

Principles of Object-Oriented Modeling and Simulation with Modelica

Mini-Tutorial for MODPROD Workshop 2009

Peter Fritzson

Linköping University, Dept. of Comp. & Inform. Science
SE 581-83, Linköping, Sweden

petfr@ida.liu.se

The image shows several screenshots from a Modelica simulation environment. The top-left window displays the title 'A Servo Mechanism Model - a micro example of a full system' and an introduction. Below it, a block diagram shows a control loop with a reference input, a feedback loop, and a motor. The bottom-right window shows a block diagram of a servo mechanism with a reference input, a disturbance, an actuator, a system, and an output. The simulation results show the response of the system over time, with plots for position, velocity, and current.

$$\tau_2 = \frac{1}{k_2} \tau_1$$

$$e = \omega_{ref} - \omega_{out}$$

$$u = K \left(e + \frac{1}{T_I} \int_0^t e dt \right)$$

$$v = u \quad u_R = R i \quad u_{emf} = k_1 \omega_{emf}$$

$$J_1 \frac{d^2 \theta_1}{dt^2} = \tau_{emf} + \tau_1$$

$$J_2 \frac{d^2 \theta_2}{dt^2} = \tau_2 + \tau_3$$

$$J_3 \frac{d^2 \theta_3}{dt^2} = -\tau_4 - \tau_{load}$$

$$v = u$$

$$\theta_2 = k_2 \theta_1$$

$$u_L = L \frac{di}{dt}$$

$$u = K \left(e + \frac{1}{T_I} \int_0^t e dt \right)$$

$$e = \omega_{ref} - \omega_{out}$$

$$v - u_R - u_L - u_{emf} = 0$$

$$u_{emf} = k_1 \omega_{emf} \quad i = \frac{1}{k_1} \tau_{emf} \quad \tau_2 = \frac{1}{k_2} \tau_1$$

$$\frac{J_1 - J_2 k_2^2}{k_2} \frac{d^2 \theta_2}{dt^2} = \tau_{emf} - k_2 \tau_3$$

2009-02-03 MODPROD'2009

Tutorial Aim

This tutorial aims to give the participant an

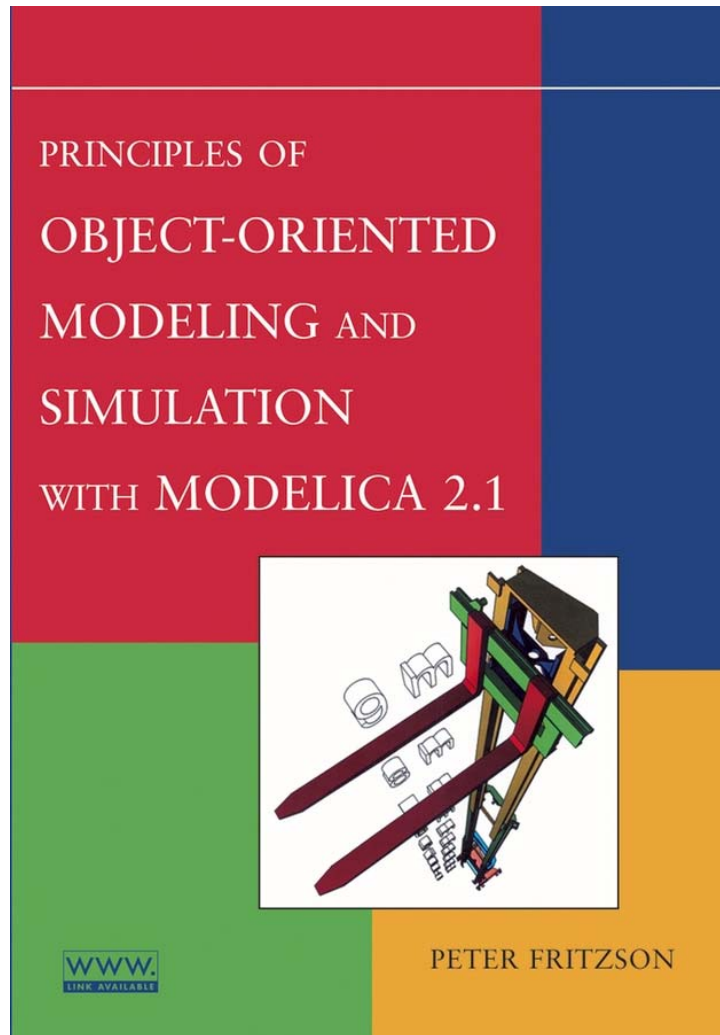
- introduction to object-oriented modeling and simulation
- introduction to the Modelica language
- hands-on experience in textual modeling
- (Probably not enough time: hands-on experience in graphical modeling)

After the course participants should be able to develop basic models on their own.

Software Installation

- Start the software installation
- Install OpenModelica-1.4.5.msi from the OpenModelica USB Stick (Windows)
- If you have Mac, install OpenModelica-1.4.5.dmg

Tutorial Based on Recent Book, 2004



Peter Fritzson Principles of Object Oriented Modeling and Simulation with Modelica 2.1

Wiley-IEEE Press

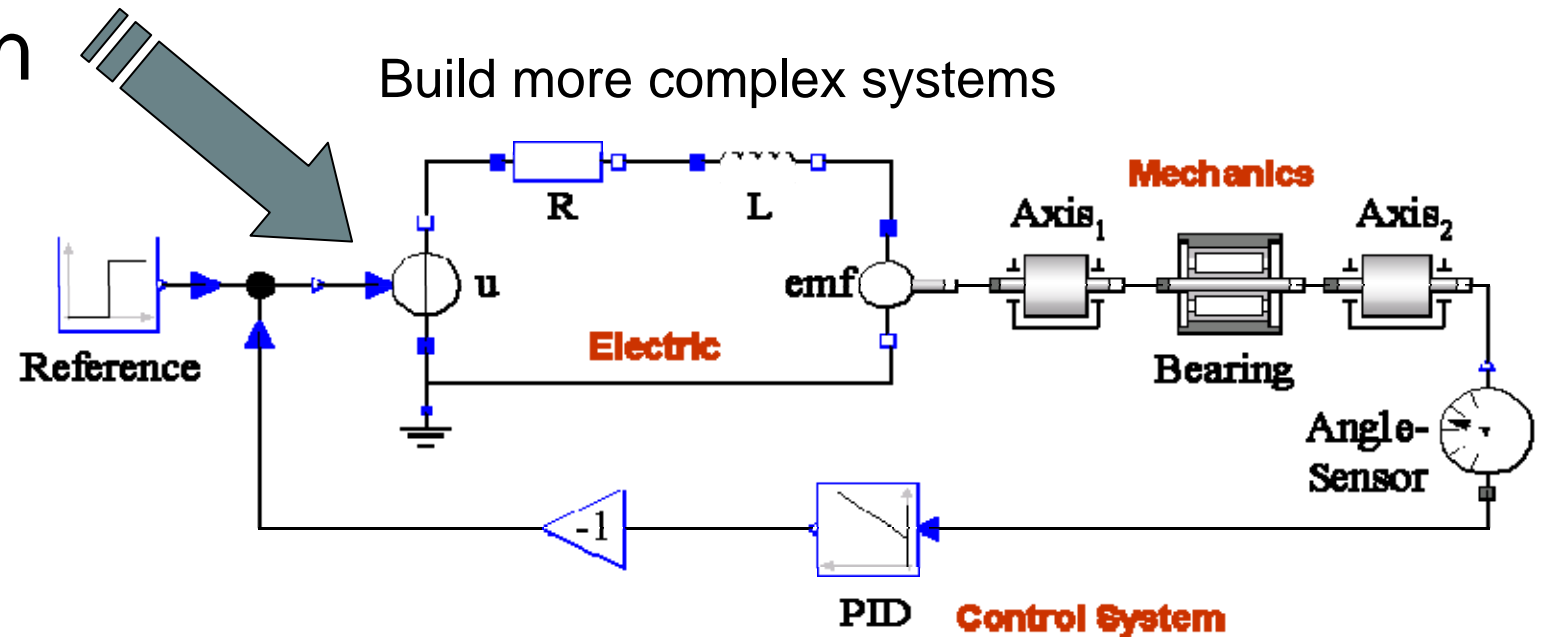
940 pages

Acknowledgements, Usage, Copyrights

- If you want to use the Powerpoint version of these slides in your own course, send an email to: peter.fritzson@ida.liu.se
- Thanks to Emma Larsson Nilsson for contributions to the layout of these slides
- Most examples and figures in this tutorial are adapted with permission from Peter Fritzson's book "Principles of Object Oriented Modeling and Simulation with Modelica 2.1", copyright Wiley-IEEE Press
- Some examples and figures reproduced with permission from Modelica Association, Martin Otter, Hilding Elmqvist, and MathCore
- Modelica Association: www.modelica.org
- OpenModelica: www.ida.liu.se/projects/OpenModelica

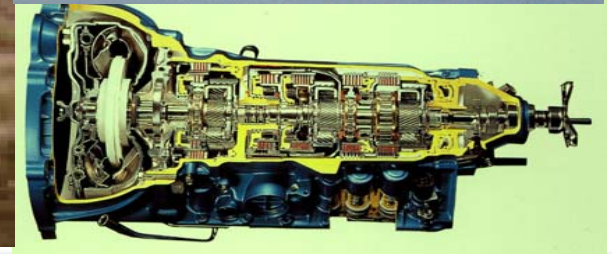
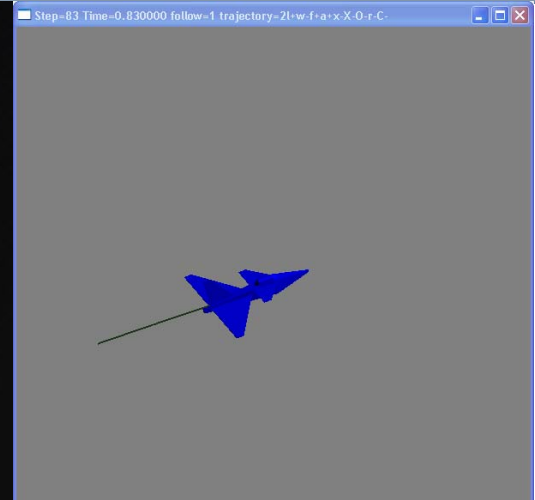
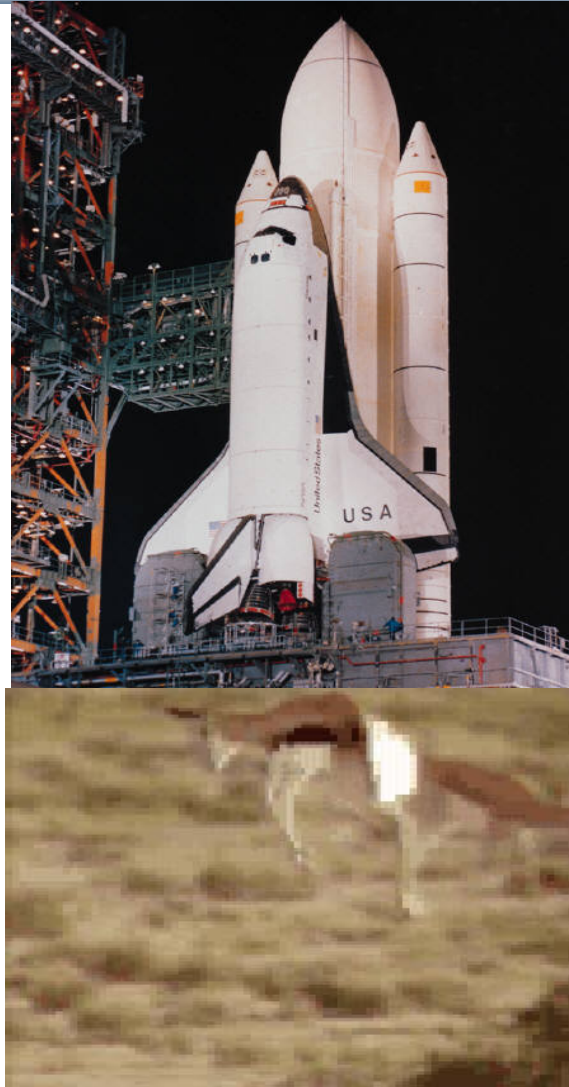
Why Modeling & Simulation?

- Increase understanding of complex systems
- Design and optimization
- Virtual prototyping
- Verification



Examples of Complex Systems

- Robotics
- Automotive
- Aircrafts
- Satellites
- Biomechanics
- Power plants
- Hardware-in-the-loop, real-time simulation



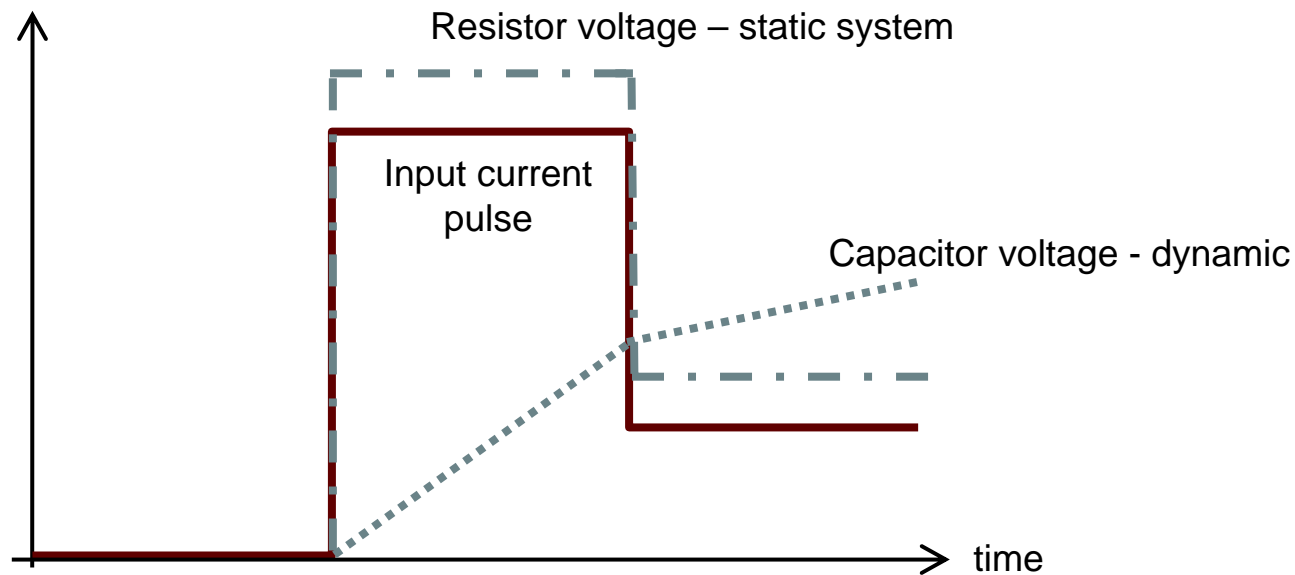
Kinds of Mathematical Models

- Dynamic vs. Static models
- Continuous-time vs. Discrete-time dynamic models
- Quantitative vs. Qualitative models

Dynamic vs. Static Models

A **dynamic** model includes *time* in the model

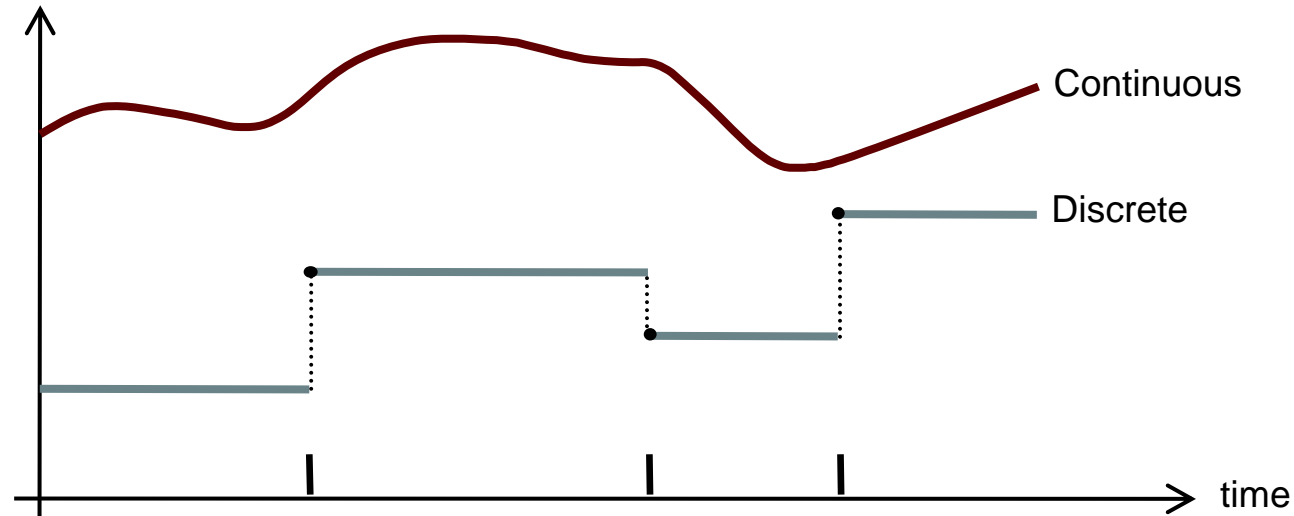
A **static** model can be defined *without* involving *time*



Continuous vs. Discrete-Time Dynamic Models

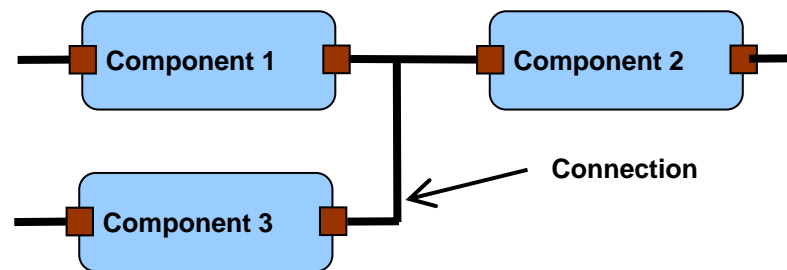
Continuous-time models may evolve their variable values *continuously* during a time period

Discrete-time variables change values a *finite* number of times during a time period



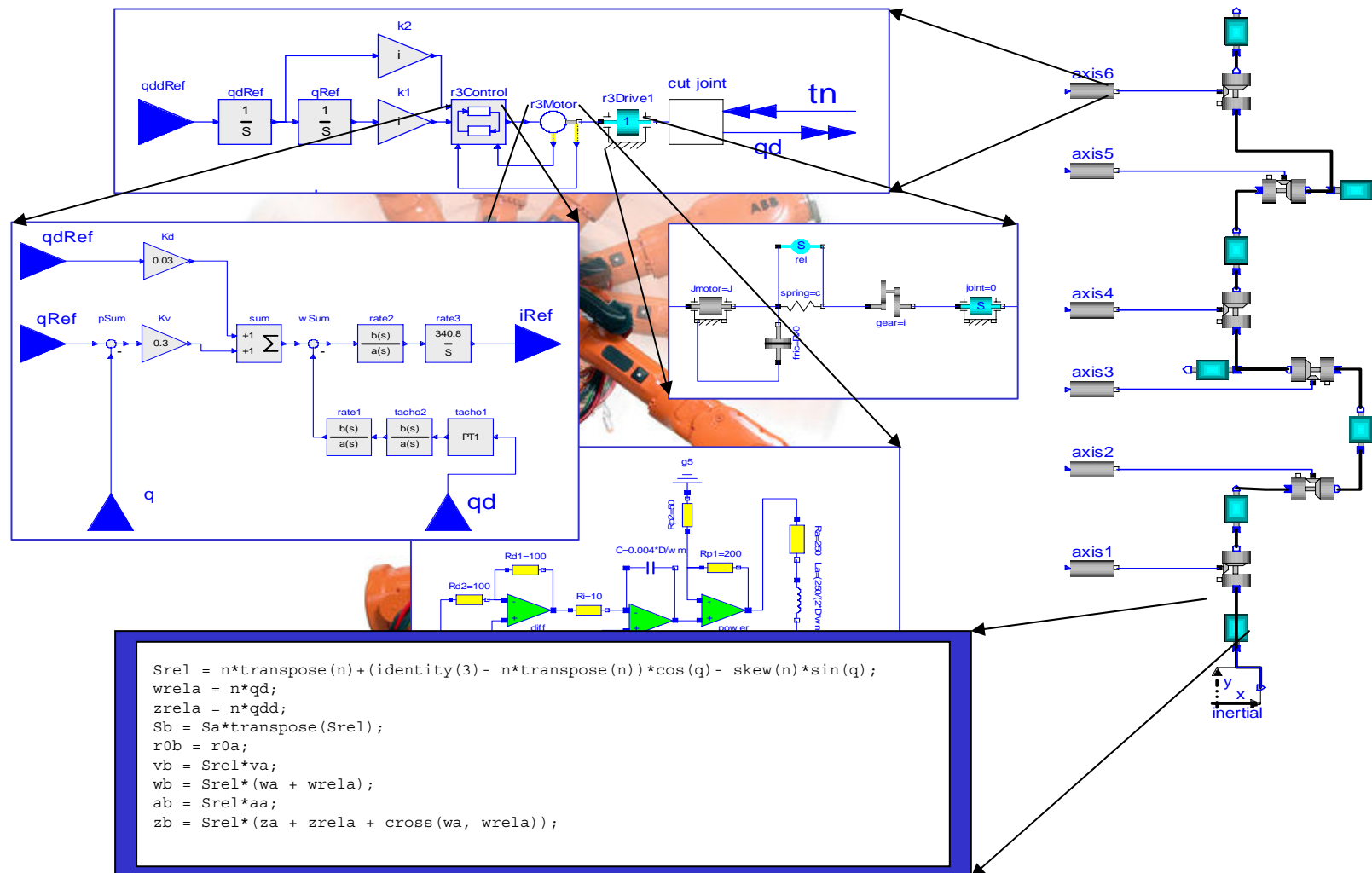
Principles of Graphical Equation-Based Modeling

- Each icon represents a physical component i.e. Resistor, mechanical Gear Box, Pump
- Composition lines represent the actual physical connections i.e. electrical line, mechanical connection, heat flow



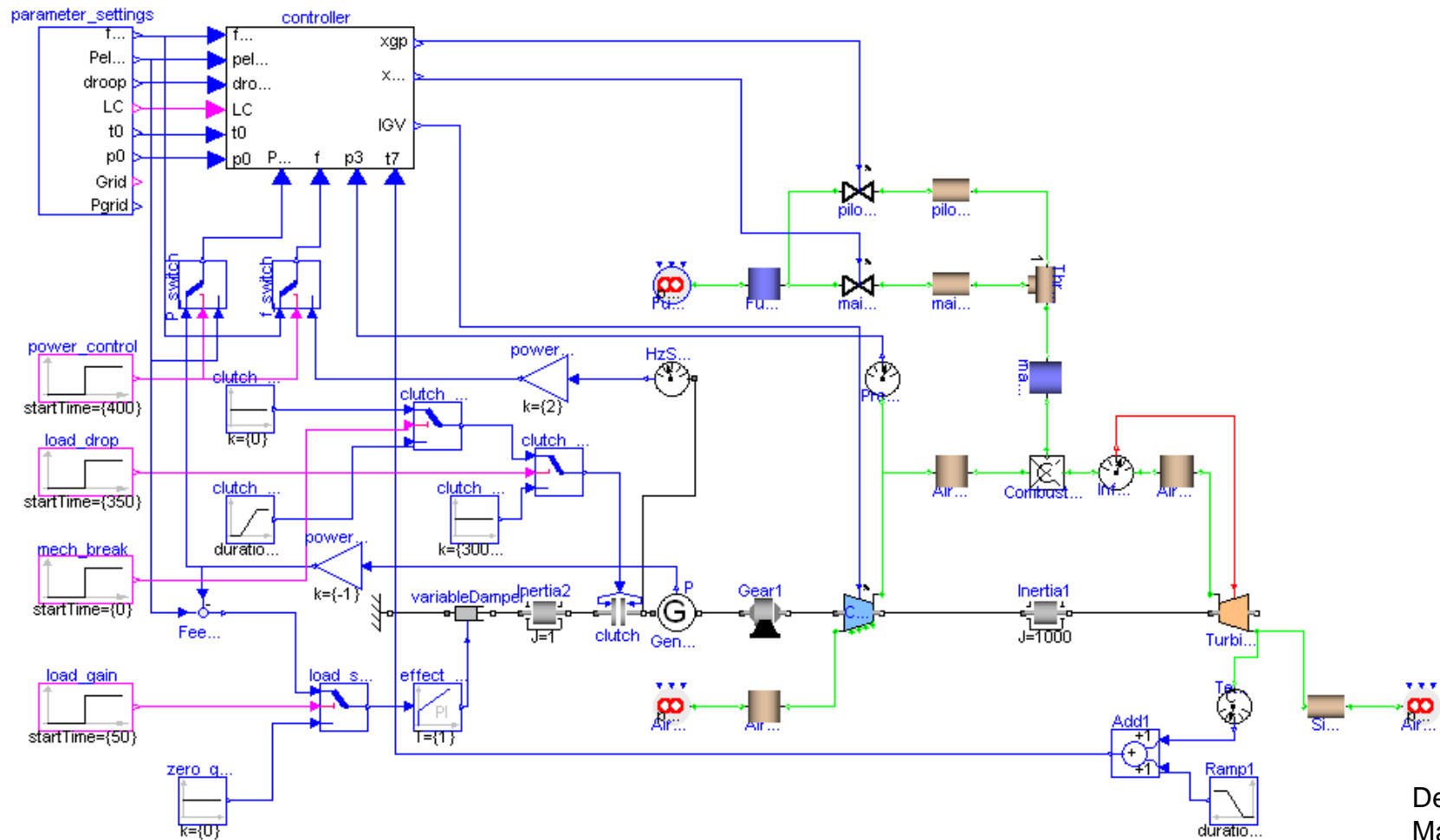
- Variables at the interfaces describe interaction with other component
- Physical behavior of a component is described by equations
- Hierarchical decomposition of components

Application Example – Industry Robot



Courtesy of Martin Otter

GTX Gas Turbine Power Cutoff Mechanism



Developed by
MathCore for
Siemens

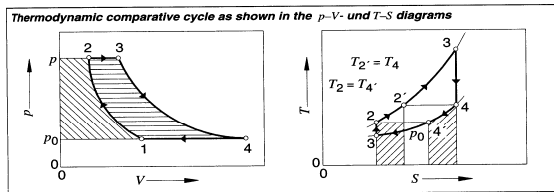
Courtesy of Siemens Industrial Turbomachinery AB

Modelica – The Next Generation Modeling Language

Stored Knowledge

Model knowledge is stored in books and human minds which computers cannot access

Internal-combustion engines 417



from T_2 to T_2' , supplied by the heat exchanger is coupled with a thermal discharge ($4 \rightarrow 4'$). If heat is completely exchanged, the quantity of heat to be added per unit of gas is reduced to

$$q_{in} = c_p \cdot (T_3 - T_2) = c_p \cdot (T_3 - T_4)$$

and the quantity of heat to be removed is

$$q_{out} = c_p \cdot (T_4' - T_1) = c_p \cdot (T_2 - T_1).$$

The maximum thermal efficiency for the gas turbine with heat exchanger is:

$$\eta_{th} = 1 - Q_{out}/Q_{in} = 1 - (T_2 - T_1)/(T_3 - T_4)$$

Where $p_2/p_1 = (T_2/T_1)^{\frac{\gamma}{\gamma-1}} = (T_3/T_4)^{\frac{\gamma}{\gamma-1}}$ and $T_4 = T_3 \cdot (T_1/T_2)$ thus

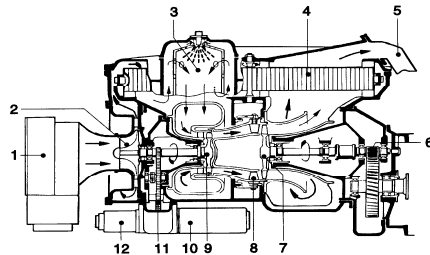
$$\eta_{th} = 1 - (T_2/T_3)$$

Current gas-turbine powerplants achieve thermal efficiencies of up to 35%.

Advantages of the gas turbine: clean exhaust without supplementary emissions-control devices; extremely smooth running; multifuel capability; good static torque curve; extended maintenance intervals.

Disadvantages: manufacturing costs still high; poor transitional response; higher fuel consumption; less suitable for low-power applications.

Gas turbine 1 Filter and silencer, 2 Radial-flow compressor, 3 Burner, 4 Heat exchanger, 5 Exhaust port, 6 Reduction gearset, 7 Power turbine, 8 Adjustable guide vanes, 9 Compressor turbine, 10 Starter, 11 Auxiliary equipment drive, 12 Lubricating oil pump.



“The change of motion is proportional to the motive force impressed”

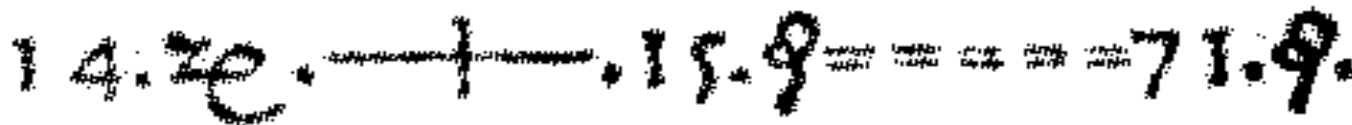
– Newton

Lex. II.

Mutationem motus proportionalem esse vi motrici impressae, & fieri secundum lineam rectam qua vis illa imprimitur.

The Form – Equations

- Equations were used in the third millennium B.C.
- Equality sign was introduced by Robert Recorde in 1557



Newton still wrote text (Principia, vol. 1, 1686)

“The change of motion is proportional to the motive force impressed”

CSSL (1967) introduced a special form of “equation”:

variable = expression

$v = \text{INTEG}(F) / m$

Programming languages usually do not allow equations!

Modelica – The Next Generation Modeling Language

Declarative language

Equations and mathematical functions allow acausal modeling, high level specification, increased correctness

Multi-domain modeling

Combine electrical, mechanical, thermodynamic, hydraulic, biological, control, event, real-time, etc...

Everything is a class

Strongly typed object-oriented language with a general class concept, Java & MATLAB-like syntax

Visual component programming

Hierarchical system architecture capabilities

Efficient, non-proprietary

Efficiency comparable to C; advanced equation compilation, e.g. 300 000 equations, ~150 000 lines on standard PC

Object Oriented Mathematical Modeling

- The static *declarative structure* of a mathematical model is emphasized
- OO is primarily used as a *structuring concept*
- OO *is not* viewed as dynamic object creation and sending messages
- *Dynamic model* properties are expressed in a *declarative way* through equations.
- Acausal classes supports *better reuse of modeling and design knowledge* than traditional classes

Modelica Acausal Modeling

What is *acausal* modeling/design?

Why does it increase *reuse*?

The acausality makes Modelica library classes *more reusable* than traditional classes containing assignment statements where the input-output causality is fixed.

Example: a resistor *equation*:

$$\mathbf{R} * \mathbf{i} = \mathbf{v};$$

can be used in three ways:

$$\mathbf{i} := \mathbf{v} / \mathbf{R};$$

$$\mathbf{v} := \mathbf{R} * \mathbf{i};$$

$$\mathbf{R} := \mathbf{v} / \mathbf{i};$$

Brief Modelica History

- First Modelica design group meeting in fall 1996
 - International group of people with expert knowledge in both language design and physical modeling
 - Industry and academia
- Modelica Versions
 - 1.0 released September 1997
 - 2.0 released March 2002
 - 2.2 released March 2005
 - 3.0 released September 2007
- Modelica Association established 2000
 - Open, non-profit organization

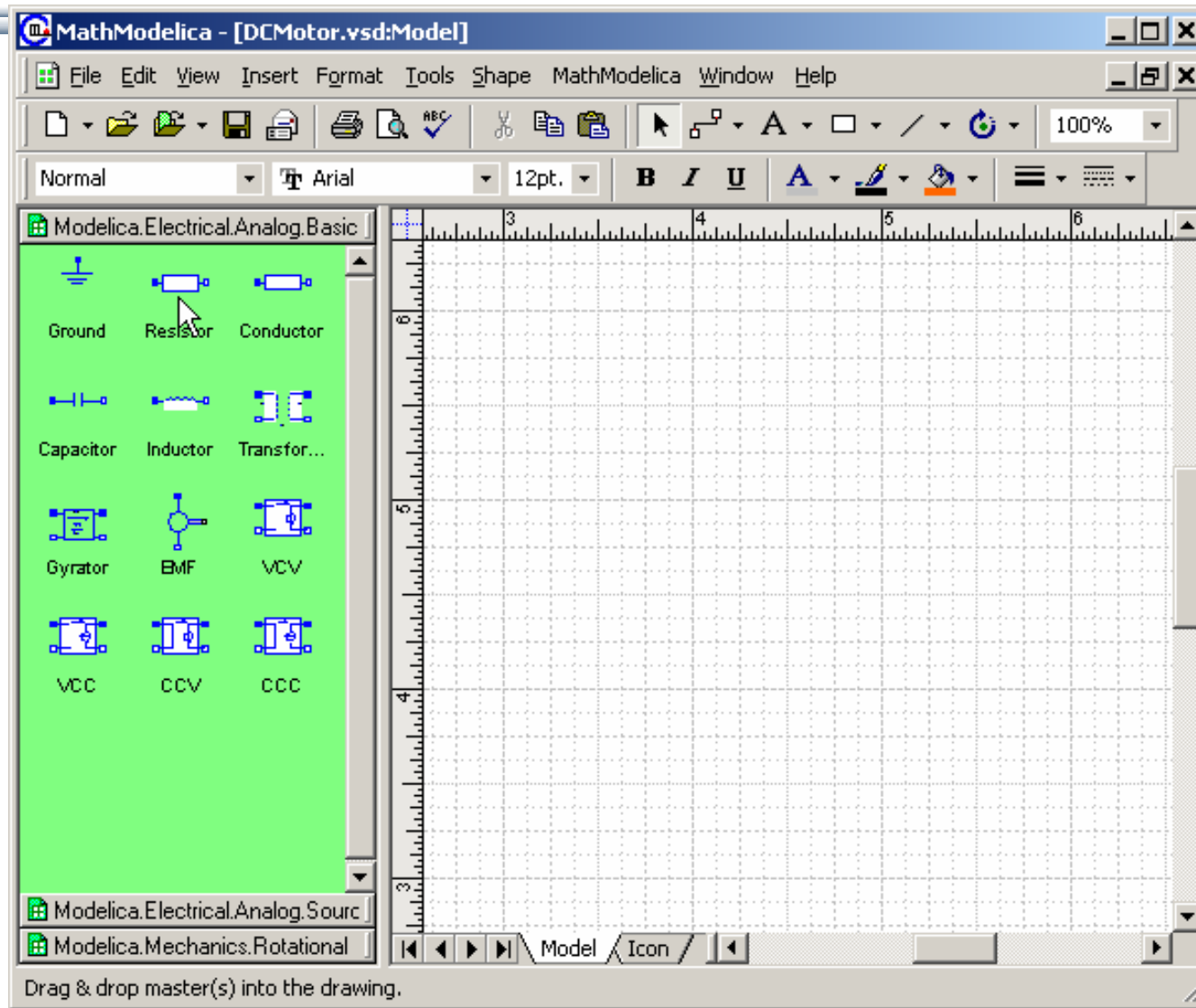
Modelica Conferences

- The 1st International Modelica conference October, 2000
- The 2nd International Modelica conference March 18-19, 2002
- The 3rd International Modelica conference November 5-6, 2003 in Linköping, Sweden
- The 4th International Modelica conference March 6-7, 2005 in Hamburg, Germany
- The 5th International Modelica conference September 4-5, 2006 in Vienna, Austria
- The 6th International Modelica conference March 3-4, 2008 in Bielefeld, Germany

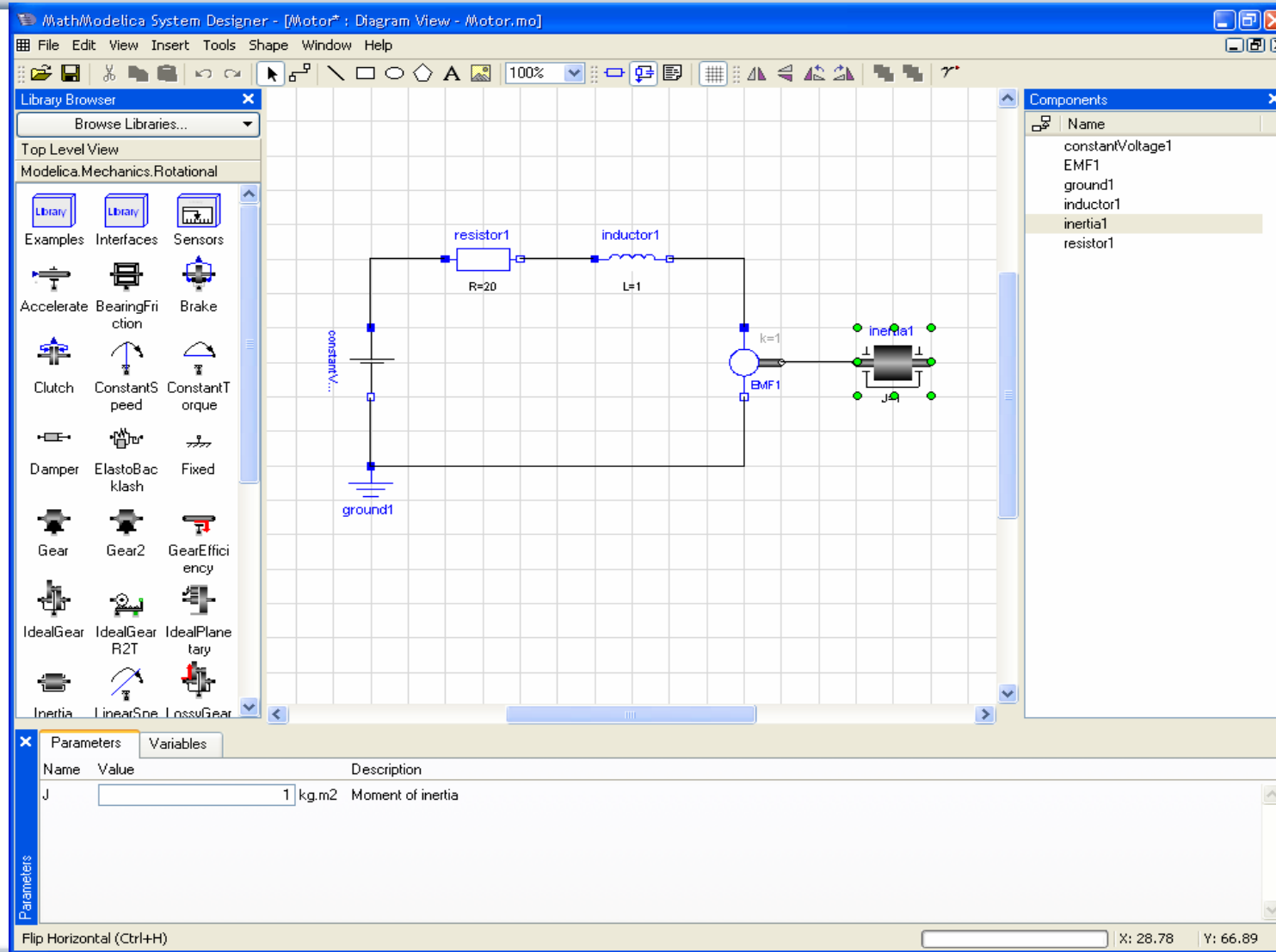
Demo – Basic Graphical Modeling

- Demo (Can use the SimForge Graphical Editor)

Graphical Modeling - Using Drag and Drop Composition



Completed DCMotor using Graphical Composition



Courtesy
MathCore
Engineering AB

Multi-Domain (Electro-Mechanical) Modelica Model

- A DC motor can be thought of as an electrical circuit which also contains an electromechanical component

model DCMotor

```
Resistor R(R=100);
```

```
Inductor L(L=100);
```

```
VsourceDC DC(f=10);
```

```
Ground G;
```

```
ElectroMechanicalElement EM(k=10, J=10, b=2);
```

```
Inertia load;
```

equation

```
connect (DC.p, R.n);
```

```
connect (R.p, L.n);
```

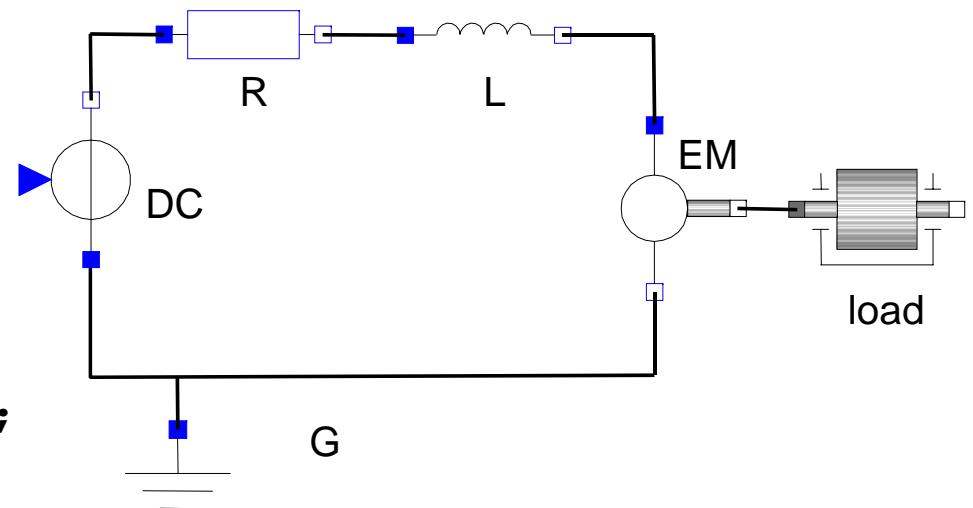
```
connect (L.p, EM.n);
```

```
connect (EM.p, DC.n);
```

```
connect (DC.n, G.p);
```

```
connect (EM.flange, load.flange);
```

end DCMotor



Corresponding DCMotor Model Equations

The following equations are automatically derived from the Modelica model:

0 == DC.p.i + R.n.i	EM.u == EM.p.v - EM.n.v	R.u == R.p.v - R.n.v
DC.p.v == R.n.v	0 == EM.p.i + EM.n.i	0 == R.p.i + R.n.i
	EM.i == EM.p.i	R.i == R.p.i
0 == R.p.i + L.n.i	EM.u == EM.k * EM.omega	R.u == R.R * R.i
R.p.v == L.n.v	EM.i == EM.M / EM.k	
	EM.J * EM.omega == EM.M - EM.b * EM.omega	L.u == L.p.v - L.n.v
0 == L.p.i + EM.n.i		0 == L.p.i + L.n.i
L.p.v == EM.n.v	DC.u == DC.p.v - DC.n.v	L.i == L.p.i
	0 == DC.p.i + DC.n.i	L.u == L.L * L.i'
0 == EM.p.i + DC.n.i	DC.i == DC.p.i	
EM.p.v == DC.n.v	DC.u == DC.Amp * Sin[2 * pi * DC.f * t]	
0 == DC.n.i + G.p.i		
DC.n.v == G.p.v		

(load component not included)

Automatic transformation to ODE or DAE for simulation:

$$\frac{dx}{dt} == f[x, u, t] \quad g\left[\frac{dx}{dt}, x, u, t\right] == 0$$

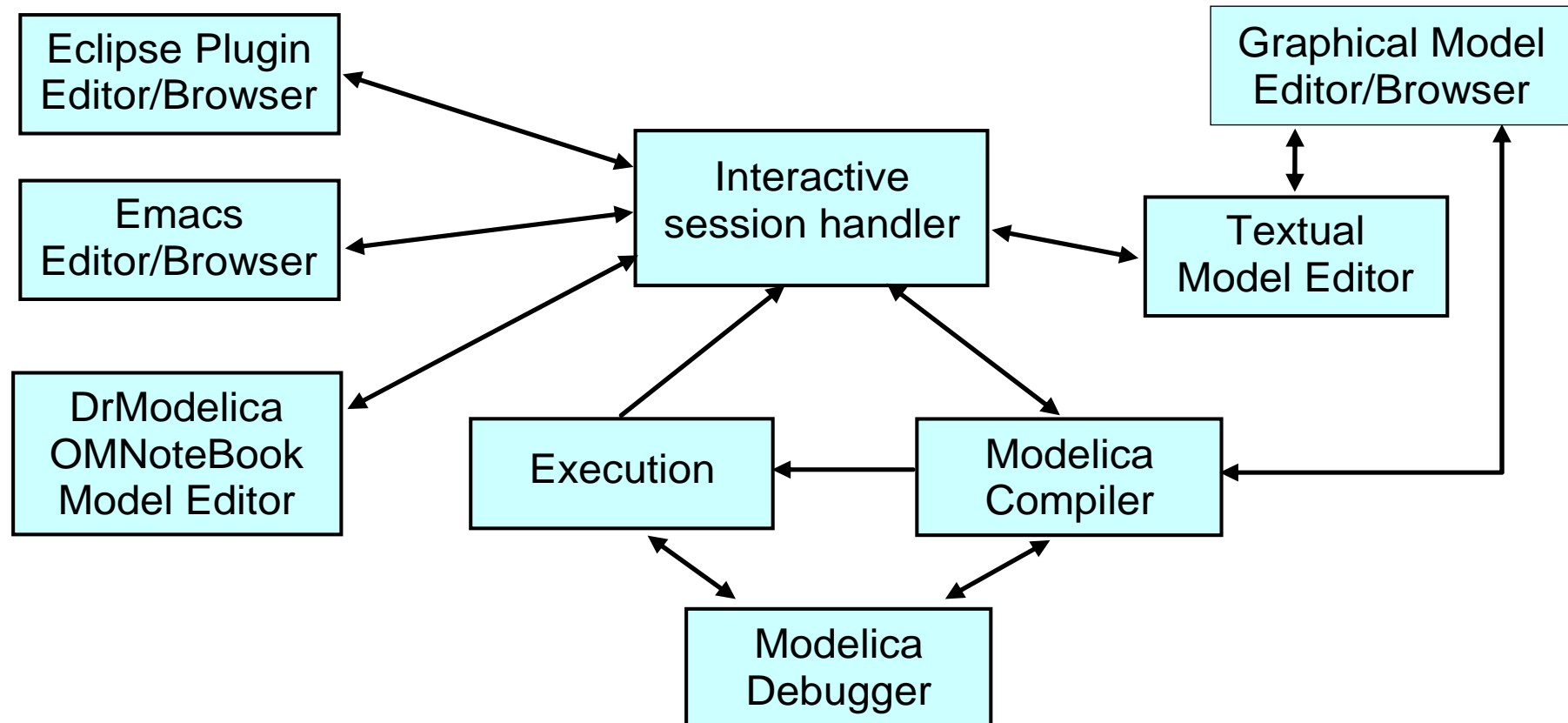
The OpenModelica Environment

www.OpenModelica.org

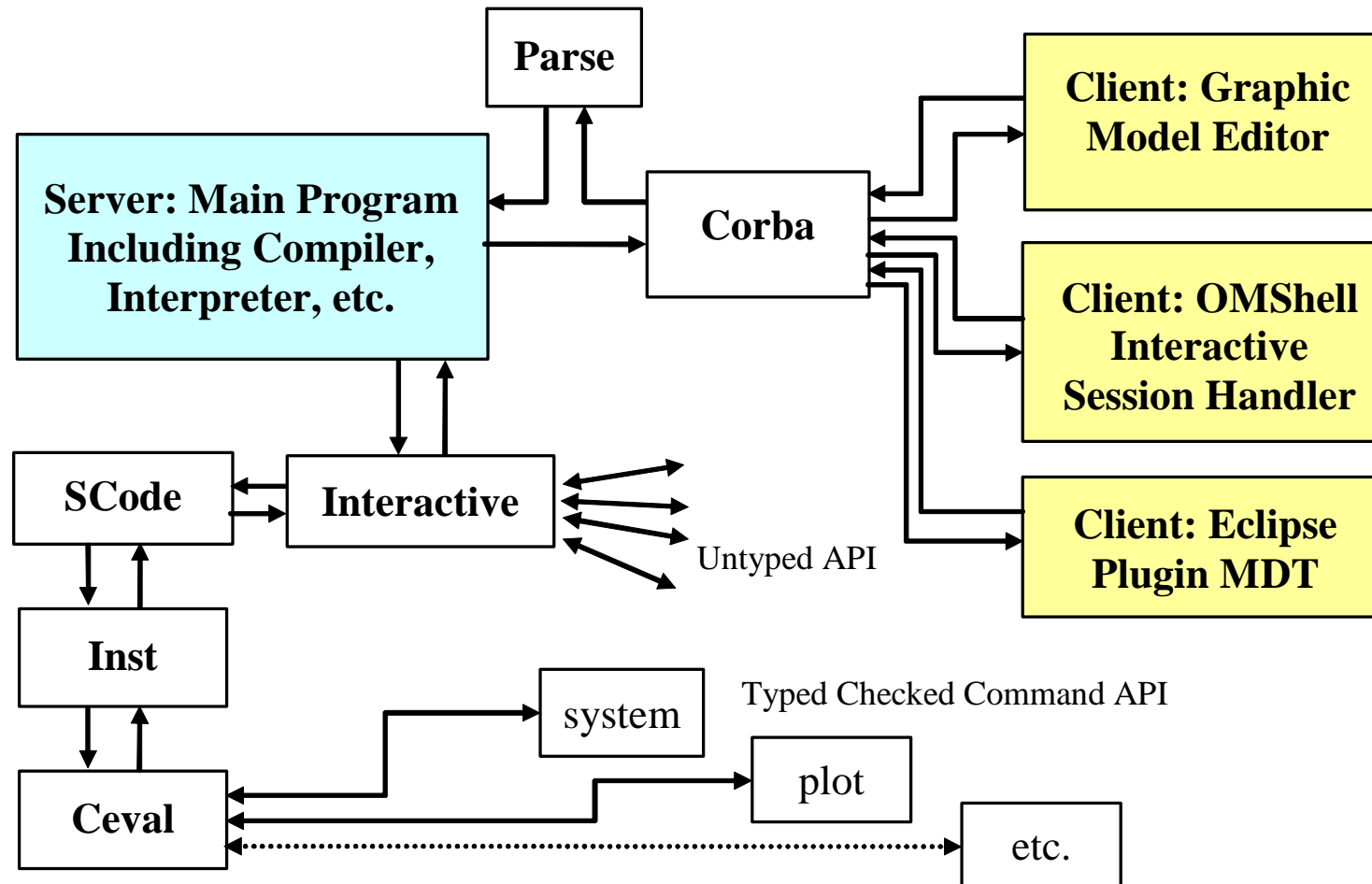
OpenModelica

- **Goal:** comprehensive modeling and simulation environment for research, teaching, and industrial usage
- Free, open-source for academic use
- Commercial users have to be members of Open Source Modelica Consortium (OSMC)
- Available under OSMC-PL license
- The OpenModelica compiler (OMC) implemented in MetaModelica, a slightly extended Modelica
- Invitation for open-source cooperation around OpenModelica, tools, and applications

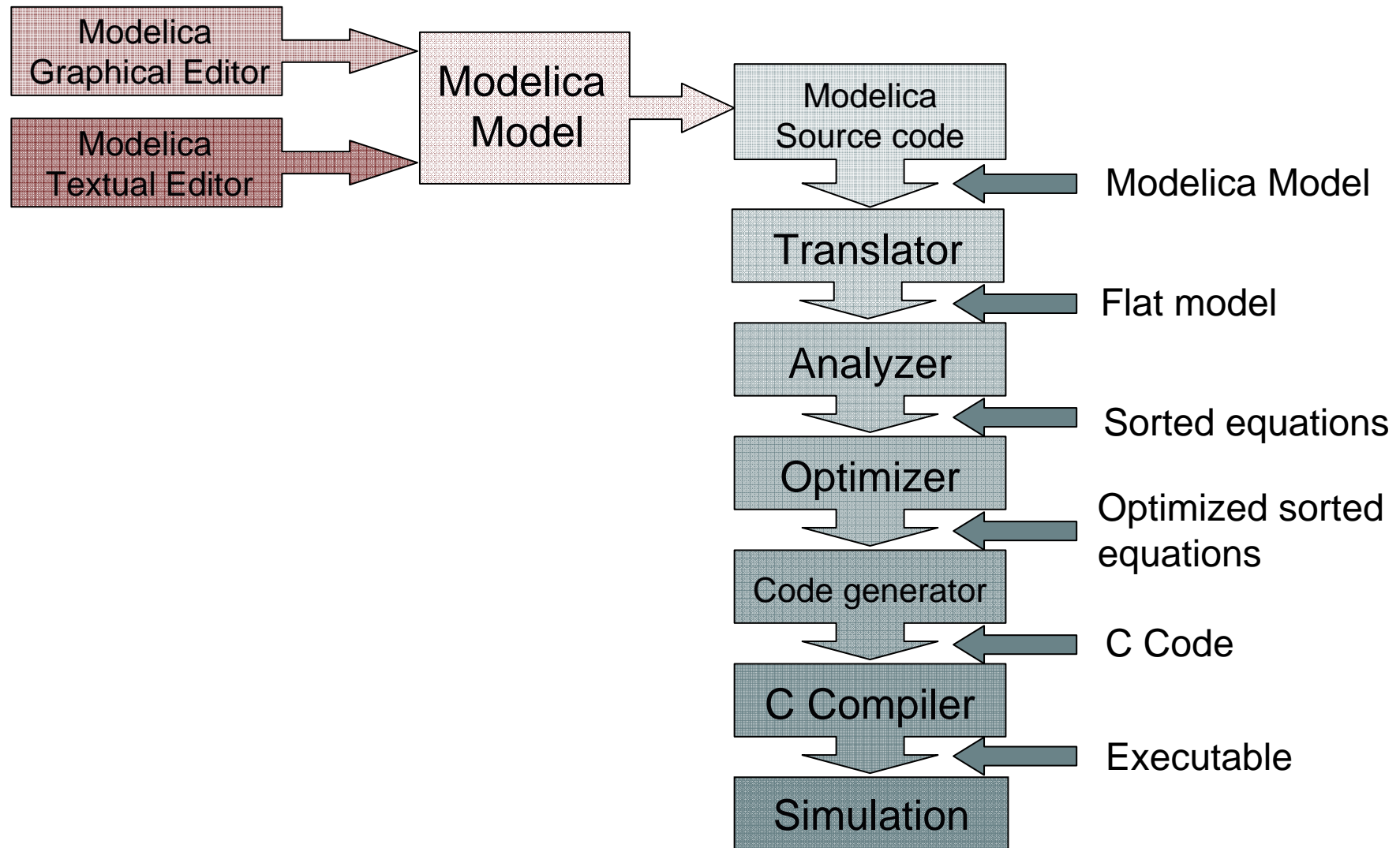
OpenModelica Environment Architecture



OpenModelica Client-Server Architecture



Translation of Models to Simulation Code



Released in OpenModelica 1.4.5

- Compiler/interpreter – OMC
- Interactive session handler – OMShell
- Notebook with DrModelica – OMNotebook
- Eclipse plugin Modelica Development Tooling (MDT)

- Also available: SimForge graphical editor

Platforms

- All OpenModelica GUI tools (OMShell, MDT, OMNotebook) are developed on the Qt4 GUI library, portable between Windows, Linux, Mac
- Both compilers (OMC, MMC) are portable between the three platforms

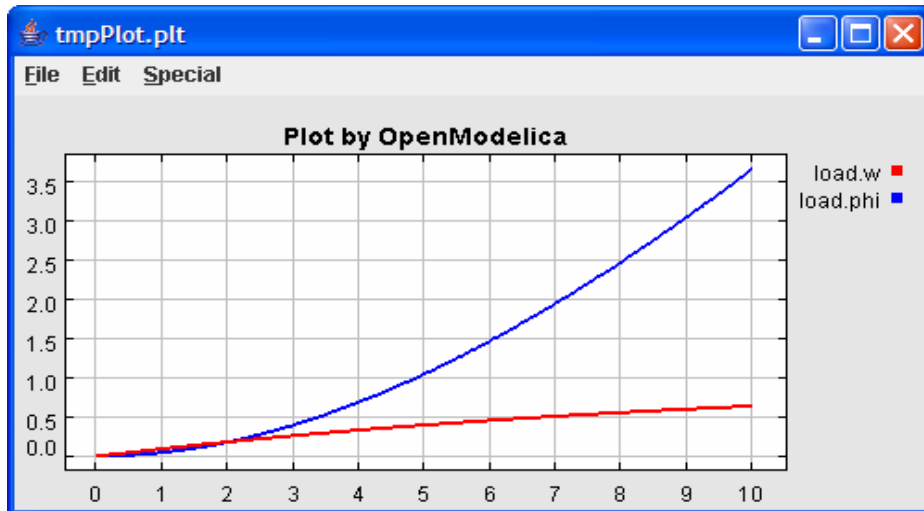
- Windows – main development and release platform
- Linux – available
- Mac –available

Interactive Session Handler – dcmotor Example

- OMShell – OpenModelica Shell

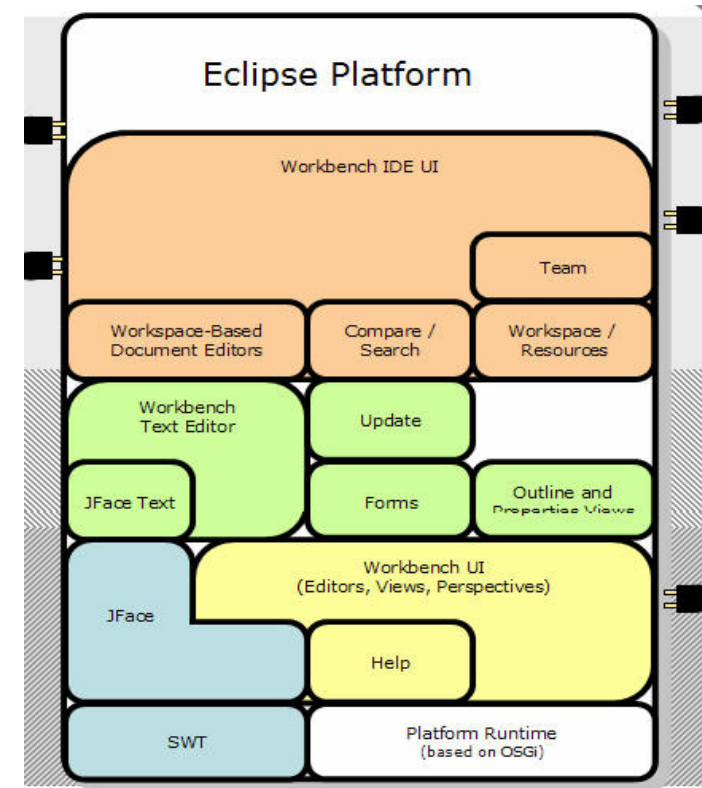
```
>>simulate(dcmotor,startTime=0.0,stopTime=10.0)  
>>plot({load.w,load.phi})
```

```
model dcmotor  
  Modelica.Electrical.Analog.Basic.Resistor r1(R=10);  
  Modelica.Electrical.Analog.Basic.Inductor il;  
  Modelica.Electrical.Analog.Basic.EMF emf1;  
  Modelica.Mechanics.Rotational.Inertia load;  
  Modelica.Electrical.Analog.Basic.Ground g;  
  Modelica.Electrical.Analog.Sources.ConstantVoltage v;  
equation  
  connect(v.p,r1.p);  
  connect(v.n,g.p);  
  connect(r1.n,il.p);  
  connect(il.n,emf1.p);  
  connect(emf1.n,g.p);  
  connect(emf1.flange_b,load.flange_a);  
end dcmotor;
```

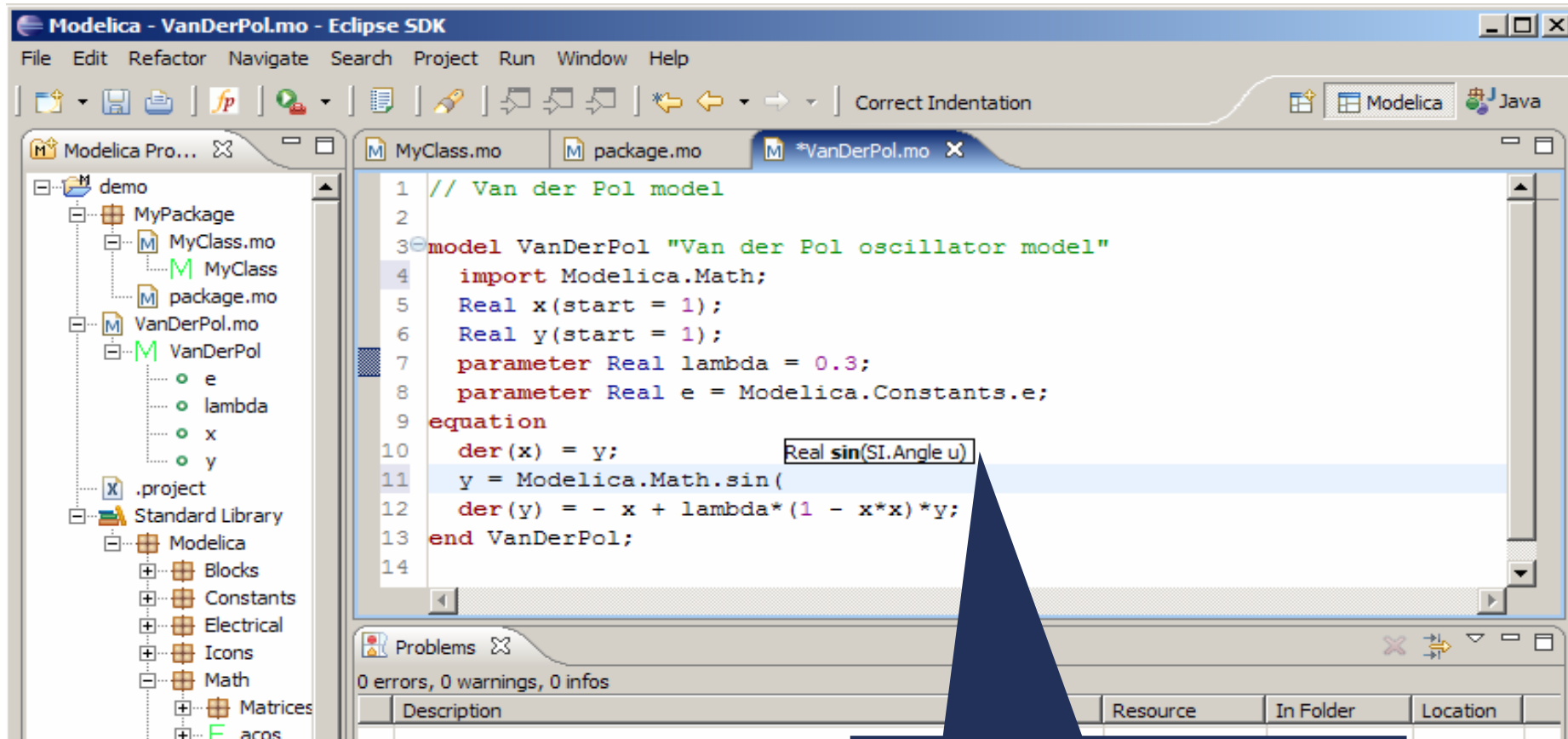


OpenModelica MDT – Eclipse Plugin

- Browsing of packages, classes, functions
- Automatic building of executables;
separate compilation
- Syntax highlighting
- Code completion,
Code query support for developers
- Automatic Indentation
- Debugger
(Prel. version for algorithmic subset)



OpenModelica MDT – Usage Example



Code Assistance on function calling.

The Modelica Language – Modelica Classes and Inheritance

Variables and Constants

Built-in primitive data types

Boolean	true or false
Integer	Integer value, e.g. 42 or -3
Real	Floating point value, e.g. 2.4e-6
String	String, e.g. “Hello world”
Enumeration	Enumeration literal e.g. ShirtSize.Medium

Variables and Constants cont'

- Names indicate meaning of constant
- Easier to maintain code
- Parameters are constant during simulation
- Two types of constants in Modelica
 - `constant`
 - `parameter`

```
constant Real    PI=3.141592653589793;  
constant String  redcolor = "red";  
constant Integer one = 1;  
parameter Real   mass = 22.5;
```

Comments in Modelica

1) Declaration comments, e.g. `Real x "state variable";`

```
class VanDerPol "Van der Pol oscillator model"
  Real x(start = 1) "Descriptive string for x"; // x starts at 1
  Real y(start = 1) "y coordinate";           // y starts at 1
  parameter Real lambda = 0.3;
equation
  der(x) = y; // This is the 1st diff equation //
  der(y) = -x + lambda*(1 - x*x)*y; /* This is the 2nd diff equation */
end VanDerPol;
```

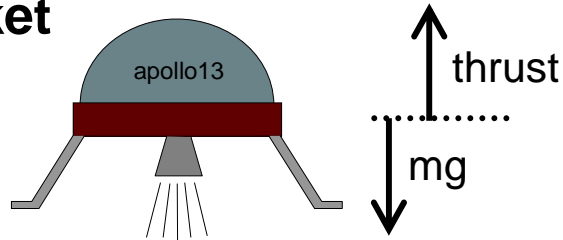
2) Source code comments, disregarded by compiler

2a) C style, e.g. `/* This is a C style comment */`

2b) C++ style, e.g. `// Comment to the end of the line..`

A Simple Rocket Model

Rocket



$$acceleration = \frac{thrust - mass \cdot gravity}{mass}$$

$$mass' = -massLossRate \cdot abs(thrust)$$

$$altitude' = velocity$$

$$velocity' = acceleration$$

new model
parameters (changeable
before the simulation)

floating point
type

differentiation with
regards to time

```
class Rocket "rocket class"
  parameter String name;
  Real mass(start=1038.358);
  Real altitude(start= 59404);
  Real velocity(start= -2003);
  Real acceleration;
  Real thrust; // Thrust force on rocket
  Real gravity; // Gravity forcefield
  parameter Real massLossRate=0.000277;
equation
  (thrust-mass*gravity)/mass = acceleration;
  der(mass) = -massLossRate * abs(thrust);
  der(altitude) = velocity;
  der(velocity) = acceleration;
end Rocket;
```

declaration
comment

start value

name + default value

mathematical
equation (acausal)

Celestial Body Class

A class declaration creates a *type name* in Modelica

```
class CelestialBody
  constant Real    g = 6.672e-11;
  parameter Real   radius;
  parameter String name;
  parameter Real   mass;
end CelestialBody;
```

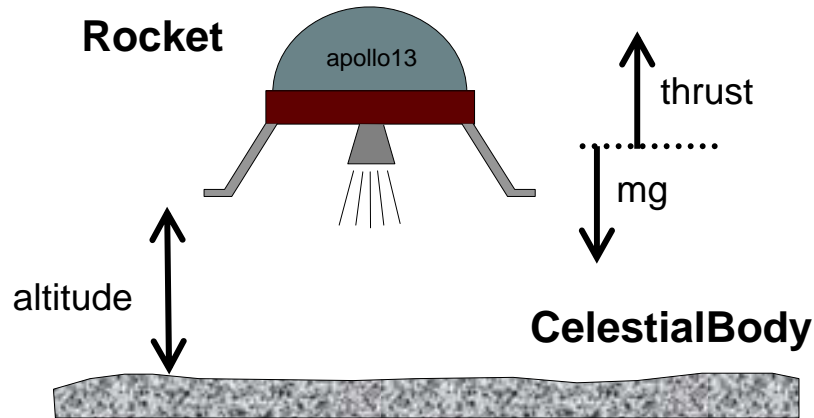


An *instance* of the class can be declared by *prefixing* the type name a variable name

```
...
CelestialBody moon;
...
```

The declaration states that `moon` is a variable containing an object of type `CelestialBody`

Moon Landing



$$apollo .gravity = \frac{moon .g \cdot moon .mass}{(apollo .altitude + moon .radius)^2}$$

only access inside the class

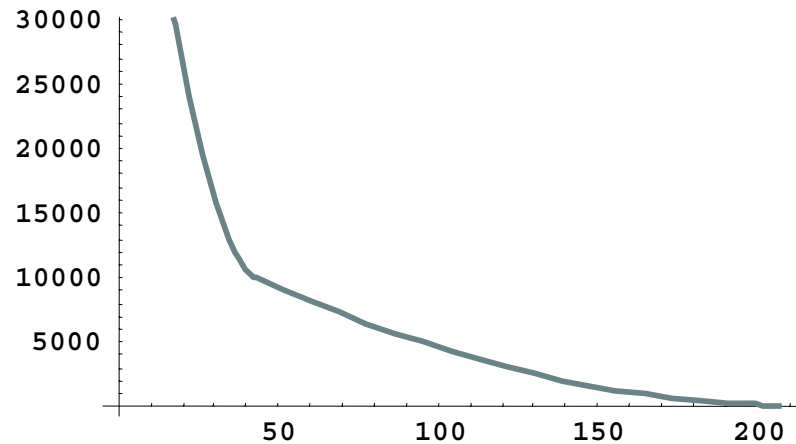
access by dot notation outside the class

```

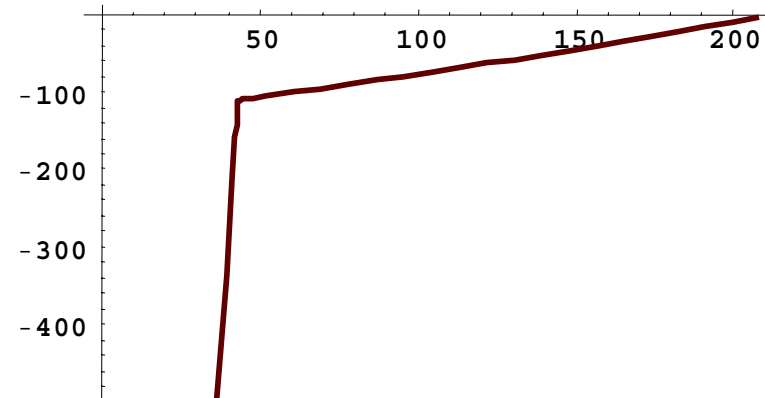
class MoonLanding
  parameter Real force1 = 36350;
  parameter Real force2 = 1308;
  protected
  parameter Real thrustEndTime = 210;
  parameter Real thrustDecreaseTime = 43.2;
  public
  Rocket      apollo(name="apollo13");
  CelestialBody moon(name="moon",mass=7.382e22,radius=1.738e6);
  equation
  apollo.thrust = if (time < thrustDecreaseTime) then force1
                  else if (time < thrustEndTime) then force2
                  else 0;
  apollo.gravity=moon.g*moon.mass/(apollo.altitude+moon.radius)^2;
end MoonLanding;
  
```

Simulation of Moon Landing

```
simulate(MoonLanding, stopTime=230)  
plot(apollo.altitude, xrange={0,208})  
plot(apollo.velocity, xrange={0,208})
```



It starts at an altitude of 59404 (not shown in the diagram) at time zero, gradually reducing it until touchdown at the lunar surface when the altitude is zero



The rocket initially has a high negative velocity when approaching the lunar surface. This is reduced to zero at touchdown, giving a smooth landing

Restricted Class Keywords

- The `class` keyword can be replaced by other keywords, e.g.: `model`, `record`, `block`, `connector`, `function`, ...
- Classes declared with such keywords have restrictions
- Restrictions apply to the contents of restricted classes
- Example: A `model` is a class that cannot be used as a connector class
- Example: A `record` is a class that only contains data, with no equations
- Example: A `block` is a class with fixed input-output causality

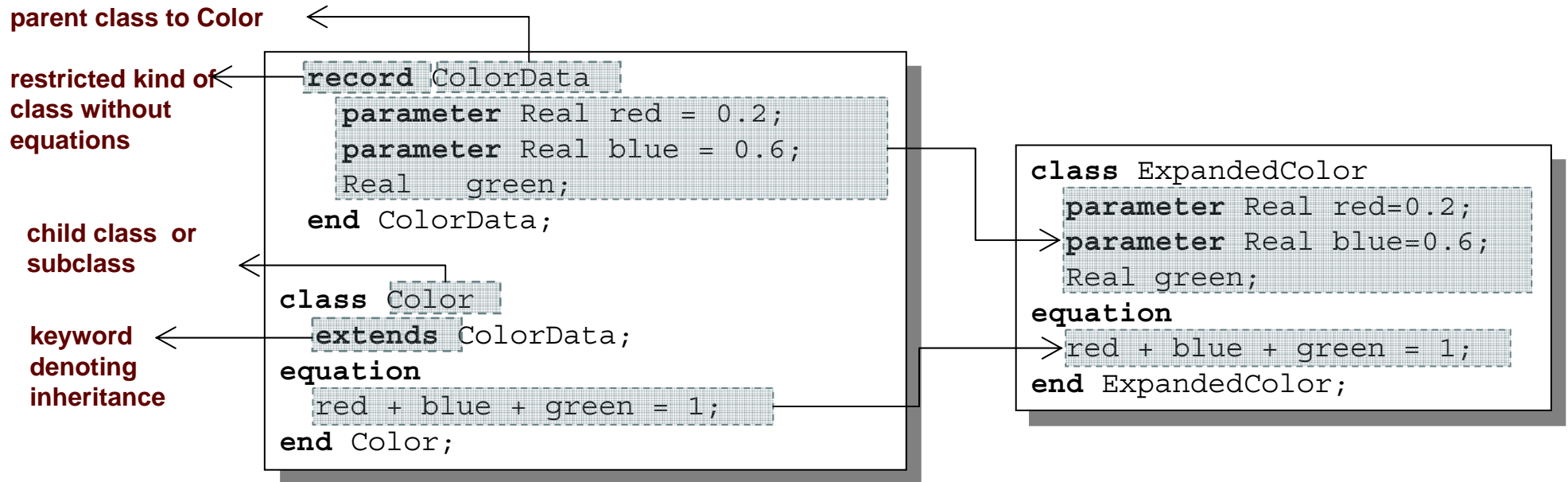
```
model CelestialBody
  constant Real    g = 6.672e-11;
  parameter Real   radius;
  parameter String name;
  parameter Real   mass;
end CelestialBody;
```

Modelica Functions

- Modelica Functions can be viewed as a special kind of restricted class with some extensions
- A function can be called with arguments, and is instantiated dynamically when called
- More on functions and algorithms later

```
function sum
  input  Real arg1;
  input  Real arg2;
  output Real result;
algorithm
  result := arg1+arg2;
end sum;
```

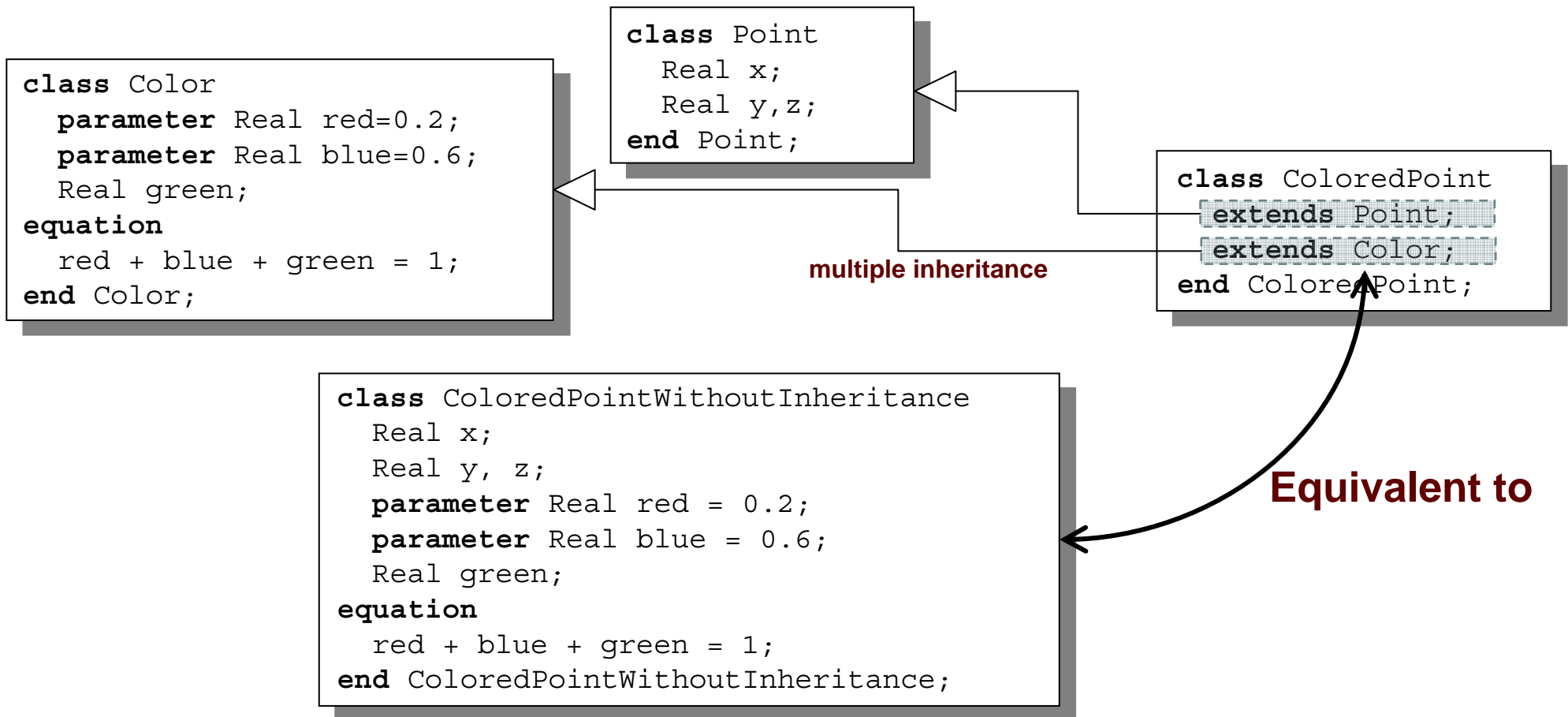
Inheritance



Data and behavior: field declarations, equations, and certain other contents are copied into the subclass

Multiple Inheritance

Multiple Inheritance is fine – inheriting both geometry and color



Simple Class Definition

- Simple Class Definition
 - Shorthand Case of Inheritance
- Example:

```
class SameColor = Color;
```

Equivalent to:

inheritance ←

```
class SameColor  
  extends Color;  
end SameColor;
```

- Often used for introducing new names of types:

```
type Resistor = Real;
```

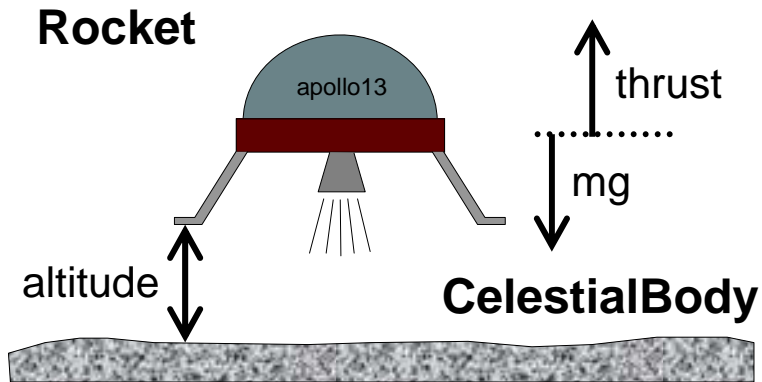
```
connector MyPin = Pin;
```

Inheritance Through Modification

- Modification is a concise way of combining inheritance with declaration of classes or instances
- A modifier modifies a declaration equation in the inherited class
- Example: The class `Real` is inherited, modified with a different `start` value equation, and instantiated as an `altitude` variable:

```
...  
  Real altitude(start= 59404);  
...
```

The Moon Landing - Example Using Inheritance (I)

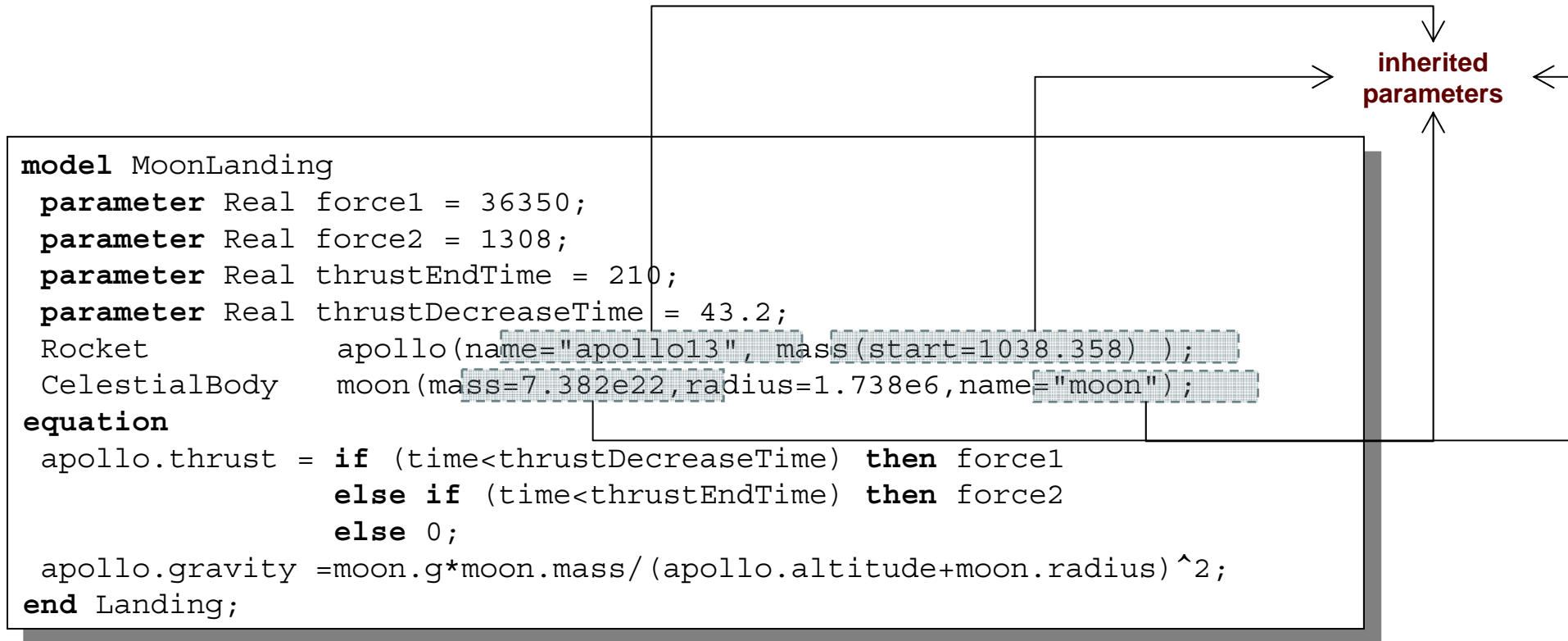


```
model Body "generic body"  
  Real mass;  
  String name;  
end Body;
```

```
model CelestialBody  
  extends Body;  
  constant Real g = 6.672e-11;  
  parameter Real radius;  
end CelestialBody;
```

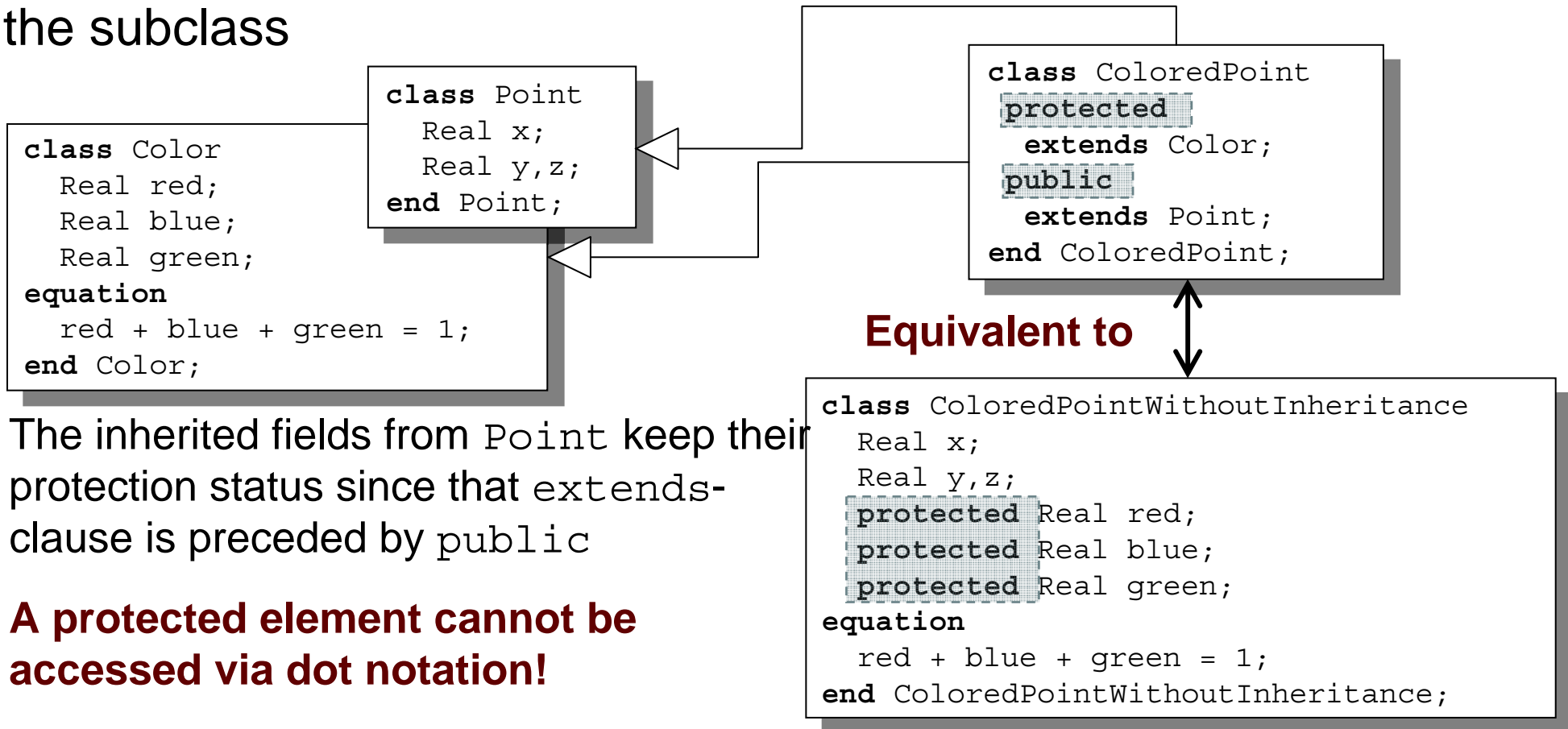
```
model Rocket "generic rocket class"  
  extends Body;  
  parameter Real massLossRate=0.000277;  
  Real altitude(start= 59404);  
  Real velocity(start= -2003);  
  Real acceleration;  
  Real thrust;  
  Real gravity;  
equation  
  thrust-mass*gravity= mass*acceleration;  
  der(mass)= -massLossRate*abs(thrust);  
  der(altitude)= velocity;  
  der(velocity)= acceleration;  
end Rocket;
```

The Moon Landing - Example using Inheritance (II)



Inheritance of Protected Elements

If an `extends`-clause is preceded by the `protected` keyword, all inherited elements from the superclass become protected elements of the subclass



The inherited fields from `Point` keep their protection status since that `extends`-clause is preceded by `public`

A protected element cannot be accessed via dot notation!

Exercises Part II

(30 minutes)

Exercises Part II

- Start OMNotebook
 - Start->Programs->OpenModelica->OMNotebook
 - Open File: Exercises-ModelicaTutorial.onb
- Open Exercises-ModelicaTutorial.pdf

Exercises 2.1 and 2.2

- Open the **Exercises-ModelicaTutorial.onb** found in the Tutorial directory.
- **Exercise 2.1.** Simulate and plot the HelloWorld example. Do a slight change in the model, re-simulate and re-plot. Try command-completion, `val()`, etc.

```
class HelloWorld "A simple equation"  
  Real x(start=1);  
equation  
  der(x) = -x;  
end HelloWorld;
```

```
simulate(HelloWorld, stopTime = 2)  
plot(x)
```

- Locate the VanDerPol model in DrModelica (link from Section 2.1), using OMNotebook!
- **Exercise 2.2:** Simulate and plot VanDerPol. Do a slight change in the model, re-simulate and re-plot.

Exercise 2.1 – Hello World!

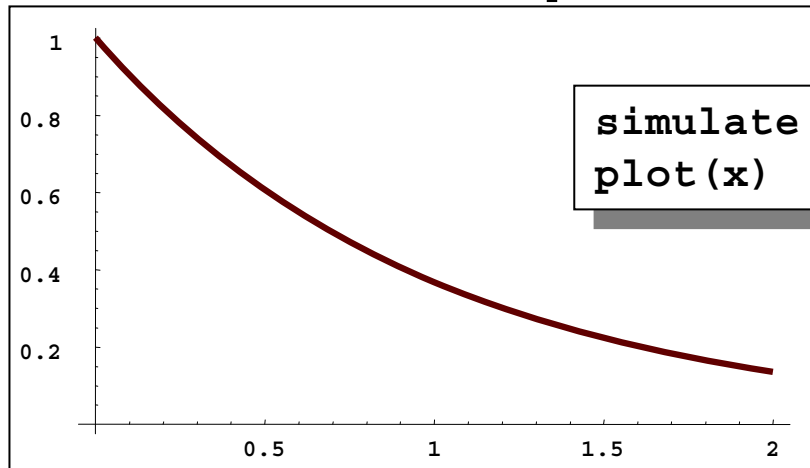
A Modelica “Hello World” model

Equation: $x' = -x$

Initial condition: $x(0) = 1$

```
class HelloWorld "A simple equation"  
  parameter Real a=-1;  
  Real x(start=1);  
equation  
  der(x) = a*x;  
end HelloWorld;
```

Simulation in OpenModelica environment

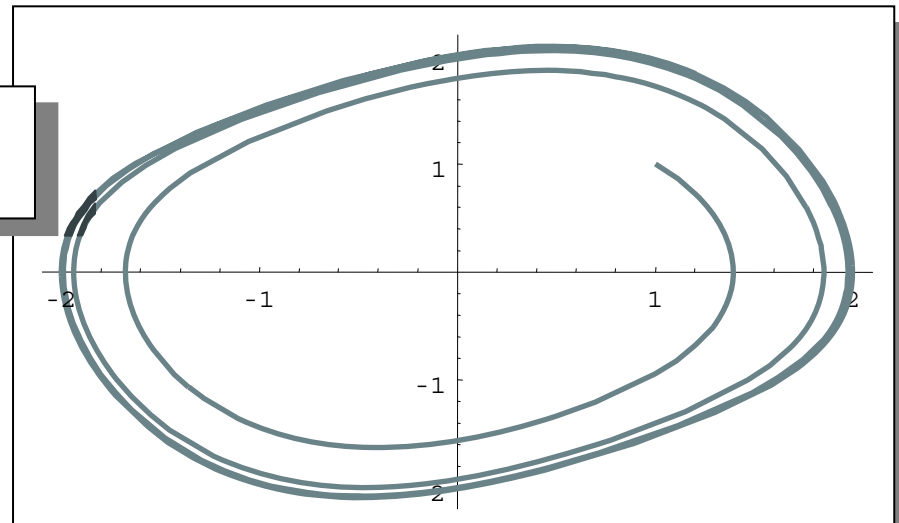


```
simulate(HelloWorld, stopTime = 2)  
plot(x)
```

Exercise 2.2 – Van der Pol Oscillator

```
class VanDerPol "Van der Pol oscillator model"  
  Real x(start = 1) "Descriptive string for x"; // x starts at 1  
  Real y(start = 1) "y coordinate";           // y starts at 1  
  parameter Real lambda = 0.3;  
equation  
  der(x) = y;                                // This is the 1st diff equation //  
  der(y) = -x + lambda*(1 - x*x)*y;          /* This is the 2nd diff equation */  
end VanDerPol;
```

```
simulate(VanDerPol, stopTime = 25)  
plotParametric(x, y)
```



Exercise 2.5 – Model the system below

- Model this Simple System of Equations in Modelica

$$\dot{x} = 2 * x * y - 3 * x$$

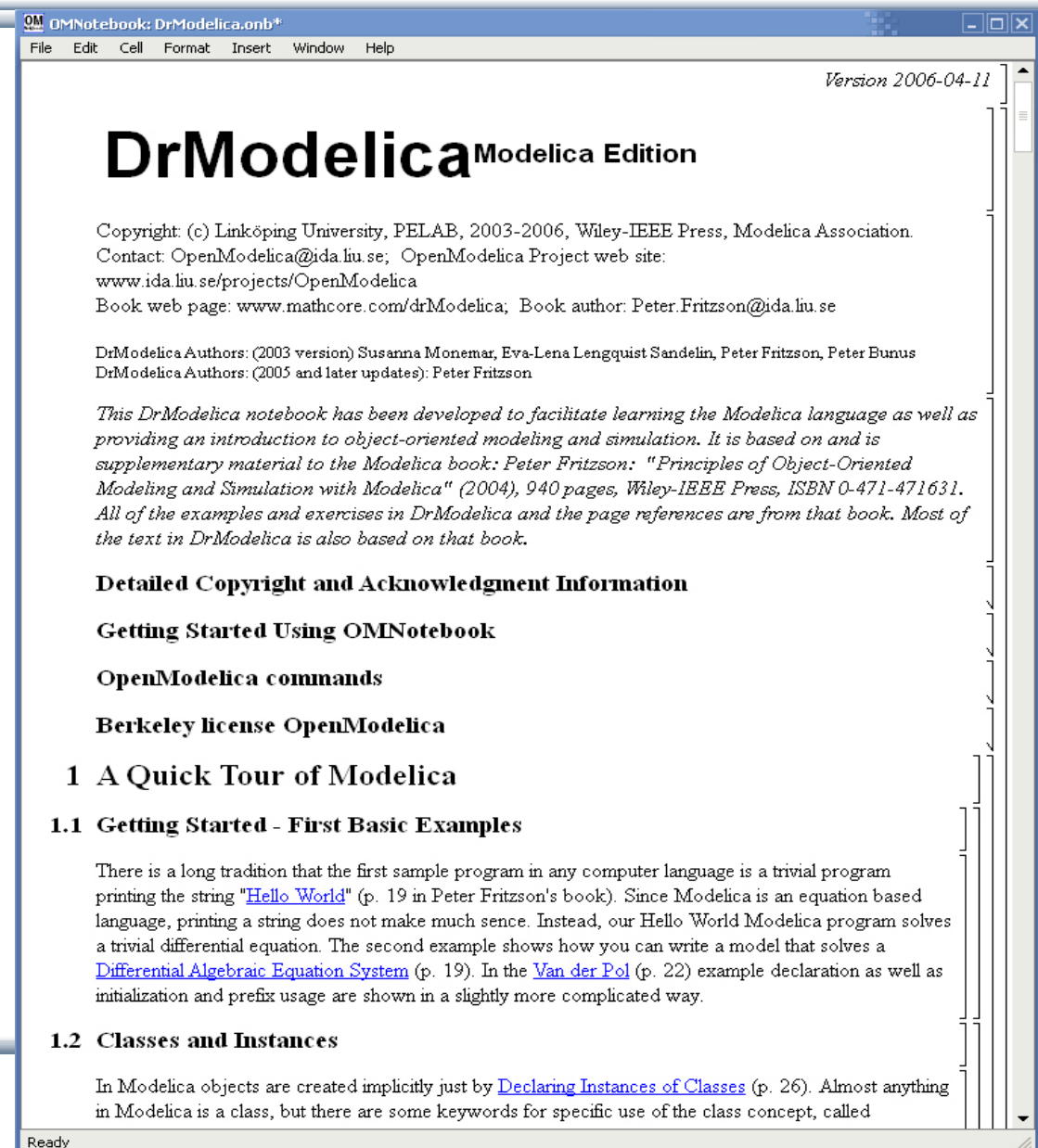
$$\dot{y} = 5 * y - 7 * x * y$$

$$x(0) = 2$$

$$y(0) = 3$$

OMNotebook Electronic Notebook with DrModelica

- Primarily for teaching
- Interactive electronic book
- Platform independent



The screenshot shows a window titled "OMNotebook: DrModelica.onb*" with a menu bar (File, Edit, Cell, Format, Insert, Window, Help) and a status bar (Ready). The main content area displays the following text:

Version 2006-04-11

DrModelica Modelica Edition

Copyright: (c) Linköping University, PELAB, 2003-2006, Wiley-IEEE Press, Modelica Association.
Contact: OpenModelica@ida.liu.se; OpenModelica Project web site:
www.ida.liu.se/projects/OpenModelica
Book web page: www.mathcore.com/drModelica; Book author: Peter.Fritzson@ida.liu.se

DrModelica Authors: (2003 version) Susanna Monemar, Eva-Lena Lengquist Sandelin, Peter Fritzson, Peter Bunus
DrModelica Authors: (2005 and later updates): Peter Fritzson

This DrModelica notebook has been developed to facilitate learning the Modelica language as well as providing an introduction to object-oriented modeling and simulation. It is based on and is supplementary material to the Modelica book: Peter Fritzson: "Principles of Object-Oriented Modeling and Simulation with Modelica" (2004), 940 pages, Wiley-IEEE Press, ISBN 0-471-471631. All of the examples and exercises in DrModelica and the page references are from that book. Most of the text in DrModelica is also based on that book.

Detailed Copyright and Acknowledgment Information

Getting Started Using OMNotebook

OpenModelica commands

Berkeley license OpenModelica

1 A Quick Tour of Modelica

1.1 Getting Started - First Basic Examples

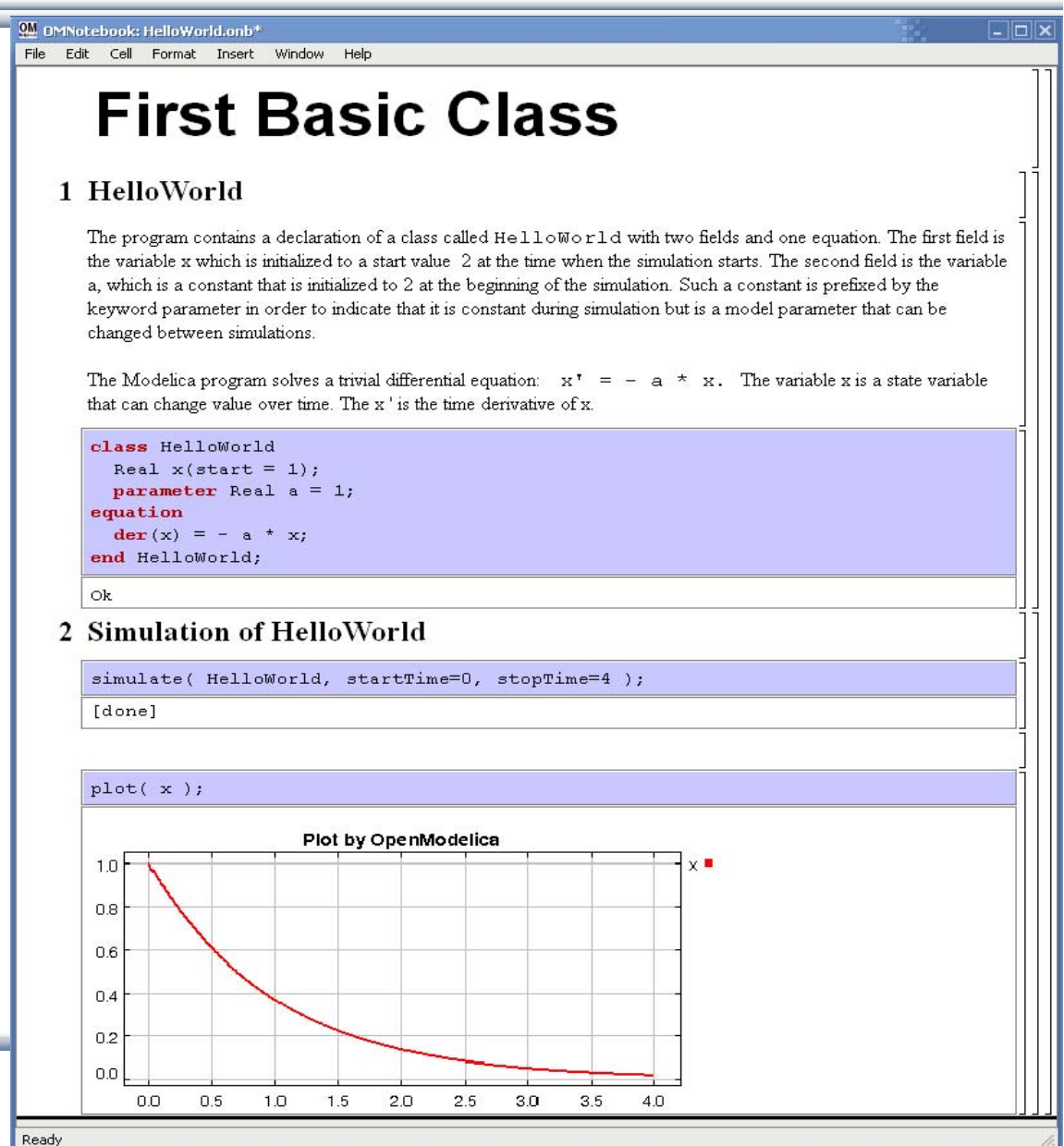
There is a long tradition that the first sample program in any computer language is a trivial program printing the string "Hello World" (p. 19 in Peter Fritzson's book). Since Modelica is an equation based language, printing a string does not make much sense. Instead, our Hello World Modelica program solves a trivial differential equation. The second example shows how you can write a model that solves a [Differential Algebraic Equation System](#) (p. 19). In the [Van der Pol](#) (p. 22) example declaration as well as initialization and prefix usage are shown in a slightly more complicated way.

1.2 Classes and Instances

In Modelica objects are created implicitly just by [Declaring Instances of Classes](#) (p. 26). Almost anything in Modelica is a class, but there are some keywords for specific use of the class concept, called

Cells with both Text and Graphics

- Text Cells
- Input Cells
- Graphic Cells



OMNotebook: HelloWorld.onb*

File Edit Cell Format Insert Window Help

First Basic Class

1 HelloWorld

The program contains a declaration of a class called HelloWorld with two fields and one equation. The first field is the variable x which is initialized to a start value 2 at the time when the simulation starts. The second field is the variable a , which is a constant that is initialized to 2 at the beginning of the simulation. Such a constant is prefixed by the keyword parameter in order to indicate that it is constant during simulation but is a model parameter that can be changed between simulations.

The Modelica program solves a trivial differential equation: $x' = -a * x$. The variable x is a state variable that can change value over time. The x' is the time derivative of x .

```
class HelloWorld
  Real x(start = 1);
  parameter Real a = 1;
equation
  der(x) = - a * x;
end HelloWorld;
```

Ok

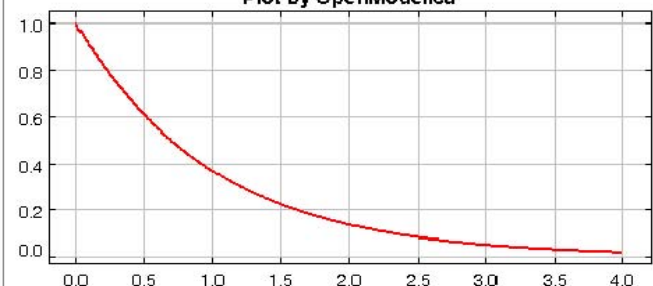
2 Simulation of HelloWorld

```
simulate( HelloWorld, startTime=0, stopTime=4 );
```

[done]

```
plot( x );
```

Plot by OpenModelica

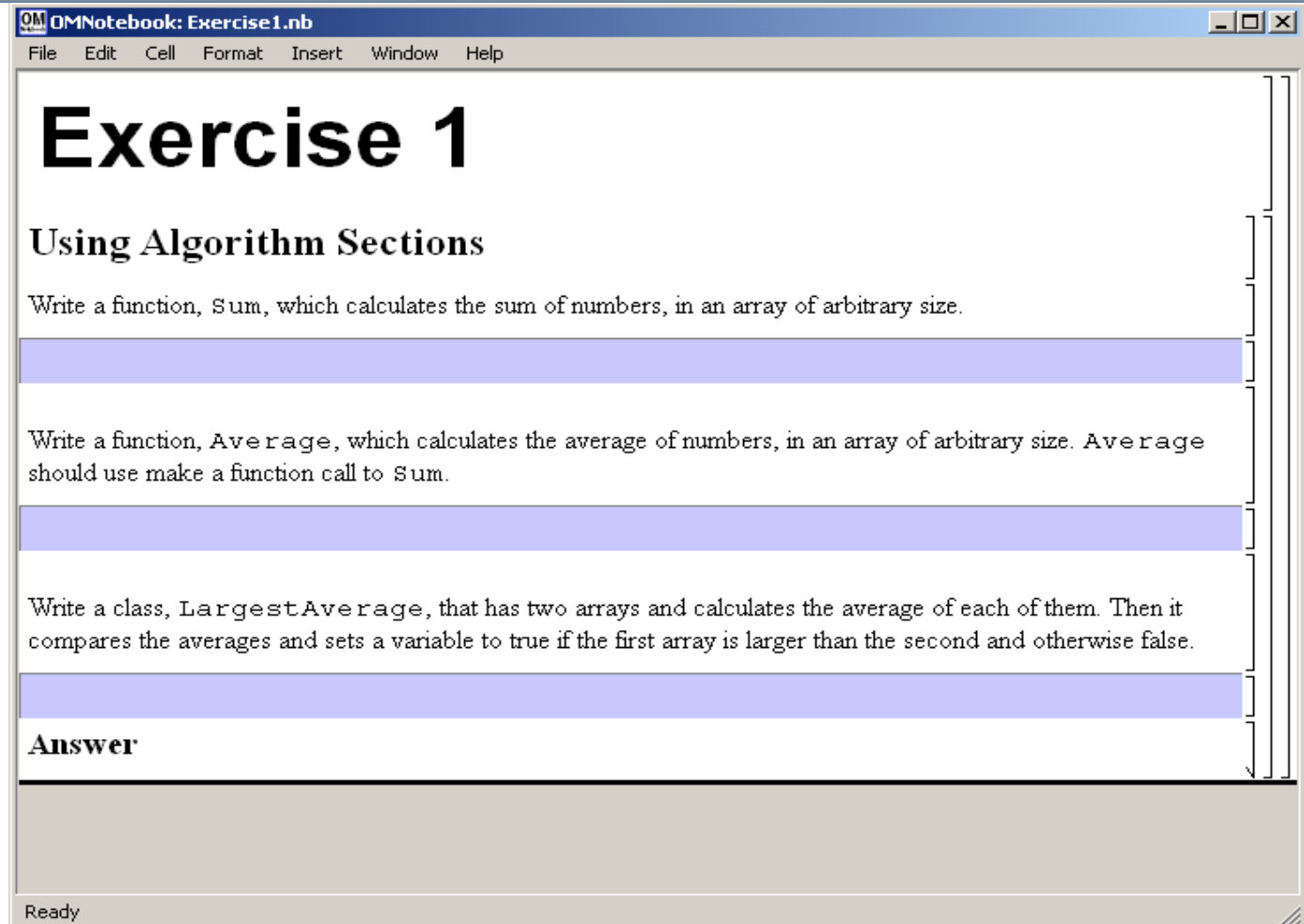


Time	x
0.0	1.0
0.5	0.6
1.0	0.4
1.5	0.25
2.0	0.15
2.5	0.08
3.0	0.05
3.5	0.03
4.0	0.02

Ready

Exercises and Answers in OMNotebook DrModelica

Easy to follow
Exercises with
solutions hidden in
collapsed cells



The screenshot shows a window titled "OMNotebook: Exercise1.nb" with a menu bar (File, Edit, Cell, Format, Insert, Window, Help). The main content area displays "Exercise 1" and "Using Algorithm Sections". It contains three paragraphs of text, each followed by a collapsed cell (indicated by a blue bar and a right-side handle). The first paragraph asks for a function `Sum`. The second asks for a function `Average` that uses `Sum`. The third asks for a class `LargestAverage` that compares two arrays. Below the paragraphs is an "Answer" section, which is also collapsed. The status bar at the bottom shows "Ready".

Exercise 1

Using Algorithm Sections

Write a function, `Sum`, which calculates the sum of numbers, in an array of arbitrary size.

Write a function, `Average`, which calculates the average of numbers, in an array of arbitrary size. `Average` should use make a function call to `Sum`.

Write a class, `LargestAverage`, that has two arrays and calculates the average of each of them. Then it compares the averages and sets a variable to true if the first array is larger than the second and otherwise false.

Answer

Ready

Some OMNotebook Commands

- Shift-return (evaluates a cell)
- File Menu (open, close, etc.)
- Text Cursor (vertical), Cell cursor (horizontal)
- Cell types: text cells & executable code cells
- Copy, paste, group cells
- Copy, paste, group text
- Command Completion (shift-tab)

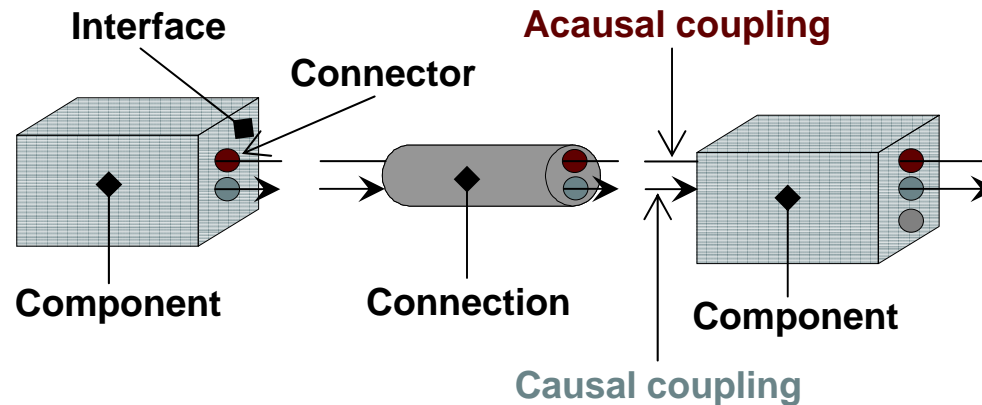
- (see also OpenModelica Users Guide or OMNotebook help)

OMNotebook Cell Editing Exercises (optional)

- Select and copy a cell (tree to the right)
 - Position the horizontal cell cursor: first click between cells; then click once more on top of the horizontal line
 - Paste the cell
-
- Note: You can find most Users Guide examples in the UsersGuideExamples.onb in the testmodels directory of the OpenModelica installation.

Components, Connectors and Connections – Modelica Libraries and Graphical Modeling

Software Component Model



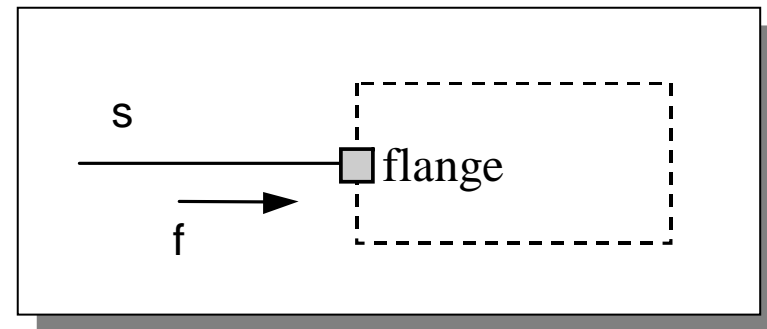
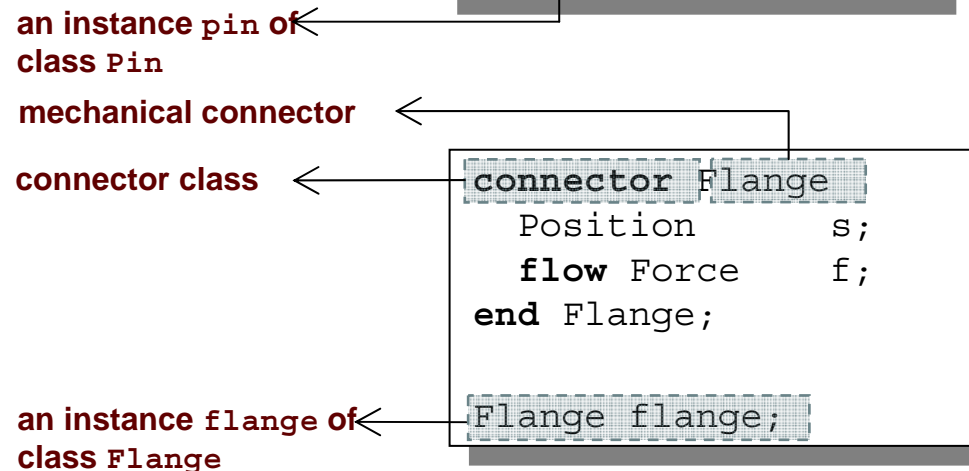
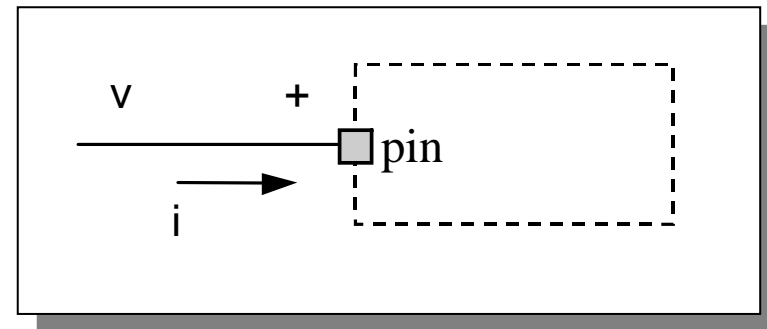
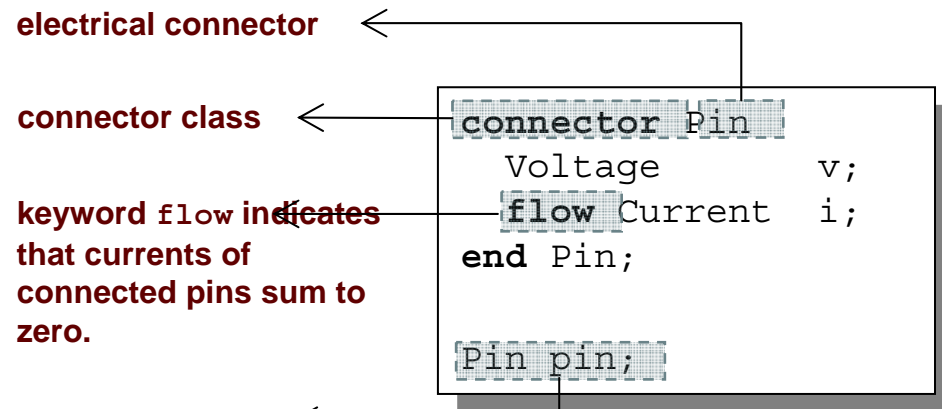
A component class should be defined *independently of the environment*, very essential for *reusability*

A component may internally consist of other components, i.e. *hierarchical* modeling

Complex systems usually consist of large numbers of *connected* components

Connectors and Connector Classes

Connectors are instances of *connector classes*



The `flow` prefix

Two kinds of variables in connectors:

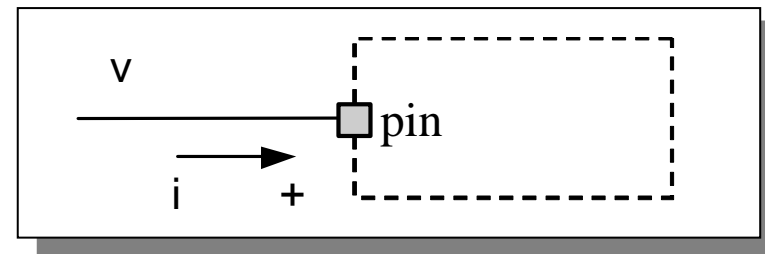
- *Non-flow variables* *potential* or energy level
- *Flow variables* represent some kind of flow

Coupling

- *Equality coupling*, for non-`flow` variables
- *Sum-to-zero coupling*, for `flow` variables

The value of a `flow` variable is *positive* when the current or the flow is *into* the component

positive flow direction:



Physical Connector

- Classes Based on Energy Flow

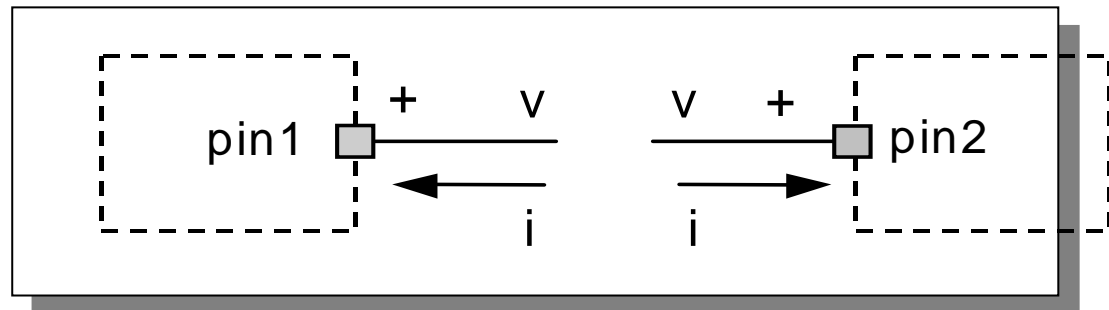
Domain Type	Potential	Flow	Carrier	Modelica Library
Electrical	Voltage	Current	Charge	Electrical. Analog
Translational	Position	Force	Linear momentum	Mechanical. Translational
Rotational	Angle	Torque	Angular momentum	Mechanical. Rotational
Magnetic	Magnetic potential	Magnetic flux rate	Magnetic flux	
Hydraulic	Pressure	Volume flow	Volume	HyLibLight
Heat	Temperature	Heat flow	Heat	HeatFlow1D
Chemical	Chemical potential	Particle flow	Particles	Under construction
Pneumatic	Pressure	Mass flow	Air	PneuLibLight

connect-equations

Connections between connectors are realized as *equations* in Modelica

```
connect (connector1,connector2)
```

The two arguments of a `connect`-equation must be references to *connectors*, either to be declared directly *within* the *same class* or be *members* of one of the declared variables in that class



```
Pin pin1, pin2;  
//A connect equation  
//in Modelica:  
connect (pin1, pin2);
```

Corresponds to

```
pin1.v = pin2.v;  
pin1.i + pin2.i = 0;
```

Connection Equations

```
Pin pin1, pin2;  
//A connect equation  
//in Modelica  
connect(pin1, pin2);
```

Corresponds to

```
pin1.v = pin2.v;  
pin1.i + pin2.i = 0;
```

Multiple connections are possible:

```
connect(pin1, pin2); connect(pin1, pin3); ... connect(pin1, pinN);
```

Each primitive connection set of **nonflow** variables is used to generate equations of the form:

$$v_1 = v_2 = v_3 = \dots v_n$$

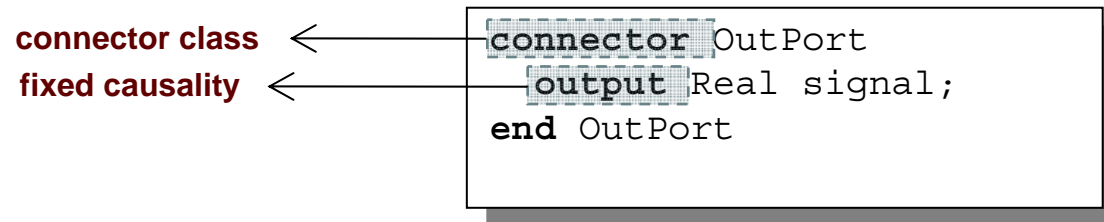
Each primitive connection set of **flow** variables is used to generate *sum-to-zero* equations of the form:

$$i_1 + i_2 + \dots (-i_k) + \dots i_n = 0$$

Acausal, Causal, and Composite Connections

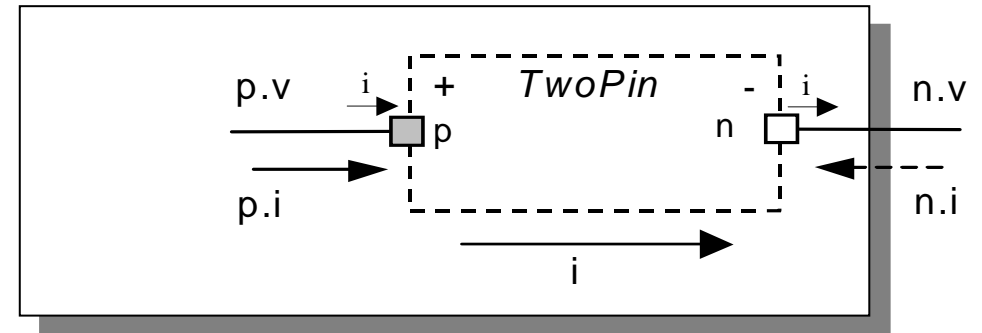
Two *basic* and one *composite* kind of connection in Modelica

- Acausal connections
- Causal connections, also called signal connections
- Composite connections, also called structured connections, composed of basic or composite connections



Common Component Structure

The base class `TwoPin` has two connectors `p` and `n` for positive and negative pins respectively



partial class
(cannot be
instantiated)

positive pin
negative pin

```

partial model TwoPin
  Voltage v
  Current i
  Pin p;
  Pin n;
equation
  v = p.v - n.v;
  0 = p.i + n.i;
  i = p.i;
end TwoPin;
// TwoPin is same as OnePort in
// Modelica.Electrical.Analog.Interfaces
    
```

```

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

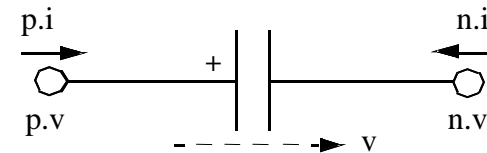
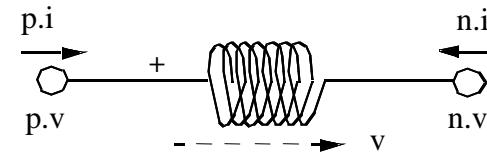
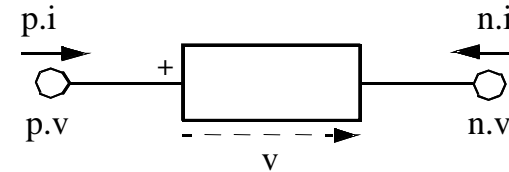
electrical connector class

Electrical Components

```
model Resistor "Ideal electrical resistor"  
  extends TwoPin;  
  parameter Real R;  
equation  
  R*i = v;  
end Resistor;
```

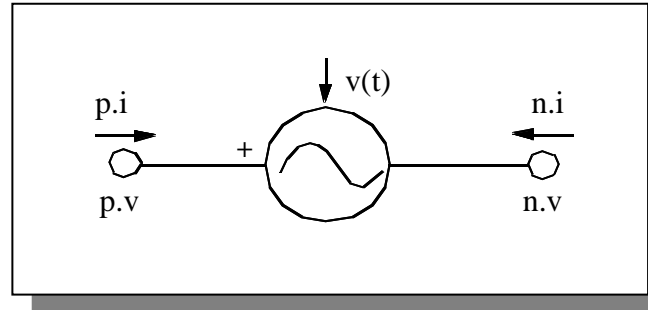
```
model Inductor "Ideal electrical inductor"  
  extends TwoPin;  
  parameter Real L "Inductance";  
equation  
  L*der(i) = v;  
end Inductor;
```

```
model Capacitor "Ideal electrical capacitor"  
  extends TwoPin;  
  parameter Real C ;  
equation  
  i=C*der(v) ;  
end Capacitor;
```

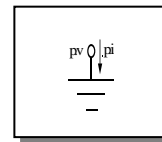


Electrical Components cont'

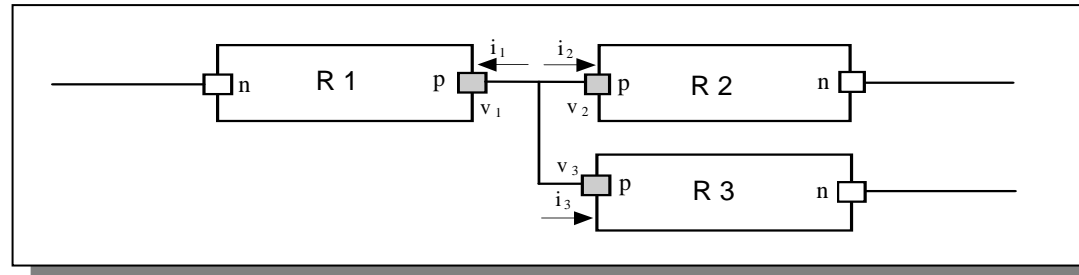
```
model Source
  extends TwoPin;
  parameter Real A,w;
equation
  v = A*sin(w*time);
end Resistor;
```



```
model Ground
  Pin p;
equation
  p.v = 0;
end Ground;
```



Resistor Circuit



```
model ResistorCircuit
  Resistor R1(R=100);
  Resistor R2(R=200);
  Resistor R3(R=300);
equation
  connect(R1.p, R2.p);
  connect(R1.p, R3.p);
end ResistorCircuit;
```

Corresponds to

```
R1.p.v = R2.p.v;
R1.p.v = R3.p.v;
R1.p.i + R2.p.i + R3.p.i = 0;
```

Modelica Standard Library - Graphical Modeling

- *Modelica Standard Library* (called Modelica) is a standardized predefined package developed by Modelica Association
- It can be used freely for both commercial and noncommercial purposes under the conditions of *The Modelica License*.
- Modelica libraries are available online including documentation and source code from <http://www.modelica.org/library/library.html>

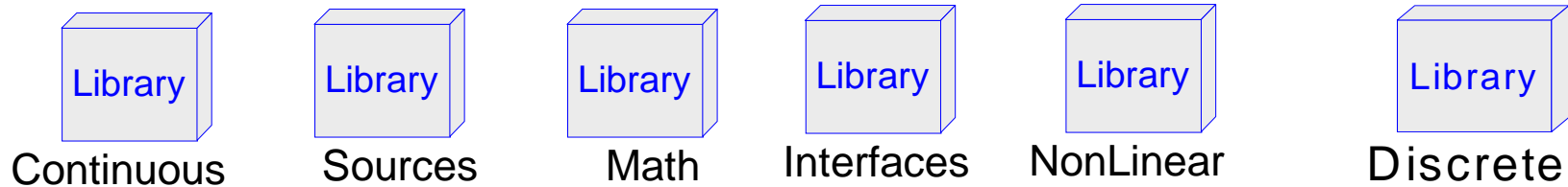
Modelica Standard Library cont'

Modelica Standard Library contains components from various application areas, with the following sublibraries:

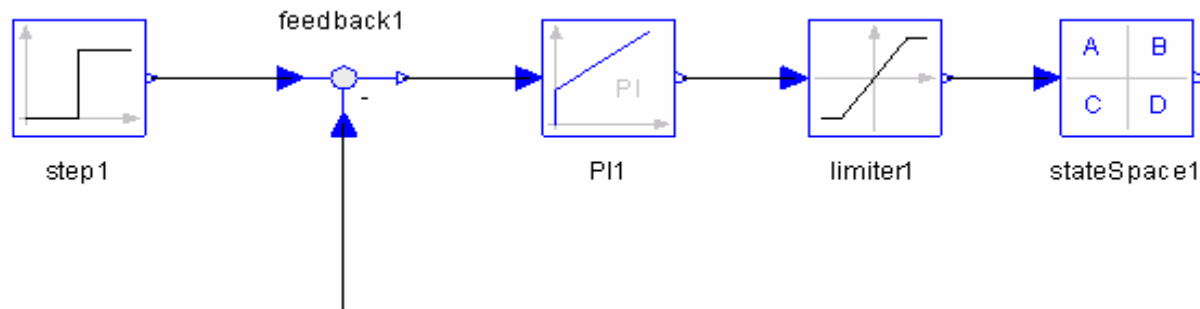
- **Blocks** Library for basic input/output control blocks
- **Constants** Mathematical constants and constants of nature
- **Electrical** Library for electrical models
- **Icons** Icon definitions
- **Math** Mathematical functions
- **Mechanics** Library for mechanical systems
- **Media** Media models for liquids and gases
- **Slunits** Type definitions based on SI units according to ISO 31-1992
- **Stategraph** Hierarchical state machines (analogous to Statecharts)
- **Thermal** Components for thermal systems
- **Utilities** Utility functions especially for scripting

Modelica.Blocks

This library contains input/output blocks to build up block diagrams.



Example:



Modelica.Electrical

Electrical components for building analog, digital, and multiphase circuits



Analog



Digital

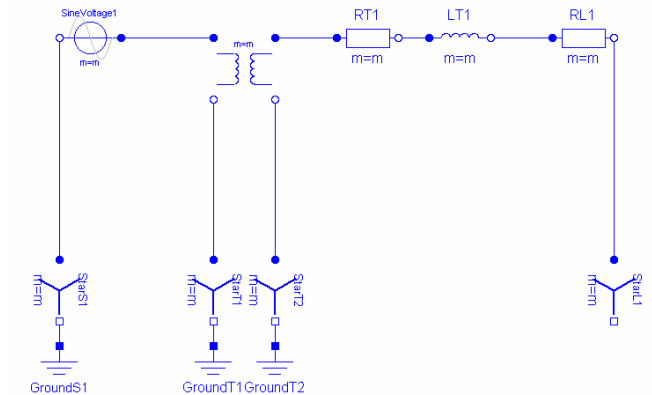
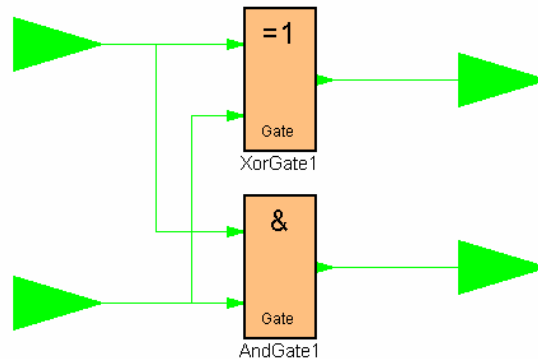
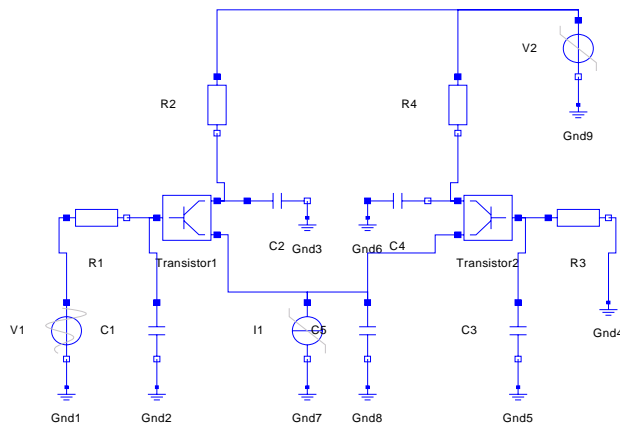


Machines



MultiPhase

Examples:

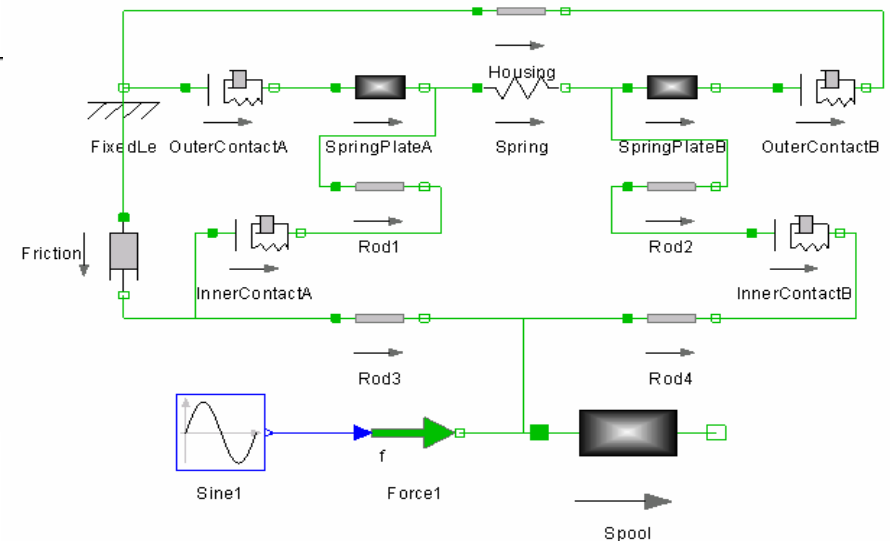
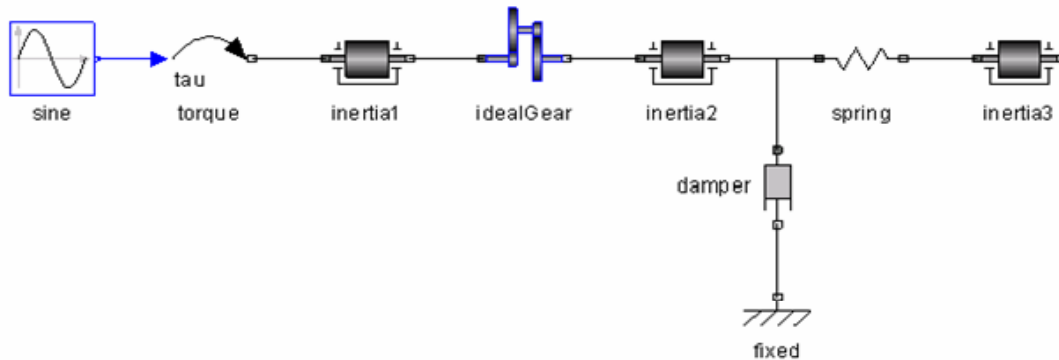


Modelica.Mechanics

Package containing components for mechanical systems

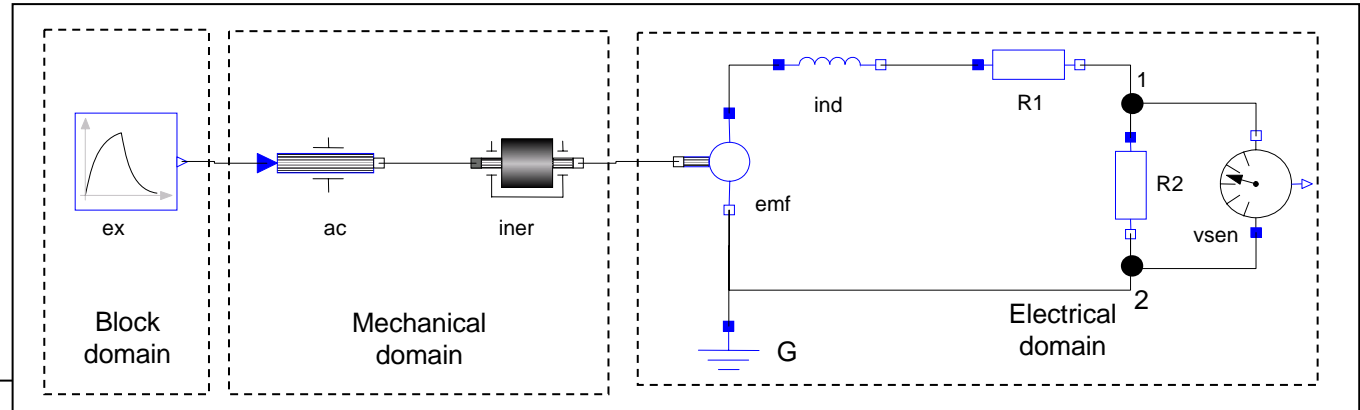
Subpackages:

- Rotational 1-dimensional rotational mechanical components
- Translational 1-dimensional translational mechanical components
- MultiBody 3-dimensional mechanical components



Connecting Components from Multiple Domains

- Block domain
- Mechanical domain
- Electrical domain



model Generator

```
Modelica.Mechanics.Rotational.Accelerate ac;  
Modelica.Mechanics.Rotational.Inertia iner;  
Modelica.Electrical.Analog.Basic.EMF emf(k=-1);  
Modelica.Electrical.Analog.Basic.Inductor ind(L=0.1);  
Modelica.Electrical.Analog.Basic.Resistor R1,R2;  
Modelica.Electrical.Analog.Basic.Ground G;  
Modelica.Electrical.Analog.Sensors.VoltageSensor vsens;  
Modelica.Blocks.Sources.Exponentials ex(riseTime={2},riseTimeConst={1});
```

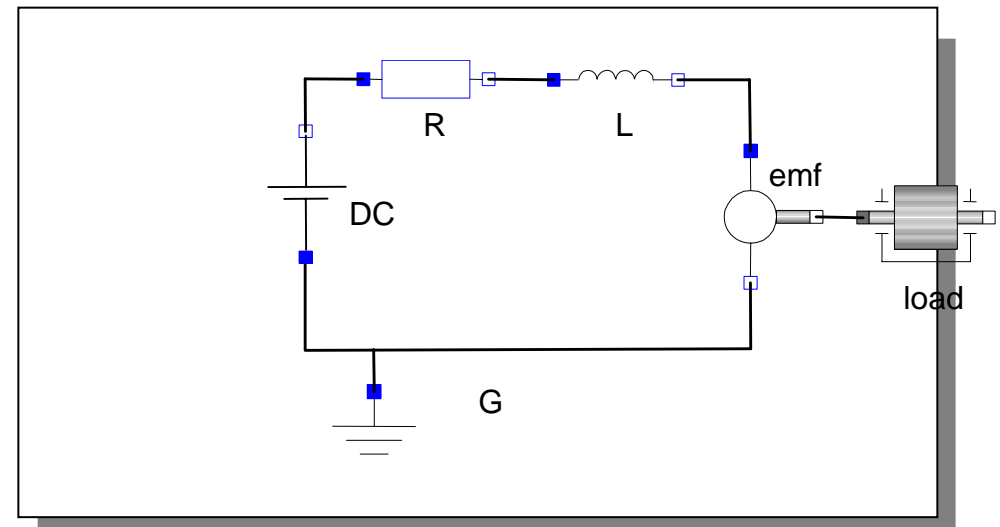
equation

```
connect(ac.flange_b, iner.flange_a); connect(iner.flange_b, emf.flange_b);  
connect(emf.p, ind.p); connect(ind.n, R1.p); connect(emf.n, G.p);  
connect(emf.n, R2.n); connect(R1.n, R2.p); connect(R2.p, vsens.n);  
connect(R2.n, vsens.p); connect(ex.outPort, ac.inPort);  
end Generator;
```

DCMotor Model Multi-Domain (Electro-Mechanical)

A DC motor can be thought of as an electrical circuit which also contains an electromechanical component.

```
model DCMotor
  Resistor R(R=100);
  Inductor L(L=100);
  VsourceDC DC(f=10);
  Ground G;
  EMF emf(k=10,J=10, b=2);
  Inertia load;
equation
  connect(DC.p,R.n);
  connect(R.p,L.n);
  connect(L.p, emf.n);
  connect(emf.p, DC.n);
  connect(DC.n,G.p);
  connect(emf.flange,load.flange);
end DCMotor;
```



Graphical Modeling

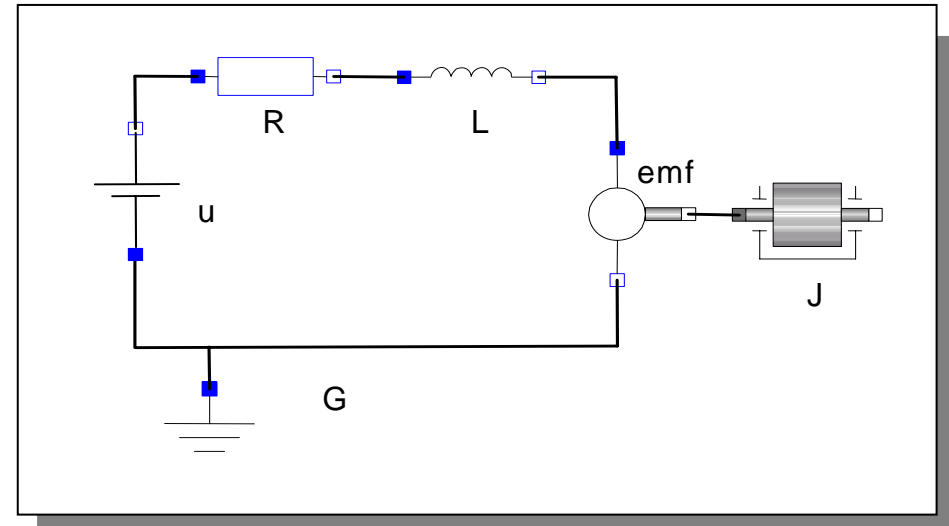
Exercises Part III

Graphical Modeling Exercises

(Probably not enough time
Need to install simForge)

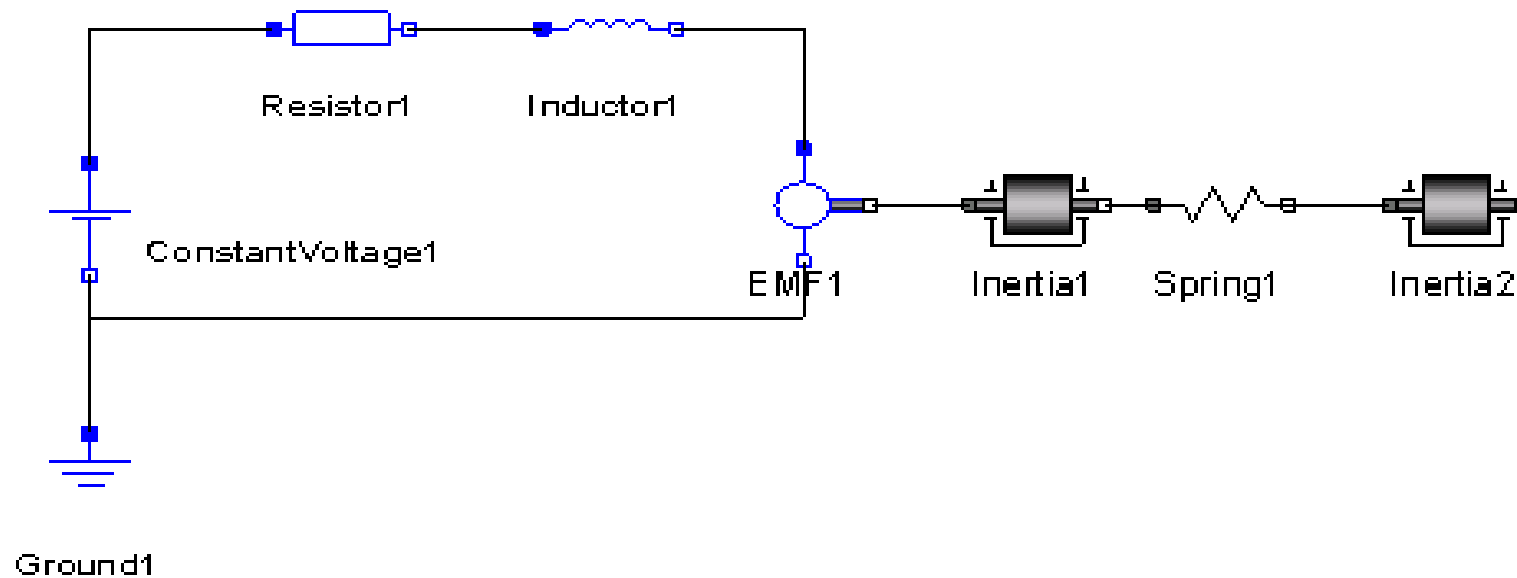
Exercise 3.1

- Draw the `DCMotor` model using the graphic connection editor using models from the following Modelica libraries:
`Mechanics.Rotational`,
`Electrical.Analog.Basic`,
`Electrical.Analog.Sources`
- Simulate it for 15s and plot the variables for the outgoing rotational speed on the inertia axis and the voltage on the voltage source (denoted `u` in the figure) in the same plot.



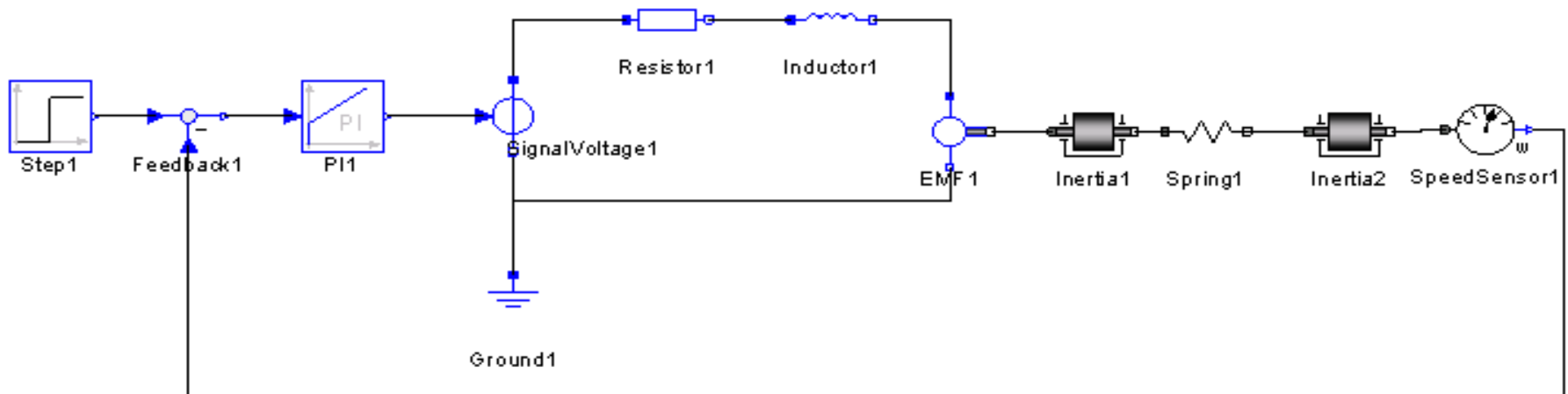
Exercise 3.2

- If there is enough time: Add a torsional spring to the outgoing shaft and another inertia element. Simulate again and see the results. Adjust some parameters to make a rather stiff spring.



Exercise 3.3

- If there is enough time: Add a PI controller to the system and try to control the rotational speed of the outgoing shaft. Verify the result using a step signal for input. Tune the PI controller by changing its parameters in MathModelica.

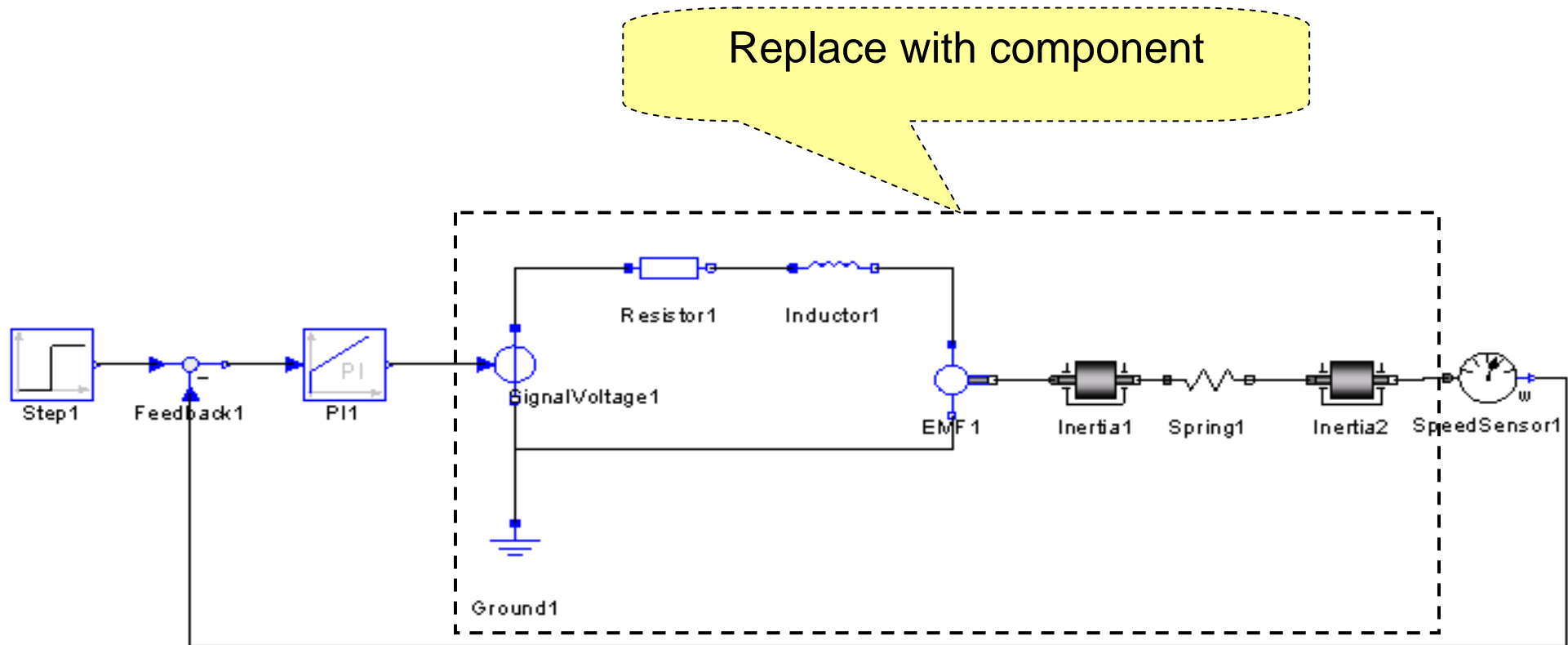


Live example – Graphical Modeling

- Building a component with icon

Exercise 3.4

- Make a component of the model in Exercise 2.2, and use it when building the model in exercise 2.3.



The End

OpenModelica: www.openmodelica.org

Additional non-covered slides for
Algorithms and Functions
and
Discrete Events and Hybrid Systems

Algorithms and Functions

Algorithm Sections

Whereas equations are very well suited for physical modeling, there are situations where computations are more conveniently expressed as algorithms, i.e., sequences of instructions, also called statements

```
algorithm
...
<statements>
...
<some keyword>
```

Algorithm sections can be embedded among equation sections

```
equation
  x = y*2;
  z = w;
algorithm
  x1 := z+x;
  x2 := y-5;
  x1 := x2+y;
equation
  u = x1+x2;
  ...
```

Iteration using for-statements in Algorithm sections

```
for <iteration-variable> in <iteration-set-expression> loop
  <statement1>
  <statement2>
  ...
end for
```

The general structure of a for-statement with a single iterator

```
class SumZ
  parameter Integer n = 5;
  Real[n] z(start = {10,20,30,40,50});
  Real sum;
algorithm
  sum := 0;
  for i in 1:n loop // 1:5 is {1,2,3,4,5}
    sum := sum + z[i];
  end for;
end SumZ;
```

A simple for-loop summing the five elements of the vector z, within the class SumZ

Examples of for-loop headers with different range expressions

```
for k in 1:10+2 loop // k takes the values 1,2,3,...,12
for i in {1,3,6,7} loop // i takes the values 1, 3, 6, 7
for r in 1.0 : 1.5 : 5.5 loop // r takes the values 1.0, 2.5, 4.0, 5.5
```

Iterations using while-stmts in Algorithm Sections

```
while <conditions> loop  
  <statements>  
end while;
```

The general structure of a while-loop with a single iterator.

```
class SumSeries  
  parameter Real eps = 1.E-6;  
  Integer i;  
  Real sum;  
  Real delta;  
algorithm  
  i := 1;  
  delta := exp(-0.01*i);  
  while delta >= eps loop  
    sum := sum + delta;  
    i := i+1;  
    delta := exp(-0.01*i);  
  end while;  
end SumSeries;
```

The example class SumSeries shows the while-loop construct used for summing a series of exponential terms until the loop condition is violated, i.e., the terms become smaller than eps.

if-statements

```
if <condition> then
  <statements>
elseif <condition> then
  <statements>
else
  <statements>
end if
```

The if-statements used in the class SumVector perform a combined summation and computation on a vector v .

The general structure of if-statements.

The elseif-part is optional and can occur zero or more times whereas the optional else-part can occur at most once

```
class SumVector
  Real sum;
  parameter Real v[5] = {100,200,-300,400,500};
  parameter Integer n = size(v,1);
algorithm
  sum := 0;
  for i in 1:n loop
    if v[i]>0 then
      sum := sum + v[i];
    elseif v[i] > -1 then
      sum := sum + v[i] -1;
    else
      sum := sum - v[i];
    end if;
  end for;
end SumVector;
```

Function Declaration

The structure of a typical function declaration is as follows:

```
function <functionname>
  input Type1 in1;
  input Type2 in2;
  input Type3 in3;
  ...
  output TypeO1 out1;
  output TypeO2 out2;
  ...
  protected
    <local variables>
  ...
  algorithm
  ...
    <statements>
  ...
end <functionname>;
```

All internal parts of a function are optional, the following is also a legal function:

```
function <functionname>
end <functionname>;
```

Modelica functions are *declarative mathematical functions*:

- Always return the same result(s) given the same input argument values

Function Call

Two basic forms of arguments in Modelica function calls:

- *Positional* association of actual arguments to formal parameters
- *Named* association of actual arguments to formal parameters

Example function called on next page:

```
function PolynomialEvaluator
  input Real A[:]; // array, size defined
                        // at function call time
  input Real x := 1.0; // default value 1.0 for x
  output Real sum;
protected
  Real xpower; // local variable xpower
algorithm
  sum := 0;
  xpower := 1;
  for i in 1:size(A,1) loop
    sum := sum + A[i]*xpower;
    xpower := xpower*x;
  end for;
end PolynomialEvaluator;
```

The function PolynomialEvaluator computes the value of a polynomial given two arguments: a coefficient vector A and a value of x.

Positional and Named Argument Association

Using *positional* association, in the call below the actual argument $\{1, 2, 3, 4\}$ becomes the value of the coefficient vector A , and 21 becomes the value of the formal parameter x .

```
...  
algorithm  
  ...  
  p:= polynomialEvaluator({1,2,3,4},21)
```

The same call to the function `polynomialEvaluator` can instead be made using *named* association of actual parameters to formal parameters.

```
...  
algorithm  
  ...  
  p:= polynomialEvaluator(A={1,2,3,4},x=21)
```

External Functions

It is possible to call functions defined outside the Modelica language, implemented in C or FORTRAN 77

```
function polynomialMultiply
  input Real a[:], b[:];
  output Real c[:] := zeros(size(a,1)+size(b, 1) - 1);
  external
end polynomialMultiply;
```

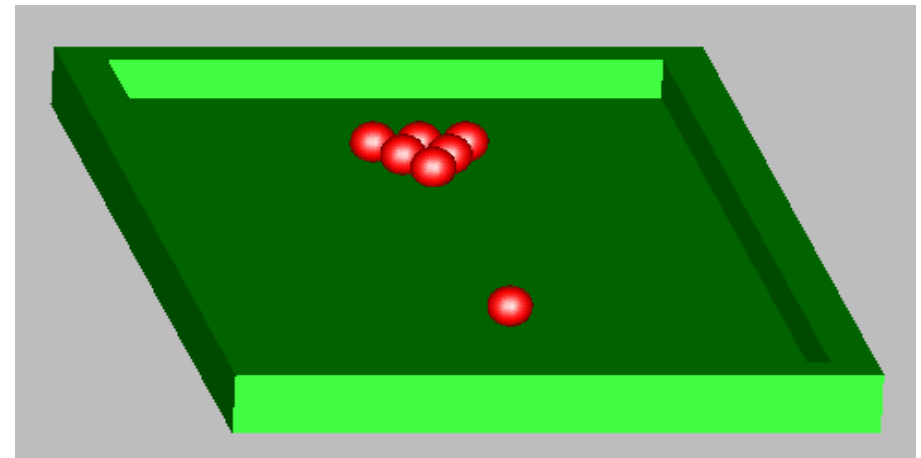
The body of an external function is marked with the keyword **external**

If no language is specified, the implementation language for the external function is assumed to be C. The external function `polynomialMultiply` can also be specified, e.g. via a mapping to a FORTRAN 77 function:

```
function polynomialMultiply
  input Real a[:], b[:];
  output Real c[:] := zeros(size(a,1)+size(b, 1) - 1);
  external "FORTRAN 77"
end polynomialMultiply;
```

**If enough time, Do Exercise 2.6
on Functions and algorithms**

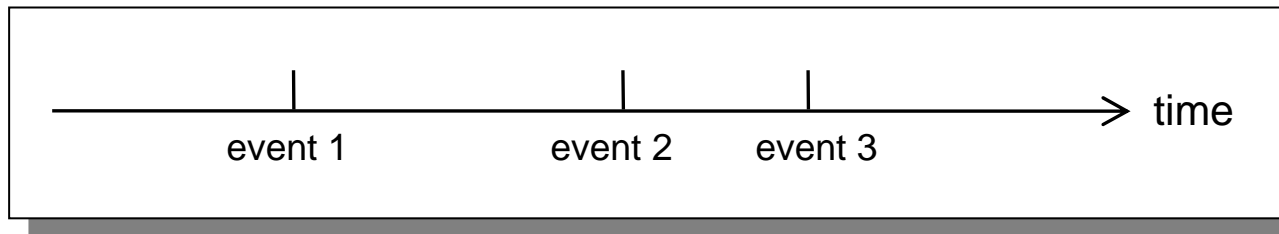
Discrete Events and Hybrid Systems



Picture: Courtesy Hilding Elmqvist

Events

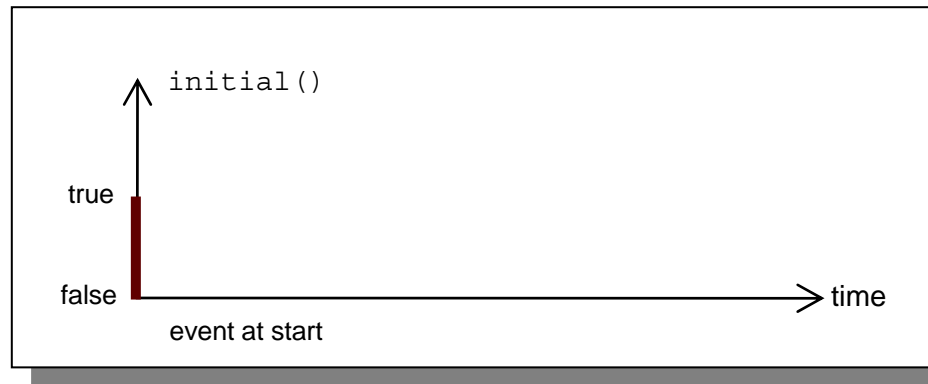
Events are ordered in time and form an event history



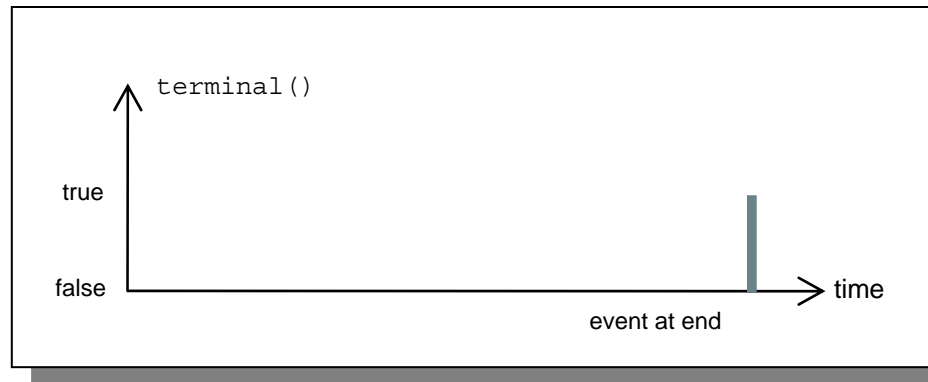
- A *point* in time that is instantaneous, i.e., has zero duration
- An *event condition* that switches from false to true in order for the event to take place
- A set of *variables* that are associated with the event, i.e. are referenced or explicitly changed by equations associated with the event
- Some *behavior* associated with the event, expressed as *conditional equations* that become active or are deactivated at the event. *Instantaneous equations* is a special case of conditional equations that are only active *at* events.

initial and terminal events

Initialization actions are triggered by `initial()`

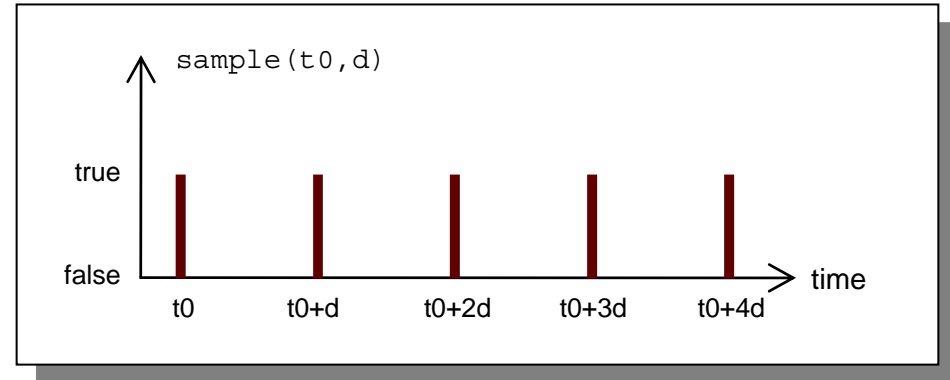


Actions at the end of a simulation are triggered by `terminal()`



Generating Repeated Events

The call `sample(t0, d)` returns true and triggers events at times $t_0 + i \cdot d$, where $i = 0, 1, \dots$



```
class SamplingClock
  parameter Modelica.SIunits.Time first, interval;
  Boolean clock;
equation
  clock = sample(first, interval);
  when clock then
    ...
  end when;
end SamplingClock;
```

Expressing Event Behavior in Modelica

if-equations, if-statements, and if-expressions express different behavior in different operating regions

```
if <condition> then
  <equations>
elseif <condition> then
  <equations>
else
  <equations>
end if;
```

```
model Diode "Ideal diode"
  extends TwoPin;
  Real s;
  Boolean off;
equation
  off = s < 0;
  if off then
    v=s
  else
    v=0;
  end if;
  i = if off then 0 else s;
end Diode;
```

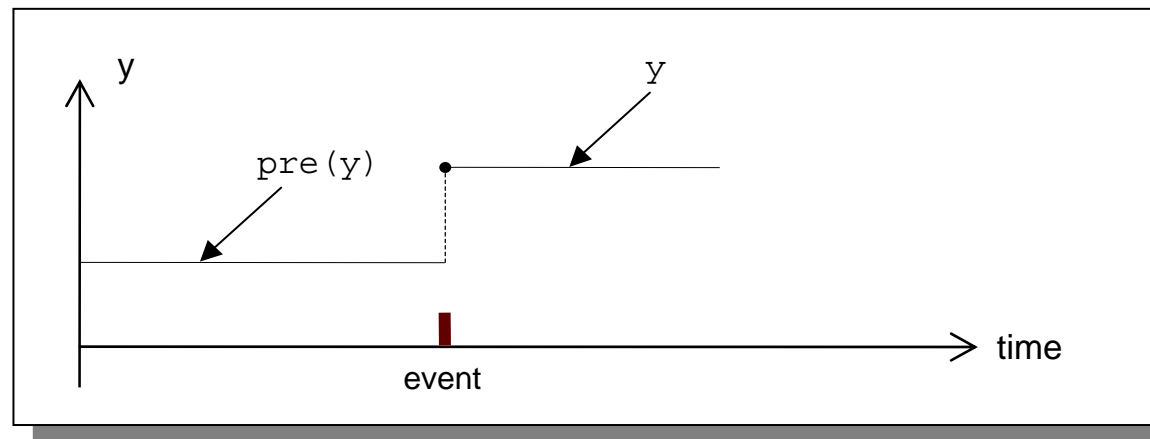
when-equations become active at events

```
when <conditions> then
  <equations>
end when;
```

```
equation
  when x > y.start then
    ...
```

Obtaining Predecessor Values using `pre ()`

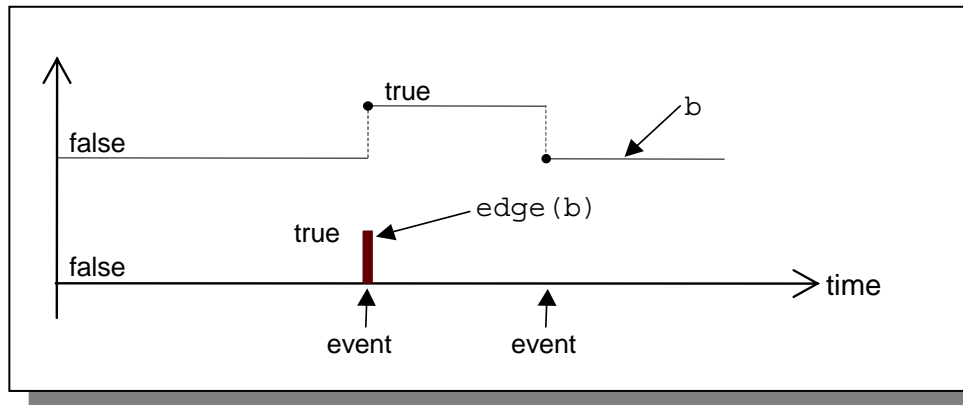
At an event, `pre (y)` gives the previous value of `y` immediately before the event, except for event iteration of multiple events at the same point in time when the value is from the previous iteration



- The variable `y` has one of the basic types `Boolean`, `Integer`, `Real`, `String`, or `enumeration`, a subtype of those, or an array type of one of those basic types or subtypes
- The variable `y` is a discrete-time variable
- The `pre` operator can *not* be used within a function

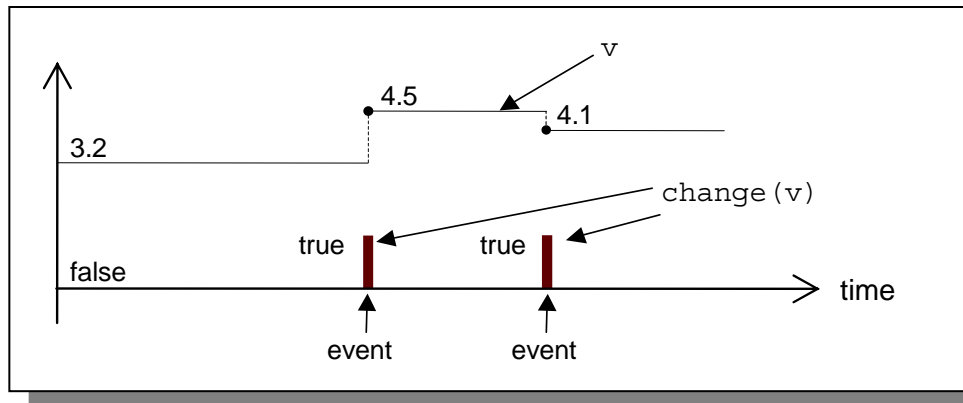
Detecting Changes using `edge()` and `change()`

Detecting changes of boolean variables using `edge()`



The expression `edge(b)` is true at events when `b` switches from false to true

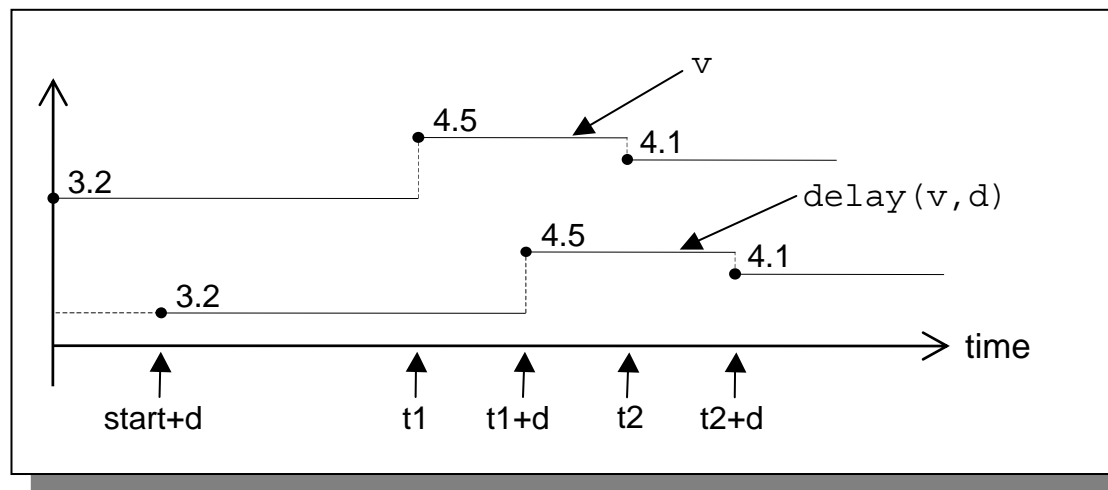
Detecting changes of discrete-time variables using `change()`



The expression `change(v)` is true at instants when `v` changes value

Creating Time-Delayed Expressions

Creating time-delayed expressions using `delay()`

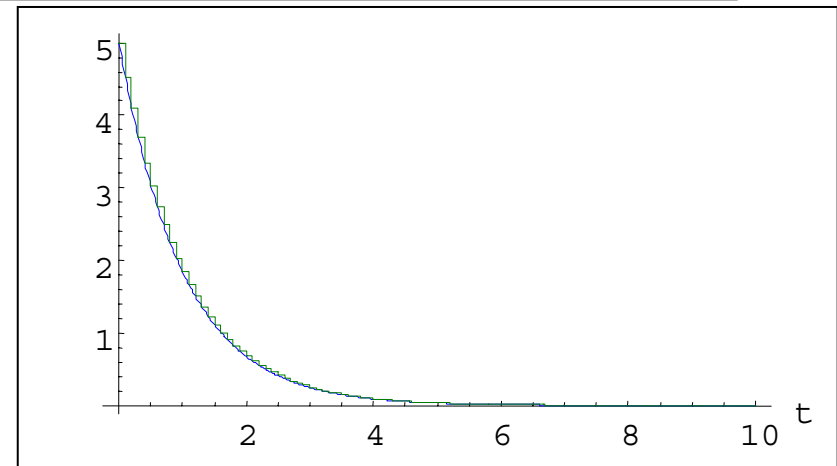


In the expression `delay(v, d)` v is delayed by a delay time d

A Sampler Model

```
model Sampler
  parameter Real sample_interval = 0.1;
  Real x(start=5);
  Real y;
equation
  der(x) = -x;
  when sample(0, sample_interval) then
    y = x;
  end when;
end Sampler;
```

```
simulate(Sampler, startTime = 0, stopTime = 10)
plot({x,y})
```

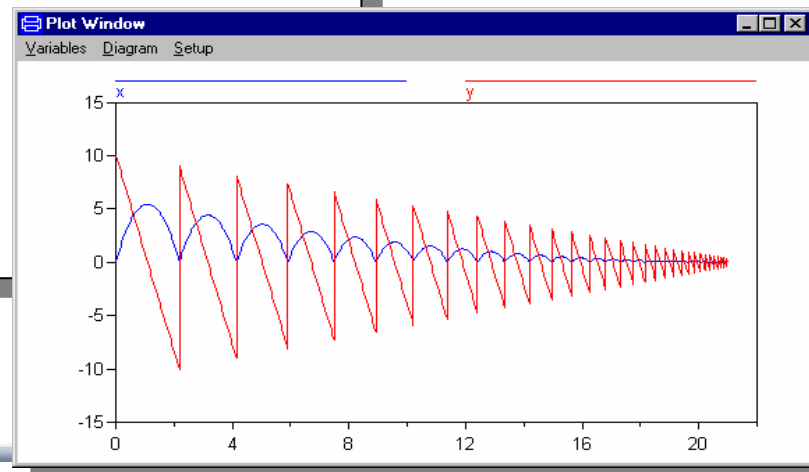
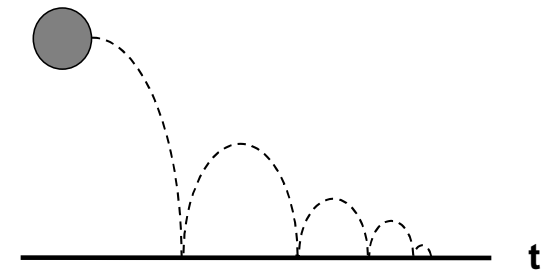


Discontinuous Changes to Variables at Events via When-Equations/Statements

The value of a *discrete-time* variable can be changed by placing the variable on the left-hand side in an equation within a when-equation, or on the left-hand side of an assignment statement in a when-statement

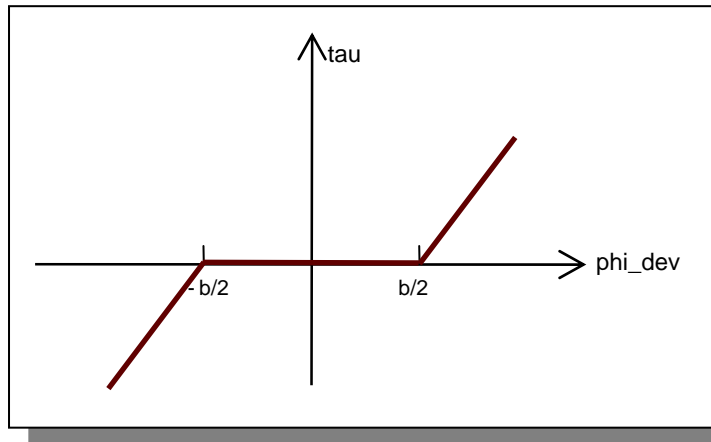
The value of a *continuous-time* state variable can be instantaneously changed by a `reinit`-equation within a when-equation

```
model BouncingBall "the bouncing ball model"
  parameter Real g=9.18; //gravitational acc.
  parameter Real c=0.90; //elasticity constant
  Real height(start=0),velocity(start=10);
equation
  der(height) = velocity;
  der(velocity)=-g;
  when height<0 then
    reinit(velocity, -c*velocity);
  end when;
end BouncingBall;
```

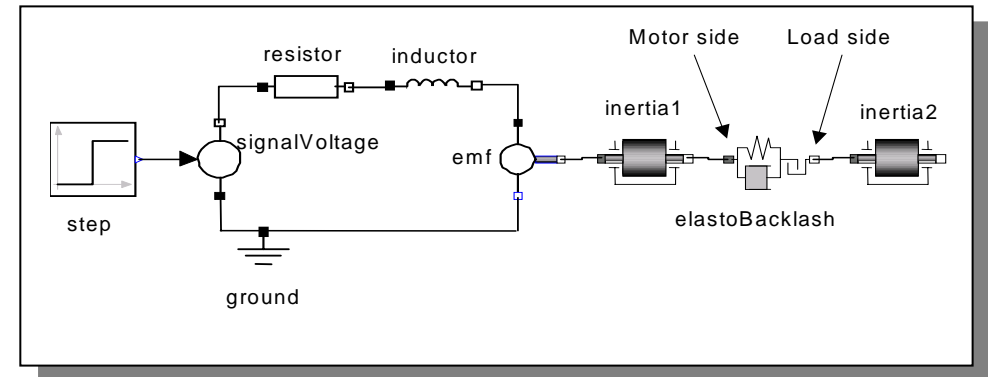


A Mode Switching Model Example

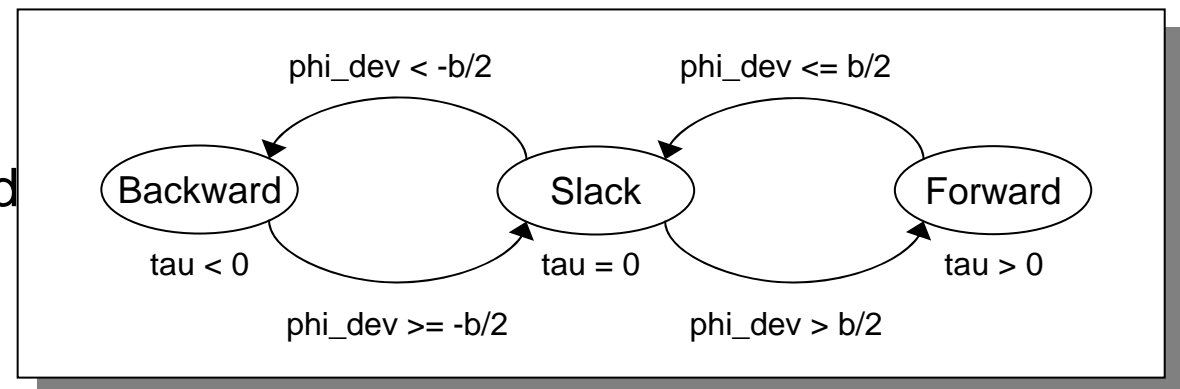
Elastic transmission with slack



DC motor transmission with elastic backlash



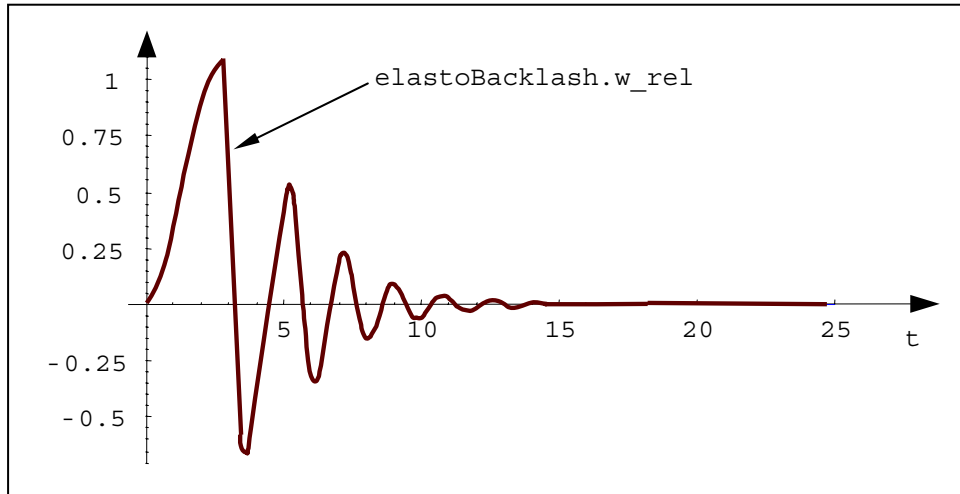
A finite state automaton
SimpleElastoBacklash model



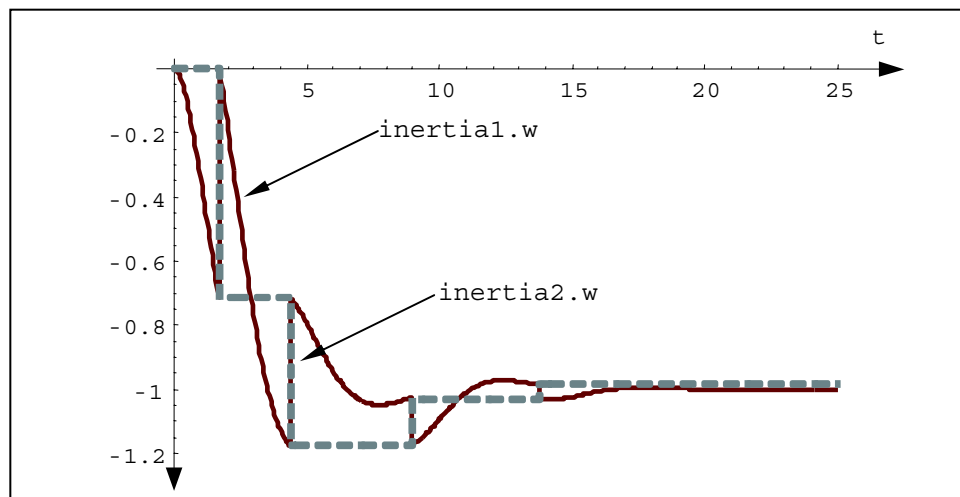
A Mode Switching Model Example cont'

```
partial model SimpleElastoBacklash
  Boolean backward, slack, forward; // Mode variables
  parameter Real b "Size of backlash region";
  parameter Real c = 1.e5 "Spring constant (c>0), N.m/rad";
  Flange_a flange_a "(left) driving flange - connector";
  Flange_b flange_b "(right) driven flange - connector";
  parameter Real phi_rel0 = 0 "Angle when spring exerts no torque";
  Real phi_rel "Relative rotation angle betw. flanges";
  Real phi_dev "Angle deviation from zero-torque pos";
  Real tau "Torque between flanges";
equation
  phi_rel = flange_b.phi - flange_a.phi;
  phi_dev = phi_rel - phi_rel0;
  backward = phi_rel < -b/2; // Backward angle gives torque tau<0
  forward = phi_rel > b/2; // Forward angle gives torque tau>0
  slack = not (backward or forward); // Slack angle gives no torque
  tau = if forward then // Forward angle gives
    c*(phi_dev - b/2) // positive driving torque
  else if backward then // Backward angle gives
    c*(phi_dev + b/2) // negative braking torque
  else // Slack gives
    0); // zero torque
end SimpleElastoBacklash
```

A Mode Switching Model Example cont'



Relative rotational speed between the flanges of the Elastobacklash transmission



We define a model with less mass in `inertia2` ($J=1$), no damping $\bar{d}=0$, and weaker string constant $c=1e-5$, to show even more dramatic backlash phenomena

The figure depicts the rotational speeds for the two flanges of the transmission with elastic backlash

Exercise 2.3 – BouncingBall

- Locate the BouncingBall model in one of the hybrid modeling sections of DrModelica (the When-Equations link in Section 2.9), run it, change it slightly, and re-run it.

