Introduction to OMJL

OpenModelica in Julia

John Tinnerholm
St. Anna IT Research

MODPROD 2026 Tutorial

Attribution

These slides and some examples are Based on book and lecture notes by Peter Fritzson and the tutorial for OpenModelica with the contributions listed as follows:

Contributions 2004-2005 by Emma Larsdotter Nilsson, Peter Bunus
Contributions 2006-2018 by Adrian Pop and Peter Fritzson
Contributions 2009 by David Broman, Peter Fritzson, Jan Brugård, and Mohsen Torabzadeh-Tari
Contributions 2010 by Peter Fritzson
Contributions 2011 by Peter F., Mohsen T,. Adeel Asghar,
Contributions 2012-2018 by Peter Fritzson, Lena Buffoni, Mahder Gebremedhin, Bernhard Thiele, Lennart Ochel
Contributions 2019-2023 by Peter Fritzson, Arunkumar Palanisamy, Bernt Lie, Adrian Pop

Tutorial Outline

- Part 1: Modelica Basics
- Part 2: Julia-Modelica Integration
- Part 3: Language Extensions

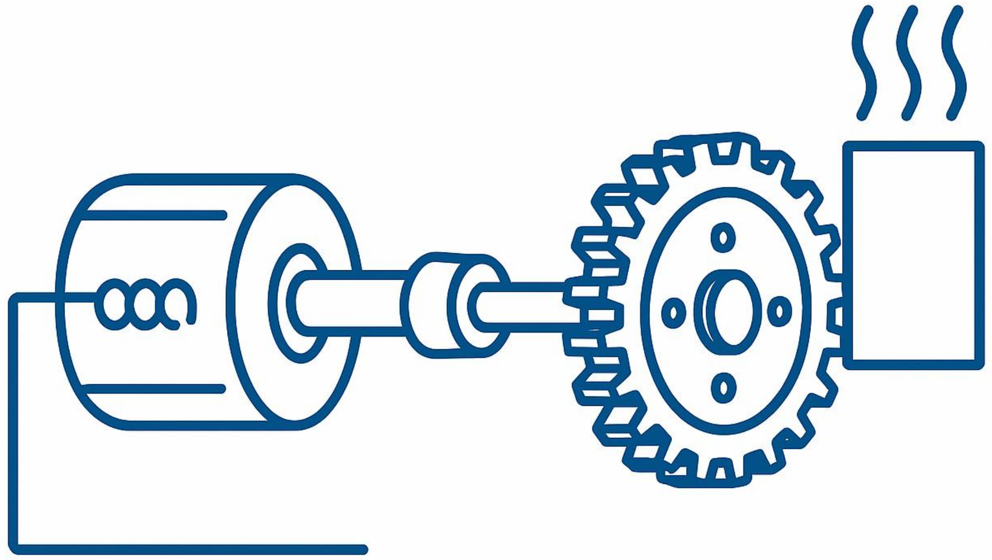
- Each part includes some hands-on notebook exercises

Part 1

Modelica Basics

What is Modelica?

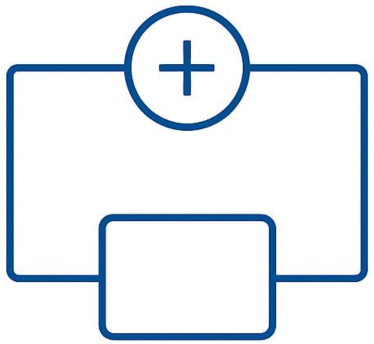
- Equation-based modeling language
- Declarative: describe what, not how
- Multi-domain physical modeling
- Object-oriented: classes, inheritance



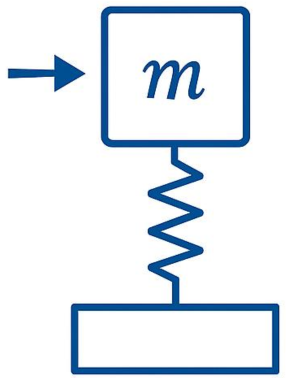
Equation-Based Modeling

- Describe physical relationships directly
  - V = I * R (Ohm's law)
  - F = m * a (Newton's second law)
- Compiler transforms equations for simulation

$$V = IR$$

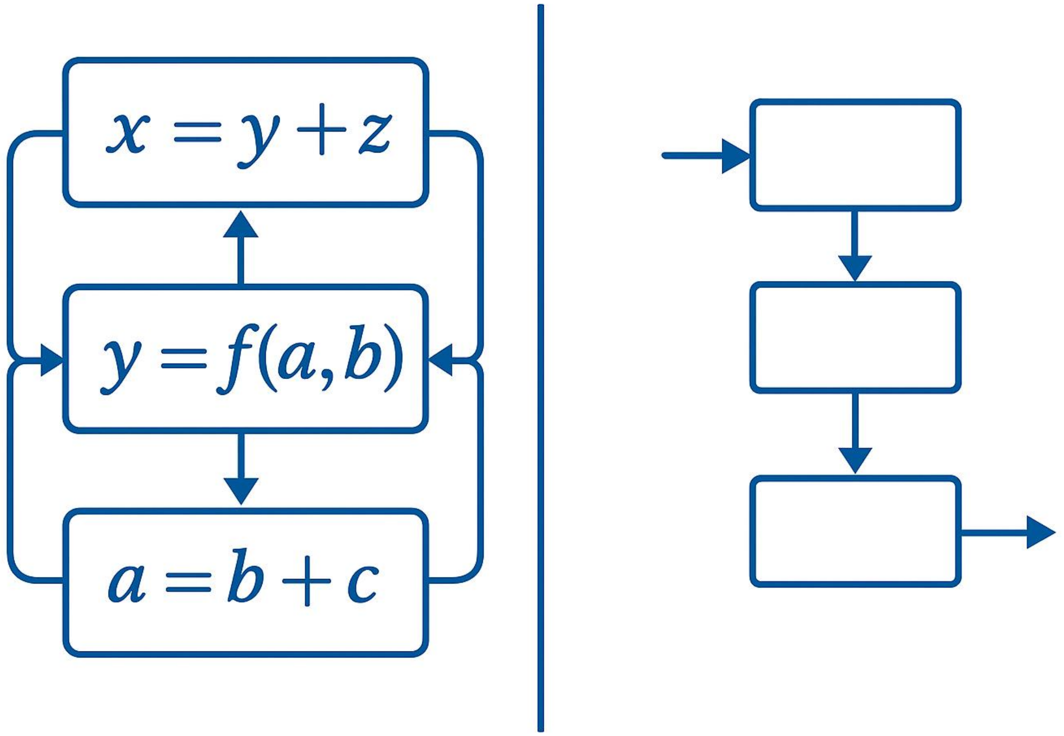


$$F = ma$$



Declarative vs Imperative

- Imperative (C, C++, Python, Julia..):
  - y = f(x) means compute f(x), store in y
- Declarative (Modelica):
  - y = f(x) means y and f(x) are always equal!
  - y := f(x) means compute f(x), store in y
- Equations can be solved in any direction



A Simple Modelica Model

```
model HelloWorld
  Real x(start = 1);
equation
  der(x) = -x;
end HelloWorld;
```

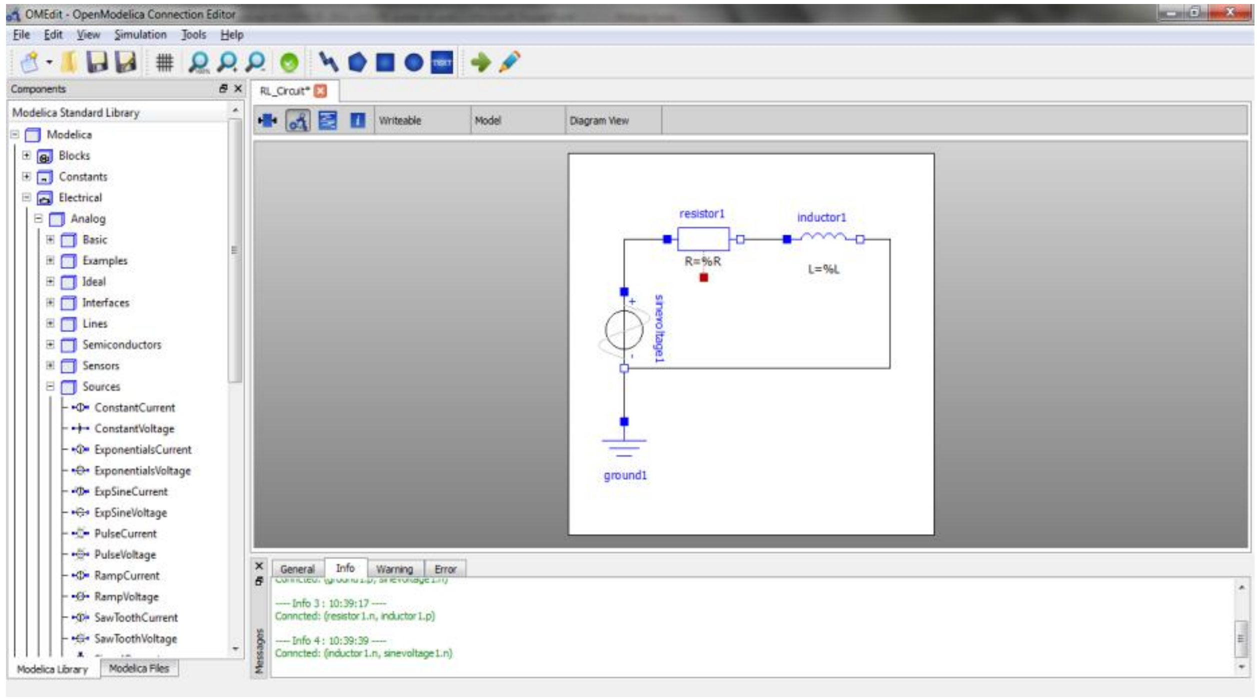
Variables and Types (Simplified)

- Real: Continuous variables
- Integer: Discrete integers
- Boolean: True/False
- String: Text

- Modifiers: parameter, constant, input, output

Graphical Modeling

- Components connected visually
- Drag and drop from libraries
- Connections generate equations
- OMJL results viewable in OMEdit
  - ❑ **To do this if you have OpenModelica installed use the export function in OMJL on the obtained simulation results**



```
---- Info 3 : 10:39:17 ----
Conncted: (resistor1.n, inductor1.p)

---- Info 4 : 10:39:39 ----
Conncted: (inductor1.n, sinevoltage1.n)
```

Notebook Exercise 1

- Open the notebook

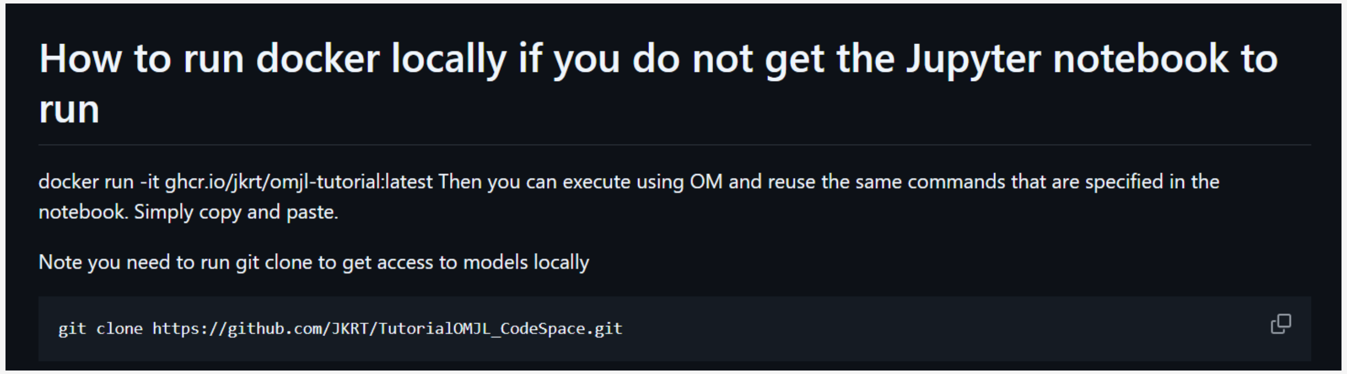
https://github.com/JKRT/TutorialOMJL_CodeSpace

➔ docker pull ghcr.io/jkrt/omjl-tutorial:latest (Docker container with the notebook built in)

➔ export JUPYTER_IP=0.0.0.0

➔ julia -e 'using IJulia; notebook(dir="/workspaces/TutorialOMJL_CodeSpace/notebooks")'

- 1. Run your first Modelica model in OMJL
- 2. Simulate the HelloWorld model
- 3. Plot the results
- 4. Try modifying the initial value



How to run docker locally if you do not get the Jupyter notebook to run

docker run -it ghcr.io/jkrt/omjl-tutorial:latest Then you can execute using OM and reuse the same commands that are specified in the notebook. Simply copy and paste.

Note you need to run git clone to get access to models locally

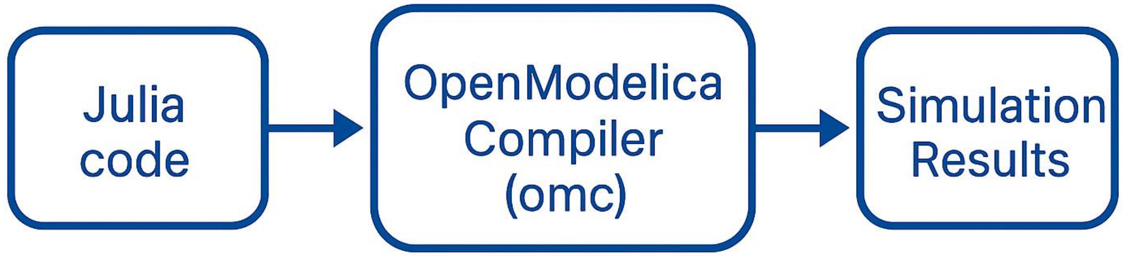
```
git clone https://github.com/JKRT/TutorialOMJL_CodeSpace.git
```

Part 2

Julia-Modelica Integration

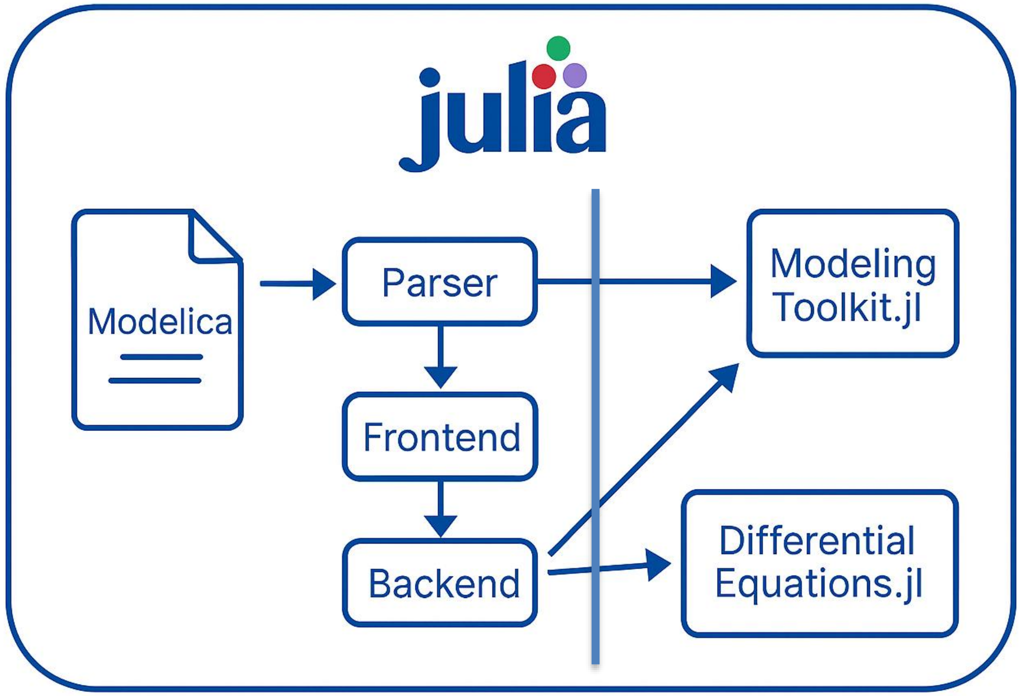
OMJulia: OpenModelica + Julia

- Julia interface to OpenModelica compiler
- Calls omc via scripting API
- Modelica code is compiled externally by omc
- Julia for scripting, pre/post-processing



OMJL: Modelica in Julia

- Native Julia implementation
- Parser, Frontend, Backend in Julia
- Generates code via ModelingToolkit.jl
- *Full Julia ecosystem integration*



OMJulia vs OMJL

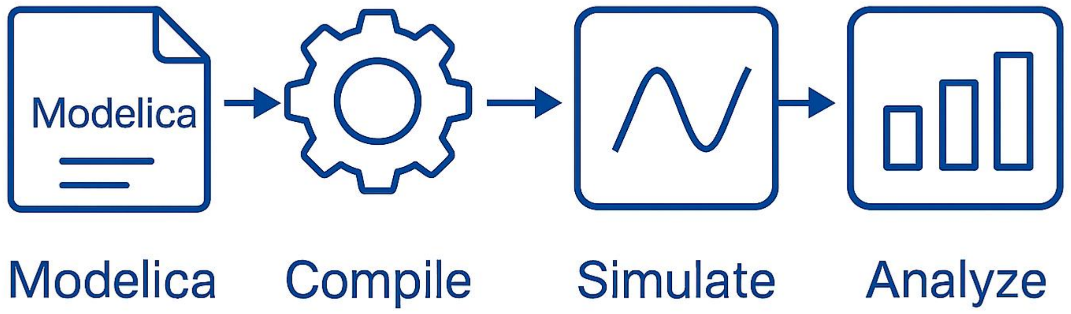
- OMJL:
  - Native Julia implementation
  - Research platform, extensible
  - Deeper Julia/MTK integration

- OMJulia:
  - Uses existing omc compiler
  - Mature, full Modelica support
  - Julia as scripting layer

OMJL API

```julia
using OM, Plots

# Translate and simulate
OM.translate("HelloWorld",
             "HelloWorld.mo")
sol = OM.simulate("HelloWorld")

# Plot results
plot(sol)

# Export for OMEdit
OM.exportCSV("HelloWorld", sol)
```

Simple workflow: translate, simulate, analyze



From Modelica to Julia

Modelica Model

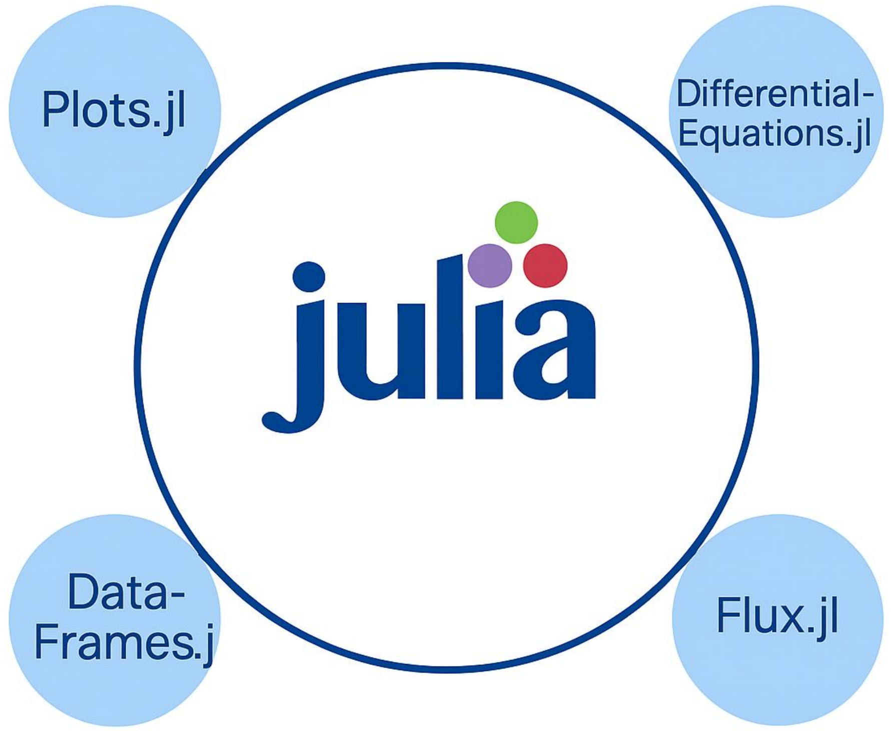
```
model LotkaVolterra
  Real x(start=10);
  Real y(start=10);
  parameter Real a=0.1;
  parameter Real b=0.02;
  parameter Real c=0.3;
  parameter Real d=0.01;
equation
  der(x) = x*(a - b*y);
  der(y) = y*(d*x - c);
end LotkaVolterra;
```

Julia Usage

```
using OM, Plots

# Compile model
OM.translate("LotkaVolterra",
    "LotkaVolterra.mo")

# Simulate
sol = OM.simulate("LotkaVolterra";
    stopTime=100.0)

# Plot predator-prey dynamics
plot(sol, label=["prey" "predator"])
```

Potential for Machine Learning Integration

- Julia has strong ML ecosystem (SciML, ModelingToolkit etc.)
- OMJL enables:
  - Training ML on Modelica simulation data
  - Parameter estimation
  - ....



Notebook Exercise 2

- Open the Part 2 notebook

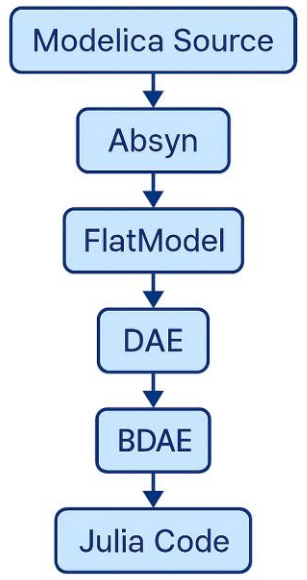
- 1. Load and simulate LotkaVolterra
- 2. Run parameter sweep varying a and b
- 3. Plot multiple trajectories
- 4. Export results to CSV

Part 3

Language Extensions

OMJL Architecture

- Compilation pipeline:
  - Modelica source -> Absyn
  - Absyn -> SCode -> FlatModel
  - FlatModel -> DAE -> BDAE
  - BDAE -> SimulationCode -> MTK -> Julia
- Uses DifferentialEquations.jl solvers

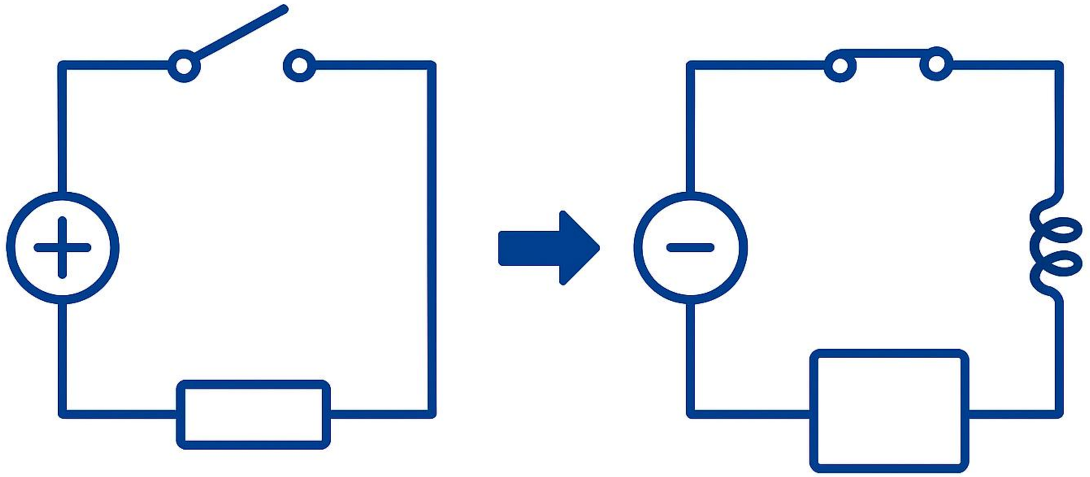


Beyond Standard Modelica

- OMJL as a research platform
- Experiment with language extensions
- Test features before standardization
- Tight Julia/MTK integration

Variable-Structure Systems

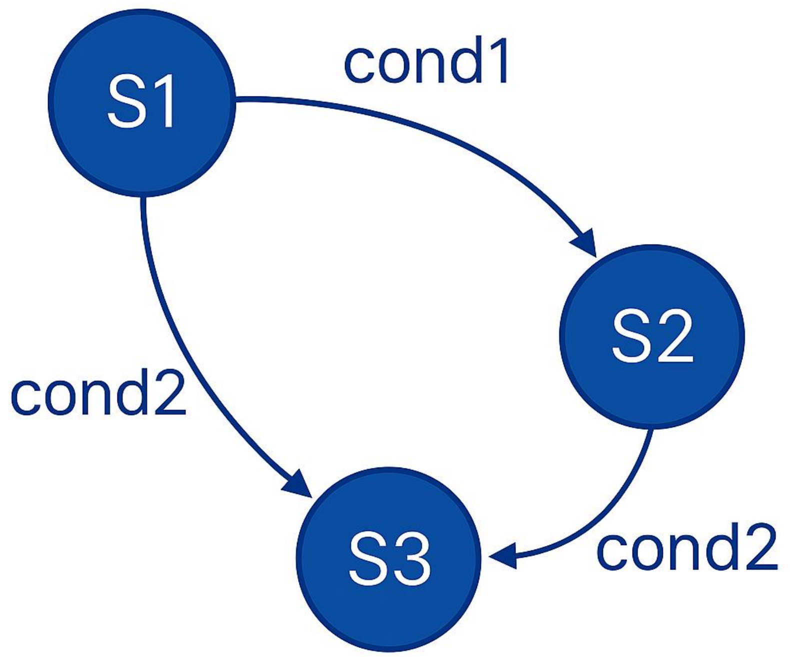
- Systems that change during simulation
- Number of equations can change
- Example: circuit breakers, clutches
- OMJL supports structural modes



Structural Mode Transitions

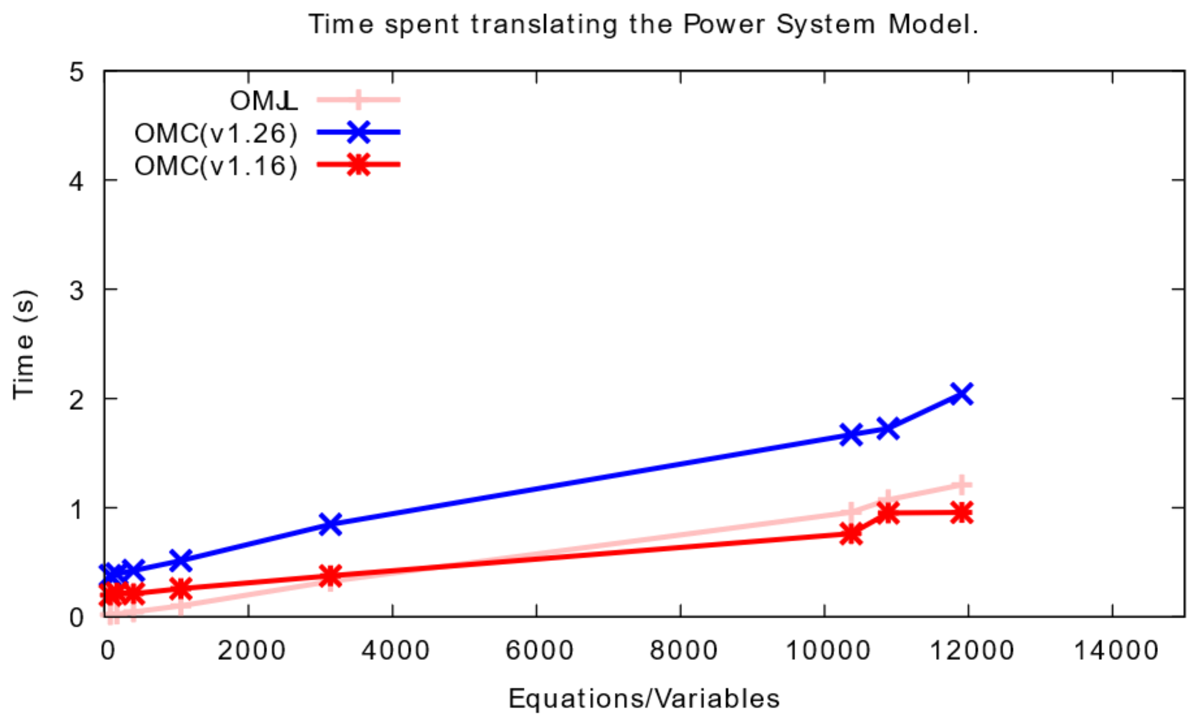
```
model VSS
  structuralmode S1 s1;
  structuralmode S2 s2;
  Real x(start=1.0);
equation
  initialStructuralState(s1);
  structuralTransition(s1, s2,
      time > 0.5);
end VSS;
```

Mode transitions trigger recompilation



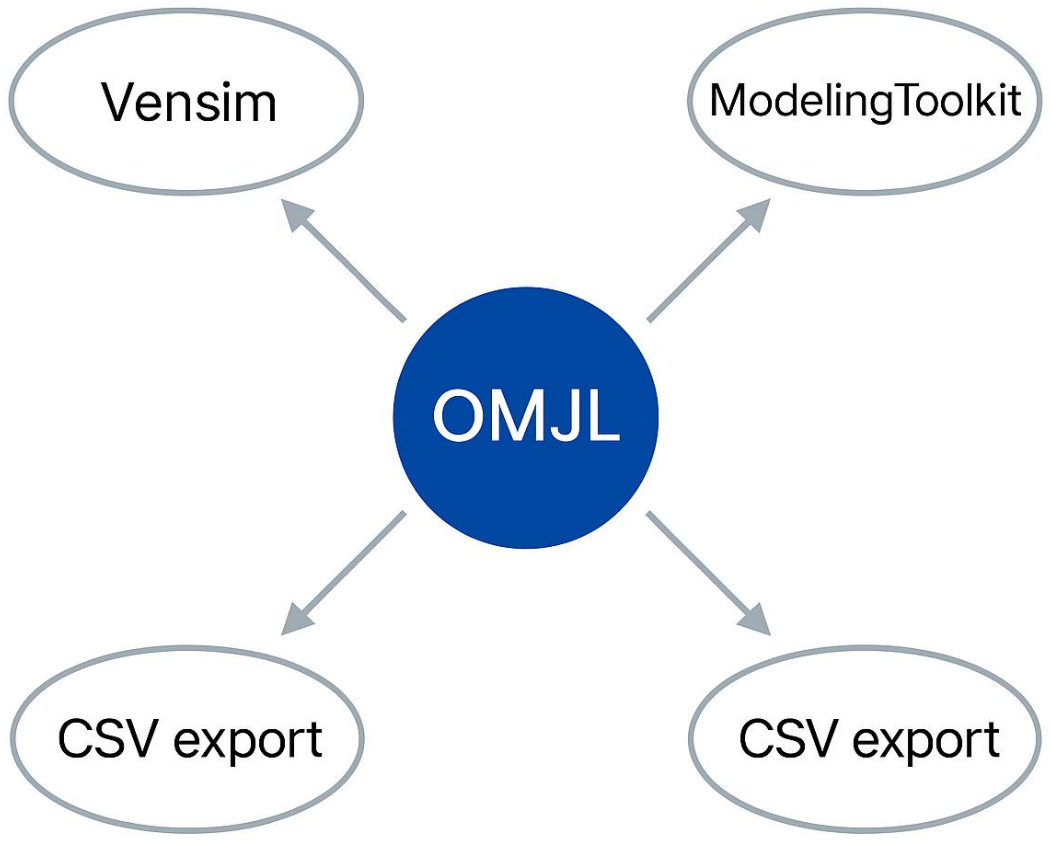
Performance

- OMJL vs OMC
- Benefits from Julia JIT compilation
- Scales to large models
- Ongoing optimization work



Interoperability

- Integration with other tools
  - Vensim system dynamics
  - MTK
  - CSV export for OMEdit



Future Directions

- Improved MSL (Standard Library) support
- Improved MTK integration
- Improved code generation
- Community contributions welcome

Notebook Exercise 3

- Open the Part 3 notebook

- 1. Explore the BouncingBall hybrid model
- 2. Try a Variable-Structure System example
- 3. Experiment with the API

Summary

What We Covered

- Part 1: Modelica basics
- Part 2: Julia integration (OMJulia vs OMJL)
- Part 3: Language extensions

Resources

- OpenModelica
- Julia
- SciML DifferentialEquations.jl
- ModelingToolkit
- Modelica homepage

john.Tinnerholm@liu.se

Questions?