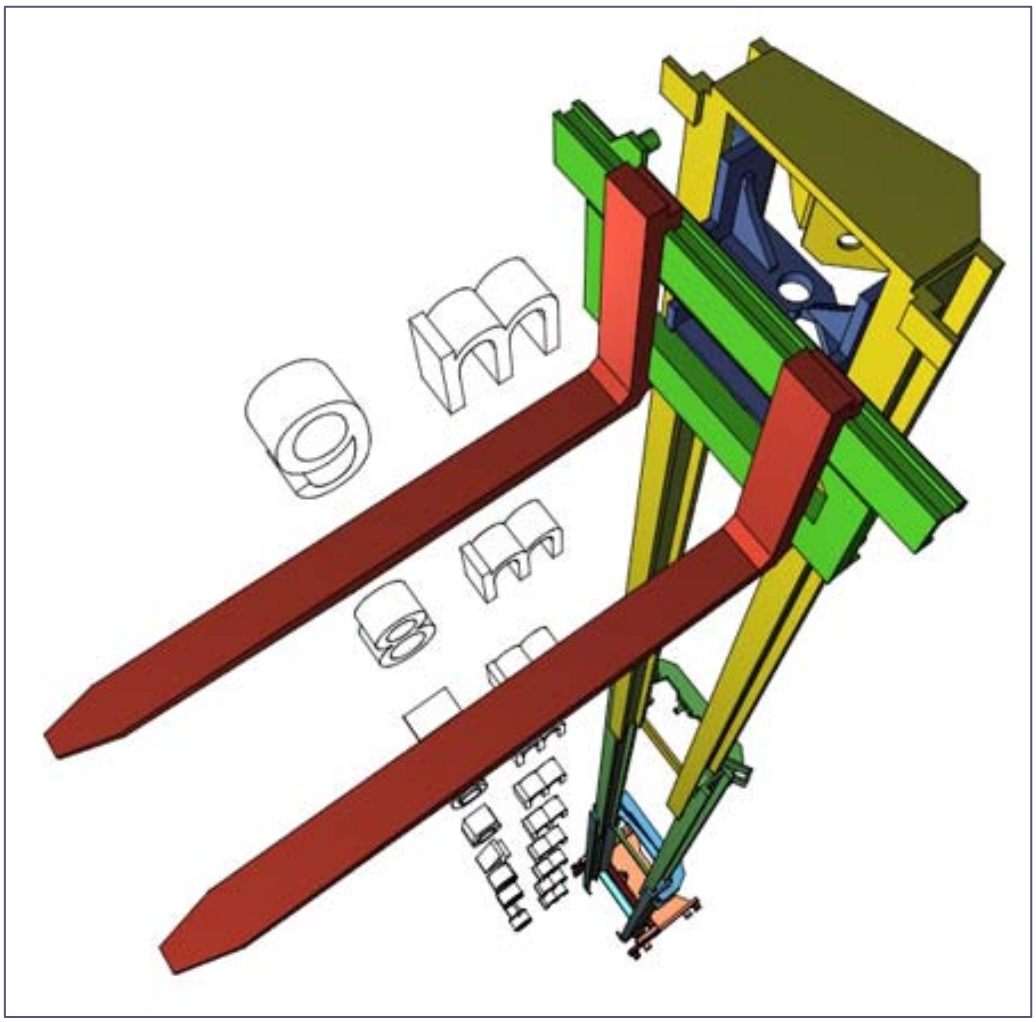
TUTORIAL – COURSE

Introduction to Object-Oriented Modeling and Simulation with Modelica Using OpenModelica

Peter Fritzson



Version 2012

Abstract

Object-Oriented modeling is a fast-growing area of modeling and simulation that provides a structured, computer-supported way of doing mathematical and equation-based modeling. Modelica is today the most promising modeling and simulation language in that it effectively unifies and generalizes previous object-oriented modeling languages and provides a sound basis for the basic concepts.

The Modelica modeling language and technology is being warmly received by the world community in modeling and simulation with major applications in virtual prototyping. It is bringing about a revolution in this area, based on its ease of use, visual design of models with combination of lego-like predefined model building blocks, its ability to define model libraries with reusable components, its support for modeling and simulation of complex applications involving parts from several application domains, and many more useful facilities. To draw an analogy, Modelica is currently in a similar phase as Java early on, before the language became well known, but for virtual prototyping instead of Internet programming.

The tutorial presents an object-oriented component-based approach to computer supported mathematical modeling and simulation through the powerful Modelica language and its associated technology. Modelica can be viewed as an almost universal approach to high level computational modeling and simulation, by being able to represent a range of application areas and providing general notation as well as powerful abstractions and efficient implementations.

The tutorial gives an introduction to the Modelica language to people who are familiar with basic programming concepts. It gives a basic introduction to the concepts of modeling and simulation, as well as the basics of object-oriented component-based modeling for the novice, and an overview of modeling and simulation in a number of application areas.

The tutorial has several goals:

- Being easily accessible for people who do not previously have a background in modeling, simulation.
- Introducing the concepts of physical modeling, object-oriented modeling and component-based modeling and simulation.
- Giving an introduction to the Modelica language.
- Demonstrating modeling examples from several application areas.
- Giving a possibility for hands-on exercises.

Presenter's data

Peter Fritzson is Professor and research director of the Programming Environment Laboratory, at Linköping University. He is also director of the Open Source Modelica Consortium, director of the MODPROD center for model-based product development, and vice chairman of the Modelica Association, organizations he helped to establish. During 1999-2007 he served as chairman of the Scandinavian Simulation Society, and secretary of the European simulation organization, EuroSim. Prof. Fritzson's current research interests are in software technology, especially programming languages, tools and environments; parallel and multi-core computing; compilers and compiler generators, high level specification and modeling languages with special emphasis on tools for object-oriented modeling and simulation where he is one of the main contributors and founders of the Modelica language. Professor Fritzson has authored or co-authored more than 250 technical publications, including 17 books/proceedings.

1. Useful Web Links

The Modelica Association Web Page

http://www.modelica .org

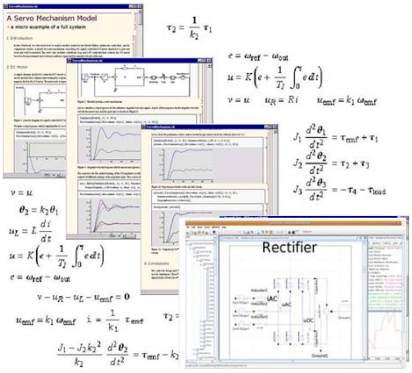
Modelica publications

http://www.modelica.org/publications.shtml

The OpenModelica open source project with download of the free OpenModelica modeling and simulation environment

http://www.openmodelica.org

Principles of Object-Oriented Modeling and Simulation with Modelica



Peter Fritzson
Linköping University, peter.fritzson@liu.se

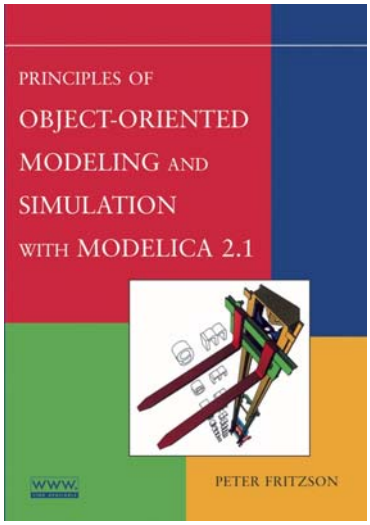
Slides
Based on book and lecture notes by Peter Fritzson
Contributions 2004-2005 by Emma Larsdotter, Peter Bunus, Peter F
Contributions 2007-2008 by Adrian Pop, Peter Fritzson
Contributions 2009 by David Broman, Jan Brugård, Mohsen Torabzadeh-Tari, Peter Fritzson
Contributions 2010 by Mohsen Torabzadeh-Tari, Peter Fritzson
Contributions 2012 by Olena Rogovchenko, Peter Fritzson

Linköpings universitet

2012-10-24 Course

MODELICA

Tutorial Based on Book, 2004
Download OpenModelica Software



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

Wiley-IEEE Press, 2004, 940 pages

- OpenModelica
  - www.openmodelica.org
- Modelica Association
  - www.modelica.org

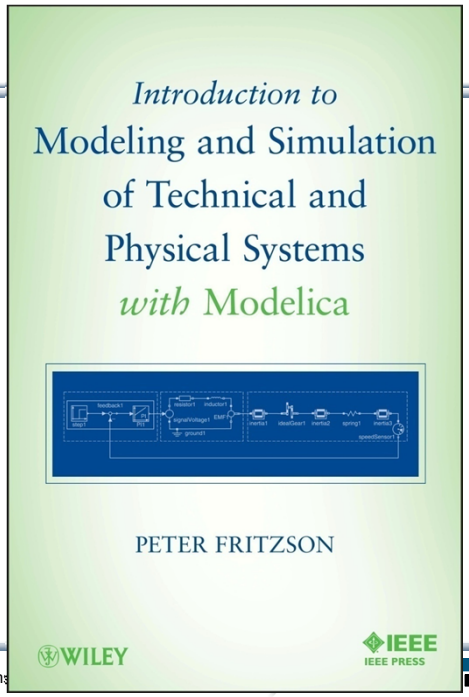
MODELICA pelab

New Introductory Modelica Book

September 2011
232 pages

Wiley
IEEE Press

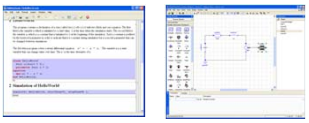
For Introductory Short Courses on Object Oriented Mathematical Modeling



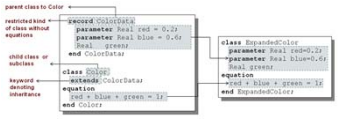
Outline Day 1

Part I
Introduction to Modelica and a demo example

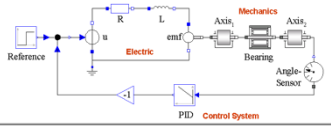
Part II
Modelica environments



Part III
Modelica language concepts and textual modeling



Part IV
Graphical modeling and the Modelica standard library



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 Larsdotter Nilsson, Peter Bunus, David Broman, Jan Brugård, Mohsen-Torabzadeh-Tari, Adeel Asghar for contributions to 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.openmodelica.org

Software Installation - Windows

- Start the software installation
- Install OpenModelica-1.9.0beta2.exe from the USB Stick

Software Installation – Linux (requires internet connection)

- Go to https://openmodelica.org/index.php/download/download-linux and follow the instructions.

Software Installation – MAC (requires internet connection)

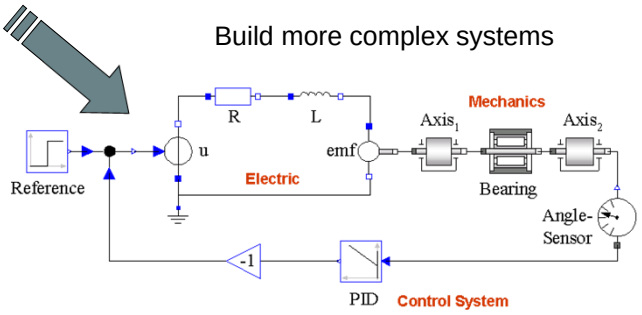
- Go to https://openmodelica.org/index.php/download/download-mac and follow the instructions or follow the instructions written below.
- The installation uses MacPorts. After setting up a MacPorts installation, run the following commands on the terminal (as root):
  - *echo rsync://build.openmodelica.org/macports/ >> /opt/local/etc/macports/sources.conf # assuming you installed into /opt/local*
  - *port selfupdate*
  - *port install openmodelica-devel*

Outline

- Introduction to Modeling and Simulation
- Modelica - The next generation modeling and Simulation Language
- Modeling and Simulation Environments and OpenModelica
- Classes
- Components, Connectors and Connections
- Equations
- Discrete Events and Hybrid Systems
- Algorithms and Functions
- Demonstrations

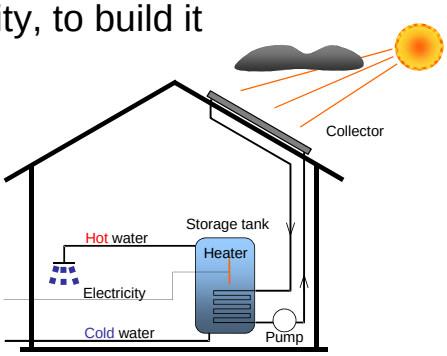
Why Modeling & Simulation ?

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



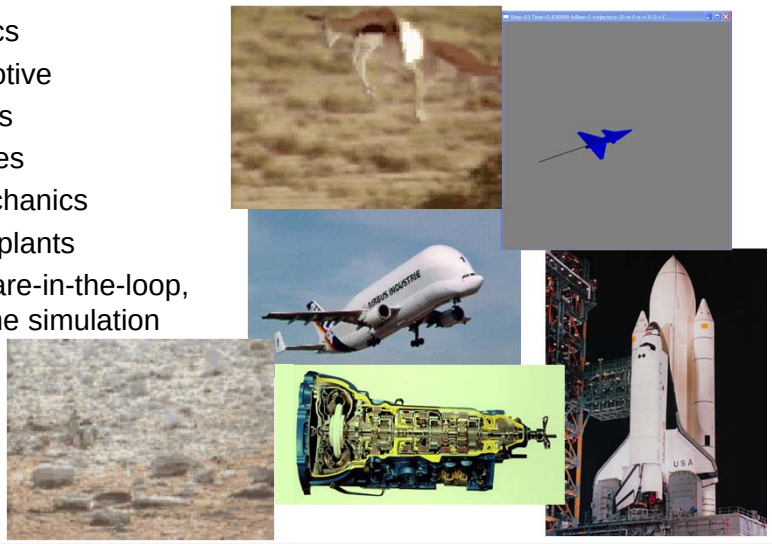
What is a system?

- A system is an object or collection of objects whose properties we want to study
- Natural and artificial systems
- Reasons to study: curiosity, to build it



Examples of Complex Systems

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



Experiments

An *experiment* is the process of extracting information from a system by exercising its inputs

Problems

- Experiment might be too *expensive*
- Experiment might be too *dangerous*
- System needed for the experiment might *not yet exist*

Model concept

A *model* of a system is anything an *experiment* can be applied to in order to answer questions about that *system*

Kinds of models:

- **Mental model** – statement like “a person is reliable”
- **Verbal model** – model expressed in words
- **Physical model** – a physical object that mimics the system
- **Mathematical model** – a description of a system where the relationships are expressed in mathematical form – a *virtual prototype*
- **Physical modeling** – also used for mathematical models built/structured in the same way as physical models

Simulation

A *simulation* is an *experiment* performed on a *model*

Examples of simulations:

- **Industrial process** – such as steel or pulp manufacturing, study the behaviour under different operating conditions in order to improve the process
- **Vehicle behaviour** – e.g. of a car or an airplane, for operator training
- **Packet switched computer network** – study behaviour under different loads to improve performance

Reasons for Simulation

- Suppression of *second-order effects*
- Experiments are too *expensive*, too *dangerous*, or the system to be investigated does *not* yet *exist*
- The *time scale* is not compatible with experimenter (Universe, million years, …)
- Variables may be *inaccessible*.
- Easy *manipulation* of models
- Suppression of *disturbances*

Dangers of Simulation

Falling in love with a model

The Pygmalion effect (forgetting that model is not the real world, e.g. introduction of foxes to hunt rabbits in Australia)

Forcing reality into the constraints of a model

The Procrustes effect (e.g. economic theories)

Forgetting the model's level of accuracy

Simplifying assumptions

Building Models Based on Knowledge

System knowledge

- The collected *general experience* in relevant domains
- The *system* itself

Specific or generic knowledge

- E.g. software engineering knowledge

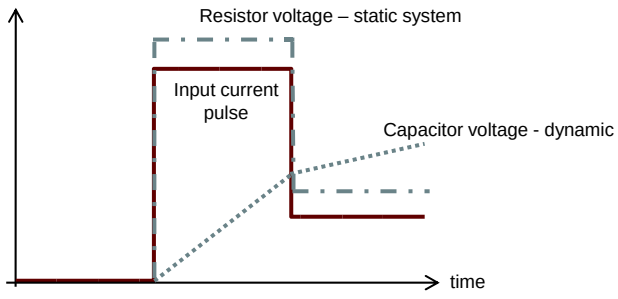
Kinds of Mathematical Models

- 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

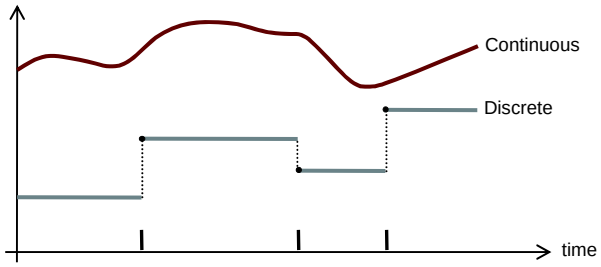
A **static** model can be defined *without* involving *time*



Continuous-Time 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

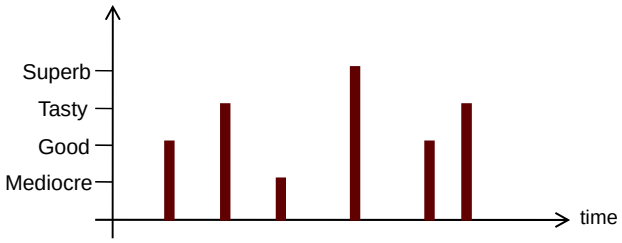


Quantitative vs. Qualitative Models

Results in qualitative data

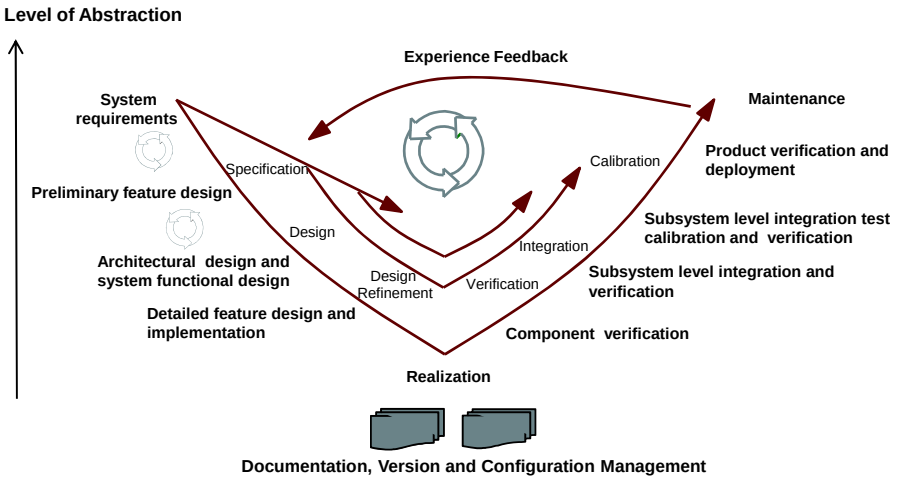
Variable values cannot be represented numerically

Mediocre = 1, Good = 2, Tasty = 3, Superb = 4



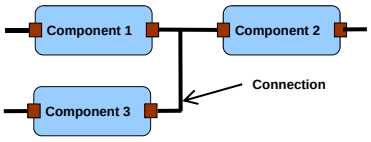
Quality of food in a restaurant according to inspections at irregular points in time

Using Modeling and Simulation within the Product Design-V

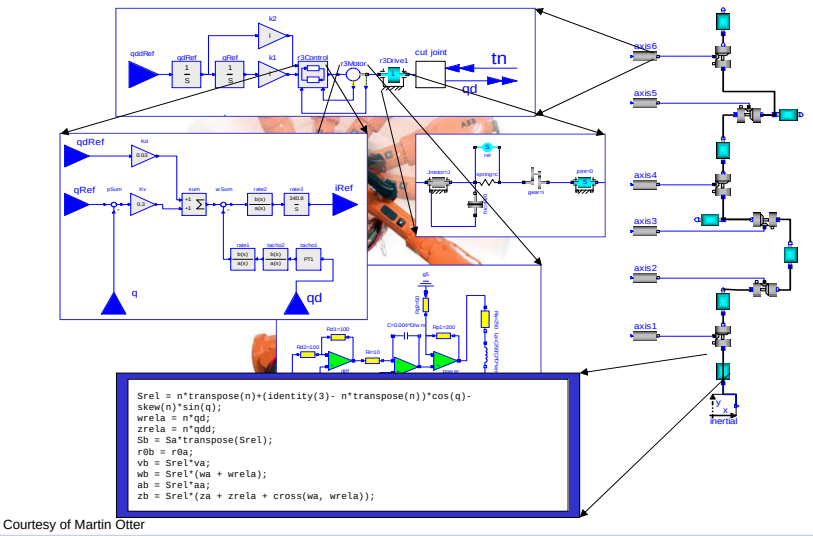


Principles of Graphical Equation-Based Modeling

- 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

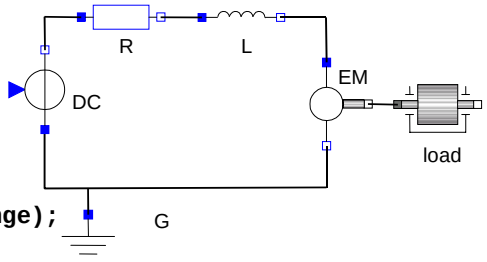


Courtesy of Martin Otter

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
DC.p.v == R.n.v

0 == R.p.i + L.n.i
R.p.v == L.n.v

0 == L.p.i + EM.n.i
L.p.v == EM.n.v

0 == EM.p.i + DC.n.i
EM.p.v == DC.n.v

0 == DC.n.i + G.p.i
DC.n.v == G.p.v
```

```
EM.u == EM.p.v - EM.n.v
0 == EM.p.i + EM.n.i
EM.i == EM.p.i
EM.u == EM.k * EM.ω
EM.i == EM.M / EM.k
EM.J * EM.ω == EM.M - EM.b * EM.ω

DC.u == DC.p.v - DC.n.v
0 == DC.p.i + DC.n.i
DC.i == DC.p.i
DC.u == DC.Amp * Sin[2 π DC.f * t]
```

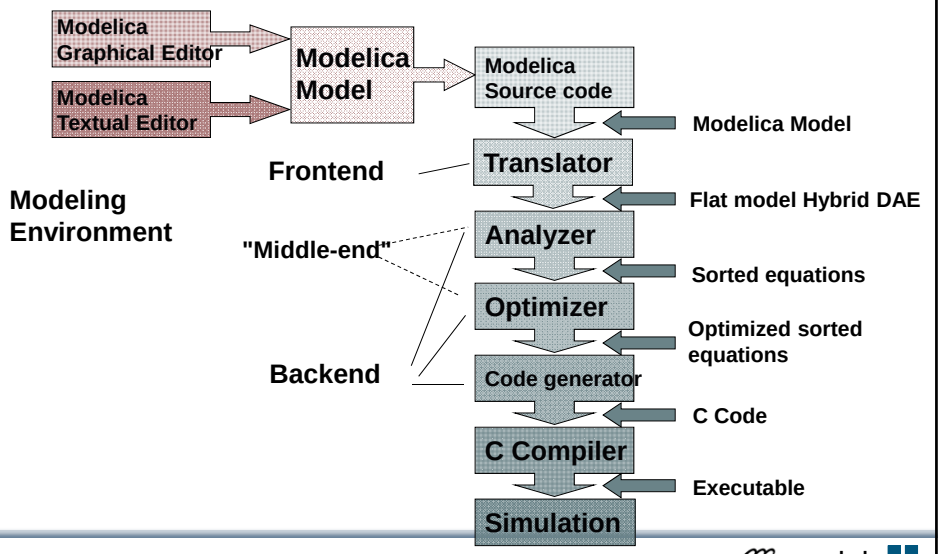
```
R.u == R.p.v - R.n.v
0 == R.p.i + R.n.i
R.i == R.p.i
R.u == R.R * R.i

L.u == L.p.v - L.n.v
0 == L.p.i + L.n.i
L.i == L.p.i
L.u == L.L * L.i'
```

(`load` component not included)

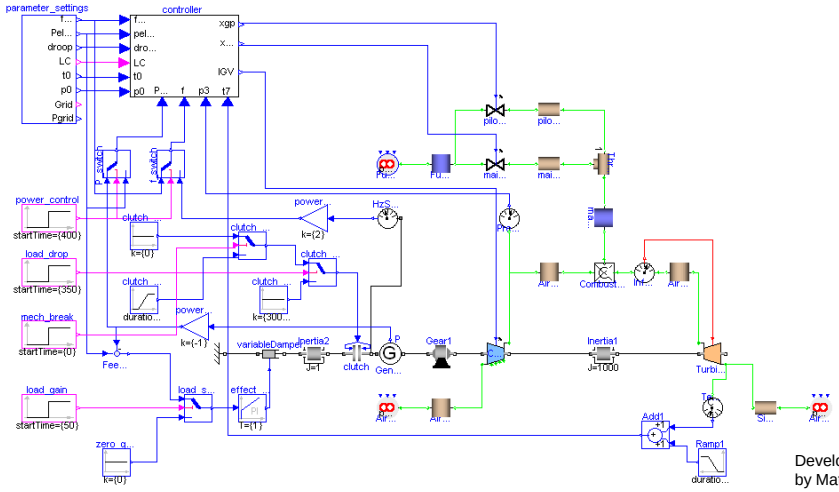
Automatic transformation to ODE or DAE for simulation:

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

Model Translation Process to Hybrid DAE to Code



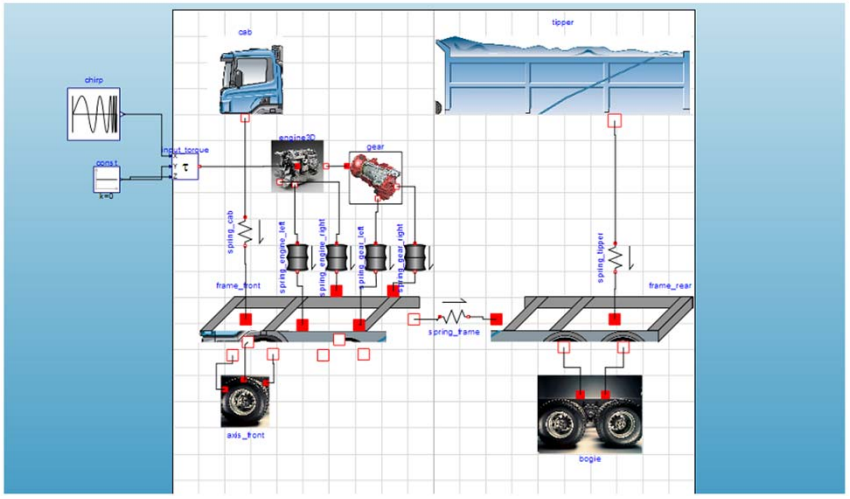
GTX Gas Turbine Power Cutoff Mechanism



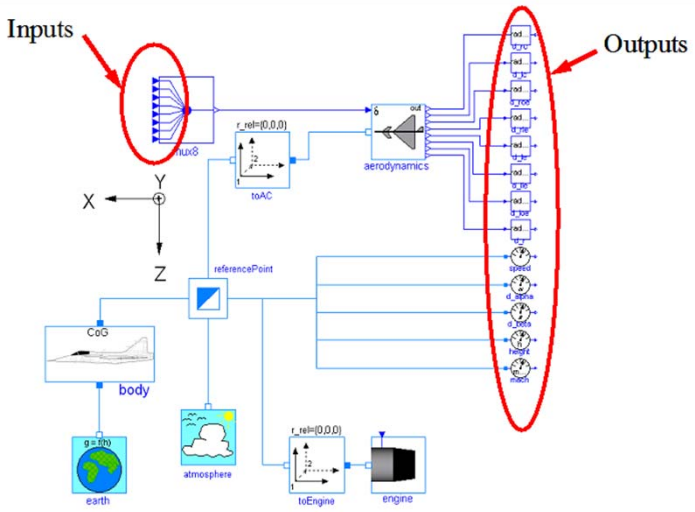
Courtesy of Siemens Industrial Turbomachinery AB

Developed by MathCore for Siemens

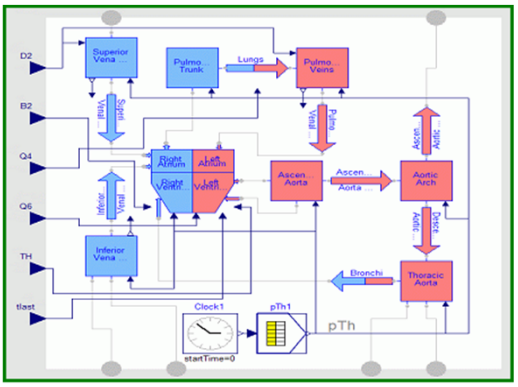
Modelica in Automotive Industry



Modelica in Avionics

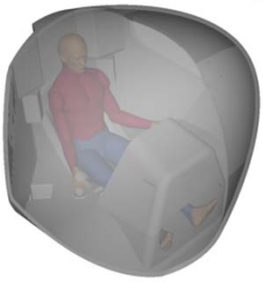


Modelica in Biomechanics



Application of Modelica in Robotics Models
Real-time Training Simulator for Flight, Driving

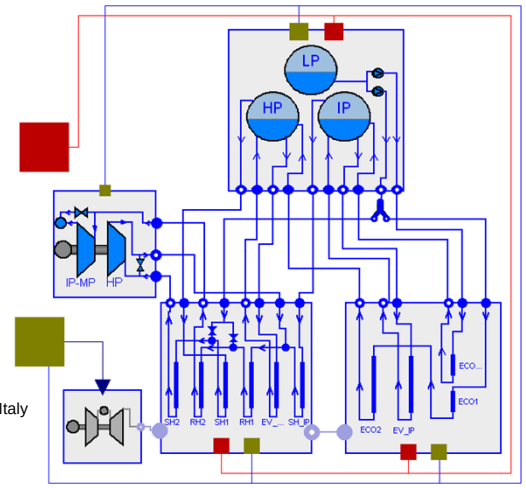
- Using Modelica models generating real-time code
- Different simulation environments (e.g. Flight, Car Driving, Helicopter)
- Developed at DLR Munich, Germany
- Dymola Modelica tool



Courtesy of Martin Otter, DLR, Oberphaffenhofen, Germany

Combined-Cycle Power Plant
Plant model – system level

- GT unit, ST unit, Drum boilers unit and HRSG units, connected by thermo-fluid ports and by signal buses
- Low-temperature parts (condenser, feedwater system, LP circuits) are represented by trivial boundary conditions.
- GT model: simple law relating the electrical load request with the exhaust gas temperature and flow rate.

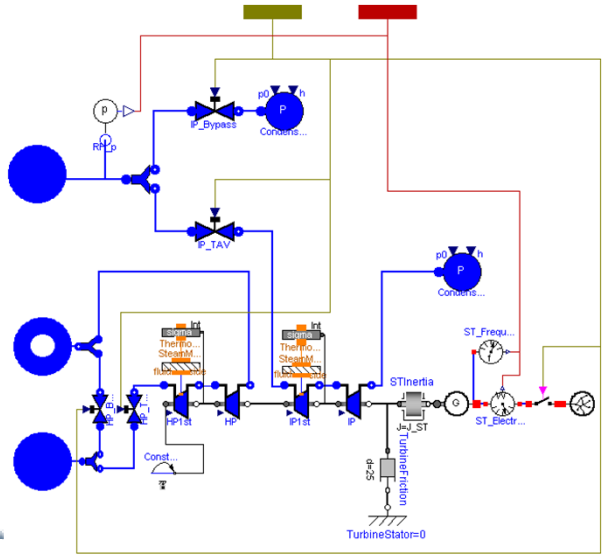


Courtesy Francesco Casella, Politecnico di Milano – Italy and Francesco Pretolani, CESI SpA - Italy

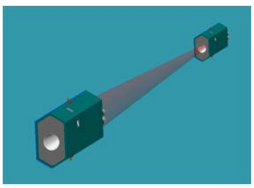
Combined-Cycle Power Plant
Steam turbine unit

- Detailed model in order to represent all the start-up phases
- Bypass paths for both the HP and IP turbines allow to start up the boilers with idle turbines, dumping the steam to the condenser
- HP and IP steam turbines are modelled, including inertia, electrical generator, and connection to the grid

Courtesy Francesco Casella, Politecnico di Milano – Italy
and Francesco Pretolani, CESI SpA - Italy

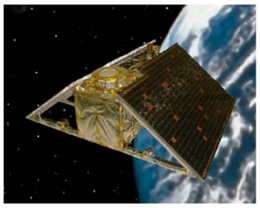


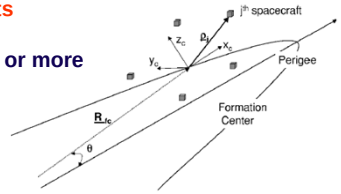
Modelica Spacecraft Dynamics Library



Formation flying on elliptical orbits

Control the relative motion of two or more spacecraft





Attitude control for satellites using magnetic coils as actuators

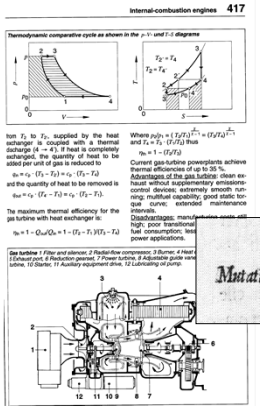
Torque generation mechanism: interaction between coils and geomagnetic field

Courtesy of Francesco Casella, Politecnico di Milano, Italy

Modelica – The Next Generation Modeling Language

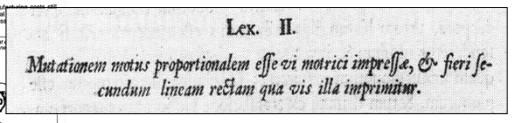
Stored Knowledge

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


Internal-combustion engines 417

Thermodynamic comparative cycle as shown in the p-V- und T-S diagrams

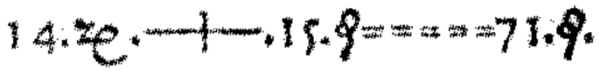
"The change of motion is proportional to the motive force impressed"
– Newton


Lex. II.

Mutationem motus proportionalem esse vi motrici impressæ, & 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 = INTEG(F)/m
```

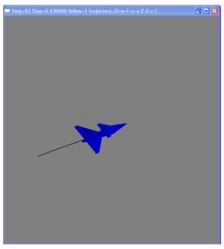
Programming languages usually do not allow equations!

What is Modelica?

A language for modeling of complex physical systems

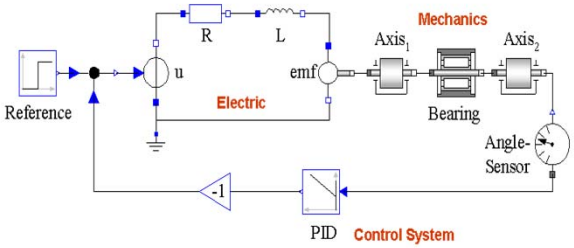
- Robotics
- Automotive
- Aircrafts
- Satellites
- Power plants
- Systems biology

MODELICA



What is Modelica?

A language for modeling of complex physical systems



Primary designed for **simulation**, but there are also other usages of models, e.g. optimization.

What is Modelica?

A language for modeling of complex cyber physical systems

i.e., Modelica is **not** a tool

Free, open language specification:

Available at: www.modelica.org

Developed and standardized by Modelica Association

There exist several free and commercial tools, for example:

- **OpenModelica from OSMC**
- Dymola from Dassault systems
- Wolfram System Modeler from Wolfram
- SimulationX from ITI
- MapleSim from MapleSoft
- AMESIM from LMS
- Jmodelica.org from Modelon
- MWORKS from Tongyang Sw & Control
- IDA Simulation Env, from Equa
- CyDesign Modeling tool, CyDesign Labs

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

Modelica – The Next Generation Modeling Language

High level language

MATLAB-style array operations; Functional style; iterators, constructors, object orientation, equations, etc.

MATLAB similarities

MATLAB-like array and scalar arithmetic, but strongly typed and efficiency comparable to C.

Non-Proprietary

- Open Language Standard
- Both Open-Source and Commercial implementations

Flexible and powerful external function facility

- LAPACK interface effort started

Modelica Language Properties

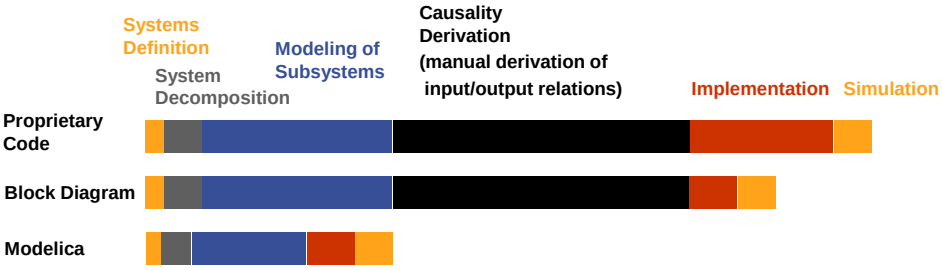
- **Declarative** and **Object-Oriented**
- **Equation-based**; continuous and discrete equations
- **Parallel** process modeling of real-time applications, according to synchronous data flow principle
- **Functions** with algorithms without global side-effects (but local data updates allowed)
- **Type system** inspired by Abadi/Cardelli
- **Everything is a class** – Real, Integer, models, functions, packages, parameterized classes....

Object Oriented Mathematical Modeling with Modelica

- 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 – Faster Development, Lower Maintenance than with Traditional Tools

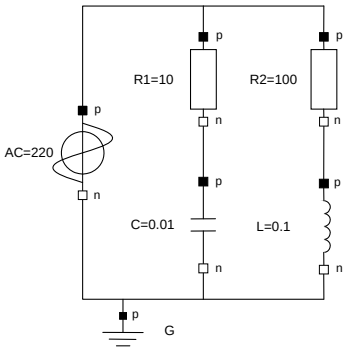
Block Diagram (e.g. Simulink, ...) or
Proprietary Code (e.g. Ada, Fortran, C,...)
vs Modelica



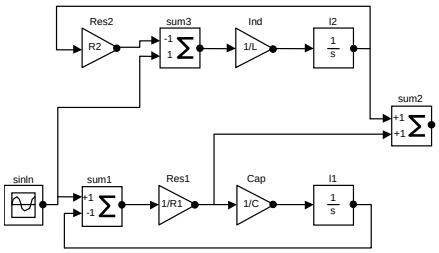
Modelica vs Simulink Block Oriented Modeling Simple Electrical Model

Modelica:
Physical model – easy to understand

Keeps the physical structure



Simulink:
Signal-flow model – hard to understand



Brief Modelica History

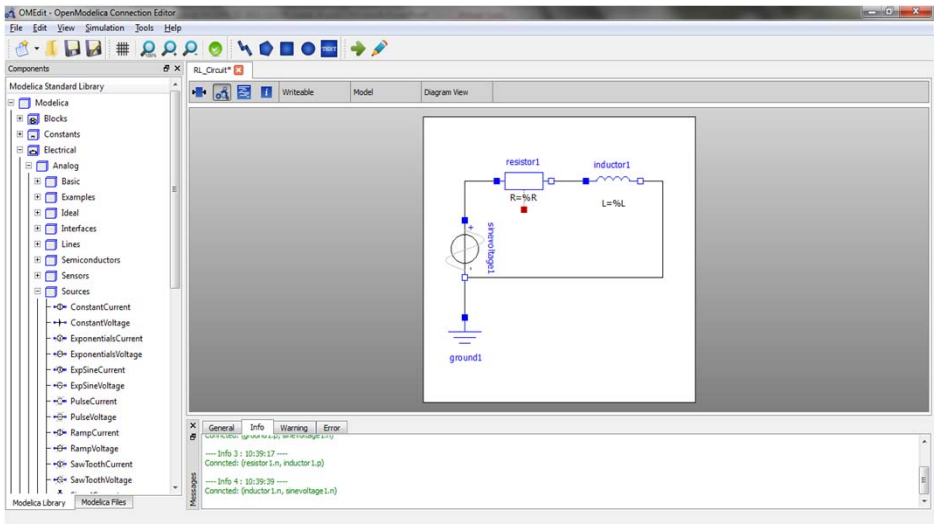
- First Modelica design group meeting in fall 1996
  - International group of people with expert knowledge in both language design and physical modeling
  - Industry and academia
- Modelica Versions
  - 1.0 released September 1997
  - 2.0 released March 2002
  - 2.2 released March 2005
  - 3.0 released September 2007
  - 3.1 released May 2009
  - 3.2 released March 2010
  - 3.3 released May 2012
- Modelica Association established 2000 in Linköping
  - 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
- The 7th International Modelica conference Sept 21-22, 2009 in Como, Italy
- The 8th International Modelica conference March 20-22, 2011 in Dresden, Germany
- The 9th International Modelica conference Sept 3-5, 2012 in Munich, Germany

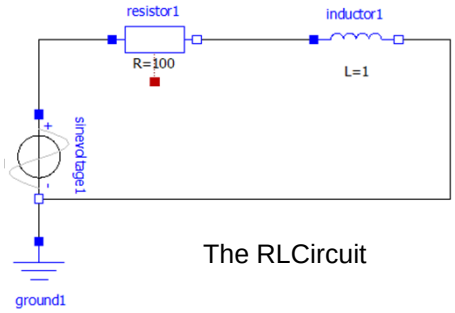
Exercises Part I
Hands-on graphical modeling
(20 minutes)

Graphical Modeling - Using Drag and Drop Composition

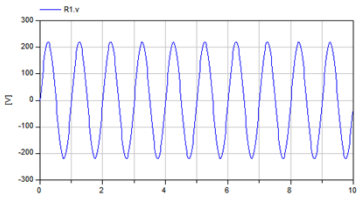


Exercises Part I – Basic Graphical Modeling

- (*See instructions on next two pages*)
- Start the OMEdit editor (part of OpenModelica)
- Draw the RLCircuit
- Simulate



The RLCircuit



Simulation

Exercises Part I – OMEdit Instructions (Part I)

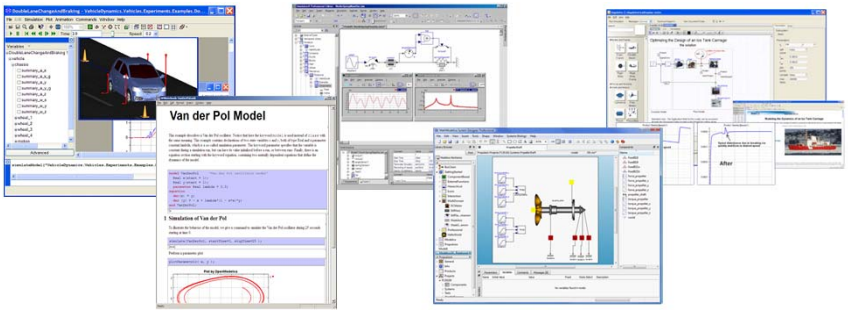
- Start OMEdit from the Program menu under OpenModelica
- Go to **File** menu and choose **New**, and then select **Model**.
- E.g. write *RLCircuit* as the model name.
- For more information on how to use OMEdit, go to **Help** and choose **User Manual** or press **F1**.

• Under the **Modelica Library:**
  • Contains The standard Modelica library components
  • The **Modelica files** contains the list of models you have created.

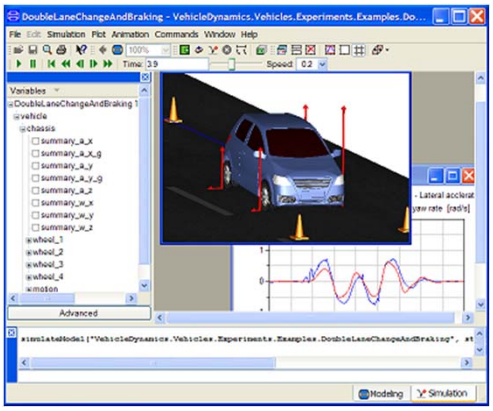
Exercises Part I – OMEdit Instructions (Part II)

- For the RLCircuit model, browse the Modelica standard library and add the following component models:
  - Add `Ground`, `Inductor` and `Resistor` component models from Modelica.Electrical.Analog.Basic package.
  - Add SineVolagte component model from Modelica.Electrical.Analog.Sources package.
- Make the corresponding connections between the component models as shown in previous slide on the RLCiruit.
- Simulate the model
  - Go to Simulation menu and choose simulate or click on the siumulate button in the toolbar. 
- Plot the instance variables
  - Once the simulation is completed, a plot variables list will appear on the right side. Select the variable that you want to plot.

Modelica Environments and OpenModelica

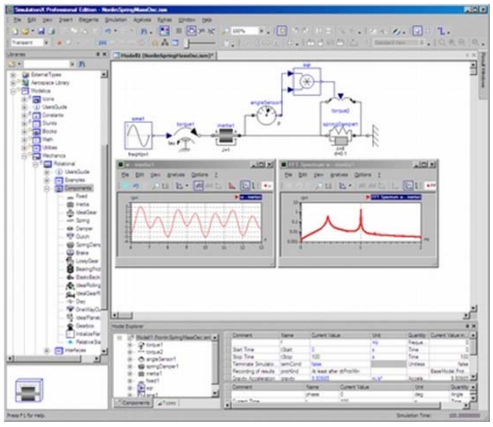


Dymola



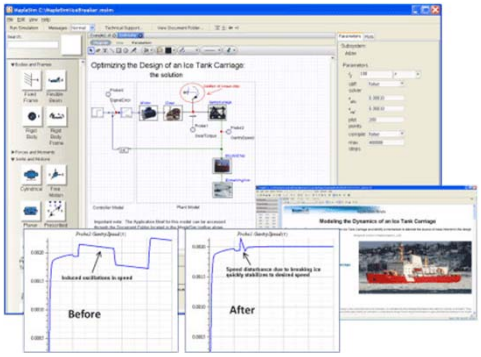
- Dynasim (Dassault Systemes)
- Sweden
- First Modelica tool on the market
- Main focus on automotive industry
- www.dynasim.com

Simulation X



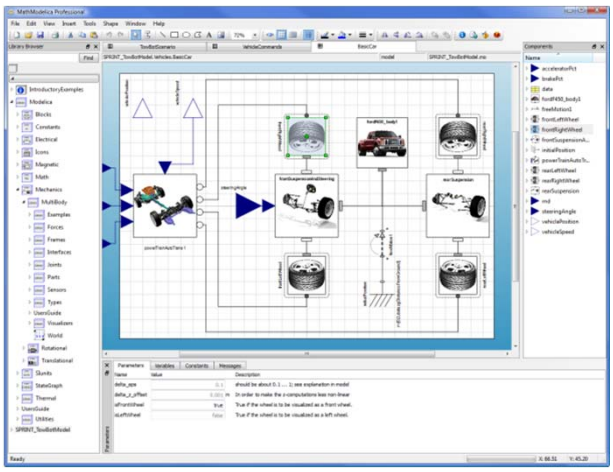
- ITI
- Germany
- Mechatronic systems
- www.simulationx.com

MapleSim



- Maplesoft
- Canada
- Recent Modelica tool on the market
- Integrated with Maple
- www.maplesoft.com

Wolfram System Modeler – MathCore / Wolfram Research

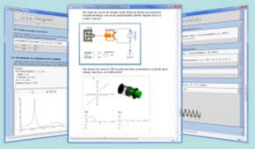


Car model graphical view

Courtesy Wolfram Research

- Wolfram Research
- USA, Sweden
- General purpose
- Mathematica integration
- www.wolfram.com
- www.mathcore.com

Mathematica



Simulation and analysis

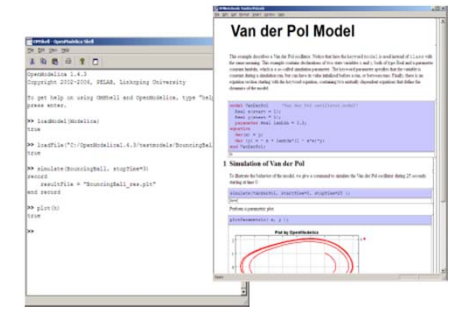
MODELICA

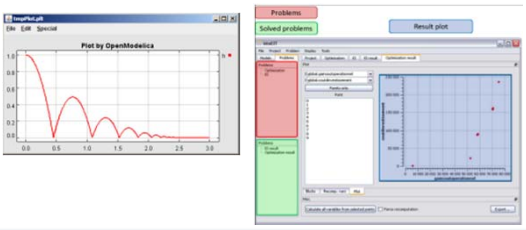
The OpenModelica Environment

www.OpenModelica.org

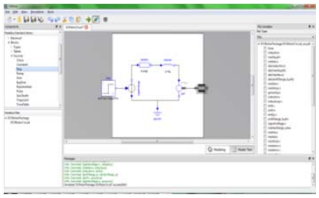
MODELICA

OpenModelica (Part I)





- OpenModelica
- Open Source Modelica Consortium (OSMC)
- Sweden and other countries
- Open source
- www.openmodelica.org

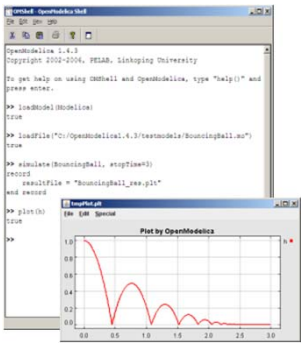


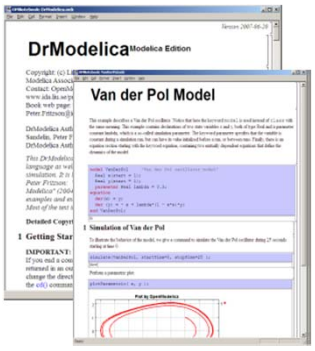
- OMEdit, graphical editor
- OMOptim, optimization subsystem

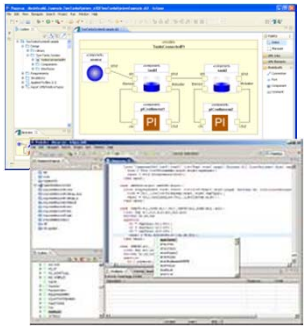
OpenModelica (Part II)

- Advanced Interactive Modelica compiler (OMC)
  - Supports most of the Modelica Language
- Basic environment for creating models
  - **OMShell** – an interactive command handler
  - **OMNotebook** – a literate programming notebook
  - **MDT** – an advanced textual environment in Eclipse

- **ModelicaML** UML Profile
- **MetaModelica** extension
- **ParModelica** extension
- **Modelica** & **Python** scripting







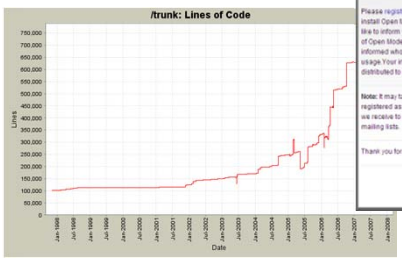
OSMC – Open Source Modelica Consortium
45 organizational members October 2012

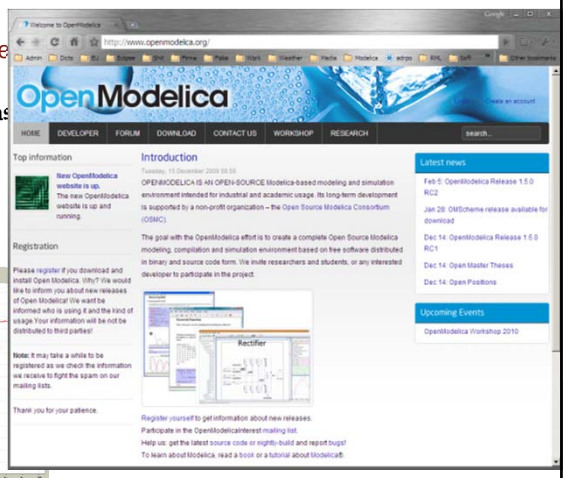
Founded Dec 4, 2007

Open-source community service

- **Website and Support Forum**
- **Version-controlled source bas**
- **Bug database**
- **Development courses**
- **www.openmodelica.org**

Code Statistics





OSMC 45 Organizational Members, October 2012 (initially 7 members, 2007)

Companies and Institutes (24 members)

- **ABB Corporate Research, Sweden**
- **Bosch Rexroth AG, Germany**
- **Siemens PLM, California, USA**
- **Siemens Turbo Machinery AB, Sweden**
- **CDAC Centre for Advanced Compu, Kerala, India**
- **Creative Connections, Prague, Czech Republic**
- **DHI, Aarhus, Denmark**
- **Evonik, Dehli, India**
- **Equa Simulation AB, Sweden**
- **Fraunhofer FIRST, Berlin, Germany**
- **Frontway AB, Sweden**
- **Gamma Technology Inc, USA**
- **IFP, Paris, France**
- **InterCAX, Atlanta, USA**
- **ISID Dentsu, Tokyo, Japan**
- **ITI, Dresden, Germany**
- **MathCore Engineering/ Wolfram, Sweden**
- **Maplesoft, Canada**
- **TLK Thermo, Germany**
- **Sozhou Tongyuan Software and Control, China**
- **Vl-grade, Italy**
- **VTI, Linköping, Sweden**
- **VTT, Finland**
- **XRG Simulation, Germany**

Universities (21 members)

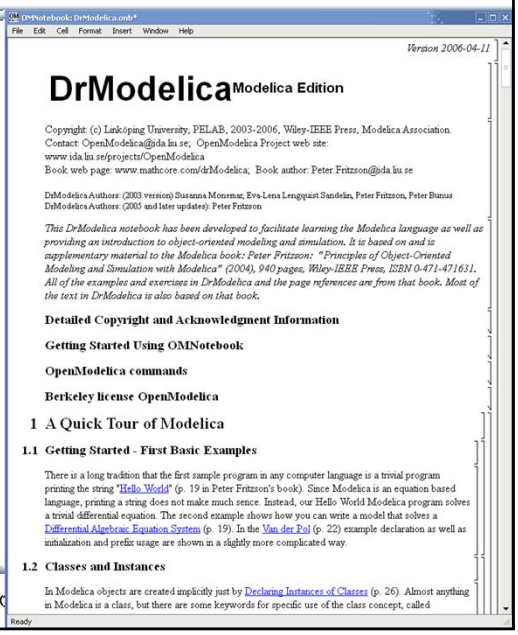
- **TU Berlin, Inst. UEBB, Germany**
- **FH Bielefeld, Bielefeld, Germany**
- **TU Braunschweig, Germany**
- **University of Calabria, Italy**
- **TU Dortmund, Germany**
- **TU Dresden, Germany**
- **Georgia Institute of Technology, USA**
- **Ghent University, Belgium**
- **Griffith University, Australia**
- **TU Hamburg/Harburg Germany**
- **KTH, Stockholm, Sweden**
- **Université Laval, Canada**
- **Linköping University, Sweden**
- **Univ of Maryland, Syst Eng USA**
- **Univ of Maryland, CEEE, USA**
- **Politecnico di Milano, Italy**
- **Ecoles des Mines, CEP, France**
- **Mälardalen University, Sweden**
- **Univ Pisa, Italy**
- **Telemark Univ College, Norway**
- **University of Ålesund, Norwlay**

OMNotebook Electronic Notebook with DrModelica

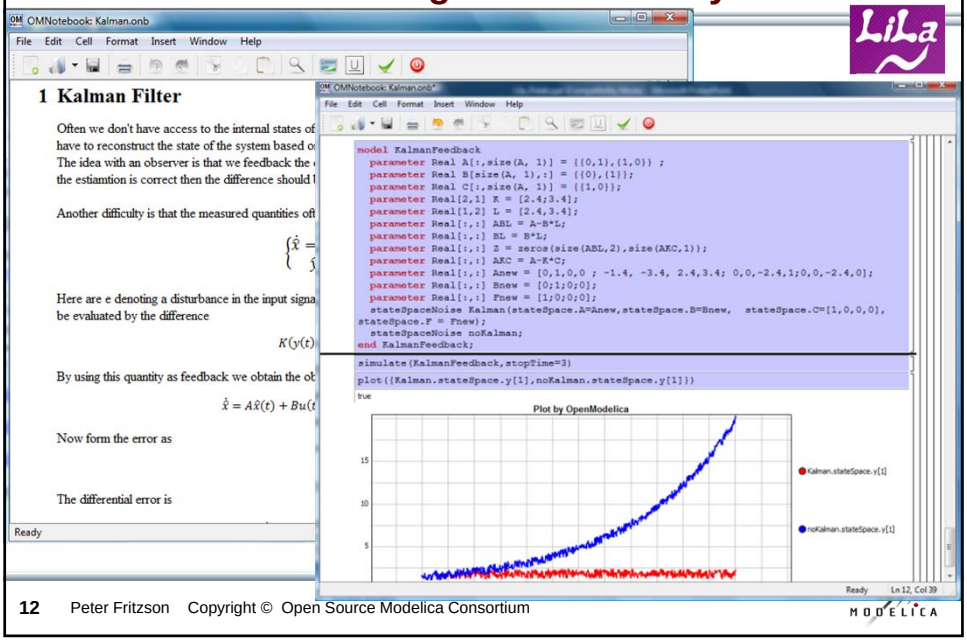
- Primarily for teaching
- Interactive electronic book
- Platform independent

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)



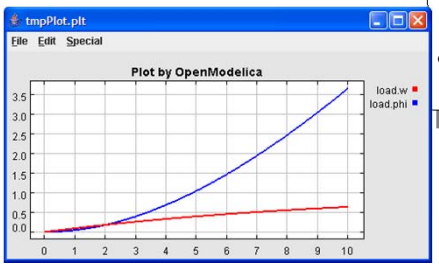
OMnotebook Interactive Electronic Notebook Here Used for Teaching Control Theory



Interactive Session Handler – on dcmotor Example
(Session handler called 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 i1;
  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,i1.p);
  connect(i1.n,emf1.p);
  connect(emf1.n,g.p);
  connect(emf1.flange_b,load.flange_a);
end dcmotor;
```

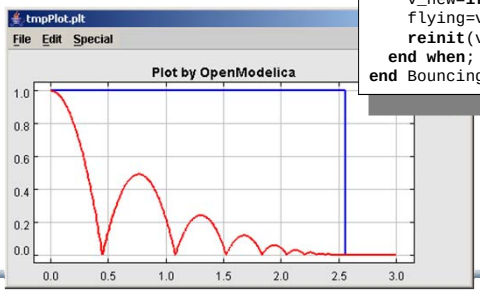


MODELICA

Event Handling by OpenModelica – BouncingBall

```
>>simulate(BouncingBall,
           stopTime=3.0);
>>plot({h,flying});
```

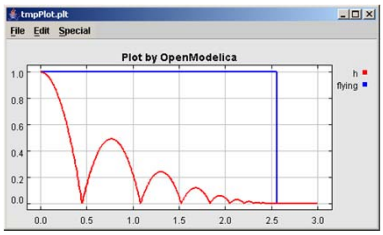
```
model BouncingBall
  parameter Real e=0.7 "coefficient of restitution";
  parameter Real g=9.81 "gravity acceleration";
  Real h(start=1) "height of ball";
  Real v "velocity of ball";
  Boolean flying(start=true) "true, if ball is flying";
  Boolean impact;
  Real v_new;
equation
  impact=h <= 0.0;
  der(v)=if flying then -g else 0;
  der(h)=v;
  when {h <= 0.0 and v <= 0.0,impact} then
    v_new=if edge(impact) then -e*pre(v) else 0;
    flying=v_new > 0;
    reinit(v, v_new);
  end when;
end BouncingBall;
```



MODELICA

Run Scripts in OpenModelica

- RunScript command interprets a .mos file
- .mos means MOdelica Script file
- Example:
  >> **`runScript("sim_BouncingBall.mos")`**



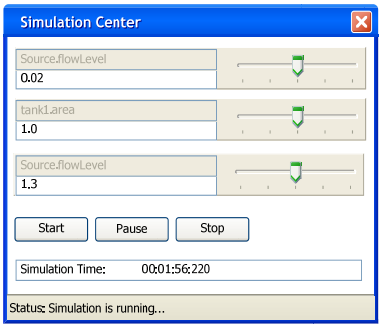
The file sim_BouncingBall.mos :

```
loadFile("BouncingBall.mo");
simulate(BouncingBall, stopTime=3.0);
plot({h,flying});
```

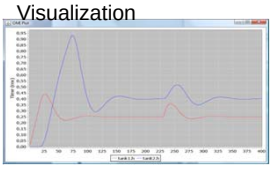

MODELICA

Interactive Simulation with OpenModelica 1.8.0

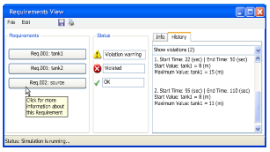
Simulation Control



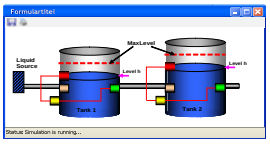
Examples of Simulation Visualization



Plot View



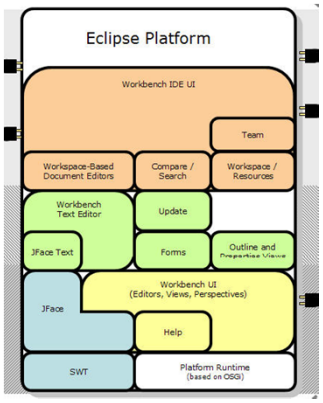
Requirements Evaluation View in ModelicaML



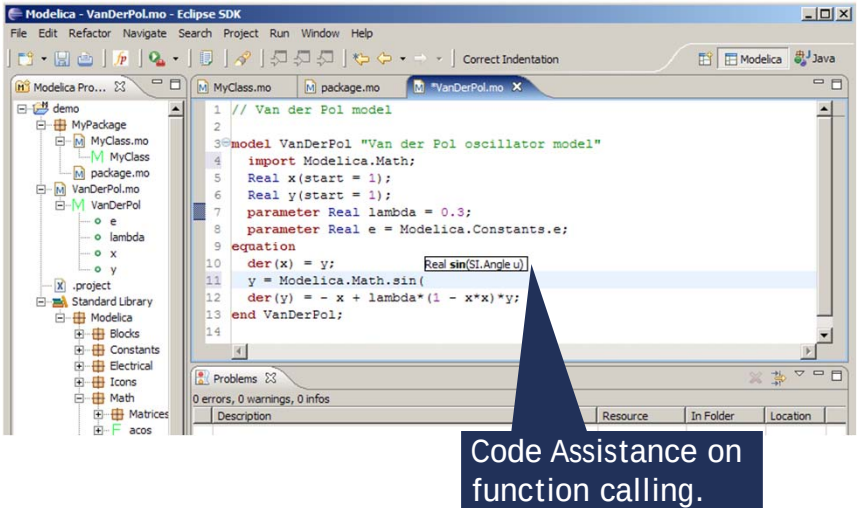
Domain-Specific Visualization View

OpenModelica MDT – Eclipse Plugin

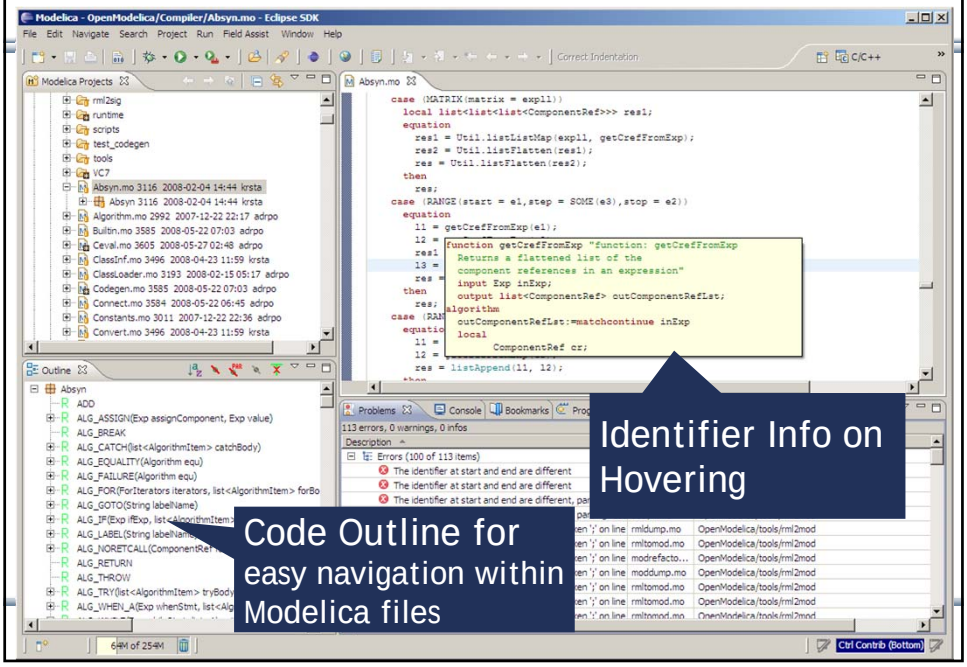
- Browsing of packages, classes, functions
- Automatic building of executables; separate compilation
- Syntax highlighting
- Code completion, Code query support for developers
- Automatic Indentation
- Debugger (Prel. version for algorithmic subset)



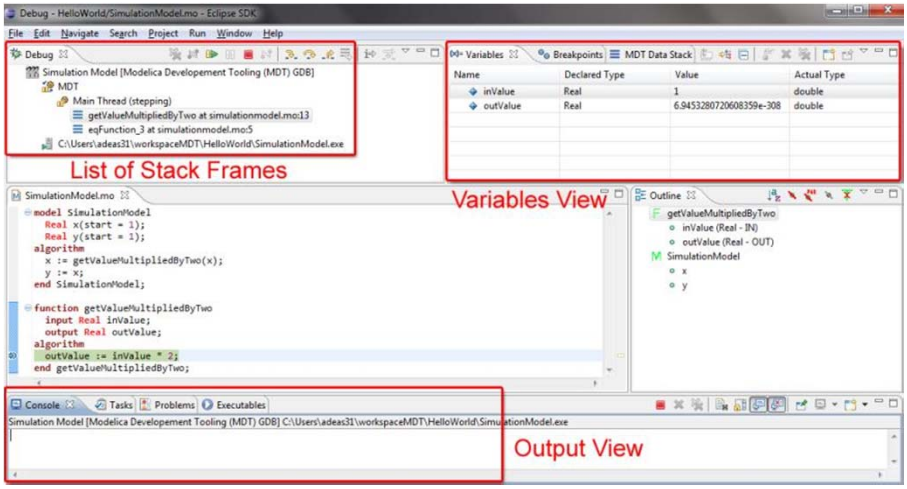
OpenModelica MDT – Usage Example



OpenModelica MDT: Code Outline and Hovering Info

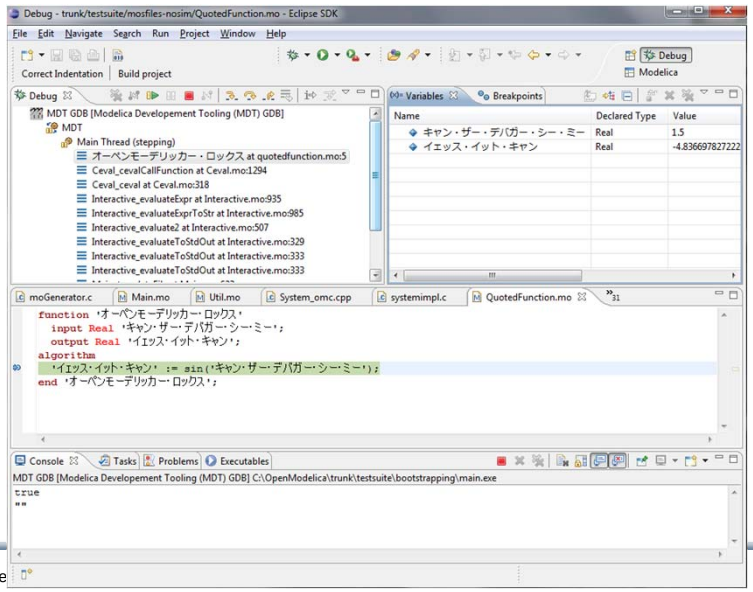


OpenModelica MDT Debugger



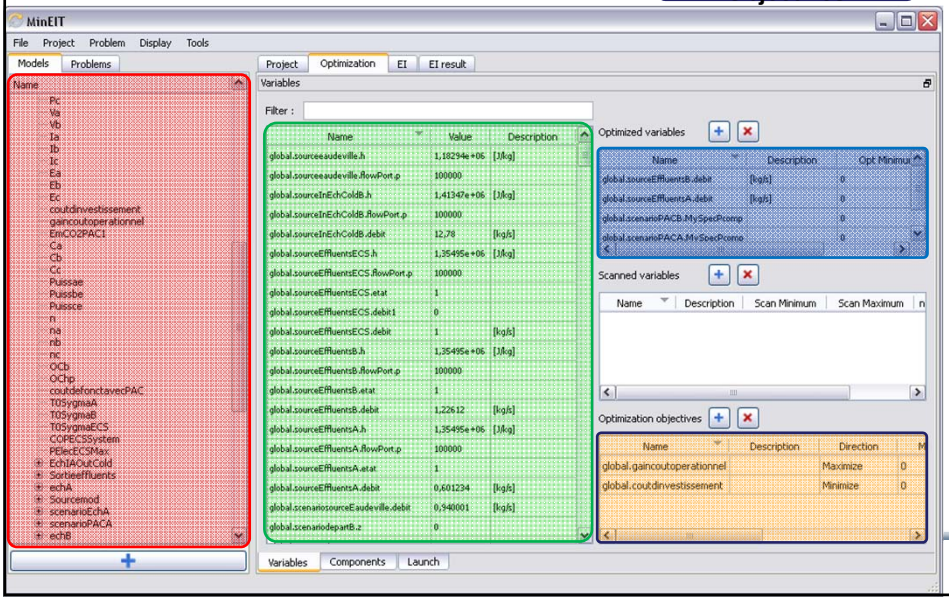
The OpenModelica MDT Debugger (Eclipse-based)

Using Japanese Characters



OMOptim – Optimization (1)

Model structure | **Model Variables** | **Optimized parameters** | **Optimized Objectives**



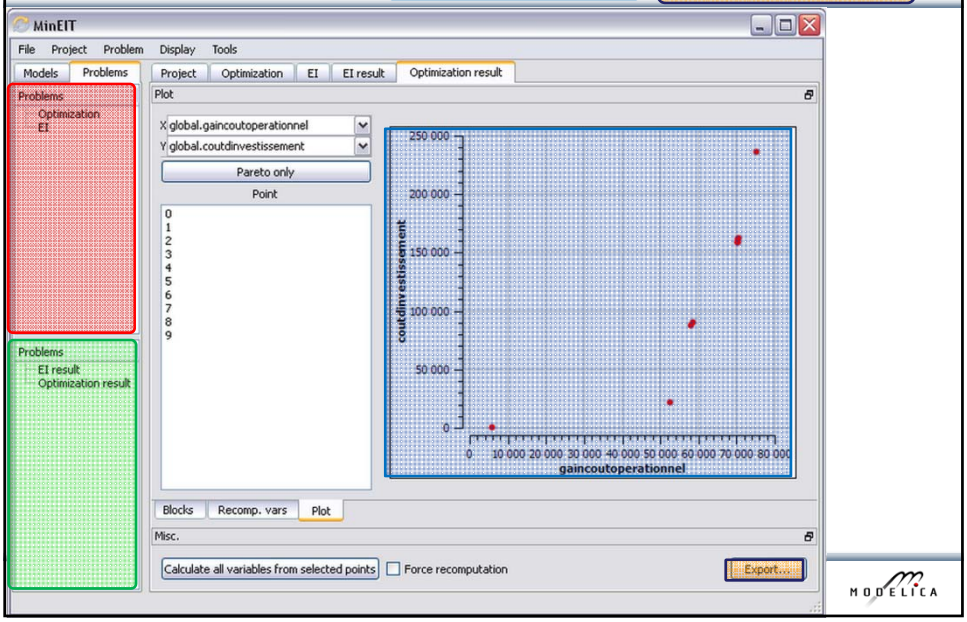
OMOptim – Optimization (2)

Problems

Solved problems

Result plot

Export result data .csv

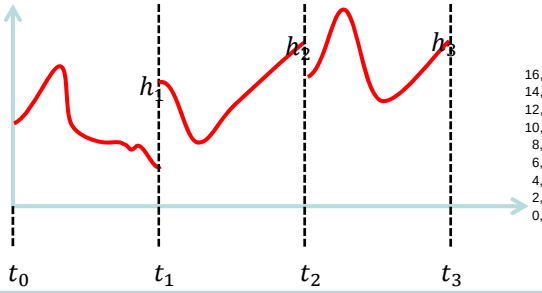


Parallel Multiple-Shooting and Collocation Dynamic Trajectory Optimization

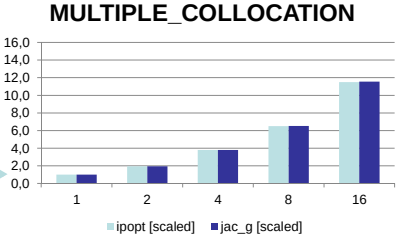
- Minimize a goal function subject to model equation constraints, useful e.g. for NMPC
- Multiple Shooting/Collocation
  - Solve sub-problem in each sub-interval

Paper in Modelica'2012 Conf. Prototype, not yet in OpenModelica release. Planned release spring 2013.

$$x_i(t_{i+1}) = h_i + \int_{t_i}^{t_{i+1}} f(x_i(t), u(t), t)\, dt \approx F(t_i, t_{i+1}, h_i, u_i), \qquad x_i(t_i) = h_i$$



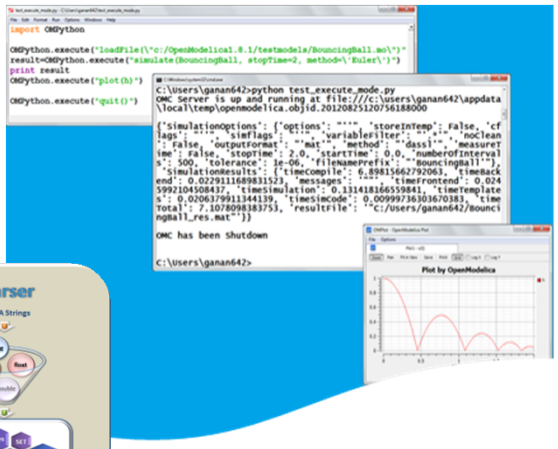
Example speedup, 16 cores:

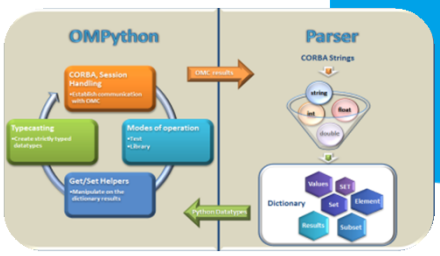


MODELICA

OMPython – Python Scripting with OpenModelica

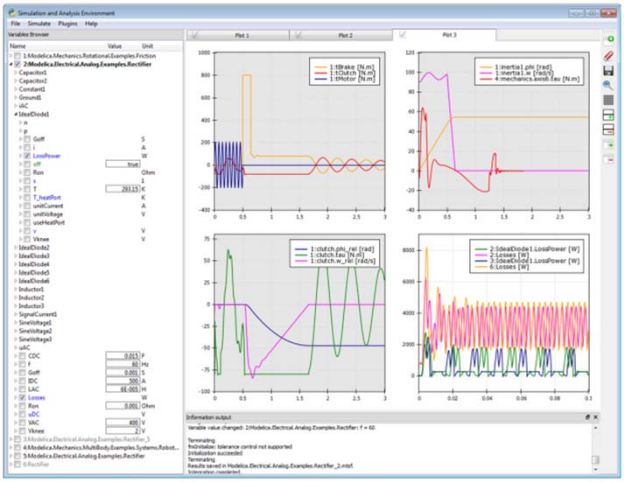
- *Interpretation of Modelica* commands and expressions
- Interactive Session handling
- Library / Tool
- Optimized Parser results
- Helper functions
- Deployable, Extensible and Distributable





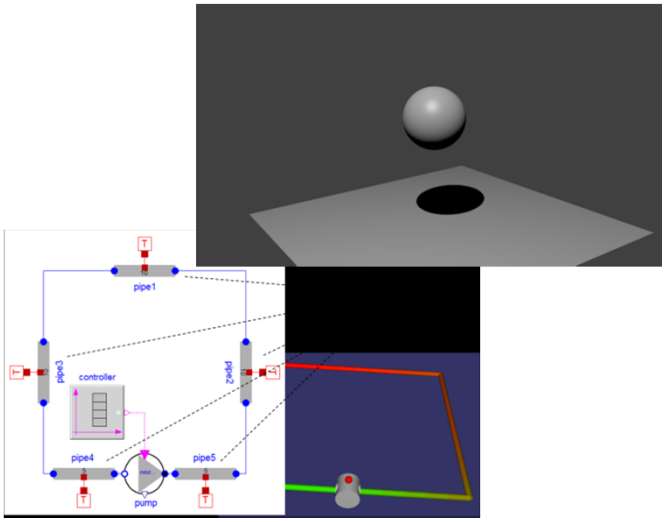
PySimulator Package

- PySimulator, a simulation and analysis package developed by DLR
- Free, downloadable
- Uses OMPython to simulate Modelica models by OpenModelica



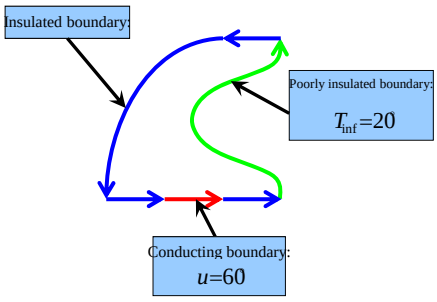
Modelica3D Library

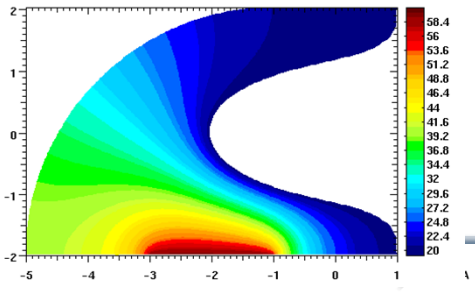
- Modelica 3D Graphics Library by Fraunhofer FIRST, Berlin
- Free, downloadable
- Can be used for 3D graphics in OpenModelica
- Shipped with OpenModelica



Extending Modelica with PDEs for 2D, 3D flow problems – Research

```
class PDEModel
   HeatNeumann h_iso;
   Dirichlet h_heated(g=50);
   HeatRobin h_glass(h_heat=30000);
   HeatTransfer ht;
   Rectangle2D dom;
equation
   dom.eq=ht;
   dom.left.bc=h_glass;
   dom.top.bc=h_iso;
   dom.right.bc=h_iso;
   dom.bottom.bc=h_heated;
end PDEModel;
```





Prototype in OpenModelica 2005
PhD Thesis by Levon Saldamli
www.openmodelica.org
Currently not operational

General Tool Interoperability & Model Exchange Functional Mock-up Interface (FMI)

The FMI development is part of the MODELISAR 29-partner project

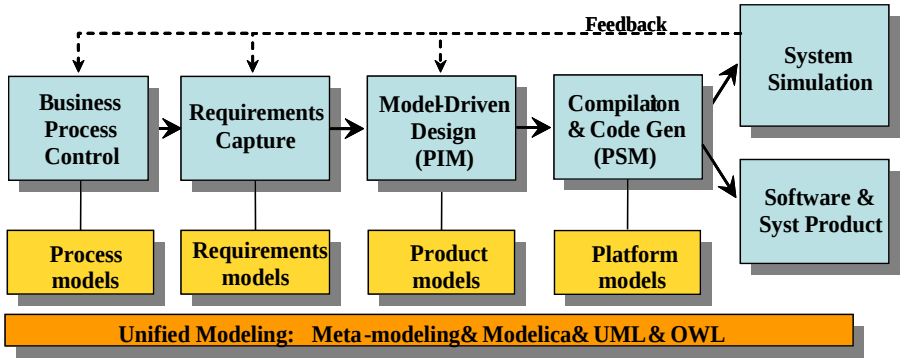
- FMI development initiated by **Daimler**
- Improved Software/Model/Hardware-in-the-Loop Simulation, of **physical** models and of **AUTOSAR** controller models from **different vendors** for automotive applications with **different levels of detail**.
- **Open Standard**
- **14 automotive use cases** for evaluation
- **> 10 tool vendors** are supporting it

Engine with ECU | Gearbox with ECU | Thermal systems | Automated cargo door | Chassis components, roadway, ECU (e.g. ESP) | etc.

functional mockup interface for model exchange and tool coupling

courtesy Daimler

OPENPROD – Large 28-partner European Project, 2009-2012 Vision of Cyber-Physical Model-Based Product Development



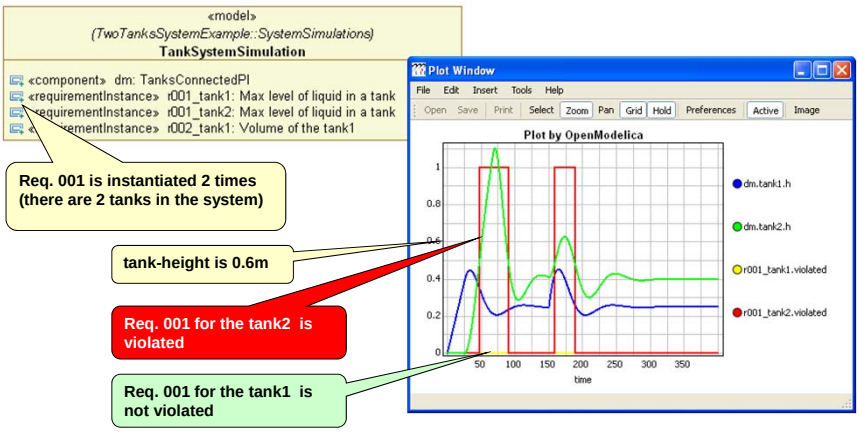
OPENPROD Vision of unified modeling framework for model-driven product development from platform independent models (PIM) to platform specific models (PSM)

Current work based on Eclipse, UML/SysML, OpenModelica

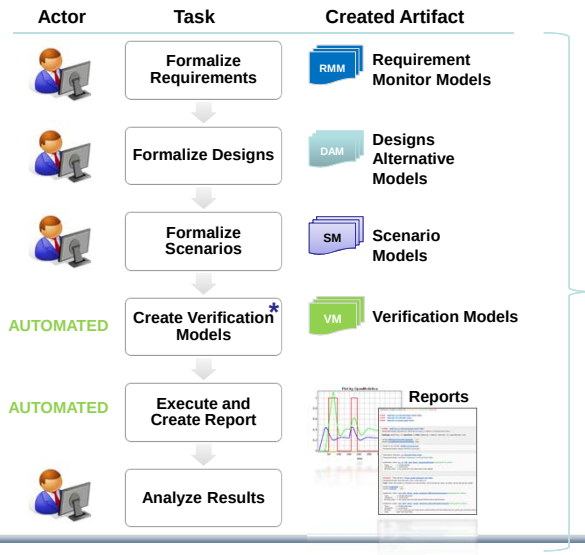
OpenModelica – ModelicaML UML Profile
SysML/UML to Modelica OMG Standardization

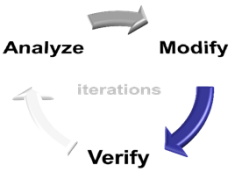
- ModelicaML is a UML Profile for SW/HW modeling
  - Applicable to "pure" UML or to other UML profiles, e.g. SysML
- Standardized Mapping UML/SysML to Modelica
  - Defines transformation/mapping for **executable** models
  - Being **standardized** by OMG
- ModelicaML
  - Defines graphical concrete syntax (graphical notation for diagram) for representing Modelica constructs integrated with UML
  - Includes graphical formalisms (e.g. State Machines, Activities, Requirements)
    - Which do not exist in Modelica language
    - Which are translated into executable Modelica code
  - Is defined towards generation of executable Modelica code
  - Current implementation based on the Papyrus UML tool + OpenModelica

Example: Simulation and Requirements Evaluation



vVDR Method – virtual Verification of Designs vs Requirements



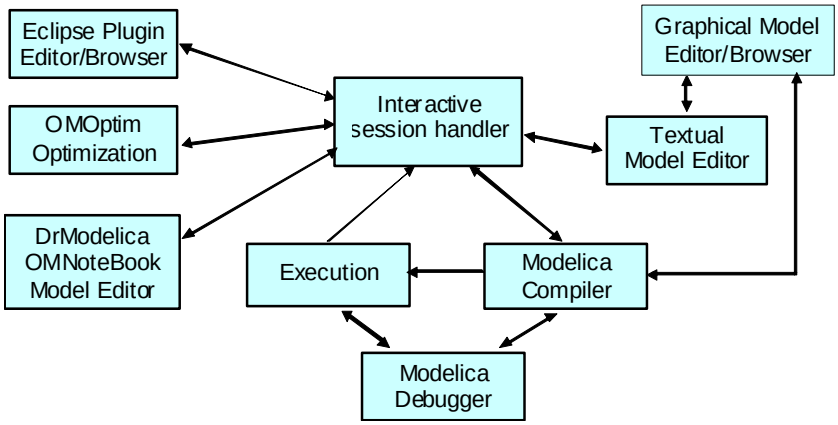


Goal: Enable on-demand verification of designs against requirements using automated model composition at any time during development.

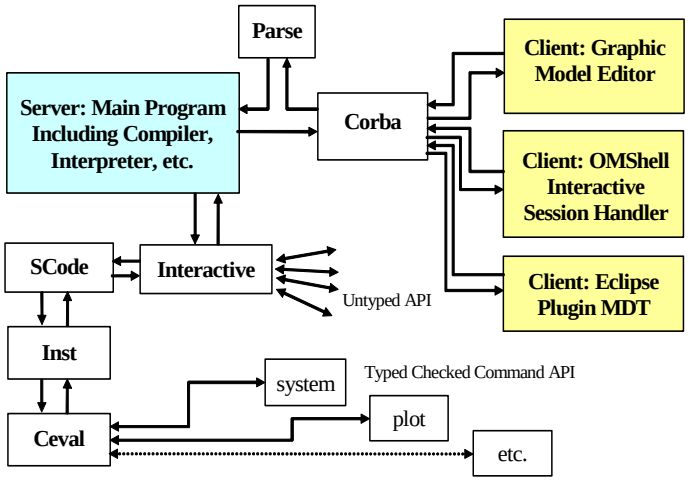
MetaModelica Language Extension Meta-Level Operations on Models

- Model Operations
  - Creation, Query, Manipulation,
  - Composition, Management
- Manipulation of model equations for
  - Optimization purposes
  - Parallelization, Model checking
  - Simulation with different solvers
- MetaModelica language features
  - Lists, trees, pattern matching, symbolic transformations
  - Garbage collection
- Very Large MetaModelica Application
  - OpenModelica compiler, implemented in 150 000 lines of MetaModelica, compiles itself efficiently.

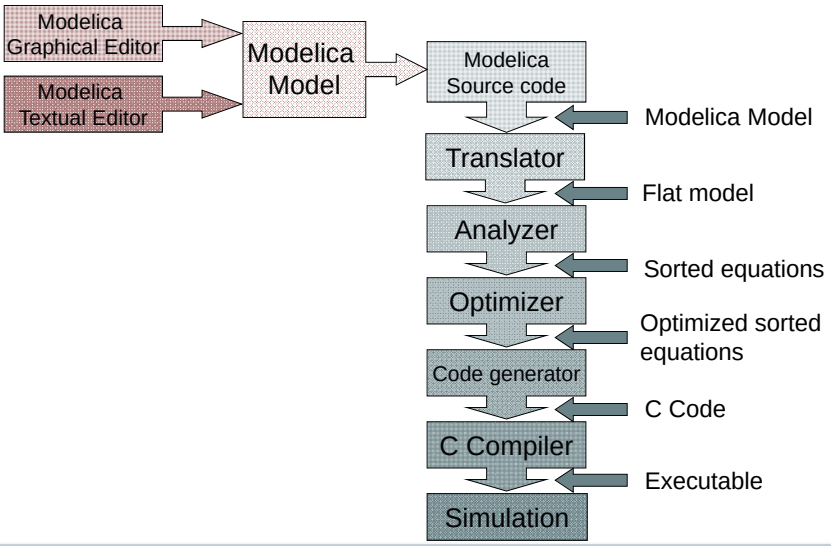
OpenModelica Environment Architecture



OpenModelica Client-Server Architecture



Translation of Models to Simulation Code



Corba Client-Server API

- Simple text-based (string) communication in Modelica Syntax
- API supporting model structure query and update

Example Calls:
Calls fulfill the normal Modelica function call syntax.:

```
saveModel("MyResistorFile.mo",MyResistor)
```

will save the model MyResistor into the file “MyResistorFile.mo”.

For creating new models it is most practical to send a model, e.g.:

```
model Foo   end Foo;
```

or, e.g.,

```
connector Port   end Port;
```


Some of the Corba API functions

`saveModel(A1<string>,A2<cref>)`	Saves the model (A2) in a file given by a string (A1). This call is also in typed API.
`loadFile(A1<string>)`	Loads all models in the file. Also in typed API. Returns list of names of top level classes in the loaded files.
`loadModel(A1<cref>)`	Loads the model (A1) by looking up the correct file to load in `$MODELICAPATH`. Loads all models in that file into the symbol table.
`deleteClass(A1<cref>)`	Deletes the class from the symbol table.
`addComponent(A1<ident>,A2<cref>, A3<cref>,annotate=<expr>)`	Adds a component with name (A1), type (A2), and class (A3) as arguments. Optional annotations are given with the named argument `annotate`.
`deleteComponent(A1<ident>, A2<cref>)`	Deletes a component (A1) within a class (A2).
`updateComponent(A1<ident>, A2<cref>, A3<cref>,annotate=<expr>)`	Updates an already existing component with name (A1), type (A2), and class (A3) as arguments. Optional annotations are given with the named argument `annotate`.
`addClassAnnotation(A1<cref>, annotate=<expr>)`	Adds annotation given by A2(in the form `annotate= classmod(...)`) to the model definition referenced by A1. Should be used to add Icon Diagram and Documentation annotations.
`getComponents(A1<cref>)`	Returns a list of the component declarations within class A1: `{{Atype,varidA,"commentA"},{Btype,varidB,"commentB"}, {...}}`
`getComponentAnnotations(A1<cref>)`	Returns a list `{...}` of all annotations of all components in A1, in the same order as the components, one annotation per component.
`getComponentCount(A1<cref>)`	Returns the number (as a string) of components in a class, e.g return `"2"` if there are 2 components.
`getNthComponent(A1<cref>,A2<int>)`	Returns the belonging class, component name and type name of the nth component of a class, e.g. `"A.B.C,R2,Resistor"`, where the first component is numbered 1.
`getNthComponentAnnotation( A1<cref>,A2<int>)`	Returns the flattened annotation record of the nth component (A2) (the first is has no 1) within class/component A1. Consists of a comma separated string of 15 values, see Annotations in Section 2.4.4 below, e.g `"false,10,30,..."`
`getNthComponentModification( A1<cref>,A2<int>)??`	Returns the modification of the nth component (A2) where the first has no 1) of class/component A1.
`getInheritanceCount(A1<cref>)`	Returns the number (as a string) of inherited classes of a class.
`getNthInheritedClass(A1<cref>,`	Returns the type name of the nth inherited class of a class. The first class has number 1

Platforms

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

- Windows – currently main development and release platform
- Linux – available. Also used for development
- Mac – available

Main Events March 2011 – October 2012

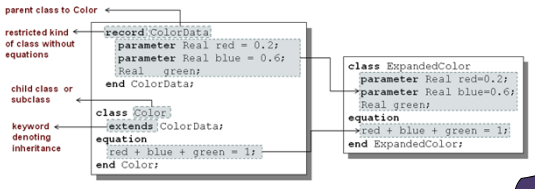
- OSMC expanded from 32 to 45 organizational members
- OpenModelica **1.7 release** (April 2011)
- OpenModelica **1.8 release** (Nov 2011)
  - Support for FMI Export and Import
  - Flattening of the whole MSL 3.1 Media library, and about half of the Fluid library.
  - Improved index reduction with dynamic state selection
  - Beta release of new efficient debugger for algorithmic Modelica/MetaModelica
- OpenModelica **1.8.1 release** (April 2012)
  - Operator Overloading support
  - Dramatically improved flattening speed for some models
  - Improved simulation run-time
  - ModelicaML with Modelica library import (MSL) and value-bindings
- OpenModelica **1.9.0 beta release** (October 2012)
  - MSL 3.1 simulation improved, from 36 to 74 to 100 example models
  - Improved simulation of other libraries, e.g. ThermoSysPro, PlanarMechanics, etc.
  - Improved algorithms for tearing, matching, dynamic state selection, index reduction
  - Full version of OMPython, updated ModelicaML for requirements verification

OpenModelica – Outlook for fall 2012 & spring 2013

- 2012/2013. Continued high priority on better support for the Modelica standard **libraries** and other libraries (e.g. ThermoSysPro)
- Fall 2012. Complete **MultiBody** simulation, already almost complete
- Nov-Dec 2012. Flattening of most of **Fluid**, and simulation of a large part of Fluid.
- Oct-Nov 2012. **FMI** co-simulation
- Oct-Dec 2012. **FMI** import and export, some FMI 2.0 features
- Oct-Nov 2012. OMEdit graphic editor enhancements
- Spring 2013. Full Fluid library simulation
- Spring 2013. **Integrated** Modelica equation **debugger**
- Spring 2013. Shifting OM development to using new bootstrapped OpenModelica Compiler

Modelica Language Concepts and Textual Modeling

Classes and Inheritance

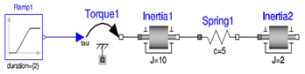
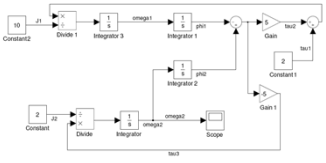


Typed
Declarative
Equation-based
Textual Language

Hybrid
Modeling

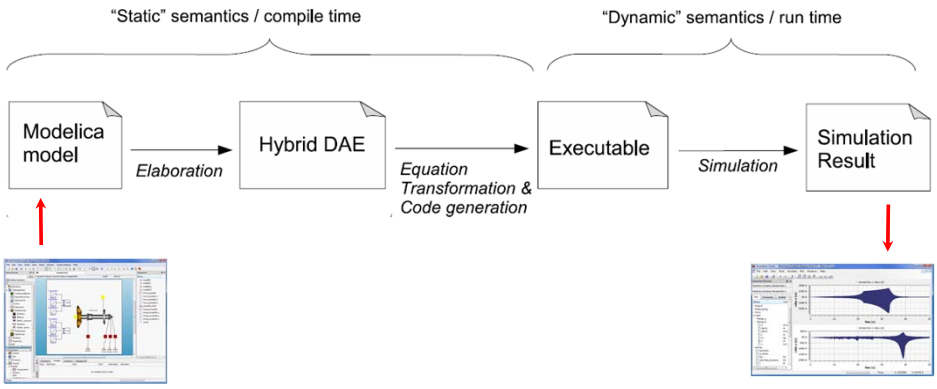
Acausal Modeling

The order of computations is not decided at modeling time

	Acausal	Causal
Visual Component Level		

Equation Level	A resistor *equation*:
	`R*i = v;`
		Causal possibilities:
		`i := v/R;`
		`v := R*i;`
		`R := v/i;`

Typical Simulation Process

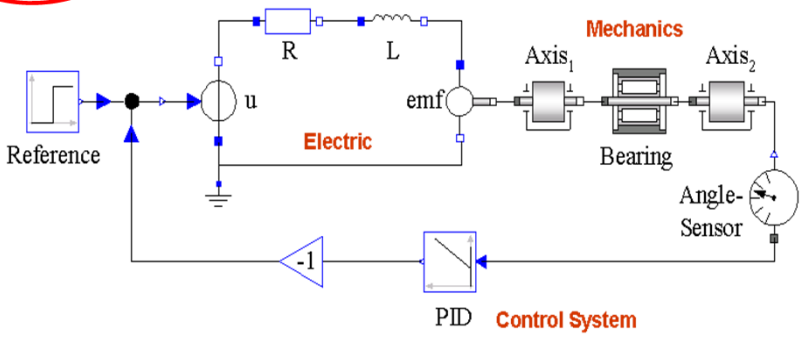


What is Special about Modelica?

- Multi-Domain Modeling
- Visual acausal hierarchical component modeling
- Typed declarative equation-based textual language
- Hybrid modeling and simulation

What is Special about Modelica?

Multi-Domain Modeling



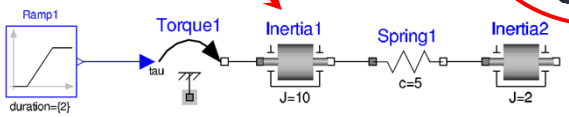
What is Special about Modelica?

Multi-Domain Modeling

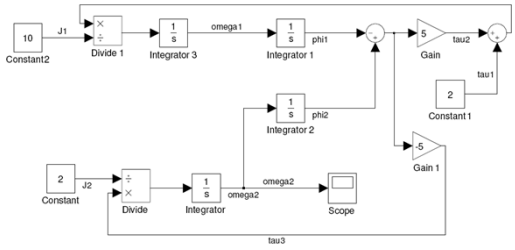
Visual Acausal Hierarchical Component Modeling

Keeps the physical structure

Acausal model (Modelica)



Causal block-based model (Simulink)

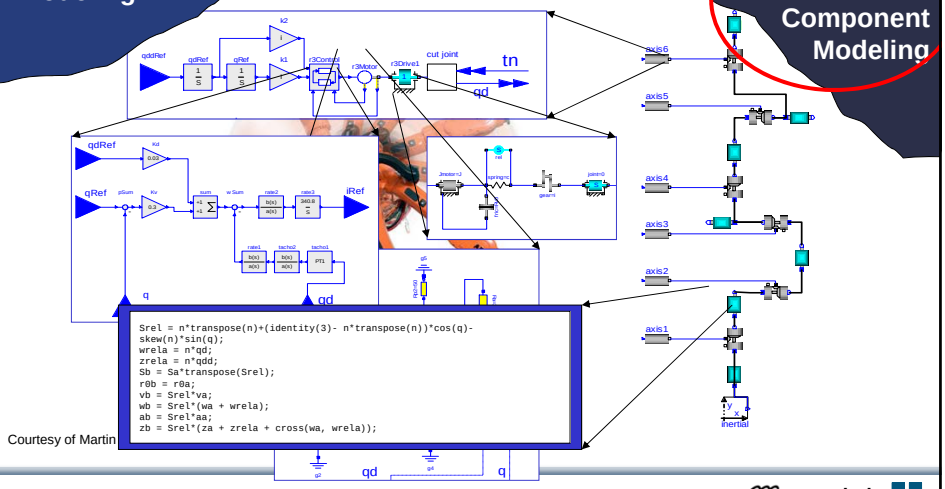


What is Special about Modelica?

Multi-Domain Modeling

Hierarchical system modeling

Visual Acausal Hierarchical Component Modeling



Courtesy of Martin Otter

Courtesy of Martin Otter

What is Special about Modelica?

Multi-Domain Modeling

Visual Acausal Hierarchical Component Modeling

A textual *class-based* language
OO primary used for as a structuring concept

Behaviour described declaratively using

- Differential algebraic equations (DAE) (continuous-time)
- Event triggers (discrete-time)

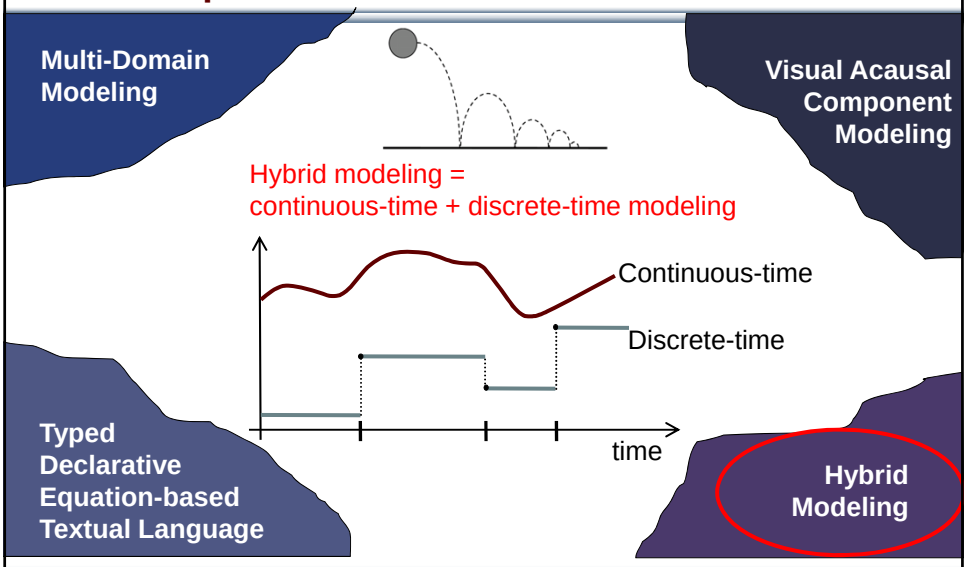
Variable declarations

```
class VanDerPol  "Van der Pol oscillator model"
  Real x(start = 1)  "Descriptive string for x";
  Real y(start = 1)  "y coordinate";
  parameter Real lambda = 0.3;
equation
  der(x) = y;
  der(y) = -x + lambda*(1 - x*x)*y;
end VanDerPol;
```

Differential equations

Typed Declarative Equation-based Textual Language

What is Special about Modelica?



Modelica Classes and Inheritance

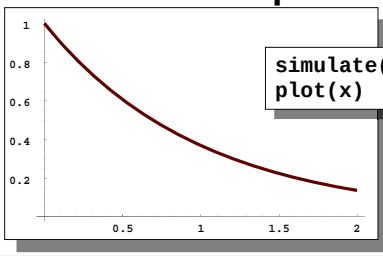
Simplest Model – Hello World!

A Modelica "Hello World" model

Equation: x' = - x
Initial condition: x(0) = 1

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

Simulation in OpenModelica environment




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

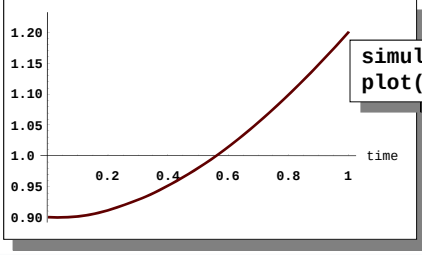

Model Including Algebraic Equations

Include algebraic equation

Algebraic equations contain no derivatives

```
class DAEexample
  Real x(start=0.9);
  Real y;
equation
  der(y)+(1+0.5*sin(y))*der(x)
    = sin(time);
  x - y = exp(-0.9*x)*cos(y);
end DAEexample;
```

Simulation in OpenModelica environment

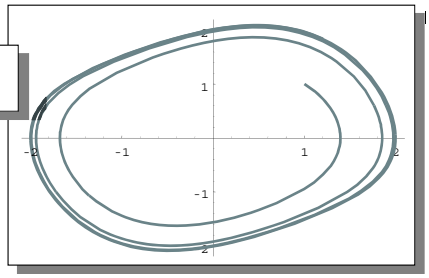



```
simulate(DAEexample, stopTime = 1)
plot(x)
```


Example class: 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)
```



Exercises – Simple Textual Modeling

- Start OMNotebook
  - Start->Programs->OpenModelica->OMNotebook
  - Open File: Exercise01-classes-simple-textual.onb
- Open Exercise01-classes-simple-textual.pdf

Exercises 2.1 and 2.2

- Open the **Exercise01-classes-simple-textual.onb** found in the Tutorial directory.
- Locate the VanDerPol model in DrModelica (link from Section 2.1), using OMNotebook!
- **Exercise 2.1**: Simulate and plot VanDerPol. Do a slight change in the model, re-simulate and re-plot.
- **Exercise 2.2**. 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)
```


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";`

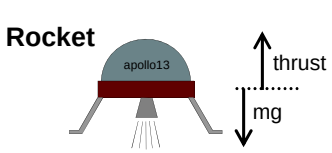
```
class VanDerPol  "Van der Pol oscillator model"
  Real x(start = 1)  "Descriptive string for x"; // x starts at 1
  Real y(start = 1)  "y coordinate";             // y starts at 1
  parameter Real lambda = 0.3;
equation
  der(x) = y;                          // This is the 1st diff equation //
 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



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

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

$$altitude' = velocity$$

$$velocity' = acceleration$$

new model parameters (changeable before the simulation)

floating point type

differentiation with regards to time

declaration comment

start value

name + default value

mathematical equation (acausal)

```
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;
```


Celestial Body Class

A class declaration creates a *type name* in Modelica

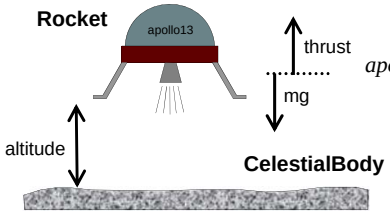
```
class CelestialBody
  constant  Real    g = 6.672e-11;
  parameter Real    radius;
  parameter String  name;
  parameter Real    mass;
end CelestialBody;
```

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

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

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

Moon Landing



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

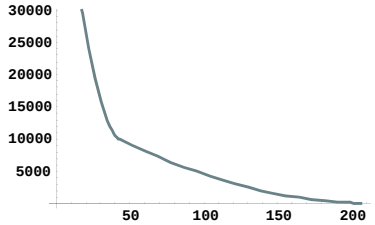
only access inside the class

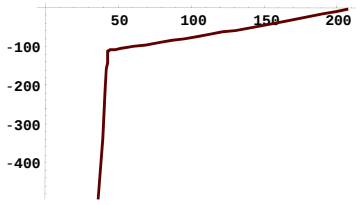
access by dot notation outside the class

```
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

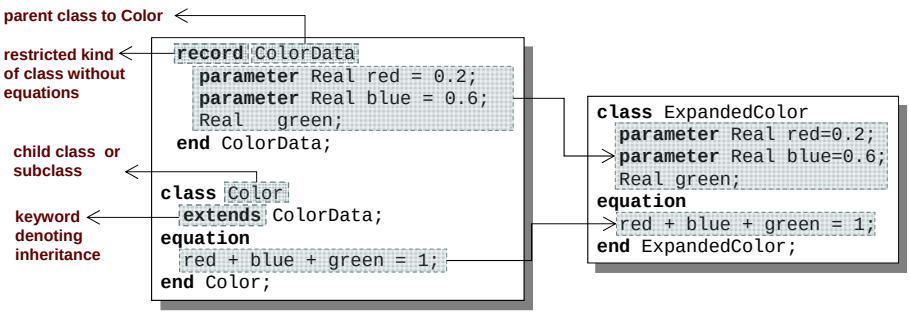
```
model CelestialBody
  constant  Real    g = 6.672e-11;
  parameter Real    radius;
  parameter String  name;
  parameter Real    mass;
end CelestialBody;
```


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 in Lecture 4

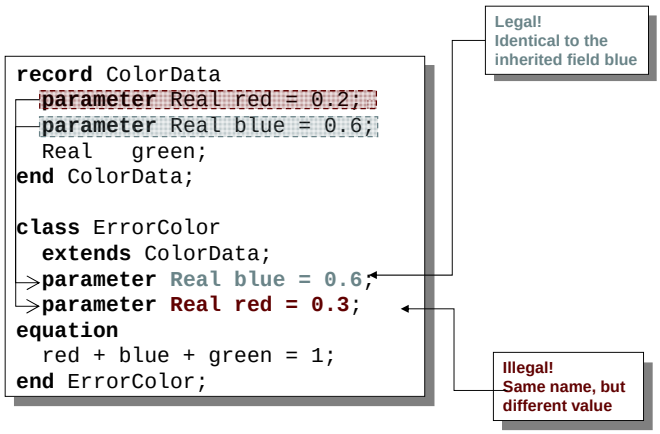
```
function sum
  input  Real arg1;
  input  Real arg2;
  output Real result;
algorithm
  result := arg1+arg2;
end sum;
```


Inheritance



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

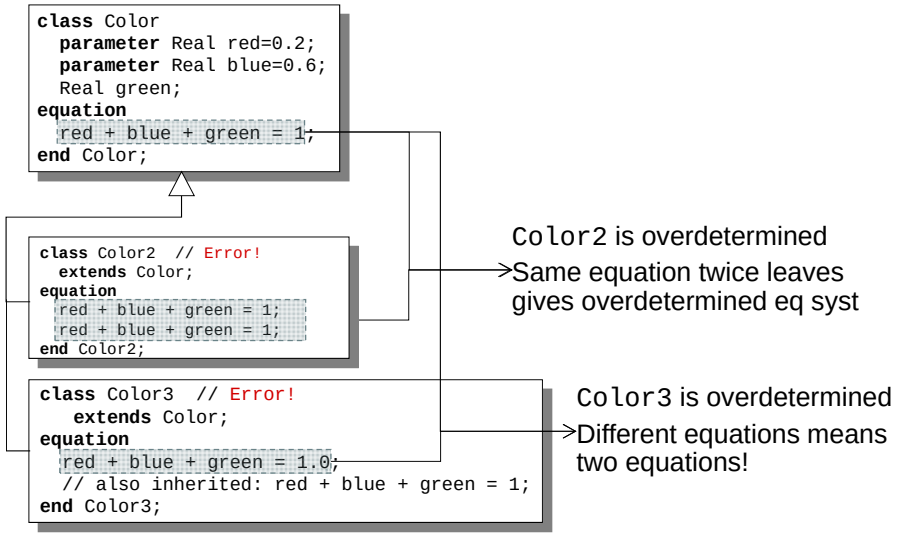
Inheriting definitions



Inheriting multiple identical definitions results in only one definition

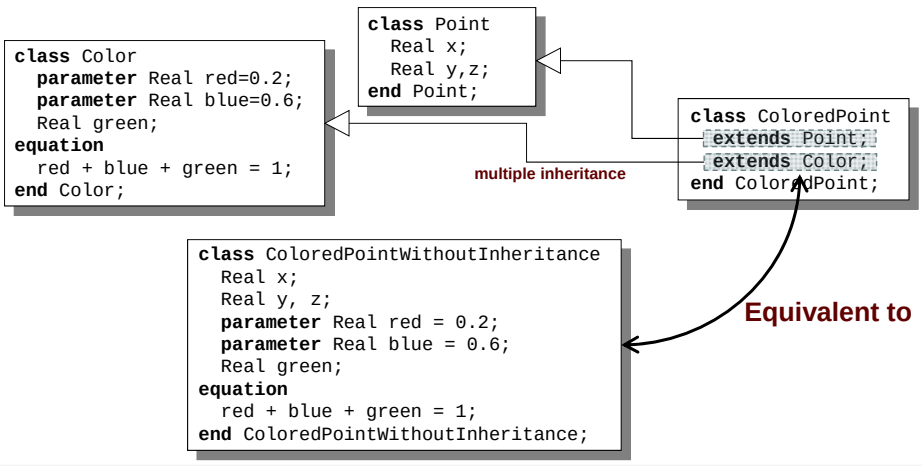
Inheriting multiple different definitions of the same item is an error

Inheritance of Equations



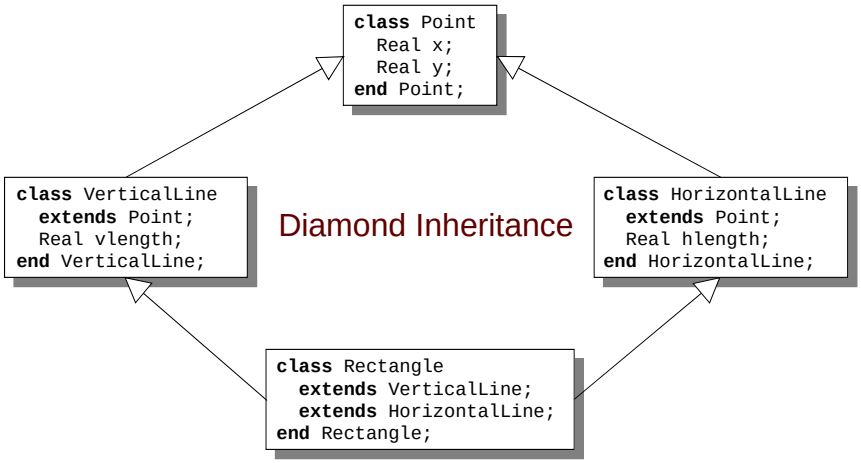
Multiple Inheritance

Multiple Inheritance is fine – inheriting both geometry and color



Multiple Inheritance cont'

Only one copy of multiply inherited class `Point` is kept



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;
```

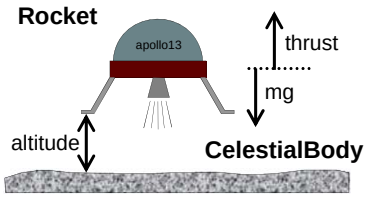
```
connector MyPin = Pin;
```


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

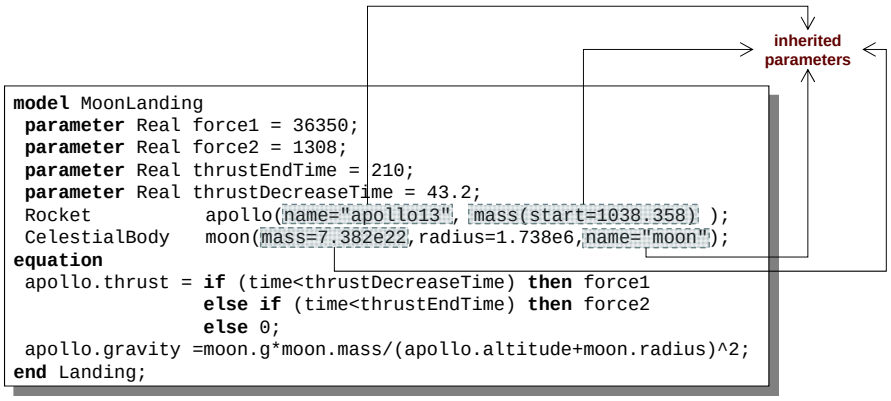



```
model Body "generic body"
  Real   mass;
  String name;
end Body;
```

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

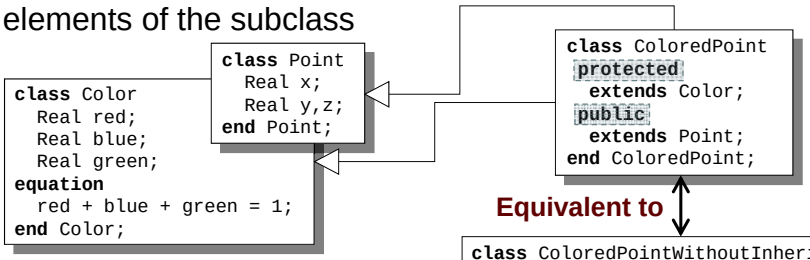
```
model Rocket "generic rocket class"
  extends Body;
  parameter Real massLossRate=0.000277;
  Real altitude(start= 59404);
  Real velocity(start= -2003);
  Real acceleration;
  Real thrust;
  Real gravity;
equation
  thrust-mass*gravity= mass*acceleration;
  der(mass)= -massLossRate*abs(thrust);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

The Moon Landing Example using Inheritance cont'



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!

Equivalent to

```
class ColoredPointWithoutInheritance
  Real x;
  Real y,z;
  protected Real red;
  protected Real blue;
  protected Real green;
equation
  red + blue + green = 1;
end ColoredPointWithoutInheritance;
```


Advanced Topic

- Class parameterization

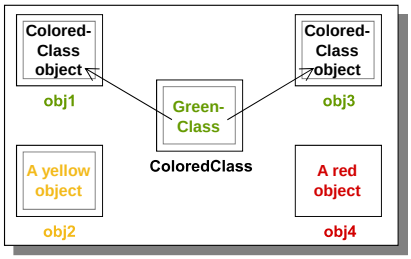
Generic Classes with Type Parameters

Formal class parameters are replaceable variable or type declarations within the class (usually) marked with the prefix `replaceable`

```
class C
  replaceable class ColoredClass = GreenClass;
  ColoredClass            obj1(p1=5);
  replaceable YellowClass obj2;
  ColoredClass            obj3;
  RedClass                obj4;
equation
end C;
```

Actual arguments to classes are modifiers, which when containing whole variable declarations or types are preceded by the prefix `redeclare`

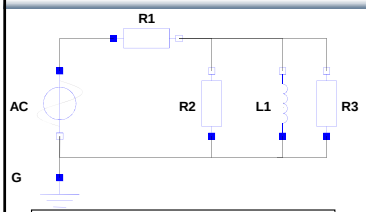
```
class C2 =
  C(redeclare class ColoredClass = BlueClass);
```



Equivalent to

```
class C2
  BlueClass   obj1(p1=5);
  YellowClass obj2;
  BlueClass   obj3;
  RedClass    obj4;
equation
end C2;
```

Class Parameterization when Class Parameters are Components



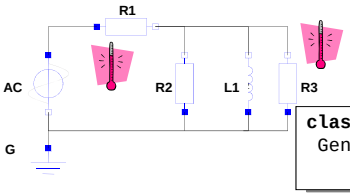
The class `ElectricalCircuit` has been converted into a parameterized generic class `GenericElectricalCircuit` with three formal class parameters R1, R2, R3, marked by the keyword `replaceable`

```
class ElectricalCircuit
  Resistor R1(R=100);
  Resistor R2(R=200);
  Resistor R3(R=300);
  Inductor L1;
  SineVoltage AC;
  Groung G;
equation
  connect(R1.n,R2.n);
  connect(R1.n,L1.n);
  connect(R1.n,R3.n);
  connect(R1.p,AC.p);
  .....
end ElectricalCircuit;
```

Class parameterization

```
class GenericElectricalCircuit
  replaceable Resistor R1(R=100);
  replaceable Resistor R2(R=200);
  replaceable Resistor R3(R=300);
  Inductor L1;
  SineVoltage AC;
  Groung G;
equation
  connect(R1.n,R2.n);
  connect(R1.n,L1.n);
  connect(R1.n,R3.n);
  connect(R1.p,AC.p);
  .....
end GenericElectricalCircuit;
```


Class Parameterization when Class Parameters are Components - cont'



A more specialized class `TemperatureElectricalCircuit` is created by changing the types of R1, R3, to `TempResistor`

```
class TemperatureElectricalCircuit =
 GenericElectricalCircuit (redeclare TempResistor R1
                           redeclare TempResistor R3);
```

```
class TemperatureElectricalCircuit
  parameter Real Temp=20;
  extends GenericElectricalCircuit(
    redeclare TempResistor R1(RT=0.1, Temp=Temp),
    redeclare TempResistor R3(R=300));
end TemperatureElectricalCircuit
```

We add a temperature variable `Temp` for the temperature of the resistor circuit and modifiers for R1 and R3 which are now `TempResistors`.

equivalent to

```
class ExpandedTemperatureElectricalCircuit
  parameter Real Temp;
  TempResistor R1(R=200, RT=0.1, Temp=Temp),
  replaceable Resistor R2;
  TempResistor R3(R=300);
equation
  ....
end ExpandedTemperatureElectricalCircuit
```


Exercises 1 Simple Textual Continued

- Continue exercises in Exercise01-classes-simple-textual.onb
- Do Exercises 1.3, 1.4, 1.5 and 2

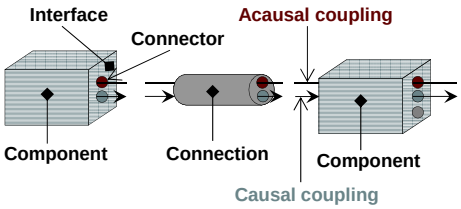
Exercise 1.3 – Model the System Below

- Model this Simple System of Equations in Modelica

$$\dot{x} = 2 * x * y - 3 * x$$
$$\dot{y} = 5 * y - 7 * x * y$$
$$x(0) = 2$$
$$y(0) = 3$$

Components, Connectors and Connections

Software Component Model



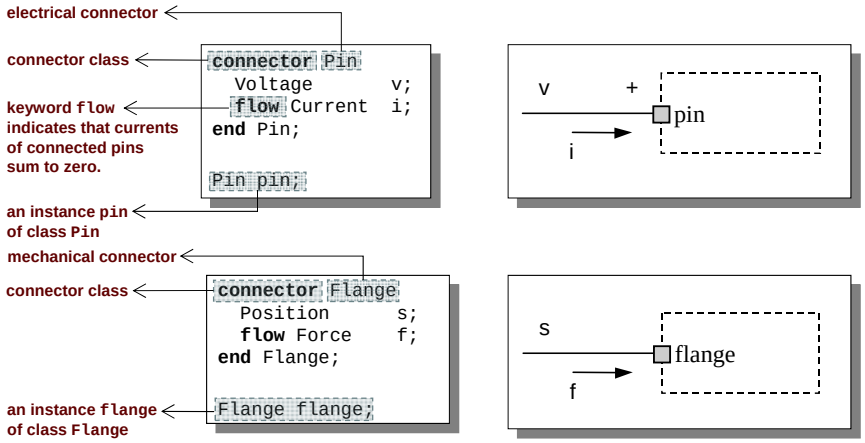
A component class should be defined *independently of the environment,* very essential for *reusability*

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

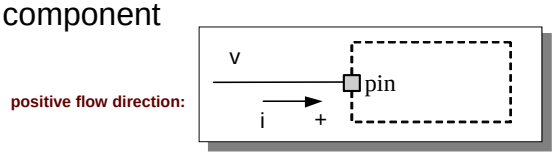
Two kinds of variables in connectors:

- *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



Physical Connector Classes Based on Energy Flow

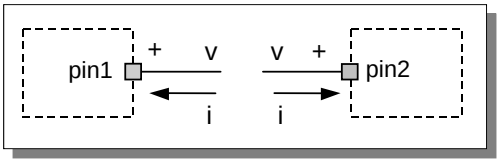
Domain Type	Potential	Flow	Carrier	Modelica Library
Electrical	Voltage	Current	Charge	`Electrical.Analog`
Translational	Position	Force	Linear momentum	`Mechanical.Translational`
Rotational	Angle	Torque	Angular momentum	`Mechanical.Rotational`
Magnetic	Magnetic potential	Magnetic flux rate	Magnetic flux	
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:

$$v_1 = v_2 = v_3 = \ldots v_n$$

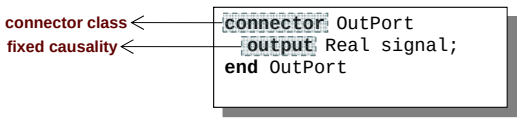
Each primitive connection set of flow variables is used to generate *sum-to-zero* equations of the form:

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

Acausal, Causal, and Composite Connections

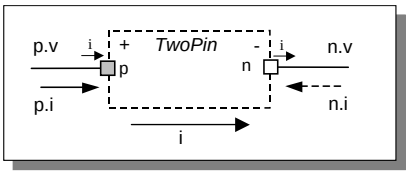
Two *basic* and one *composite* kind of connection in Modelica

- *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;
```

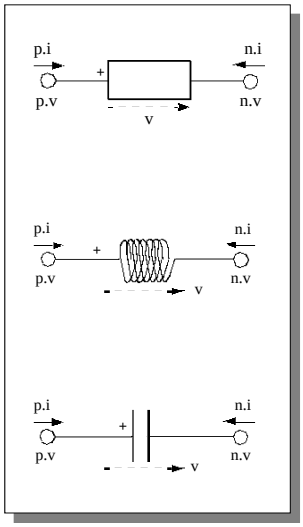
electrical connector class

Electrical Components

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

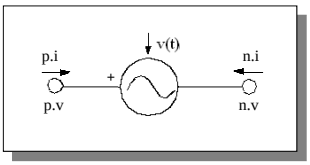
```
model Inductor "Ideal electrical inductor"
  extends TwoPin;
  parameter Real L "Inductance";
equation
  L*der(i) = v;
end Inductor;
```

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

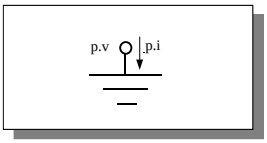


Electrical Components cont’

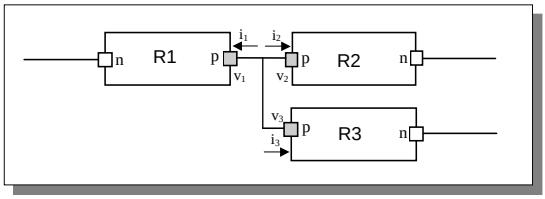
```
model Source
   extends TwoPin;
   parameter Real A,w;
equation
   v = A*sin(w*time);
end Resistor;
```




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



Resistor Circuit

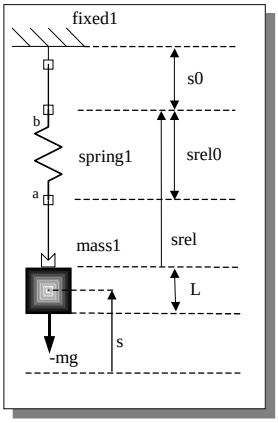



```
model ResistorCircuit
  Resistor R1(R=100);
  Resistor R2(R=200);
  Resistor R3(R=300);
equation
  connect(R1.p, R2.p);
  connect(R1.p, R3.p);
end ResistorCircuit;
```

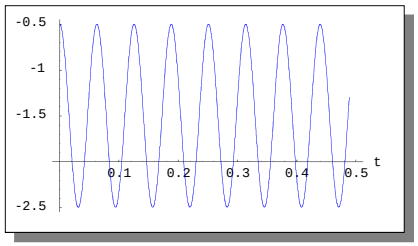
Corresponds to

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


An Oscillating Mass Connected to a Spring



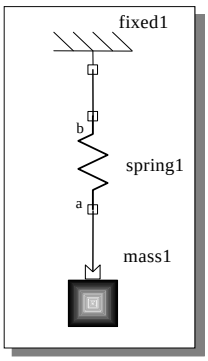

```
model Oscillator
  Mass    mass1(L=1, s(start=-0.5));
  Spring  spring1(srel0=2, c=10000);
  Fixed   fixed1(s0=1.0);
equation
  connect(spring1.flange_b, fixed1.flange_b);
  connect(mass1.flange_b, spring1.flange_a);
end Oscillator;
```



Extra Exercise

- Locate the Oscillator model in DrModelica using OMNotebook!
- Simulate and plot the example. Do a slight change in the model e.g. different elasticity c, re-simulate and re-plot.

- Draw the `Oscillator` model using the graphic connection editor e.g. using the library `Modelica.Mechanical.Translational`
- Including components `SlidingMass`, `Force`, `Blocks.Sources.Constant`
- Simulate and plot!

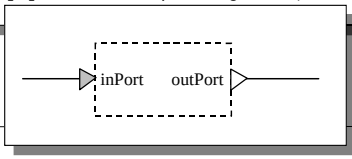


Signal Based Connector Classes

fixed causality

```
connector InPort "Connector with input signals of type Real"
  parameter Integer n=1 "Dimension of signal vector";
  input Real signal[n]  "Real input signals";
end InPort;

connector OutPort "Connector with output signals of type Real"
  parameter Integer n=1  "Dimension of signal vector";
  output Real signal[n]  "Real output signals";
end OutPort;
```

fixed causality

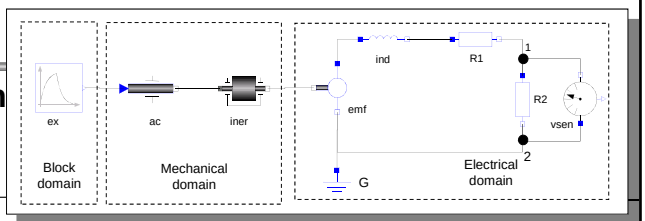


multiple input
single output
block

```
partial block MISO
  "Multiple Input Single Output continuous control block"
  parameter Integer nin=1 "Number of inputs";
  InPort  inPort(n=nin) "Connector of Real input signals";
  OutPort outPort(n=1)  "Connector of Real output signal";
protected
  Real u[:] = inPort.signal     "Input signals";
  Real y    = outPort.signal[1] "Output signal";
end MISO; // From Modelica.Blocks.Interfaces
```


Connecting Components from Multiple Domains

- Block domain
- Mechanical domain
- Electrical domain

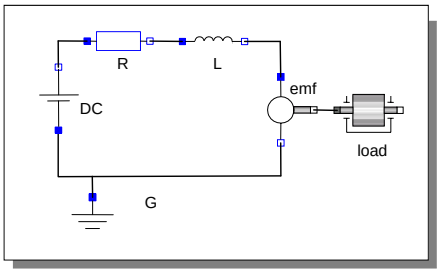



```
model Generator
  Modelica.Mechanics.Rotational.Accelerate ac;
  Modelica.Mechanics.Rotational.Inertia iner;
  Modelica.Electrical.Analog.Basic.EMF emf(k=-1);
  Modelica.Electrical.Analog.Basic.Inductor ind(L=0.1);
  Modelica.Electrical.Analog.Basic.Resistor R1,R2;
  Modelica.Electrical.Analog.Basic.Ground 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;
```


Simple Modelica 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;
```



Corresponding DCMotor Model Equations

The following equations are automatically derived from the Modelica model:

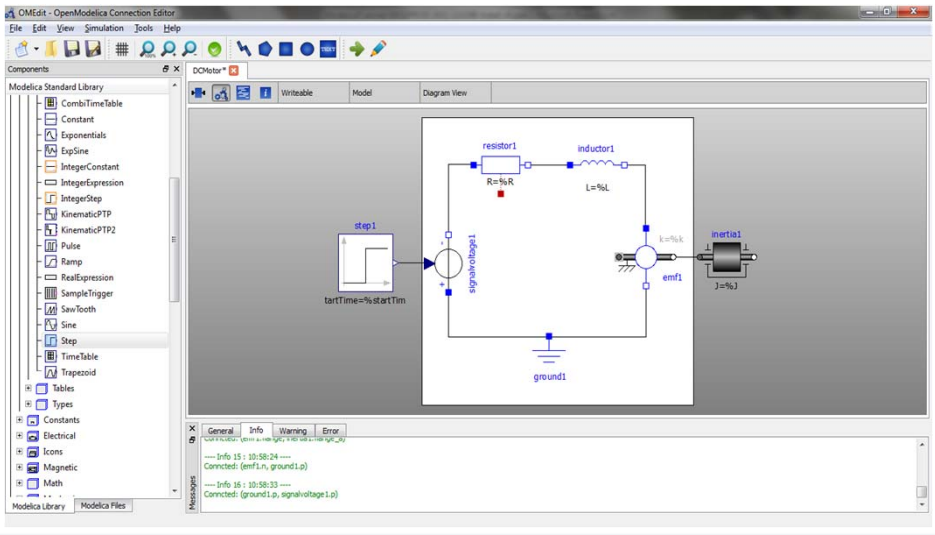
```
0 == DC.p.i + R.n.i
DC.p.v == R.n.v

0 == R.p.i + L.n.i
R.p.v == L.n.v

0 == L.p.i + EM.n.i
L.p.v == EM.n.v

0 == EM.p.i + DC.n.i
EM.p.v == DC.n.v

0 == DC.n.i + G.p.i
DC.n.v == G.p.v

EM.u == EM.p.v - EM.n.v
0 == EM.p.i + EM.n.i
EM.i == EM.p.i
EM.u == EM.k * EM.ω
EM.i == EM.M / EM.k
EM.J * EM.ω == EM.M - EM.b * EM.ω

DC.u == DC.p.v - DC.n.v
0 == DC.p.i + DC.n.i
DC.i == DC.p.i
DC.u == DC.Amp * Sin[2 π DC.f * t]

R.u == R.p.v - R.n.v
0 == R.p.i + R.n.i
R.i == R.p.i
R.u == R.R * R.i

L.u == L.p.v - L.n.v
0 == L.p.i + L.n.i
L.i == L.p.i
L.u == L.L * L.i'
```

(`load` component not included)

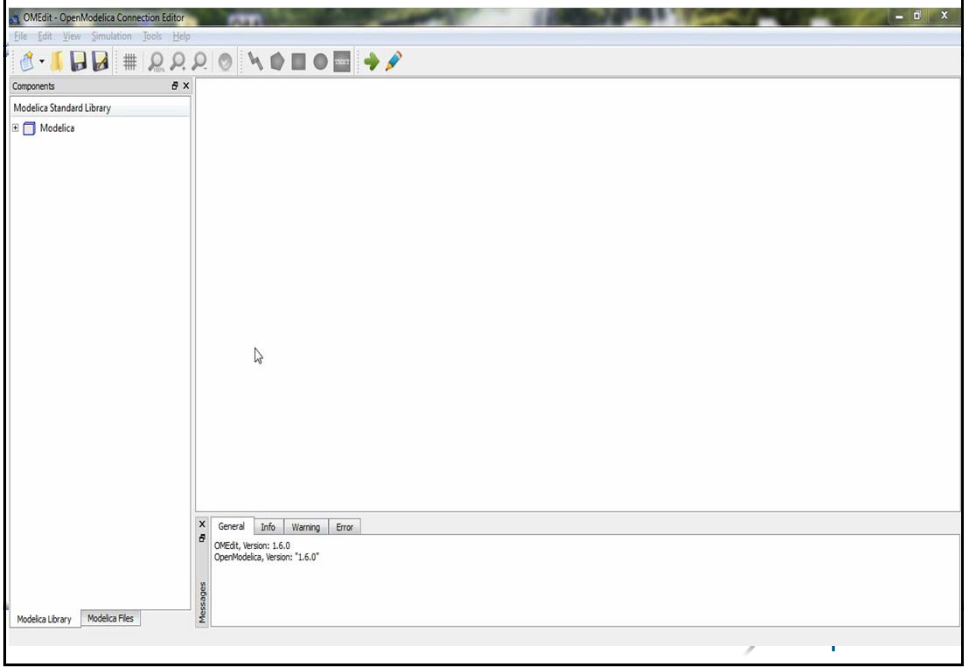
Automatic transformation to ODE or DAE for simulation:

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

Graphical Modeling - Using Drag and Drop Composition

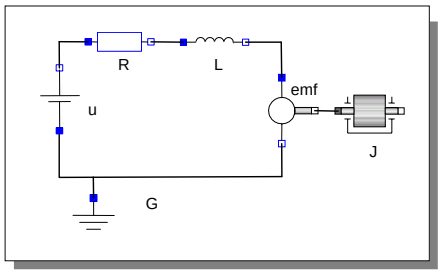


Graphical Modeling Animation – DCMotor



Graphical Exercise 3.1

- Open `Exercise02-graphical-modeling.onb` and the corresponding .pdf
- 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.

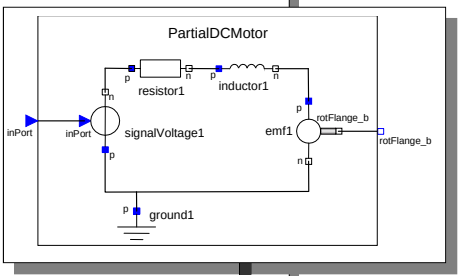


Hierarchically Structured Components

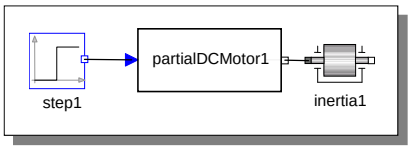
An *inside connector* is a connector belonging to an *internal component* of a structured component class.

An *outside connector* is a connector that is part of the *external interface* of a structured component class, is declared directly within that class

```
partial model PartialDCMotor
  InPort        inPort;      //  Outside signal connector
  RotFlange_b   rotFlange_b; // Outside rotational flange connector
  Inductor      inductor1;
  Resistor      resistor1;
  Ground        ground1;
  EMF           emf1;
  SignalVoltage signalVoltage1;
equation
  connect(inPort,signalVoltage1.inPort);
  connect(signalVoltage1.n, resistor1.p);
  connect(resistor1.n,      inductor1.p);
  connect(signalVoltage1.p, ground1.p);
  connect(ground1.p,        emf1.n);
  connect(inductor1.n,      emf1.p);
  connect(emf1.rotFlange_b, rotFlange_b);
end PartialDCMotor;
```



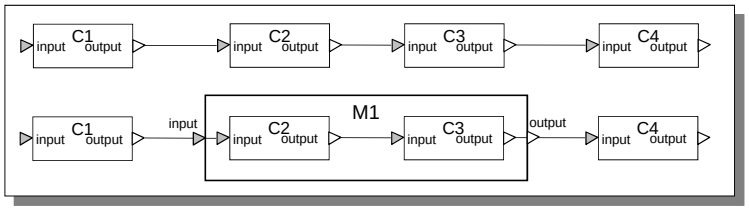
Hierarchically Structured Components cont'




```
model DCMotorCircuit2
  Step           step1;
  PartialDCMotor partialDCMotor1;
  Inertia        inertia1;
equation
  connect(step1.outPort, partialDCMotor1.inPort);
  connect(partialDCMotor1.rotFlange_b, inertia1.rotFlange_a);
end DCMotorCircuit2;
```


Connection Restrictions

- Two *acausal* connectors can be connected to each other
- An `input` connector can be connected to an `output` connector or vice versa
- An `input` or `output` connector can be connected to an *acausal* connector, i.e. a connector without `input`/`output` prefixes
- An *outside* `input` connector behaves approximately like an `output` connector internally
- An *outside* `output` connector behaves approximately like an `input` connector internally



Connector Restrictions cont'

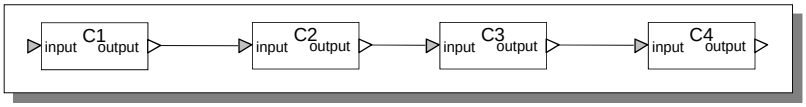
```
connector RealInput
  input Real signal;
end RealInput;
```

```
connector RealOutput
  output Real signal;
end RealOutput;
```

```
class C
  RealInput  u;  // input connector
  RealOutput y; // output connector
end C;
```

```
class CInst
  C  C1, C2, C3, C4; // Instances of C
equation
  connect(C1.outPort, C2.inPort);
  connect(C2.outPort, C3.inPort);
  connect(C3.outPort, C4.inPort);
end CInst;
```

A circuit consisting of four connected components `C1, C2, C3, and C4` which are instances of the class `C`

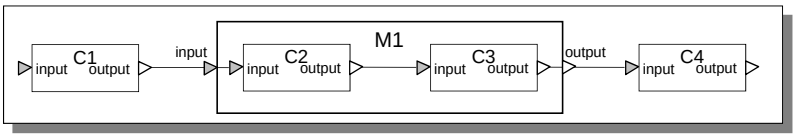


Connector Restrictions cont'

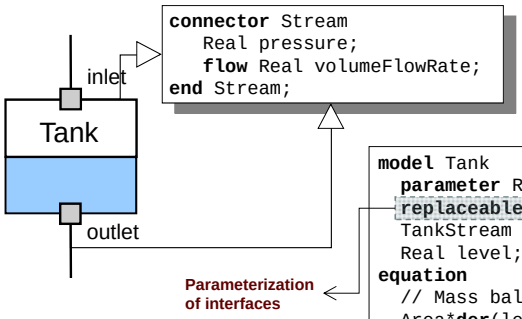
```
class M   "Structured class M"
  RealInput  u; // Outside input connector
  RealOutput y; // Outside output connector
  C  C2;
  C  C3;
end M;
```

A circuit in which the middle components `C2` and `C3` are placed inside a structured component `M1` to which two outside connectors `M1.u` and `M1.y` have been attached.

```
class MInst
  M  M1;   // Instance of M
equation
  connect(C1.y, M1.u); // Normal connection of outPort to inPort
  connect(M1.u, C2.u); // Outside inPort connected to inside inPort
  connect(C2.y, C3.u); // Inside outPort connected to inside inPort
  connect(C3.y, M1.y); // Inside outPort connected to outside outPort
  connect(M1.y, C4.u); // Normal connection of outPort to inPort
end MInst;
```



Parameterization and Extension of Interfaces

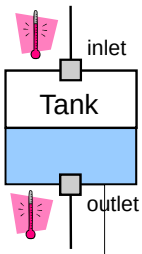


External interfaces to component classes are defined primarily through the use of connectors.

```
model Tank
  parameter Real Area=1;
  replaceable connector TankStream = Stream;
  TankStream inlet, outlet; // The connectors
  Real level;
equation
  // Mass balance
  Area*der(level) = inlet.volumeFlowRate +
     outlet.volumeFlowRate;
  outlet.pressure = inlet.pressure;
end Tank;
connector Stream    // Connector class
  Real pressure;
  flow Real volumeFlowRate;
end Stream
```

The `Tank` model has an external interface in terms of the connectors `inlet` and `outlet`

Parameterization and Extension of Interfaces – cont'



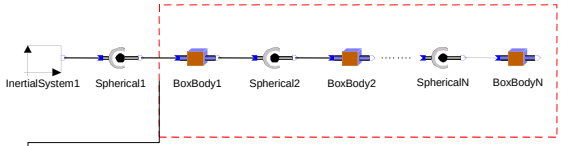
We would like to extend the Tank model to include temperature-dependent effects, analogous to how we extended a resistor to a temperature-dependent resistor

```
model HeatTank
  extends Tank(redeclare connector TankStream = HeatStream);
  Real temp;
equation
  // Energy balance for temperature effects
  Area*level*der(temp) = inlet.volumeFlowRate*inlet.temp +
                         outlet.volumeFlowRate*outlet.temp;
  outlet.temp = temp;   // Perfect mixing assumed.
end HeatTank;
```

```
connector HeatStream
  extends Stream;
  Real temp;
end HeatStream;
```


Arrays of Connectors

Part built up with a for-equation (see Lecture 4)

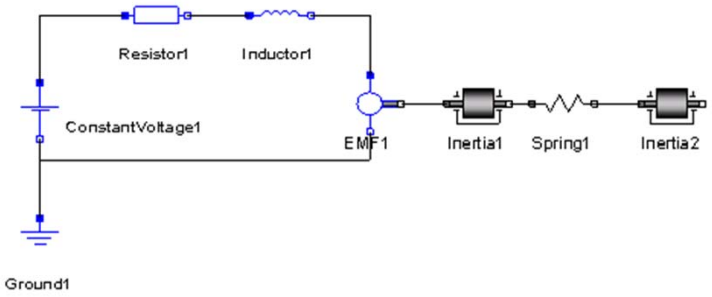


The model uses a for-equation to connect the different segments of the links

```
model ArrayOfLinks
   constant Integer n=10 "Number of segments (>0)";
   parameter Real[3,n] r={fill(1,n),zeros(n),zeros(n)};
   ModelicaAdditions.MultiBody.Parts.InertialSystem InertialSystem1;
   ModelicaAdditions.MultiBody.Parts.BoxBody[n]
              boxBody(r = r, Width=fill(0.4,n));
   ModelicaAdditions.MultiBody.Joints.Spherical spherical[n];
equation
   connect(InertialSystem1.frame_b, spherical[1].frame_a);
   connect(spherical[1].frame_b, boxBody[1].frame_a);
   for i in 1:n-1 loop
      connect(boxBody[i].frame_b, spherical[i+1].frame_a);
      connect(spherical[i+1].frame_b, boxBody[i+1].frame_a);
   end for;
end ArrayOfLinks;
```

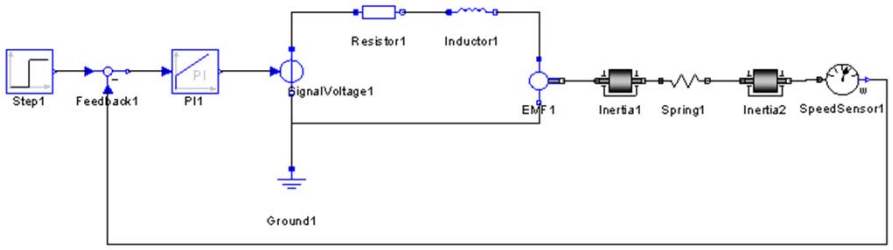

Exercise 3.2

- 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 simForge.



Equations

Usage of Equations

In Modelica equations are used for many tasks

- The main usage of equations is to represent relations in mathematical models.
- *Assignment statements* in conventional languages are usually represented as equations in Modelica
- *Attribute assignments* are represented as equations
- Connections between objects generate equations

Equation Categories

Equations in Modelica can informally be classified into three different categories

- *Normal equations* (e.g., *expr1* = *expr2*) occurring in equation sections, including `connect` equations and other equation types of special syntactic form
- *Declaration equations*, (e.g., Real x = 2.0) which are part of variable, parameter, or constant declarations
- *Modifier equations*, (e.g. x(unit="V"))which are commonly used to modify attributes of classes.

Constraining Rules for Equations

Single Assignment Rule

The total number of “equations” is identical to the total number of “unknown” variables to be solved for

Synchronous Data Flow Principle

- All variables keep their actual values until these values are explicitly changed
- At every point in time, during “continuous integration” and at event instants, the *active* equations express relations between variables which have to be fulfilled *concurrently*
  Equations are not active if the corresponding `if`-branch or `when`-equation in which the equation is present is not active because the corresponding branch condition currently evaluates to `false`
- Computation and communication at an event instant does not take time

Declaration Equations

Declaration equations:

```
constant Integer one = 1;
parameter Real mass = 22.5;
```

It is also possible to specify a declaration equation for a normal non-constant variable:

```
Real speed = 72.4;
```

declaration equations

```
model MoonLanding
  parameter Real force1 = 36350;
  parameter Real force2 = 1308;
  parameter Real thrustEndTime = 210;
  parameter Real thrustDecreaseTime = 43.2;
  Rocket         apollo(name="apollo13", mass(start=1038.358) );
  CelestialBody  moon(mass=7.382e22,radius=1.738e6,name="moon");
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 Landing;
```


Modifier Equations

Modifier equations occur for example in a variable declaration when there is a need to modify the default value of an attribute of the variable
A common usage is modifier equations for the start attribute of variables

```
Real speed(start=72.4);
```

Modifier equations also occur in type definitions:

```
type Voltage = Real(unit="V", min=-220.0, max=220.0);
```

modifier equations

```
model MoonLanding
  parameter Real force1 = 36350;
  parameter Real force2 = 1308;
  parameter Real thrustEndTime = 210;
  parameter Real thrustDecreaseTime = 43.2;
  Rocket         apollo(name="apollo13", mass(start=1038.358) );
  CelestialBody  moon(mass=7.382e22,radius=1.738e6,name="moon");
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 Landing;
```

Kinds of Normal Equations in Equation Sections

Kinds of equations that can be present in equation sections:

- equality equations
- `connect` equations
- `assert` and `terminate`
- `reinit`
- repetitive equation structures with `for`-equations
- conditional equations with `if`-equations
- conditional equations with `when`-equations

conditional if-equation

equality equation

```
model MoonLanding
  parameter Real force1 = 36350;
  parameter Real force2 = 1308;
  parameter Real thrustEndTime = 210;
  parameter Real thrustDecreaseTime = 43.2;
  Rocket         apollo(name="apollo13", mass(start=1038.358) );
  CelestialBody  moon(mass=7.382e22,radius=1.738e6,name="moon");
equation
  if (time<thrustDecreaseTime) then
    apollo.thrust = force1;
  elseif (time<thrustEndTime) then
    apollo.thrust = force2;
  else
    apollo.thrust = 0;
  end if;
  apollo.gravity=moon.g*moon.mass/(apollo.altitude+moon.radius)^2;
end Landing;
```


Equality Equations

```
expr1 = expr2:
(out1, out2, out3,...) = function_name(in_expr1, in_expr2, ...);
```

simple equality equation

```
class EqualityEquations
  Real x,y,z;
equation
  (x, y, z)       = f(1.0, 2.0); // Correct!
  (x+1, 3.0, z/y) = f(1.0, 2.0); // Illegal!
                                 // Not a list of variables
                                 // on the left-hand side
end EqualityEquations;
```

Repetitive Equations

The syntactic form of a `for`-equation is as follows:

```
for <iteration-variable> in <iteration-set-expression> loop
    <equation1>
    <equation2>
    ...
end for;
```

Consider the following simple example with a `for`-equation:

```
class FiveEquations
  Real[5]  x;
equation
  for i in 1:5 loop
    x[i] = i+1;
  end for;
end FiveEquations;
```

Both classes have equivalent behavior!

```
class FiveEquationsUnrolled
  Real[5]  x;
equation
  x[1] = 2;
  x[2] = 3;
  x[3] = 4;
  x[4] = 5;
  x[5] = 6;
end FiveEquationsUnrolled;
```

In the class on the right the `for`-equation has been unrolled into five simple equations

`connect`-equations

In Modelica `connect`-equations are used to establish connections between components via connectors

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

Repetitive `connect`-equations

```
class RegComponent
  Component components[n];
equation
  for i in 1:n-1 loop
    connect(components[i].outlet,components[i+1].inlet);
  end for;
end RegComponent;
```

Conditional Equations: `if`-equations

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

if-equations for which the conditions have higher variability than constant or parameter must include an *else-part*

Each `then`-, `elseif`-, and `else`-branch must have the *same number of equations*

```
model MoonLanding
  parameter Real force1 = 36350;
  ...
  Rocket         apollo(name="apollo13", mass(start=1038.358) );
  CelestialBody  moon(mass=7.382e22,radius=1.738e6,name="moon");
equation
  if (time<thrustDecreaseTime) then
    apollo.thrust = force1;
  elseif (time<thrustEndTime) then
    apollo.thrust = force2;
  else
    apollo.thrust = 0;
  end if;
  apollo.gravity=moon.g*moon.mass/(apollo.altitude+moon.radius)^2;
end Landing;
```

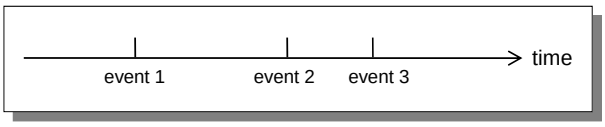

Conditional Equations: `when`-equations

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

```
when x > 2 then
  y1 = sin(x);
  y3 = 2*x + y1+y2;
end when;
```

<*equations*> in when-equations are instantaneous equations that are active at events when <*conditions*> become true

Events are ordered in time and form an event history:



- An event is a *point* in time that is instantaneous, i.e., has zero duration
- An *event condition* switches from false to true in order for the event to take place

Conditional Equations: when-equations cont'

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

when-equations are used to express instantaneous equations that are only valid (become active) *at events*, e.g. at discontinuities or when certain conditions become true

```
when x > 2 then
  y1 = sin(x);
  y3 = 2*x + y1+y2;
end when;
```

```
when {x > 2, sample(0,2), x < 5} then
  y1 = sin(x);
  y3 = 2*x + y1+y2;
end when;
```

```
when initial() then
  ...   // Equations to be activated at the beginning of a simulation
end when;
...
when terminal() then
  ...   // Equations to be activated at the end of a simulation
end when;
```


Restrictions on when-equations

Form restriction

```
model WhenNotValid
  Real x, y;
equation
  x + y = 5;
  when sample(0,2) then
    2*x + y = 7;
    // Error: not valid Modelica
  end when;
end WhenNotValid;
```

Modelica restricts the allowed equations within a when-equation to: variable = expression, if-equations, for-equations,...

In the `WhenNotValid` model when the equations within the when-equation are not active it is not clear which variable, either x or y, that is a "result" from the when-equation to keep constant outside the when-equation.

A corrected version appears in the class `WhenValidResult` below

```
model WhenValidResult
  Real x,y;
equation
  x + y = 5;           // Equation to be used to compute x.
  when sample(0,2) then
    y = 7 - 2*x;       // Correct, y is a result variable from the when!
  end when;
end WhenValidResult;
```


Restrictions on when-equations cont'

Restriction on nested when-equations

```
model ErrorNestedWhen
  Real x,y1,y2;
equation
  when x > 2 then
    when y1 > 3 then  // Error!
      y2 = sin(x);    // when-equations
    end when;         // should not be nested
  end when;
end ErrorNestedWhen;
```

when-equations cannot be nested!

Restrictions on when-equations cont'

Single assignment rule: same variable may not be defined in several when-equations.

A conflict between the equations will occur if both conditions would become true at the same time instant

```
model DoubleWhenConflict
  Boolean close;   // Error: close defined by two equations!
equation
  ...
  when condition1 then
    close = true;  // First equation
   end when;
  ...
  when condition2 then
    close = false; //Second equation
  end when;
end DoubleWhenConflict
```


Restrictions on when-equations cont'

Solution to assignment conflict between equations in independent when-equations:

- Use elsewhen to give higher priority to the first when-equation

```
model DoubleWhenConflictResolved
  Boolean close;
equation
  ...
  when condition1 then
    close = true;  // First equation has higher priority!
  elsewhen condition2 then
    close = false; //Second equation
  end when;
end DoubleWhenConflictResolved
```


Restrictions on when-equations cont'

Vector expressions

The equations within a when-equation are activated when any of the elements of the vector expression becomes true

```
model VectorWhen
  Boolean close;
equation
  ...
  when {condition1,condition2} then
    close = true;
  end when;
end DoubleWhenConflict
```


`assert`-equations

assert(assert-expression, message-string)

`assert` is a predefined function for giving error messages taking a Boolean condition and a string as an argument

The intention behind `assert` is to provide a convenient means for specifying checks on model validity within a model

```
class AssertTest
  parameter Real lowlimit = -5;
  parameter Real highlimit = 5;
  Real x;
equation
  assert(x >= lowlimit and x <= highlimit,
         "Variable x out of limit");
end AssertTest;
```


`terminate`-equations

The `terminate`-equation successfully terminates the current simulation, i.e. no error condition is indicated

```
model MoonLanding
  parameter Real force1 = 36350;
  parameter Real force2 = 1308;
  parameter Real thrustEndTime = 210;
  parameter Real thrustDecreaseTime = 43.2;
  Rocket    apollo(name="apollo13", mass(start=1038.358) );
  CelestialBody moon(mass=7.382e22,radius=1.738e6,name="moon");
equation
  apollo.thrust = if (time<thrustDecreaseTime) then force1
                  else if (time<thrustEndTime) then force2
                  else 0;
  apollo.gravity = moon.g * moon.mass /(apollo.height + moon.radius)^2;
  when apollo.height < 0 then      // termination condition
    terminate("The moon lander touches the ground of the moon");
  end when;
end MoonLanding;
```

- Exercise03-classes-textual-circuit.onb

Arrays, Algorithms, and Functions

Array Data Structures

- An **array variable** is an ordered collection of scalar variables all of the same type
- **Dimensions** – Each array has a certain **number of dimensions**, a vector has 1 dimension, a matrix 2 dimensions
- Modelica arrays are "**rectangular**", i.e., all rows in a matrix have equal length, and all columns have equal length
- **Construction** – An array is **created** by declaring an array variable or calling an array constructor
- **Indexing** – An array can be **indexed** by Integer, Boolean, or enumeration values
- Lower/higher **bounds** – An array dimension indexed by integers has 1 as lower bound, and the size of the dimension as higher bound

Two forms of Array Variable Declarations

Array declarations with dimensions in the type

2 dimensions

```
Real[3]          positionvector = {1,2,3};
Real[3,3]        identitymatrix = {{1,0,0}, {0,1,0}, {0,0,1}};
Integer[n,m,k]   arr3d;
Boolean[2]       truthvalues = {false,true};
Voltage[10]      voltagevector;
String[3]        str3 = {"one", "two", "blue"};
Real[0,3]        M;     // An empty matrix M
```

Array declarations with dimensions behind the variable

1 dimension

```
Real       positionvector[3] = {1,2,3};
Real       identitymatrix[3,3] = {{1,0,0}, {0,1,0}, {0,0,1}};
Real       arr3d[n,m,k];
Boolean    truthvalues[2] = {false,true};
Voltage    voltagevector[10]
String     str3[3] = {"one", "two", "blue"};
Integer    x[0];       // An empty vector x
```


Flexible Array Sizes

Arrays with unspecified dimension sizes are needed to make flexible models that adapt to different problem sizes

An unspecified Integer dimension size is stated by using a colon (:) instead of the usual dimension expression

Flexible array sizes can only be used in functions and partial models

```
Real[:,:]  Y;              // A matrix with two unknown dimension sizes
Real[:,3]  Y2;             // A matrix where the number of columns is known
Real       M[:,size(M,1)]  // A square matrix of unknown size

Real[:]  v1, v2;           // Vectors v1 and v2 have unknown sizes

Real[:]  v3 = fill(3,14, n);   // A vector v3 of size n filled with 3.14
```

Modifiers for Array Variables

Array variables can be initialized or given start values using modifiers

```
Real A3[2,2];                        // Array variable
Real A4[2,2](start={{1,0},{0,1}}); // Array with modifier
Real A5[2,2](unit={{"Voltage","Voltage"},{"Voltage","Voltage"}});
```

Modification of an indexed element of an array is illegal

```
Real A6[2,2](start[2,1]=2.3);
// Illegal! indexed element modification
```

The **each** operator can give compact initializations

```
record Crec
  Real b[4];
end Crec;
model B
  Crec A7[2,3](b = {
 {{1,2,3,4},{1,2,3,4},{1,2,3,4}},
 {{1,2,3,4},{1,2,3,4},{1,2,3,4}} } );
end B;
```

same A7 as

```
model C
  Crec A7[2,3](each b = {1,2,3,4});
end C;
```


Array Constructors { } and Range Vectors

An array constructor is just a function accepting scalar or array arguments and returning an array result. The array constructor function array(A,B,C,...), with short-hand notation {A, B, C, ...}, constructs an array from its arguments

```
{1,2,3}                    // A 3-vector of type Integer[3]
array(1.0,2.0,3)           // A 3-vector of type Real[3]
{{11,12,13}, {21,22,23}}   // A 2x3 matrix of type Integer[2,3]
{{{1.0, 2.0, 3.0}}}        // A 1x1x3 array of type Real[1,1,3]
{ {1}, {2,3} }             // Illegal, arguments have different size
```

Range vector constructor

Two forms: *startexpr* : *endexpr* or *startexpr* : *deltaexpr* : *endexpr*

```
Real v1[5] = 2.7 : 6.8;                // v1 is {2.7, 3.7, 4.7, 5.7, 6.7}
Real v2[5] = {2.7, 3.7, 4.7, 5.7, 6.7}; // v2 is equal to v1
Integer v3[3] = 3 : 5;                 // v3 is {3,4,5}
Integer v4empty[0] = 3 : 2             // v4empty is an empty Integer vector
Real    v5[4]  = 1.0 : 2  : 8;         // v5 is {1.0,3.0,5.0,7.0}
Integer v6[5]  = 1   : -1 : -3;        // v6 is {1,0,-1,-2,-3}
```


Set Formers – Array Constructors with Iterators

Mathematical notation for creation of a set of expressions, e.g. A = {1, 3, 4, 6...}

{ $expr_i$ | i ε A} equivalent to { $expr_1$, $expr_3$, $expr_4$, $expr_6$, ... }

Modelica has array constructors with iterators, single or multiple iterators

{ $expr_i$ **for** i **in** A} { $expr_{ij}$ **for** i in A, j **in** B }

If only one iterator is present, the result is a vector of values constructed by evaluating the expression for each value of the iterator variable

```
{r for r in 1.0 : 1.5 : 5.5} // The vector 1.0:1.5:5.5={1.0, 2.5, 4.0, 5.5}
{i^2 for i in {1,3,7,6}}     // The vector {1, 9, 49, 36}
```

Multiple iterators in an array constructor

```
 {(1/(i+j-1) for i in 1:m, j in 1:n};        // Multiple iterators
{{(1/(i+j-1) for j in 1:n} for i in 1:m}     // Nested array constructors
```

Deduction of range expression, can leave out e.g. `1:size(x,1)`

```
Real s1[3] = {x[i]*3 for i in 1:size(x,1)};  // result is {6,3,12}
Real s2[3] = {x[i] for i};    // same result, range deduced to be 1:3
```


Array Concatenation and Construction

General array concatenation can be done through the array concatenation operator cat(k,A,B,C,...) that concatenates the arrays A,B,C,... along the kth dimension

```
cat(1, {1,2}, {10,12,13} )           // Result: {1,2,10,12,13}
cat(2, {{1,2},{3,4}}, {{10},(11}} )  // Result: {{1,2,10},{3,4,11}}
```

The common special cases of concatenation along the first and second dimensions are supported through the special syntax forms [A;B;C;...] and [A,B,C,...] respectively, that also can be mixed

Scalar and vector arguments to these special operators are promoted to become matrices before concatenation – this gives MATLAB compatibility

```
[1,2; 3,4]                    // Result: {{1,2}, {3,4}}
[ [1,2; 3,4], [10; 11] ]      // Result: [1,2,10; 3,4,11]
cat(2, [1,2; 3,4], [10; 11] )  // Same result: {{1,2,10}, {3,4,11}}
```


Array Indexing

The array indexing operator ...[...] is used to access array elements for retrieval of their values or for updating these values

arrayname **[** ***indexexpr1***, ***indexexpr2***, ...**]**

```
Real[2,2] A = {{2,3},{4,5}};  // Definition of Array A
A[2,2]                        // Retrieves the array element value 5
A[1,2] := ...                 // Assignment to the array element A[1,2]
```

Arrays can be indexed by integers as in the above examples, or Booleans and enumeration values, or the special value **end**

```
type Sizes = enumeration(small, medium);
Real[Sizes]   sz = {1.2, 3.5};
Real[Boolean] bb = {2.4, 9.9};

sz[Sizes.small]    // Ok, gives value 1.2
bb[true]           // Ok, gives value 9.9
A[end-1,end]       // Same as:  A[size(A,1)-1,size(A,2)]
A[v[end],end]      // Same as:  A[v[size(v,1)],size(A,2)]
```


Array Addition, Subtraction, Equality and Assignment

Element-wise addition and subtraction of scalars and/or arrays can be done with the (+), (.+), and (-), (.-) operators. The operators (.+) and (.-) are defined for combinations of scalars with arrays, which is not the case for (+) and (-)

```
{1,2,3} + 1                    // Not allowed!
{1,2,3} - {1,2,0}              // Result: {0,0,3}
{1,2,3} + {1,2}                // Not allowed, different array sizes!
{{1,1},{2,2}} + {{1,2},{3,4}}  // Result: {{2,3},{5,6}}
```

Element-wise addition (.+) and element-wise subtraction (.□)

```
{1,2,3} .+ 1                    // Result: {2,3,4}
1 .+ {1,2,3}                    // Result: {2,3,4}
{1,2,3} .- {1,2,0}              // Result: {0,0,3}
{1,2,3} .+ {1,2}                // Not allowed, different array sizes!
{{1,1},{2,2}} .+ {{1,2},{3,4}}  // Result: {{2,3},{5,6}}
```

Equality (array equations), and array assignment

```
v1 = {1,2,3};
v2 := {4,5,6};
```

Array Multiplication

The linear algebra multiplication operator (*) is interpreted as scalar product or matrix multiplication depending on the dimensionality of its array arguments.

```
{1,2,3} * 2                      // Elementwise mult: {2,4,6}
3 * {1,2,3}                      // Elementwise mult: {3,6,9}
{1,2,3} * {1,2,2}                // Scalar product:   11

{{1,2},{3,4}} * {1,2}            // Matrix mult:      {5,11}
{1,2,3} * {{1},{2},{10}}         // Matrix mult:      {35}
{1,2,3} * [1;2;10]               // Matrix mult:      {35}
```

Element-wise multiplication between scalars and/or arrays can be done with the (*) and (.*) operators. The (.*) operator is equivalent to (*) when both operands are scalars.

```
{1,2,3} .* 2                     // Result: {2,4,6}
2 .* {1,2,3}                     // Result: {2,4,6}
{2,4,6} .* {1,2,2}               // Result: {2,8,12}
{1,2,3} .* {1,2}                 // Not allowed, different array sizes!
```


Array Dimension and Size Functions

An array reduction function “reduces” an array to a scalar value, i.e., computes a scalar value from the array

ndims(A)	Returns the number of dimensions k of array A, with k >= 0.
size(A,i)	Returns the size of dimension i of array A where 1 <= i <= ndims(A).
size(A)	Returns a vector of length ndims(A) containing the dimension sizes of A.

```
Real[4,1,6] x;  // declared array x

ndims(x);       // returns 3 – no of dimensions
size(x,1);      // returns 4 – size of 1st dimension
size(x);        // returns the vector {4, 1, 6}
size(2*x+x) = size(x);  // this equation holds
```


Array Reduction Functions and Operators

An array reduction function “reduces” an array to a scalar value, i.e., computes a scalar value from the array. The following are defined:

min(A)	Returns the smallest element of array A.
max(A)	Returns the largest element of array A.
sum(A)	Returns the sum of all the elements of array A.
product(A)	Returns the product of all the elements of array A.

```
min({1,-1,7})                    // Gives the value -1
max([1,2,3; 4,5,6])              // Gives the value 6
sum({{1,2,3},{4,5,6}})           // Gives the value 21
product({3.14, 2, 2})            // Gives the value 12.56
```

Reduction functions with iterators

```
min(i^2 for i in {1,3,7})     // min(min(1, 9), 49)) = 1
max(i^2 for i in {1,3,7})     // max(max(1, 9), 49)) = 49
sum(i^2 for i in {1,3,7,5})
```


Vectorization of Function Calls with Array Arguments

Modelica functions with one scalar return value can be applied to arrays elementwise, e.g. if v is a vector of reals, then sin(v) is a vector where each element is the result of applying the function sin to the corresponding element in v

```
sin({a,b,c})      // Vector argument, result: {sin(a),sin(b),sin(c)}
sin([1,2; 3,4])   // Matrix argument, result: [sin(1),sin(2);
sin(3),sin(4)]
```

Functions with more than one argument can be generalized/vectorized to elementwise application. All arguments must be the same size, traversal in parallel

```
atan2({a,b,c},{d,e,f})   // Result: {atan2(a,d), atan2(b,e), atan2(c,f)}
```


Algorithms and Statements

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-statements 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
  <statementss>
end if
```

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

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

```
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;
```


`when`-statements

```
when <conditions> then
  <statements>
elsewhen <conditions> then
  <statements>
end when;
```

when-statements are used to express actions (statements) that are only executed at events, e.g. at discontinuities or when certain conditions become true

There are situations where several assignment statements within the same when-statement is convenient

```
when x > 2 then
  y1 := sin(x);
  y3 := 2*x + y1 + y2;
end when;
```

```
when {x > 2, sample(0,2), x < 5} then
  y1 := sin(x);
  y3 := 2*x + y1 + y2;
end when;
```

```
algorithm
  when x > 2 then
    y1 := sin(x);
  end when;
equation
  y2 = sin(y1);
algorithm
  when x > 2 then
    y3 := 2*x + y1 + y2;
  end when;
```

Algorithm and equation sections can be interleaved.

Functions

Function Declaration

The structure of a typical function declaration is as follows:

```
function <functionname>
  input TypeI1 in1;
  input TypeI2 in2;
  input TypeI3 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)
```

Functions with Multiple Results

```
function PointOnCircle"Computes cartesian coordinates of point"
  input  Real angle  "angle in radians";
  input  Real radius;
  output Real x;    // 1:st result formal parameter
  output Real y;    // 2:nd result formal parameter
algorithm
  x := radius * cos(phi);
  y := radius * sin(phi);
end PointOnCircle;
```

Example calls:

```
(out1,out2,out3,...) = function_name(in1, in2, in3, in4, ...);  // Equation
(out1,out2,out3,...) := function_name(in1, in2, in3, in4, ...); // Statement

(px,py)  = PointOnCircle(1.2, 2);   // Equation form
(px,py) := PointOnCircle(1.2, 2);   // Statement form
```

Any kind of variable of compatible type is allowed in the parenthesized list on the left hand side, e.g. even array elements:

```
(arr[1],arr[2]) := PointOnCircle(1.2, 2);
```


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;
```


- Exercise04-equations-algorithms-functions.onb

Packages

Packages for Avoiding Name Collisions

- Modelica provide a safe and systematic way of avoiding name collisions through the package concept
- A package is simply a container or name space for names of classes, functions, constants and other allowed definitions

Packages as Abstract Data Type: Data and Operations in the Same Place

Keywords denoting a package

encapsulated makes package dependencies (i.e., imports) explicit

Declarations of substract, divide, realPart, imaginaryPart, etc are not shown here

```
encapsulated package ComplexNumber

  record Complex
    Real re;
    Real im;
  end Complex;

  function add
    input  Complex x,y;
    output Complex z;
  algorithm
    z.re := x.re + y.re;
    z.im := x.im + y.im
  end add;

  function multiply
    input  Complex x,y;
    output Complex z;
  algorithm
    z.re := x.re*y.re – x.im*y.im;
    z.im := x.re*y.im + x.im*y.re;
  end multiply;
  ……..
end ComplexMumbers
```

Usage of the `ComplexNumber` package

```
class ComplexUser
  ComplexNumbers.Complex  a(re=1.0, im=2.0);
  ComplexNumbers.Complex  b(re=1.0, im=2.0);
  ComplexNumbers.Complex  z,w;
equation
  z = ComplexNumbers.multiply(a,b);
  w = ComplexNumbers.add(a,b);
end ComplexUser
```

The type `Complex` and the operations `multiply` and `add` are referenced by prefixing them with the package name `ComplexNumber`

Accessing Definitions in Packages

- Access reference by prefixing the package name to definition names

```
class ComplexUser
  ComplexNumbers.Complex  a(re=1.0, im=2.0);
  ComplexNumbers.Complex  b(re=1.0, im=2.0);
  ComplexNumbers.Complex  z,w;
equation
  z = ComplexNumbers.multiply(a,b);
  w = ComplexNumbers.add(a,b);
end ComplexUser
```

- Shorter access names (e.g. `Complex`, `multiply`) can be used if definitions are first imported from a package (see next page).

Importing Definitions from Packages

- Qualified import
- Single definition import
- Unqualified import
- Renaming import

```
import <packagename>
import <packagename> . <definitionname>
import <packagename> . *
import <shortpackagename> = <packagename>
```

The four forms of import are exemplified below assuming that we want to access the addition operation (`add`) of the package `Modelica.Math.ComplexNumbers`

```
import Modelica.Math.ComplexNumbers;      //Access as ComplexNumbers.add
import Modelica.Math.ComplexNumbers.add;  //Access as add
import Modelica.Math.ComplexNumbers.*     //Access as add
import Co = Modelica.Math.ComplexNumbers  //Access as Co.add
```


Qualified Import

Qualified import

```
import <packagename>
```

The *qualified import* statement
`import` *`<packagename>;`*
imports all definitions in a package, which subsequently can be referred to by (usually shorter) names
`simplepackagename . definitionname`, where the simple package name is the *`packagename`* without its prefix**.**

```
encapsulated package ComplexUser1
  import Modelica.Math.ComplexNumbers;
  class User
    ComplexNumbers.Complex  a(x=1.0, y=2.0);
    ComplexNumbers.Complex  b(x=1.0, y=2.0);
    ComplexNumbers.Complex  z,w;
  equation
    z = ComplexNumbers.multiply(a,b);
    w = ComplexNumbers.add(a,b);
  end User;
end ComplexUser1;
```

This is the most common form of `import` that eliminates the risk for name collisions when importing from several packages

Single Definition Import

Single definition import ← `import <packagename> . <definitionname>`

The *single definition import* of the form
`import <packagename>.<definitionname>;`
allows us to import a single specific definition (a constant or class but not a subpackage) from a package and use that definition referred to by its *definitionname* without the package prefix

```
encapsulated package ComplexUser2
  import ComplexNumbers.Complex;
  import ComplexNumbers.multiply;
  import ComplexNumbers.add;
 class User
  Complex  a(x=1.0, y=2.0);
  Complex  b(x=1.0, y=2.0);
  Complex  z,w;
 equation
  z = multiply(a,b);
  w = add(a,b);
 end User;
end ComplexUser2;
```

There is no risk for name collision as long as we do not try to import two definitions with the same short name

Unqualified Import

Unqualified import ← `import <packagename> . *`

The unqualified import statement of the form
`import packagename.*;`
imports all definitions from the package using their short names without qualification prefixes.
Danger: Can give rise to name collisions if imported package is changed.

```
class ComplexUser3
  import ComplexNumbers.*;
  Complex  a(x=1.0, y=2.0);
  Complex  b(x=1.0, y=2.0);
  Complex  z,w;
equation
  z = multiply(a,b);
  w = add(a,b);
end ComplexUser3;
```

This example also shows direct `import` into a class instead of into an enclosing package

Renaming Import

Renaming import ← `import <shortpackagename> = <packagename>`

The *renaming import* statement of the form:
`import <shortpackagename> = <packagename>;`
imports a package and renames it locally to *shortpackagename*. One can refer to imported definitions using *shortpackagename* as a presumably shorter package prefix**.**

```
class ComplexUser4
  import Co = ComplexNumbers;
  Co.Complex  a(x=1.0, y=2.0);
  Co.Complex  b(x=1.0, y=2.0);
  Co.Complex  z,w;
equation
  z = Co.multiply(a,b);
  w = Co.add(a,b);
end ComplexUser4;
```

This is as safe as qualified import but gives more concise code

Package and Library Structuring

A well-designed package structure is one of the most important aspects that influences the complexity, understandability, and maintainability of large software systems. There are many factors to consider when designing a package, e.g.:

- The name of the package.
- Structuring of the package into subpackages.
- Reusability and encapsulation of the package.
- Dependencies on other packages.

Subpackages and Hierarchical Libraries

The main use for Modelica packages and subpackages is to structure hierarchical model libraries, of which the standard Modelica library is a good example.

```
encapsulated package Modelica         // Modelica
   encapsulated package Mechanics     // Modelica.Mechanics
      encapsulated package Rotational // Modelica.Mechanics.Rotational
          model Inertia // Modelica.Mechanics.Rotational.Inertia
           ...
          end Inertia;
          model Torque  // Modelica.Mechanics.Rotational.Torque
           ...
          end Torque;
          ...
     end Rotational;
   ...
end Mechanics;
...
end Modelica;
```


Ecapsulated Packages and Classes

An encapsulated package or class *prevents* direct reference to public definitions *outside* itself, but as usual allows access to public subpackages and classes inside itself.

- Dependencies on other packages become explicit
  – more readable and understandable models!
- Used packages from outside must be *imported*.

```
encapsulated model TorqueUserExample1
   import Modelica.Mechanics.Rotational;  // Import package Rotational
   Rotational.Torque  t2;                 // Use Torque, OK!
   Modelica.Mechanics.Rotational.Inertia  w2;
        //Error! No direct reference to the top-level Modelica package
   ...   // to outside an encapsulated class
end TorqueUserExample1;
```


`within` Declaration for Package Placement

Use *short names* without dots when declaring the package or class in question, e.g. on a separate file or storage unit. Use `within` to specify within which package it is to be placed.

The within declaration states the *prefix* needed to form the fully qualified name

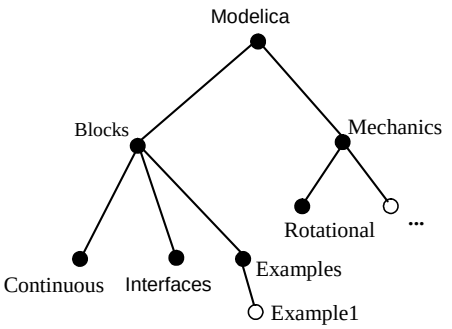
```
within Modelica.Mechanics;
encapsulated package Rotational // Modelica.Mechanics.Rotational
  encapsulated package Interfaces
     import ...;
     connector Flange_a;
        ...
     end Flange_a;
     ...
  end Interfaces;
  model Inertia
     ...
  end Inertia;
  ...
end Rotational;
```

The subpackage `Rotational` declared within `Modelica.Mechanics` has the fully qualified name `Modelica.Mechanics.Rotational`, by concatenating the *packageprefix* with the *short name* of the package.

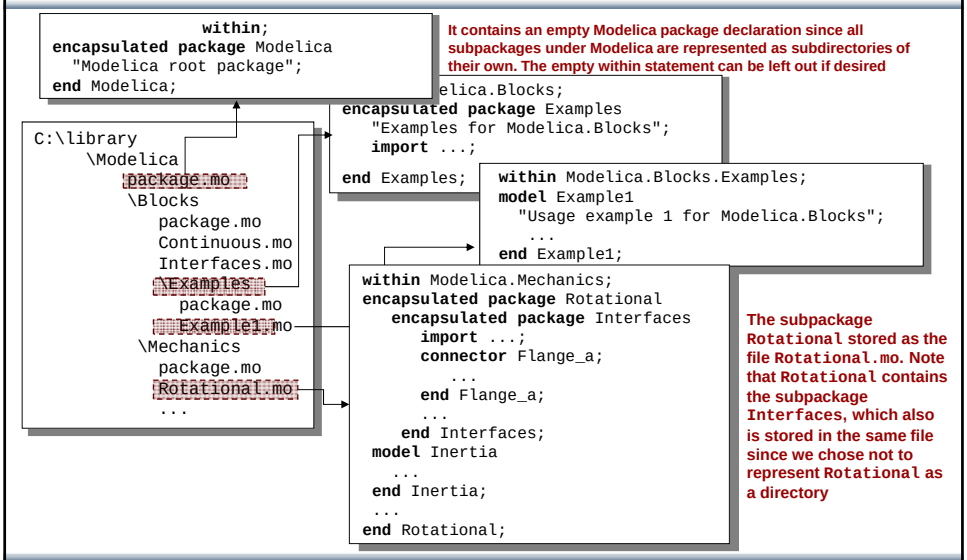
Mapping a Package Hierachy into a Directory Hirarchy

A Modelica package hierarchy can be mapped into a corresponding directory hierarchy in the file system

```
C:\library
     \Modelica
         package.mo
         \Blocks
            package.mo
            Continuous.mo
            Interfaces.mo
            \Examples
              package.mo
              Example1.mo
          \Mechanics
            package.mo
            Rotational.mo
            ...
```



Mapping a Package Hierachy into a Directory Hirarchy



Modelica Libraries

Modelica Standard Library

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*.

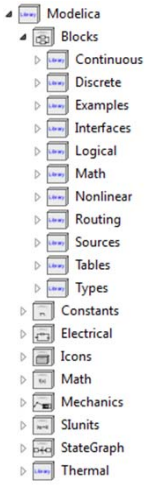
Modelica libraries are available online including documentation and source code from http://www.modelica.org/library/library.html.

Modelica Standard Library cont'

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

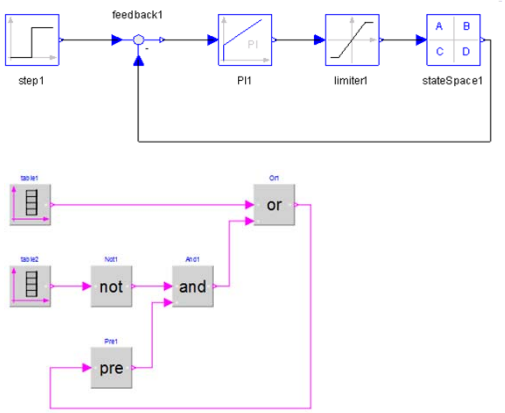
- Blocks — Library for basic input/output control blocks
- Constants — Mathematical constants and constants of nature
- Electrical — Library for electrical models
- Icons — Icon definitions
- Fluid — 1-dim Flow in networks of vessels, pipes, fluid machines, valves, etc.
- Math — Mathematical functions
- Magnetic — Magnetic.Fluxtubes – for magnetic applications
- Mechanics — Library for mechanical systems
- Media — Media models for liquids and gases
- SIunits — 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



Continuous, discrete, and logical input/output blocks to build block diagrams.

Examples:



Modelica.Constants

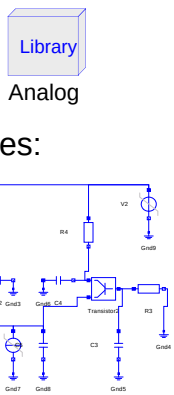
A package with often needed constants from mathematics, machine dependent constants, and constants of nature.

Examples:

```
constant Real pi=2*Modelica.Math.asin(1.0);

constant Real small=1.e-60 "Smallest number such that small and –small
                            are representable on the machine";

constant Real G(final unit="m3/(kg.s2)") = 6.673e-11 "Newtonian constant
                                                      of gravitation";

constant Real h(final unit="J.s") = 6.62606876e-34 "Planck constant";

constant Modelica.SIunits.CelsiusTemperature T_zero=-273.15 "Absolute
                                                    zero temperature”;
```


Modelica.Electrical

Electrical components for building analog, digital, and multiphase circuits



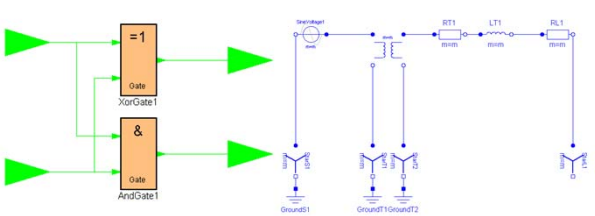
Library
Analog

Library
Digital

Library
Machines

Library
MultiPhase

Examples:



Modelica.Icons

Package with icons that can be reused in other libraries

Examples:

Info

Library1

Library2

Example


RotationalSensor


TranslationalSensor


GearIcon


MotorIcon

Modelica.Math

Package containing basic mathematical functions:

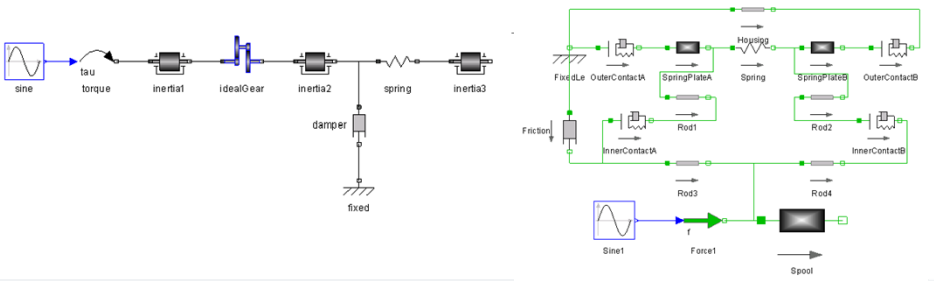
sin(*u*)	sine
cos(*u*)	cosine
tan(*u*)	tangent (*u* shall not be: …,-π/2, π/2, 3π/2,…)
asin(*u*)	inverse sine ($-1 \le u \le 1$)
acos(*u*)	inverse cosine ($-1 \le u \le 1$)
atan(*u*)	inverse tangent
atan2(*u1*, *u2*)	four quadrant inverse tangent
sinh(*u*)	hyperbolic sine
cosh(*u*)	hyperbolic cosine
tanh(*u*)	hyperbolic tangent
exp(*u*)	exponential, base *e*
log(*u*)	natural (base *e*) logarithm ($u > 0$)
log10(*u*)	base 10 logarithm ($u > 0$)

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



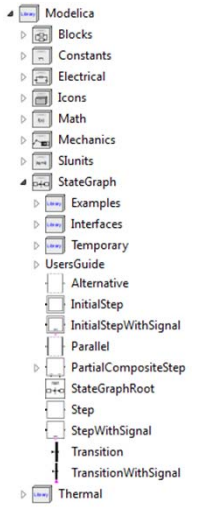
Modelica.SIunits

This package contains predefined types based on the international standard of units:

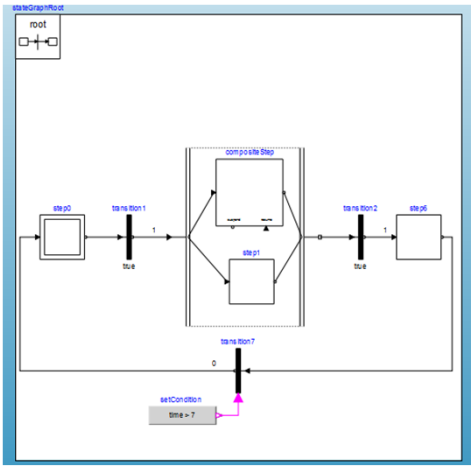
- ISO 31-1992 “General principles concerning quantities, units and symbols”
- ISO 1000-1992 “SI units and recommendations for the use of their multiples and of certain other units”.

A subpackage called `NonSIunits` is available containing non SI units such as `Pressure_bar`, `Angle_deg`, etc

Modelica.Stategraph



Hierarchical state machines (similar to Statecharts)

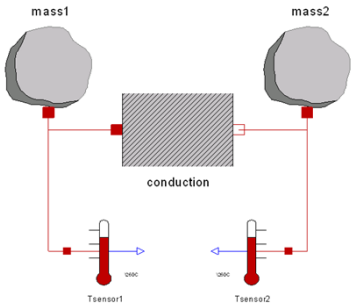


Modelica.Thermal

Subpackage `FluidHeatFlow` with components for heat flow modeling.

Sub package `HeatTransfer` with components to model 1-dimensional heat transfer with lumped elements

Example:



ModelicaAdditions Library (OLD)

`ModelicaAdditions` library contains additional Modelica libraries from DLR. This has been largely replaced by the new release of the Modelica 3.1 libraries.

Sublibraries:

- Blocks Input/output block sublibrary
- HeatFlow1D 1-dimensional heat flow (replaced by `Modelica.Thermal`)
- Multibody Modelica library to model 3D mechanical systems
- PetriNets Library to model Petri nets and state transition diagrams
- Tables Components to interpolate linearly in tables

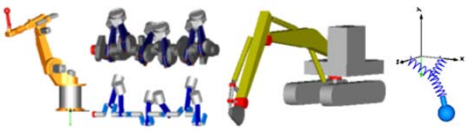
ModelicaAdditions.Multibody (OLD)

This is a Modelica library to model 3D Mechanical systems including visualization

New version has been released (march 2004) that is called Modelica.Mechanics.MultiBody in the standard library

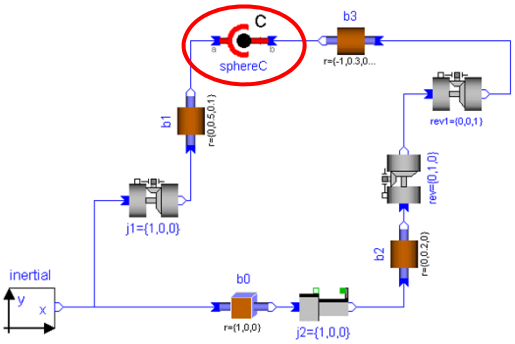
Improvements:

- Easier to use
- Automatic handling of kinematic loops.
- Built-in animation properties for all components

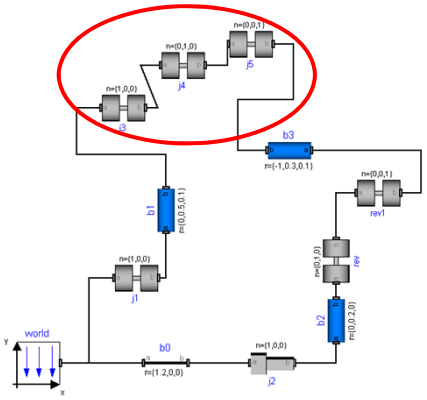


MultiBody (MBS) - Example Kinematic Loop

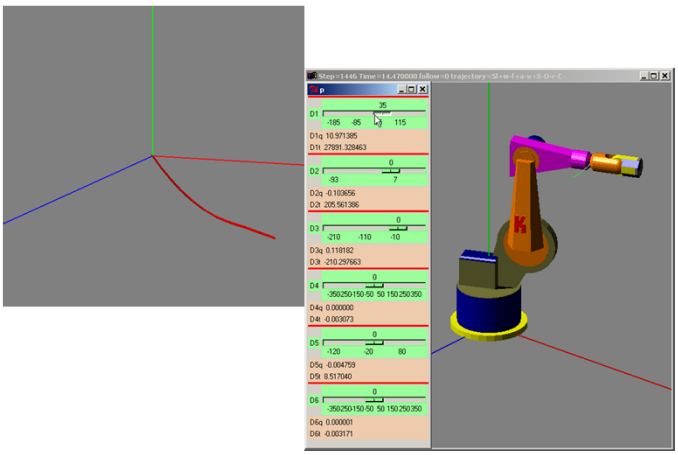
Old library
(cutjoint needed)



New library
(no cutjoint needed)



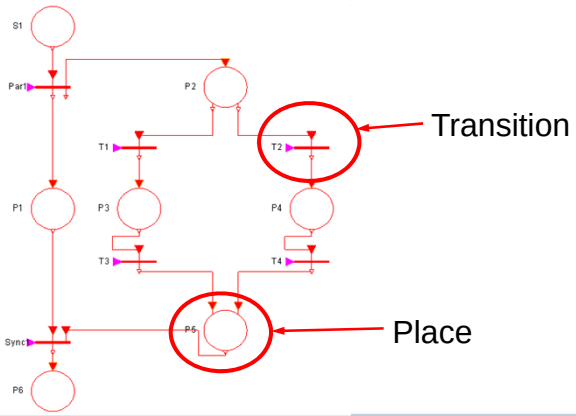
MultiBody (MBS) - Example Animations



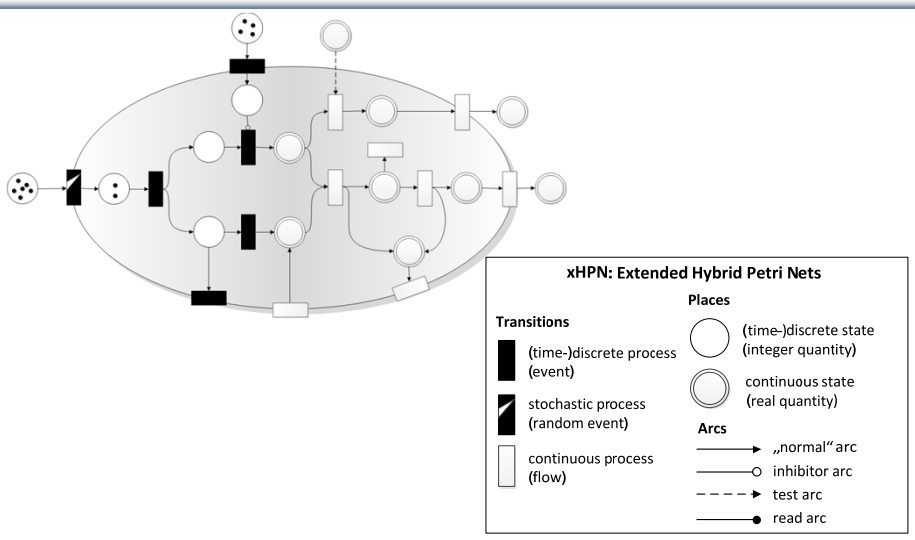
ModelicaAdditions.PetriNets (OLD)

This package contains components to model Petri nets

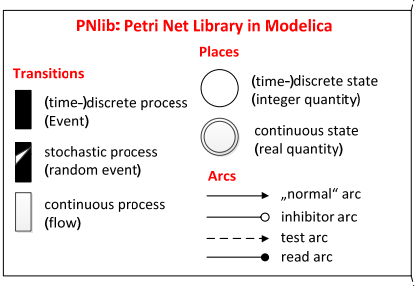
Used for modeling of computer hardware, software, assembly lines, etc



PNlib - An Advanced Petri Net Library for Hybrid Process Modeling



PNLib Cont



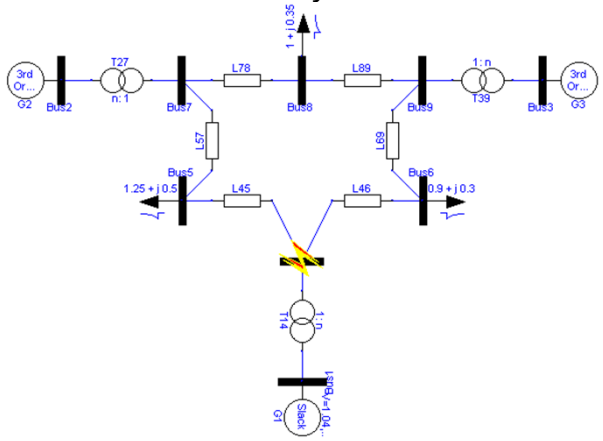

```
model PetriNetComponent1
   parameter1;
   parameter2;
    …
   variable1;
   variable2;
    …
equation
   equation1;
   equation2;
   …
//event-based equations
   when condition then
    …
   end when;
//differential equations
   der(t) = rightSide;
//algebraic equations
   a+b=c;
algorithm
   statement1;
   statement2;
   …
end PetriNetComponent1 ;
```


Other Free Libraries

- WasteWater — Wastewater treatment plants, 2003
- ATPlus — Building simulation and control (fuzzy control included), 2005
- MotorCycleDymanics — Dynamics and control of motorcycles, 2009
- NeuralNetwork — Neural network mathematical models, 2006
- VehicleDynamics — Dynamics of vehicle chassis (obsolete), 2003
- SPICElib — Some capabilities of electric circuit simulator PSPICE, 2003
- SystemDynamics — System dynamics modeling a la J. Forrester, 2007
- BondLib — Bond graph modeling of physical systems, 2007
- MultiBondLib — Multi bond graph modeling of physical systems, 2007
- ModelicaDEVS — DEVS discrete event modeling, 2006
- ExtendedPetriNets — Petri net modeling, 2002
- External.Media Library — External fluid property computation, 2008
- VirtualLabBuilder — Implementation of virtual labs, 2007
- SPOT — Power systems in transient and steady-state mode, 2007
- ...

Power System Stability - SPOT

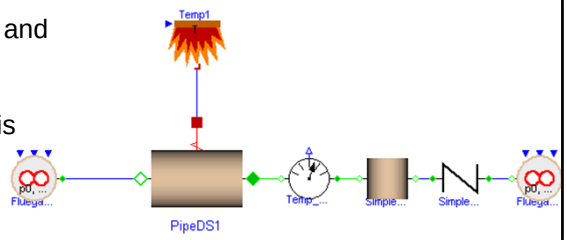
The SPOT package is a Modelica Library for Power Systems Voltage and Transient stability simulations



Thermo-hydraulics Library – ThermoFluid Replaced by the New Fluid/Media Library

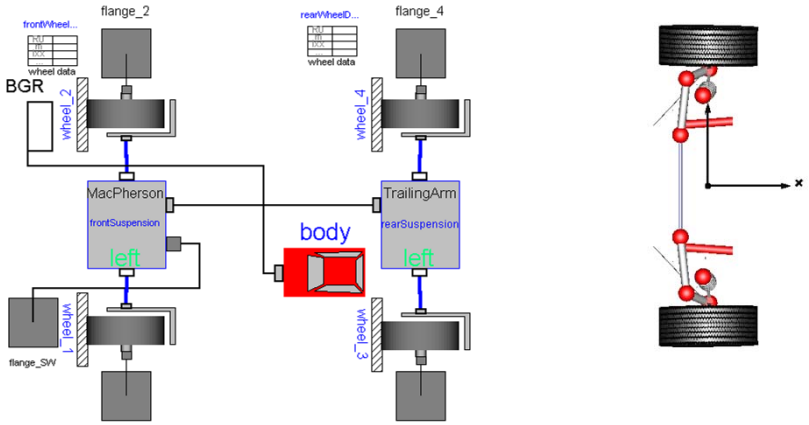
ThermoFluid is a Modelica base library for thermo-hydraulic models

- Includes models that describe the basic physics of flows of fluid and heat, medium property models for water, gases and some refrigerants, and also simple components for system modeling.
- Handles static and dynamic momentum balances
- Robust against backwards and zero flow
- The discretization method is a first-order, finite volume method (staggered grid).



Vehicle Dynamics Library – `VehicleDynamics` There is a Greatly Extended Commercial Version

This library is used to model vehicle chassis

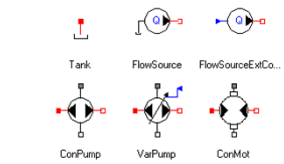


Some Commercial Libraries

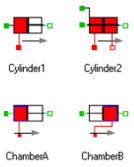
- Powertrain
- SmartElectricDrives
- VehicleDynamics
- AirConditioning
- HyLib
- PneuLib
- CombiPlant
- HydroPlant
- …

Hydraulics Library HyLib

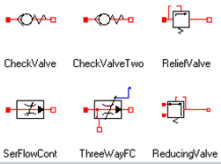
- Licensed Modelica package developed originally by Peter Beater
- More than 90 models for
  - Pumps
  - Motors and cylinders
  - Restrictions and valves
  - Hydraulic lines
  - Lumped volumes and sensors
- Models can be connected in an arbitrary way, e.g. in series or in parallel.
- `HyLibLight` is a free subset of `HyLib`
- More info: www.hylib.com



Pumps



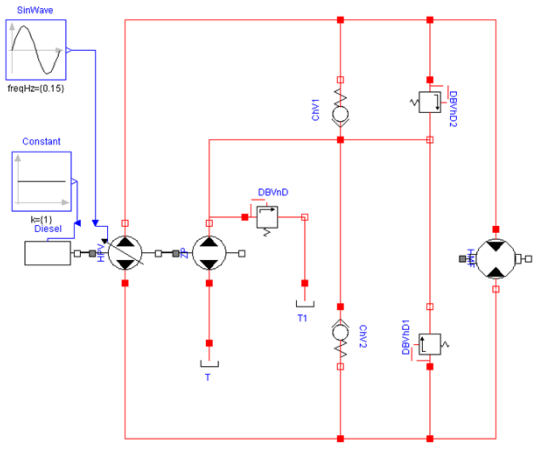
Cylinders



Valves

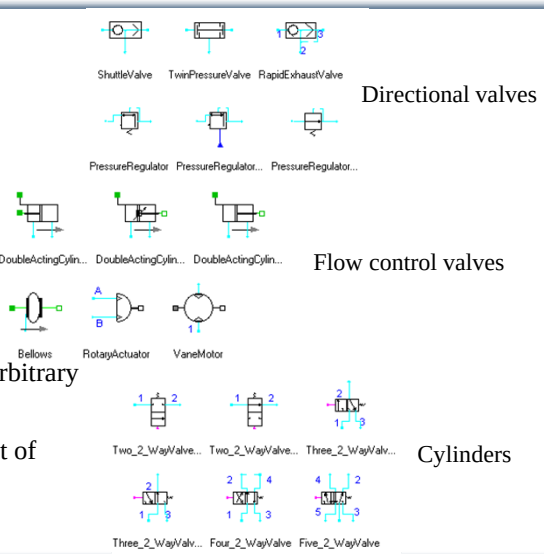
HyLib - Example

Hydraulic drive system with closed circuit



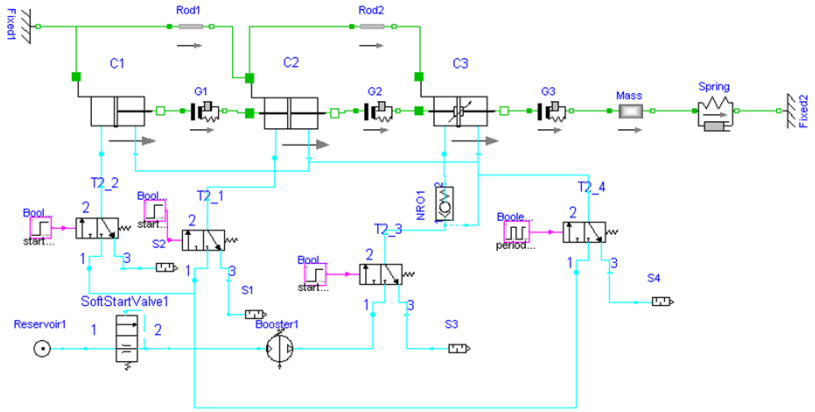
Pneumatics Library PneuLib

- Licensed Modelica package developed by Peter Beater
- More than 80 models for
  - Cylinders
  - Motors
  - Valves and nozzles
  - Lumped volumes
  - Lines and sensors
- Models can be connected in an arbitrary way, e.g. in series or in parallel.
- `PneuLibLight` is a free subset of `HyLib`.
- More info: www.pneulib.com



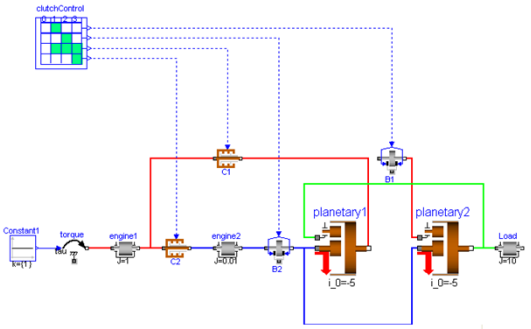
PneuLib - Example

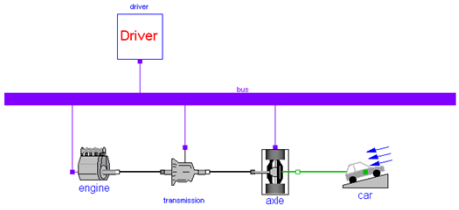
Pneumatic circuit with multi-position cylinder, booster and different valves

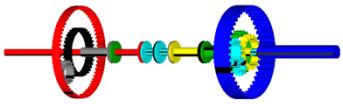


Powertrain Library - Powertrain

- Licensed Modelica package developed by DLR
- Speed and torque dependent friction
- Bus concept
- Control units
- Animation







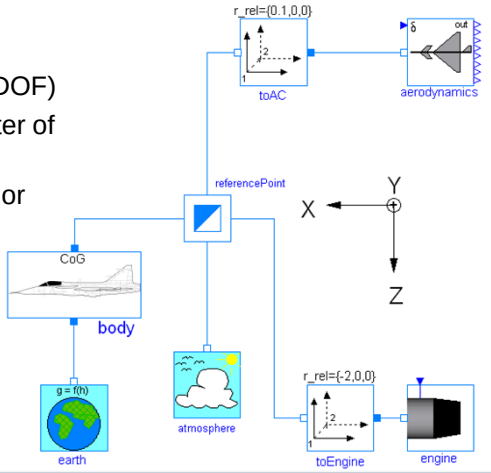
Some Modelica Applications

Example Fighter Aircraft Library

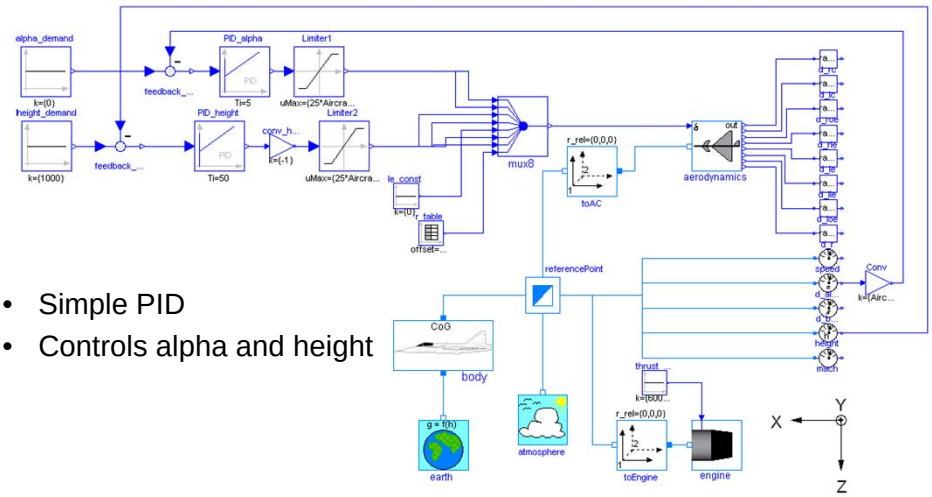
Custom made library, `Aircraft*`, for fighter aircraft applications

- Six degrees of freedom (6 DOF)
- Dynamic calculation of center of gravity (CoG)
- Use of Aerodynamic tables or mechanical rudders

*Property of FOI (The Swedish Defence Institute)



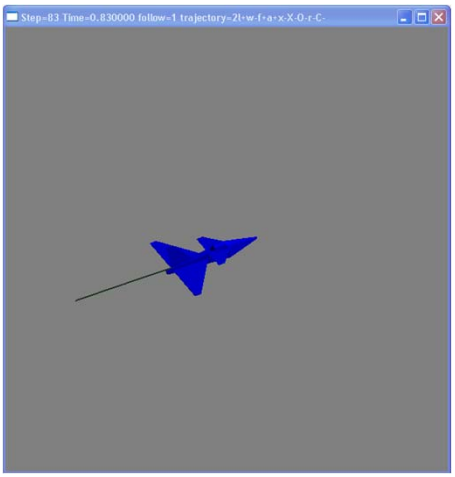
Aircraft with Controller



- Simple PID
- Controls alpha and height

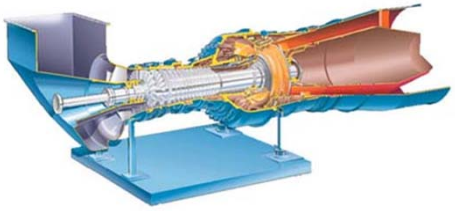
Example Aircraft Animation

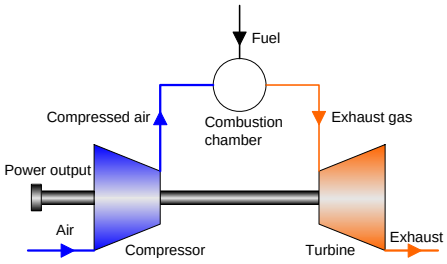
Animation of fighter aircraft with controller



Example Gas Turbine

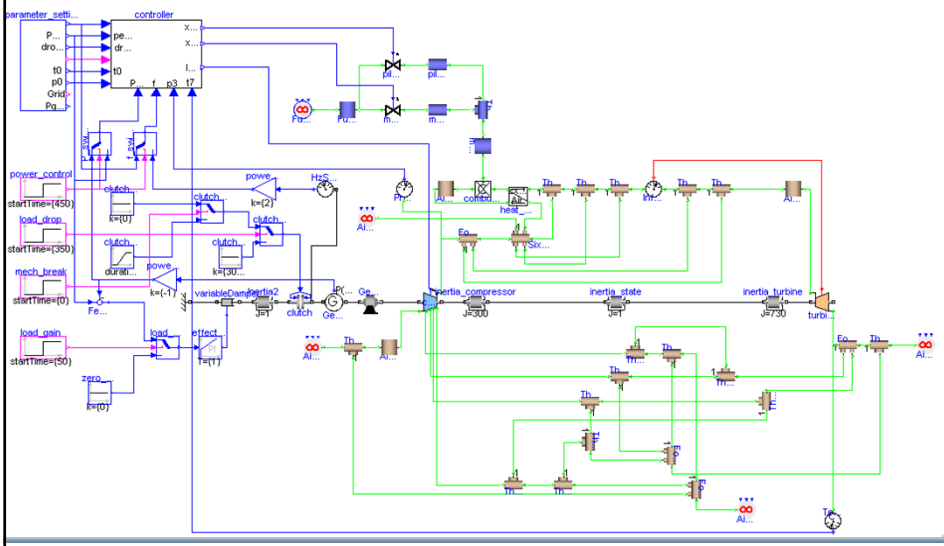
42 MW gas turbine (GTX 100) from Siemens Industrial Turbomachinery AB, Finspång, Sweden





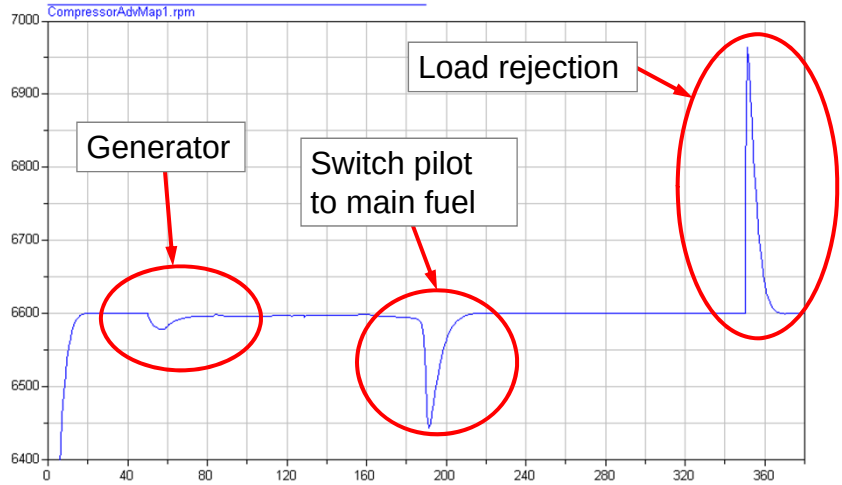
Courtesy Siemens Industrial Turbines AB

Example Gas Turbine

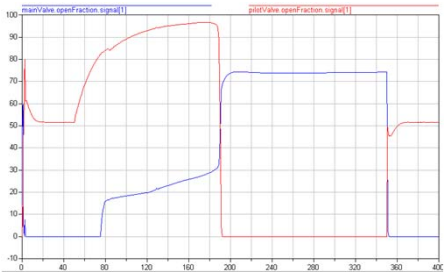


Example Gas Turbine – Load Rejection

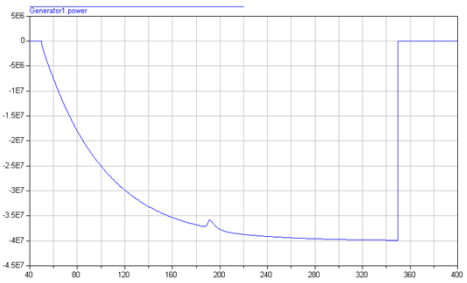
Rotational speed (rpm) of the compressor shaft



Example Gas Turbine – Load Rejection

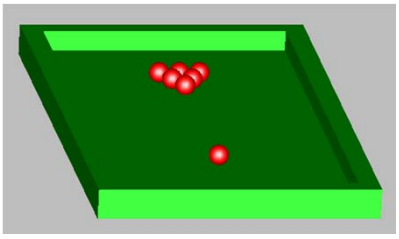


Percentage of fuel valve opening (red = pilot, blue = main)



Generated power to the simulated electrical grid

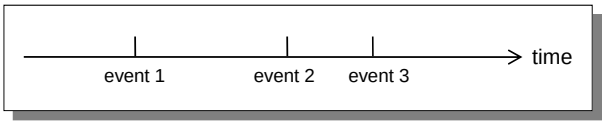
Discrete Events and Hybrid Systems



Picture: Courtesy Hilding Elmqvist

Events

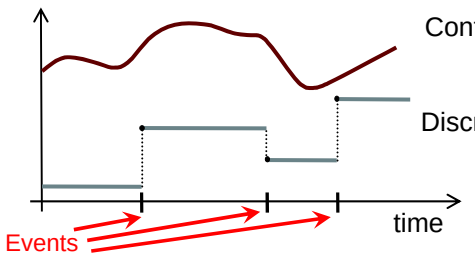
Events are ordered in time and form an event history



- 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.

Hybrid Modeling

Hybrid modeling = continuous-time + discrete-time modeling




```
Real x;
Voltage v;
Current i;
```

```
discrete Real x;
Integer i;
Boolean b;
```

- A *point* in time that is instantaneous, i.e., has zero duration
- An *event condition* so that the event can take place
- A set of *variables* that are associated with the event
- Some *behavior* associated with the event,
  e.g. *conditional equations* that become active or are deactivated at the event

Event creation – if

if-equations, if-statements, and if-expressions

```
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;
```

False if s<0

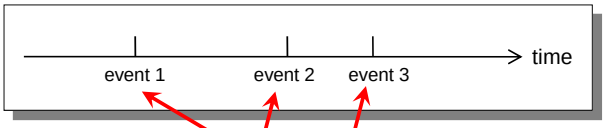
If-equation choosing equation for v

If-expression

Event creation – when

when-equations

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



Equations only active at event times

Time event

```
when time >= 10.0 then
  ...
end when;
```

Only dependent on time, can be scheduled in advance

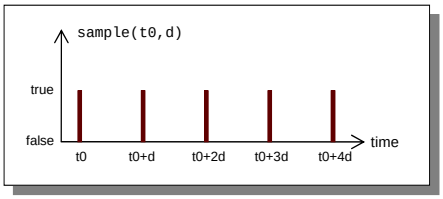
State event

```
when sin(x) > 0.5 then
  ...
end when;
```

Related to a state. Check for zero-crossing

Generating Repeated Events

The call `sample(t0,d)` returns true and triggers events at times `t0+i*d`, where `i=0,1,...`

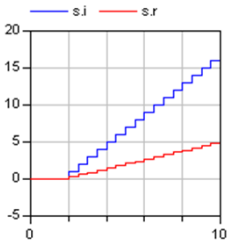


Variables need to be discrete

```
model SamplingClock
  Integer i;
  discrete Real r;
equation
  when sample(2,0.5) then
    i = pre(i)+1;
    r = pre(r)+0.3;
  end when;
end SamplingClock;
```

Creates an event after 2 s, then each 0.5 s

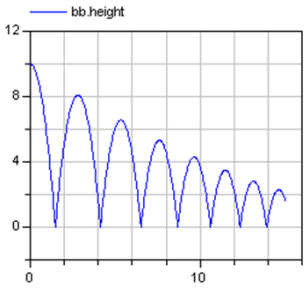
pre(...) takes the previous value before the event.



Reinit – Discontinuous Changes

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=9.81;   //gravitational acc.
  parameter Real c=0.90;   //elasticity constant
  Real height(start=10),velocity(start=0);
equation
  der(height) = velocity;
  der(velocity)=-g;
  when height<0 then
    reinit(velocity, -c*velocity);
  end when;
end BouncingBall;
```

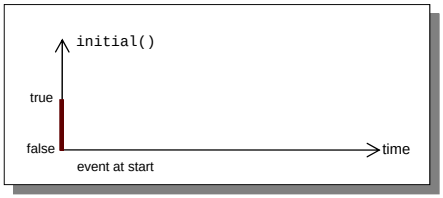


Initial conditions

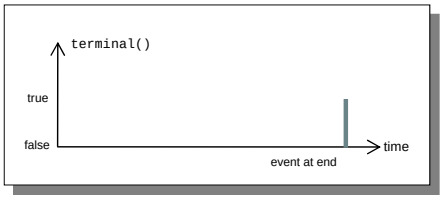
Reinit "assigns" continuous-time variable `velocity` a new value

`initial` and `terminal` events

Initialization actions are triggered by `initial()`

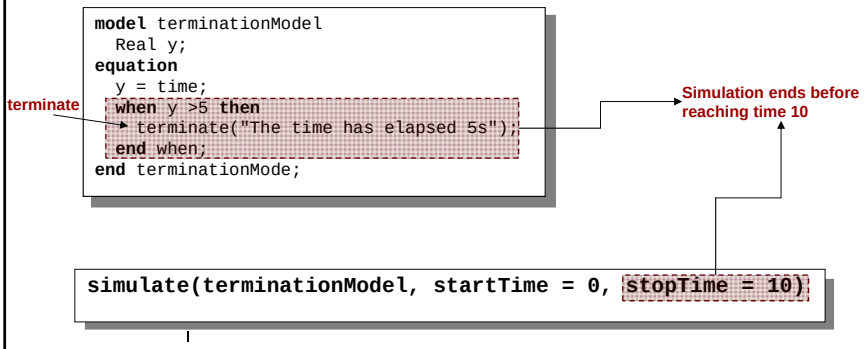


Actions at the end of a simulation are triggered by `terminal()`



Terminating a Simulation

There `terminate()` function is useful when a wanted result is achieved and it is no longer useful to continue the simulation. The example below illustrates the use:



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
    ...
```


Event Priority

Erroneous multiple definitions, single assignment rule violated

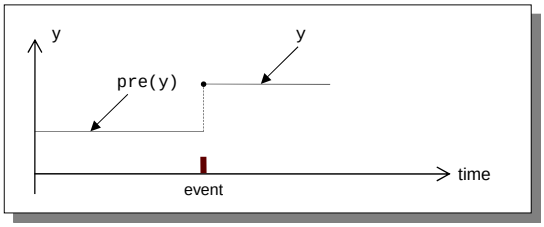
```
model WhenConflictX   // Erroneous model: two equations define x
  discrete Real x;
  equation
  when time>=2 then   // When A: Increase x by 1.5 at  time=2
    x = pre(x)+1.5;
  end when;
  when time>=1 then   // When B: Increase x by 1 at   time=1
    x = pre(x)+1;
  end when;
end WhenConflictX;
```

Using event priority to avoid erroneous multiple definitions

```
model WhenPriorityX
  discrete Real x;
equation
  when time>=2 then       // Higher priority
    x = pre(x)+1.5;
  elsewhen time>=1 then   // Lower priority
    x = pre(x)+1;
  end when;
end WhenPriorityX;
```


Obtaining Predecessor Values of a Variable Using `pre()`

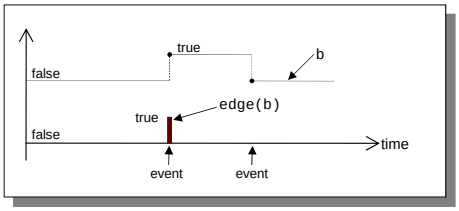
At an event, `pre(y)` gives the previous value of y immediately before the event, except for event iteration of multiple events at the same point in time when the value is from the previous iteration



- 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

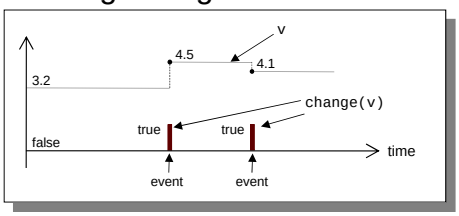
Detecting Changes of Boolean Variables Using `edge()` and `change()`

Detecting changes of boolean variables using `edge()`



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

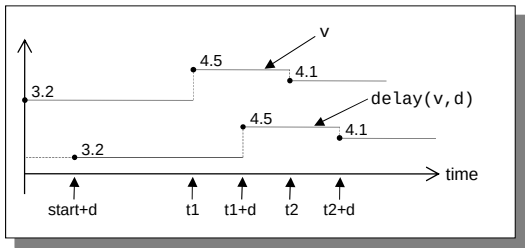
Detecting changes of discrete-time variables using `change()`



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

Creating Time-Delayed Expressions

Creating time-delayed expressions using `delay()`

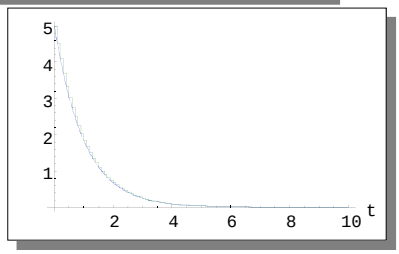


In the expression `delay(v,d)` *v* is delayed by a delay time *d*

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})
```

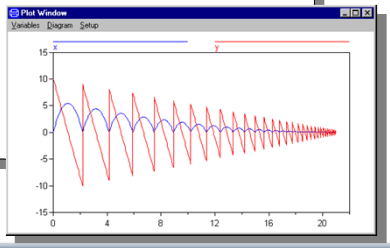


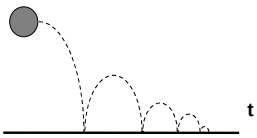
Discontinuous Changes to Variables at Events via When-Equations/Statements

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

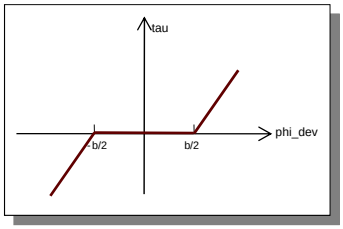
```
model BouncingBall "the bouncing ball model"
  parameter Real g=9.18;   //gravitational acc.
  parameter Real c=0.90;   //elasticity constant
  Real x(start=0),y(start=10);
equation
  der(x) = y;
  der(y)=-g;
  when x<0 then
    reinit(y, -c*y);
  end when;
end BouncingBall;
```



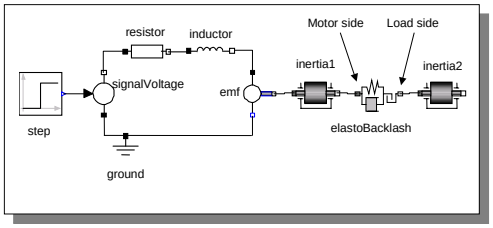


A Mode Switching Model Example

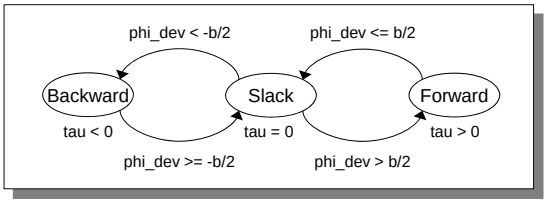
Elastic transmission with slack



DC motor transmission with elastic backlash



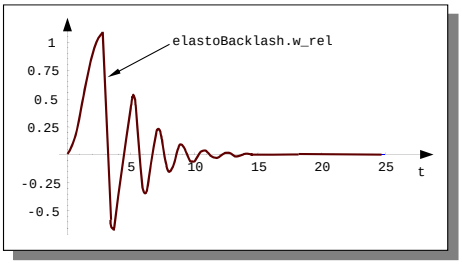
A finite state automaton `SimpleElastoBacklash` model



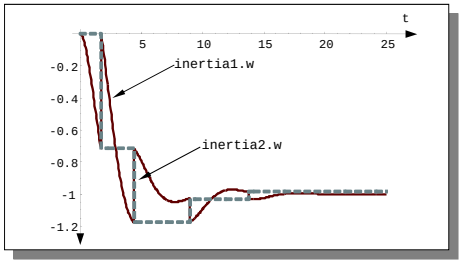
A Mode Switching Model Example cont'

```
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'



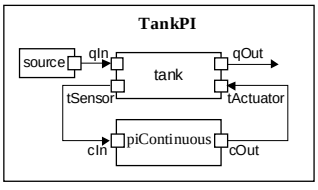
Relative rotational speed between the flanges of the `Elastobacklash` transmission



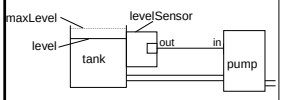
We define a model with less mass in `inertia2(J=1)`, no damping `d=0`, and weaker string constant `c=1e-5`, to show even more dramatic backlash phenomena

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

Water Tank System with PI Controller




```
model TankPI
  LiquidSource           source(flowLevel=0.02);
  Tank                   tank(area=1);
  PIcontinuousController piContinuous(ref=0.25);
equation
  connect(source.qOut, tank.qIn);
  connect(tank.tActuator, piContinuous.cOut);
  connect(tank.tSensor, piContinuous.cIn);
end TankPI;
```




```
model Tank
  ReadSignal tOut;  // Connector, reading tank level
  ActSignal  tInp;  // Connector, actuator controlling input flow
  parameter Real flowVout = 0.01;       // [m3/s]
  parameter Real area = 0.5;            // [m2]
  parameter Real flowGain = 10;         // [m2/s]
  Real h(start=0);                      // tank level [m]
  Real qIn;                  // flow through input valve[m3/s]
  Real qOut;                 // flow through output valve[m3/s]
equation
  der(h)=(qIn-qOut)/area;               // mass balance equation
  qOut=if time>100 then flowVout else 0;
  qIn = flowGain*tInp.act;
  tOut.val = h;
end Tank;
```


Water Tank System with PI Controller – cont'

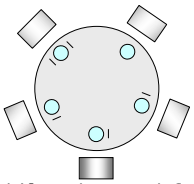
```
partial model BaseController
  parameter Real Ts(unit = "s") = 0.1   "Time period between discrete samples";
  parameter Real K = 2                  "Gain";
  parameter Real T(unit = "s") = 10     "Time constant";
  ReadSignal cIn                        "Input sensor level, connector";
  ActSignal  cOut                       "Control to actuator, connector";
  parameter Real ref                    "Reference level";
  Real error                            "Deviation from reference level";
  Real outCtr                           "Output control signal";
equation
  error = ref - cIn.val;
  cOut.act = outCtr;
end BaseController;
```

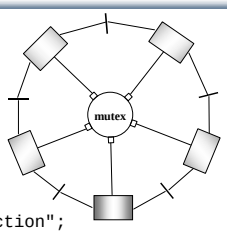
```
model PIDcontinuousController
  extends BaseController(K = 2, T = 10);
  Real  x;
  Real  y;
equation
  der(x) = error/T;
  y      = T*der(error);
  outCtr = K*(error + x + y);
end PIDcontinuousController;
```

```
model PIdiscreteController
  extends BaseController(K = 2, T = 10);
  discrete Real x;
equation
  when sample(0, Ts) then
    x = pre(x) + error * Ts / T;
    outCtr = K * (x+error);
  end when;
end PIdiscreteController;
```

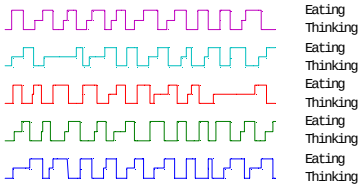

Concurrency and Resource Sharing

Dining Philosophers Example






```
model DiningTable
  parameter Integer n = 5  "Number of philosophers and forks";
  parameter Real sigma = 5 " Standard deviation for the random function";
  // Give each philosopher a different random start seed
  // Comment out the initializer to make them all hungry simultaneously.
  Philosopher phil[n](startSeed=[1:n,1:n,1:n], sigma=fill(sigma,n));
  Mutex       mutex(n=n);
  Fork        fork[n];
equation
  for i in 1:n loop
    connect(phil[i].mutexPort, mutex.port[i]);
    connect(phil[i].right, fork[i].left);
    connect(fork[i].right, phil[mod(i, n) + 1].left);
  end for;
end DiningTable;
```



Synchronous features in Modelica 3.3

- Based on a clock calculus and inference system developed by Colaco *and Pouzet*
- *‼ New language constructs*
  - **Clock Constructors**
  - **Base-clock conversion operators**
  - **Sub-clock conversion operators**
  - **Previous and Interval operators**

Why do we need a new syntax?

- Less limitations : ‼ General equations are allowed in clocked partitions and in particular also in clocked when-clauses
- Issues with the current syntax:
  - **Sampling errors cannot be detected**
  - **Unnecessary initial values have to be defined**
  - **Inverse models not supported in discrete systems:**
  - **Efficiency degradation at event points**

Clocks

- `Clock()`: Returns a clock that is inferred
- `Clock(i,r)`: Returns a variable interval clock where the next interval at the current clock tick is defined by the rational number `i/r`. If `i` is parameteric the clock is periodic.
- `Clock(ri)` : Returns a variable interval clock where the next interval at the current clock tick is defined by the Real number `ri`. If `ri` is parametric, the clock is periodic.
- `Clock(cond, ri0)`: Returns a Boolean clock that ticks whenever the condition `cond` changes from false to true. The optional `ri0` argument is the value returned by operator `interval()` at the first tick of the clock.
- `Clock(c, m)`: Returns clock `c` and associates the solver method `m` to the returned clock .

Sample and Hold

Base-clock conversion operators

- `sample(u,c)`: Returns continuous-time variable u as clocked variable that has the optional argument c as associated clock.
- `hold(u)`: Returns the clocked variable u as piecewise constant continuous-time signal. Before the first tick of the clock of u, the start value of u is returned.

Clocks : an example

```
model MassWithSpringDamper
  parameter Modelica.SIunits.Mass m=1;
  parameter Modelica.SIunits.TranslationalSpringConstant k=1;
  parameter Modelica.SIunits.TranslationalDampingConstant
d=0.1;
  Modelica.SIunits.Position x(start=1,fixed=true) "Position";
  Modelica.SIunits.Velocity v(start=0,fixed=true) "Velocity";
  Modelica.SIunits.Force f "Force";
equation
  der(x) = v
  m*der(v) =
end MassWithS
```

```
model SpeedControl
   extends MassWithSpringDamper;
parameter Real K = 20 "Gain of speed P controller";
parameter Modelica.SIunits.Velocity vref = 100 "Speed ref.";
discrete Real vd;
discrete Real u(start=0);
equation
// speed sensor
vd = sample(v, Clock(0.01)); // P controller for speed
u = K*(vref-vd);
// force actuator
f = hold(u);
end SpeedControl;
```


Biological Models
Population Dynamics
Predator-Prey

Some Well-known Population Dynamics Applications

- Population Dynamics of Single Population
- Predator-Prey Models (e.g. Foxes and Rabbits)

Population Dynamics of Single Population

- P – population size = number of individuals in a population
- $\dot{P}$ – population change rate, change per time unit
- g – growth factor of population (e.g. % births per year)
- d – death factor of population (e.g. % deaths per year)

$$growthrate = g \cdot P$$

$$deathrate = d \cdot P$$

Exponentially increasing population if (g-d)>0

$$\dot{P} = growthrate - deathrate$$

Exponentially decreasing population if (g-d)<0

$$\dot{P} = (g - d) \cdot P$$

Population Dynamics Model

- g – growth rate of population
- d – death rate of population
- P – population size

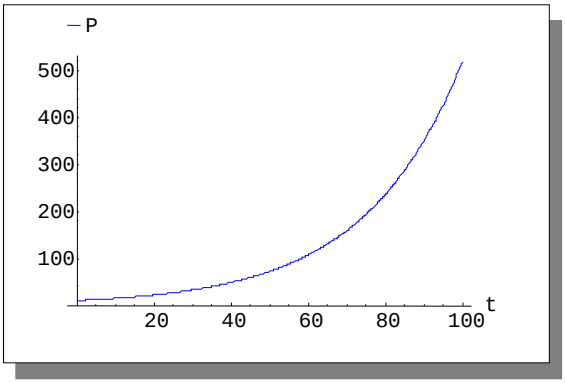
$$\dot{P} = growthrate - deathrate$$

```
class PopulationGrowth
  parameter Real g = 0.04    "Growth factor of population";
  parameter Real d = 0.0005  "Death factor of population";
  Real           P(start=10) "Population size, initially 10";
equation
  der(P) = (g-d)*P;
end PopulationGrowth;
```


Simulation of PopulationGrowth

```
simulate(PopulationGrowth, stopTime=100)
plot(P)
```

Exponentially increasing population if (g-d)>0



Population Growth Exercise!!

- Locate the PopulationGrowth model in DrModelica
- Change the initial population size and growth and death factors to get an exponentially decreasing population

```
simulate(PopulationGrowth, stopTime=100)
plot(P)
```

Exponentially decreasing population if (g-d)<0

```
class PopulationGrowth
  parameter Real g = 0.04    "Growth factor of population";
  parameter Real d = 0.0005  "Death factor of population";
  Real           P(start=10) "Population size, initially 10";
equation
  der(P) = (g-d)*P;
end PopulationGrowth;
```


Population Dynamics with both Predators and Prey Populations

- Predator-Prey models

Predator-Prey (Foxes and Rabbits) Model

- R = rabbits = size of rabbit population
- F = foxes = size of fox population
- $\dot{R}$ = der(rabbits) = change rate of rabbit population
- $\dot{F}$ = der(foxes) = change rate of fox population
- g_r = g_r = growth factor of rabbits
- d_f = d_f = death factor of foxes
- d_{rf} = d_rf = death factor of rabbits due to foxes
- g_{fr} = g_rf = growth factor of foxes due to rabbits and foxes

$$\dot{R} = g_r \cdot R - d_{rf} \cdot F \cdot R \qquad \dot{F} = g_{fr} \cdot d_{rf} \cdot R \cdot F - d_f \cdot F$$

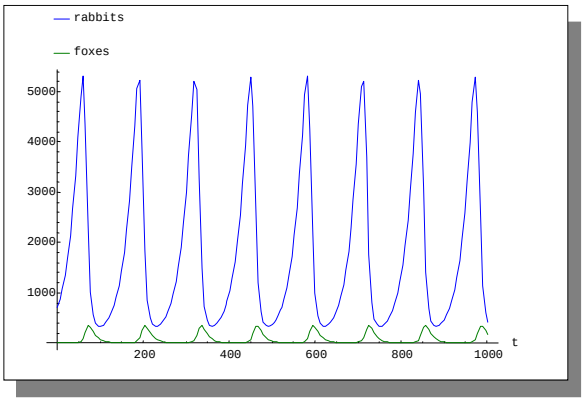
```
der(rabbits) = g_r*rabbits - d_rf*rabbits*foxes;
der(foxes)   = g_fr*d_rf*rabbits*foxes - d_f*foxes;
```


Predator-Prey (Foxes and Rabbits) Model

```
class LotkaVolterra
  parameter Real g_r =0.04    "Natural growth rate for rabbits";
  parameter Real d_rf=0.0005  "Death rate of rabbits due to foxes";
  parameter Real d_f =0.09    "Natural deathrate for foxes";
  parameter Real g_fr=0.1     "Efficency in growing foxes from rabbits";
  Real     rabbits(start=700) "Rabbits,(R) with start population 700";
  Real     foxes(start=10)    "Foxes,(F) with start population 10";
equation
  der(rabbits) = g_r*rabbits - d_rf*rabbits*foxes;
  der(foxes)   = g_fr*d_rf*rabbits*foxes - d_f*foxes;
end LotkaVolterra;
```


Simulation of Predator-Prey (LotkaVolterra)

```
simulate(LotkaVolterra, stopTime=3000)
plot({rabbits, foxes}, xrange={0,1000})
```



Exercise of Predator-Prey

- Locate the LotkaVolterra model in DrModelica
- Change the death and growth rates for foxes and rabbits, simulate, and observe the effects

```
simulate(LotkaVolterra, stopTime=3000)
plot({rabbits, foxes}, xrange={0,1000})
```

```
class LotkaVolterra
  parameter Real g_r =0.04    "Natural growth rate for rabbits";
  parameter Real d_rf=0.0005  "Death rate of rabbits due to foxes";
  parameter Real d_f =0.09    "Natural deathrate for foxes";
  parameter Real g_fr=0.1     "Efficency in growing foxes from rabbits";
  Real     rabbits(start=700) "Rabbits,(R) with start population 700";
  Real     foxes(start=10)    "Foxes,(F) with start population 10";
equation
  der(rabbits) = g_r*rabbits - d_rf*rabbits*foxes;
  der(foxes)   = g_fr*d_rf*rabbits*foxes - d_f*foxes;
end LotkaVolterra;
```


Model Design

Modeling Approaches

- Traditional state space approach
- Traditional signal-style block-oriented approach
- Object-oriented approach based on finished library component models
- Object-oriented flat model approach
- Object-oriented approach with design of library model components

Modeling Approach 1

Traditional state space approach

Traditional State Space Approach

- Basic structuring in terms of subsystems and variables
- Stating equations and formulas
- Converting the model to state space form:

$$\dot{x}(t) = f(x(t), u(t))$$
$$y(t) = g(x(t), u(t))$$

Difficulties in State Space Approach

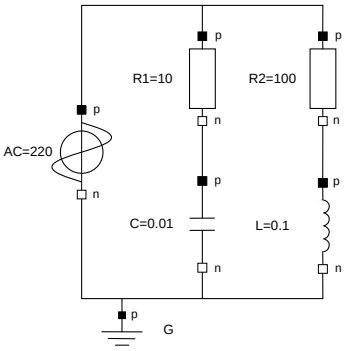
- The system decomposition does not correspond to the "natural" physical system structure
- Breaking down into subsystems is difficult if the connections are not of input/output type.
- Two connected state-space subsystems do not usually give a state-space system automatically.

Modeling Approach 2

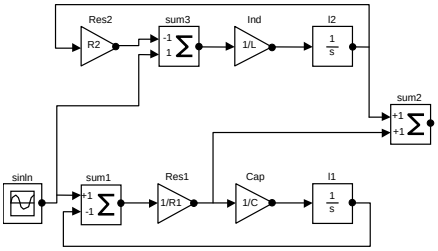
Traditional signal-style block-oriented approach

Physical Modeling Style (e.g Modelica) vs signal flow Block-Oriented Style (e.g. Simulink)

Modelica:
Physical model – easy to understand



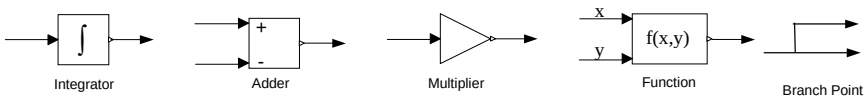
Block-oriented:
Signal-flow model – hard to understand for physical systems



Traditional Block Diagram Modeling

- Special case of model components: the causality of each interface variable has been fixed to either *input* or *output*

Typical Block diagram model components:

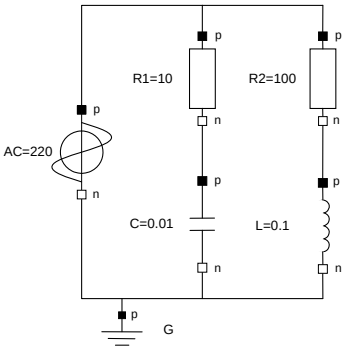


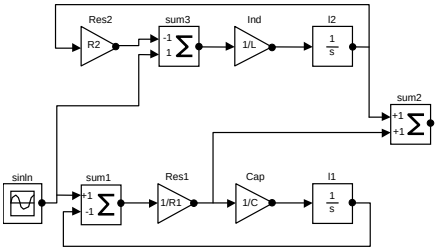
Simulink is a common block diagram tool

Physical Modeling Style (e.g Modelica) vs signal flow Block-Oriented Style (e.g. Simulink)

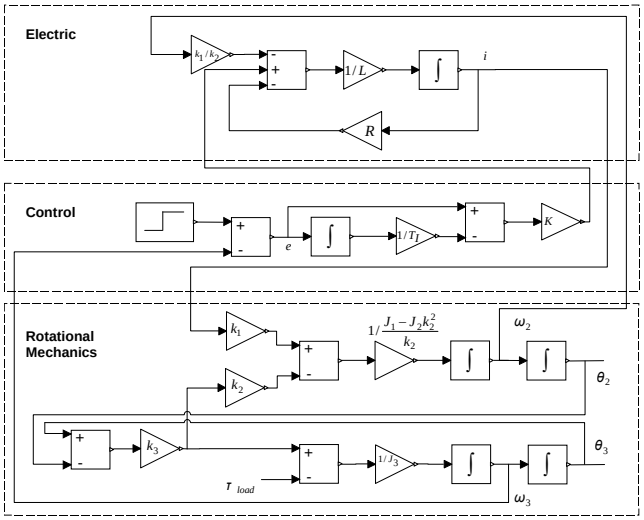
Modelica:
Physical model – easy to understand

Block-oriented:
Signal-flow model – hard to understand for physical systems





Example Block Diagram Models



Properties of Block Diagram Modeling

- - The system decomposition topology does not correspond to the "natural" physical system structure
- - Hard work of manual conversion of equations into signal-flow representation
- - Physical models become hard to understand in signal representation
- - Small model changes (e.g. compute positions from force instead of force from positions) requires redesign of whole model
- + Block diagram modeling works well for control systems since they are signal-oriented rather than "physical"

Object-Oriented Modeling Variants

- Approach 3: Object-oriented approach based on finished library component models
- Approach 4: Object-oriented flat model approach
- Approach 5: Object-oriented approach with design of library model components

Object-Oriented Component-Based Approaches in General

- Define the system briefly
  - What kind of system is it?
  - What does it do?
- Decompose the system into its most important components
  - Define communication, i.e., determine interactions
  - Define interfaces, i.e., determine the external ports/connectors
  - Recursively decompose model components of “high complexity”
- Formulate new model classes when needed
  - Declare new model classes.
  - Declare possible base classes for increased reuse and maintainability

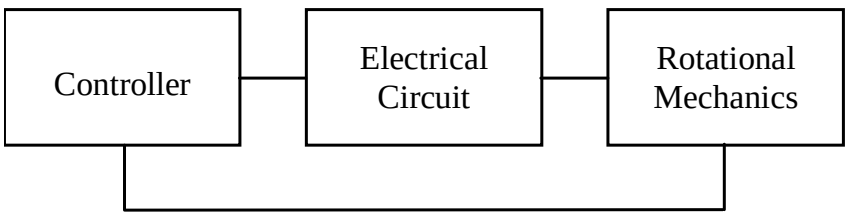
Top-Down versus Bottom-up Modeling

- Top Down: Start designing the overall view. Determine what components are needed.
- Bottom-Up: Start designing the components and try to fit them together later.

Approach 3: Top-Down Object-oriented approach using library model components

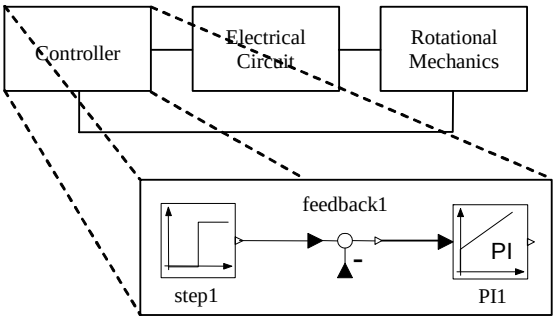
- Decompose into subsystems
- Sketch communication
- Design subsystems models by connecting library component models
- Simulate!

Decompose into Subsystems and Sketch Communication – DC-Motor Servo Example



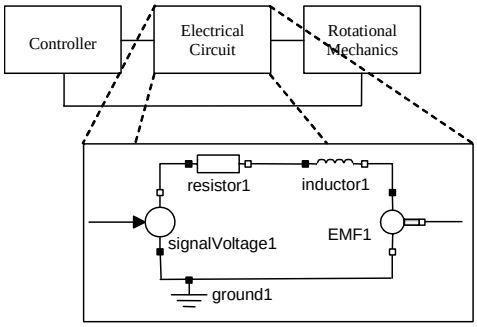
The DC-Motor servo subsystems and their connections

Modeling the Controller Subsystem



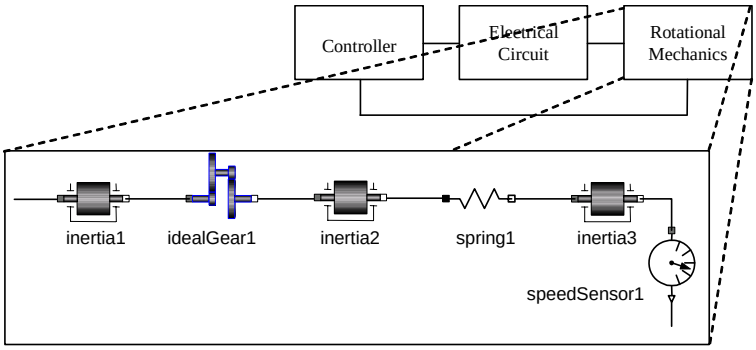
Modeling the controller

Modeling the Electrical Subsystem



Modeling the electric circuit

Modeling the Mechanical Subsystem

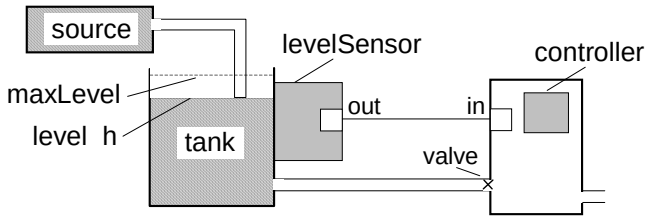


Modeling the mechanical subsystem including the speed sensor.

Object-Oriented Modeling from Scratch

- Approach 4: Object-oriented flat model approach
- Approach 5: Object-oriented approach with design of library model components

Example: OO Modeling of a Tank System



- The system is naturally decomposed into components

Object-Oriented Modeling

Approach 4: Object-oriented flat model design

Tank System Model FlatTank – No Graphical Structure

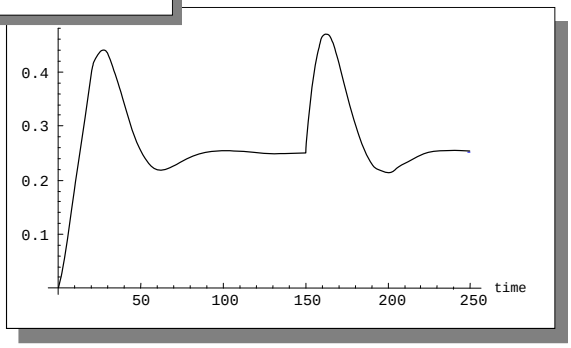
- No component structure
- Just flat set of equations
- Straight-forward but less flexible, no graphical structure

```
model FlatTank
  // Tank related variables and parameters
  parameter Real flowLevel(unit="m3/s")=0.02;
  parameter Real area(unit="m2")       =1;
  parameter Real flowGain(unit="m2/s") =0.05;
  Real           h(start=0,unit="m")   "Tank level";
  Real           qInflow(unit="m3/s")  "Flow through input valve";
  Real           qOutflow(unit="m3/s") "Flow through output valve";
  // Controller related variables and parameters
  parameter Real K=2                   "Gain";
  parameter Real T(unit="s")= 10       "Time constant";
  parameter Real minV=0, maxV=10;    // Limits for flow output
  Real           ref = 0.25  "Reference level for control";
  Real           error       "Deviation from reference level";
  Real           outCtr      "Control signal without limiter";
  Real           x;          "State variable for controller";
equation
  assert(minV>=0,"minV must be greater or equal to zero");//
  der(h) = (qInflow-qOutflow)/area;   // Mass balance equation
  qInflow  = if time>150 then 3*flowLevel else flowLevel;
  qOutflow = LimitValue(minV,maxV,-flowGain*outCtr);
  error  = ref-h;
  der(x) = error/T;
  outCtr = K*(error+x);
end FlatTank;
```


Simulation of FlatTank System

- Flow increase to flowLevel at time 0
- Flow increase to 3*flowLevel at time 150

```
simulate(FlatTank, stopTime=250)
plot(h, stopTime=250)
```

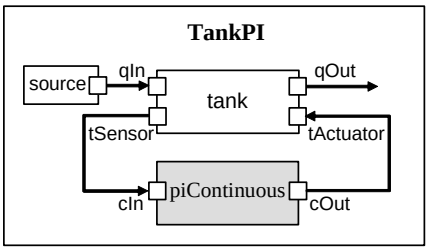


Object-Oriented Modeling

- Approach 5:
  Object-oriented approach with design of library model components

Object Oriented Component-Based Approach Tank System with Three Components

- Liquid source
- Continuous PI controller
- Tank




```
model TankPI
  LiquidSource           source(flowLevel=0.02);
  PIcontinuousController piContinuous(ref=0.25);
  Tank                   tank(area=1);
equation
  connect(source.qOut, tank.qIn);
  connect(tank.tActuator, piContinuous.cOut);
  connect(tank.tSensor, piContinuous.cIn);
end TankPI;
```

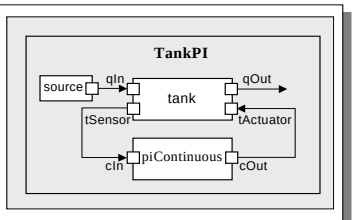

Tank model

- The central equation regulating the behavior of the tank is the mass balance equation (input flow, output flow), assuming constant pressure

```
model Tank
  ReadSignal  tSensor   "Connector, sensor reading tank level (m)";
  ActSignal   tActuator "Connector, actuator controlling input flow";
  LiquidFlow  qIn       "Connector, flow (m3/s) through input valve";
  LiquidFlow  qOut      "Connector, flow (m3/s) through output valve";
  parameter Real area(unit="m2")       = 0.5;
  parameter Real flowGain(unit="m2/s") = 0.05;
  parameter Real minV=0, maxV=10; // Limits for output valve flow
  Real h(start=0.0, unit="m") "Tank level";
equation
  assert(minV>=0,"minV – minimum Valve level must be >= 0 ");//
  der(h)      = (qIn.lflow-qOut.lflow)/area;   // Mass balance
equation
  qOut.lflow  = LimitValue(minV,maxV,-flowGain*tActuator.act);
  tSensor.val = h;
end Tank;
```


Connector Classes and Liquid Source Model for Tank System

```
connector ReadSignal "Reading fluid level"
  Real val(unit="m");
end ReadSignal;

connector ActSignal  "Signal to actuator
 for setting valve position"
  Real act;
end ActSignal;

connector LiquidFlow  "Liquid flow at inlets or outlets"
  Real lflow(unit="m3/s");
end LiquidFlow;
```




```
model LiquidSource
  LiquidFlow qOut;
  parameter  flowLevel = 0.02;
equation
  qOut.lflow = if time>150 then 3*flowLevel else flowLevel;
end LiquidSource;
```

Continuous PI Controller for Tank System

- error = (reference level – actual tank level)
- T is a time constant
- x is controller state variable
- K is a gain factor

$$\frac{dx}{dt} = \frac{error}{T}$$

$$outCtr = K * (error + x)$$

Integrating equations gives Proportional & Integrative (PI)

$$outCtr = K * (error + \int \frac{error}{T} dt)$$

base class for controllers – to be defined

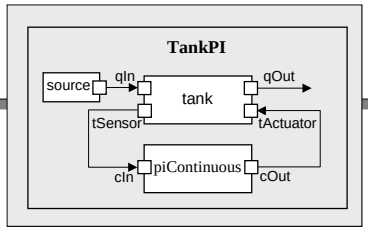
```
model PIcontinuousController
  extends BaseController(K=2,T=10);
  Real  x  "State variable of continuous PI controller";
equation
  der(x) = error/T;
  outCtr = K*(error+x);
end PIcontinuousController;
```

error – to be defined in controller base class

The Base Controller – A Partial Model

```
partial model BaseController
  parameter Real Ts(unit="s")=0.1
    "Ts - Time period between discrete samples – discrete sampled";
  parameter Real K=2              "Gain";
  parameter Real T=10(unit="s") "Time constant - continuous";
  ReadSignal     cIn              "Input sensor level,  connector";
  ActSignal      cOut             "Control to actuator, connector";
  parameter Real ref              "Reference level";
  Real           error            "Deviation from reference level";
  Real           outCtr           "Output control signal";
equation
  error    = ref-cIn.val;
  cOut.act = outCtr;
end BaseController;
```

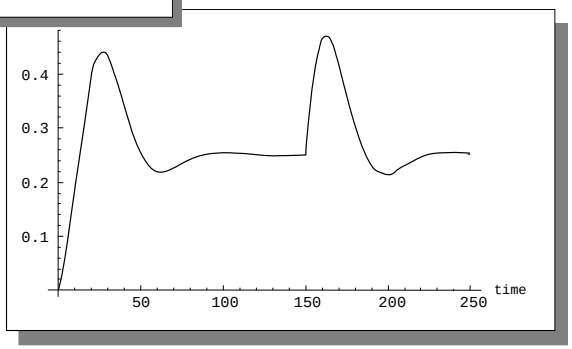
error = difference betwen reference level and actual tank level from cIn connector



Simulate Component-Based Tank System

- As expected (same equations), TankPI gives the same result as the flat model FlatTank

```
simulate(TankPI, stopTime=250)
plot(h, stopTime=250)
```

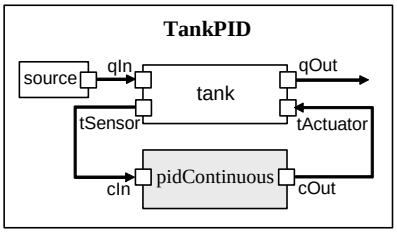


Flexibility of Component-Based Models

- Exchange of components possible in a component-based model
- Example:
  Exchange the PI controller component for a PID controller component

Tank System with Continuous PID Controller Instead of Continuous PI Controller

- Liquid source
- Continuous PID controller
- Tank




```
model TankPID
  LiquidSource            source(flowLevel=0.02);
  PIDcontinuousController pidContinuous(ref=0.25);
  Tank                    tank(area=1);
equation
  connect(source.qOut, tank.qIn);
  connect(tank.tActuator, pidContinuous.cOut);
  connect(tank.tSensor, pidContinuous.cIn);
end TankPID;
```


Continuous PID Controller

- error = (reference level – actual tank level)
- T is a time constant
- x, y are controller state variables
- K is a gain factor

$$\frac{dx}{dt} = \frac{error}{T}$$

$$y = T\frac{d\,error}{dt}$$

$$outCtr = K*(error + x + y)$$

Integrating equations gives Proportional & Integrative & Derivative(PID)

$$outCtr = K*(error + \int \frac{error}{T}dt + T\frac{d\,error}{dt})$$

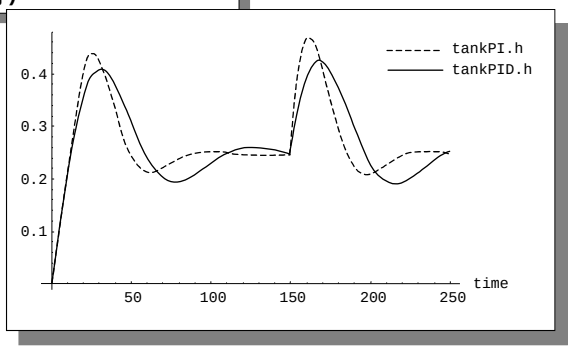
base class for controllers – to be defined

```
model PIDcontinuousController
  extends BaseController(K=2,T=10);
  Real  x; // State variable of continuous PID controller
  Real  y; // State variable of continuous PID controller
equation
  der(x) = error/T;
  y      = T*der(error);
  outCtr = K*(error + x + y);
end PIDcontinuousController;
```


Simulate TankPID and TankPI Systems

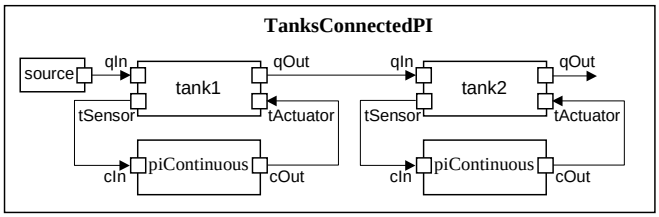
- TankPID with the PID controller gives a slightly different result compared to the TankPI model with the PI controller

```
simulate(compareControllers, stopTime=250)
plot({tankPI.h,tankPID.h})
```



Two Tanks Connected Together

- Flexibility of component-based models allows connecting models together

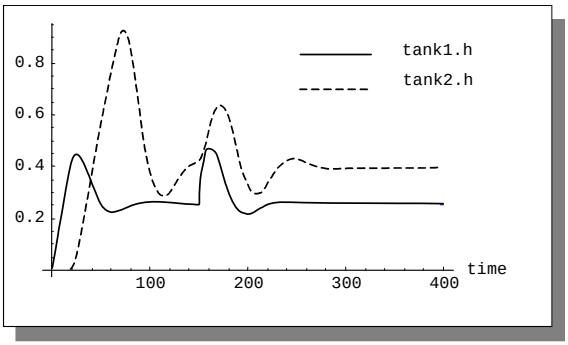



```
model TanksConnectedPI
  LiquidSource  source(flowLevel=0.02);
  Tank          tank1(area=1), tank2(area=1.3);;
  PIcontinuousController piContinuous1(ref=0.25), piContinuous2(ref=0.4);
equation
  connect(source.qOut,tank1.qIn);
  connect(tank1.tActuator,piContinuous1.cOut);
  connect(tank1.tSensor,piContinuous1.cIn);
  connect(tank1.qOut,tank2.qIn);
  connect(tank2.tActuator,piContinuous2.cOut);
  connect(tank2.tSensor,piContinuous2.cIn);
end TanksConnectedPI;
```


Simulating Two Connected Tank Systems

- Fluid level in tank2 increases after tank1 as it should
- Note: tank1 has reference level 0.25, and tank2 ref level 0.4

```
simulate(TanksConnectedPI, stopTime=400)
plot({tank1.h,tank2.h})
```



Exchange: Either PI Continous or PI Discrete Controller

```
partial model BaseController
  parameter Real Ts(unit = "s") = 0.1   "Time period between discrete samples";
  parameter Real K = 2                  "Gain";
  parameter Real T(unit = "s") = 10     "Time constant";
  ReadSignal cIn                        "Input sensor level, connector";
  ActSignal  cOut                       "Control to actuator, connector";
  parameter Real ref                    "Reference level";
  Real error                            "Deviation from reference level";
  Real outCtr                           "Output control signal";
equation
  error = ref - cIn.val;
  cOut.act = outCtr;
end BaseController;
```

```
model PIDcontinuousController
  extends BaseController(K = 2, T = 10);
  Real  x;
  Real  y;
equation
  der(x) = error/T;
  y      = T*der(error);
  outCtr = K*(error + x + y);
end PIDcontinuousController;
```

```
model PIdiscreteController
  extends BaseController(K = 2, T = 10);
  discrete Real x;
equation
  when sample(0, Ts) then
    x = pre(x) + error * Ts / T;
    outCtr = K * (x+error);
  end when;
end PIdiscreteController;
```


Exercises

- Replace the PIcontinuous controller by the PIdiscrete controller and simulate. (see also the book, page 461)
- Create a tank system of 3 connected tanks and simulate.

Principles for Designing Interfaces – i.e., Connector Classes

- Should be *easy* and *natural* to connect components
  - For interfaces to models of physical components it must be physically possible to connect those components
- Component interfaces to facilitate *reuse* of existing model components in class libraries
- Identify kind of interaction
  - If there is interaction between two *physical* components involving energy flow, a combination of one potential and one flow variable in the appropriate domain should be used for the connector class
  - If information or *signals* are exchanged between components, input/output signal variables should be used in the connector class
- Use composite connector classes if several variables are needed

Simplification of Models

- When need to simplify models?
  - When parts of the model are too complex
  - Too time-consuming simulations
  - Numerical instabilities
  - Difficulties in interpreting results due to too many low-level model details
- Simplification approaches
  - Neglect small effects that are not important for the phenomena to be modeled
  - Aggregate state variables into fewer variables
  - Approximate subsystems with very slow dynamics with constants
  - Approximate subsystems with very fast dynamics with static relationships, i.e. not involving time derivatives of those rapidly changing state variables

Library Design
Influenza Model

Spreading of Influenza Epidemic

- Four state variables.
- Initial population of 10 000 non-infected individuals.
- A stem of influenza is introduced into the system.
- Infections occur spontaneously
- The virus spreads in infected individuals and after some time, the individuals become sick.

Spreading of Influenza Epidemic

- Contagious people = sick + infected, spread the disease further
- The sick people eventually get cured and become immune
- The immune period is temporary due to the mutation of the virus.
- Immune people become non-infected people who are again susceptible to infection.

Influenza Model Parameters

- Time to breakdown, 4 weeks
- Actual sickness period, 2 weeks
- Immune period, 26 weeks

$$Incubation = floor\left(\frac{Infected\ population}{Time\ to\ break\ down}\right)$$

$$Activation = floor\left(\frac{Immune\ population}{Immune\ period}\right)$$

$$Cure\_Rate = floor\left(\frac{Sick\ population}{Sickness\ duration}\right)$$

Influenza Model Parameters

- Average weekly contacts of a person with others, C_{Wk}=15
- Contraction rate per contact, $Rate_C$=0.25

*Infection_rate = min(floor(Non_infected_population**
*C_{Wk}*Perc_infected*$Rate_C$+*
Initial), Non_infected_population)

Governing Equations

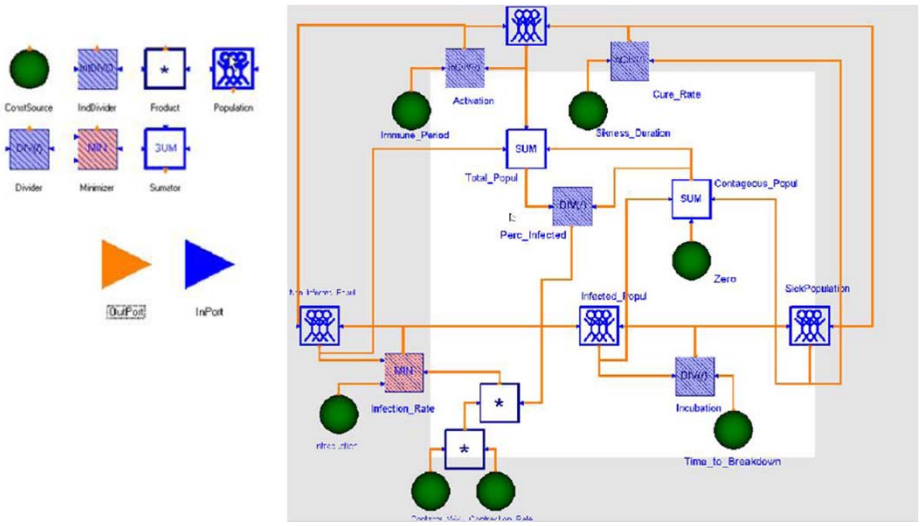
$$\frac{d(Non\_Infected\_population)}{dt} = Activation - Infection\_Rate$$

$$\frac{d(Infected\_population)}{dt} = Infection\_Rate - Incubation$$

$$\frac{d(Immune\_population)}{dt} = Cure\_Rate - Activation$$

$$\frac{d(Sick\_population)}{dt} = Incubation - Cure\_Rate$$

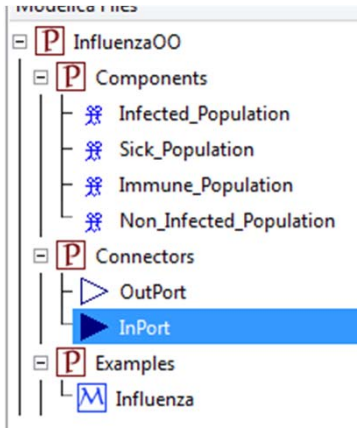
Block Oriented Approach



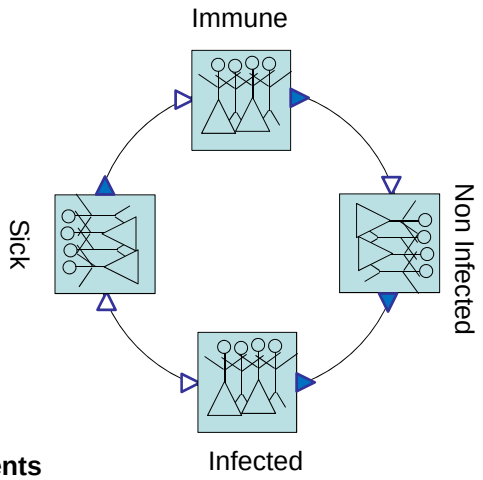
Block Oriented Approach

- Influenza
  - Components
    - IndDivider
    - ConstantSource
    - Product
    - Population
    - Divider
    - Minimizer
    - Sum
  - Connectors
    - OutPort
    - InPort
  - Examples
    - Influenza

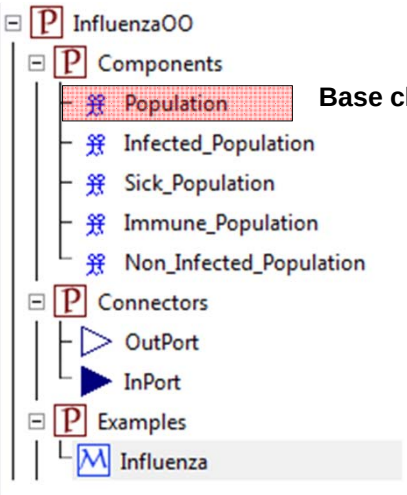
Object Oriented Approach #1



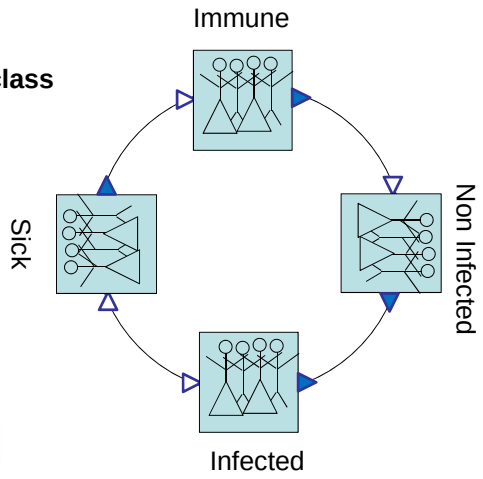
Less Components



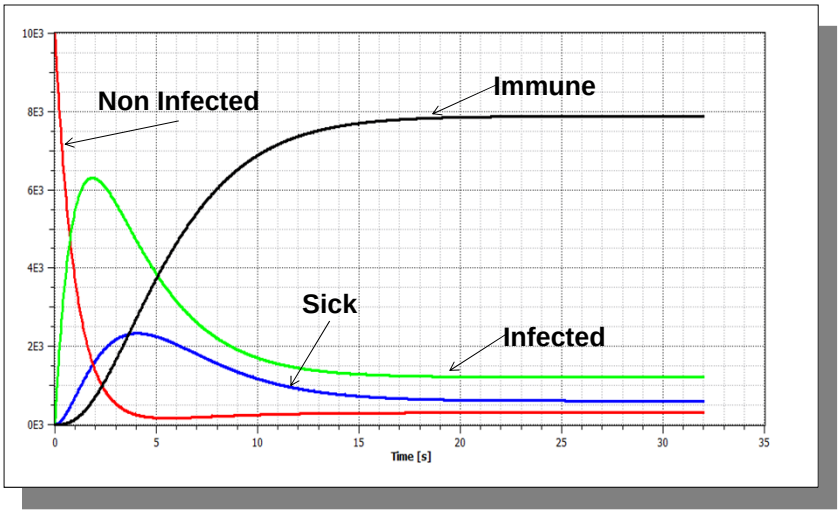
Object Oriented Approach #2



Base class



Simulation



Coclusions

- The influenza epidemic spreads rapidly
- Within 4 weeks, the percentage of sick people reaches its maximum of roughly 25%
- A steady state is reached about 20 weeks
- The disease does not die out naturally.
- A certain percentage loses immunity sufficiently fast to get infected before the virus stem has disappeared.

Goal

Design the influenza library by implement the different design patterns discussed :

- Block Oriented
- Object Oriented
- Object Oriented #2

Short Version of Exercises Using OpenModelica

Version 2012-02-07

Peter Fritzson

PELAB – Programming Environment Laboratory
SE-581 83 Linköping, Sweden

The complete exercises can be found under the Modelica course in the OpenModelica.org home page.

1 Short Introduction to the Graphical Modeling Editor

Install OpenModelica. Start the OpenModelica Connection Editor OMEdit. Do the RLCircuit graphical exercise. See instructions in slides.

2 Simple Textual Modelica Modeling Exercises

Start OMNotebook: Start->Programs->OpenModelica->OMNotebook.
Look in the directory to where you copied the course material.
Open in OMNotebook the file: **Exercises-ModelicaTutorial.onb**

There will be additional instructions in the notebook.

2.1 HelloWorld

Simulate and plot the following example with one differential equation and one initial condition. Do a slight change in the model, re-simulate and re-plot.

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

2.2 Try DrModelica with VanDerPol

Locate the VanDerPol model in DrModelica (link from Section 2.1), run it, change it slightly, and re-run it.

2.3 DAEExample

Locate the DAEExample in DrModelica (Section 2.1: Differential Algebraic Equation System). Simulate and plot.

2.4 A Simple Equation System

Develop a Modelica model that solves the following equation system with initial conditions:

$$\dot{x} = 2 * x * y - 3 * x$$
$$\dot{y} = 5 * y - 7 * x * y$$
$$x(0) = 2$$
$$y(0) = 3$$

2.5 Functions and Algorithm Sections (if you have time)

a) Write a function, `sum`, which calculates the sum of Real numbers, for a vector of arbitrary size.

b) Write a function, `average`, which calculates the average of Real numbers, in a vector of arbitrary size. The function `average` should make use of a function call to `sum`.

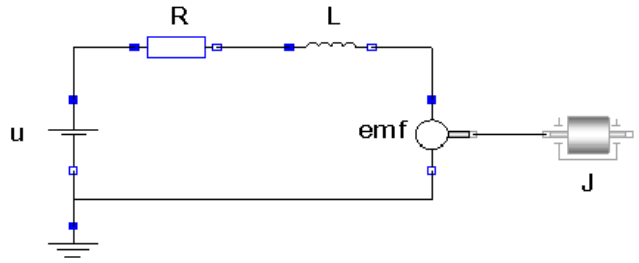
2.6 Hybrid Models - BouncingBall

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

3 Graphical Design using the Graphical Connection Editor

3.1 Simple DC-Motor

Make a simple DC-motor using the Modelica standard library that has the following structure:



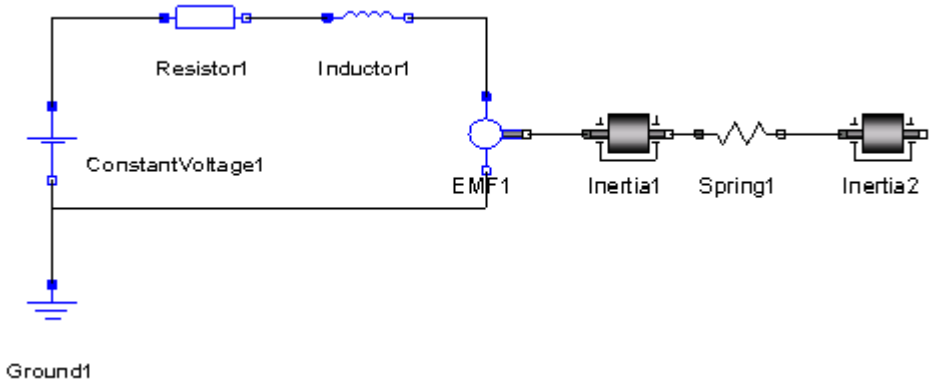
You can simulate and plot the model directly from the graphical editor. Simulate for 15s and plot the variables for the outgoing rotational speed on the inertia axis and the voltage on the voltage source (denoted u in the figure) in the same plot.

Option: You can also save the model, load it and simulate it using OMShell or OMNotebook. You can also go to the graphical editor text view and copy/paste the model into a cell in OMNotebook.

Hint: if you use the plot command in OMNotebook and you have difficulty finding the names of the variables to plot, you can flatten the model by calling instantiateModel, which exposes all variable names.

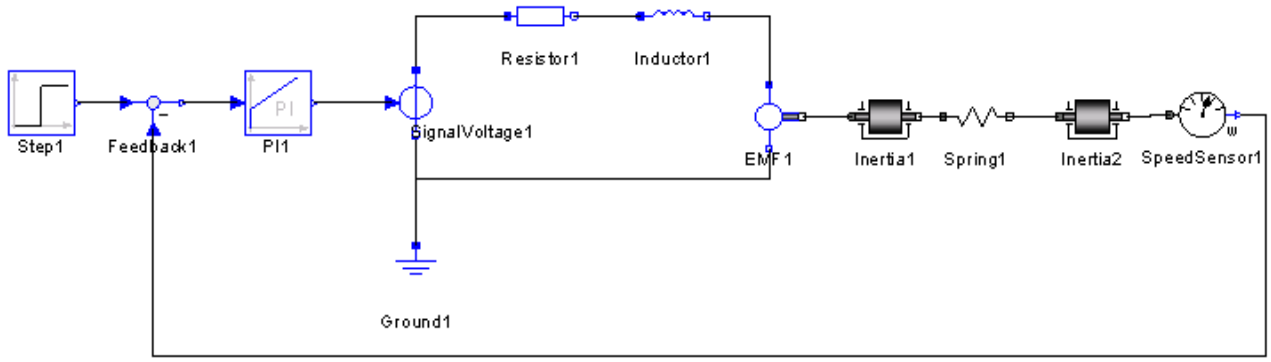
3.2 DC-Motor with Spring and Inertia

Add a torsional spring to the outgoing shaft and another inertia element. Simulate again and see the results. Adjust some parameters (right-click on corresponding model component icon) to make a rather stiff spring.



3.3 DC-Motor with Controller (Extra)

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. Right-click on the PI Controller Icon to change parameters.



3.4 Exercises in Control Theory and Modelica with DrControl (Extra)

Look in the handout slides for further instructions.