Modelica in the Julia Environment: Latest Developments and Prospects

John Tinnerholm
An OpenModelica Environment in Julia

- **Goal**
  - An OpenModelica Environment in Julia

- **This talk**
  - Overview of OpenModelica.jl
  - Some Current Challenges
  - Future Development

---

**Experimental OpenModelica Environment in Julia**

- **OpenModelica.jl**: A Modular and Extensible Modelica compiler framework in Julia
- Translated the high-performance front end.
- Able to execute and translate Modelica/MetaModelica functions
- Able to simulate discrete-hybrid systems + regular continuous systems
- **Experimental backends developed**
  - Targeting DifferentialEquations.jl and ModelingToolkit.jl (MTK)
  - Casualization sorting, matching...
  - Integrated LightGraphs.jl package, DAG representation of the hybrid DAE
  - Integration with Sundials. IDAS used for numerical integration
  - Integrated Plots.jl for interactive plotting and animation
- Alpha is released, a Beta Release in the workings
  - MSL support (New 2022/2023)
    - Working on full coverage in the frontend
    - Full code generation for the backend
  - New Low Level Code Generator (New 2023)
  - Support for Algorithmic Code Generation (New 2023)
  - A System Dynamic importer in progress
  - Updated for Julia 1.10 (January 2024)
  - Optimization of both runtime and compiler structures in the frontend and in the backend
  - MsC Thesis using OpenModelica.jl to propose a new language done at TU-Dresden Autumn 2023
  - Two Bachelor Theses in progress at TU Dresden

- **Supporting Experimental Modelica Features**:
  - Language extensions for variable-structure system support (2022)
  - Dynamic Overconstrained Connectors (2022)
  - THETA (New 2023)
Frontend was modified so it can flatten models in separation.

Note requires the structural mode keyword.

Possible to formulate model with varying index and compile AOT.

To the left we can see the flat model definition in the Julia Modelica compiler and an example of a breaking pendulum model.
Language extensions for variable-structure system support (2022)

Possible to formulate recursive models that expand during simulation
Dynamic Overconstrained Connectors

• Currently, Overconstrained Connectors in Modelica cannot be used in If-Equations\(^3\)
  • Relaxing constraints
• Allowing a special If-Equation construct where the Connectors.branch operator is allowed
  • Allowing changing the connection graph dynamically at runtime.
• More efficient simulations
• Allows the simulations of models current tools are unable to simulate

```model TransmissionLineVariableBranch
  extends TransmissionLineBase;
  equation
    if closed then
      port_a.omegaRef = port_b.omegaRef;
      Connections.branch(port_a.omegaRef, port_b.omegaRef);
    end if;
end TransmissionLineVariableBranch;
```

➢ Restricted case of VSS, efficient no recompilation needed. Could be implemented in a traditional compiler using value propagation and pointer swapping

\(^3\)https://specification.modelica.org/maint/3.5/connectors-and-connections.html#restrictions-of-connections-and-connectors
THETA-Operator

```plaintext
package CircuitTest
model ThetaCircuit2Dynamic
    parameter Real THETA = 1.0;
    extends Circuit1Static;
    Capacitor Cp(C = 1e-12 + THETA);
    equation
        connect(Cp.n, ground.p);
        connect(diode.n, Cp.p);
end ThetaCircuit2Dynamic;
end CircuitTest;
```
System Dynamics and Algorithmic Modelica

The Resulting Translation

The translation was done according to the XMILE specification

SD Elements were mapped to the corresponding Modelica Elements

Table 1. Subset of Modelica to SD type mappings

<table>
<thead>
<tr>
<th>SD Type</th>
<th>Modelica Formulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>stock</td>
<td>$\text{der(stock)} = \text{inflows} - \text{outflows}$</td>
</tr>
<tr>
<td>smooth</td>
<td>$\text{der(smooth)} = \frac{\text{input-smooth}}{\text{averagingTimeVariable}}$</td>
</tr>
<tr>
<td>flow</td>
<td>$\text{flow} = \text{inflow}$</td>
</tr>
</tbody>
</table>

Initial support for Algorithmic Modelica was added in order to simulate this model

➢ Ongoing work on expanding support for algorithmic Modelica

In order to achieve this a *SD to Modelica translator* was developed

- Mapping XMILE to Modelica
  - Oasis XML Interchange
- We used ESCIMO, which is a fairly complicated SD model, to validate the translator


Validation of the Translator

- The translator was validated by first simulating the results in Vensim and then simulating the same model in Modelica and in Julia.
- The simulation was run from 1850 to 2500.
- We then compared the results for each tenth-year to examine deviations from the original model.
- Three variables were examined in detail:
  - Temperature surface anomaly compared to 1850 (Celsius), that is, the difference in average global surface temperature compared to 1850.
  - pH in warm surface water, that is, the acidity of warm surface water.
  - CO2 Concentration in PPM, that is, the concentration of carbon dioxide in the atmosphere.

Solver Settings:
- Runge-Kutta-4 (Vensim)
- DASSL with absolute and relative tolerance of 1E-6 for Modelica
- Rodas5 with absolute and relative tolerance of 1E-6 for Julia

Practical Examples

Simulating models and profile memory

```plaintext
  Profile.clear()
  #= Precompile file to use
  println("# Testing a model")
  @time flattenModelInMML_TST();
  println("Running a second time")
  @time flattenModelInMML_TST(model);
  #= Try to flatten an engine model in the multibody library.
  println("Profile memory allocations of that model")
  Profile.Allocs.@profile bsample_rate=0.1 flattenModelInMML_TST(model);
  println("Running profiler")
  PProf.Allocs.pprof()
end
```

Simulating and Plotting

```plaintext
julia> begin
import OM
import CSV
import DataFrames
using Plots
sol = OM.simulate("HelloWorld", ".Models/HelloWorld.mo")
OM.exportCSV("modelName", sol; filePath = "filename.csv")
f = CSV.read("./filename.csv", DataFrame)
DataFrames.DataFrame(f)
plot(f.time, f.x)
end
```

![Graph](image)

LINKÖPING UNIVERSITY
Practical Examples Continued

Possible to simulate a model and export the result to OMEdit

OM.exportCSV("modelName", sol; filePath = "filename.csv")
Practical Examples using the Standard Library

```plaintext
model ElectricalComponentTestMSL
  // MSL Imports
  // Equation for the serial RLC circuit.
  model SimpleCircuit
    Resistor R1{R=30};
    Capacitor C{C=0.61};
    Resistor R2{R=100};
    Inductor L{L=0.3};
    SineVoltage AC{freqR2 = 1., phase = 1.};
    Ground 0;
  equation
    connect({C.p, R1.p}); // 1, Capacitor circuit
    connect({R1.n, C.p}); // Wire 2
    connect({C.n, AC.n}); // Wire 3
    connect({R1.p, R2.p}); // 2, Inductor circuit
    connect({R2.n, L.p}); // Wire 5
    connect({L.n, AC.n}); // Wire 6
    connect({C.n, 0.p}); // 1, Ground
  end SimpleCircuit;
end ElectricalComponentTestMSL;
```

```plaintext
julia> begin
import OM
import CSV
import DataFrames
using Plots

    sol = OM.simulate("ElectricalComponentTestMSL.SimpleCircuit",
                      "/MSL_USE/ElectricalComponentTest.mo"
                      
                      MSL = true, MSL_VERSION = "MSL:3.2.3"
                      
                      )

    OM.exportCSV("modelName", sol; filePath = "filename.csv")

    f = CSV.read("./filename.csv", DataFrames.DataFrame)

    plot(f.time, f.C_v)
end
```
Notes on Performance
Performance in OpenModelica.jl

• **Some History**
  • Translation performance has historically been low in the frontend
  • First version (Early 2020), correct but around ~ 1 hour to compile simple programs like HelloWorld...
  • Issues
    • Julia Type Inference: Julia historically struggled with type inference for mutually recursive data structures with deep recursion
      • Solved by introducing manually introducing barriers to type inference
      • Hindrance of fully automatic translation
    • OpenModelica relies on exceptions for control flow for operations such as lookup and error handling
      • Exceptions, while expensive, are comparable inexpensive in MetaModelica compared to Julia
  • Julia 1.5 - 1.6 ~ Gamechanger
    • Could compile model without inference issues
  • Julia 1.10
    • Faster Garbage Collector
  • More or less OpenModelica.jl has become faster and more memory efficient for each Julia Version since
    • Furthermore, various improvements to the runtime and runtime data structures has been made
    • Examples include statically generating less code when detecting a match expression that can not fail
  • Current version
    • Faster than omc translating some models...
For small models, the Julia variant of the frontend perform better

Conclusion is OM.jl better than OpenModelica?

Not really…
Memory Patterns for HelloWorld

model HelloWorld
  Real x(start = 1, fixed = true);
  parameter Real a = 1;
end HelloWorld;
equation
  der(x) = -a * x;
end HelloWorld;

Experience: For small models such as the HelloWorld model and others, OpenModelica.jl > OpenModelica

- Here, OM.jl uses less memory even during instantiation...
- As we will see next it also uses less memory for other operations...
  - However, the Julia Modelica Compiler is cheating....
  - Note, OM.jl requires less memory here than omc requires for just one phase,

OpenModelica:
Notification: Performance of NFInst.instantiate(HelloWorld): time 0.00136/0.001485, allocations: 366.4 kB / 14.4 MB, free: 220 kB / 13.93 MB
A rough comparison of memory and speed for the **Engine1a** model in the Multibody library.

Here, OpenModelica consumes more memory *in total* but is around 3 times faster.
- Certain phases are cheaper in OpenModelica
- **More on that later..**

For other model's similar patterns can be observed.

---

```julia
include("perf.jl");
[ Info: Loading MSL Version: MSL:4.0.0
  0.000003 seconds
[ Info: Loaded MSL successfully

Attempting to instantiate...Test.MBTest

7.353518 seconds (7.66 M allocations: 336.631 MiB, 4.91% gc time)
```

```
FrontEnd: time 2.415e-06/2.294, allocations: 0 / 402.6 MB, free: 14.48 MB / 331.4 MB
```
Reason: Issue with certain MetaModelica practices

• Not apparent at first glance, but due to the heavy recursion used for this function, the named arguments here create a significant number of allocations

• Removing the keyword arguments here saved 5% of memory
  ➢ Small models where lookup processing is a significant part of the translation

✓ Changing this in the omc could possibly save some memory as well
Challenges

High Memory Consumption in the Frontend for some Models

• Currently, the frontend consumes too much memory for operations, around 10X that of OpenModelica
  • Also, the OpenModelica Frontend has improved future in terms of memory efficiency since I started this work
• Memory patterns vary from model to model
• The backend (OMBackend.jl) could use some efficiency improvements during translation.
• Not yet 100% coverage of the Standard Library in the Frontend
  • Some issues with Fluids

The Silver Lining

• For small models, OM.jl typically consumes less memory than OM
  • No need to reparse libraries
  • Faster instantiations for some models
  • Libraries such as MSL are loaded upon compilation…..
  ✓ The translation performance of the frontend for the Julia Modelica Compiler is now decent at large
• While we do not fully cover the MSL. A significant amount of the Modelica Standard Library is handled by the frontend
• The MTK and the wider Julia ecosystem provide a wide array of solvers and libraries
  • Better simulation time than OpenModelica

¹For some models, and also via new language features such as DOCC and the THETA operator
Using OpenModelica + OM.jl

Flat Modelica Capabilities
- Using tools such as OMJulia it is possible to directly interface the rest of the OpenModelica environment
- For models currently not handled by the Julia OpenModelica Compiler it is possible to output flat Modelica and then feed to the OpenModelica.jl environment

Consequences
- One can imagine a combination of OMJulia.jl and OpenModelica.jl to use advanced libraries such as the Buildings Library within the Julia Environment
  - Backend can be reused with the frontend
Conclusions

- OM.jl is moving closer to an initial real release
- Due to improvements of, Julia I now believe that one can feasibly implement a full-fledged compiler in it that not only works but also has decent performance
  - Some details...
- OM.jl provides a way of working with the Modelica ecosystem in the Julia Environment
- The capabilities of Julia allow one to quickly implement and prototype new language features
References


Questions

Visualization of OpenModelica.jl by Chat GPT
Bonus Slide I did one for OpenModelica as well