MetaModelica

A Unified Equation-Based Semantical and Mathematical Modeling Language

Adrian Pop and Peter Fritzson

Programming Environment Laboratory
Department of Computer and Information Science
Linköping University
2006-09-14

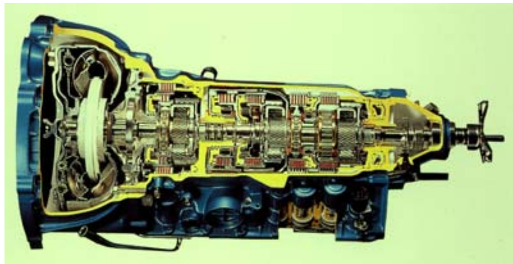
JMLC'2006, September 13-15,
Oxford, UK

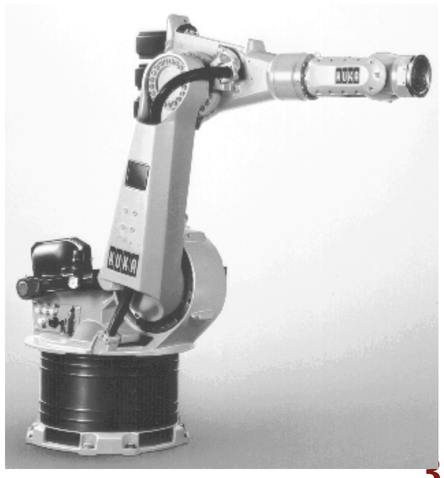
Outline

- Modelica
  - Introduction
  - Language properties
  - Example
- MetaModelica
  - Motivation
  - MetaModelica extensions to Modelica
  - Example
- Future Work
- Conclusions

Modelica – General Formalism to Model Complex Systems

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





Modelica - The Next Generation *Modeling* Language

- *Declarative language*
  - Equations and mathematical functions allow acausal modeling, high level specification, increased correctness
- *Multi-domain modeling*
  - Combine electrical, mechanical, thermodynamic, hydraulic, biological, control, event, real-time, etc...
- *Everything is a class*
  - Strongly typed object-oriented language with a general class concept, Java & Matlab like syntax
- *Visual component programming*
  - Hierarchical system architecture capabilities
- *Efficient, nonproprietary*
  - Efficiency comparable to C; advanced equation compilation, e.g. 300 000 equations

Modelica Language Properties

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

Equations were used in the third millenium B.C.

Equality sign was introduced by Robert Recorde in 1557

Newton (Principia, vol. 1, 1686) still wrote text:

“The change of motion is proportional to the motive force impressed; ...”

$$\frac{d}{dt}(m \cdot v) = \sum F_i$$

CSSL (1967) introduced special form of “equation”:

variable = expression

v = INTEG(F) / m

Programming languages usually do not allow equations

- What is *acausal* modeling/design?
- Why does it increase *reuse*?

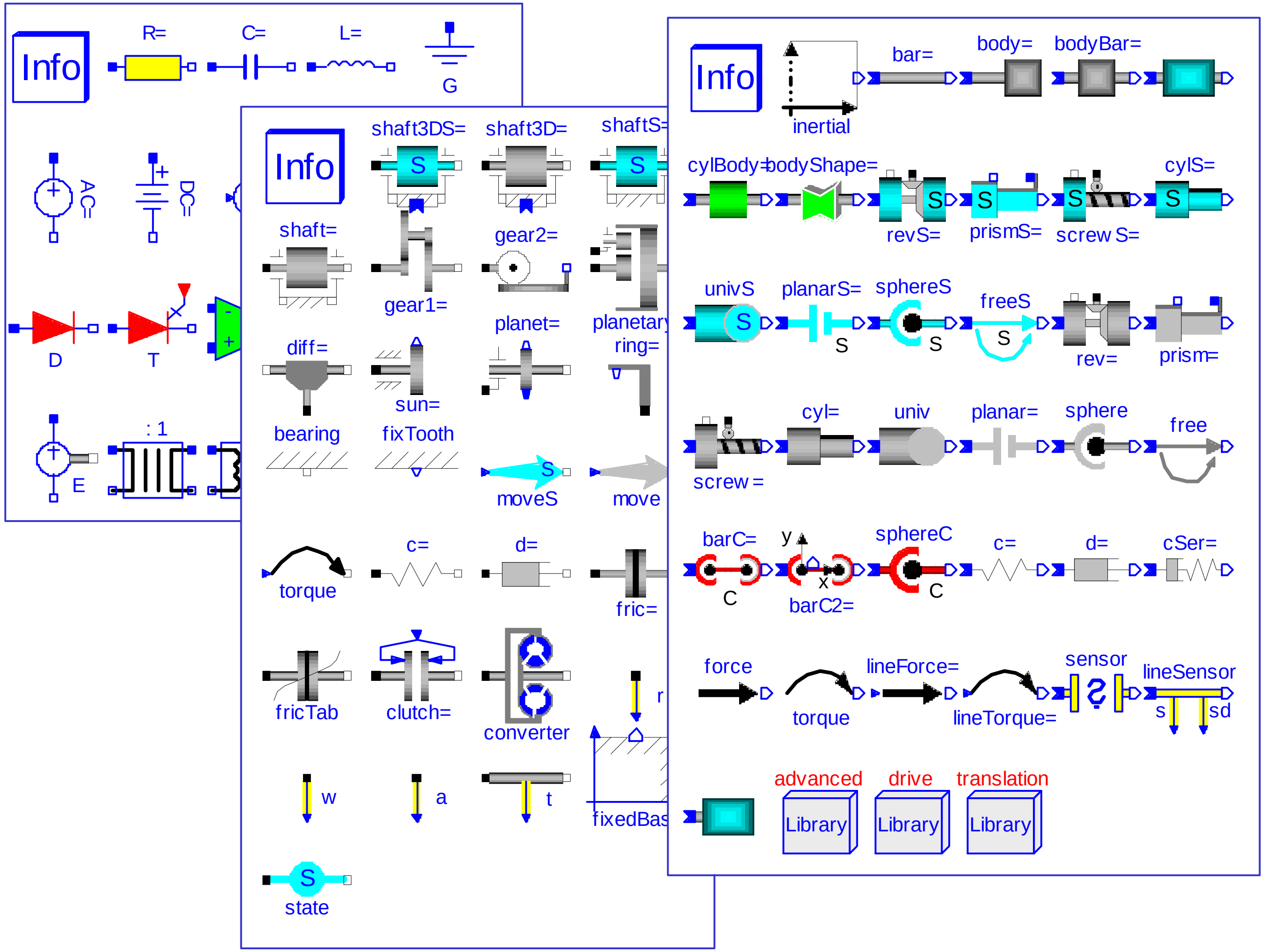
  The acausality makes Modelica classes *more reusable* than traditional classes containing assignment statements where the input-output causality is fixed.

- Example: a resistor *equation*:

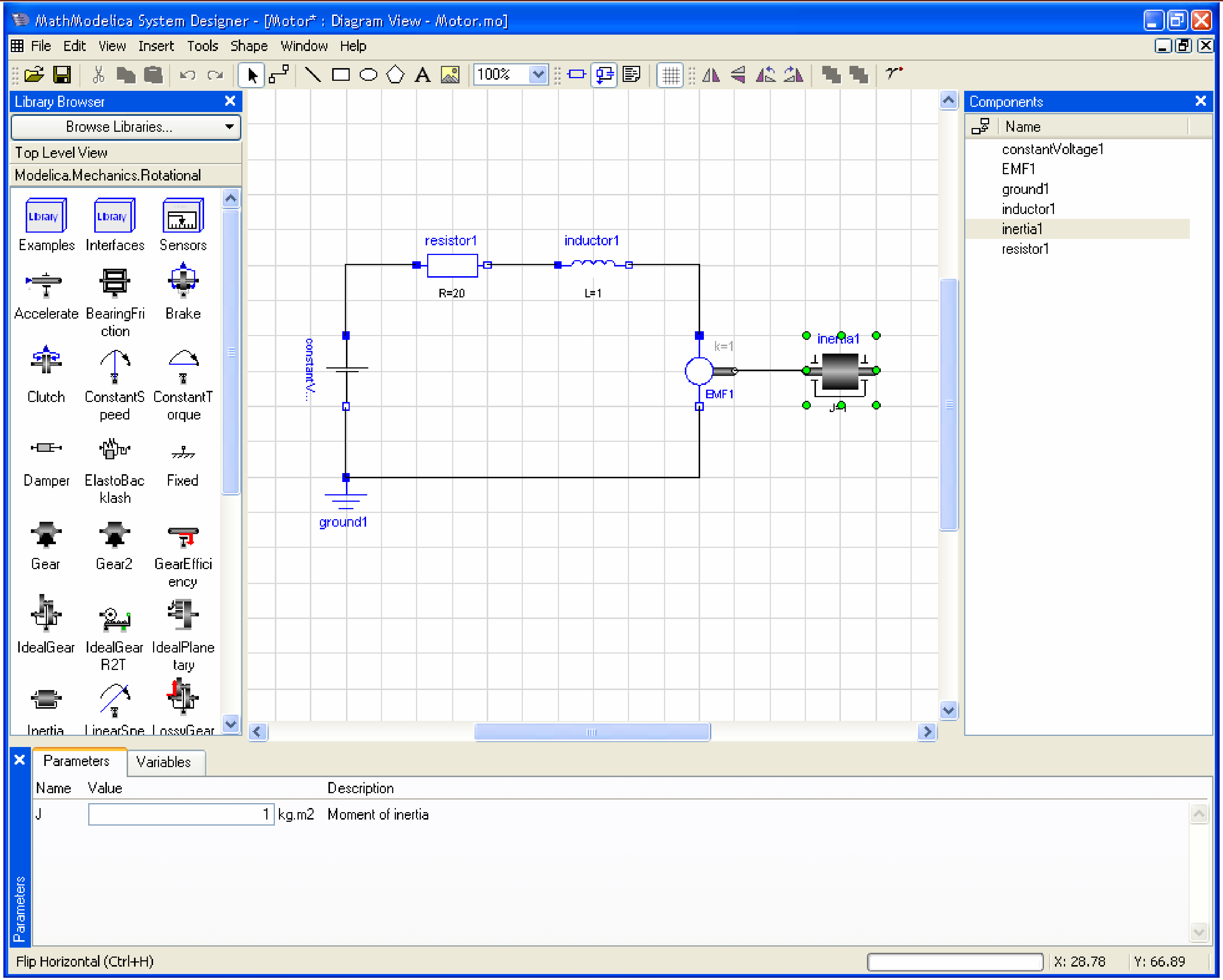
```
R*i = v;
```

can be used in three ways:

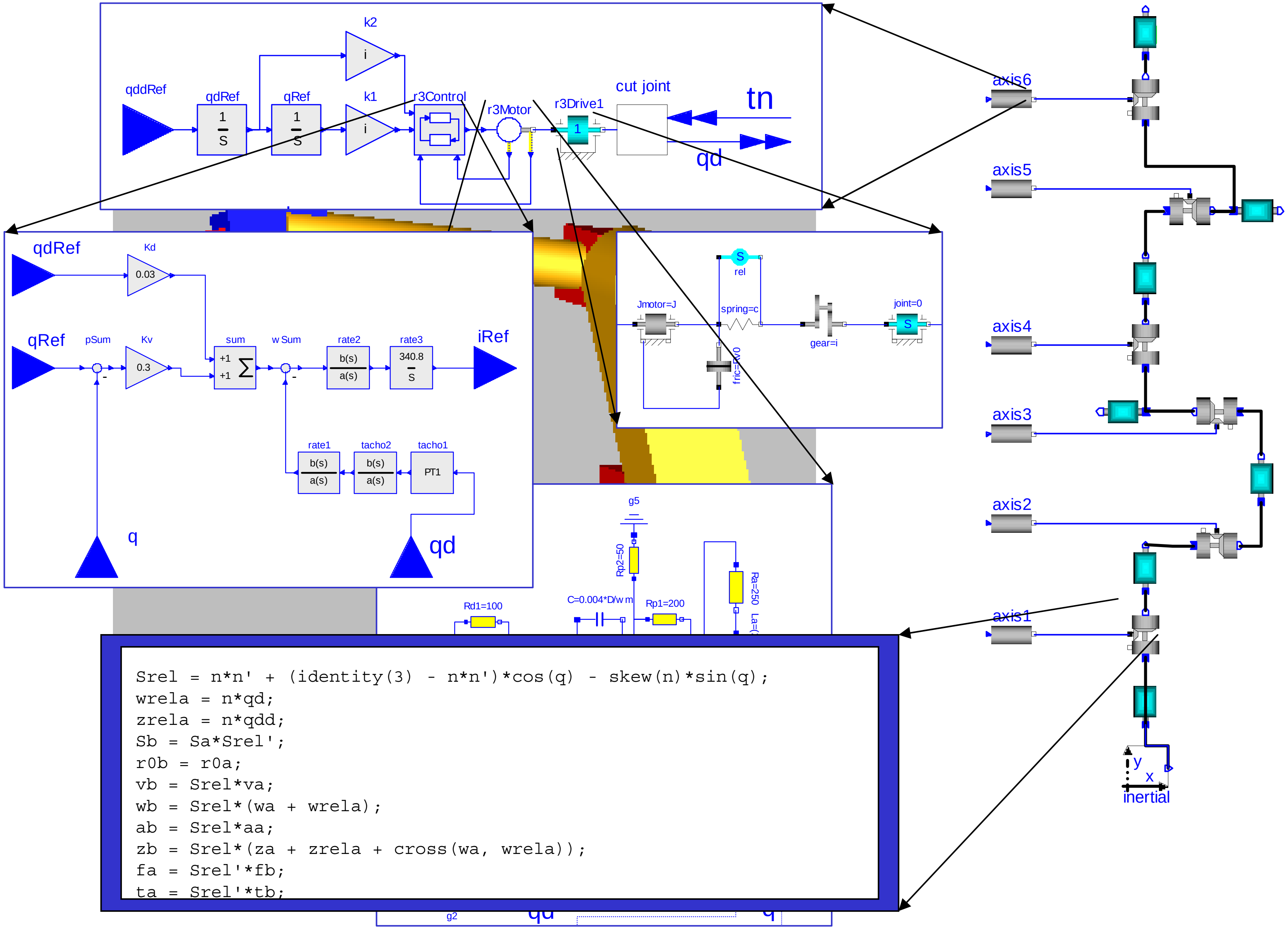
```
i := v/R;
v := R*i;
R := v/i;
```


Info
R=
C=
L=
G
AC=
DC=
D
T
E
: 1
Info
shaft3DS=
shaft3D=
shaftS=
shaft=
gear1=
gear2=
planet=
planetary
ring=
diff=
sun=
bearing
fixTooth
moveS
move
torque
c=
d=
fric=
fricTab
clutch=
converter
fixedBase
w
a
t
r
state
Info
inertial
bar=
body=
bodyBar=
cylBody=
bodyShape=
revS=
prismS=
screw S=
cylS=
univS
planarS=
sphereS
freeS
rev=
prism=
screw =
cyl=
univ
planar=
sphere
free
barC=
barC2=
sphereC
c=
d=
cSer=
force
torque
lineForce=
lineTorque=
sensor
lineSensor
advanced
drive
translation
Library
Library
Library

Graphical Modeling - Drag and Drop Composition



Hierarchical Composition Diagram for a Model of a Robot

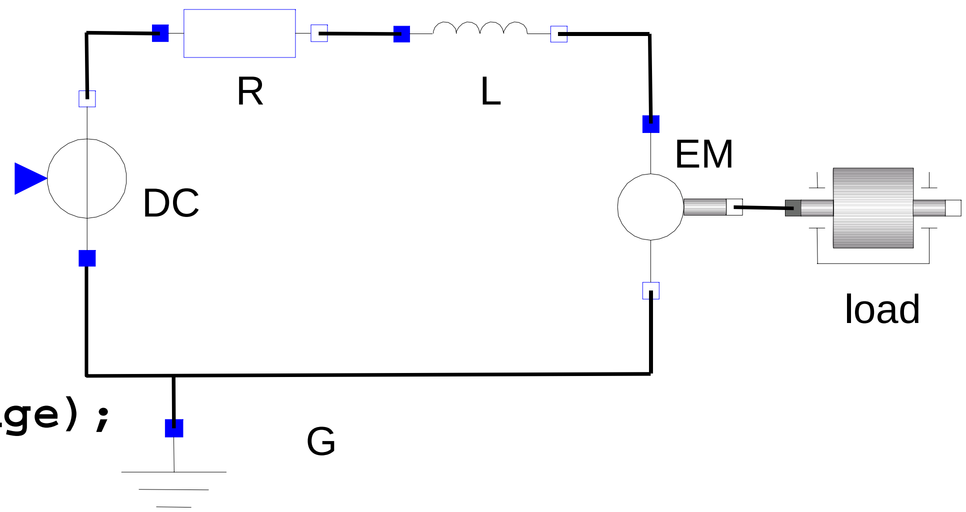



```
Srel = n*n' + (identity(3) - n*n')*cos(q) - skew(n)*sin(q);
wrela = n*qd;
zrela = n*qdd;
Sb = Sa*Srel';
r0b = r0a;
vb = Srel*va;
wb = Srel*(wa + wrela);
ab = Srel*aa;
zb = Srel*(za + zrela + cross(wa, wrela));
fa = Srel'*fb;
ta = Srel'*tb;
```

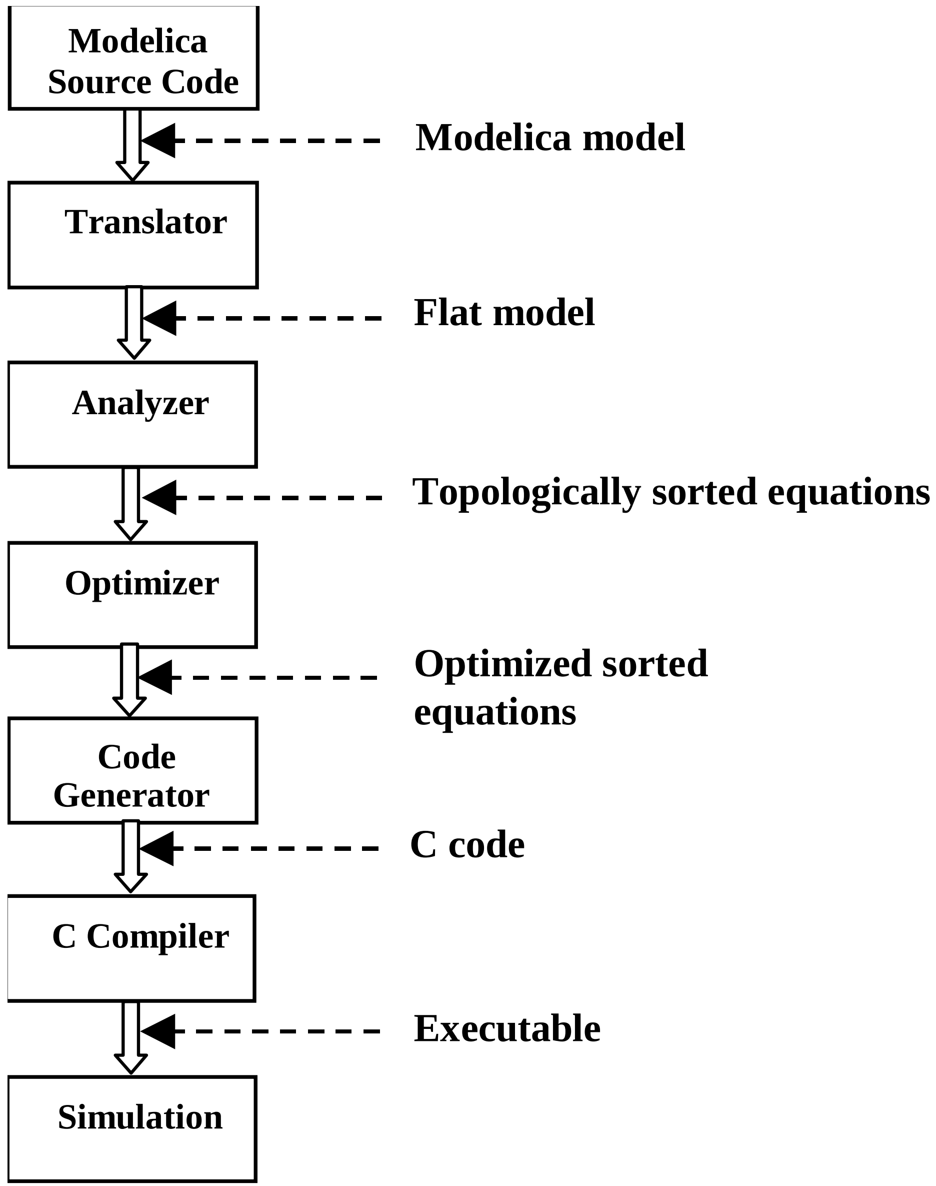
Multi-Domain Modelica Model - DCMotor

- A DC motor can be thought of as an electrical circuit which also contains an electromechanical component.

```
model DCMotor
   Resistor R(R=100);
   Inductor L(L=100);
   VsourceDC DC(f=10);
   Ground G;
   ElectroMechanicalElement EM(k=10,J=10, b=2);
   Inertia load;
equation
   connect(DC.p,R.n);
   connect(R.p,L.n);
   connect(L.p, EM.n);
   connect(EM.p, DC.n);
   connect(DC.n,G.p);
   connect(EM.flange,load.flange);
end DCMotor
```



Modelica compilation stages



Corresponding DCMotor Model Equations

The following equations are automatically derived from the Modelica model:

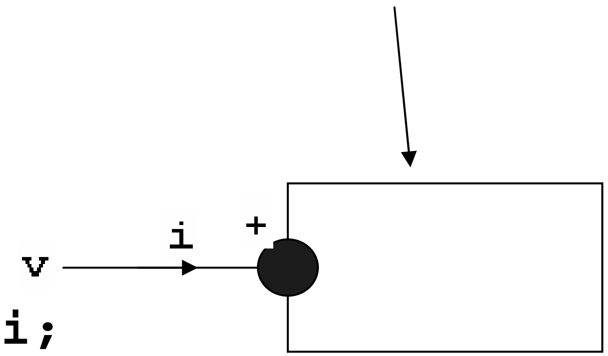
```
0 == DC.p.i + R.n.i
DC.p.v == R.n.v

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

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

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

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

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

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

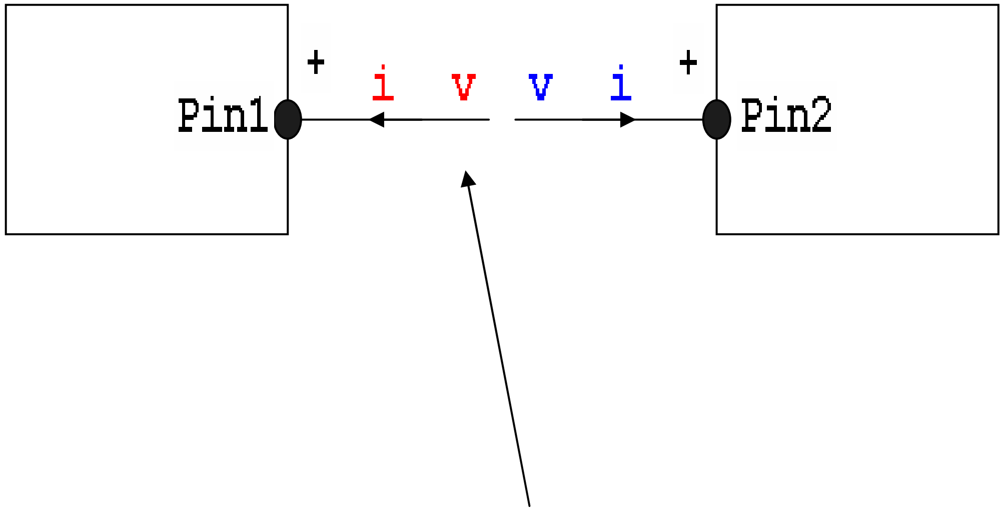
```
R.u == R.p.v - R.n.v
0 == R.p.i + R.n.i
R.i == R.p.i
R.u == R.R * R.i

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

(`load` component not included)

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



Keyword **flow** indicates that currents of connected pins sums to zero.



A connect statement in Modelica

```
connect(Pin1,Pin2)
```

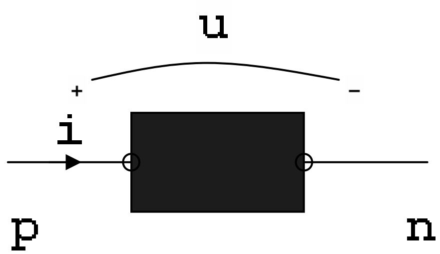
corresponds to

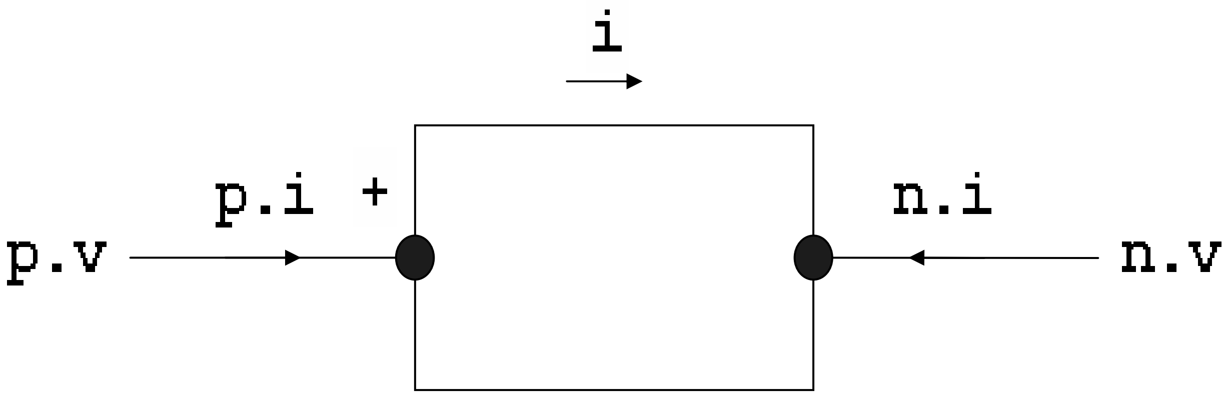
```
Pin1.v = Pin2.v
Pin1.i + Pin2.i = 0
```

Connection between Pin1 and Pin2

Common Component Structure as SuperClass

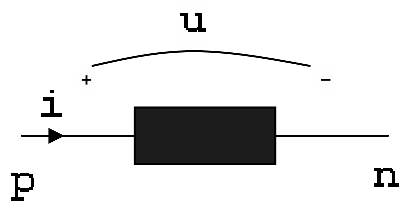
```
model TwoPin
   "Superclass of elements with two electrical pins"
   Pin p,n;
   Voltage v;
   Current i;
equation
  v = p.v - n.v;
  0 = p.i + n.i;
  i = p.i;
end TwoPin;
```



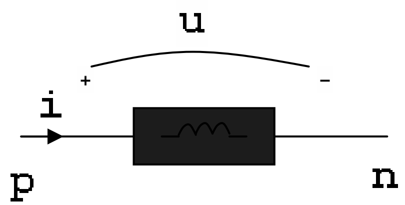


Electrical Components Reuse TwoPin SuperClass

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




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



Corresponding DCMotor Model Equations

The following equations are automatically derived from the Modelica model:

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

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

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

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

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

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

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

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

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

(`load` component not included)

MetaModelica - Context

- Syntax - there are many efficient parser generator tools
  - lex (flex), yacc (bison), ANTLR, Coco, etc.

- *Semantics:*
  - *there are no standard efficient and easy to use compiler-compiler tools*

MetaModelica - Motivation

- Can we adapt the Modelica equation-based style to define semantics of programming languages?
  - *Answer: Yes!*
- MetaModelica is just a part of the answer
  - executable language specification based on
    - a model (abstract syntax tree)
    - semantic functions over the model
      - elaboration and typechecking
      - translation
      - meta-programming
      - transformation
      - etc.
- Further improvement – more reuse of language specification parts when building specifications for a new language (Future Work)

MetaModelica - Idea

- We started from
  - The Relational Meta-Language (RML)
    - A system for building executable natural semantics specifications
    - Used to specify Java, Pascal-subset, C-subset, Mini-ML, etc.
  - The OpenModelica compiler for Modelica specified in RML

- Idea: *integrate RML meta-modeling and meta-programming facilities within OpenModelica by extending the Modelica language.*
  The notion of equation is used as the unifying feature

- Now we have
  - The MetaModelica language
  - The Modelica executable language specification (OpenModelica compiler) in MetaModelica (~114232 lines of code)
  - Meta-programming facilities for Modelica

MetaModelica extensions to Modelica (I)

- Modelica
  - classes, models, records, functions, packages
  - behaviour is defined by equations or/and functions
  - equations
    - differential equations
    - algebraic equations
    - partial differential equations
    - difference equations
    - conditional equations
- MetaModelica extensions
  - local equations
  - pattern equations
  - match expressions
  - lists, tuples, option and uniontypes

MetaModelica extensions to Modelica (II)

- pattern equations
  - unbound variables get their value by unification

```
Env.BOOLVAL(x,y) = eval_something(env, e);
```

- match expressions
  - pattern matching
  - case rules

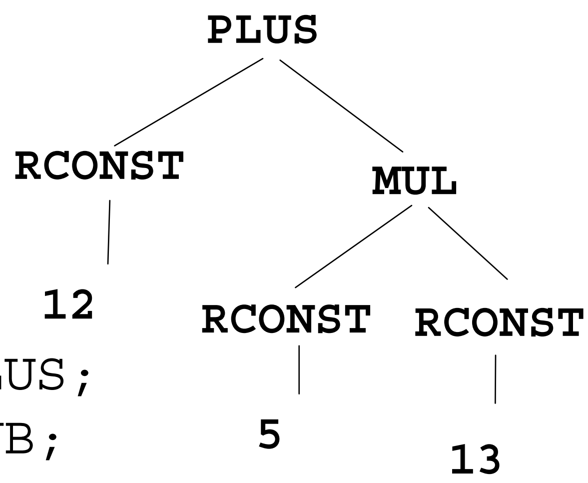
pattern := match *expression optional-local-declarations*
  case *pattern-expression opt-local-declarations*
    optional-local-equations **then** *value-expression*;
  case …
  …
  else *optional-local-declarations*
    optional-local-equations **then** *value-expression*;
end match;

MetaModelica – Example (I)

```
package ExpressionEvaluator


// abstract syntax declarations
...
// semantic functions
...

end ExpressionEvaluator;
```

MetaModelica – Example (II)

```
package ExpressionEvaluator

// abstract syntax declarations
uniontype Exp
  record RCONST Real x1; end RCONST;
  record PLUS   Exp x1; Exp x2; end PLUS;
  record SUB    Exp x1; Exp x2; end SUB;
  record MUL    Exp x1; Exp x2; end MUL;
  record DIV    Exp x1; Exp x2; end DIV;
  record NEG    Exp x1;         end NEG;
end Exp;


// semantic functions
...


end ExpressionEvaluator;
```



Expression: 12+5*13

Representation:

```
PLUS(
  RCONST(12),
  MUL(
   RCONST(5),
   RCONST(13)
  )
)
```

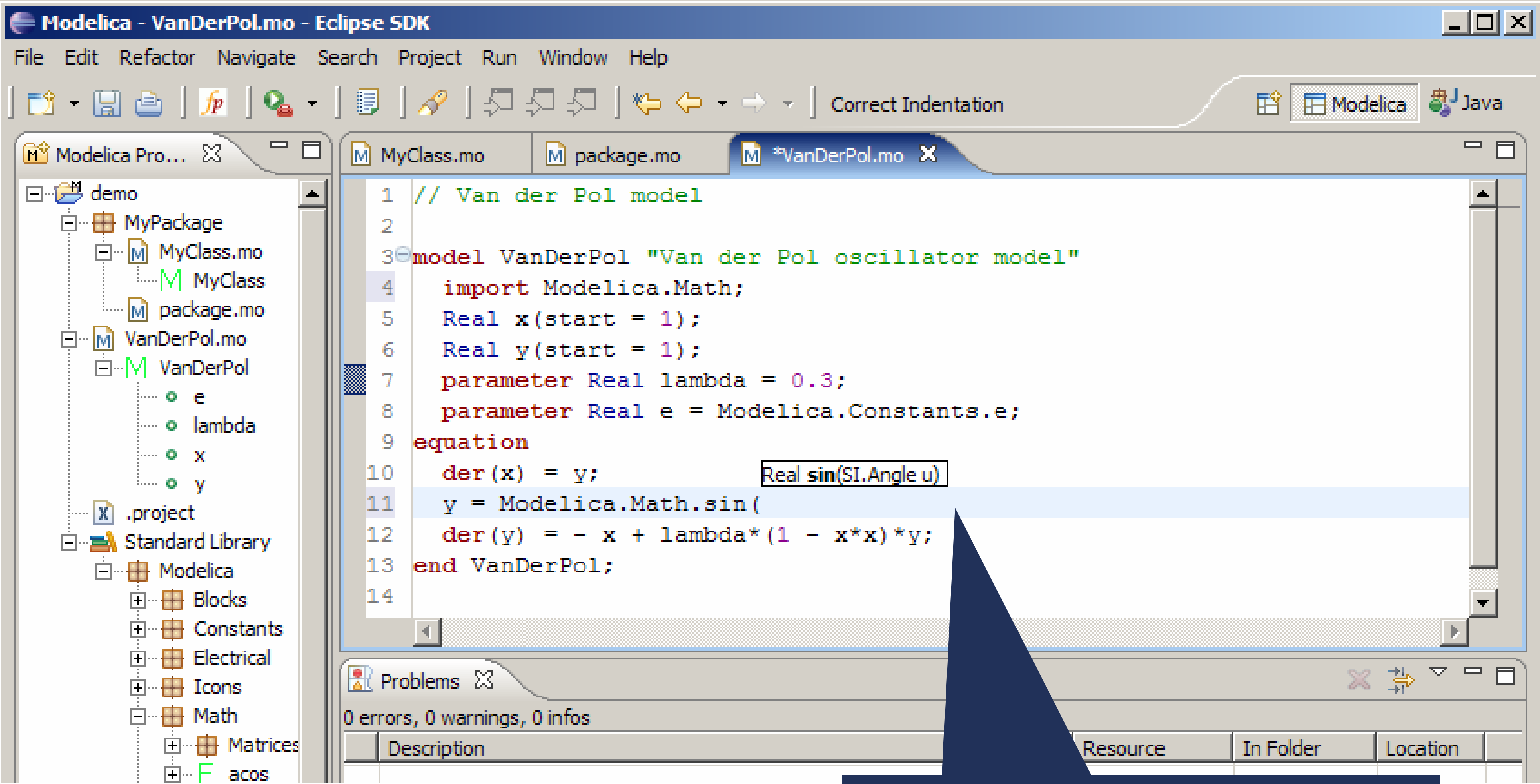
MetaModelica – Example (III)

```
package ExpressionEvaluator
// abstract syntax declarations
...

// semantic functions
function eval
  input  Exp  in_exp;
  output Real out_real;
algorithm
 out_real := match in_exp
    local Real v1,v2,v3;  Exp e1,e2;
    case RCONST(v1) then v1;
    case ADD(e1,e2) equation
      v1 = eval(e1);  v2 = eval(e2); v3 = v1 + v2;  then v3;
    case SUB(e1,e2) equation
      v1 = eval(e1);  v2 = eval(e2); v3 = v1 - v2;  then v3;
    case MUL(e1,e2) equation
      v1 = eval(e1);  v2 = eval(e2); v3 = v1 * v2;  then v3;
    case DIV(e1,e2) equation
      v1 = eval(e1);  v2 = eval(e2); v3 = v1 / v2;  then v3;
    case NEG(e1) equation
      v1 = eval(e1); v2 =  -v1;  then v2;
   end match;
end eval;

end ExpressionEvaluator;
```

Modelica/MetaModelica Development Tooling (MDT)

- Supports textual editing of Modelica/MetaModelica code as an Eclipse plugin
- Was created to ease the development of the OpenModelica development (114232 lines of code) and to support advanced Modelica library development
- It has most of the functionality expected from a Development Environment
  - code browsing
  - code assistance
  - code indentation
  - code highlighting
  - error detection
  - automated build of Modelica/MetaModelica projects
  - debugging

Modelica/MetaModelica Development Tooling



Code Assistance on function calling.

Conclusions and Future Work

- MetaModelica a language that integrates modeling of
  - physical systems
  - programming language semantics
- at the **equation** level

- MetaModelica is a step towards reusable libraries of specifications for programming language semantics

- Future Work
  - How do devise a suitable component model for the specification of a programming language semantics in terms of reusable components.
  - Tools to support such language modeling.

Thank you!
Questions?

http://www.ida.liu.se/labs/pelab/rml

http://www.ida.liu.se/labs/pelab/modelica/OpenModelica.html