Modelica - A Unified Object-Oriented Language for System Modeling and Simulation

Peter Fritzson, Vadim Engelson

PELAB — Programming Environment Lab
Department of Computer and Information Science
Linköping University

Sweden

http://www.ida.liu.se/~pelab/modelica

Existing Modeling languages

- Block-oriented simulation languages
- Special purpose lectronic simulation programs
- Special multibody mechanical analysis tools

Problems:

- High performance is needed for simulation of complex systems
- Better technology for reusable components is needed
- Difficult to achieve true reusability in OO-modeling
- Gap between physical structure of the system and the model created by the tool.
- Difficult to integrate models consisting of elements from different domains

The Modelica Design Effort

- A general language for design of models of physical systems
- Multi-formalism, multi-domain
- Continuous and hybrid models
- International effort
- EUROSIM, Simulation in Europe
  - **EUROSIM, Technical Committee 1**
  - **http://www.Dynasim.se/Modelica**
- Industrial support
- Aim: to become a de facto standard in object oriented modeling of physical systems

EuroSim Technical Committee 1 designing Modelica

Hilding Elmqvist, Dynasim AB. (Chairman)

Francois Boudaud, Gaz de France, Paris, France

Jan Broenink, Univ. Twente, Netherlands

Dag Brück, Dynasim AB, Lund, Sweden

Thilo Ernst, GMD-First, Berlin, Germany

Peter Fritzson, Linköping Univ., Sweden

Alexandre Jeandel, Gaz de France, Paris, France

Kai Juslin, VTT, Espoo, Finland

Matthias Klose, GMD-First, Berlin, Germany

Sven-Erik Mattson, Lund Univ. Sweden

Martin Otter, DLR, Oberpfaffenhofen, Germany

Per Sahlin, Brisdata AB, Stockholm, Sweden

Hubertus Tummescheit, DLR, Cologne, Germany

Hans Vangheluwe, Univ. of Gent, Belgium

Modelica features

- Non-causal modeling

  Based on equations instead of assignment statements. Better reuse of classes since equations does not specify data flow direction.

- Multidomain modeling capability

  Electrical, mechanical, thermodynamic, hydraulic etc. Model components correspond to physical objects in real world.

- Object-orientation

  OO, templates, and general subtyping are supported within the *class* construct. Supports reuse of components and evolution of models.

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.

- An **object** is collection of variables, equations, functions that share a state (= instance variables).

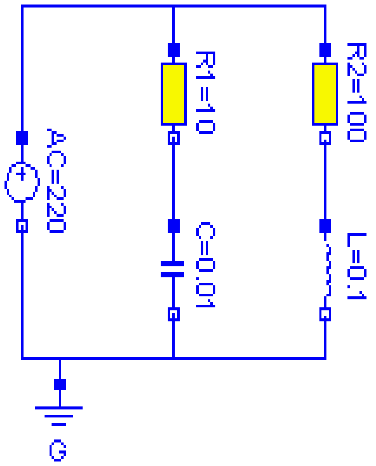
- Classes = templates to create objects

- Inheritance = reuse of equations, functions, variables

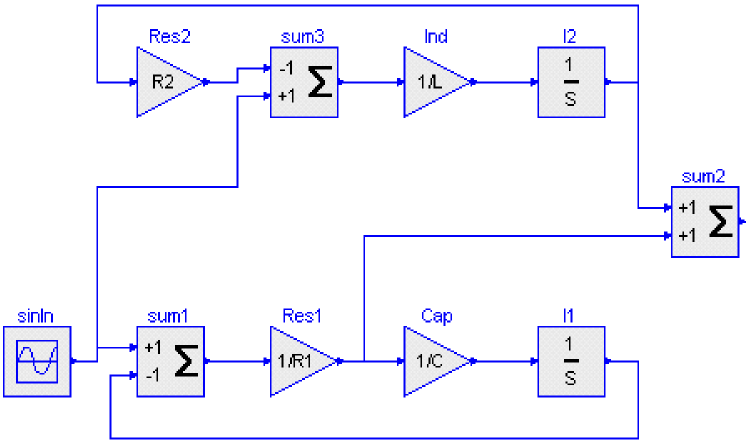
Non-causal classes supports better reuse of modeling/design knowledge than traditional classes

Advantages of non-causal physical modeling supported by Modelica

- Non-causal object oriented circuit model example



- Block-oriented (causal) circuit model



- Disadvantages of the causal model
  - **Physical topology lost**
  - **Resistor implementation is context-dependent - reuse hard**
  - **Difficult to maintain**

Examples of early object oriented modeling/design languages and tools

- Dymola, Omola
- ObjectMath
- NMF, U.L.M.
- Ascend, gProms
- SIDOPS+, Smile

Significant experience of using these in many different application domains.

Modelica extends and replaces these formalisms

Non-causal modeling/design

- What is non-causal modeling/design?

- Why does it increase re-use?

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

- Example: a resistor equation:

```
R*i = v;
```

can be used in two ways:

```
i := v/R;
v := R*i;
```

Modelica Semantics

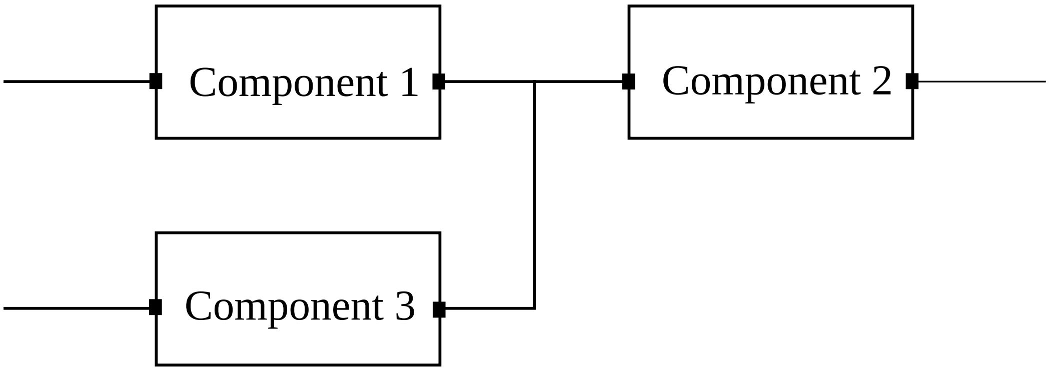
Modelica is truly equation-based:

- Assignment statements are represented as equations
- Connections between objects generate equations
- Attribute assignments can be represented as equations

The semantics rules describe definition expansion, type structures, etc.

A formal definition of Modelica semantics specified in *Natural Semantics* is being developed by us using the RML tool (http://www.ida.liu.se/~pelab/rml).

Object/Component connection diagrams



- Every rectangle represents a ***physical component***, e.g. resistor, mechanical gear, pump.

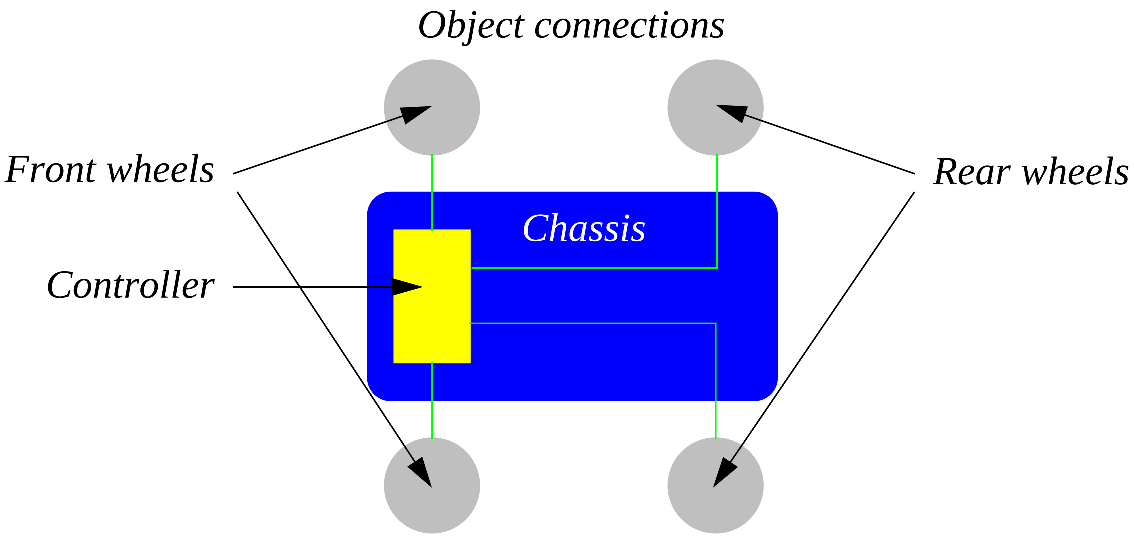
- The connections corresponds to the real, ***physical connections***. For example:
  electrical wire, stiff mechanical connections, heat exchange between components.

- Variables at the ***interface*** points define the interaction between objects.

- A component is modeled ***independently*** of the environment. That is, for the definition of the component only ***interface variables*** and ***local variables*** are used!

- A component consists of ***other connected components*** (hierarchical modeling), or is described by ***equations.***

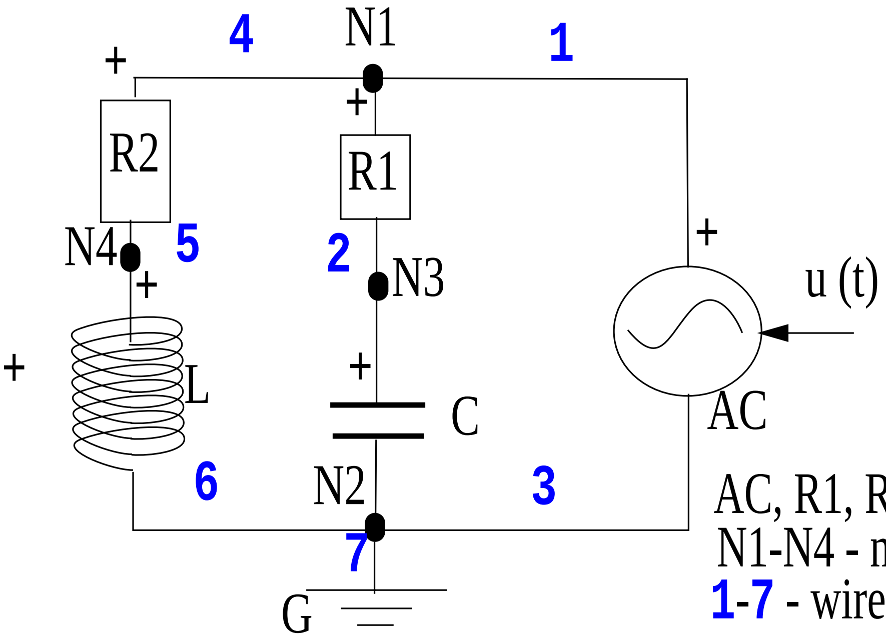
Simplified example car model in Modelica using a connection diagram



```
class Car "Abstract class to connect subclasses"
  Wheel          w1,w2,w3,w4  "Four wheels";
  Chassis        chassis      "Chassis";
  CarController  controller   "Car controller"; ...
```

- Visualization using the Vega tool on Silicon Graphics:

Circuit example — a Modelica model



Legend

AC, R1, R2, L, C, G - components
N1-N4 - nodes
1-7 - wires
- positive pins
u (t) = VA sin(2π f t)
(alternate voltage source)

```
class circuit
   Resistor  R1(R=10);
   Capacitor C(C=0.01);
   Resistor  R2(R=100);
   Inductor  L(L=0.1);
   VsourceAC AC;
   Ground    G;

 equation
  connect (AC.p, R1.p);    // 1, Capacitor circuit
   connect (R1.n, C.p);     //     Wire 2
   connect (C.n, AC.n);     //     Wire 3
  connect (R1.p, R2.p);    // 2, Inductor circuit
   connect (R2.n, L.p);     //     Wire 5
   connect (L.n,  C.n);     //     Wire 6
   connect (AC.n, G.p);     // 7, Ground
 end circuit;
```

Component details

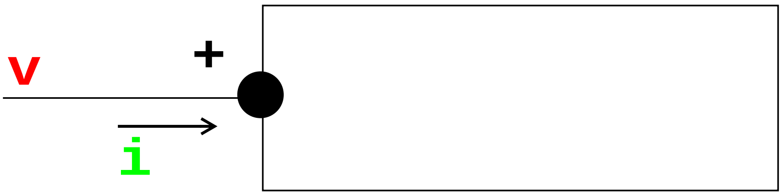
- Type definitions

```
type Voltage = Real(Unit="V");
type Current = Real(Unit="A");
```

  - **Good tools will support unit checking of equations**

- Connectors specify external interfaces for interaction

  Pin is a connector class that can be used for electrical components which have pins



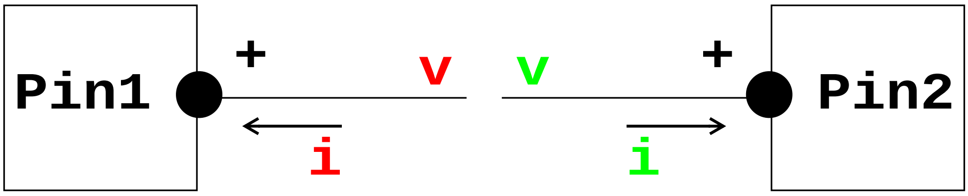
```
connector Pin
  Voltage       v;
  flow Current  i;
end;
```

The keyword *flow* indicates that all currents in connected pins are summed to zero, according to Kirchoff’s 2:nd law

Connecting components

- Example

  Connecting two components which have pins:

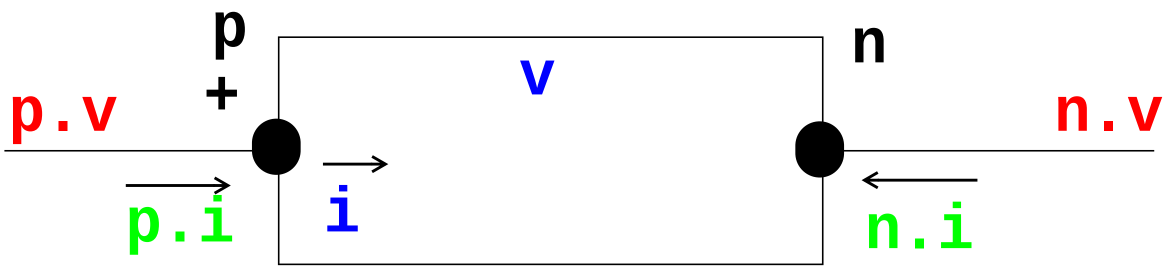


- Equations produced by the connection:

```
Pin1.v = Pin2.v
Pin1.i + Pin2.i = 0
```

Describing common properties by classes

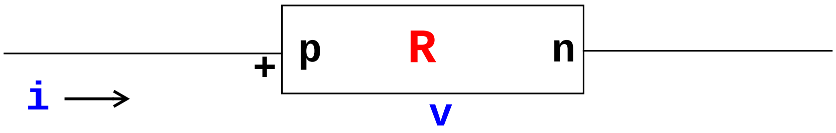
- TwoPin — electrical components with two pins



```
partial class TwoPin
  "Superclass of elements with two electrical pins"

  Pin      p, n;
  Voltage  v;
  Current  i;

equation
  v = p.v - n.v;
  p.i + n.i = 0;
  i = p.i;

end;
```

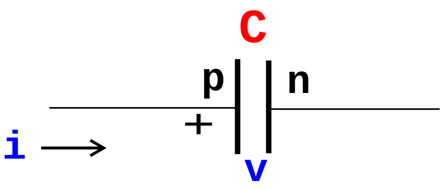
Component classes

- Resistor



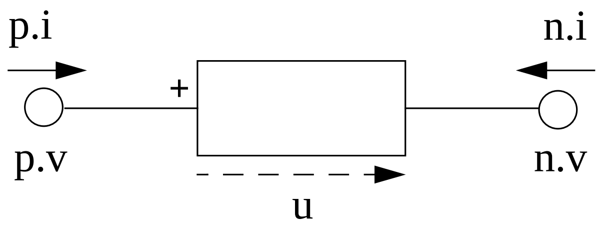
```
class Resistor "Ideal electrical resistor"
  extends TwoPin;
parameter Real R(Unit="Ohm") "Resistance";

equation
  R*i = v;

end;
```

- Capacitor



```
class Capacitor "Ideal electrical capacitor"
  extends TwoPin;
  parameter Real C(Unit="F") "Capacitance";

equation
  C*der(v) = i;
end;
```

Examples of simple electrical components

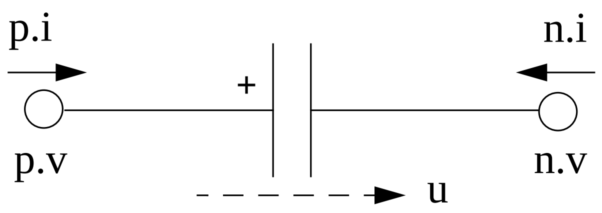


0 = p.i + n.i

u = p.v - n.v

i = p.i

u = R*i

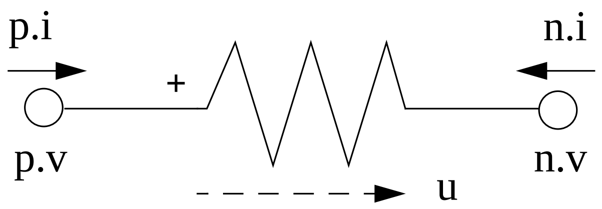


0 = p.i + n.i

u = p.v - n.v

i = p.i

i = C*der(u)

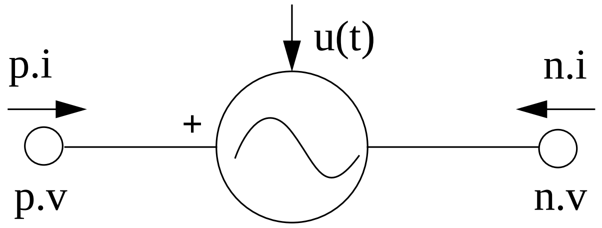


0 = p.i + n.i

u = p.v - n.v

i = p.i

u = L*der(i)

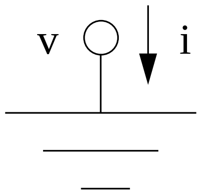


0 = p.i + n.i

u = p.v - n.v

i = p.i

u = A*sin(w*t)



v = 0

Equations from the simple circuit

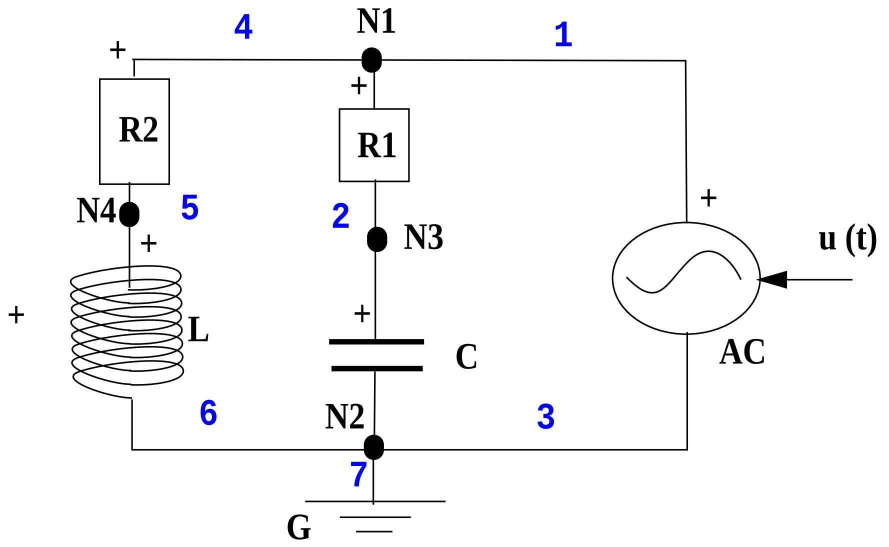


Table 1:

C	0=AC.p.i+Ac.n.i AC.v=Ac.p.v-AC.n.v AC.i=AC.p.i AC.v=AC.VA*sin(2*PI*AC.f*time);	L	0=L.p.i+L.n.i L.v=L.p.v-L.n.v L.i=L.p.i L.v = L.L*L.der(i)
R1	0=R1.p.i+R1.n.i R1.v=R1.p.v-R1.n.v R1.i=R1.p.i R1.v = R1.R*R1.i	G	G.p.v = 0

R2	0=R2.p.i+R2.n.i
	R2.v=R2.p.v-R2.n.v
	R2.i=R2.p.i
	R2.v = R2.R*R2.i
		wires	R1.p.v=AC.p.v // wire 1
			C.p.v=R1.n.v // wire 2
			AC.n.v=C.n.v // wire 3
			R2.p.v=R1.p.v // wire 4
			L.p.v=R2.n.v // wire 5
			L.n.v=C.n.v // wire 6
			G.p.v= AC.n.v // wire 7

C	0=C.p.i+C.n.i C.v=C.p.v-C.n.v C.i=C.p.i C.i = C.C*C.der(v)	flow at nodes	0=AC.p.i+R1.p.i+R2.p.i //1 0=C.n.i+G.i+AC.n.i+L.n.i //2 0=R1.n.i+ C.p.i // 3 0 =R2.n.i + L.p.i // 4

Sought: Transformation to state space form

$$\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}, t)$$

That is,
from a given state **x**, the derivative of the state, $\dot{x}$, should be calculated.

Here:

given: C.**u**, L.**i, t** (constants: R1.***R*,** R2.***R***, C.***C***, L.***L***, A.***A***, A.***w***)
sought: **der**(C.**u**), **der**(L.**i**)

Here are the 31 unknowns

R1.**p.i,** R1.**n.i,** R1.**p.v,** R1.**n.v,** R1.**u,**
R2.**p.i,** R2.**n.i,** R2.**p.v,** R2.**n.v,** R2.**u,**
C.**p.i,** C.**n.i,** C.**p.v,** C.**n.v,** **der**(C.**u**),
der(L.**i**), L.**n.i,** L.**p.v,** L.**n.v,** L.**u,**
A.**p.i,** A.**n.i,** A.**p.v,** A.**n.v,** A.**u,**
g.**i,** g.**v** R1.i R2.i L.i
C.i

Solution method

1. Use the equations that contains the unknowns you want to calculate (here: **der**(C.**u**), **der**(L.**i**))

    der(C.**u) =** C.**p.i/**C.***C***

    der(L.**p.i) =** L.**u/**L.***L***

2. Use other equations to calculate the unknowns in the equations from 1.

    C.**p.i**   = R1.**u**/*R1.**R***

    R1.**u**   **=** R1.**p.v - **C.**u**

    R1**.p.v** **=** A.***A****sin(A.***w****t**)

    L.**u**   = R1.**p.v -** R2.**u**

    R2.**u**   **=** *R2.**R****L.**p.i**

3. Sort the equations in dependency order

    given: C.**u**, L.**p.i, t**

    R2.**u**   **=** *R2.**R****L.**p.i**
    R1.**p.v**   **=** A.***A****sin(A.***w****t**)
    L.**u**   = R1.**p.v -** R2.**u**
    R1.**u**   **=** R1.**p.v -** C.**u**
    C.**p.i**   = R1.**u**/*R1.**R***
    der(L.**p.i) =** L.**u/**L.***L***
    der(C.**u) =** C.**p.i/**C.***C***

4. Generate code and solve numerically

Automated solution method

- Equations are sorted, symbolically simplified, and translated to efficient C/C++ code
- This method is completely automated and handles tens of thousands of equations efficiently
- Currently primarily for Differential-algebraic equations. Ongoing work for partial differential equations

Time in Modelica

The behaviour evolves as a function of time.
A predefined variable **time** is used:

```
class VsourceAC “Sin-wave voltage source”
   extends TwoPin;
   parameter Voltage VA = 220 “Amplitude”;
    parameter Real f(unit=”Hz”) = 50
                                    “Frequency”;
   constant  Real PI=3.141592653589793;
equation
   v = VA*sin(2*PI*f*time);
end VsourceAC;
```

The construct **der(v)** means the time derivative of **v**.

Functions in Modelica

Sometimes there is need for model components expressed in algorithmically

```
function PolynomialEvaluator
        input Real a[:];
           // array, size defined at run time
        input  Real x;
        output Real y;
      protected
        Real   xpower;
      algorithm
        y := 0;
        xpower := 1;
        for i in 1:size(a, 1) loop
          y := y + a[i]*xpower;
          xpower := xpower*x;
        end for;
      end PolynomialEvaluator;
```

Functions have input parameters and output results.

Subtypes in Modelica

According to several type systems by Abadi & Cardelli:

Class A is a subtype of class B iff

- Class A contains all public variables of B
- The types of these variables are subtypes of types of corresponding variables in B.

Where is subtyping used?

- Initialization of variables: variable A can be initialized by B
- Redeclarations (discussed later)

Redeclarations

The type of class member can be changed when the class is inherited.

Redeclaration example in Modelica

Two classes, Resistor and TempResistor:

```
class Resistor “Ideal electrical resistor”
   extends TwoPin;
   parameter Real R(unit=”Ohm”) “Resistance”;
 equation
   R*i = v;
end Resistor;

class TempResistor
   extends TwoPin
   parameter Real R, RT, Tref ;
   Real T;
 equation
   v=i*(R+RT*(T-Tref));
end TempResistor
```

Note that **TempResistor** is a subtype of **Resistor**.

Redeclaration example (continued.)

There is a class SimpleCircuit:

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

The types of variables R1 and R2 can be replaced:

```
class RefinedSimpleCircuit = SimpleCircuit(
       redeclare TempResistor R1,
       redeclare TempResistor R2);
```

The result is equivalent to:

```
class RefinedSimpleCircuit
    TempResistor R1(R=100),
    TempResistor R2(R=200),
    Resistor R3(R=300);
 equation
    connect(R1.p, R2.p);
    connect(R1.p, R3.p);
end RefinedSimpleCircuit
```

Comparison of redeclaration with C++ templates

In C++ classes can be defined as below:

```
template <class TResistor, class TResistor1>
class SimpleCircuit {
  public:
SimpleCircuit(){
             R1.R=100.0;
             R2.R=200.0;
             R3.R=300.0; };
TResistor  R1;
TResistor1 R2;
Resistor   R3;
}
```

and then declared by

```
SimpleCircuit<TempResistor,TempResistor>
```

In C++ it is necessary to explicitly specify which two resistors are replaced.

Redeclaration in Java

Assume that:

class **TempResistor** extends **Resistor**

and

class **RefinedSimpleCircuit** extends **SimpleCircuit**

then the problem is solved at run-time:

```
class RefinedSimpleCircuit extends
      SimpleCircuit
{ public
  RefinedSimpleCircuit() {
     R1=new TempResistor();
     R2=new TempResistor();
}}
```

Java - using `Object`.

Otherwise, the problem is solved by using casting:

```
class SimpleCircuit
{  public  SimpleCircuit() {
 R1=new Resistor();  ((Resistor)R1).R=100.0;
 R2=new Resistor();  ((Resistor)R2).R=200.0;
 R3=new Resistor();  ((Resistor)R3).R=300.0;};
    Object R1, R2, R3;
    };
};

class RefinedSimpleCircuit extends SimpleCircuit
{ public
  RefinedSimpleCircuit() {
  R1=new TempResistor();
  R2=new TempResistor();
  ((TempResistor)R1).RT=0.1;
  ((TempResistor)R1).TRef=20.0;}
};
```

Advanced Modelica Modeling Features

- matrix equations
  - **for mechanical models, control systems, etc.**
- arrays of components and regular connection patterns
  - **such as a distillation column**
- class parameters
  - **reuse of a model diagram but replacing component models**
- discontinuities, events and event synchronization
  - **for modeling friction, sampled control systems, etc.**
- algorithms and functions
  - **for procedural style of modeling/design**
- units and quantities
  - **for consistency checks**
- graphical annotations
  - **also icons and model diagrams become portable**
- Modelica base library
  - **standard variable and connector types promotes reuse**

Plans

- The aim is to make Modelica a de-facto standard
- Modelica version 1.0
    - **Differential Algebraic Equations (DAE)**
    - **Hybrid models**
    - **Published September 1997 (www.dynasim.se/Modelica)**
- Modelica version 1.1
    - **Semantics formally defined**
    - **Standard libraries**
    - **Expected Sept.-Oct 1998**
- Modelica version 2
    - **Support for partial differential equations**
    - **Mathematical modeling of dynamic object creation, etc.**
- Planned books
    - **Modelica language**
    - **Modelica libraries**
- Tools, ...

Conclusion

- Modelica is a new object-oriented design language for modeling/design of complex systems, usually for the purpose of simulation
- An international technical committee, currently under EuroSim, is standardizing and designing Modelica
- The language has a good chance of becoming the next generation simulation language
- Modelica is strongly typed and can be compiled to very efficient C/C++ code
- Ongoing efforts to generate efficient parallel code from Modelica