Modia - A Domain Specific Extension of Julia for Modeling and Simulation

Hilding Elmqvist, Mogram AB, Lund, Sweden

Co-developers: Toivo Henningsson, Martin Otter
Outline

• Rationale for Modia project
• Modia Language by examples
• Modia Prototype
• Summary
Modelica for Systems Modeling

- www.Modelica.org
- 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
Modelica Basics

• Object- and equation-oriented modeling language
• Successfully utilized in industry for modeling, simulating and optimizing complex systems such as automobiles, aircraft, power systems, etc.
• The dynamic behavior of system components is modelled by equations, for example, mass- and energy-balances.
  • Ordinary Differential Equations
  • Algebraic Equations
  • = DAE (Differential Algebraic Equations)
• Modelica is quite different from ordinary programming languages since equations with mathematical expressions on both sides of the equals sign are allowed.
• Structural and symbolic methods are used to compile such equations into efficient executable code.
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
Modia – “Hello Physical World” model

@model FirstOrder
begin
    x = Variable(start=1)
    T = Parameter(0.5, info="Time constant")
    u = 2.0 # Same as Parameter(2.0)
@equations
begin
    T*der(x) + x = u
end
end
Connectors and Components - Electrical

@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
Coupled Models - Electrical Circuit

@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
Web App – for connecting components

- Bachelor thesis project
- Create, connect, set parameters, simulate, plot, animate in 3D
- Automatic placement and routing
- Together with Modelon AB
Web App – on smart phone

- Scenario:
- Working in your autonomous car
Variable Constructor

In general: time varying variable with attributes:

type Variable
  variability::Variability
  T::DataType
  size
  value
  unit::SIUnits.SIUnit
  min
  max
  start
  nominal
  info::AbstractString
  flow::Bool
end

Short version:

`Var(; args...) = Variable(; args...)`

Specialization for parameters:

`Parameter(value; args...) = Var(variability=parameter, value=value; args...)`
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
• Unit handling with SIUnits.jl
Type Declarations

# 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...)
MultiBody modeling

@model Frame begin
r_0 = Position3()
R = Rotation3()
f = Force3(flow=true) # Cut-force resolved in connector frame
t = Torque3(flow=true) # Cut-torque resolved in connector frame
end

@model Revolute begin # Revolute joint (1 rotational degree-of-freedom, 2 potential states, optional axis flange)
n = [0,0,1] # Axis of rotation resolved in frame_a
frame_a = Frame()
frame_b = Frame()

phi = Angle(start=0)
w = AngularVelocity(start=0)
a = AngularAcceleration()
tau = Torque() # Driving torque in direction of axis of rotation
R_rel = Rotation3()
end

equations begin
R_rel = n*n' + (eye(3) - n*n')*cos(phi) – skew(n)*sin(phi)
w = der(phi)
a = der(w)
frame_b.r_0 = frame_a.r_0
frame_b.R = R_rel*frame_a.R
frame_a.f + R_rel'*frame_b.f = zeros(3)
frame_a.t + R_rel'*frame_b.t = zeros(3)

d'Alemberts principle
tau = -n'*frame_b.t
tau = 0 # Not driven
end
end

• Matrix equations
• DAE index reduction needed
• R_Rel equation differentiated (only phi time varying)
• Rotation3() implies "special orthogonal group", SO(3)
Type and Size Inference - Generic switch

@model Switch begin
sw=True()
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
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
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)
end

@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
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 Clutch begin
flange1 = Flange()
flange2 = Flange()
engaged = Boolean()
@end

@equations begin
if ! engaged
    flange1.tau = 0
    flange2.tau = 0
else
    flange1.w = flange2.w
    flange1.tau + flange2.tau = 0
end
end

• Set of model equations and the DAE index is changing when clutch is engaged or disengaged
• 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
Functions and data structures

@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
  return force
end

• built-in operator allInstances(v) creates a vector of all the variables v within all instances of the class where v is declared
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.
BLT (Block Lower Triangular) form

\[
\begin{align*}
\text{error.u1} &= \text{step.offset} + \text{(if time < step.startTime then 0 else step.height)} \\
\text{error.y} &= \text{error.u1} - \text{load.w} \\
\text{Vs.p.v} &= \text{P.k} \times \text{error.y} \\
\text{Ra.R} \times \text{La.p.i} &= \text{Vs.p.v} - \text{Ra.n.v} \\
\text{Jm.w} &= \text{gear.ratio} \times \text{load.w} \\
\text{emf.k} \times \text{Jm.w} &= \text{La.n.v} \\
\text{La.L} \times \text{der(La.p.i)} &= \text{Ra.n.v} - \text{La.n.v} \\
\text{emf.flange.tau} &= -\text{emf.k} \times \text{La.p.i} \\
\end{align*}
\]

// System of 4 simultaneous equations

\[
\begin{align*}
\text{der(Jm.w)} &= \text{gear.ratio} \times \text{der(load.w)} \\
\text{Jm.J} \times \text{der(Jm.w)} &= \text{Jm.flange_b.tau} - \text{emf.flange.tau} \\
0 &= \text{gear.flange_b.tau} \times \text{gear.ratio} \times \text{Jm.flange_b.tau} \\
\text{load.J} \times \text{der(load.w)} &= -\text{gear.flange_b.tau} \\
\text{der(load.flange_a.phi)} &= \text{load.w} \\
\text{emf.flange.phi} &= \text{gear.ratio} \times \text{load.flange_a.phi} \\
\text{G.p.i} + \text{La.p.i} &= \text{La.p.i}
\end{align*}
\]
strongConnect (BLT)

Find minimal systems of equations that have to be solved simultaneously.

Reference:

function strongConnect(G, assign, v, nextnode, stack, components, lowlink, number)
const notOnStack = typemax(Int)

if v == 0
    return nextnode
end

nextnode += 1
lowlink[v] = number[v] = nextnode
push!(stack, v)

for w in [assign[j] for j in G[v]] # for w in the adjacency list of v
    if w > 0  # is assigned
        if number[w] == 0 # if not yet numbered
            nextnode = strongConnect(G, assign, w, nextnode, stack, components, lowlink, number)
            lowlink[v] = min(lowlink[v], lowlink[w])
        else
            if number[w] < number[v]
                # (v, w) is a frond or cross-link
                # if w is on the stack of points. Always valid since otherwise number[w]=notOnStack (a big number)
                lowlink[v] = min(lowlink[v], number[w])
            end
        end
    end
end

if lowlink[v] == number[v]
    # v is the root of a component
    # start a new strongly connected component
    comp = []
    repeat = true
    while repeat
        # delete w from point stack and put w in the current component
        w = pop!(stack)
        number[w] = notOnStack
        push!(comp, w)
        repeat = w != v
    end
    push!(components, comp)
end

return nextnode
end
Julia AST for Meta-programming

julia> equ = :(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]))
:

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]))

- Quoted expression :()
- Any expression in LHS
- Operators are functions
- $ for “interpolation”
Summary – Modia Prototype

• Modelica-like, but more powerful and simpler
• Algorithmic part: Julia functions (more powerful than Modelica functions)
• Model part: Julia meta-programming (no Modia parser)
• Equation part: Julia expressions (no Modia parser)
• 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)
Next Immediate Steps

• Larger test suit
• Handle larger models (problem with code generation of big functions)
• Automated testing
• Coverage
• Julia package
• Proper web server (now Python SimpleJSONRPCServer and PyJulia)
• Cloud deployment
• Release to github (https://github.com/ModiaSim/Modia.jl)

• Begins on Saturday hackaton (hopefully with some help)
References

