OpenModelica.jl: A modular and extensible Modelica compiler framework in Julia targeting ModelingToolkit.jl

John Tinnerholm[1] Adrian Pop[1] Andreas Heuermann[2] Martin Sjölund[1]

[1]Department of Computer and Information Science, Linköping University, Sweden, {first.last}@liu.se
[2]Faculty of Engineering and Mathematics, Bielefeld University of Applied Sciences, Germany, {first.last}@fh-bielefeld.de

Abstract

This paper presents current work on our Modelica Compiler framework in Julia: OpenModelica.jl.[1] We provide a brief overview of this novel framework and its features, and we also present the latest addition to the possible backend options. We target ModelingToolkit.jl (MTK), a framework for symbolic-numerical computation and scientific machine learning. We evaluated the performance of our new backend using the ScalableTestsuite, a benchmark suite for Modelica Compilers. In our experiment, we demonstrate that MTK can be used as a backend with competitive simulation performance. In addition, using the scientific machine learning features of the Modeling toolkit, we were able to approximate models in the ScalableTestsuite using surrogate techniques and how such techniques can be used to accelerate the solving of non-linear algebraic loops during tearing.

Based on our experiments, we propose using this new framework to automatically generate surrogate components of a Modelica model during the simulation to increase performance. The experimental work presented here provides one of the first investigations concerning the integration of the symbolic-numerical abilities of Julia within a Modelica tool.

Keywords: Modelica, OpenModelica, Julia, Equation-based modeling, Compiler-construction

1 Introduction

The ability to model cyber-physical systems (CPS) is essential for many scientific and industrial processes. Modelica is a standardized declarative equation-based object-oriented language with a solid tool and library support. Recently, researchers have shown an increased interest in the Julia language (Bezanson et al. 2017), with the release of several packages that bring acausal modeling to Julia, such as Modia (Elmqvist and Otter 2017) and ModelingToolkit (MTK) (Ma et al. 2021).

Thus several studies have begun to examine the implications of using Julia as a foundation to design new modeling frameworks. In this paper, we present our contribution to this effort within the OpenModelica programming environment (Fritzson, Pop, Abdelhak, et al. 2020).

[1]On GitHub: OpenModelica/OpenModelica.jl

1.1 Motivation

The main motivation for the work presented here is that previous studies do not attempt to integrate Modelica within Julia. Instead, they provide the possibility of Modelica-like acausal modeling using Julia as a host language. Tinnerholm et al. (2020) presented our first Modelica compiler prototype in Julia. This compiler was developed with the goal to utilize Julia's symbolic-numerical capabilities and extend the current capabilities of Modelica. In this paper, we expand this work to implement a full Modelica compiler using Julia with the goal to improve and optimize existing models, and by adhering to the standards of the Modelica language, we hope to facilitate the reuse of modeling know-how contained in existing Modelica libraries. Updates to this first iteration of this compiler include automatic translation of the high-performance frontend (Pop et al. 2019) along with experimental support for hybrid systems and a new backend targeting ModelingToolkit. We have used this framework to simulate Modelica models of systems containing thousands of equations and variables to assess the performance of our compiler.

1.2 Contributions

While several Modelica Compilers have been designed before, no compiler has previously used Julia as an implementation language. Thus, a central contribution of this paper is evidence that such a compiler is both feasible and is easily extendable. We demonstrate this by using the symbolic-numerical capabilities of Julia and the scientific machine learning capabilities of MTK to automatically generate surrogate models within a modular and extendable pipeline. Another contribution of this paper is, to our knowledge, the first empirical investigation concerning the performance characteristics of ModelingToolkit when employed as a backend for a Modelica Compiler, demonstrating its claimed usefulness as a compiler component for equation-oriented languages.

1.3 Paper Organization

This paper is organized as follows: The background is presented in section 2, this is followed by section 3 where we present the structure of the compiler. In section 4 we recount how we verified the frontend together with some

current performance characteristics when flattening the cascaded first-order system from the ScalableTestsuite in Listing 4 along with a description of experiments to assess the correctness of the frontend. This section is followed by section 5 which provides a benchmark highlighting the current simulation performance of the new MTK backend and comparing it to the OpenModelica Compiler (OMC). This is followed by section 6 in which we demonstrate how the scientific machine learning features of MTK can be used to approximate models or subcomponents of models. In section 7 where we present our conclusions along with recommendations for future work.

2 Background

As stated in section 1 there exist as of 2021 several modeling environments that provide the option of causal and acausal modeling within the Julia ecosystem. *DifferentialEquations.jl* (Rackauckas and Nie 2017) is one such environment. It provides a seamless foreign function interface that allows interfacing algorithmic Julia code and a variety of different solvers. A user of DifferentialEquations.jl writes imperative code in the Julia language to conform to systems such as Nonlinear-systems, ODE-systems, and DAE-systems. Tinnerholm et al. (2020) selected DifferentialEquations.jl as the default backend target. A model of a hybrid system representing a bouncing ball using DifferentialEquations.jl can be studied in Listing 1.

While DifferentialEquations.jl provides the abstractions necessary to write causal models in Julia, it does not provide the abstractions of a full-fledged modeling language. *ModelingToolkit.jl* (MTK) aims address this issue (Ma et al. 2021). MTK is a recent modeling framework to automate symbolic operations common for equation-oriented languages, such as methods for index reduction. It does so by using the symbolic-numerical capabilities of Julia to preprocess an MTK model description into a format that can be solved using the set of solvers provided by DifferentialEquations.jl. In other words, the iterative process from an acausal description based on equations to a causal representation acceptable for a solver is similar to that of a typical Modelica Compiler. The language defined by MTK does not at the time of this writing support hybrid systems. However, it is possible to post-process MTK models to add events similar to Listing 1 where the Modelica *when-equation* is represented using a `ContinuousCallback` which is illustrated in Listing 2. If we compare the generated code in Listing 1 with that of Listing 2 we can see that MTK is closer to Modelica in the level of abstraction; however, MTK lacks control structures found in Modelica such as **`for`** and **`if`**.

MTK does not only target DAE-systems, it also targets several areas which are not the primary target of the Modelica language such as:

- Stochastic differential(-algebraic) equations
- Partial differential equations
- Optimization problems
- Continuous-Time Markov Chains
- Nonlinear Optimal Control

This enables users of MTK to combine different systems from different domains (Ma et al. 2021). Conceivably, model exchange between this framework and Modelica would be useful for efficient modeling and simulation of large dynamic systems.

The main difference between Modelica and the language defined by MTK is the level of abstraction. To give an example, as of this writing, ModelingToolkit.jl requires users to specify the application of index reduction explicitly; it also requires systems to be specified explicitly with the state derivatives on the right-hand side. Thus, the user specifies the transformation from a DAE-System into an ODE-system, whereas in a Modelica compiler, these decisions are generally abstracted away. Still, as we will illustrate in this paper, MTK is suitable as a backend framework for Modelica Compilers or other equation-oriented languages frameworks in Julia.

Modia.jl (Elmqvist and Otter 2017) is another framework that brings acausal modeling to Julia. Syntactically it is more similar to Modelica when compared to the language defined by MTK. However, it is different from the work presented here because its constructs are implemented using Julia metaprogramming rather than traditional data structures used by compilers.

Yet another modeling framework is *Causal.jl* (Sarı and Günel 2019). However, as the name implies, it is a causal modeling framework reminiscent of Simulink.

3 Compiler Structure

In this section, we will elaborate on the different components that make up OpenModelica.jl. To provide a brief overview of the size of this application, a summary of the current size of this compiler by lines of code (LOC) is provided in Table 1. For comparison, the OMC compiler has about 1,100,000 LOC MetaModelica code for the frontend+backend and about 67,000 LOC C code for the runtime system. Using Julia is clearly an advantage as one can delegate functionalities such as finding strongly connected components to existing Julia libraries.

The frontend is made up of OMParser and OMFrontend; the backend of OMBackend and the runtime of MetaModelica.jl. Internally three intermediate representations are used: Absyn[2], SCode[3] and DAE.[4] An overview of the compiler pipeline is presented in Figure 1. An example of how to simulate and plot a Modelica model in Julia is given in Listing 3.

[2]On GitHub: OpenModelica/Absyn.jl
[3]On GitHub: OpenModelica/SCode.jl
[4]On GitHub: OpenModelica/DAE.jl

DOI 10.3384/ecp21181109

Listing 1. Automatically generated Julia code for a simple hybrid system. The Julia code presented in this listing is targeting the IDA solver in Sundials (Hindmarsh et al. 2005) using DifferentialEquations.jl.

```
function BouncingBallRealsStartConditions(
    aux, t)
  local x = zeros(2)
  local dx = zeros(2)
  local p = aux[1]
  local reals = aux[2]
  reals[1] = 1.0
  dx[1] = reals[2]
  dx[2] = -(p[2])
  x[2] = reals[2]
  x[1] = reals[1]
  return (x, dx)
end
function BouncingBallRealsDifferentialVars
    ()
  return Bool[1, 1]
end
function BouncingBallRealsDAE_equations(res
    , dx, x, aux, t)
  local p = aux[1]
  local reals = aux[2]
  res[1] = dx[2] - -(p[2])
  res[2] = dx[1] - reals[2]
  reals[2] = x[2]
  reals[1] = x[1]
end
function BouncingBallRealsParameterVars()
  local aux = Array{Array{Float64}}(undef,
      2)
  local p = Array{Float64}(undef, 2)
  local reals = Array{Float64}(undef, 2)
  aux[1] = p
  aux[2] = reals
  p[2] = 9.81
  p[1] = 0.7
  return aux
end
saved_values_BouncingBallReals =
    SavedValues(Float64, Tuple{Float64,
    Array})
function BouncingBallRealsCallbackSet(aux)
  local p = aux[1]
  function condition1(x, t, integrator)
    x[1] - 0.0
  end
  function affect1!(integrator)
    integrator.u[2] = -(p[1] * integrator.u
        [2])
  end
  cb1 = ContinuousCallback(
         condition1,
          affect1!,
          rootfind = true,
          save_positions = (true, true),
          affect_neg! = affect1!,
        )
  savingFunction(u, t, integrator) =
    let
      (t, deepcopy(integrator.p[2]))
     end
  cb2 = SavingCallback(savingFunction,
      saved_values_BouncingBallReals)
  return CallbackSet(cb1, cb2)
end
```

Listing 2. An MTK version of the bouncing ball produced by the new backend.

```
using ModelingToolkit
using DiffEqBase
using DifferentialEquations
function BouncingBallRealsModel(tspan =
    (0.0, 1.0))
  pars = ModelingToolkit.@parameters(begin
      (e, g, t) end)
  vars = ModelingToolkit.@variables(begin (
      h(t), v(t)) end)
  der = Differential(t)
  eqs = [
    der(h) ~ v,
    der(v) ~ -g
  ]
  nonLinearSystem = ModelingToolkit.
      ODESystem(eqs, t, vars, pars,
    name = :($(Symbol("BouncingBallReals"))
        ),
  )
  pars = Dict(e => float(0.7), g => float
      (9.81), t => tspan[1])
  initialValues = [h => 1.0, v => 0.0]
  firstOrderSystem = ModelingToolkit.
      ode_order_lowering(nonLinearSystem)
  reducedSystem = ModelingToolkit.
      dae_index_lowering(firstOrderSystem)
  problem = ModelingToolkit.ODEProblem(
      reducedSystem, initialValues, tspan,
      pars)
  return problem
end
function BouncingBallRealsCallbackSet()
  function condition1(x, t, integrator)
    x[1] - 0.0
  end
  function affect1!(integrator)
    integrator.u[2] = -(integrator.p[1] *
        integrator.u[2])
  end
  cb1 = ContinuousCallback(
    condition1,
    affect1!,
    rootfind = true,
    save_positions = (true, true),
    affect_neg! = affect1!,
  )
  return CallbackSet(cb1)
end
BouncingBallRealsModel_problem =
    BouncingBallRealsModel()
function BouncingBallRealsSimulate(tspan =
    (0.0, 1.0))
  solve(BouncingBallRealsModel_problem,
      tspan = tspan, callback =
      BouncingBallRealsCallbackSet())
end
function BouncingBallRealsSimulate(tspan =
    (0.0, 1.0); solver = Tsit5())
  solve(BouncingBallRealsModel_problem,
      tspan = tspan, solver)
end
```

Listing 3. This listing illustrates how to export and simulate a Modelica model in Julia. Once preprocessed by the frontend the call to *OMBackend.translate* stores the final MTK program for future analysis or simulation. The input parameter modelName provided by the caller and it is assumed that the file containing a Modelica model is named after this parameter.

```
function runTest(modelName)
  #= OMParser phase =#
  ast::Absyn.Program = OMFrontend.parseFile
      ("./Models/$(modelName).mo")
  #= OMFrontend phases =#
  scodeProgram::SCode.Program = OMFrontend.
      translateToSCode(ast)
  (dae, cache) = OMFrontend.
      instantiateSCodeToDAE(modelName,
      scodeProgram)
  #= Backend phases =#
  OMBackend.translate(dae; BackendMode =
      OMBackend.MODELING_TOOLKIT_MODE)
  res = OMBackend.simulateModel(modelName,
      tspan = (0.0, 1.0))
  #= Optionally plot the result =#
  OMBackend.plot(res)
end
```

Table 1. Current sizes of OpenModelica.jl compiler phases by LOC.

Compiler Phase	*Lines*
Runtime	1,853
FrontEnd	135,103
BackEnd & Code generation	6,073
Total size	*143,029*

3.1 OMParser.jl

The parser, OpenModelicaParser.jl was adapted from the existing OMC parser.[5] which is written in ANTLR (Parr and Quong 1995) It is currently capable of parsing large files such as the entire Modelica Standard Library.[6] After a successful parse, the AST can be fed to the frontend module, *OMFrontend.jl* the data structure representing the AST is defined by *Absyn.jl*.[2] In Listing 3 this parser is invoked by `OMFrontend.parseFile("./Models/$(modelName).mo")`.

3.2 OMFrontend.jl

To flatten the Modelica code, we use the OMFrontend.jl, which was automatically generated from the high-performance frontend of the OMC (Pop et al. 2019). Previously, we used the old frontend (Tinnerholm et al. 2020); however, as part of the work presented here, the *MetaModelica-Julia translator* introduced by Tinnerholm et al. (2020) was used to automatically generate a Julia implementation of the high-performance frontend (Pop et al. 2019). Consequently, the frontend is implemented in automatically generated Julia code generated by translating the existing OpenModelica frontend. While the translation of the old frontend[7] was achieved without any major modifications, we had to manually resolve cases of mutually circular module dependencies for the new since Julia does not handle them while MetaModelica does.

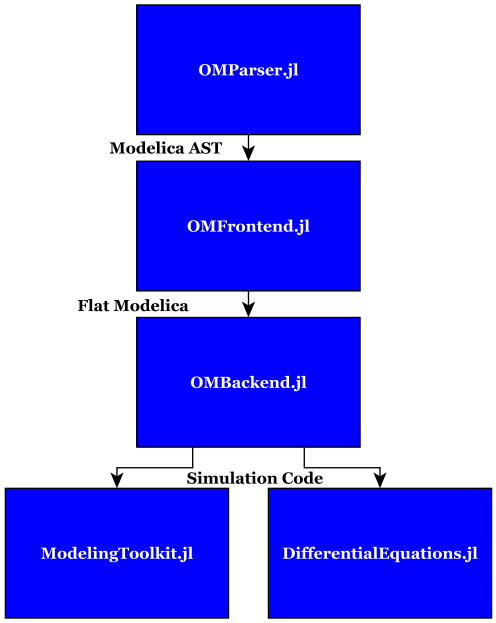

Figure 1. An illustration of the compiler pipeline including current available backends targeting ModelingToolkit.jl and DifferentialEquations.jl. The Modelica AST is represented using Absyn.jl. Inside OMFrontend.jl the SCode representation defined by SCode.jl is used. Flat Modelica is encoded using DAE.jl

3.3 OMBackend.jl

OMBackend is the backend module of this compiler, and it is implemented as a separate package. Current backend targets include both ModelingToolkit and DifferentialEquations.jl. Simulations targeting DifferentialEquations.jl use the Sundials IDA solver and is based on the DAE-mode implementation by W. Braun, Casella, Bachmann, et al. (2017). It currently provides support for continuous systems and experimental support for hybrid systems. An example of code generated for a hybrid system is the bouncing ball model, see Listing 1.

3.3.1 The MTK-Backend

The new backend based on ModelingToolkit (MTK-Backend) is capable of automatically translating Modelica models into equivalent MTK models. The MTK backend works by accepting the flat Modelica/Hybrid DAE that is described by *DAE.jl*. Since MTK automatically handles transformations such as index reduction, no such algorithm is applied by the backend. MTK is also acausal in

[5] On GitHub: adrpo/OpenModelicaParser.jl

[6] Modelica Standard Library (Version 3.2.1)
On GitHub: modelica/ModelicaStandardLibrary

[7] The old frontend is the frontend the *high-performance frontend* replaced (Pop et al. 2019).

DOI 10.3384/ecp21181109

the sense that there need not to be a causal order between the equations. However, MTK requires the system to be in explicit form, so the derivatives are reordered using MTK's symbolic algebra routines. An MTK translation of the Modelica code in Listing 4 can be studied in Listing 5. In Listing 5 the function Casc10Model defines the model, the parameters are defined using *parameters* and the variables are defined using *variables*. The statement *ode_order_lowering* transforms the system to first-order form and *dae_index_lowering* performs index reduction[8].

3.4 MetaModelica.jl

MetaModelica.jl[9] provides a compatibility layer between Julia and MetaModelica (Fritzson, Pop, and Aronsson 2005). It reimplements several constructs of MetaModelica such as *match* and *matchcontinue*. Furthermore, MetaModelica.jl replicates the existing runtime of OMC. This package is used extensively in the translated modules.

4 Verification

The compiler presented in this paper consists of several components, see Figure 1 where each component contains thousands of LOC, see Table 1.

To verify the parser, we parsed the Modelica Standard Library, along with some other models, some of which contained errors. This was done to establish that it reported the same errors as the existing parser in OpenModelica.[10] Verifying the correctness of the frontend was more difficult since it consists of over 130,000 lines of automatically generated Julia code. We tested our implementation by lowering the hybrid DAE produced by the frontend and compared the result of simulating these models with the simulation results of OpenModelica. One excerpt of the verification experiments can be studied in Figure 2. The runnable code is available in listing 5. The corresponding Modelica model is available in Listing 4. The results are by no means exhaustive, but our preliminary verification experiments suggest that the translated frontend behaves correctly for the set of models that we tested.

5 Simulation Performance

The previous section has shown that we can generate and simulate Modelica models targeting DifferentialEquations.jl and MTK directly. In this section, we present the current simulation performance of the new MTK backend using the cascaded first-order system from the ScalableTestsuite (Casella 2015) and how the simulation performance of the MTK backend compares to OpenModelica. Two model from the benchmark suite where used *CascadedFirstOrder*, see Listing 4. The parameters of these were modified to increase the number of equations and events gradually.

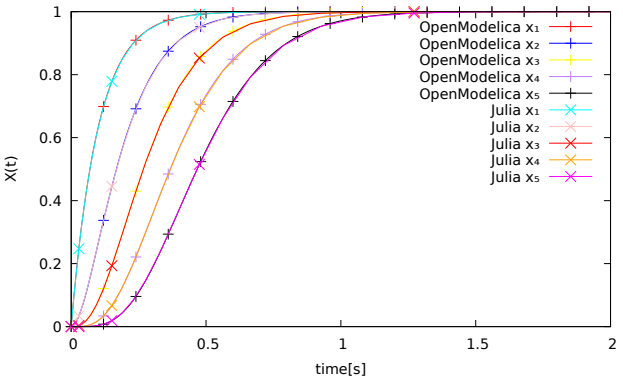

Figure 2. The result of simulating the cascading first-order system between 0.0 to 2.0 seconds, with $N = 10$ simulated using the new MTK backend and OMC. The plot shows $x_1, ..., x_5$ of $X(t)$. Since the curves are almost identical, markers are added to differentiate between the MTK and OpenModelica solutions.

5.1 Experimental setup

In this subsection, we describe the experimental setup of the model included our experiments.

To assess simulation performance of *CascadedFirstOrder* we generated code using the model in Listing 4 with the following values for the parameter N: $10, 100, 200, 400, 800, 1600, 3200, 6400, 12800, 25600$. The resulting model with $N = 10$ can be studied in Listing 5. However, due to long compilation times of MTK when $N > 3200$ we limited[11] our benchmark to the following values for N: $10, 100, 200, 400, 800, 1600, 3200$. We simulated models using ModelingToolkit v5.16.0 with Julia 1.6.1 and OpenModelica 1.18.0. All tests were run on a 16-core AMD TR 1950X with 128GB of RAM. When measuring simulation times for OMC and for Julia, we used Benchmarktools.jl (Chen and Revels 2016) Two experiments were run for the *CascadedFirstOrder* model.

For the first experiment we used the following solvers:

- **OMC** Sundials IDA solver (Hindmarsh et al. 2005)
- **MTK** The default solver,[12] invoked when calling solve without auxiliary arguments

In the second experiment, we selected Tsit5 (Tsitouras 2011) as the solver for MTK. For OpenModelica, we kept the IDA solver since, at the time of writing, the Tsit5-solver is not available within the OpenModelica environment.

5.2 Evaluating simulation performance

In Figure 3 we present the result of our first experiment. From the graph, we can see that the simulation time of

[8] Index reduction and the lowering of the ODE is not always necessary. For instance, an index 1 DAE does not need to have its' index reduced. Still, for generality, we do this for all systems.

[9] On GitHub: OpenModelica/MetaModelica.jl

[10] A subset of these tests are available on GitHub: adrpo/OpenModelicaParser.jl/test

[11] The MTK-models where $N > 3200$ are available on request.

[12] The default solver is selected automatically by DifferentialEquations.jl depending on the characteristics of the model (Rackauckas and Nie 2019). See solver selection algorithm.

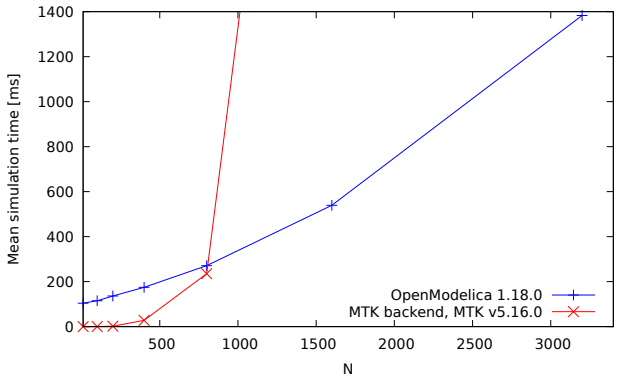

Figure 3. The mean simulation time of the cascading first-order system, with $N = 10, 100, 200, 400, 800, 1600, 3200$ simulated using the new MTK backend and OMC.

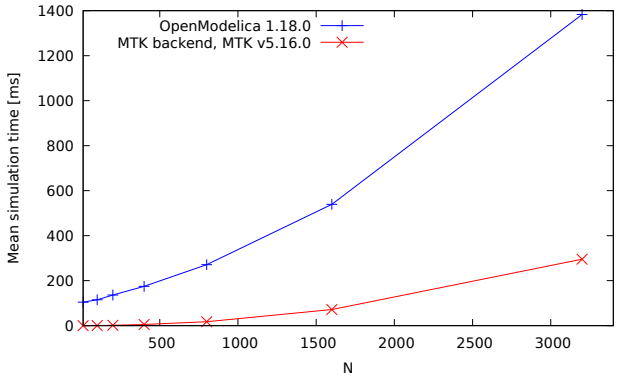

Figure 4. Same simulation as in Figure 3 using Tsit5 together with MTK.

MTK is superior to the OMC until $N > 400$, then the simulation time of MTK increases exponentially. This is probably due to conservative solver selection by MTK. This behavior was also observed when we compiled the automatically generated MTK code, where we observed an exponential increase in compilation time when $N > 3200$. Because of this, we omitted these models from the experiments. The result of the second experiment can be studied in Figure 4, it demonstrates superior simulation performance of MTK when compared to OMC when using an alternative solver, Tsit5 (Tsitouras 2011). While this is due to Tsit5 being more efficient for non-stiff problems in comparison to the IDA solver of OpenModelica 1.18.0, these results indicate that MTK can compete with OMC in terms of simulation performance. Still, while we were able to generate MTK code for up to 25,600 equations and variables, we experienced an exponential increase in compilation time when $N > 3200$ the reason for this behavior seems to be performance issues during the symbolic transformations of MTK v5.16.0.

6 Surrogate-based Optimization

The use of surrogates/metamodeling to accelerate computationally heavy models is not new. The idea is to replace computationally expensive components of a model with a surrogate model/metamodel to reduce computation cost and consequently reducing the feedback loop for modelers (Wang and Shan 2006).

Listing 4. The Cascaded first order system from the scalable testsuite (Casella 2015).

```
model CascadedFirstOrder
  "N cascaded first order systems,
      approximating a pure delay"
  parameter Integer N = 10 "Order of the
      system";
  parameter Real T = 1 "System delay";
  final parameter Real tau = T/N "
      Individual time constant";
  Real x[N] (each start = 0, each fixed =
      true);
equation
  tau*der(x[1]) = 1 - x[1];
  for i in 2:N loop
    tau*der(x[i]) = x[i-1] - x[i];
  end for;
end CascadedFirstOrder;
```

In the context of Julia, Yingbo et al. have previously shown how to accelerate models by employing surrogates using MTK, where they employed surrogates to accelerate a Heating, ventilation, and air conditioning (HVAC) model claiming a 590X speedup compared to Dymola (Ma et al. 2021). This suggests that the ability to generate surrogates in a Modelica Compiler is a useful feature for users.

In this paper, we introduced a new backend in our Julia-based Modelica Compiler, the MTK backend. Consequently, it is able to use the surrogate facilities of MTK. In Listing 6 we illustrate how a surrogate can be generated from one of the models used in section 5 with *Surrogates.jl*[13]. In this example, we translate a Modelica model into an equivalent MTK model. We then create a radial surrogate of this model based on 30 samples, and the resulting simulation can be seen in Figure 5.

It is possible to use other software to generate surrogates based on Modelica models within a Julia environment. However, a novelty of the work presented here is the ability to generate surrogates for internal equations during compilation time. One application is employing surrogates for computational expensive external functions or (non)linear loops. Another feature could be to introduce a Modelica annotation `Surrogatize` to indicate to the compiler to automatically replace that component with a suitable surrogate.

6.1 Employing surrogates in the context of solving nonlinear systems of equations

In this subsection, we demonstrate how such a nonlinear algebraic loop can be replaced with a surrogate.

[13] On Github:On GitHub: SciML/Surrogates.jl

DOI 10.3384/ecp21181109

Listing 5. Automatically generated MTK version of the Modelica code in Listing 4.

```
using ModelingToolkit
using DiffEqBase
using DifferentialEquations
function Casc10Model(tspan = (0.0, 1.0))
  pars = @parameters((T, tau, N, t))
  vars = @variables((x7(t),
                     x1(t),
                     x10(t),
                     x3(t),
                     x2(t),
                     x8(t),
                     x9(t),
                     x4(t),
                     x5(t),
                     x6(t)))
  der = Differential(t)
  eqs = [der(x7) ~ (tau^-1)*(x6 - x7), der(
      x1) ~ (tau^-1)*(1.0 - x1),
          der(x10) ~ (tau^-1)*(x9 - x10),
              der(x3) ~ (tau^-1)*(x2 - x3),
          der(x2) ~ (tau^-1)*(x1 - x2), der(
              x8) ~ (tau^-1)*(x7 - x8),
          der(x9) ~ (tau^-1)*(x8 - x9), der(
              x4) ~ (tau^-1)*(x3 - x4),
          der(x5) ~ (tau^-1)*(x4 - x5), der(
              x6) ~ (tau^-1)*(x5 - x6)]
  nonLinearSystem = ODESystem(eqs, t, vars,
                              pars,
                              name = :($(
                                  Symbol("
                                  Casc10"))
                                  ))
  pars = Dict(T => float(1.0),
              tau => float(T / float(N)),
              N => float(10), t => tspan
                  [1])
  initialValues = [x7 => 0.0, x1 => 0.0,
                   x10 => 0.0, x3 => 0.0,
                   x2 => 0.0, x8 => 0.0,
                   x9 => 0.0, x4 => 0.0,
                   x5 => 0.0, x6 => 0.0]
  firstOrderSystem = ode_order_lowering(
      nonLinearSystem)
  reducedSystem = dae_index_lowering(
      firstOrderSystem)
  problem = ODEProblem(reducedSystem,
      initialValues, tspan, pars)
  return problem
end
Casc10Model_problem = Casc10Model()
function Casc10Simulate(tspan = (0.0, 1.0))
  solve(Casc10Model_problem, tspan = tspan)
end
function Casc10Simulate(tspan = (0.0, 1.0);
    solver = Tsit5())
  solve(Casc10Model_problem, tspan = tspan,
      solver)
end
```

Listing 6. An example on how to automatically create a surrogate for Casc400 via a Julia script.

```
modelName = "Casc400"
n_sample = 30
surrogateFunction = (x, y, startTime,
    stopTime) -> Surrogates.RadialBasis(x,y
    ,startTime,stopTime)
#= Use backend target =#
ast = OMFrontend.parseFile("./Models/$(
    modelName).mo")
scodeProgram = OMFrontend.translateToSCode(
    ast)
(dae, cache) = OMFrontend.
    instantiateSCodeToDAE(modelName,
    scodeProgram)
OMBackend.translate(dae; BackendMode =
    OMBackend.MODELING_TOOLKIT_MODE)
#= Run Modelica model =#
omResult = OMBackend.simulateModel(
    modelName, tspan = (0.0, 1.0))
solution = getSolution(omResult)
#= Create surrogates for all states =#
x = sample(n_sample, startTime, stopTime,
    SobolSample())
y = omResult.(x)
surrogates = populateSurrogateArray()
#= Evaluate surrogates =#
surrogateResult = Array{Float64}(undef,
    modelN, length(omResult.u))
for i = 1:length(stateVars)
  surrogateResult[i,:] = surrogates[i].(
      omResult.t)
end
```

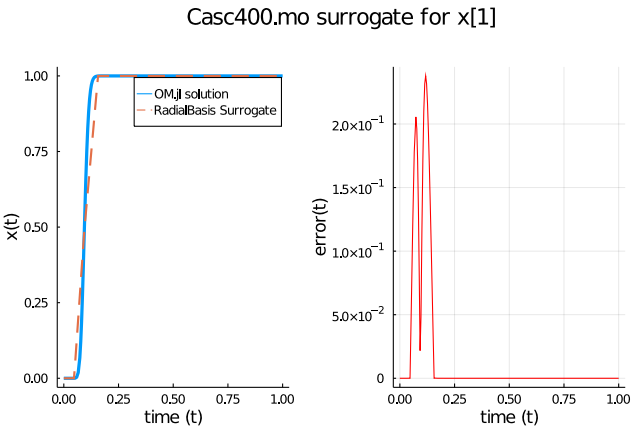

Figure 5. The graph to the left depicts the result of evaluating a Radial surrogate generated for the CascadedFirstOrder model with $N = 400$. The true function is represented in blue, the result of evaluating the surrogate is depicted in red. The right graph depicts the error as a function of time.

Listing 7. Modelica model where large parts of the simulation time are needed to solve a nonlinear loop.

```
model nonLinearScalable
  parameter Real a = 0.5;
  parameter Integer N = 10;
  Real x[N](each start=2.5);
  Real y(start=0,fixed=true);
equation
  for i in 1:N loop
    N+1 = exp(time*i*a+x[i]) + sum(x);
  end for;
  der(y) = sum(x)*time;
end nonLinearScalable;
```

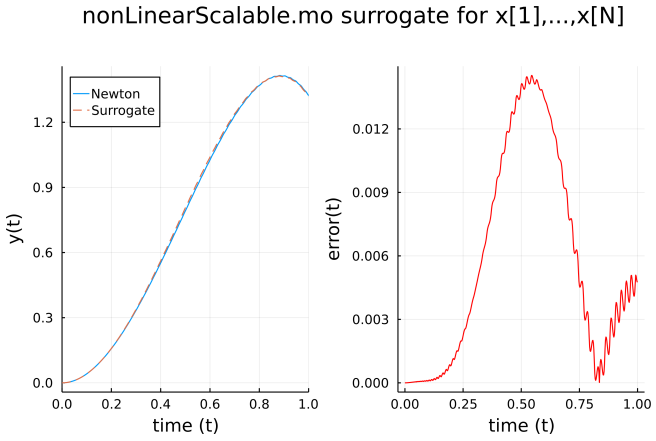

Figure 6. Simulation of nonLinearScalable using OMBackend and our surrogate function. The error is shown in red.

For the Modelica model nonLinearScalable, displayed in Listing 7, there is a dense nonlinear loop in variables $x_1, \ldots, x_N$, that needs to be solved in each integration step to calculate the state variable y. The system is scalable in N.

During compilation, the loop is detected, and we collect all equations and variables involved in the loop to generate a function to solve the algebraic loop. For this example, we manually generated training data by solving the loop at time points in $[0, 1]$ with a Newton method. Then a small neural network was trained on these solutions to replace the Newton solver normally used to solve the nonlinear algebraic system in the function *nonLinearScalableModel_surrogateODE*.

The speedup of the simulation time with the surrogate compared to simulating with a nonlinear solver that uses the Newton method was approximately 163.2 while the memory consumption was reduced from around 10.6 MiB to 0.4 MiB.

The solution of state variable y of `nonLinearScalable` where the nonlinear equation system is solved with Newton's method and with a surrogate can be studied in Figure 6. Because a simple structure for the neural network was chosen, some accuracy was lost while speeding up the simulation significantly.

To conclude, this integration enables the possibility of

Listing 8. The generated Julia code corresponding to listing 7. Some equations in the nonlinear system are left but are available upon request.

```
function nonLinearScalableAlgebraicLoop()
  parameters = ModelingToolkit.@parameters
      ((a, N, t))
  vars = ModelingToolkit.@variables((y(t),
      x₁, x₂, x₃, x₄, x₅, x₆, x₇, x₈, x₉, x
      ₁₀))
  eqs = [
        0 ~ 0.0 - (((t * a + x₁) + (x₁₀ +
            (x₉ + (x₈ + (x₇ + (x₆ + (x₅ +
            (x₄ + (x₃ + (x₂ + x₁)))))))))
             - float(N + 1)),
        ....]
  nonLinearSystem = ModelingToolkit.
      NonlinearSystem(eqs, vars, parameters
      , name = :($(Symbol("
      nonLinearScalable"))))
  return nonLinearSystem
  end
function makeNLProblem()
  loop = nonLinearScalableAlgebraicLoop()
  nlsys_function = (generate_function(loop,
       expression = Val{false}))[2]
end

nonLinearScalableNonLinearFunction =
    makeNLProblem()
function nonLinearScalableModel_ODE(dx, x,
    aux, t)
  p = aux[1]
  u = aux[2]
  func!(res, u) =
      nonLinearScalableNonLinearFunction(
      res, u, vcat(p, [t]))
  sol = nlsolve(func!, u, ftol = 1.0e-12;
      method = :newton)
  aux[2] = sol.zero
  dx[1] = (u[8] + u[9] + u[7] + u[6] + u[5]
       + u[4] + u[3] + u[2] + u[11] + u
      [10]) * t
end

function
    nonLinearScalableModel_surrogateODE(dx,
     x, aux, t)
  u = aux[2]
  aux[2] = m([t])
  dx[1] = (u[8] + u[9] + u[7] + u[6] + u[5]
       + u[4] + u[3] + u[2] + u[11] + u
      [10]) * t
end
```


DOI 10.3384/ecp21181109

auto-tuning Modelica models or parts of such models to accelerate simulation time in a fashion similar to that of Ma et al. (2021). This trade-off between accuracy and simulation time may be suitable for industrial applications, e.g. automatically reducing the high detailed development model to a real-time capable surrogate running on an integrated chip.

7 Conclusion and Future Work

The results of this paper indicate the feasibility of a Modelica Compiler in Julia. Concerning the experiments in section 5, we acknowledge that there is a wide range of different solvers available in MTK and in OpenModelica. Thus, the goal of these experiments is not to assess the performance of handwritten MTK-models but rather to characterize the performance of automatically generated MTK-models in the context of the compiler presented here.

Furthermore, having access to the ModelingToolkit ecosystem enables several extensions. One such extension is the possibility to generate surrogates during compilation time automatically. Challenges for such a scheme include not only the selection of surrogatisation techniques but also how to decide what part of a model to replace and what kind of surrogate to employ.

Another direction for future work would be to examine further the application of surrogatisation techniques in the context of algebraic loops or whole sub-models. That is replacing algebraic loops in large industrial grade DAE systems with suitable surrogates during compilation time. While we presented some initial examples as a part of this paper, further work is required to establish the efficiency of such techniques. To conclude, in this paper, we present OpenModelica.jl, a non-monolithic Modelica Compiler written in Julia using the high-performance frontend from the OMC that can connect the Modelica ecosystem with the ecosystem of Julia and ModelingToolkit.

Acknowledgements

This work has been supported by SSF in the LARGEDYN project. This work has also been supported by Vinnova in the ITEA EMPHYSIS project and the EMISYS project. Support from the Swedish Government has also been received through the ELLIIT project. OpenModelica development is supported by the Open Source Modelica Consortium. Many students, researchers, and engineers have contributed to the OpenModelica environment. There is not enough room here to mention all these people, but we gratefully acknowledge their contributions. We would also like to thank the reviewers for their comments to improve this paper.

References

Bezanson, Jeff et al. (2017). "Julia: A fresh approach to numerical computing". In: *SIAM review* 59.1, pp. 65–98.

Braun, Willi, Francesco Casella, Bernhard Bachmann, et al. (2017). "Solving large-scale Modelica models: new approaches and experimental results using OpenModelica". In: *12 International Modelica Conference*. Linkoping University Electronic Press, pp. 557–563.

Casella, Francesco (2015). "Simulation of large-scale models in modelica: State of the art and future perspectives". In: *11th International Modelica Conference*, pp. 459–468.

Chen, Jiahao and Jarrett Revels (2016-08). "Robust benchmarking in noisy environments". In: *arXiv e-prints*, arXiv:1608.04295. arXiv: 1608.04295 [cs.PF].

Elmqvist, Hilding and Martin Otter (2017). "Innovations for future Modelica". In: *Proceedings of 12th International Modelica Conference*. Linköping University Electronic Press.

Fritzson, Peter, Adrian Pop, Karim Abdelhak, et al. (2020). "The OpenModelica Integrated Environment for Modeling, Simulation, and Model-Based Development". In: *Modeling, Identification and Control* 41.4, pp. 241–295.

Fritzson, Peter, Adrian Pop, and Peter Aronsson (2005). "Towards Comprehensive Meta-Modeling and Meta-Programming Capabilities in Modelica". In: *Proceedings of the 4th International Modelica Conference, Hamburg, Germany*. Citeseer.

Hindmarsh, Alan C et al. (2005). "SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers". In: *ACM Transactions on Mathematical Software (TOMS)* 31.3, pp. 363–396.

Ma, Yingbo et al. (2021). *ModelingToolkit: A Composable Graph Transformation System For Equation-Based Modeling*. arXiv: 2103.05244 [cs.MS].

Parr, Terence J. and Russell W. Quong (1995). "ANTLR: A predicated-LL (k) parser generator". In: *Software: Practice and Experience* 25.7, pp. 789–810.

Pop, Adrian et al. (2019). "A New OpenModelica Compiler High Performance Frontend". In: *13th International Modelica Conference*. Vol. 157, pp. 689–698.

Rackauckas, Christopher and Qing Nie (2017). "Differentialequations.jl–a performant and feature-rich ecosystem for solving differential equations in julia". In: *Journal of Open Research Software* 5.1.

Rackauckas, Christopher and Qing Nie (2019). "Confederated modular differential equation APIs for accelerated algorithm development and benchmarking". In: *Advances in Engineering Software* 132, pp. 1–6.

Sarı, Zekeriya and Serkan Günel (2019). "Causal. jl: A Modeling and Simulation Framework for Causal Models". In: *Proceedings of JuliaCon* 1, p. 1.

Tinnerholm, John et al. (2020). "Towards an Open-Source Modelica Compiler in Julia". In: *Proceedings of Asian Modelica Conference 2020, Tokyo, Japan, October 08-09, 2020*, pp. 143–151. DOI: 10.3384/ecp20174143.

Tsitouras, Ch (2011). "Runge–Kutta pairs of order 5 (4) satisfying only the first column simplifying assumption". In: *Computers & Mathematics with Applications* 62.2, pp. 770–775.

Wang, G Gary and S Shan (2006). "Review of metamodeling techniques in support of engineering design optimization". In: *International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*. Vol. 4255, pp. 415–426.