Innovations for Future Modelica

Hilding Elmqvist, Toivo Henningsson, Martin Otter

Mogram Toivo H. DLR

Outline

- Rationale for *Modia* project
- Julia language
- Introduction to *Modia* Language
- *Modia* Prototype
- Summary

Why Modia?

- New needs of modeling features are requested
- Need an experimental language platform

- Modelica specification is becoming large and hard to comprehend
- Could be complemented by a reference implementation

- Functions/Algorithms in Modelica are not powerful
  - no advanced data structures such as union types, no matching construct, no type inference, etc
- Possibility to utilize other language efforts for functions
- Julia has perfect scientific computing focus
- Modia - Julia macro set

We hope to use this work to make contributions to the Modelica effort

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
- https://julialang.org/

Modia – "Hello Physical World" model

```
@model FirstOrder begin
  x = Variable(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;
```

Connectors and Components - Electrical

Modelica

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

@model OnePort begin
  p=Pin()
  n=Pin()
  v=Float()
  i=Float()
@equations begin
  v = p.v - n.v # Voltage drop
  0 = p.i + n.i # KCL within component
  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
```

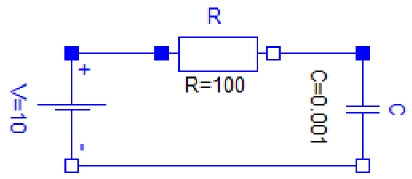
```
connector Pin
  Modelica.SIunits.Voltage v;
  flow Modelica.SIunits.Current I;
end Pin;

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;
```

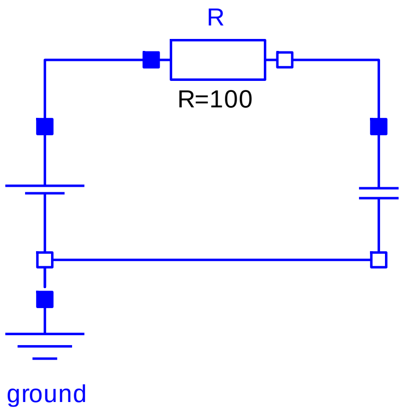
Coupled Models - Electrical Circuit

Modelica

```
@model LPfilter begin
  R = Resistor(R=100)
  C = Capacitor(C=0.001)
  V = ConstantVoltage(V=10)
@equations begin
  connect(R.n, C.p)
  connect(R.p, V.p)
  connect(V.n, C.n)
  end
end
```

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





Mogram Toivo H. DLR

Type and Size Inference - Generic switch

```
@model Switch begin
  sw=Boolean()
  u1=Variable()
  u2=Variable()
  y=Variable()
@equations begin
  y = if sw; u1 else u2 end
  end
end
```

- Avoid duplication of models with different types
- Types and sizes can be inferred from the environment of a model or start values provided,
  either initial conditions for states or
  approximate start values for algebraic constraints.
- Inputs u1 and u2 and output y can be of any type

Variable Declarations

```
# With Float64 type
v1 = Var(T=Float64)

# With array type
array = Var(T=Array{Float64,1})
matrix = Var(T=Array{Float64,2})

# With fixed array sizes
scalar = Var(T=Float64, size=())
array3 = Var(T=Float64, size=(3,))
matrix3x3 = Var(T=Float64, size=(3,3))
```

```
# With unit
v2 = Var(T=Volt)

# Parameter with unit
m = 2.5kg
length = 5m
```

- Often natural to provide type and size information

Type Declarations

- Reuse of type and size definitions
- Rotatation matrices

```
# Type declarations
Float3(; args...) = Var(T=Float64, size=(3,); args...)
Voltage(; args...) = Var(T=Volt; args...)

# Use of type declarations
v3 = Float3(start=zeros(3))
v4 = Voltage(size=(3,), start=[220.0, 220.0, 220.0]Volt)

Position(; args...) = Var(T=Meter; size=(), args...)

Position3(; args...) = Position(size=(3,); args...)
Rotation3(; args...) = Var(T=SIPrefix; size=(3,3), property=rotationGroup3D, args...)
```

Redeclaration of submodels

```
MotorModels = [Motor100KW, Motor200KW, Motor250KW] # array of Modia models
selectedMotor = motorConfig(  ) # Int

@model HybridCar begin
  @extends BaseHybridCar(
    motor = MotorModels[selectedMotor](),
    gear  = if gearOption1; Gear1(i=4) else Gear2(i=5) end)
end
```

- More powerful than **replaceable** in Modelica

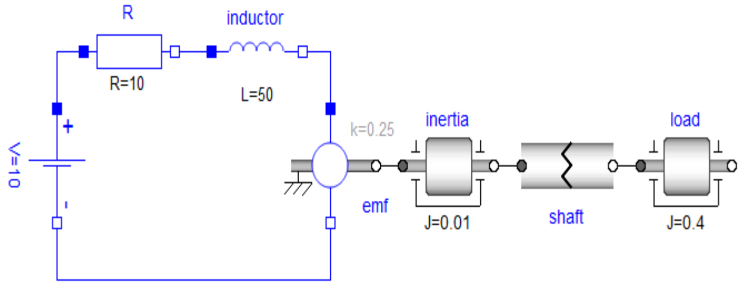
Indexing

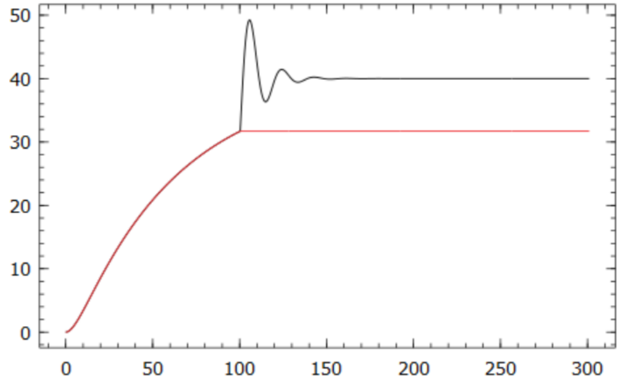
Conditional selection

Multi-mode Modeling

```
@model BreakingShaft begin
  flange1 = Flange()
  flange2 = Flange()
  broken = Boolean()
@equations begin
    if broken
      flange1.tau = 0
      flange2.tau = 0
    else
      flange1.w = flange2.w
      flange1.tau + flange2.tau = 0
    end
  end
end
```

- set of model equations and the DAE index is changing when the shaft breaks
- new symbolic transformations and just-in-time compilation is made for each mode of the system
- final results of variables before an event is used as initial conditions after the event
- mode changes with conditional equations might introduces inconsistent initial conditions causing Dirac impulses to occur
- this more general problem is treated in another publication





MultiBody Modeling

```
@model Frame begin
  r_0 = Position3()
  R = Rotation3()
  f = Force3(flow=true)
  t = Torque3(flow=true)
end
```

- Rotation3() implies ”special orthogonal group”, SO(3)
- Compared to current Modelica, the benefit is that
  no special operators Connections.branch/.root/.isRoot etc are needed anymore

Functions and data structures

- built-in operator allInstances(v) creates a vector of all the variables v within all instances of the class where v is declared

```
@model Ball begin
  r = Var()
  v = Var()
  f = Var()
  m = 1.0
@equations begin
  der(r) = v
  m*der(v) = f
  f = getForce(r, v, allInstances(r), allInstances(v), (r,v) -> (k*r + d*v))
  end
end

@model Balls begin
  b1 = Ball(r = Var(start=[0.0,2]), v = Var(start=[1,0]))
  b2 = Ball(r = Var(start=[0.5,2]), v = Var(start=[-1,0]))
  b3 = Ball(r = Var(start=[1.0,2]), v = Var(start=[0,0]))
end
```

```
function getForce(r, v, positions, velocities, contactLaw)
  force = zeros(2)
  for i in 1:length(positions)
    pos = positions[i]
    vel = velocities[i]
    if r != pos
      delta = r - pos
      deltaV = v - vel
      f = if norm(delta) < 2*radius;
        -contactLaw((norm(delta)-2*radius)*delta/norm(delta), deltaV)
        else zeros(2) end
      force += f
    end
  end
  return force
end
```

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

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]))
```

Summary - *Modia*

- Modelica-like, but more powerful and simpler
- Algorithmic part: Julia functions (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)

Summary

- *Modia:* environment to experiment with
  - new algorithms (see companion paper)
  - new language elements for Modelica (e.g. allInstances for contact handling, ...)
- Structural and Symbolic algorithms: Julia data structures / functions
- Algorithmic part: Julia functions (more powerful than Modelica)
- Model part: Julia meta-programming (no Modia compiler)
- Equation part: Julia expressions (no Modia compiler)
- Target equations: Sparse index-1 DAE (no ODE)
- Structural and Symbolic algorithms: Julia data structures / functions
- Just-in-time compilation (build Modia model and simulate at once)
- Simulation engine: IDA (Sundials) + KLU sparse matrix