code>&lt;, &lt;=, &gt;, &gt;=, ==, &lt;&gt;</code>	<code>a&lt;b, a&lt;=b, a&gt;b</code>	Relational
<code>not expr</code>	<code>not b1</code>	Unary negation
<code>and</code>	<code>b1 and b2</code>	Logical and
<code>or</code>	<code>b1 or b2</code>	Logical or
<code>expr : expr : expr</code>	<code>1:5, start:step:stop</code>	Array range or vector construction
<code>if expr then expr else expr</code>	<code>if b then 3 else x</code>	Conditional
<code>ident = expr</code>	<code>x = 2.26</code>	Named argument or modifier

Equality “=” and assignment “:=” are not expression operators since they are allowed only in equations and in assignment statements respectively. All binary expression operators are left associative. The unary minus and plus in Modelica is slightly different than in, say MATLAB, since the following expressions are illegal in Modelica (whereas in MATLAB these are valid expressions):

```

2*-2  // = -4 in MATLAB; is illegal in Modelica
--2   // =  2 in MATLAB; is illegal in Modelica
++2   // =  2 in MATLAB; is illegal in Modelica
2--2  // =  4 in MATLAB; is illegal in Modelica

```

## 21.2 Mathematical Functions

Table 21.6: Mathematical functions.

*All functions have Real or Integer input arguments, have Real output arguments, are vectorizable and none triggers an event.*

<i>Operator</i>	<i>Description</i>
<b>abs</b> (x)	Is expanded into “ <b>noEvent</b> ( <b>if</b> x >= 0 <b>then</b> x <b>else</b> -x)”.
<b>asin</b> (x)	Inverse sine ( $-1 \leq x \leq 1$ ).
<b>acos</b> (x)	Inverse cosine ( $-1 \leq x \leq 1$ ).
<b>atan</b> (x)	Inverse tangent. Returns result in the range: $-\pi/2 \dots +\pi/2$
<b>atan2</b> (x, y)	Four quadrant inverse tangent, x and y not both zero ( $\tan(\cdot) = x/y$ ). Returns result in the range: $-\pi \dots +\pi$
<b>cos</b> (x)	cosine (x is in [rad])
<b>cosh</b> (x)	hyperbolic cosine.
<b>exp</b> (x)	exponential of x, base <i>e</i> .
<b>log</b> (x)	natural (base <i>e</i> ) logarithm ( $x > 0$ )
<b>log10</b> (x)	base 10 logarithm ( $x > 0$ )
<b>sign</b> (x)	Is expanded into “ <b>noEvent</b> ( <b>if</b> x>0 <b>then</b> 1 <b>else if</b> x<0 <b>then</b> -1 <b>else</b> 0)”.

Operator	Description
<b>sin</b> ( $x$ )	sine ( $x$ is in [rad])
<b>sinh</b> ( $x$ )	hyperbolic sine
<b>sqrt</b> ( $x$ )	Returns the square root of $x$ if $x \geq 0$ , otherwise an error occurs.
<b>tan</b> ( $x$ )	tangent ( $x$ is in [rad] and shall not be: $-\pi/2, \pi/2, 3\pi/2, \dots$ )
<b>tanh</b> ( $x$ )	hyperbolic tangent

### 21.3 Conversion Functions

Table 21.7: Conversion functions.  
All functions are vectorizable and trigger events.

Operator	Description
<b>ceil</b> ( $x$ )	Returns the smallest integer not less than $x$ . Result and argument shall have type Real. [Note, outside of a when-clause state events are triggered when the return value changes discontinuously.]. Examples: <b>ceil</b> (2.3) = 3.0, <b>ceil</b> (-2.3) = -2.0
<b>div</b> ( $x, y$ )	Returns the algebraic quotient $x/y$ with any fractional part discarded (also known as truncation toward zero). [Note: this is defined for / in C99; in C89 the result for negative numbers is implementation-defined, so the standard function div() must be used.]. Result and arguments shall have type Real or Integer. If either of the arguments is Real the result is Real otherwise Integer. [Note, outside of a when-clause state events are triggered when the return value changes discontinuously.]. Examples: <b>div</b> (4, 3) = 1, <b>div</b> (-4, 3) = -1, <b>div</b> (7.5, 2) = 3.0
<b>floor</b> ( $x$ )	Returns the largest integer not greater than $x$ . Result and argument shall have type Real. [Note, outside of a when-clause state events are triggered when the return value changes discontinuously.]. Examples: <b>floor</b> (2.3) = 2.0, <b>floor</b> (-2.3) = -3
<b>integer</b> ( $x$ )	Returns the largest integer not greater than $x$ . The argument shall have type Real. The result has type Integer. [Note, outside of a when-clause state events are triggered when the return value changes discontinuously.]. Examples: <b>integer</b> (2.3) = 2, <b>integer</b> (-2.3) = -3
<b>Integer</b> ( $e$ )	<u>Not (yet) supported by Dymola 7.0</u> Returns the ordinal number of the enumeration value $e$ . Example: <pre>type Day = enumeration(Mo, Tue, Wed, Thur, Fr, Sa, Su); Day day; Integer w = Integer(day.Wed); // = 3</pre>
<b>mod</b> ( $x, y$ )	Returns the integer modulus of $x/y$ , i.e. $\text{mod}(x, y) = x - \text{floor}(x/y) * y$ . Result and arguments shall have type Real or Integer. If either of the arguments is Real the result is Real otherwise Integer. [Note, outside of a when-clause state events are triggered when the return value changes discontinuously.]. Examples: <b>mod</b> (3, 1.4) = 0.2, <b>mod</b> (-3, 1.4) = 1.2, <b>mod</b> (3, -1.4) = -1.2]
<b>rem</b> ( $x, y$ )	Returns the integer remainder of $x/y$ , such that $\text{div}(x, y) * y + \text{rem}(x, y) = x$ . Result and arguments shall have type Real or Integer. If either of the arguments is Real the result is Real otherwise Integer. [Note, outside of a when-clause state events are triggered when the return value changes discontinuously.]. Examples: <b>rem</b> (3, 1.4) = 0.2, <b>rem</b> (-3, 1.4) = -0.2]

Additionally to the above conversion functions, function “String(...)” converts a Real, Integer, Boolean and enumeration variable in to its String representation. This function call is not vectorizable. The following

function calls are supported (arguments with a default value need not be provided when calling the function; in such a case the default value is used):

```
String(b_expr, minimumLength=0, leftJustified=true)
String(i_expr, minimumLength=0, leftJustified=true)
String(e_expr, minimumLength=0, leftJustified=true)
String(r_expr, significantDigits=6, minimumLength=0,
        leftJustified=true)
String(r_expr, format)
```

The arguments have the following meaning:

Table 21.8: Optional arguments of operator “String(..)”.

Arguments	Description								
Boolean b_expr	Boolean expression								
Integer i_expr	Integer expression								
<b>type</b> e_expr = <b>enumeration</b> (..)	Enumeration expression								
Real r_expr	Real expression								
Integer minimumLength = 0	Minimum length of the resulting string. If necessary, the blank character is used to fill up unused space.								
Boolean leftJustified= <b>true</b>	If <b>true</b> , the converted result is left justified; if <b>false</b> , it is right justified in the string.								
Integer significantDigits = 6	Defines the number of significant digits in the result string (e.g. "12.3456", "0.0123456", "12345600", "1.23456E-10")								
String format	Defines the string formatting according to ANSI-C without "%" and "*" character (e.g. ".6g", "14.5e", "+6f"). In particular (“[...]” is optional): format = "[<flags>] [<width>] [.<precision>] <conversion>" with <table border="1" data-bbox="619 1352 1433 2011"> <tr> <td>&lt;flags&gt;</td><td>zero, one or more of "-": left adjustment of the converted number "+": number will always be printed with a sign "0": padding to the field width with leading zeros</td></tr> <tr> <td>&lt;width&gt;</td><td>Minimum field width. The converted number will be printed in a field at least this wide and wider if necessary. If the converted number has fewer characters it will be padded on the left (or the right depending on &lt;flags&gt;) with blanks or 0 (depending on &lt;flags&gt;).</td></tr> <tr> <td>&lt;precision&gt;</td><td>The number of digits to be printed after the decimal point for e, E, or f conversions, or the number of significant digits for g or G conversions.</td></tr> <tr> <td>&lt;conversion&gt;</td><td>= "e": Exponential notation using a lower case e = "E": Exponential notation using an upper case E = "f": Fixed point notation = "g": Either "e" or "f" = "G": Same as "g", but with upper case E</td></tr> </table>	<flags>	zero, one or more of "-": left adjustment of the converted number "+": number will always be printed with a sign "0": padding to the field width with leading zeros	<width>	Minimum field width. The converted number will be printed in a field at least this wide and wider if necessary. If the converted number has fewer characters it will be padded on the left (or the right depending on <flags>) with blanks or 0 (depending on <flags>).	<precision>	The number of digits to be printed after the decimal point for e, E, or f conversions, or the number of significant digits for g or G conversions.	<conversion>	= "e": Exponential notation using a lower case e = "E": Exponential notation using an upper case E = "f": Fixed point notation = "g": Either "e" or "f" = "G": Same as "g", but with upper case E
<flags>	zero, one or more of "-": left adjustment of the converted number "+": number will always be printed with a sign "0": padding to the field width with leading zeros								
<width>	Minimum field width. The converted number will be printed in a field at least this wide and wider if necessary. If the converted number has fewer characters it will be padded on the left (or the right depending on <flags>) with blanks or 0 (depending on <flags>).								
<precision>	The number of digits to be printed after the decimal point for e, E, or f conversions, or the number of significant digits for g or G conversions.								
<conversion>	= "e": Exponential notation using a lower case e = "E": Exponential notation using an upper case E = "f": Fixed point notation = "g": Either "e" or "f" = "G": Same as "g", but with upper case E								

Examples:

```

Real    r  = 1.23456789;
Boolean b  = false;
String  sr = "r = " + String(r);    // sr = "r = 1.23457"
String  sb = "b = " + String(b);    // sb = "b = false"
String  sr2 = String(r, significantDigits=16, minLength=15,
                    leftJustified=false);
           // = "      1.23456789"
String  sr3 = String(r, format=".15g"); // = "1.23456789"

```

## 21.4 Derivative and Special Purpose Operators with Function Syntax

Table 21.9: Derivative and special purpose operators with function syntax

Operator	Description
<b>der</b> (expr)	The time derivative of expr. If the expression expr is a scalar it needs to be a subtype of Real. The expression and all its subexpressions must be differentiable. If expr is an array, the operator is applied to all elements of the array. For non-scalar arguments the function is vectorized. For Real parameters and constants the result is a zero scalar or array of the same size as the variable.
<b>delay</b> (expr, delayTime, delayMax) <b>delay</b> (expr, delayTime)	Returns: expr(time-delayTime) for time > time.start + delayTime and expr(time.start) for time <= time.start + delayTime. The arguments, i.e., expr, delayTime and delayMax, need to be subtypes of Real. DelayMax needs to be additionally a parameter expression. The following relation shall hold: 0 <= delayTime <= delayMax, otherwise an error occurs. If delayMax is not supplied in the argument list, delayTime need to be a parameter expression. For non-scalar arguments the function is vectorized.
<b>cardinality</b> (c)	This is a <i>deprecated operator</i> . It should no longer be used, since it will be removed in one of the next Modelica releases. Returns the number of (inside and outside) occurrences of connector instance c in a <b>connect</b> -equation as an Integer number.
<b>semiLinear</b> (x, positiveSlope, negativeSlope)	Returns: <b>if</b> x>=0 <b>then</b> positiveSlope*x <b>else</b> negativeSlope*x The result is of type Real. For non-scalar arguments the function is vectorized. There are specialized symbolic transformation rules defined, if x=0. Due to the new method to handle fluid flow with <b>stream</b> variables, this operator is no longer needed and should no longer be used

## 21.5 Event-Related Operators with Function Syntax

Table 21.10: Event related operators with function syntax  
(operators **noEvent**, **pre**, **edge**, and **change**, are vectorizable)

Operator	Description
<b>initial</b> ()	Returns true during the initialization phase and false otherwise [ <i>thereby triggering a time event at the beginning of a simulation</i> ].
<b>terminal</b> ()	Returns true at the end of a successful analysis [ <i>thereby ensuring an event at the end of a successful simulation</i> ]



Operator	Description
<b>noEvent</b> ( <i>expr</i> )	Real elementary relations within <i>expr</i> are taken literally, i.e., no state or time event is triggered. This operator must be used if the definition range of a variable is left and this would trigger an error. In the following example <b>noEvent</b> (..) is used to avoid a division by zero: $y = \text{noEvent}(\text{if } x > \text{eps} \text{ then } 1/x \text{ else } 1/\text{eps});$
<b>smooth</b> ( <i>p</i> , <i>expr</i> )	<b>If</b> $p \geq 0$ <b>then</b> <b>smooth</b> ( <i>p</i> , <i>expr</i> ) returns <i>expr</i> and states that <i>expr</i> is <i>p</i> times continuously differentiable, i.e.: <i>expr</i> is continuous in all real variables appearing in the expression and all partial derivatives with respect to all appearing real variables exist and are continuous up to order <i>p</i> . The only allowed types for <i>expr</i> in <b>smooth</b> are: Real expressions, arrays of allowed expressions, and records containing only components of allowed expressions. Examples: $y1 = \text{smooth}(1, \text{if } x > 0 \text{ then } x^2 \text{ else } x^3);$ $y2 = \text{smooth}(1, \text{noEvent}(\text{if } x < 0 \text{ then } 0 \text{ else } \text{sqrt}(x) * x));$ // noEvent is necessary.
<b>sample</b> ( <i>start</i> <i>interval</i> )	Returns true and triggers time events at time instants $\text{start} + i * \text{interval} \quad (i=0, 1, \dots).$ During continuous integration the operator returns always <b>false</b> . The starting time <b>start</b> and the sample interval <b>interval</b> need to be parameter expressions and need to be a subtype of Real or Integer. Example: <pre>when sample(0, 0.1) then ... // executed every 0.1 s exactly once end when;</pre>
<b>pre</b> ( <i>y</i> )	Returns the “left limit” $y(t^{\text{pre}})$ of variable <i>y</i> ( <i>t</i> ) at a time instant <i>t</i> . At an event instant, $y(t^{\text{pre}})$ is the value of <i>y</i> after the last event iteration at time instant <i>t</i> . The <b>pre</b> () operator can be applied if the following three conditions are fulfilled simultaneously: (a) variable <i>y</i> is a subtype of a simple type (Real, Integer, Boolean, String), (b) <i>y</i> is a discrete-time expression, (c) the operator is <i>not</i> applied in a function class. (d) Before initialization starts, <b>pre</b> ( <i>y</i> ) = <i>y</i> .start.
<b>edge</b> ( <i>b</i> )	Is expanded into “( <i>b</i> <b>and not pre</b> ( <i>b</i> ))” for Boolean variable <i>b</i> . The same restrictions as for the <b>pre</b> () operator apply (e.g. not to be used in function classes).
<b>change</b> ( <i>v</i> )	Is expanded into “( <i>v</i> <> <b>pre</b> ( <i>v</i> ))”. The same restrictions as for the <b>pre</b> () operator apply.
<b>reinit</b> ( <i>x</i> , <i>expr</i> )	In the body of a <b>when</b> clause, reinitializes <i>x</i> with <i>expr</i> at an event instant. <i>x</i> is a Real variable (resp. an array of Real variables, in which case vectorization applies) that must be selected as a <i>state</i> (resp., states) at least when the enclosing <b>when</b> clause becomes active. <i>expr</i> needs to be type-compatible with <i>x</i> . The <b>reinit</b> operator can only be applied <i>once</i> for the same variable (resp. array of variables). It can only be applied in the body of a <b>when</b> clause.

## 21.6 Array Operators

Table 21.11: Operations and functions for arrays

Operator	Description
Array dimension and size functions	

<i>Operator</i>	<i>Description</i>
<b>ndims</b> (A)	Returns the number of dimensions k of array expression A, with $k \geq 0$ .
<b>size</b> (A, i)	Returns the size of dimension i of array expression A where i shall be $> 0$ and $\leq \mathbf{ndims}(A)$ .
<b>size</b> (A)	Returns a vector of length <b>ndims</b> (A) containing the dimension sizes of A.
<i>Conversion functions between dimensions</i>	
<b>scalar</b> (A)	Returns the single element of array A. <b>size</b> (A, i) = 1 is required for $1 \leq i \leq \mathbf{ndims}(A)$ .
<b>vector</b> (A)	Returns a 1-vector, if A is a scalar and otherwise returns a vector containing all the elements of the array, provided there is at most one dimension size $> 1$ .
<b>matrix</b> (A)	Returns <b>promote</b> (A,2), if A is a scalar or vector and otherwise returns the elements of the first two dimensions as a matrix. <b>size</b> (A,i) = 1 is required for $2 < i \leq \mathbf{ndims}(A)$ .
<i>Specialized array constructor functions</i>	
<b>identity</b> (n)	Returns the $n \times n$ Integer identity matrix, with ones on the diagonal and zeros at the other places.
<b>diagonal</b> (v)	Returns a square matrix with the elements of vector v on the diagonal and all other elements zero.
<b>zeros</b> ( $n_1, n_2, \dots$ )	Returns the $n_1 \times n_2 \times \dots$ Integer array with all elements equal to zero ( $n_i \geq 0$ ).
<b>ones</b> ( $n_1, n_2, \dots$ )	Return the $n_1 \times n_2 \times \dots$ Integer array with all elements equal to one ( $n_i \geq 0$ ).
<b>fill</b> (s, $n_1, n_2, \dots$ )	Returns the $n_1 \times n_2 \times \dots$ array with all elements equal to scalar or array expression s ( $n_i \geq 0$ ). The returned array has the same type as s.
<b>linspace</b> (x1, x2, n)	Returns a Real vector with n equally spaced elements, such that $v = \mathbf{linspace}(x1, x2, n)$ , $v[i] = x1 + (x2 - x1) * (i - 1) / (n - 1)$ for $1 \leq i \leq n$ . It is required that $n \geq 2$ . The arguments x1 and x2 shall be scalar Real or Integer expressions.
<i>Matrix and vector algebra functions</i>	
<b>transpose</b> (A)	Permutes the first two dimensions of array A. It is an error, if array A does not have at least 2 dimensions. If A is a matrix, <b>transpose</b> (A) is the transpose of the matrix.
<b>outerProduct</b> (v1, v2)	Returns the outer product of vectors v1 and v2 (= <b>matrix</b> (v) * <b>transpose</b> ( <b>matrix</b> (v) ) ).
<b>symmetric</b> (A)	Returns a matrix where the diagonal elements and the elements above the diagonal are identical to the corresponding elements of matrix A and where the elements below the diagonal are set equal to the elements above the diagonal of A, i.e., <pre> B := <b>symmetric</b> (A) -&gt;     B[i,j] := A[i,j], if i &lt;= j,     B[i,j] := A[j,i], if i &gt; j.</pre>

<i>Operator</i>	<i>Description</i>
<b>cross</b> ( <i>x</i> , <i>y</i> )	Returns the vector cross product of the vectors <i>x</i> and <i>y</i> , i.e. $\mathbf{cross}(x, y) = \mathbf{vector} \left( \begin{bmatrix} x[2]*y[3]-x[3]*y[2]; \\ x[3]*y[1]-x[1]*y[3]; \\ x[1]*y[2]-x[2]*y[1] \end{bmatrix} \right);$
<b>skew</b> ( <i>x</i> )	Returns the 3 x 3 skew symmetric matrix associated with a 3-vector, i.e., $\mathbf{cross}(x, y) = \mathbf{skew}(x) * y;$ $\mathbf{skew}(x) = \begin{bmatrix} 0, & -x[3], & x[2]; \\ x[3], & 0, & -x[1]; \\ -x[2], & x[1], & 0 \end{bmatrix};$
<i>Array reduction functions and operators (reduces an array or several scalars to one value)</i>	
<b>min</b> ( <i>A</i> )	Returns the smallest element of array expression <i>A</i> .
<b>min</b> ( <i>x</i> , <i>y</i> )	Returns the smallest element of the scalars <i>x</i> and <i>y</i> .
<b>min</b> ( <i>e</i> ( <i>i</i> , ..., <i>j</i> ) <b>for</b> <i>i in u</i> , ..., <i>j in v</i> )	Returns the smallest value of the scalar expression <i>e</i> ( <i>i</i> , ..., <i>j</i> ) evaluated for all combinations of <i>i in u</i> , ..., <i>j in v</i> . Example: $\mathbf{min}(A[i, j] * B[j, i] \text{ for } i \text{ in } 1:n1, j \text{ in } 1:n2)$
<b>max</b> ( <i>A</i> )	Returns the largest element of array expression <i>A</i> .
<b>max</b> ( <i>x</i> , <i>y</i> )	Returns the largest element of the scalars <i>x</i> and <i>y</i> .
<b>max</b> ( <i>e</i> ( <i>i</i> , ..., <i>j</i> ) <b>for</b> <i>i in u</i> , ..., <i>j in v</i> )	Returns the largest value of the scalar expression <i>e</i> ( <i>i</i> , ..., <i>j</i> ) evaluated for all combinations of <i>i in u</i> , ..., <i>j in v</i> . Example: $\mathbf{max}(A[i, j] * B[j, i] \text{ for } i \text{ in } 1:n1, j \text{ in } 1:n2)$
<b>sum</b> ( <i>A</i> )	Returns the scalar sum of all the elements of array expression <i>A</i> : $A[1, \dots, 1] + A[2, \dots, 1] + \dots + A[\text{end}, \dots, 1] + A[\text{end}, \dots, \text{end}]$
<b>sum</b> ( <i>e</i> ( <i>i</i> , ..., <i>j</i> ) <b>for</b> <i>i in u</i> , ..., <i>j in v</i> )	Returns the sum of the expression <i>e</i> ( <i>i</i> , ..., <i>j</i> ) evaluated for all combinations of <i>i in u</i> , ..., <i>j in v</i> : $e(u[1], \dots, v[1]) + e(u[2], \dots, v[1]) + \dots + e(u[\text{end}], \dots, v[1]) + \dots + e(u[\text{end}], \dots, v[\text{end}])$ <p>The type of <b>sum</b>(<i>e</i>(<i>i</i>, ..., <i>j</i>) <b>for</b> <i>i in u</i>, ..., <i>j in v</i>) is the same as the type of <i>e</i>(<i>i</i>, ..., <i>j</i>). Example:  <math display="block">\mathbf{sum}(A[i, j] * B[j, i] \text{ for } i \text{ in } 1:n1, j \text{ in } 1:n2)</math></p>
<b>product</b> ( <i>A</i> )	Returns the scalar product of all the elements of array expression <i>A</i> . $A[1, \dots, 1] * A[2, \dots, 1] * \dots * A[\text{end}, \dots, 1] * A[\text{end}, \dots, \text{end}]$
<b>product</b> ( <i>e</i> ( <i>i</i> , ..., <i>j</i> ) <b>for</b> <i>i in u</i> , ..., <i>j in v</i> )	Returns the product of the scalar expression <i>e</i> ( <i>i</i> , ..., <i>j</i> ) evaluated for all combinations of <i>i in u</i> , ..., <i>j in v</i> : $e(u[1], \dots, v[1]) * e(u[2], \dots, v[1]) * \dots * e(u[\text{end}], \dots, v[1]) * \dots * e(u[\text{end}], \dots, v[\text{end}])$ <p>The type of <b>product</b>(<i>e</i>(<i>i</i>, ..., <i>j</i>) <b>for</b> <i>i in u</i>, ..., <i>j in v</i>) is the same as the type of <i>e</i>(<i>i</i>, ..., <i>j</i>). Example:  <math display="block">\mathbf{product}(A[i, j] * B[j, i] \text{ for } i \text{ in } 1:n1, j \text{ in } 1:n2)</math></p>

## Chapter 22 Literature

- Adams R.J., and S. Banda (1993): **Robust Flight Control Design Using Dynamic Inversion and Structured Singular Value Synthesis**. IEEE Transactions on Control Systems Technology, 1(2):80-92, June 1993.
- Årzen K.-E., R. Olsson, and J. Akesson (2002): **Grafchart for Procedural Operator Support Tasks**. Proceedings of the 15th IFAC World Congress, Barcelona, Spain
- Aström K.J., and B. Wittenmark (1997): **Computer Controlled Systems: Theory and Design**. Prentice Hall. 3rd edition.
- Barton P. (1992): **The Modelling and Simulation of Combined Discrete/Continuous Processes**. PhD thesis, University of London, Imperial College.
- Bauschat M., W. Mönnich, D. Willemsen, and G. Looye (2001): **Flight testing Robust Autoland Control Laws**. In Proceedings of the AIAA Guidance, Navigation and Control Conference, Montreal CA.
- Benveniste A., P. Caspi, S. Edwards, N. Halbwachs, P. le Guernic, and R. de Simone (2003): **The Synchronous Languages, 12 Years Later**. Proceedings of the IEEE, Vol. 91, No. 1, pp. 64-83.
- Berry G. (1999): **The Constructive Semantics of Pure Esterel**.  
<http://www.esterel-technologies.com/files/book.zip>
- Carpanzano E., and R. Girelli (1997): **The Tearing Problem: Definition, Algorithm and Application to Generate Efficient Computational Code from DAE Systems**. Proc. of 2nd Mathmod Vienna, IMACS Symposium on Mathematical Modeling, Vienna. pp. 1039-1046.
- Clauss C., J. Haase, G. Kurth, and P. Schwarz (1995). **Extended Amittance Description of Nonlinear n-Poles**. Archiv für Elektronik und Übertragungstechnik/International Journal of Electronics and Communications, vol. 40, pp. 91–97.
- Dymola (2008). **Dymola Version 7.1**. Dynasim AB, Lund, Sweden. Homepage: <http://www.dynasim.se/>.
- Enns D., D. Bugajski, R. Hendrick, and G. Stein (1994). **Dynamic Inversion: An Evolving Methodology for Flight Control Design**. In AGARD Conference Proceedings 560: Active Control Technology: Applications and Lessons Learned, pages 7-1 – 7-12, Turin, Italy, May 1994.
- Elmqvist H. (1978): **A Structured Model Language for Large Continuous Systems**. Dissertation. Report CODEN:LUTFD2/(TFRT--1015), Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, 1978.
- Elmqvist H. (1992): **An Object and Data-Flow based Visual Language for Process Control**. ISA/92-Canada Conference & Exhibit, Toronto, Canada, Instrument Society of America.
- Elmqvist H., F. Cellier and M. Otter (1993): **Object-Oriented Modeling of Hybrid Systems**. Proceedings ESS'93, European Simulation Symposium, Delft, The Netherlands, pp. xxxi-xli.
- Elmqvist H., S.E. Mattsson and M. Otter (2001): **Object-Oriented and Hybrid Modeling in Modelica**. Journal Europeen des systemes automatises, 35, 1/2001, pp. 1 a X.
- Franke R., M. Rode M., and K. Krüger (2003): **On-line Optimization of Drum Boiler Startup**. 3rd Int. Modelica Conference, Linköping, Nov. 3-4, pp. 287 – 296. Download: <http://www.Modelica.org/Conference2003/papers.shtml>.
- Föllinger O. (1994): **Regelungstechnik**. Hütthig Verlag, 8. Auflage.
- Föllinger O. (1998): **Nichtlineare Regelungen I**. Oldenbourg Verlag, 8. Auflage.
- Gautier T., P. le Guernic, and O. Mafféis (1994): **For a New Real-Time Methodology**. Publication Interne No. 870, Institut de Recherche en Informatique et Systemes Aleatoires, Campus de Beaulieu, 35042 Rennes Cedex, France.  
<ftp://ftp.inria.fr/INRIA/publication/publi-pdf/RR/RR-2364.pdf>
- Halbwachs N., P. Caspi, P. Raymond, and D. Pilaud (1991): **The synchronous data flow programming language LUSTRE**. Proceedings of IEEE, vol. 79, pp. 1305–1321.  
<http://www.esterel-technologies.com/files/LUSTRE-synchronous-programming-language.pdf>

- Halbwachs N. (1993): **Synchronous Programming of Reactive Systems**. Springer Verlag.  
<http://www.esterel-technologies.com/files/synchronous-programming-of-reactive-systems-tutorial-and-references.pdf>
- Holman J. (2001): **Heat Transfer**. 9<sup>th</sup> Edition, McGraw-Hill.
- Isidori A. (1995): **Nonlinear Control Systems**. 3rd Edition, Springer Verlag.
- Isidori A. (2006): **Nonlinear Control Systems II**. Springer Verlag.
- Kral C., A. Haumer, M. Plainer (2005): **Simulation of a thermal model of a surface cooled squirrel cage induction machine by means of the SimpleFlow-library**. Proceedings of the 4<sup>th</sup> International Modelica Conference, Hamburg, Germany, ed. G. Schmitz, pp. 213-218.  
[http://www.modelica.org/events/Conference2005/online\\_proceedings/Session3/Session3b1.pdf](http://www.modelica.org/events/Conference2005/online_proceedings/Session3/Session3b1.pdf)
- Kreisselmeier G. (1999): **Struktur mit zwei Freiheitsgraden**. Automatisierungstechnik at 6, pp. 266-269.
- Kuhn M., M. Otter, and L. Raulin. (2008): A Multi Level Approach for Aircraft Electrical Systems Design. Proceedings of the 6<sup>th</sup> International Modelica Conference, Bielefeld, Germany, ed. B. Bachmann, pp. 95-101.  
<http://www.modelica.org/events/modelica2008/Proceedings/sessions/session1d1.pdf>
- Lötstedt, P (1982): **Mechanical systems of rigid bodies subject to unilateral constraints**. SIAMJ. Appl. Math., vol. 42, pp. 281–296.
- Looye G. (2001): **Design of Robust Autopilot Control Laws with Nonlinear Dynamic Inversion**. Automatisierungstechnik at 49-12, p. 523-531.
- Mah R.S.H. (1990): **Chemical Process Structures and Information Flows**. Butterworth publisher.
- MathModelica (2006). Homepage: <http://www.mathcore.com/products/mathmodelica/>.
- Mattsson S. E., and G. Söderlind (1993): **Index reduction in differential-algebraic equations using dummy derivatives**. SIAM Journal of Scientific and Statistical Computing, Vol. 14, pp. 677-692.
- Mattsson S. E., H. Olsson and H. Elmqvist (2000). **Dynamic Selection of States in Dymola**. Proceedings of the Modelica Workshop 2000. pp. 61-67. <http://www.modelica.org/Workshop2000/papers/Mattsson.pdf>.
- Modelica Association (2007): **Modelica – A Unified Object-Oriented Language for Physical Systems Modeling, Language Specification, Version 3.0**. <http://www.modelica.org/documents/ModelicaSpec30.pdf>.
- Mosterman P., and G. Biswas (1996): **A Formal Hybrid Modeling Scheme for Handling Discontinuities in Physical System Models**. Proceedings of AAAI-96, Portland, OR, USA, pp. 985–990.
- Mosterman P., M. Otter, and H. Elmqvist. (1998): **Modeling Petri Nets as Local Constraint Equations for Hybrid Systems using Modelica**. Proceedings of SCSC’98, Reno, Nevada, USA, Society for Computer Simulation International, pp. 314–319.
- Mugica F., and F.E. Cellier (1994): **Automated synthesis of a fuzzy controller for cargo ship steering by means of qualitative simulation**. Proceedings of the European Simulation MultiConference (ESM’94), Barcelona, Spain, pp. 523-528,
- Olsson H., M. Otter, S.E. Mattsson, and H. Elmqvist (2008): **Balanced Models in Modelica 3.0 for Increased Model Quality**. Proceedings of the 6<sup>th</sup> International Modelica Conference, Bielefeld, Germany, ed. B. Bachmann, pp. 21-33. <http://www.modelica.org/events/modelica2008/Proceedings/sessions/session1a3.pdf>.
- OpenModelica (2006). Homepage: <http://www.OpenModelica.org>.
- Otter M., and F.E. Cellier (1996): **Software for Modeling and Simulating Control Systems**. The Control Handbook, by W.S. Levine (editor), CRC Press, pp. 415 – 428.
- Otter M., H. Elmqvist, and S.E. Mattsson (1999): **Hybrid Modeling in Modelica based on the Synchronous Data Flow Principle**. IEEE International Symposium on Computer Aided Control System Design (CACSD’99), Hawaii, U.S.A.
- Otter M., K.-E. Årzén, and I. Dressler (2005): **StateGraph – A Modelica Library for Hierarchical State Machines**. Proceedings of the 4th International Modelica Conference, Hamburg, Germany, ed. G. Schmitz, pp. 569-578. [http://www.modelica.org/events/Conference2005/online\\_proceedings/Session7/Session7b2.pdf](http://www.modelica.org/events/Conference2005/online_proceedings/Session7/Session7b2.pdf)
- Pantelides C. (1988): **The consistent initialization of differential-algebraic systems**. SIAM Journal of Scientific and Statistical Computing, pp. 213-231.
- Roussel J.-M., J.-M. Faure (2002): **An algebraic approach for PLC programs verification**. Proc. of the Sixth International Workshop on Discrete Event Systems (WODES’02). 0-7695-1683-1/02
- Pfeiffer F. and C. Glocker (1996): **Multibody Dynamics with Unilateral Contacts**. John Wiley.

- Schumacher J.M. and A.J. van der Schaft (1998): **Complementarity Modeling of Hybrid Systems**. IEEE Transactions on Automatic Control, vol. 43, pp. 483–490.
- SIMPACK (2006). Homepage: <http://www.simpack.com>.
- Snell A. (2002): **Decoupling of Nonminimum Phase Plants and Application to Flight Control**. AIAA-2002-4760 AIAA Guidance, Navigation, and Control Conference and Exhibit, Monterey, California.
- Slotine J.E. and W. Li (1991): **Applied Nonlinear Control**. Prentice Hall, Englewood Cliffs, N.J.
- Steinhauser R., G. Looye, and O. Brieger (2004): **Design and Evaluation of a Dynamic Inversion Control Law for X-31A**. Proc. 6th ONERA-DLR Aerospace Symposium, Berlin, June 22-23, pp. 25-33.
- Thümmel M., G. Looye, M. Kurze, M. Otter, J. Bals (2005): **Nonlinear Inverse Models for Control**. 4th Proceedings of the 4th International Modelica Conference, Hamburg, Germany, ed. G. Schmitz, pp. 267-279. [http://www.modelica.org/events/Conference2005/online\\_proceedings/Session3/Session3c3.pdf](http://www.modelica.org/events/Conference2005/online_proceedings/Session3/Session3c3.pdf)
- Thümmel M., M. Otter, and J. Bals (2001): [Control of Robots with Elastic Joints based on Automatic Generation of Inverse Dynamics Models](#). IEEE/RSJ Conference on Intelligent Robots and Systems, Oct. 29- Nov. 3rd, Hawaii, U.S.A.
- Tietze U., and C. Schenk (2002): **Halbleiter-Schaltungstechnik**. Springer Verlag, 12. Auflage, pp. 815-852.
- Walther N. (2002): **Praxisgerechte Modelica-Bibliothek für Abtastregler. Diplomarbeit**, HTWK Leipzig, Fachbereich Elektro- und Informationstechnik, supervised by Prof. Müller (HTWK) and Prof. Martin Otter (DLR), 12 Nov. 2002.
- Wichmann B.A., and I.D. Hill (1982): Algorithm AS 183: **An efficient and portable pseudo-random number generator**. Applied Statistics 31, pp. 188-190. See also:  
Wichmann B.A. and I.D. Hill (1984): Correction to Algorithm AS 183. Applied Statistics 33, pp. 123.  
McLeod A.I. (1985): A remark on Algorithm AS 183. Applied Statistics 34, pp. 198-200.