Modelica
A Language for Physical System Modeling

Hubertus Tummescheit, Lund University
Hilding Elmqvist and Sven Erik Mattsson, Dynasim
Contents

- Introduction
- Modelica Design Effort
- Model Composition Diagrams
- Modelica Language Details
- Symbolic Processing for Efficient Simulation
- Object Oriented Modeling
- Experiences: Modelica at Ford
- Simulation with Dymola
- Conclusions
The Need

- Analyzing and synthesizing more complex systems
- Linking previously unrelated domains
- Tighter coupling between subsystems
- Built-up from several engineering domains
The Current Situation

- Application specific simulation tools and languages
  - Mechanics (ADAMS, SIMPACK, ...),
  - Electronic (Spice, Saber, VHDL-AMS, ...),
  - Hydraulic (Flowmaster)
  - Often closed set of model components
  - Not suited for multi-domain modeling

- General purpose simulation tools
  - Simulink, ACSL, System Build, EASY5, etc
  - Manual derivation of equations needed
  - Not suited for physical modeling
The Knowledge

Model knowledge is stored in books and human minds computers can not make use of that knowledge
The Form - Equations

Equations were used in the third millenium B.C.

Equality sign was introduced by Robert Recorde in 1557

\[ 14.2 \text{e} \rightarrow 15.9 \rightarrow 71.9 \]

Newton (Principia, vol. 1, 1686) still wrote text:
“The change of motion is proportional to the motive force impressed; ...”

\[ \frac{d}{dt} (m \cdot v) = F_i \]

CSSL (1967) introduced special form of “equation”:
variable = expression
\[ v = \text{INTEG}(F) \div m \]

Computer languages usually do not allow equations
The Heritage

- Analog computing paradigm
- Block Diagrams
  - Computer-oriented modeling
  - Ineffective for humans performing physical modeling
The Physical Structure

- Physical **objects**: resistors, capacitors, pumps, valves, tanks, bodies, joints, etc
- Couplings: electrical wires, shafts, weldings, pipes, etc
The Solution

- New Methodology
  - Object-oriented modeling
  - Equation based modeling
- Formal language
  - For sharing and reuse of modeling knowledge
  - For long-term storage of modeling knowledge
  - Designed by experts from many fields

- Modelica
Some Applications

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

- Design of a language allowing reuse of physical models
- Allow heterogeneous models
- Unify object-oriented modeling languages
  - Dymola, Omola, NMF, U.L.M., ObjectMath, Smile, etc
- According to modern language design principles
- Declarative instead of procedural
- Make sure efficient simulation can be achieved
Organization of Modelica Design

- Eurosimm Technical Committee 1
- Technical Chapter of SCS
- Non-Profit Modelica Association
Modelica Status

- Design started September 1996
- Modelica version 1.0 - September 1997
- Modelica version 1.3 - December 1999
- 21 meetings - each 3 days
- Ongoing effort, open for participation
- > 25 members of design group
- > 120 members of interest group
- Libraries and tools are available
Model Composition Diagrams

- Decomposition and Abstraction
- Use of models from libraries
- High level modeling by Composition
  - Instantiation
  - Parameter modifications
  - Connectors
  - Connections
The Art of Decomposition

- Connected subsystems
- Determine boundaries - isolate connection points control volumes, free body diagrams, ...
- Make abstraction
- Model connections
- Model each subsystem
Decomposition and Abstraction - Automatic Gearbox
Models of Connection Nodes

- Small - neglect mass and extent
  \[ \frac{d}{dt}(m \cdot v) = F_i \quad \xrightarrow{m=0} \quad F_i = 0, \quad r_i = r_j \]

- Small - neglect volume and losses
  \[ \frac{d}{dt}(\rho \cdot V) = f_{in} - f_{out} \quad \xrightarrow{v=0} \quad \pm f_i = 0, \quad p_i = p_j \]

- Kirchhoff current and voltage laws

- Two basic laws: \( I_i = 0, \quad V_i = V_j \)
  Variables sum to zero  Variables are equal
Model Libraries - for Reuse
Drag and Drop Composition
Hierarchical Composition Diagram for a Model of an Industrial Robot

Srel = n*n' + (identity(3) - n*n')*cos(q) - skew(n)*sin(q);
wrela = n*qd;
zrela = n*qdd;
Sb = Sa*Srel';
r0b = r0a;
vb = Srel*va;
wb = Srel*(wa + wrela);
ab = Srel*aa;
zb = Srel*(za + zrela + cross(wa, wrela));
fa = Srel'*fb;
ta = Srel'*tb;
Modelica Language Details

- Equations
- Object-oriented model structure
- Variable types
- Classes (model, record, connector, block)
- Arrays and Matrices
- Functions and Algorithms
- Hybrid Modeling
- Class Parameters
Equations

The general form needed:

\[ expression = expression \]
\[ R \times i = v \]  \hspace{1cm} (Ohm’s law)

The “unknown” depend on the model’s environment:

\[ i := \frac{v}{R} \]
\[ v := R \times i \]
\[ R := \frac{v}{i} \]

or several “unknowns” (system of simultaneous equations)

\[ \varepsilon := R \times i - v \]  \hspace{1cm} (residue)
Object-oriented model structure

- Separation of concerns
  Control volumes, free-body diagrams
- Hierarchical decomposition
- Non-causal connections
  **No arrows**
- Based on first principles
- Connection properties
  variables equal or summed to zero
Modelica by Example: Drive Train

Rigidly coupled Inertias

Pt controller

motor

n=100

Jl=10

Vs

emf

La=0.05

Ra=0.5

G

Jm=1.0E-3
model MotorDrive

PI controller;
Motor motor;
Gearbox gearbox(n=100);
Shaft Jl(J=10);
Tachometer wl;

equation
connect(controller.out, motor.inp);
connect(motor.flange, gearbox.a);
connect(gearbox.b, Jl.a);
connect(Jl.b, wl.a);
connect(wl.w, controller.inp);

end MotorDrive;
Variables

```modelica
type Angle = Real(quantity = "Angle", unit = "rad", displayUnit = "deg");
type Torque = Real(quantity = "Torque", unit = "N.m");
```

- **Predefined data types:**
  Real, Integer, Boolean, String
- **Attributes:**
  quantity, unit, displayUnit, nominal, start, min, max
- **Modelica base library**
  ISO standard quantities (450+ predefined types)
Connectors

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

connector Flange
Angle r;
flow Torque t;
end Flange;

connect (motor.flange, gearbox.a);

- Group of variables describing interaction
- Connected flow variables are summed to zero
- Other variables set equal
Partial Models and Inheritance

```plaintext
partial model TwoPin
  Pin p, n;
  Voltage v;
 equation
    v = p.v - n.v;
    p.i + n.i = 0;
end TwoPin;

model Resistor "Ideal resistor"
  extends TwoPin;
  parameter Resistance R;
 equation
    R*p.i = v;
end Resistor;
```
Non-Causal Connections
versus Block Diagrams

• Gearbox, inertia of motor and load are combined into a gain block

• Manual conversion
  involving differentiation and
  solving linear system of equations

Block diagrams are not suitable for large scale physical modeling
Arrays and Matrices

- Multi-dimensional arrays
- \{\} is the array constructor
  \text{Real } A[2,2,2] = \{\{\{1, 2\}, \{3, 4\}\}, \{11, 12\}, \{13, 14\}\}\};
- Matlab compatible [ ]
  \{[1, 2; 3, 4], [11, 12; 13, 14]\}
- Adjustable size: \text{Real } A[:, :, :];
- Usual matrix operators
Matrices and Blocks

```plaintext
partial block SISO "Single Input/Single Output block"
  input  Real u "input";
  output Real y "output";
end SISO;

block TransferFunction
  extends SISO;
  parameter Real a[:]={1, 1} "Denominator";
  parameter Real b[:]={1} "Numerator";
protected
  constant Integer na=size(a, 1);
  constant Integer nb(max=na) = size(b, 1);
  constant Integer n=na-1 "System order";
  Real b0[na] = cat(1, b, zeros(na - nb)) "Zero expanded b vector.";
  Real x[n] "State vector";
  equation
    // Controllable canonical form
    der(x[2:n]) = x[1:n-1];
    a[na]*der(x[1]) + a[1:n]*x = u;
    y = (b0[1:n] - b0[na]/a[na]*a[1:n])*x + b0[na]/a[na]*u;
end TransferFunction;
```
Functions

- Input, output and local variables
- The same matrix facilities as in equations
- for, while, if-statement
- Pure function (no side-effects)
  - initialization of locals and outputs
- Calling
  \[(y_1, y_2, \ldots) = f(u_1, u_2, \ldots);\]
- Sorting of call among equations
- External C and Fortran functions
  - Automatic checking of type and size of arguments
  - Allocation of work arrays
Example - Functions

function polynomialMultiply
  input Real a[:], b[:];
  output Real c[:] = zeros(size(a,1)+size(b, 1) - 1);
algorithm
  for i in 1:size(a, 1) loop
    for j in 1:size(b, 1) loop
      c[i+j-1] := c[i+j-1] + a[i]*b[j];
    end for;
  end for;
end polynomialMultiply;
Example - External Functions I

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

Assumes external C-function:

extern void (polynomialMultiply)(double const *, int ,
                               double const *, int ,
                               double *, int );
Example - External Functions II

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

Assumes external C-function:

extern void (polmult)(double const *, double const *, double *,
    int, int);
Example - External Functions

function BilinearSampling

"Slicot function for Discrete-time <- continuous-time systems conversion by a bilinear transformation."

input  Real alpha=1, beta=1;
input  Real A[:, size(A, 1)], B[size(A, 1), :],
       C[:, size(A, 1)], D[size(C, 1), size(B, 2)];
input  Boolean isContinuous = true;
output Real Ares[size(A, 1), size(A, 2)]=A, // Ares is in-out
        Bres[size(B, 1), size(B, 2)]=B,
        Cres[size(C, 1), size(C, 2)]=C,
        Dres[size(D, 1), size(D, 2)]=D;

output Integer info;
protected

  Integer iwork[size(A, 1)]; // Work arrays
  Real   dwork[size(A,1)];
  String c2dstring=if isContinuous then "C" else "D";
  external "Fortran 77" ab04md(c2dstring,size(A,1),size(B,2),size(C,1),
                              alpha,beta,Ares,size(Ares,1),Bres,size(Bres,1),
                              Cres,size(Cres,1),Dres,size(Dres,1),
                              iwork,dwork,size(dwork,1),info);

end BilinearSampling;
Functions: Current Development

• Automatic differentiation of Modelica functions
  – (needed for index reduction)
  – integrates functions completely into symbolic processing

• Jacobians to external functions:
  – improves efficiency
  – easier, more efficient integration of “legacy code”
Algorithms

- “Inline” function calls
- Automatic detection of inputs and outputs
- Sorted among equations
- Multiple assignment to the same variable allowed
Example - algorithm

**algorithm**

\[ dx := -a*x+b*u; \]

**if** \( x \geq 10 \) and \( dx \geq 0 \) **then**

\[ \text{der}(x) := 0; \]

**else**

\[ \text{der}(x) := dx; \]

**end if;**

Elsewhere:

**equation**

\[ u = \sin(\text{time}); \]
Hybrid Modeling

Discontinuities
\[ y = \text{if } u > \text{limit} \text{ then limit else } u; \]

When statement (instantaneous equation)
\[
\text{when } \{\text{condition1, condition2, ...}\} \text{ then equations end when;}
\]

Operators
• \( \text{pre}(x_d) \rightarrow x_d(t^-) \)
• \( \text{reinit}(x_c, \text{expr}) \rightarrow x_c(t^+) = \text{expr} \)

Translation to discrete events for efficient simulation
• crossing functions introduced (\text{u-limit})
• allows interpolation to find time of event
• solution of mixed integer and real systems of equations
Ideal Diode

Parametric description of non-linearities

![Diode symbol]

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

The diagrams illustrate the concept of friction, showing the relationship between force, acceleration, and velocity. The graphs depict the friction forces acting on different masses over time. The equations and symbols used in the diagrams are consistent with the principles of friction, including the peak friction force and the sliding friction force.
Hybrid Modeling: Impulses

Last Modelica Meeting:
introduce \texttt{impulse}(condition,equation) as a built-in operator

- Handling of impulses in a declarative and physical way
- symbolic manipulation of equations with Dirac impulses
- result: system of equations for all variables affected by the impulse, using Pantelides Algorithm and dummy Derivatives
  - in simple cases: symbolic solution
  - otherwise: special system of equations for \( t^+ \) after impulse
- Problems: handle combinatoric explosion of many impulses, depending on where the impulse appears, different equations have to be differentiated
Class Parameters

• Redeclare **class**
  – Replace the class of many components
  – Checking for consistency
  – Keep connections

• Redeclare **component**
  – Individually change class of a component

• Compare C++ templates and Ada generics
Class Parameters

- Redeclare component
- Individually change class
- Keep connections and parameters
- Checking for consistency
class C
   replaceable GreenClass comp1(p1=5);
   replaceable YellowClass comp2;
   replaceable GreenClass comp3;
   connect(...);
end C;

class C2 =
   C(redeclare RedClass comp1,
      redeclare GreenClass comp2);

Equivalent to

class C
   RedClass comp1(p1=5);
   GreenClass comp2;
   GreenClass comp3;
   connect(...);
end C;
Example - redeclare component

```plaintext
model MotorDrive
    replaceable PI controller;
    Motor motor;
    Gearbox gearbox(n=100);
    Shaft Jl(J=10);
    Tachometer wl;

equation
    connect(controller.out, motor.inp);
    connect(motor.flange, gearbox.a);
    connect(gearbox.b, Jl.a);
    connect(Jl.b, wl.a);
    connect(wl.w, controller.inp);
end MotorDrive;

model MotorDrive2 = MotorDrive
    (redeclare AutoTuningPI controller);
```
Class Parameters

- Redeclare \textbf{class}
- replace the class of many components
Modelica realization

class C
    replaceable class ColouredClass = GreenClass;
    ColouredClass comp1(p1=5);
    YellowClass comp2;
    ColouredClass comp3;
    connect(...);
end C;

class C2 =
    C(redeclare class ColouredClass = BlueClass);

Equivalent to

class C
    BlueClass comp1(p1=5);
    YellowClass comp2;
    BlueClass comp3;
    connect(...);
end C;
Example - redeclare class

partial block SISOController
    input Real ref;
    input Real inp;
    output Real out;
end SiSOController;

model MotorDrive2
    replaceable block ControllerModel = SISOController;
protected
    ControllerModel controller;
    // then same as MotorDrive.
end MotorDrive2;

model PIDControlledDrive = MotorDrive2
    (redeclare block ControllerModel = PID);
Symbolic Processing for Efficient Simulation

Model instantiation gives \textbf{implicit} DAE
(Differential Algebraic Equation system)

\[
F(t, \frac{dx}{dt}, x, w, p, u, y) = 0
\]

What are known variables depend on problem formulation
- known forces and torque, unknown positions
- known positions, velocities and accelerations,
  unknown required force and torques

Direct use of DAE solver often not feasible:
- dimension of \( w \) (auxiliary variables) high
- large Jacobian gives inefficient simulation
Example - Simple Circuit - DAE

**DAE:**

- **R1:** \( R_1.v = AC.Vp - R_1.Vn \)
  \[ R_1.R*R_1.i = R_1.v \]

- **AC:** \( AC.v = AC.Vp - G.Vp \)
  \[ AC.Vp - G.Vp = AC.VA*\sin(2\pi AC.freq*time) \]

- **C:** \( C.v = R_1.Vn - G.Vp \)
  \[ C.C*\text{der}(C.v) = R_1.i \]

- **G:** \( G.Vp = 0 \)

- **R2:** \( R_2.v = AC.Vp - L.Vp \)
  \[ R_2.R*L.i = R_2.v \]

- **L:** \( L.v = L.Vp - G.Vp \)
  \[ L.L*\text{der}(L.i) = L.Vp - G.Vp \]

**Circuit:** \( G.i = AC.i + R_1.i + L.i \)
\[ AC.i + R_1.i + L.i = 0 \]
Structural Processing

- Conversion to **explicit** ODE form
  \[
  \frac{dx}{dt} = f(t, x, p, u) \\
y = g(t, x, p, u)
  \]

- Graph theoretical methods used
  (bipartite graph)
  for assigning causalities and sorting equations
  (strongly connected components, Tarjan)

- Gives sequence of assignments statements
  (solver does not handle w)
  and simultaneous systems of equations (algebraic loops)
  - finding minimal loops
- Jacobian - Block Lower Triangular
- Tearing used to reduce sparse matrices
Symbolic Formula Manipulation

Formula manipulation
- abstract syntax tree for expressions
- algebraic transformation rules recursively applied to tree, such as:

\[(a + bx) - (c + dx) \rightarrow a - c + (b - d)x\]

Example of manipulations
- solving linear equations and certain non-linear equations
- finding matrix coefficients for linear systems of equations
- solving small linear systems of equations
- finding Jacobian for nonlinear systems of equations

Specialized computer algebra algorithms needed
- high capacity
- appropriate heuristics
Example - Simple Circuit - ODE

ODE:

G: \( G.Vp = 0 \)

AC: \( AC.Vp = AC.VA* \sin(2*\pi*AC.freq*\text{time}) + G.Vp \)

C: \( R1.Vn = G.Vp + C.v \)

R1: \( R1.v = AC.Vp - R1.Vn \)

\[ \text{R1.i} = \frac{R1.v}{R1.R} \]

Circuit: \( AC.i = -(R1.i + L.i) \)

AC: \( AC.v = AC.Vp - G.Vp \)

C: \( \text{der}(C.v) = R1.i/C.C \)

Circuit: \( G.i = AC.i + R1.i + L.i \)

R2: \( R2.v = R2.R \times L.i \)

L.Vp = AC.Vp - R2.v

L: \( L.v = Vp - G.Vp \)

\( \text{der}(L.i) = (L.Vp - G.Vp)/L.L \)
Simplifications of equations

• General library models
• Needs specialization in its environment
• Example: 3D mechanical model constrained to move in 2D
  \[ \text{AxisOfRotation} = \{0, 0, 1\} \]

• Manipulations:
  - substitute constants and fixed parameters
  - partial evaluation of expressions:
    \[ 0 \times \text{expr} = 0, \quad \text{expr}/\text{expr} = 1, \text{etc} \]

• Reduction in number of arithmetic operations:
  typically a factor of 10
Higher index DAE's

- Constraints on differentiated variables
- Dependent initial conditions
- Reduced degree-of-freedom

- Example: capacitors in parallel, rigidly connected masses
- Cannot solve for all derivatives

- Differentiate certain equations symbolically
  algorithm by Pantelides
- Automatic state variable selection
Mode handling

- Efficient solution of linear systems of equations
- Coefficients depending on switches
- Different code for each combination
- All different combinations: $2^n$
- Instead, determine used combinations of switches (modes) by off-line simulation
- Speed-up: 6 times
Object-Oriented Modeling

• OO Modeling is not OO Programming

• OO-Programming:
  – dynamic objects created and destroyed at runtime
  – run time type checking needed
  – message passing between objects
  – well suited for pure discrete event systems
Object-Oriented Modeling

- **Structuring** of complex systems
- Global analysis of all equations from all objects at compile time
- Can make good use of those constructs in OO-languages that are used for
  - Structuring
  - **Static** checking and analysis of the code
Structuring Object-Oriented Models

• Top level: engineering components
• More fine grained Classes:
  – Needed: flexibility in level of detail
  – physical phenomena:
    • chemical equilibrium
    • momentum balance
  – use composition through multiple inheritance
• Problem: all combinations should be a well-posed simulation problem
Structuring Object-Oriented Models

- Misleading expectation by newcomers:
- Compatible interfaces are not enough
- **Compatible model assumptions** needed!
  - Combining e.g. flow models that use steady and unsteady Bernoulli equations may be
    - a modeling error (impossible to conserve energy)
    - or justified for a certain applications
    - and may or may not result in a well posed problem
- Robust Model Libraries a big help for non-experts
Free Modelica Library

Modelica language design is accompanied with the design of a large, free, multi-domain component library (download from www.Modelica.org):

- **Control** systems (input/output block).
- **Electric** and **electronic** systems (SPICE elements).
- **1D** and **3D mechanical** systems.
- **Hydraulic** components.
- **Thermo-fluid** systems based on finite volume method.
- **Electric power** systems.
- **State machines** (simple, Petri-nets, Statecharts).
Modelica: Experiences at Ford

• Emerging technology
  – great enthusiasm for new possibilities
  – Modelica not fully implemented in Dymola
  – many further possibilities for symbolic analysis

• Tool issues vs. language issues
  – What is missing: the implementation in Dymola or
  – a Modelica language feature

• Main factor of success: Robust libraries
  – easy success: transmission models (well tested libraries)
  – moderate progress: hydraulics (unfinished pre-release library)
  – hard work: new models in areas where no libraries exist.

• A lifetime of procedural thinking:
  **how to teach declarative modeling?**
Experiences At Ford: Engine Modeling

- Starting out into a “new” domain is a major task
- Building robust and useful model libraries is difficult and time consuming!
- Example combustion chemistry:
  - easy to express in Modelica
  - different requirements for numerics and tool support
  - e.g. demonstrated need to be able to differentiate functions
Checklist:
Using Modelica for new Applications

• Is the expressiveness of the language up to the task?
  – Yes? Fine!
  – Almost? The Modelica group is open for participation!
• Does Modelica offer structuring elements that cope with the complexity issues of the task?
• Are the existing Tools sufficient?

Depending on answers to these questions
• Different possible scenarios for using the Modelica language and existing tools
Example gallery

• Hybrid Models
  – ideal diode
  – friction

• Object Oriented Model Structuring
  – class parameters
  – propagation of class parameter

• Partial differential equations
  – finite volumes: Thermo-fluid systems
  – finite differences: heat conduction
  – characteristics method: plug flow

• Field Couplings

• Regular connection structures
Object Oriented Model Structuring

Model HeatExchanger
extends HexShell;
replaceable model Medium = Water;
replaceable model PlateData = Hx4748;
Tube warmside(redeclare model Medium=Medium);
Tube coldside(redeclare model Medium=Medium);
Wall wall(redeclare record Data = PlateData);
equation
  connect(wall.qTa,warmside.qT);
  connect(wall.qTb,coldside.qT);
  connect(cinflow,coldside.inflow);
  connect(coutflow,coldside.outflow);
  ....
end HeatExchanger;
Field couplings

- Inner/outer components may be used to model simple fields, where some physical quantities, such as gravity vector, environment temperature or environment pressure, are accessible from all components in a specific model hierarchy.

```plaintext
class A
   outer Real T0;
   ...
end A;

class B
   inner Real T0;
   A a1, a2; // B.T0, B.a1.T0 and B.a2.T0 is the same variable
   ...
end B;
```
Field couplings - gravity I

partial function gravity
    input Real x[3];
    output Real F[3];
end gravity;

function uniformGravity
    extends gravity;
algorithm
    F := {0, -9.81, 0};
end uniformGravity;

function pointGravity
    extends gravity;
    parameter Real k=1;
algorithm
    F := -k*x/sqrt(x*x)/(x*x);
end pointGravity;

partial model environmentGravity
    outer function g=gravity;
end environmentGravity;

partial model Gravity
    inner function g=pointGravity;
end Gravity;

model particle
    extends environmentGravity;
    parameter Real m=1;
    Real pos[3](start={1,1,0}), vel[3](start={0,1,0}), a[3];
algorithm
    a := g(pos)/m;
equation
    der(vel) = a;
    der(pos) = vel;
end particle;
model composite1
  extends Gravity;
  particle p1, p2;
end composite1;

model composite2
  inner function g=uniformGravity;
  particle p1, p2;
end composite2;

model system
  composite1 d1;
  composite2 d2;
end system;
model Component
    HeatConnector heat;
end Component;

model TwoComponents
    Component Comp[2];
    HeatConnector heat;
equation
    connect(Comp[1].heat, heat);
    connect(Comp[2].heat, heat);
end TwoComponents;

model CircuitBoard
    HeatConnector globalHeat;
    Component comp1;
    TwoComponent comp2;
equation
    connect(comp1.heat,globalHeat);
    connect(comp2.heat,globalHeat);
end CircuitBoard;
model Component
    HeatConnector heat;
    outer HeatConnector globalHeat;
equation
    connect(heat, globalHeat);
end Component;

model TwoComponents
    Component Comp[2];
end TwoComponents;

model CircuitBoard
    inner HeatConnector globalHeat;
    Component comp1;
    TwoComponent comp2;
end CircuitBoard;
Field coupling of many Particles

function ElectrostaticForce
    input Position3 s1, s2;
    input SIunits.Charge Q1, Q2;
    output Force3 F;
algorithm
    F := (s1-s2)/distance(s1,s2)*Q1*Q2/
        (4*PI*epsilon_0*distance(s1,
        s2)^2);
end ElectroStaticForce;

model EParticles
    SIunits.Charge Q = 1.0;
    Estatic e(Q=Q);
    parameter SIunits.Mass m=1.0;
    Velocity3 vel;
    Acceleration3 a;
    equation
        der(vel) = a;
        der(e.s) = vel;
        a *m = e.F;
end EParticles;

model CloudOfParticles
    parameter Integer n=3;
    EParticles p[n];
    Estatic e[n];
    equation
        for i in 1:n loop
            connect(p[i].e,e[i]);
        end for;
    algorithm
        for i in 1 : n loop
            F[i, :] := zeros(3);
            for j in 1 : n loop
                if i<>j then
                    F[i, :] := F[i, :] +
                        ESForce(s[i,:],s[j,:],Q[i],Q[j]);
                end if;
            end for;
        end for;
end CloudOfParticles;
Simulation with Dymola

• Animation
• Initialization
• Open Block Interface
• Experiment Scripts
Animation of simulation results
Initializer

- Initial conditions on any variable
- Also parameter calculations
- Specify as Fixed or Free
- Or Desired with Weight
Open Block Interface

- Modelica model
- Add I/O graphically

- Use in Simulink
- S-function MEX file
Experiment Scripts

- Control simulations
- Parameter sweeps
  - for loops
- Plotting
- Use of Modelica syntax
- User defined functions
- Interface to Lapack
- Interface to analysis and design package Slicot
openModel("controllerTest.mo");
omega = 1;    // Declare omega.
k = 1;        // Declare gain.
for D in {0.1, 0.2, 0.4, 0.7} loop
    // Parameter sweep over damping coefficient.
    tr.a = {1, 2*D*omega, omega**2};
    tr.b = {k*omega**2};
    simulateModel("controllerTest", 0, 10);
    plot({"u", "y"});
end for;
Conclusions

• The goal is that Modelica becomes a (de-facto) **standard language** for representing models
• Joint effort with many experts
• Networked development
• Libraries and tools are available
• One step towards full scale virtual prototyping
Modelica Future Development

- Design Meetings
  - May 24 - 26, Aveiro, Portugal
  - September 21 - 23, Lund, Sweden
  - November, Munich, Germany

- Language refinements

- Library development

- Extensions
  - Partial Differential Equations
  - API’s to solvers, visualization, experimentation

- www.Modelica.org

- Modelica Design Group is open for participation
Future Directions

- Parameter identification
- Design optimization
- Verification suits
- Different levels of detail
- Model reduction
- Visualization for grasping complexity
- Web-links to models and model libraries
- **Part vendors distribute part models**
Acknowledgement

• Manuel Alfonseca  Universidad Autonoma de Madrid, Spain
• Bernhard Bachmann  Universität Paderborn, Paderborn, Germany
• Fabrice Boudaud  Gaz de France, Paris, France
• Alexandre Jeandel  Gaz de France, Paris, France
• Nathalie Loubere  Gaz de France, Paris, France
• Jan Broenink  University of Twente, The Netherlands
• Dag Brueck  Dynasim AB, Lund, Sweden
• Hans Olsson  Dynasim AB, Lund, Sweden
• Hilding Elmqvist  Dynasim AB, Lund, Sweden
• Thilo Ernst  GMD-FIRST, Berlin, Germany
• Jorge Ferreira  Universidade de Aveiro, Portugal
• Ruediger Franke  ABB Corporate Research Center, Heidelberg, Germany
• Peter Fritzson  Linköping University, Sweden
• David Kågedal  Linköping University, Sweden
• Sven Erik Mattsson  Dynasim AB, Lund, Sweden
• Henrik Nilsson  Linköping University, Sweden
• Pavel Grozman  BrisData AB, Stockholm, Sweden
• Per Sahlin  BrisData AB, Stockholm, Sweden
• Kaj Juslin  VTT, Finland
• Matthias Klose  Technical University of Berlin, Germany
• Pieter Mosterman  DLR Oberpfaffenhofen, Germany
• Andre Schneider  Fraunhofer Institute for Integrated Circuits, Dresden, Germany
• Peter Schwarz  Fraunhofer Institute for Integrated Circuits, Dresden, Germany
• Michael Tiller  Ford Motor Company, Dearborn, U.S.A.
• Hans Vangheluwe  University of Gent, Belgium
www.Modelica.org

Springbock