Systems Modeling and Programming in a Unified Environment based on Julia

Hilding Elmqvist, Toivo Henningsson, Martin Otter

Hilding Elmqvist

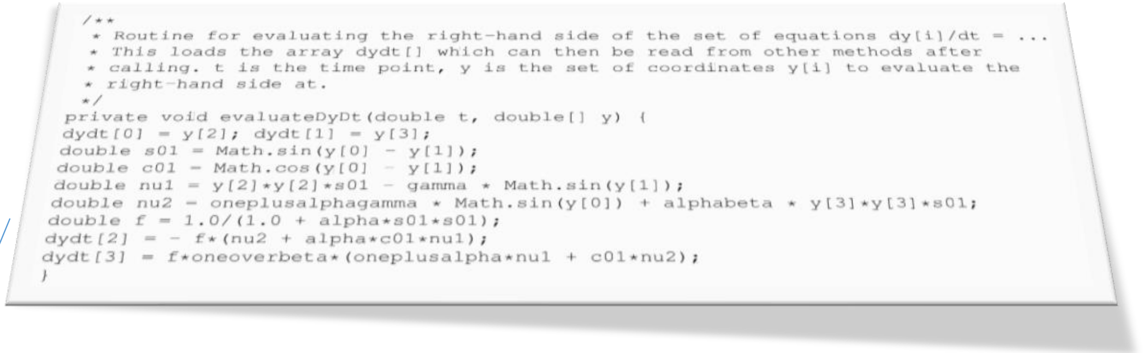
- Founded Dynasim (1992)
- Architect of Dymola
- Modelica Initiative – one of key architects (1996)
- Dynasim acquired by Dassault Systémes (2006)
- FMI – one of key architects
- Founded Mogram (2016)
- Modia – open source initiative

Outline

- What´s in a System Model
- Modelica
- Rationale for *Modia* project
- Julia
- Introduction to *Modia* Language
- *Modia* Prototype
- Summary

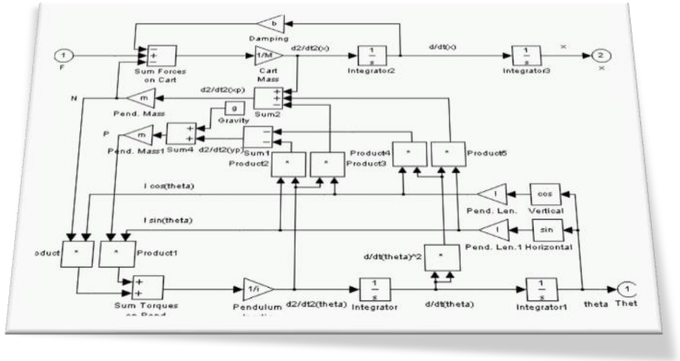
What´s in a System Model?

- Lumped Element Model
  - Discrete set of only time-varying variables, i.e. No partial derivatives
  - Ordinary Differential Equations
  - Algebraic Equations

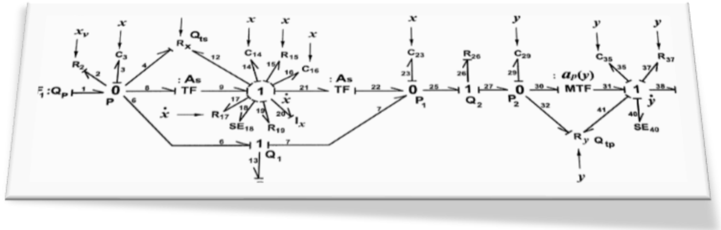


```
/**
 * Routine for evaluating the right-hand side of the set of equations dy[i]/dt = ...
 * This loads the array dydt[] which can then be read from other methods after
 * calling. t is the time point, y is the set of coordinates y[i] to evaluate the
 * right-hand side at.
 */
private void evaluateDyDt(double t, double[] y) {
dydt[0] = y[2]; dydt[1] = y[3];
double s01 = Math.sin(y[0] - y[1]);
double c01 = Math.cos(y[0] - y[1]);
double nu1 = y[2]*y[2]*s01 - gamma * Math.sin(y[1]);
double nu2 = oneplusalphagamma * Math.sin(y[0]) + alphabeta * y[3]*y[3]*s01;
double f = 1.0/(1.0 + alpha*s01*s01);
dydt[2] = - f*(nu2 + alpha*c01*nu1);
dydt[3] = f*oneoverbeta*(oneplusalpha*nu1 + c01*nu2);
}
```

- Assignment statements (Algorithm)? **No**
- Data flow diagrams (Block diagrams)? **No**



- Bond graphs? **No**



Problem: The system topology is not shown

- Manual derivation of algorithm
- Manual derivation of diagram

Engineering Practice I – Use Equations

Differential-Algebraic Equations (DAE)

$$
\begin{aligned}
0 &= f\left(\frac{\mathrm{d}\,x}{\mathrm{d}\,t}, x, w, p, u, y\right) \\
0 &= g\,(x, w, p, u, y)
\end{aligned}
$$

Example:

$$
m_1 \frac{d^2 x_1}{dt^2} + (\lambda_1 + \lambda_2)\frac{dx_1}{dt} - \lambda_2 \frac{dx_2}{dt} + (k_1 + k_2)\,x_1 - k_2 x_2 = 0
$$

$$
m_2 \frac{d^2 x_2}{dt^2} - \lambda_2 \frac{dx_1}{dt} + (\lambda_2 + \lambda_3)\frac{dx_2}{dt} - k_2 x_1 + (k_2 + k_3) x_2 = 0
$$

Howbeit, for easie alteratiō of equations. I will propounde a fewe exāples, bicause the extraction of their rootes, maie the more aptly bee wroughte. And to auoide the tediouse repetition of these woordes : is equalle to : I will sette as I doe often in woorke use, a paire of paralleles, or Gemowe lines of one lengthe, thus: ═══════, bicause noe .2. thynges, can be moare equalle. And now marke these nombers.

14.ze. —+— .15.9 ══════ 71.9.

Equality sign introduced in 1557 by Robert Recorde

[12]

AXIOMATA SIVE LEGES MOTUS

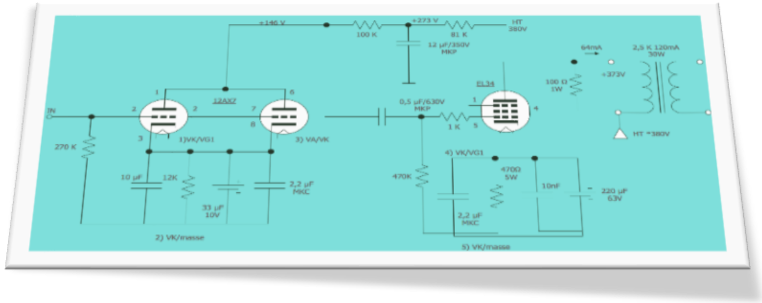
Lex. II.

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

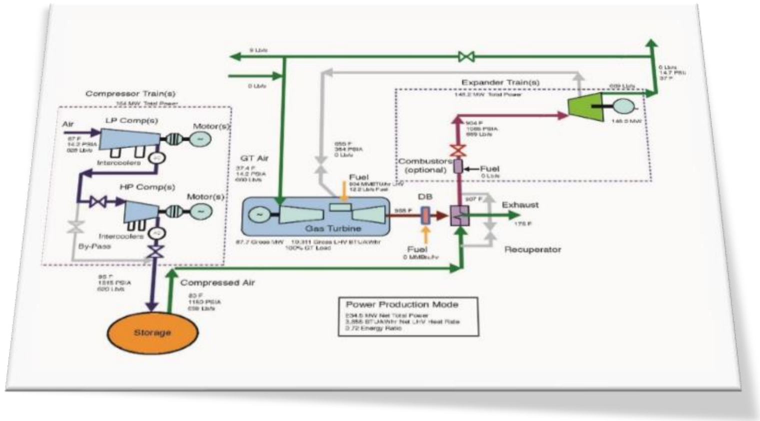
Newton´s 2nd law (July 5, 1687)

Engineering Practice II – Use Schematics

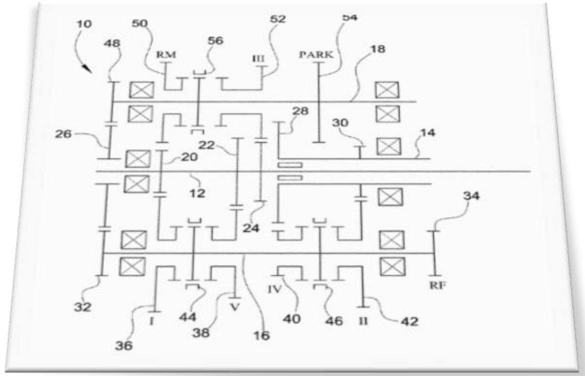
Circuit Diagram



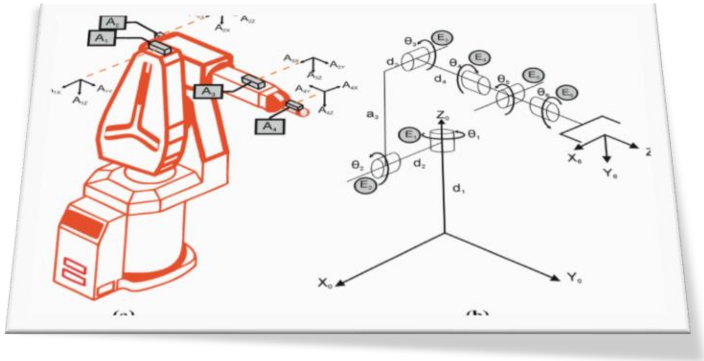
Process Flow Diagram



Gear box



Multibody System



History - Kirchhoff

- Kirchhoff published his voltage and current laws in 1845
- In 1847, Kirchhoff discussed the solution of these equations:

> Let μ be the least number of wires that must be removed from an arbitrary system so that all closed figures are destroyed. Then μ is also the independent equations that can be derived by the use of Theorem 1 [Kirchhoff's voltage law].

Kirchhoff discusses closed loops in circuit diagrams

> More than m - 1 [m is number of crossing points] independent equations cannot be derived by Theorem 2 [Kirchhoff's current law]. For if we apply Theorem 2 to all m crossing points, each I occurs two times in the equations thereby formed, one time with coefficient +1, the other time with the coefficient -1. Therefore, the sum of all equations yields the identical equation $0 = 0$. The equations obtained by application of that theorem to m - 1 arbitrary crossing points are, on the other hand, independent.

Kirchhoff discusses singular systems of equations

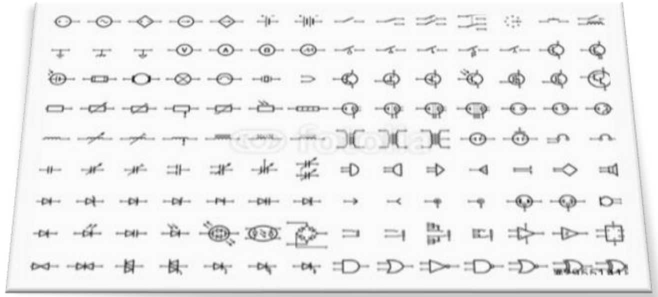
History - Analogies

- Maxwell (1873) introduced Force-Voltage Analogy
  - Effort and flow variables
  - Mass ≈ inductance
  - **Series** connection of electrical component **correspond** to **parallel** connection of mechanical components and vice versa
  - Paynter (1960): Bond graphs
- Firestone (1933) introduced Force-Current Analogy
  - Across (relative quantities) and Through variables
  - Mass ≈ Capacitor (Mass has reference to ground)
  - Kirchhoff's current law – sum of through variables are zero
- Trent (1955): Isomorphism between Oriented Linear Graphs and Lumped Physical Systems

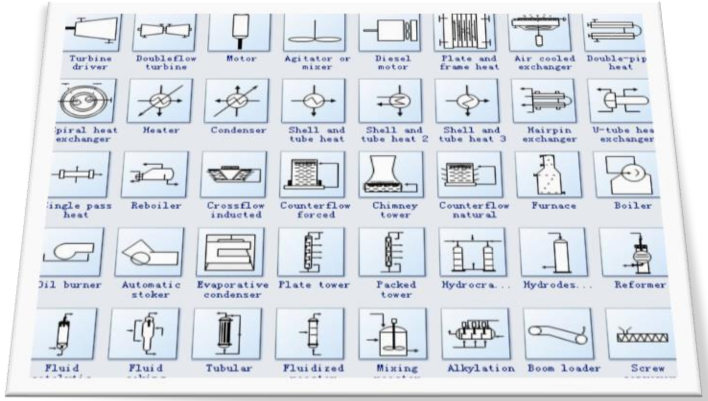
> - Variables of terminals associated with connections

Engineering Practice III – Use Catalogs of Symbols

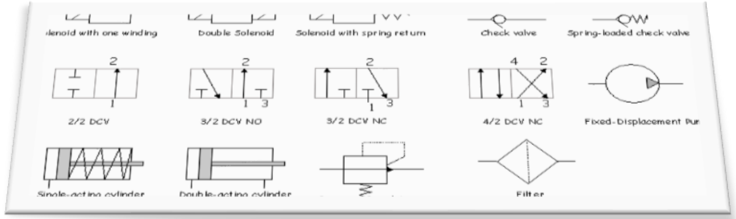
Electrical



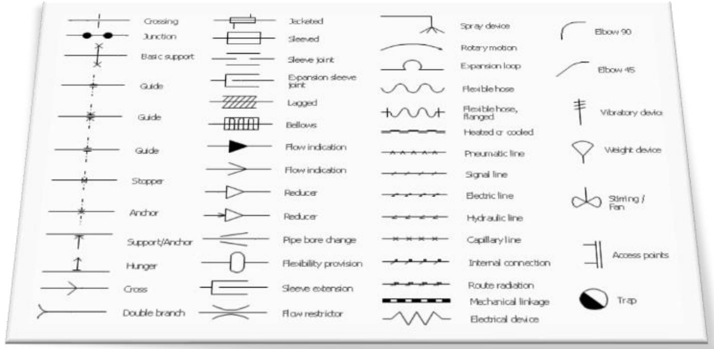
Process



Hydraulics



Fluid

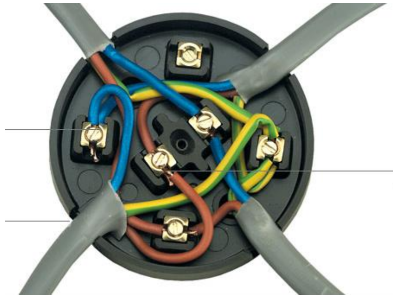


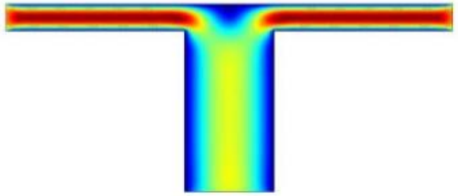
Ideal Connection Semantics

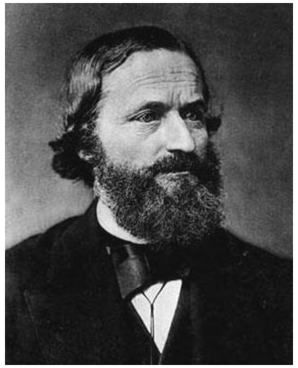
- Electrical: Kirchhoff’s current law, 1845
  - **Sum of currents** at junction is zero

- Mechanics: Newton´s (1687) and Euler’s (about 1737) second laws:
  - The **vector sum of the forces** on an object is equal to the mass of that object multiplied by the acceleration vector of the object.
  - The rate of change of angular momentum about a point that is fixed in an inertial reference frame, is equal to the **sum of torques** acting on that body about that point.
  - Neglect mass and moment of inertia at junction → **Sum of forces** are zero **and sum of torques** are zero

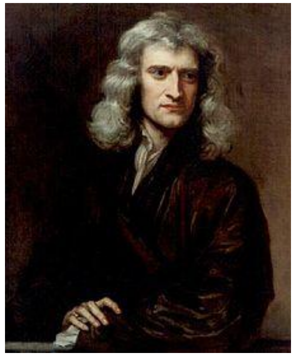
- Fluid systems:
  - Consider a small volume at junction →
  - Mass balance: **Sum of mass flow rates** are zero
  - Energy balance: **Sum of energy flow rates** are zero







Kirchhoff



Newton

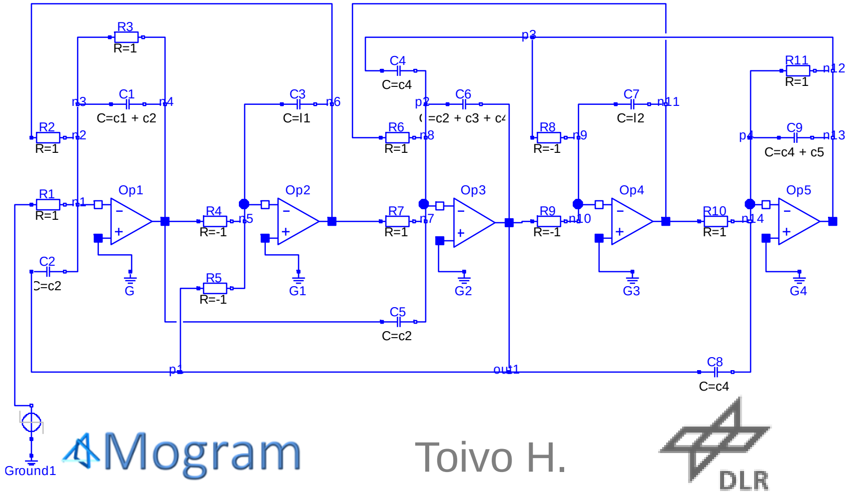
Euler

Model Based Systems Engineering Needs

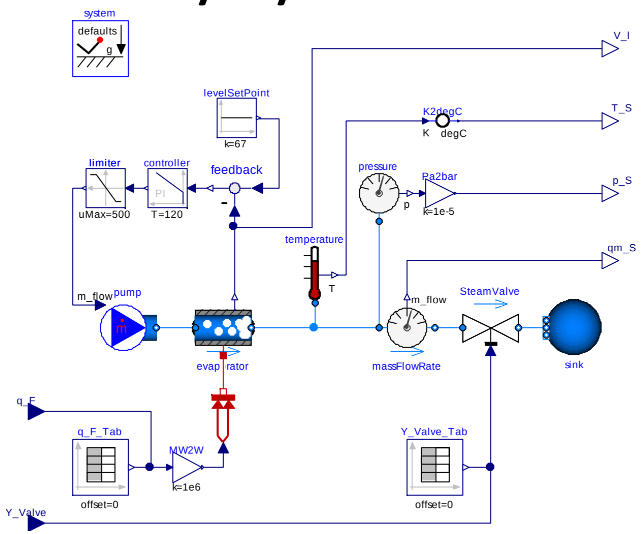
- Modeling continuous behavior using differential and algebraic equations
- System composition using graphs
- Graphical user experience
- Generic model parameters and templates
- Problem solving using advanced scripting
- Events and safe controllers using synchronous semantics

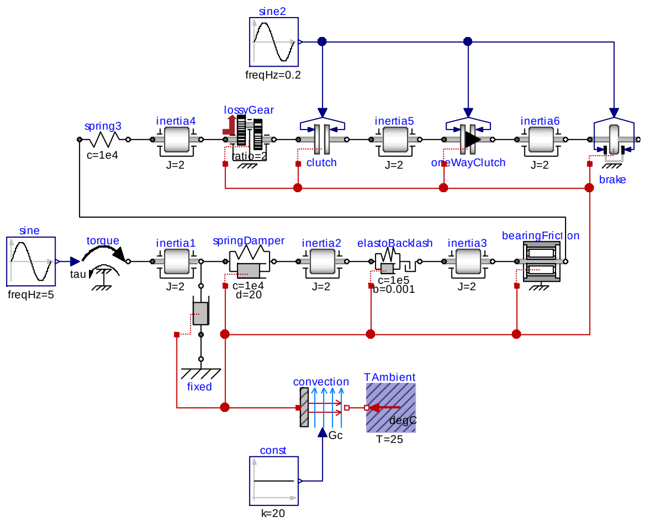
Unification by Modelica

- Modelica: A formal language to capture modeling knowhow
- Equation based language - for convenience
- Object oriented - for reuse
- System topology - by connections
- Terminal definitions - connectors
- Icons AND equations - not only symbols



Toivo H.





Why Modia?

- Evolution of Modelica language has slowed down
- Tool vendors are currently catching up
- Need an experimental language platform

- Modelica specification is becoming large and hard to comprehend
- Tool vendors want more details into the specification
- Better to make reference implementation

- Functions/Algorithms in Modelica are weak
  - no advanced data structures such as union types, no matching construct, no type inference, etc
- Better to utilize other language efforts for functions
- Julia has perfect scientific computing focus
- Modia - Julia macro set

Julia - Main Features

- Dynamic programming language for technical computing
- Strongly typed with Any-type and type inference
- JIT compilation to machine code (using LLVM)
- Matlab-like notation/convenience for arrays
- Advanced features:
  - Multiple dispatch (more powerful/flexible than object-oriented programming)
  - Matrix operators for all LAPACK types (+ LAPACK calls)
  - Sparse matrices and operators
  - Parallel processing
  - Meta programming
- Developed at MIT since 2012, current version 0.5.0, MIT license

Functions: Modelica vs Julia

Modelica:

```
function planarRotation "Return orientation object of a planar rotation"
  import Modelica.Math;
  extends Modelica.Icons.Function;
  input Real e[3](each final unit="1") "Normalized axis of rotation (must have length=1)";
  input Modelica.SIunits.Angle angle "Rotation angle to rotate frame 1 into 2 along axis e";
  output TransformationMatrices.Orientation T "Orientation object to rotate frame 1 into 2";
algorithm
  T := [e]*transpose([e]) + (identity(3) - [e]*transpose([e]))*Math.cos(angle) - skew(e)*Math.sin(angle);
  annotation(Inline=true);
end planarRotation;
```

Julia:

```
planarRotation(e, angle) = e*e' + (eye(3) - e*e')*cos(angle) - skew(e)*sin(angle)
```

Julia AST for Meta-programming

- Quoted expression :( )
  - Any expression in LHS
- Operators are functions
- $ for "interpolation"

```
julia> equ = :(0 = x + 2y)
:(0 = x + 2y)

julia> dump(equ)
Expr
  head: Symbol =
  args: Array(Any,(2,))
    1: Int64 0
    2: Expr
      head: Symbol call
      args: Array(Any,(3,))
        1: Symbol +
        2: Symbol x
        3: Expr
          head: Symbol call
          args: Array(Any,(3,))
          typ: Any
      typ: Any
  typ: Any
```

```
julia> solved = Expr(:(=), equ.args[2].args[2], Expr(:call, :-, equ.args[2].args[3]))
:(x = -(2y))


julia> y = 10
10
julia> eval(solved)
-20
julia> @show x
x = -20

Julia> # Alternatively (interpolation by $):
julia> solved = :($(equ.args[2].args[2]) = - $(equ.args[2].args[3]))
```

Modia – “Hello Physical World” model

```
@model FirstOrder begin
  x = Float(start=1)
  T = Parameter(0.5, "Time constant")
  u = 2.0 # Same as Parameter(2.0)
@equations begin
  T*der(x) + x = u
  end
end
```

Modelica

```
model M
  Real x(start=1);
  parameter Real T=0.5 "Time constant";
  parameter Real u = 2.0;
equation
  T*der(x) + x = u;
end M;
```

Variable Constructor

Current design (should be parametric to constrain the types of value, min, max, start, nominal to be of typ):

```
type Variable
    variability::Variability
    typ::DataType
    value
    unit::SIUnits.SIUnit
    displayUnit
    min
    max
    start
    nominal
    description::AbstractString
    flow::Bool
    state::Bool
end
```

```
Parameter(value,unit=SIPrefix,description="") =
    Variable(parameter, typeof(value), value, unit,
                unit, nothing, nothing, value, true, value,
                description, false, false)
```

```
Float(value=nothing, description=""; unit=SIPrefix,
        displayUnit=SIPrefix, min=nothing, max=nothing,
         start=nothing, fixed::Bool=false, nominal=nothing,
         variability=continuous, flow::Bool=false, state::Bool=nothing) =
    Variable(variability, Float64, value, unit, displayUnit, min,
         max, start, fixed, nominal, description, flow, state)
```

Electrical components

Modelica

```
@model Pin begin
 v=Float()
 i=Float(flow=true)
end

@model OnePort begin
 v=Float()
 i=Float()
 p=Pin()
 n=Pin()
@equations begin
 v = p.v - n.v
 0 = p.i + n.i
 i = p.i
 end
end

@model Resistor begin # Ideal linear electrical resistor
 @extends OnePort()
 @inherits i, v
 R=1 # Resistance
@equations begin
 R*i = v
 end
end
```

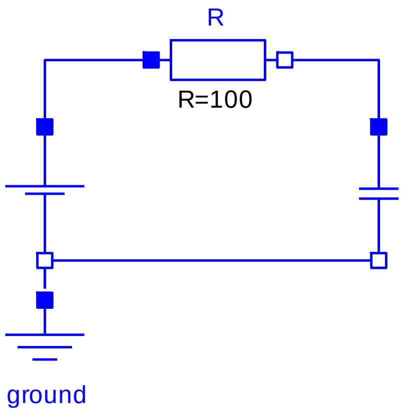
```
connector Pin
  Modelica.SIunits.Voltage v;
  flow Modelica.SIunits.Current I;
end Pin;

partial model OnePort
  SI.Voltage v;
  SI.Current i;
  PositivePin p;
  NegativePin n;
equation
  v = p.v - n.v;
  0 = p.i + n.i;
  i = p.i;
end OnePort;

model Resistor
  parameter Modelica.SIunits.Resistance R;
  extends Modelica.Electrical.Analog.Interfaces.OnePort;
equation
  v = R*i;
end Resistor;
```

Electrical Circuit

Modelica

```
@model LPfilter begin
  resistor=Resistor(R=1)
  capacitor=Capacitor(C=1)
  constantVoltage=ConstantVoltage(V=1)
  ground=Ground()
@equations begin
  connect(resistor.n, capacitor.p)
  connect(resistor.p, constantVoltage.p)
  connect(constantVoltage.n, capacitor.n)
  connect(constantVoltage.n, ground.p)
  end
end
```

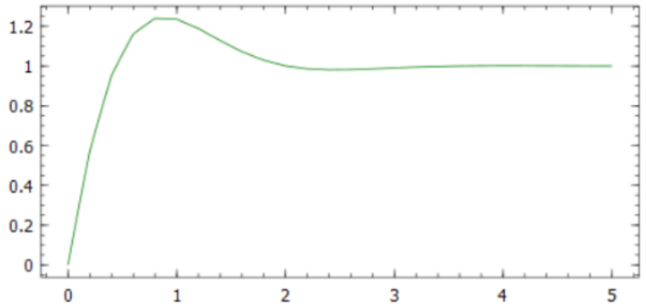
```
model LPfilter
  Resistor resistor(R=1)
  Capacitor capacitor(C=1)
  ConstantVoltage constantVoltage(V=1)
  Ground ground
equation
  connect(resistor.n, capacitor.p)
  connect(resistor.p, constantVoltage.p)
  connect(constantVoltage.n, capacitor.n)
  connect(constantVoltage.n, ground.p)
end
```

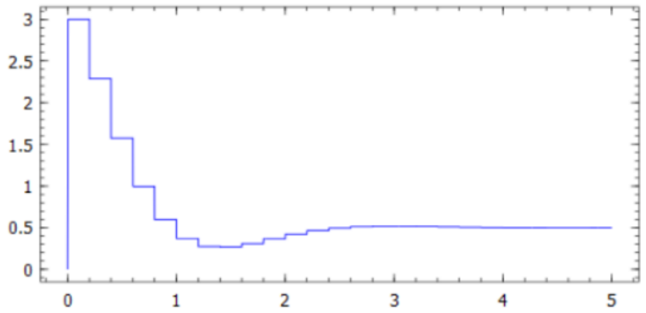


- Clock partitioning of equations
- Clock inference
- Clocked equations active at ticks

Synchronous Controllers

```
@model DiscretePIController begin
  K=1 # Gain
  Ti=1E10 # Integral time
  dt=0.1 # sampling interval
  ref=1 # set point
  u=Float(); ud=Float()
  y=Float(); yd=Float()
  e=Float(); i=Float(start=0)
@equations begin
  # sensor:
  ud = sample(u, Clock(dt))
  # PI controller:
  e = ref-ud
  i = previous(i, Clock(dt)) + e
  yd = K*(e + i/Ti)
  # actuator:
  y = hold(yd)
  end
end
```

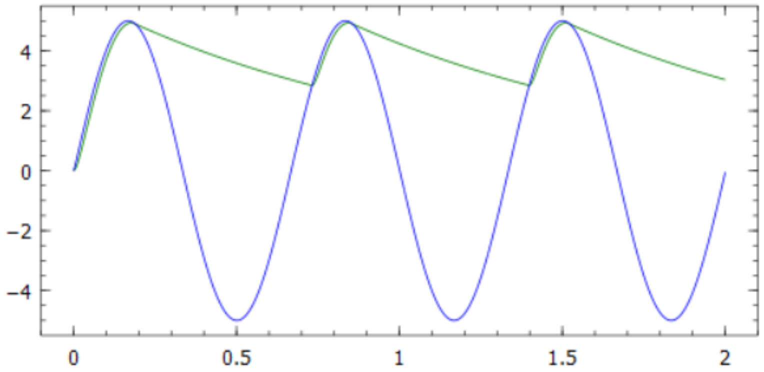


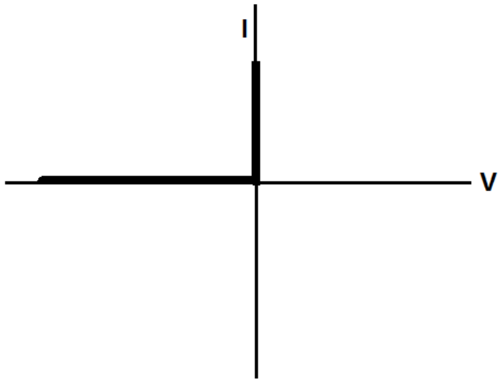


Discontinuities - State Events

```
@model IdealDiode begin
  @extends OnePort()
  @inherits v, i
  s = Float(start=0.0)
@equations begin
  v = if positive(s); 0 else s end
  i = if positive(s); s else 0 end
  end
end
```

- positive() and negative() introduces crossing functions



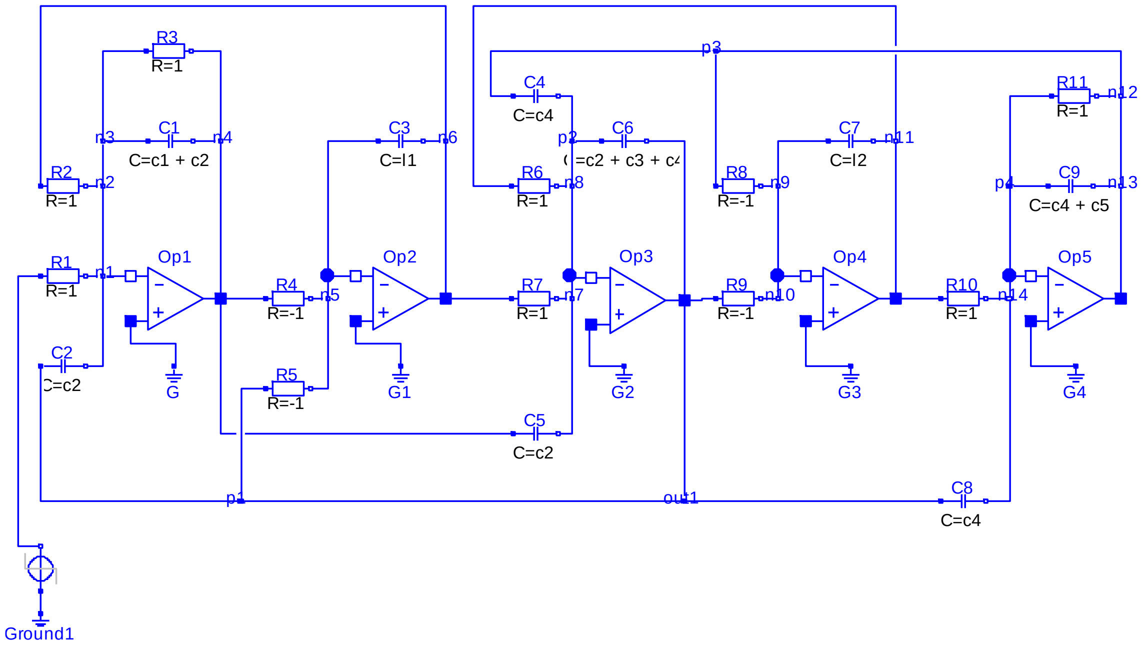


Cauer Low Pass Filter

- Parameter propagation
- The use of nodes to define connections
- Manual non-state selection

l1=1.304
l2=0.8586
c1=1.072
c2=1/(1.704992^2*l1)
c3=1.682
c4=1/(1.179945^2*l2)
c5=0.7262

C1=Capacitor(C=c1 + c2)
C2=Capacitor(C=c2)
C3=Capacitor(C=l1)
C4=Capacitor(C=c4)
C5=Capacitor(C=c2, v=Float(state=false))
R1=Resistor(R=1)
…
n1=Pin()
n2=Pin()
n3=Pin()
…



Rotational and Blocks Components

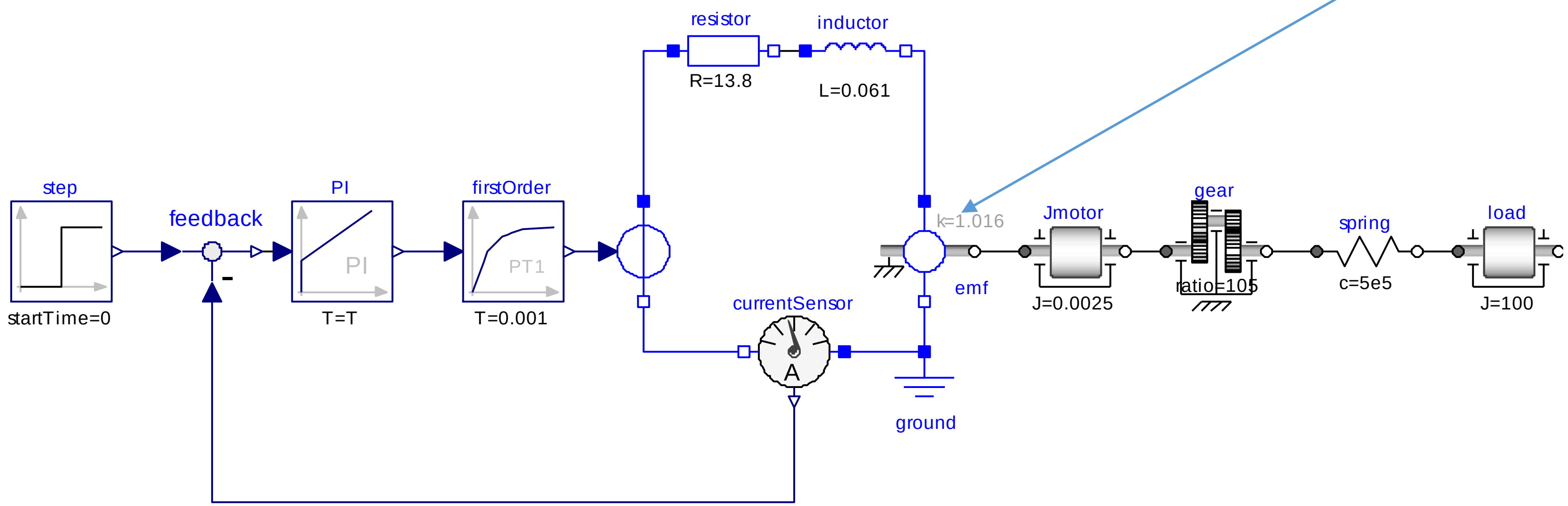
- Data flow blocks are special case

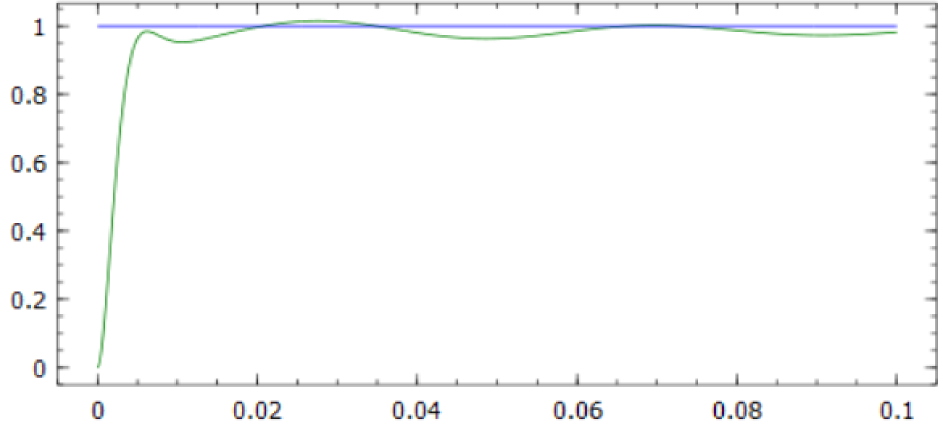
```
@model Flange begin
  phi=Float()
  tau=Float(flow=true)
end

@model Inertia begin
  J=Parameter(0, min=0)   # Moment of inertia

  flange_a=Flange()  # Left flange of shaft
  flange_b=Flange()  # Right flange of shaft

  phi=Float(start=0)
  w=Float(start=0)
  a=Float()
@equations begin
  phi = flange_a.phi
  phi = flange_b.phi
  w = der(phi)
  a = der(w)
  J*a = flange_a.tau + flange_b.tau
  end
end
```

```
@model SISO begin # Single Input Single Output
  u=Float()
  y=Float()
end

@model FirstOrder begin # First order transfer function
  k=1   # Gain
  T=1   # Time Constant
  @extends SISO()
  @inherits u, y
@equations begin
  der(y) = (k*u - y)/T
  end
end
```

Multi domain - Servo system

- Emf refers to der(phi)
- Causes index reduction
- Emf.phi not state variable





Differentiate: this.Jmotor.phi = this.Jmotor.flange_a.phi

introducing derivative: der(this.Jmotor.flange_a.phi)

giving: der(this.Jmotor.phi) = der(this.Jmotor.flange_a.phi)

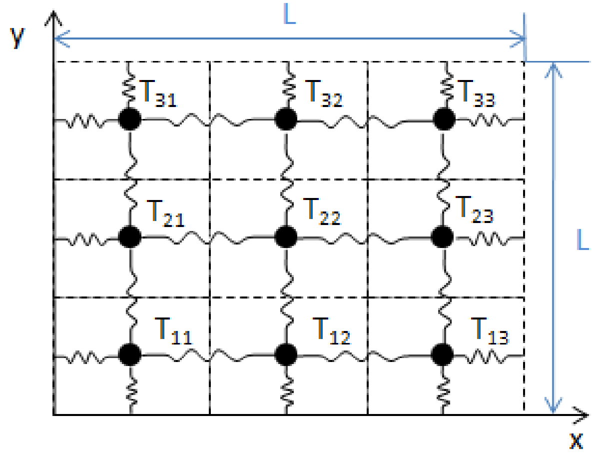
…

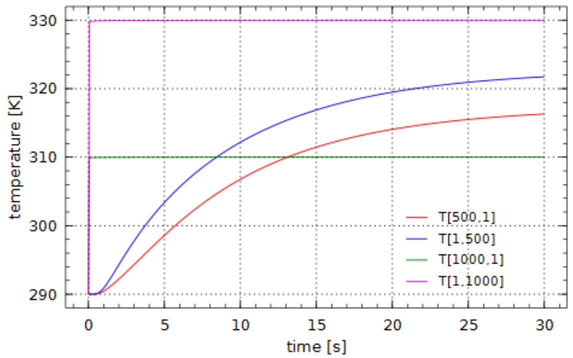
2-dimensional heat transfer

- One million states
- Very sparse Jacobian
- SundialsDAE has sparse handling

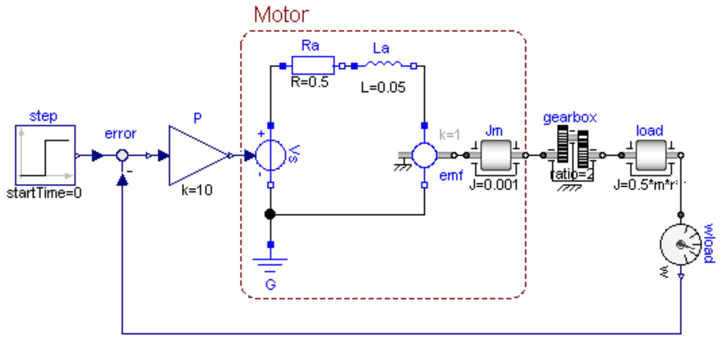
```
const N=1000; L=0.2; T0=290, ...
function heatTransfer2D(T)
  for i in 1:N, j in 1:N
    qx1=i>1 ? T[i-1,j]-T[i,j] : 0.0
    ...
    derT[i,j]=c*(qx1+qx2+qy1+qy2)
  end; return derT
end
@model HeatTransfer begin
  T = Float(start=fill(T0,N,N))
@equations begin
  der(T) = heatTransfer2D(T)
end
```

```
function  jacobian_incidence(::typeof(heatTransfer2D),args...)
  I::Vector{Int} = fill(1, 5*N*N)
  J::Vector{Int} = fill(1, 5*N*N)
  for i in 1:N, j in 1:N
   ...
  return sparse(I,J,1)
end
```

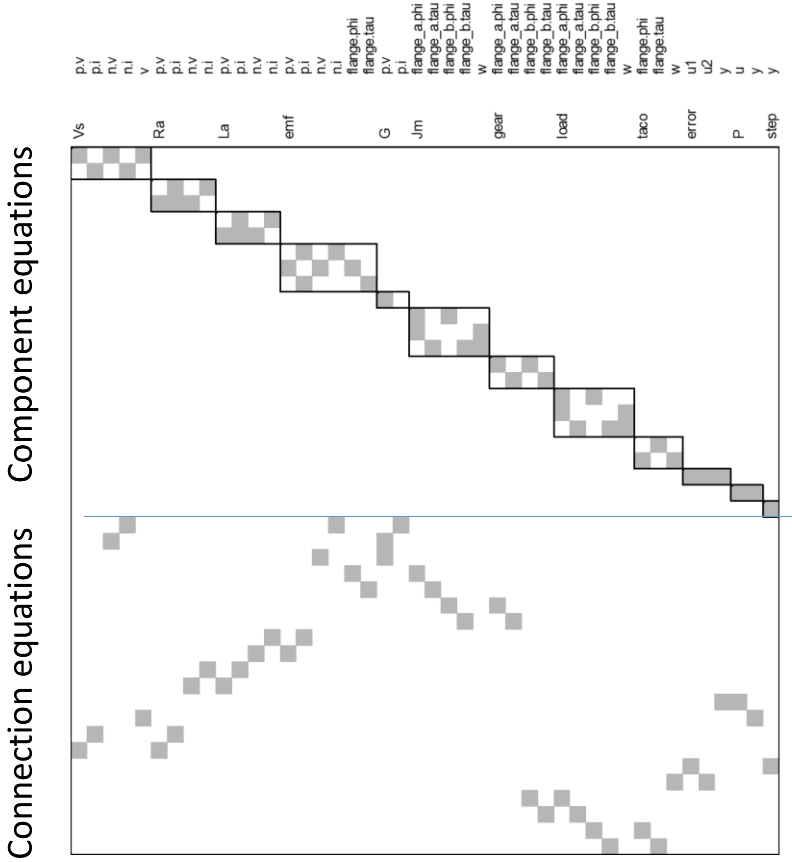




How To Simulate a Model



- Instantiate model, i.e. create sets of variables and equations
- Structurally analyze the equations
  - Which variable appear in which equation
  - Handle constraints (index reduction)
    - Differentiate certain equations
  - Sort the equations into execution order (BLT)
- Symbolically solve equations for unknowns and derivatives
- Generate code
- Numerically solve DAE
- Etc.



- Gives a sequence of subproblems
- Symbolically solve for variable in bold

BLT (Block Lower Triangular) form

error.u1 = step.offset+(if time < step.startTime then 0 else step.height);

error.y = error.u1-load.w;

Vs.p.v = P.k*error.y;

Ra.R*La.p.i = Vs.p.v-**Ra.n.v**;

Jm.w = gear.ratio*load.w;

emf.k*Jm.w = **La.n.v**;

La.L***der(La.p.i)** = Ra.n.v-La.n.v;

emf.flange.tau = -emf.k*La.p.i;

  // System of 4 simultaneous equations

  der(Jm.w) = gear.ratio*der(load.w);

  Jm.J*der(Jm.w) = **Jm.flange_b.tau**-emf.flange.tau;

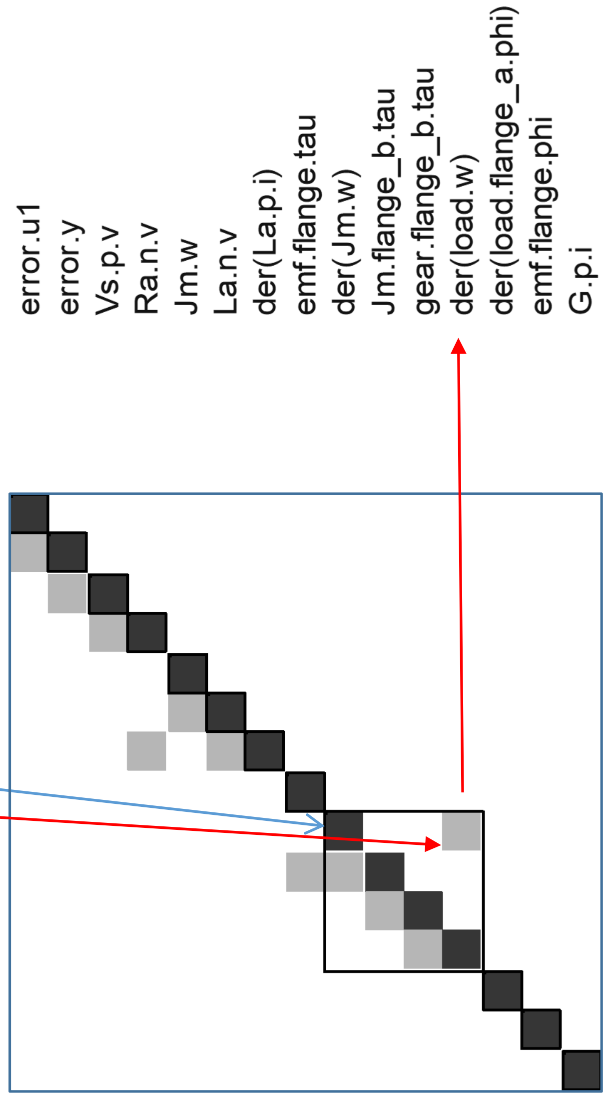
  0 = **gear.flange_b.tau**-gear.ratio*Jm.flange_b.tau;

  load.J***der(load.w)** = -gear.flange_b.tau;

der(load.flange_a.phi) = load.w;

emf.flange.phi = gear.ratio*load.flange_a.phi;

G.p.i+La.p.i = La.p.i;



Modia Prototype

- Work since January 2016
- Hilding Elmqvist / Toivo Henningsson / Martin Otter
- So far focus on:
  - Models, connectors, connections, extends
  - Flattening
  - BLT
  - Symbolic solution of equations (also matrix equations)
  - Symbolic handling of DAE index (Pantelides, equation differentiation)
  - Basic synchronous features
  - Basic event handling
  - Simulation using Sundials DAE solver, with sparse Jacobian
  - Test libraries: electrical, rotational, blocks, multibody
- Partial translator from Modelica to Modia (PEG parser in Julia)
- Will be open source

Summary - *Modia*

- Modelica-like, but much more powerful and simpler
- Algorithmic part: Julia functions (much more powerful than Modelica)
- Model part: Julia meta-programming (no Modia compiler)
- Equation part: Julia expressions (no Modia compiler)
- Structural and Symbolic algorithms: Julia data structures / functions
- Target equations: Sparse DAE (no ODE)
- Simulation engine: IDA + KLU sparse matrix (Sundials 2.6.2)
- Revisiting all typically used algorithms: operating on arrays (no scalarization), improved algorithms for index reduction, overdetermined DAEs, switches, friction, Dirac impulses, ...
- Just-in-time compilation (build Modia model and simulate at once)