

# Introduction to the Dynamic Modelling of Thermo-Fluid Systems using Modelica

---

Francesco Casella

Dipartimento di Elettronica e Informazione  
Politecnico di Milano



# Course Programme

---

- Introduction to Equation-Based, Object-Oriented modelling
- Introduction to the Modelica language
- Properties of Differential-Algebraic Equations systems (DAE)
- Symbolic manipulation of DAEs obtained from Modelica models
- The Modelica.Media and ExternalMedia libraries for fluid property computations
- The Modelica.Fluid library for thermo-fluid system modelling
- The ThermoPower library for power plant modelling

---

# Basic concepts in Object-Oriented Modelling

# Different Approaches to Modular Modelling

It is convenient to build mathematical models of complex systems by *aggregation* of the models of their constituent parts (*modular* modelling). From this point of view, two complementary approaches can be followed:

## *Procedural or Causal Approach*

- The model is described in a form which is close to the solution **algorithm**
- The interaction between the models is formalized in terms of **input** and **output variables**
- *Pro*: It is rather straightforward to simulate elementary and aggregate models
- *Con*: The code might be difficult to read a posteriori
- *Con*: The model of a given system can only be re-used in the same context (i.e. with the same prescribed variables at the system boundary):  
low re-usability of basic models


## *Declarative or A-Causal Approach*

- The model is described by **equations** in a context-independent form, without caring about the actual solution algorithm
- The interaction between the models is formalized in terms of **connection equations** without any specification on causality
- *Pro*: High re-usability of basic models: (arbitrary connection between models)
- *Pro*: High readability of models
- *Con*: It is more difficult to go from the mathematical model to the numerical simulation algorithm (the model is not *oriented* to the solution algorithm)

# Causal / Procedural Modelling - I

## Causal models $\Leftrightarrow$ Procedural models

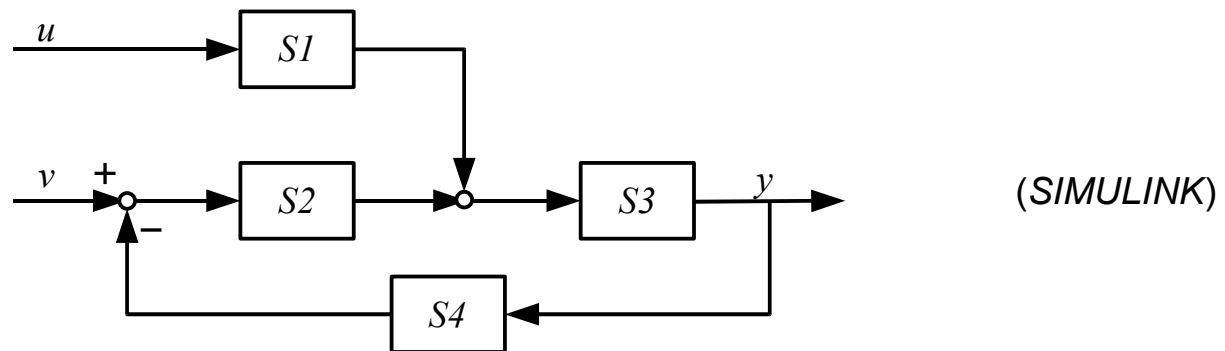
- The model of each system is given by identifying inputs and outputs


$$\dot{x} = \frac{I}{C} \quad (\text{State-Space ODE})$$
$$V = x + RI$$

- It is quite straightforward to solve the equations of a single model

$$x_{k+1} := x_k + \Delta t \cdot f(x_k, u_k) \quad (\text{Forward Euler's Algorithm})$$
$$y_k := g(x_k, u_k)$$

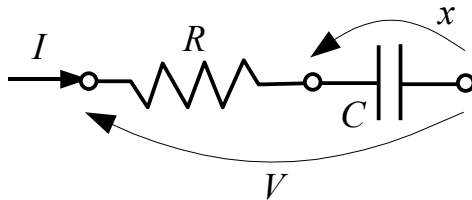
- It is quite straightforward to simulate aggregate systems (block diagrams), in particular if the dynamic systems are strictly proper ( $y = g(x) \Rightarrow$  no algebraic loops)



# Causal / Procedural Modelling - II

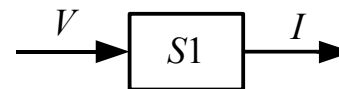
## Causal models $\Leftrightarrow$ Procedural models

- **But:** each sub-system model depends on the **selection of input and output** variables at the system **boundary** (not re-usable).
- Example: RC network



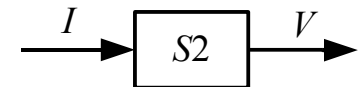
Prescribed Voltage

$$\dot{x} = \frac{V - x}{RC}$$
$$I = \frac{V - x}{R}$$



Prescribed Current

$$\dot{x} = \frac{I}{C}$$
$$V = x + RI$$



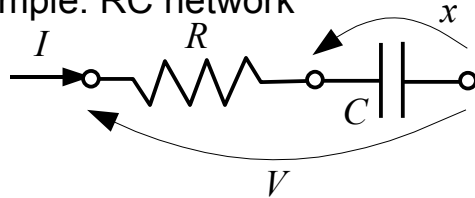
- The same physical model (same equations) requires different models depending on its connection to the outside world

# A-Causal / Declarative Modelling

## A-Causal models $\Leftrightarrow$ Declarative models

- The model of each system is given by the constituent equations
- The model formulation is independent of the actual boundary conditions, therefore it is re-usable in different contexts  $\Rightarrow$  truly *modular* approach
- The physical connection between components is represented by connection equations
- The simulation of an aggregate system is a difficult task, because in general it could require some form of symbolic manipulation of the system of equations, prior to the use of some numerical integration algorithm.

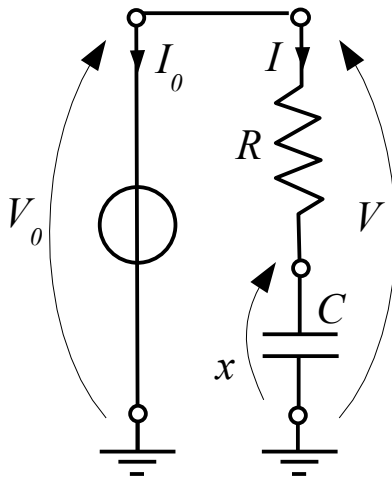
- Example: RC network



$$\begin{aligned} x + RI &= V \\ C \dot{x} &= I \end{aligned}$$

(DAE – declarative model)

- Causality is only determined at the *aggregate system* level.



$$\begin{aligned} x + RI &= V \\ C \dot{x} &= I \end{aligned} \quad \text{(RC network)}$$

$$V_0 = u \quad \text{(voltage generator)}$$

$$V_0 = V \quad \text{(Kirchoff's law - mesh)}$$

$$I_0 + I = 0 \quad \text{(Kirchoff's law - node)}$$



Symbolic  
manipulation  
(simulation tool)

$$\dot{x} = \frac{V_0 - x}{RC}$$

$$V = V_0$$

$$I = \frac{V_0 - x}{R}$$

$$I_0 = -\frac{V_0 - x}{R}$$

# Causal vs. A-Causal Modelling

---

## CONCLUSIONS

An object-oriented approach to physical system modelling requires the adoption of *declarative models*.

In this way, *modularity* is guaranteed:

- the model of an aggregate of sub-systems can always be obtained by composition elementary models
- independently of their actual connections
- provided that *standard interfaces* are defined for the elementary models

Life gets simpler for the *model developer*!

On the other hand, life gets tougher for the *designer* of the *simulation tool*, which must provide:

- sophisticated capabilities of symbolic analysis for large DAE systems (up to many thousands of equations), to analyse and simplify the structure of the problem
- numerical and symbolic algorithms to reduce the DAE system to an ODE system and then integrate it

During this short course, the basic mathematics and algorithms to carry out this task will be discussed.



# Connectors / Ports

In an Object-Oriented context, models are connected together by means of *Connectors* (or *Ports*)

The Port concepts originates from the modelling of physical systems exchanging power (or energy): the connection between models always involves two coupled variables:

- Electrical systems:  $W = V * I$
- Mechanical systems:  $W = s * F$       Work/Energy = Effort variable \* Flow variable
- Hydraulic systems:  $W = P * q$

When  $N$  ports are connected together, the corresponding connection equations are:

- Effort variables  $e_1 = e_2 = \dots = e_N$  (Same voltage / displacement / pressure)
- Flow variables  $\sum f_j = 0$  (Currents / Forces / Flow rates sum to zero)

Connectors can be generalised to cases with more than two variables on each port (abandoning the idea that the product of the two must necessarily be a power)

Connectors are inherently *a-causal*: no variable is declared a-priori as *input* or *output*.

When needed, it is of course possible to declare causal connectors, to describe the connection of input and output signals, such as in control systems.

# Connectors - Example

Mechanical translational systems (1 d.o.f.)

- Standard connector: force (flow) and displacement (effort)

Mechanical rotational systems (1 d.o.f.)

- Standard connector: torque (flow) and angle (effort)

Electrical systems

- Standard connector: current (flow) and voltage (effort)

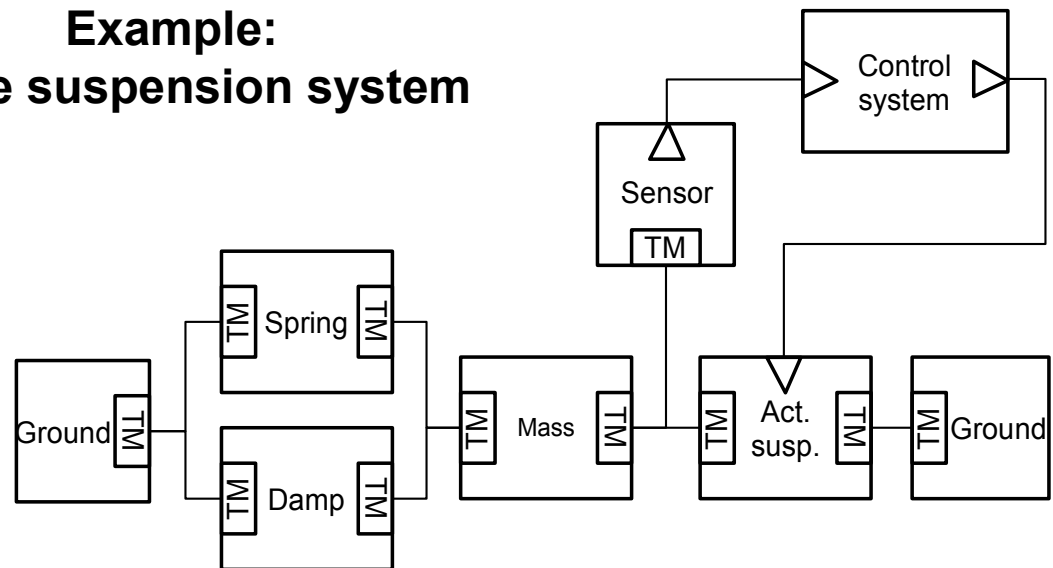
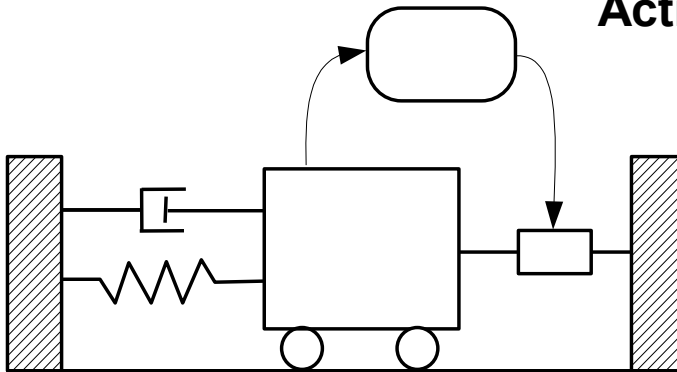
Thermal systems

- Standard connector: heat flow (flow) and temperature (effort)

Hydraulic systems (incompressible fluid)

- Standard connector: volume flow rate (flow) and pressure (effort)

## Example: Active suspension system



# O-O Principles I: Standard Interfaces (Encapsulation)

- To ensure the maximum re-usability of developed models, it is mandatory to define Standard Connectors for the different domains (mechanical, electrical, hydraulic, thermo-hydraulic, etc.), to be used consistently
- All the models using the same standard connectors are interchangeable
- It is advisable that the interface
  - has a clearly defined physical meaning
  - is as independent as possible from the implementation of the models (modelling assumptions, degree of detail, etc.)



The user is shielded from the internal implementation details

- Example: Resistor
  - Basic model
  - Model with thermal effects on resistance
  - Model with stray capacitance
  - Model with high-frequency electromagnetic radiation
  - ...



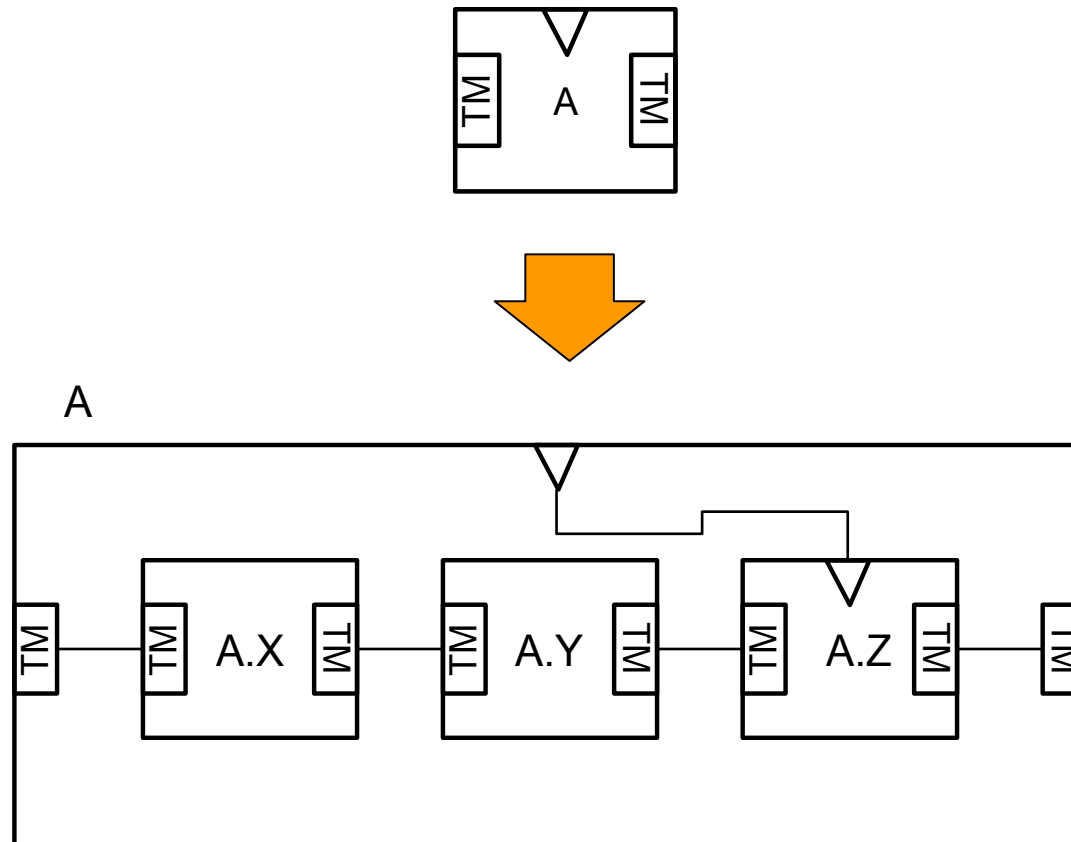
All models share a common (current, voltage) type of connector

- The choice is not trivial in some cases, e.g. 3D multibody mechanical systems, thermo-hydraulic systems

# O-O Principles II: Hierarchical Modularity (Aggregation)

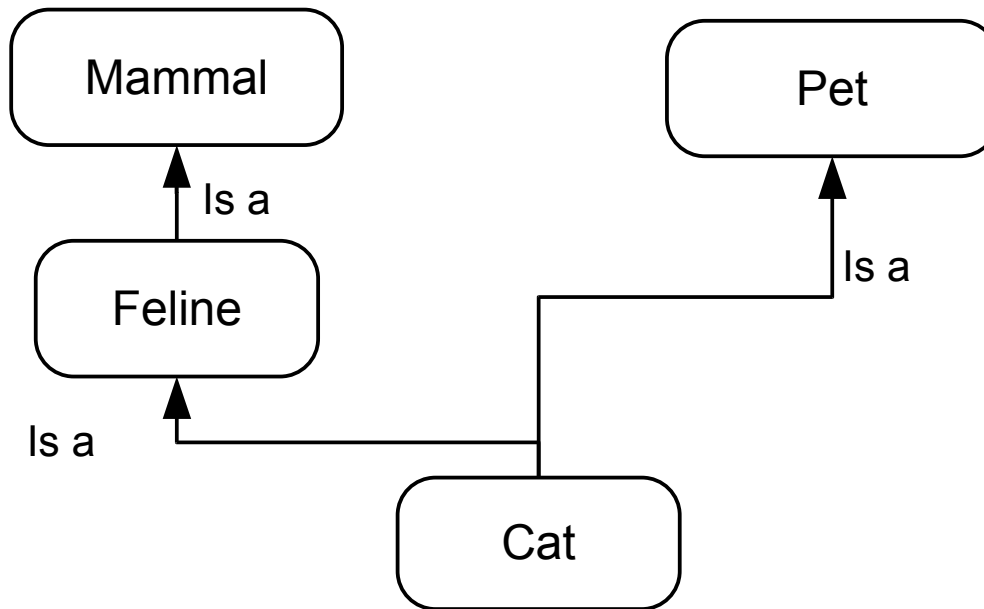
Complex models can be decomposed hierarchically:

any model can be recursively defined as an aggregate of sub-models



# O-O Principles III: Inheritance (Specialization)

- The relationship “is a” is defined between different classes
- The “child” class is a specialisation of the ancestor classes: it inherits all their features (in our case, variables, parameters, sub-models, and equations), and adds its own ones
- Multiple inheritance can be allowed:



- When defining classes through inheritance, care must be exercised, as the complete definition of the child class is scattered among the ancestors
  - Can be difficult to understand a posteriori
  - Can be difficult to modify or extend



# Introduction to the Modelica Language

# Introduction and Background

---

Between the end of the 70s and the first half of the 90s, many languages (and tools) have been defined for object-oriented modelling and simulation: Dymola, OMOLA, ASCEND, NMF, gPROMS, MOSES.

All of them are born in universities, where the first simulation tools are developed. Some of them later become commercial products (Dymola, gPROMS).

At the end of the 90s, some players decide to unite their forces and define a new language summarising all their past experiences.

The **language** is named Modelica, and its definition is property of a no-profit institution (the Modelica Association), composed by tool vendors and users, contributing to the development of the language and of a suite of standard model libraries.

Version 1.0 has been published in 1997. Other versions follow to improve and extend the original one (the latest is 3.2, released in 2010). Despite that, the language is quite stable, and most modifications are backwards-compatible.

Currently, there are several **tools** supporting Modelica to various degrees:

- Dymola (Dynasim, commercial)
- SimulationX (ITI, commercial)
- OpenModelica (PELAB, Univ. Linköping, open source)
- MathModelica (MathCore, commercial)
- MapleSim (MapleSoft, commercial)
- IDA (Equa Simulation, commercial)
- Amesim (Imagine, commercial)

# Classes

---

The basic construct of Modelica is the class (`class`), which defines how an object is made.

As in most object-oriented language, in the code it is possible to:

- **Define** a class, giving an abstract definition of an object (a sort of blueprint)
- **Instantiate** one or more instances of the classes, which will then be used to simulate some real objects

7 restrictions of `class` are defined to represent different kinds of objects:

- **model**: Describes the dynamic model of a object
- **block**: Describes a causal dynamic model (only input/output connectors)
- **function**: Describes a mathematical function (one algorithm defines the input/output mapping, no memory, states or side effects)
- **record**: Describes an aggregate of objects (without any equation relating them)
- **connector**: Describes a port
- **type**: Describes a variable type starting from the predefined ones (`Real`, `Boolean`, `Integer`)
- **package**: Describes a collection of re-usable models (a library)



# Connectors & Types

**connectors** define ports to connect a model with the outside world.

Example: electrical connector:

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

When two or more connector are connected, equality equations are generated for all the corresponding effort variables; an equation  $\text{sum}()=0$  is generated for all the corresponding **flow** variables.

**flow** variables are always assumed positive when entering the object.

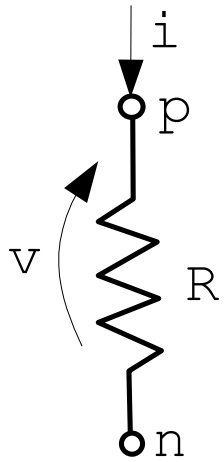
Variable types can be either predefined (**Real**, **Boolean**, **Integer**), or derived from predefined types, e.g.

```
type Voltage = Real(unit="V");
type Current = Real(unit="A");
type Temperature = Real (unit="K", min=0);
```

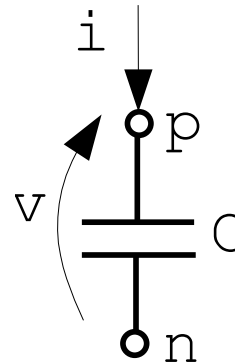
# Models

A `model` can contain variables, parameters, constants, other models, and equations relating them.

Example: resistor and capacitor models



```
model Resistor
  Pin p,n;
  Voltage v;
  Current i;
  parameter Resistance R;
equation
  v = p.v-n.v;
  i = p.i;
  0= p.i + n.i;
  v = R*i;
end Resistor;
```



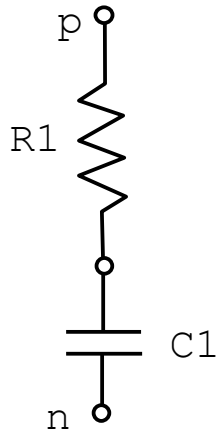
```
model Capacitor
  Pin p,n;
  Voltage v;
  Current i;
  parameter Capacitance C;
equation
  v = p.v-n.v;
  i = p.i;
  0= p.i + n.i;
  i = C*der(v);
end Capacitor;
```



Models in DECLARATIVE form!

# Aggregate Models

It is possible to define models containing other models:



```
model RCNet
  parameter Resistance Rnet;
  parameter Capacitance Cnet;
  Resistor R1(R=Rnet);
  Capacitor C1(C=Cnet);
  Pin p,n;
```

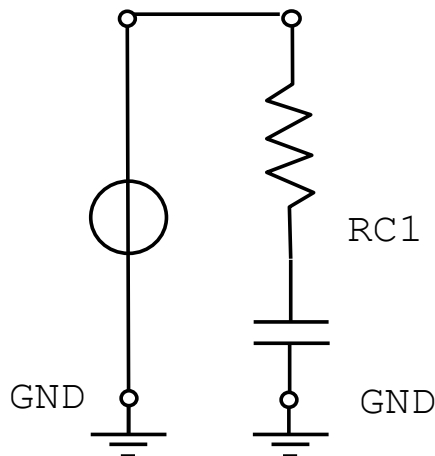
Modifiers  
(parameter propagation)

```
equation
  connect(R1.n, C1.p);
  connect(R1.p, p);
  connect(C1.n, n);
end RCNet;
```

Equivalent to:

$R1.n.v = C1.p.v;$   
 $R1.n.i + C1.p.i = 0;$

Rcnet can be used in turn to build a circuit model, and so on...



```
model SimpleCircuit
  RCnet RC1(Rnet=100, Cnet=1e-6);
  Vsource V0;
  Ground GND;
equation
  connect(RC1.n, GND.p);
  connect(RC1.p, V0.p);
  connect(V0.n, GND.p);
end SimpleCircuit;
```

# Arrays and Iterators

One can define  $n$ -dimensional arrays (vectors, matrices, ...) of objects:

```
Position p[3]; // 3D position vector
Real A[10,5]; // 10X5 matrix of real numbers
Real v[10];    // vector of 10 real numbers
Real w[5];     // vector of 5 real numbers
RCNet RC[4];   // array of four 4 RCnet networks
```

Multiplication of vector and matrices follow the usual rules of matrix algebra

**equation**

$$w = A * v;$$

It is possible to use **for** iterators to define equations with a repetitive structure:

```
model SeriesNetwork
  parameter N=3;
  RCnet RC[N];
  Pin p,n;
equation
  for i in 1:N-1 loop
    connect(RC[i].n,RC[i+1].p);
  end for;
  connect(RC[1].p, p);
  connect(RC[N].n, n);
end Series Network;
```



# Blocks

A **block** is a model having only *causal* connectors.

Blocks usually represent signal-processing objects, rather than physical devices.

Example: state-space representation of a generic linear dynamical system:

```
block StateSpace
  parameter Real A[:, :],
                B[size(A,1), :],
                C[:, size(A,2)],
                D[size(C,1), size(B,2)]=zeros(size(C,1),size(B,2));
  input      Real u[size(B,2)];
  output     Real y[size(C,1)];
  protected
    Real x[size(A,2)];
  equation
    assert(size(A,1)==size(A,2), "Matrix A must be square.");
    der(x) = A*x + B*u;
    y      = C*x + D*u;
end StateSpace
```

```
block TestSS
  StateSpace S(A=[1,2; 3,4], B=[0,1], C=[1,1]);
  equation
    S.u = sin(time);
end TestSS;
```

# Functions

**functions** are particular **models** without state (memory), thus representing functions in a *strict mathematical sense* (no side-effects). The input/output relationship is described in a causal way (algorithm)

Examples: factorial function and Bessel function

$$n! = \prod_{k=1}^n k$$

```
function fact
  input Integer n;
  output Real y ;
algorithm
  y := 1;
  for k in 2:n loop
    y := y*k;
  end loop;
end fact;
```

$$J_m(x) = \sum_{k=0}^{\infty} \frac{(-1)^k}{2^{2k+m} k! (m+k)!} x^{2k+m}$$

```
function Bessel
  input Integer m;
  input Real x;
  input N = 20 "Defaults to 20";
  output Real y ;
algorithm
  y := 0;
  for k in 0:N loop
    y := y + (-1)^k * x^(2*k+m) /
      (2^(2*k+m) * fact(k) * fact(m+k)) ;
  end loop
end Bessel;
```

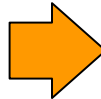
```
a = Bessel(2,x);
b = Bessel(1,x,5);
```

*Note: Functions  $\neq$  Equations!*

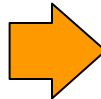
# Inheritance: Factoring Out Common Features

It is possible to define “child” objects by *extending* the definition of the “parent” objects:  
the child object inherits all the parents' attributes  
(variables, parameters, objects, equations)

```
partial model OnePort
  Pin p,n;
  Voltage v;
  Current i;
equation
  v = p.v - n.v;
  i = p.i;
  p.i = -n.i;
end OnePort;
```



```
model Resistor
  extends OnePort;
  parameter Resistance R;
equation
  v = R*i;
end Resistor;
```



```
model Capacitor
  extends OnePort;
  parameter Capacitance C;
equation
  C*der(v) = i;
end Capacitor;
```

In this way it is possible to “factor out” common parts in sets of similar models

It is also possible to extend models that are already fully functional by themselves

Modelica allows multiple inheritance: a given model can extend more than one parent object.

# Inheritance: Replaceable Objects I

A child object can replace some of the parents' attributes with their specializations

Example: an electronic circuit with DC/DC converters

- Depending on the specific needs of the simulation, you might need converter models having different degrees of accuracy, from the ideal transformer down to a detailed representation of the switched capacitor circuit
- All the models have something in common: the interface (four pins) and a parameter (the nominal output voltage)

We can define an abstract (partial) model `BaseDCDC` with the common features and the actual model implementations as derived objects:

```
partial model BaseDCDC
  Pin pa, na;
  Pin pb, nb;
  parameter Voltage vout;
end BaseDCDC;
```

```
model IdealDCDC
  extends BaseDCDC
equation
  pb.v - nb.v = vout;
  pa.i*(pa.v - na.v) + pb.i*(pb.v-nb.v) = 0;
  pb.i + nb.i = 0;
  pa.i + pb.i = 0;
end SimpleDCDC
```

```
model DetailedDCDC
  extends BaseDCDC
  // detailed implementation
  ...
end DetailedDCDC;
```



# Inheritance: Replaceable Objects II

Now we can define the template model of a circuit containing two converters, whose actual implementation will be decided later:

```
partial model BaseCircuit
  replaceable BaseDCDC Conv1, Conv2;
  // other components here
equation
  // connection equations here
end BaseCircuit;
```

Finally, it is possible to generate models of the circuit with different degrees of detail, by replacing the base converter models with the desired implementations. Conv1 and Conv2 can be described by any model inheriting from BaseDCDC, e.g:

```
model SimpleCircuit
  extends BaseCircuit(
    redeclare IdealDCDC Conv1,
    redeclare IdealDCDC Conv2);
end SimpleCircuit;
```

```
model MixedCircuit
  extends BaseCircuit(
    redeclare IdealDCDC Conv1,
    redeclare DetailedDCDC Conv2);
end MixedCircuit;
```

This very powerful feature of the Modelica language allows to deal with model variants in a clean, concise and consistent way, avoiding unnecessary code duplication.

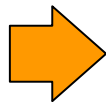
# Inheritance: Caveat Emptor!

---

In theory, inheritance is a very powerful mechanism, allowing to minimise unnecessary repetitions of code in highly structured sets of objects.

In practice, (as it happens with all O-O languages), overusing inheritance can lead to

- **Unreadable** code: the description of the functionality of a given object is scattered in a dozen or more partial models
- Code which is **difficult to modify/adapt**: child objects have a structure which is strongly constrained by the architecture of the ancestors, so that they might need to be rewritten from scratch if one wants to relax or modify any modelling assumption given for granted in the ancestors



Do not abuse inheritance!

# Conditional Equations

Inside equations, it is possible to define expressions whose value depend on a boolean expression (typically some inequality):

```
q_out = if level > 0 then K*sqrt(level) else 0 end if;
```

```
y = if          u < ymin then ymin  
    else if u > ymax then ymax  
    else                u      end if;
```

As an alternative, one can state which equation holds, depending on a boolean expression:

```
if level > 0 then q_out = K*sqrt(level)  
    else q_out = 0;
```

```
if          u < ymin then y = ymin  
else if u > ymax then y = ymax  
else                y = u  
end if;
```

This is expecially useful to define:

- Models with discontinuous behaviour (e.g. ideal diode, check valve)
- Models with different modelling options (depending on boolean parameters)

# Hybrid Models: Discrete-Time Dynamical System

It is possible to represent objects, whose variables do not vary continuously, but rather change only at certain *events*, i.e. when a certain boolean expression becomes true; for the rest of the time they remain constant.

These variables are identified by the **discrete** keyword in its declaration. Integer and Boolean variables (as well as their sub-types) are discrete by definition.

Discrete and continuous variables can co-exist in the same model (*hybrid* models).

Modelica supports time-events (known a-priori) and state-events (depending on the system state)

Example 1 (time-events): discrete-time linear dynamical system:

```
block DiscreteStateSpace
  parameter Real a, b, c, d;
  parameter Time Period;
  input Real u;
  discrete output Real y;
  discrete Real x;
equation
  when sample(0,Period) then
    x = a*pre(x) + b*u;
    y = c*pre(x) + d*u;
  end when;
end DiscreteStateSpace;
```

# Hybrid models: Event-Driven Models

Example 2: Flip-Flop (state-event)

```
block FlipFlop
  parameter Real Thr;
  input      Real Set;
  input      Real Reset;
  discrete output Real y;
equation
  when (Set>Thr or Reset>Thr)
    y = if Set>Thr then 1
        else 0
      end if;
  end when;
end FlipFlop;
```

Example 3: Relais

```
block Relais
  parameter Real Thr;
  input      Real Switch;
  output      Boolean y;
equation
  when (Switch > Thr)
    if pre(y)==true then y = false
      else y = true
    end if;
  end when;
end Relais;
```

# Physical Field Models

In some cases it happens to model objects interacting with a physical field, defined in their environment (thermal field, electrical field, gravitational field, etc.)

In these cases it is annoying to define explicitly the connection between each of the objects and the field model; it is possible to use the `inner` and `outer` constructs instead:

```
model Component
  outer Temperature T0; // defined outside the model
  Temperature T;
  parameter HeatCapacity C;
  parameter HeatTransferCoefficient h;
equation
  C*der(T)=h*(T0-T);
end Component

model Environment
  inner Temperature T0; // T0 is defined here
  Component c1, c2, c3;
equation
  T0=293+10*sin(time/(24*3600))
end Environment
```

Sophisticated fields can be modelled by declaring *models* or *functions* (instead of simple variables) as `inner/outer`.

# Graphical annotations

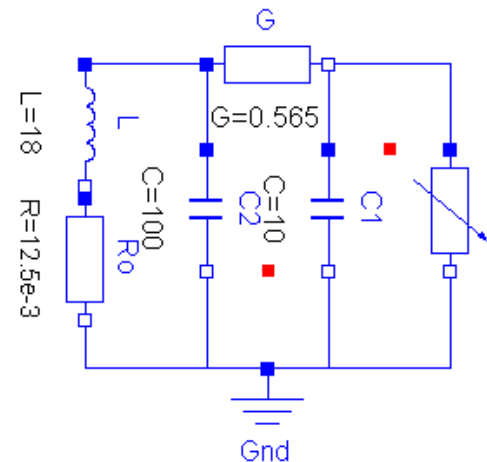
The Modelica language is a *textual* language (each Modelica model is a plain-text ASCII file); this favours the portability of models and the interoperability of the modelling tools (no binary files in obscure, proprietary formats!).

To allow using GUIs, Modelica allows to define a graphical layer (icons) for models, connectors, and connections.

The graphical appearance of each object is defined by an `annotation()` construct, in terms of graphical primitives (lines, circles, rectangles, text, bitmaps).

It is then possible to build composite models by *drag&drop*, and to connect the models by drawing lines between the corresponding connectors.

Example: electrical circuit:



Most tools (e.g. Dymola) allow to work on the same model both in text and graphics modes, which represent two different *views* on the same underlying object; the two views are always kept consistent with each other

# Libraries

---

To ease the re-use of previously developed models, it is possible to build hierarchically-structured libraries, using the **package** construct.

```
package Modelica
  package Electrical
    ...
  end Electrical;
  package Mechanics
    ...
  end Mechanics;
...
end Modelica;
```

A plant model can be built by instantiating models from the installed libraries, either by referencing to the full name, or using the import statement (or just by drag&drop, in which case the tools takes care of the naming)

```
model Circuit
  Modelica.Electrical.Analog.Basic.Resistor R1, R2;
  Modelica.Electrical.Analog.Basic.Capacitor C1, C2;
  ...
end Circuit;

model Circuit2
  import Modelica.Electrical.Analog.Basic.*;
  Resistor R1, R2;
  Capacitor C1, C2;
  ...
end Circuit2;
```



# Standard Libraries

---

The Modelica *language* is accompanied by the Modelica *library*, a large collection of standard libraries containing basic elements for the different fields of engineering:

## Modelica

- Blocks
- Constants
- Electrical
- Icons
- Math
- Mechanics
- Media
- Fluid
- Slunits
- Thermal
- ...

Other libraries have been developed for specific application fields. Some of them are *open source*, other are commercial.

The ThermoPower library is an open source library, developed at Politecnico di Milano, aimed at power plant and energy conversion systems.

# Modelica Tools – A short history

---

## 1997-2006

- **Dymola** (Dynasim AB) is de-facto the only tool supporting the whole standard in an industrial-strenght way.
- **MathModelica** (Mathcore Engineering AB) supports parts of the standard, for some years using code sub-licensed from Dynasim. MathCore lives mostly off consultancy rather than by selling tools
- **OpenModelica** (Linköping Universitet), open source compiler, developed as an academic tool for compiler theory research, not usable for any serious project

## 2006

- MathCore starts developing **MathModelica** on a branch of the OpenModelica source code base
- Dynasim is bought by Dassault Systèmes, which sees potential in Modelica technology to complement the CATIA PLM tool with a dynamic simulation module. Dymola still lives as a standalone product.

# Modelica Tools – A short history

---

## 2007-2008

- Version 3.0 del linguaggio is developed with active contributions from Dynasim, MathCore, Linköping Universitet, plus two new players: ITI GmbH (**SimulationX**) and MapleSoft (**MapleSim**). The goal is full support of the language, including advanced features for multibody and thermofluid systems.
- The Open Source Modelica Consortium (OSMC) is established, supporting the development of OpenModelica, with financial support from many companies and universities
- INRIA e LMS-Imagine fund the development of a french Modelica compiler, which covering only parts of the language

## 2009

- Dassault Systèmes releases CATIA V6 with a dynamic module based on Modelica and Dynasim technology. Dymola will keep being sold as a stand-alone product “until the user community will require that”
- MathCore joins OSMC e contributes to the front end. At the end of the year, a version of OMC whose front-end fully supports Modelica 3.1. This front end is now used by MathModelica.
- A total of **9 tools** including a parser and a Modelica compiler is presented at the 2009 Modelica Conference in Como.

# Modelica Tools – State of the art

---

**Dymola** is still the reference Modelica tool available

**SimulationX** is the closest competitor. Full support of multibody and thermofluid models (including Modelica.Media, Modelica.Fluid and ThermoPower) is planned for 2011.

**OpenModelica** has now a full-fledged and efficient front-end (Modelica code → flat equations). The back-end (the numerical solution of the equations) is still very primitive. Funded research work on the back-end is gradually improving the situation. Industrial users expect to use the tool for serious projects within one-two years.

**MathModelica** uses the OMC front end and a much better numerical back-end.

# References

---

- Home page Modelica Association: <http://www.modelica.org/>
- Modelica 3.2 Definition: <http://www.modelica.org/documents/ModelicaSpec32.pdf>
- Modelica Tutorial: <http://www.modelica.org/documents/ModelicaTutorial14.pdf>
- Proceedings Modelica Conferences  
<http://www.modelica.org/publications>
- Mattson, S.E., Elmqvist, H., Otter, M.: “Physical system modeling with Modelica”, *Control Engineering Practice*, v. 6, pp. 501-510, 1998.
- Tiller, M.: *Introduction to Physical Modeling with Modelica*. Kluwer, 2001.
- Fritzson, P: *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. Wiley 2003.