electronics

MDPI

Article

A Modular, Extensible, and Modelica-Standard-Compliant OpenModelica Compiler Framework in Julia Supporting Structural Variability

John Tinnerholm *, Adrian Pop and Martin Sjölund

Department of Computer and Information Science, Linköping University, SE-581 83 Linköping, Sweden; adrian.pop@liu.se (A.P.); martin.sjolund@liu.se (M.S.)
* Correspondence: john.tinnerholm@liu.se

Abstract: Nowadays, industrial products are getting increasingly complex, and time-to-market is significantly shorter. Modeling and simulation tools for cyber-physical systems need to keep up with the increased complexity. This paper presents OpenModelica.jl, a modular and extensible Modelica compiler framework in Julia targeting ModelingToolkit.jl and supporting Variable Structured Systems. We extended the Modelica language with three new operators to support continuous-time mode-switching and reconfiguration via recompilation at runtime. Therefore, our compiler supports the Modelica language and variable structure systems via the aforementioned extensions. To our knowledge, there are no other Modelica tools available that support both standard Modelica and variable structure systems. We evaluated our framework using a standardized benchmark suite, in terms of simulation, compilation and recompilation performance. The results concerning compilation and simulation time performance were compared with the results of running the existing OpenModelica compiler with the same set of models. A custom benchmark was devised to estimate the cost in terms of recompilation when simulating variable structure systems. The performance experiments showed that OpenModelica.jl is currently about four times slower in terms of compilation time when compiling a transmission line model with tens of thousands of equations and variables. The difference in simulation performance between the two compilers was negligable. Furthermore, the impact of recompilation during the simulation was usually small compared with the simulation time for long simulations. The results are promising for a prototype, and we outline approaches to further improve both compilation and simulation performance as future research.

Keywords: modeling and simulation; Modelica; Julia; multi-mode; variable structure systems; JIT

Citation: Tinnerholm, J.; Pop, A.; Sjölund, M. A Modular, Extensible, and Modelica-Standard-Compliant OpenModelica Compiler Framework in Julia Supporting Structural Variability. *Electronics* **2022**, *11*, 1772. https://doi.org/10.3390/electronics11111772

Academic Editor: Luis Gomes

Received: 10 May 2022
Accepted: 29 May 2022
Published: 2 June 2022

1. Introduction

Modeling cyber-physical systems (CPS) is important in many scientific and industrial processes. Modelica is a standardized, declarative, equation-based, object-oriented language with mature tool and library support for modeling and simulating systems. Recently, researchers have shown increasing interest in the Julia language [1]. This has led to the development of several domain-specific equation-based languages and frameworks to bring acausal modeling to Julia. These frameworks include Modia [2], ModelingToolkit (MTK) [3] and more. In this paper, we present our contribution to this effort within the OpenModelica modeling and simulation environment [4].

The motivation behind our work is that previous studies have not attempted to integrate Modelica within Julia. Instead, they provide the possibility of Modelica-like acausal modeling based on Julia metaprogramming, using Julia as a host language. To the best of our knowledge, no previous study has attempted to construct a full compiler for the equation-based language Modelica using Julia. In [5], we presented our first Modelica compiler prototype in Julia, and in [6] we presented our new backend targeting MTK. This compiler was developed to utilize Julia's symbolic-numerical capabilities and extend

the current capabilities of Modelica. In this text, we expand on our previous work to implement a full Modelica compiler to improve and optimize existing models and adhere to the Modelica specification standard. In this way, we facilitate the reuse of modeling know-how in the existing Modelica libraries while also extending the language with new functionality. Improvements to the first version include the automatic translation of the high-performance OpenModelica frontend [7], along with experimental support for hybrid systems and a new backend targeting MTK.

Furthermore, we are examining existing research frameworks to integrate support for so-called variable structure systems (VSS), i.e., systems that can reconfigure themselves during simulation. A Modelica framework with VSS Support can be utilized in several other areas. Applications include:

- Model reduction [8–10], which can speed up simulations by switching between a more or less complex model depending on some conditions.
- Grey-box modeling [11–13], where surrogate models are trained and employed before or during runtime to speed up simulation.
- Fault modeling [14,15], where faults can be introduced by changing the structure of a model during simulation, such that the result can be easily observed.
- Impulse handling [16–18] to properly handle ideal system components, such as ideal diodes and ideal clutches.

More details on applications are given in Sections 3, 4 and 9. In this article, we suggest and implement several extensions to the Modelica language for two classes of VSS, *explicit* VSS and *implicit* VSS.

To assess the performance in terms of both compilation and simulation efficiency, we used *OpenModelica.jl* to simulate Modelica models with thousands of equations and variables. We performed the experiments using both standard Modelica models from the ScalableTestSuite [19] and Modelica models using our extensions for variable structure.

This article is an extension of a previous paper [6]. The following extensions are presented:

- Expanded the description of the compiler design.
- Introduced initial support for handling variable structure systems.
- Improved the frontend to support a subset of the MSL.
- Expanded the performance experiments to not just include simulation time performance, but also compile time performance using a standard benchmark.
- Provided a performance experiment concerning VSS simulation.

The results regarding automatic surrogatization of algebraic loops are described in [6]. (Surrogatization means replacing a concrete model or in this case parts of model with a surrogate generated by a machine-learning algorithm [13]).

The remainder of this article is organized as follows. Section 2 presents the motivation and the research aim. Section 3 introduces the general background. Section 4 provides related research concerning VSS in the context of equation-based languages. Section 5 introduces our proposed compiler framework, and Section 6 describes our proposed language extensions. Materials and methods and the results are presented in Sections 7 and 8, respectively. Finally, we discuss the implications of our work in Section 9.

2. Motivation and Research Aim

Investigating variable structure systems (VSS) is a continuing concern within the research area of equation-based languages and numerical simulation [6,18,20]. However, support is yet to be standardized in an unified equation-based language such as Modelica. We believe that the reason for this is that other attempts, while successful, have not leveraged the functionality in existing compilers, preventing mainstream adaptation. In this paper, we present OpenModelica.jl, a compiler framework in Julia supporting the Modelica language, which we extended with new functionality to investigate VSS integration. We

roughly followed the principles of a design study [21]. Hence, by the means of empirical experiments and observations, we set out to answer the following research questions:

- RQ-1. *Can a Modelica compiler written in the Julia language have reasonable performance?*
- RQ-2. *How can the Modelica language be extended to simulate variable structured systems?*
- RQ-3. *What are possible advantages when blurring the line between compilation, modeling and simulation?*

To answer the first research question (RQ-1), we designed and implemented a modular compiler framework and compared it to an existing state-of-the-art compiler, using a standardized benchmark suite [19]. To answer the other two research questions we proposed, designed and implemented extensions to the Modelica language.

Since this is to the best of our knowledge the first dynamic compiler with the explicit aim of supporting the Modelica language fully [22], we used a custom microbenchmark to provide answers to *RQ-2* and *RQ-3*. The research framework is presented in Section 5, the extensions in Section 6 and the experiments in Section 8.

3. Background

We present background and related work by introducing the Modelica language in Section 3.1, MetaModelica in Section 3.2 and the Julia language in the context of equation-based programming languages in Section 3.3.

3.1. Modelica Language and Modelica Tools

Modelica is a standardized, declarative, equation-based, object-oriented language. Using Modelica, the modeler may write domain-independent system models using equations; a Modelica compiler then derives executable code from the equations to simulate the system. Figure 1 depicts an RLC circuit. This circuit can be modeled using Modelica as shown in Listing 1.

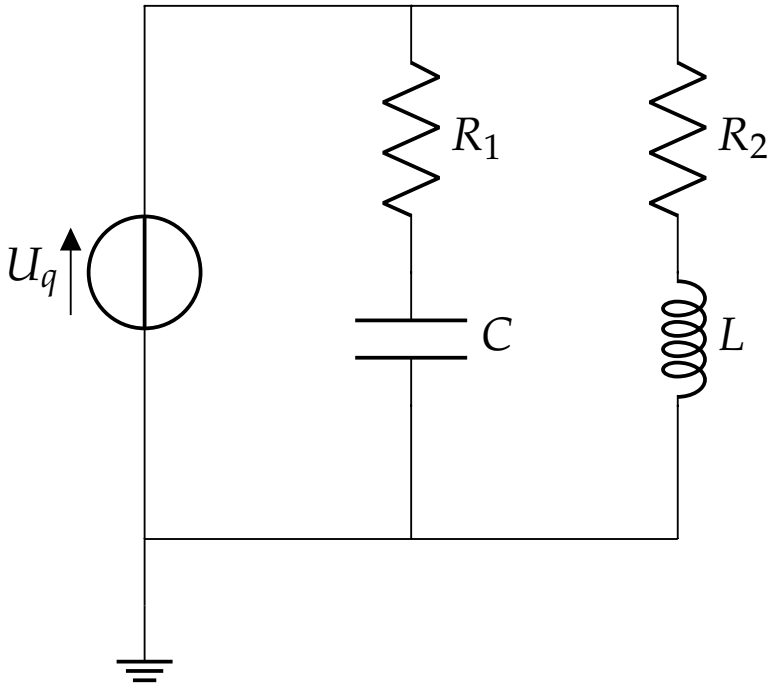

Figure 1. An RLC circuit that is connected to a sine voltage source. The figure is based on the simple circuit model in ([23], p. 37).

Listing 1. An example of how to model the RLC circuit in Figure 1 using MSL components.

```
package ElectricalTest
  import Modelica.Electrical.Analog.Basic;
  import Modelica.Electrical.Analog.Sources;
  model RLCCircuit
    Basic.Capacitor C;
    Basic.Ground G;
    Basic.Inductor L;
    Basic.Resistor R1;
    Basic.Resistor R2;
    Sources.SineVoltage AC;
  equation
    connect(AC.n, G.p);
    connect(AC.p, R1.p);
    connect(C.n, AC.n);
    connect(L.n, C.n);
    connect(R1.n, C.p);
    connect(R1.p, R2.p);
    connect(R2.n, L.p);
end RLCCircuit;

end ElectricalTest;
```

Furthermore, several Modelica tools allow the user to design and connect these objects using a graphical user interface and, from this graphical representation, infer and generate the code presented in Listing 1. Since the main elements of Modelica models are general equations, the language is not limited to the electrical domain, and the modelers may combine models from several domains.

3.2. MetaModelica

While Modelica does provide elements of procedural languages, such as functions and algorithms, making Modelica Turing complete, the language was not explicitly designed for language semantics modeling. MetaModelica is an extension of the Modelica language to enable language semantics modeling for the specification of programming languages [24]. MetaModelica extends the Modelica language with several features common in functional programming languages, such as pattern matching and recursive datatypes.

3.3. The Julia Language

The Julia programming language was created to combine the expressive power and flexibility of interactive scientific computing environments, such as those of Python and Matlab, with the performance of compiled procedural languages, such as Fortran and C. The Julia language achieves this by utilizing multiple dispatch in combination with dataflow type inference and runtime just-in-time compilation [1].

To demonstrate these features of Julia, consider the code in Listing 2. In this example, a single function, addition, is defined with two input arguments, *a* and *b*. Similarly to Python, which supports *duck-typing*, this function can be called with arguments of different types, in this case with integer and float arguments. However, Julia has a different execution strategy compared to Python. While Python interprets the program, Julia will instead infer a specialized function based on the runtime arguments to the function and create compiled code for each type specialization. In the case of the *addition* function, two such specializations will be created, one for the call with integer arguments *function addition(a::Int64, b::Int64)* and one for the call with floating-point arguments *function addition(a::Float, b::Float)*.

In this way, Julia retains the flexibility of scripting languages, such as Python, while at the same time generating code with good performance [1]. Still, the drawback of this approach is an initial overhead in terms of compilation time [22].

Listing 2. An example of a Julia function: *addition* of two variables.

```
function addition(a, b)
  return a + b
end
addition(1.0, 1.0)
addition(1, 1)
```

Equation-Based Modeling in Julia

Currently, in the year 2022, there exist several modeling frameworks for causal and acausal modeling within the Julia ecosystem. *DifferentialEquations.jl* [25] is one such framework. It provides a foreign function interface that allows interfacing algorithmic Julia code with different solvers and integration algorithms. A user of DifferentialEquations.jl defines a system of equations in sorted form in the Julia language to represent systems such as nonlinear systems, ODE systems and DAE systems.

While DifferentialEquations.jl provides the necessary abstractions to write causal models in Julia, it does not provide all abstractions typical of a full-fledged modeling language. The framework *ModelingToolkit.jl* (MTK) aims to address this issue [3]. MTK is a new modeling framework to automate symbolic operations common for equation-based languages, such as methods for index reduction. MTK does so by using the symbolic-numerical capabilities of Julia to preprocess a model description into a format that can be solved using the set of solvers provided by DifferentialEquations.jl. In other words, the translation 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. Modia.jl [2,26] is another framework that extends the Julia language with capabilities for acausal modeling. Syntactically, it is closer to Modelica when compared to the language defined by MTK. However, it is different from the work presented herein in several ways. Its constructs are implemented using Julia metaprogramming, primarily a set of macros, rather than traditional compiler phases. This method leads to quick prototypes but has the drawback of less or non-existent error checking and type checking. Moreover, it does not support standard Modelica.

3.4. Variable Structure Systems in the Context of Equation-Based Languages

In this section, we present a subset of equation-based languages that support systems of variable structure. A modeling language that supports variable structure systems allows the structure of the model to change during simulation. Consequently, changes to the equations and variables that describe the system's dynamics under simulation are allowed.

$$f = \begin{cases} g(t, \underrightarrow{\dot{x}}(t), \underrightarrow{x}(t), \underrightarrow{y}(t), \underrightarrow{p}), & \text{if } t \geq 0.5 \\ h(t, \underrightarrow{\dot{\chi}}(t), \underrightarrow{\chi}(t), \underrightarrow{\gamma}(t), \underrightarrow{p}), & \text{Otherwise} \end{cases} \tag{1}$$

Equation (1) describes a system with a highly variable structure, where the structure change due to a timing condition. (A system with variable structure need not exhibit an increase in the number of equations nor involve removing components. Such a system might also have structural changes due to ideal components, where existing equations need to be activated and deactivated [17,27]. We call the system in Equation (1) highly variable, since the new equations have no static relation to the previous equations before the structural transition.) The equation depicts a system that is initially described by the equations of $g(t, \underrightarrow{\dot{x}}(t), \underrightarrow{x}(t), \underrightarrow{y}(t), \underrightarrow{u}(t), \underrightarrow{p})$, where $\underrightarrow{x}$ is the set of state variables, $\underrightarrow{y}$

is the set of algebraic variables and p is the set of parameters and constants. When the transition condition occurs, the system changes. After the change, the system f is defined by $h(t, \underrightarrow{\dot{\chi}}(t), \underrightarrow{\chi}(t), \underrightarrow{\gamma}(t), \underrightarrow{p})$, where the state and algebraic variables ($\underrightarrow{\chi}$, $\underrightarrow{\gamma}$) in h differ from those of g. If such changes occur during simulation, we say that the system is a *variable structure system*. In Listing 3 we model a system where $\underrightarrow{x}$ increases during each such structural change. In that example, the state vector $\underrightarrow{x}(t) = \{x_1, ..., x_{10}\}$ changes to $\underrightarrow{\chi}(t) = \{x_1, ..., x_{20}\}$ The values of $\underrightarrow{x}(t)$ before the structural transition are the initial values for $\underrightarrow{\chi}(t) = \{x_1, ..., x_{10}\}$ when the simulation restarts.

Listing 3. The ArrayGrow model.

```
// This is an example of a model with structural variability
// We initially start with 10 equations, however during the
 ↪  simulation
// the amount of equations are doubled.
model ArrayGrow
   parameter Integer N = 10;
   Real x[N](start = {i for i in 1:N});
equation
  for i in 1:N loop
    x[i] = der(x[i]);
  end for;
  when time > 0.5 then
    // Recompilation with change of parameters.
    // the name of this function is the subject of change.
    // What is changed depends on the argument passed to this
     ↪  function.
    recompilation(
     N /* What we are changing */,
     2 * N /* The value of the change */
    );
  end when;
end ArrayGrow;
```

A real example of structural variability is a wind power plant that connects or disconnects from the grid depending on the wind speed. Understandably, many systems around us have some structural variability. Besides providing increased modeling capabilities, support for varying the structure of a model during simulation also has other advantages. For example, the model can change the granularity of a specific subsystem during the simulation. With this approach, the entire system's dynamics need not be specified and simulated. Instead, the model can use a low complexity approximation at the start of the simulation and switch to a more fine-grained description when some conditions are met. For example, if we model a water dam, we might not be interested in calculating the mechanics of materials until the amount of water has reached a certain threshold. Hence, computational resources can be saved, since the entire system's need not be specified initially. Due to the advantages of supporting systems with varying structure, several languages and environments have been developed. However, as it turns out, extending an equation-based language to support variable structure results in additional complications in terms of language design. In Section 4, we discuss research related to VSS.

4. Related Work

Several previous studies have attempted to provide support for VSS in other frameworks for equation-based modeling and simulation. Nevertheless, this functionality is

presently not available in any mainstream equation-based languages, and the area is still under active research in the context of equation-based languages and other similar modeling frameworks [20]. We expand on the discussion in Section 9.1.

4.1. Mosilab

An extension to the Modelica language to allow models with variable structure is the Modeling and Simulation Language (MOSILA) [28] within the modeling and simulation environment Mosilab. To support the modeling of systems with variable structure Mosilab introduces *dynamic object structures*. These structures are specified statically before the simulation and can be activated or deactivated at discrete time events.

4.2. Sol

Zimmer argues that an issue with Mosilab is that individual components of the system cannot modify themselves [29]. To overcome the limitations of the Mosilab approach and extend the expressiveness of Modelica, Zimmer [29] proposed the Sol language [29]. The Sol language while superficially similar to Modelica is a separate language designed to support VSS using the framework *SolSim*.

One example of how structural variability is used in Sol can be seen in Listing 4. The model consists of model variants, Engine1 and Engine2. Engine2 is more computationally expensive to simulate compared to Engine1. However, during the simulation, the dynamics of the engine change due to the relationship between the inertia and the torque. Due to this, the level of detail of Engine2 is no longer needed, and Engine1 can be used instead to speed up the simulation process. This transition is captured by the $F.w > 40$ condition in the when equation, which results in the switch in the engine model.

Listing 4. A machine model with a structural change from ([29], p. 78).

```
model Machine
implementation:
  static Mechanics.FlyWheel F{inertia << 1}
  static Mechanics.Gear G{ ration << 1.8}
  connection{a << G.f2, b << F.f};
  static Boolean fast;
  if fast then
    static Mechanics.Engine1 E{meanT << 10};
    connection{a << E.f, b << G.f1};
  else then
    static Mechanics.Engine2 E{meanT << 10};
    connection{a << E.f, b << G.f1};
  end;
  if initial() then
    fast << false;
  end;
  when F.w > 40 then
    fast << true;
  end;
end Machine;
```

4.3. Hydra

Hydra [30] is an embedded language implemented in Haskell according to the paradigm of functional hybrid modeling [31]. Hydra supports acausal modeling but lacks some of the object-oriented features present in languages such as Modelica. Still, Hydra compensates for the lack of these capabilities by providing increased flexibility compared to the static

Modelica language. One example of this flexibility is handling systems where the set of equations and variables change during simulation. Hydra handles this issue by utilizing just-in-time compilation [30].

4.4. Compiling Modelica: Model Composition Language and NanoModelica

In the PhD thesis, *Compiling Modelica*, Höger [32] presents both a theoretical framework and an experimental prototype that is capable of handling systems with a dynamic structure and separate compilation. A Modelica model specified in NanoModelica was translated to a Hybrid-DAE representation. This representation was then translated to the Model Composition Language (MCL). MCL is inspired by the Model Kernel Language (MKL) [33]. The purpose of the kernel language is to provide a formal framework to describe the semantics of modeling languages. MCL is used as an intermediate representation in the translation process before finally being transformed into OCaml (https://ocaml.org/, accessed on 15 February 2022). MCL is not Modelica; rather, it is a concise language meant to support the behavior of a subset of Modelica denoted NanoModelica.

4.5. Other Related Work within the Context of Variable Structure Systems

A proposal to integrate VSS support similar to [28] was presented in [34]. Like Mosilab, structural transitions between states are used to represent multi-mode models. The state machine approach by [28,34] is similar to the explicit VSS discussed in Section 6.1. Other works dealing with the theoretical background of multi-mode models are [16,17]. These techniques have previously been applied to Modelica in [18].

A Python environment that combines several existing frameworks in order to simulate VSS was proposed by Mehlhase [35].

The idea of designing compilers in a composable fashion is not new. A recent example is the LLVM Compiler Infrastructure [36]. Other examples include [37,38].

5. OpenModelica.jl

OpenModelica.jl is a modeling and simulation environment implemented in the Julia language dedicated to Modelica modeling and simulation. An overview of the various components of OpenModelica.jl is available in Figure 2. In the figure, we can see that the backend depends on the frontend. The reason for this dependency is to enable the compiler to recompile models during simulation. Following the principles of LLVM [36], the frontend and the intermediate representations are separated so that additional frontends or backends can be provided to support other equation-based languages. This is illustrated in Figure 3.

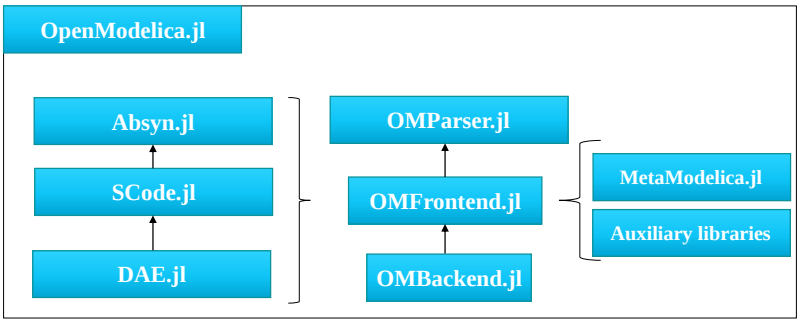

Figure 2. An overview of the dependencies between the components in OpenModelica.jl. Absyn.jl, SCode.jl and DAE.jl are the compilers' intermediate representations. The compiler runtime is implemented by MetaModelica.jl and auxiliary libraries. The frontend is provided by OMFrontend.jl and the backend is defined by OMBackend.jl.

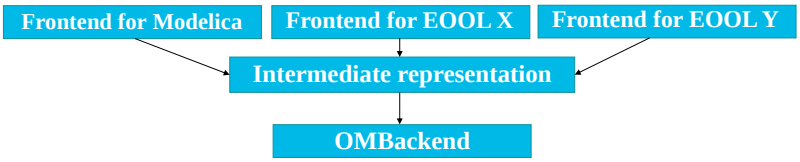

Figure 3. A high-level overview of a design separating the intermediate representation from the frontend to allow several hypothetical frontends to use the same backend.

The sections below describe the main modules of our framework. The frontend is implemented by automatic translation of the existing OpenModelica frontend written in MetaModelica. To make the translated code more readable, we implemented *MetaModelica.jl* as a compatibility layer.

We started this section describing MetaModelica.jl in Section 5.1 and end with a summary in Section 5.7.

5.1. MetaModelica.jl

MetaModelica.jl (URL: https://github.com/OpenModelica/MetaModelica.jl accessed on 5 of May 2022) provides a compatibility layer between Julia and MetaModelica [24,39] and an extension to the Julia language via Julia metaprogramming. It re-implements several constructs of MetaModelica, such as *match* and *matchcontinue*. Furthermore, MetaModelica.jl reimplements the OMC compiler runtime (the reason is that the frontend is automatically generated from the existing OpenModelica Frontend). A method for type checking array expression written in MetaModelica can be studied in Listing 5. The corresponding function in Julia that the translator generates can be studied in Listing 6.

For a more-in-depth comparison between Julia and MetaModelica, we refer to [40].

Listing 5. Original code written in MetaModelica to type check array expressions.

```
function matchArrayExpressions
  input output Expression exp1;
  input Type type1;
  input output Expression exp2;
  input Type type2;
  input Boolean allowUnknown;
  output Type compatibleType;
  output MatchKind matchKind;
protected
  Type ety1, ety2;
  list<Dimension> dims1, dims2;
algorithm
  Type.ARRAY(elementType = ety1, dimensions = dims1) := type1;
  Type.ARRAY(elementType = ety2, dimensions = dims2) := type2;
  // Check that the element types are compatible.
  (exp1, exp2, compatibleType, matchKind) :=
    matchExpressions(exp1, ety1, exp2, ety2, allowUnknown);
  // If the element types are compatible, check the dimensions
   ↪  too.
  (compatibleType, matchKind) :=
    matchArrayDims(dims1, dims2, compatibleType, matchKind,
     ↪  allowUnknown);
end matchArrayExpressions;
```

Listing 6. A function used in the type checking phase of our Modelica compiler. This code make use of the @match equation constructs from MetaModelica; for comparison, the original MetaModelica code is available in Listing 5.

```
function matchArrayExpressions(
  exp1::Expression,
  type1::NFType,
  exp2::Expression,
  type2::NFType,
  allowUnknown::Bool,
  )::Tuple{Expression, Expression, NFType, MatchKindType}
  local matchKind::MatchKindType
  local compatibleType::NFType
  local ety1::NFType
  local ety2::NFType
  local dims1::List{Dimension}
  local dims2::List{Dimension}
  @match TYPE_ARRAY(elementType = ety1,
                    dimensions = dims1) = type1
  @match TYPE_ARRAY(elementType = ety2,
                    dimensions = dims2) = type2
  #= Check that the element types are compatible. =#
  (exp1, exp2, compatibleType, matchKind) =
    matchExpressions(exp1, ety1, exp2, ety2, allowUnknown)
  #= If the element types are compatible, check the dimensions.
   ↪  =#
  (compatibleType, matchKind) =
    matchArrayDims(dims1, dims2, compatibleType, matchKind,
     ↪  allowUnknown)
  return (exp1, exp2, compatibleType, matchKind)
end
```

5.2. OMParser.jl

OMParser.jl (URL: https://github.com/OpenModelica/OMParser.jl.git accessed on 5 of May 2022) is the parser of our compiler framework. The parser is defined using the ANTLR parser generator [41] and is based on the existing OpenModelica parser. It is presently capable of parsing the entire Modelica Standard Library (see Listing 7). Following our modular design, the parser can be used in separation, and the MetaModelica.jl layer and the Absyn.jl representation produced by the parser.

Listing 7. An example of how the parser could be used, in this case to parse the Modelica Standard Library (MSL).

```
import OMParser
res = OMParser.parseFile("msl.mo")
```

5.3. OMFrontend.jl

To transform Modelica into flat Modelica, which is a flattened representation of Modelica models where object oriented constructs have been expanded, we implemented *OMFrontend.jl*. OMFrontend.jl was automatically generated from the MetaModelica code of the new high-performance frontend of OMC [7]. Previously, we used the old frontend [5]; however, as part of the work presented here, the *MetaModelica-Julia translator* was used to automatically

generate a Julia implementation of the new OMC frontend. While the translation of the old frontend was achieved without any major modifications. The design and implementation of this frontend remain roughly the same as described by Pop et al. in [7]. However, we made additions to provide support the language extensions described in Section 6.

5.4. Testing the Frontend by Using Flat-Modelica

Similarly to the frontend in OMC, our compiler is capable of generating flat Modelica from a Modelica model. This is currently used in our continuous-integration pipeline to check our frontend's fidelity compared to the original OpenModelica frontend. This feature can also be used to export a model constructed within our framework to other Modelica compilers that are less capable of handling the object-oriented structure of Modelica. In Listing 8, we can see a model representing a water tank, and in Listing 9 we can see the corresponding flat model generated by OpenModelica.jl.

Listing 8. A water tank model.

```
connector Stream
  Real pressure;
  flow Real volumeFlowRate;
end Stream;

model Tank
  parameter Real area = 1;
  replaceable connector TankStream = Stream;
  TankStream inlet, outlet;
  Real level(start=2);
equation
  inlet.volumeFlowRate = 1;
  inlet.pressure = 1;
  area * der(level) = inlet.volumeFlowRate + outlet.volumeFlowRate;
  outlet.pressure = inlet.pressure;
  outlet.volumeFlowRate = 2;
end Tank;
```

Listing 9. The flat model of the water tank.

```
class Tank
  parameter Real area = 1.0;
  Real inlet.pressure;
  flow Real inlet.volumeFlowRate;
  Real outlet.pressure;
  flow Real outlet.volumeFlowRate;
  Real level(start = 2.0);
equation
  inlet.volumeFlowRate = 0.0;
  outlet.volumeFlowRate = 0.0;
  inlet.volumeFlowRate = 1.0;
  inlet.pressure = 1.0;
  area * der(level) = inlet.volumeFlowRate + outlet.volumeFlowRate;
  outlet.pressure = inlet.pressure;
  outlet.volumeFlowRate = 2.0;
end Tank;
```

Presently, we have more than 70 tests that check various Modelica language features in our continuous integration pipeline.

5.5. Library Support

The compiler presented in this text provides support for users' own libraries. Furthermore, we have also tested our compiler on existing Modelica libraries, such as the ScalableTestSuite and the Modelica Standard Library (MSL). While OpenModelica.jl does not cover all use cases of the MSL, some models from the electrical domain are currently supported. Listing 1 is an example of how the MSL components can be used in this framework to represent the RLC circuit in Figure 1. The complete flat model produced by our frontend for this model is available in Listing A3 in the appendix, Section A.1.

In Section 8, we use the scalable transmission line model of the ScalableTestSuite [19] to estimate the current performance of our frontend to see how it compares to a state-of-the-art Modelica compiler.

5.6. OMBackend.jl

The module responsible for code generation is the backend module *OMBackend.jl*. Current backend targets include both MTK and DifferentialEquations.jl. The DifferentialEquations.jl backend uses the Sundials IDA solver [42] and roughly follows the DAE-mode implementation by [43]. It currently supports continuous systems and has experimental support for hybrid systems. The backend currently performs matching and sorting on the equations; however, the process of symbolic index reduction and other compiler optimizations, such as algebraic simplification, are outsourced to the MTK-framework. Furthermore, the backend integrates other Julia facilities, such as *Plots.jl* [44], for plotting and animation.

5.7. OpenModelica.jl as a Compiler Architecture

In the previous section, we presented the components that make up our compiler. We argue that our proposed design provides the necessary components for a retargetable Modelica compiler. In Figure 4, we illustrate how these components map to a typical compiler architecture. The compiler phases of lexing and parsing are combined in OMParser, producing the high-level intermediate code, Absyn. The Absyn representation is then simplified by OMFrontend into the lower level intermediate code called SCode. The reason for this is that Modelica allows several component declarations on the same line; in SCode these are expanded to make the handling more uniform. OMFrontend then performs semantic analysis, removal of the object orientation, constant evaluation of structural parameters, connection handling and constant folding; and generates the DAE representation containing a set of differential and algebraic equations, functions and declarations. The final result is flat Modelica. The flat Modelica is then supplied to OMBackend.jl, which generates Julia code, either in the intermediate format specified by MTK or in DifferentialEquations.jl. These two frameworks and the Julia compiler finally perform further compiler optimizations before generating machine code for future simulation using some numerical solver with a user-specified integration algorithm.

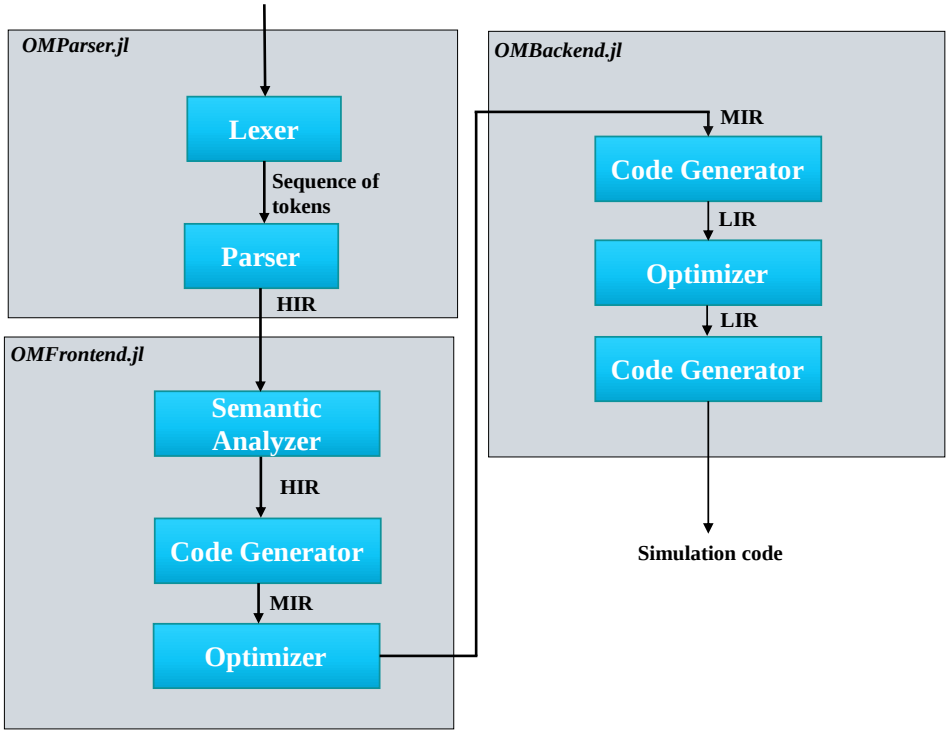

Figure 4. A mapping of the compiler components in OpenModelica.jl to a modern compiler architecture. OMParser.jl is responsible for parsing and lexing, OMFrontend.jl is responsible for semantic analysis and optimization of the high-level intermediate representation (HIR). Finally, OMBackend.jl performs the final code optimization and generation. This figure is adapted from ([45], p. 8).

Using all of these components we can construct OpenModelica.jl, our Modelica compiler. For an illustration on how to use the compiler framework to simulate the Modelica HelloWorld model (Listing 10), see Listing 11.

Listing 10. The Modelica HelloWorld model.

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

Listing 11. Simulating a Modelica model in OpenModelica.jl.

```
using OM
using Plots
res = simulate("HelloWorld",
               "HelloWorld.mo",
               startTime=0.0,
               stopTime=1.0,
               solver = Rodas5())
plot(res)
```

6. Extending the Modelica Language to Support Systems with Variable Structure

One of the main motivations of selecting Julia both as the implementation language and our compiler's target language is that the Julia environment provides several libraries for scientific computing. Another reason is that the Julia language is JIT (just-in-time) compiled. We have previously investigated the benefits of this approach in the context of bringing JIT compilation to Modelica [22]. To leverage the capabilities of the Julia JIT compiler, we extended our compiler pipeline beyond that of existing Modelica compilers, such as OMC, by enabling our compiler to call itself while a model is being simulated. To illustrate this, consider Figure 5. In the figure, Modelica code is translated using the usual process of parsing the Modelica model and proceeding with the steps depicted in Figure 4. However, the simulation code is capable of calling the compiler in the event of a *structural change*. This results in the simulation being halted, and the compiler is then invoked to produce a new model with the changes incorporated.

The simulation runtime then maps the old state to the new state and continues simulating the system again until a new such change occurs.

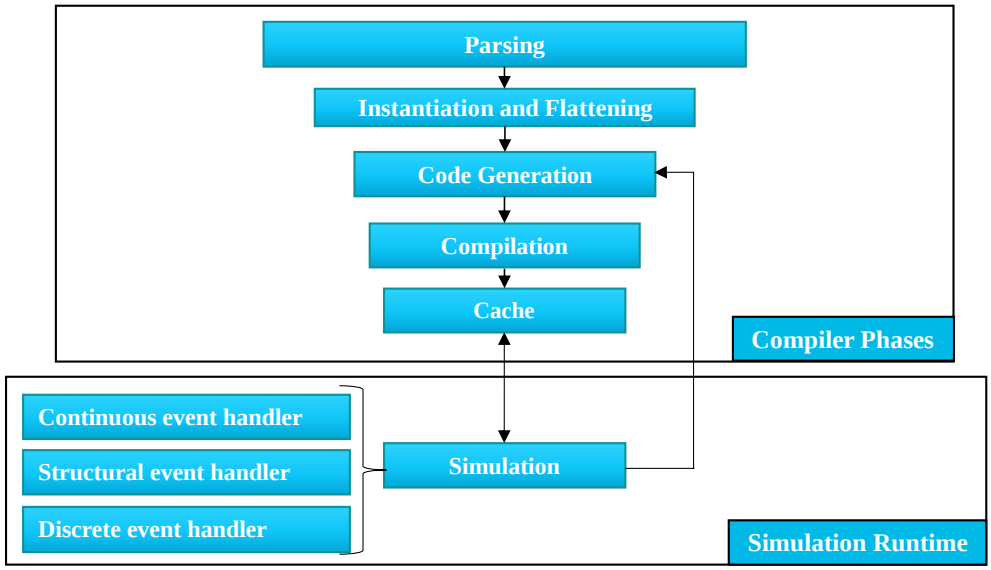

Figure 5. The compilation process of a JIT capable Modelica compiler. During simulation, a structural change may trigger recompilation, and re-initialization of the system that is simulated.

This section discusses two extensions that enable the Modelica language to represent such systems. We start by discussing so-called *explicit variable structured systems*. This discussion is followed by a discussion concerning *implicit variable structured systems*.

6.1. Explicit Variable Structured Systems

We define *explicit variable structured systems* as systems where the transitions between states of the system are explicitly encoded by the *modeler*. That is, the equations and variables of such a system are known before the system is simulated. To illustrate this, we reused the process of representing state machines in the Modelica language by providing support for *continuous state machines*, inspired by [28,34].

However, while state machines in Modelica do not support continuous-time equations or algorithms (URL: https://specification.modelica.org/v3.4/Ch17.html, accessed on 22 April 2022), our representation allows a modeler to represent structural transitions between separate continuous-time states. To be able to encode such explicit structural transitions, we introduced one new keyword, *structuralmode*, along with two operators:

- `initialStructuralState(state)`
- `structuralTransition(fromState, toState, condition)`

The operator `initialStructuralState` represents an initial structural state, and `structuralTransition` is used to specify the transition between one structural state to another such state.

Listing 12 illustrates an example of a system modeled using these constructs. The model *SimpleTwoModes* consists of two states, *Single* and *HybridSingle*. The model starts in the *Single* state, and after 0.7 s, the model transitions to the next state *HybridSingle*. This transition is modeled using the *structuralTransition* operator, and the initial structural state is specified using the `initialStructuralState` operator. In the case of the *SimpleTwoModes* model, this consists of the single variable, x.

Listing 12. An example of a simple explicit variable structured systems with two modes.

```
model SimpleTwoModes
  model Single
    parameter Real a = 1.0;
    Real x (start = 1.0);
  equation
    der(x) = 2 * x + a;
  end Single;
  model HybridSingle
    parameter Real a = 1.0;
    Real x (start = 0.0);
  equation
    der(x) = x  - a;
  end HybridSingle;
structuralmode Single firstMode;
structuralmode HybridSingle secondMode;
equation
  // We start in this intial mode
  initialStructuralState(firstMode);
  // We switch the mode when time is larger or equal 0.7
  structuralTransition(firstMode, secondMode, time  >=  0.7);
end SimpleTwoModes;
```

The code for simulating and plotting this model is available in Listing A1, and the plot is available in Figure 6.

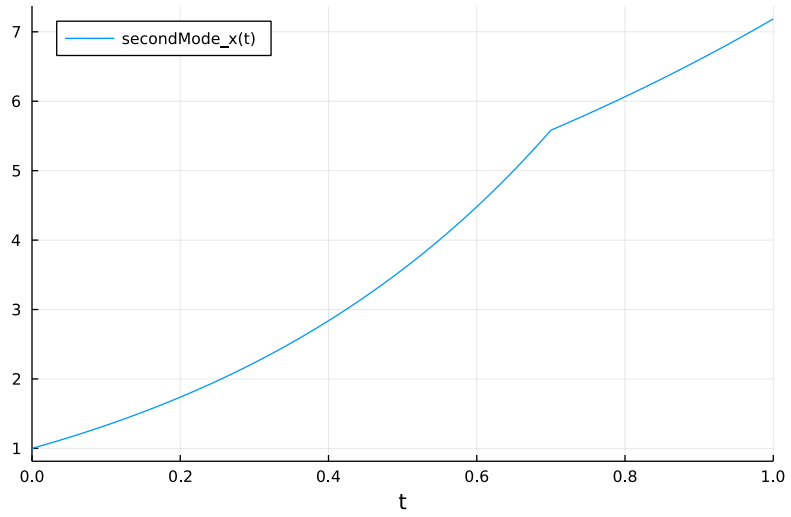

Figure 6. The result of simulating Listing 12.

Modeling a Breaking Pendulum Explicitly

Using these constructs, we can simulate two models during the same simulation where the equations and variables change. However, this requires that these changes are provided explicitly by the modeler. We can use this methodology of explicitly encoding the states to model a breaking pendulum model.

Simulating this system results in the plot seen in Figure 7. To summarize, using an explicit approach, we can increase the expressiveness of Modelica. However, there are some disadvantages to this approach. The first is that the representation is causal; the transition between the states needs to be encoded sequentially. The second drawback is that all equations need to be represented before simulation. The disadvantage of this is that the compiler and the simulation runtime need to process the entire model at once. While the transition between states can be achieved dynamically, the model may not modify itself during simulation. In the next section, we discuss the second extension to the Modelica language, *Implicit Variable Structured Systems*. These systems are similar to the systems we discussed in this section. However, the restriction that the model may not modify itself is lifted. It means that the entire behavior of the model need not to be explicitly encoded.

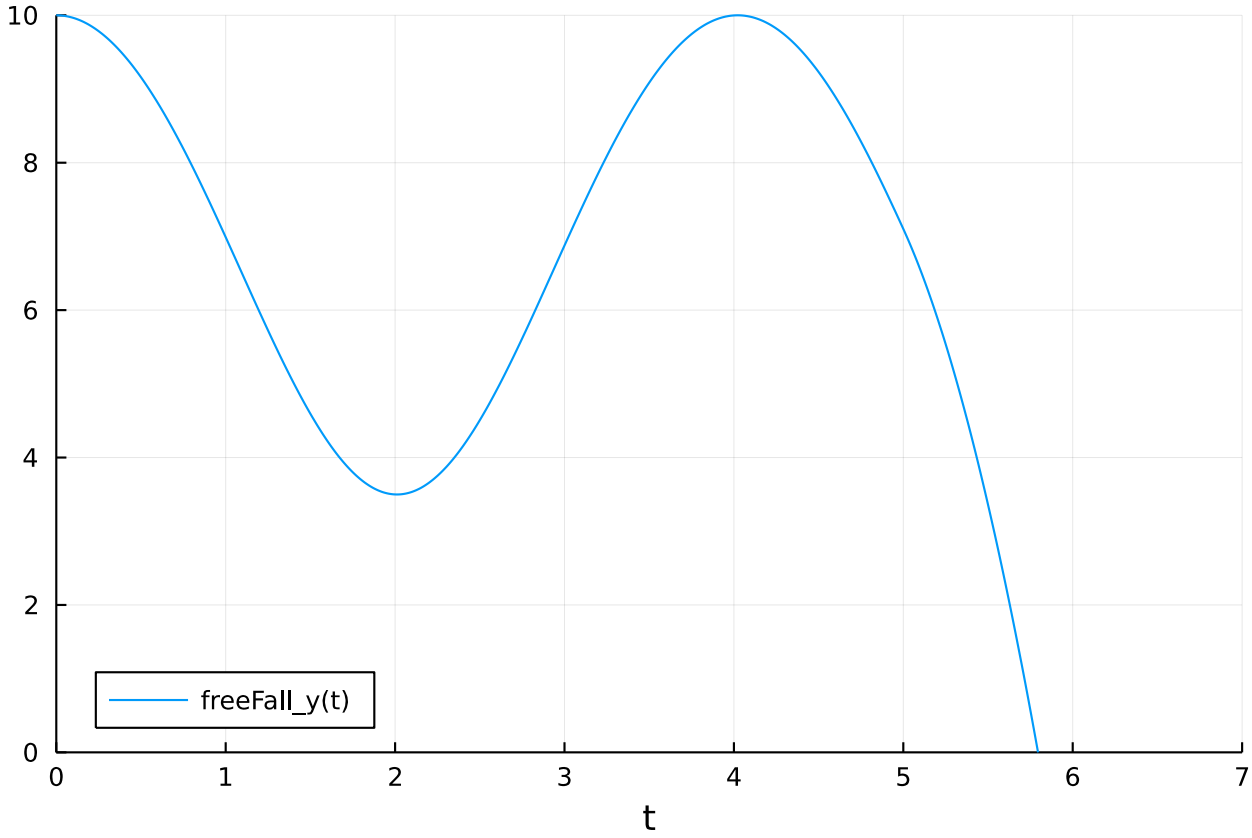

Figure 7. The result of simulating Listing 13. Note that the legend provides the prefix of the last active model which was *FreeFall*. The behavior before the structural transition at $t = 5$ is described by the equations of the pendulum model.

Listing 13. An example of the breaking pendulum model using structural transitions.

```
model FreeFall
  Real x;
  Real y;
  Real vx;
  Real vy;
  parameter Real g = 9.81;
  parameter Real vx0 = 0.0;
equation
  der(x) = vx;
  der(y) = vy;
  der(vx) = vx0;
  der(vy) = -g;
end FreeFall;
model Pendulum
  parameter Real x0 = 10;
  parameter Real y0 = 10;
  parameter Real g = 9.81;
  parameter Real L = sqrt(x0^2 + y0^2);
  // Common variables
  Real x(start = x0);
  Real y(start = y0);
  Real vx;
  Real vy;
  // Model specific variables
  Real phi(start = 1.0, fixed = true);
  Real phid;
equation
  der(phi) = phid;
  der(x) = vx;
  der(y) = vy;
  x = L * sin(phi);
  y = -L * cos(phi);
  der(phid) =  -g / L * sin(phi);
end Pendulum;

model BreakingPendulum
structuralmode Pendulum pendulum;
structuralmode FreeFall freeFall;
equation
  initialStructuralState(pendulum);
  structuralTransition(pendulum,
                       freeFall,
                       time - 5.0 <= 0);
end BreakingPendulum;
```

6.2. Implicit Variable Structured Systems

In the previous section we discussed systems that we denote *explicit variable structure systems*. According to some explicitly stated scheme, these are models where the variables and equations change during simulation. This section provides examples of implicit systems where we lift the restriction on this explicit encoding. We do so by introducing a single new keyword: *recompile*.

Recompile allows structural events to trigger a modification and subsequently recompilation of the model under simulation. To achieve this, we extended the flat Modelica

representation to also contain a *MetaModel* of itself (at the time of writing, this meta-model is encoded using the SCode representation). During compilation, the model may query itself and change the values of its parameters. In this way, the different sets of equations and variables need not be explicitly encoded before structural transitions. Consequently, values computed by some models during simulation may be used to modify the model itself.

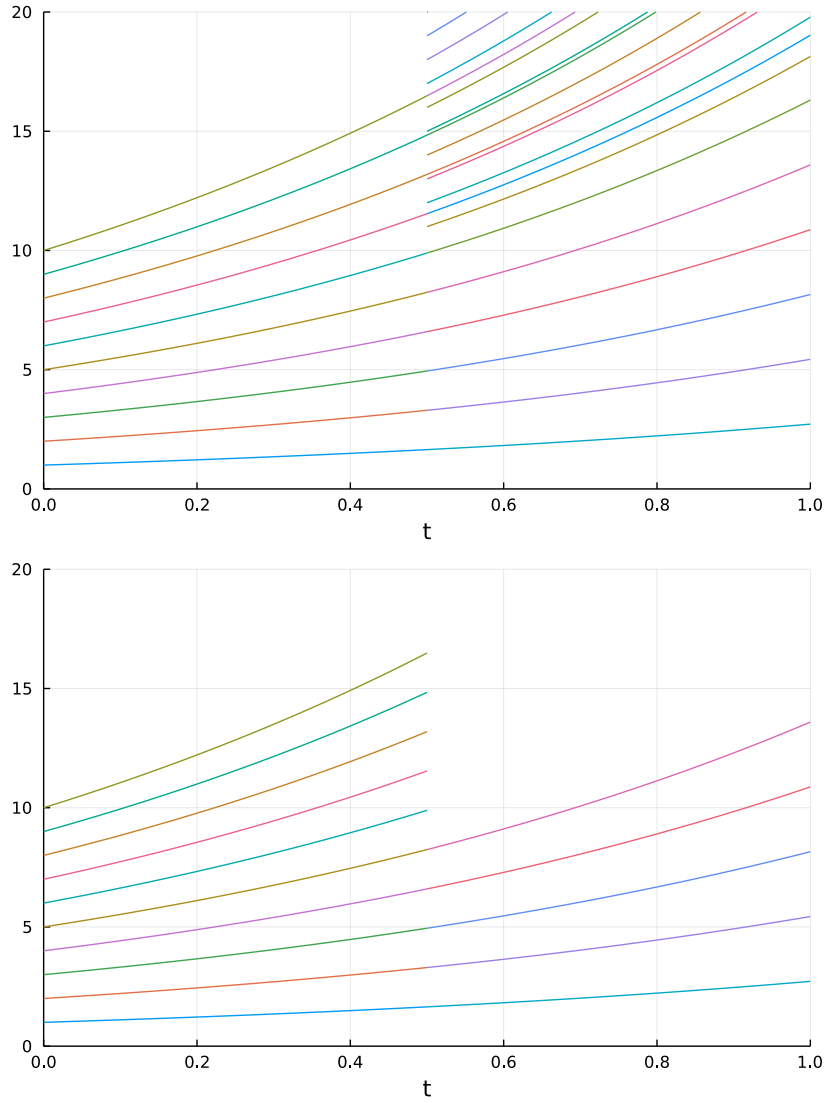

Figure 8. The results of simulating Listing 3 (**Top**) and Listing 14 (**Bottom**). The structural change and subsequent recompilation of the model occur at $t = 0.5$ seconds. The curve in the graph represents all x_i variables of $\underrightarrow{x}$ in Listing 3 and Listing 14.

To illustrate, consider the two examples in Listings 3 and 14, respectively. At the start of the simulation, *ArrayShrink* consists of ten equations and variables. However, after 0.5 s, the system changes radically, and the number of equations and variables shrinks to five. For the second example in Listing 3, the system initially consists of 10 equations; however, during the simulation, the set of equations and variables doubles to be of size 20. The resulting plot when simulating this system can be studied in Figure 8.

Listing 14. The ArrayShrink model.

```
// This is an example of a model with structural variability
// We initially start with 10 equations, however during the
// simulation, the amount of equations are decreased to 5.
model ArrayShrink
   parameter Integer N = 10;
   Real x[N](start = {i for i in 1:N});
equation
  for i in 1:N loop
    x[i] = der(x[i]);
  end for;
  when time > 0.5 then
    // Recompilation with change of parameters.
    // the name of this function is the subject of change.
    // What is changed depends on the argument passed to this
     ↪  function.
    recompilation(
      N /* What we are changing */,
      5 /* The value of the change */
    );
  end when;
end ArrayShrink;
```

The code and the resulting plot of this system is presented in Listing A2. The benefit of this approach is that it can also model the explicit models discussed previously. Using the *recompilation* construct, we can the model during simulation by querying and updating the meta-model. If we consider the Modelica model representing the breaking pendulum from Listing 13, all the modes are held in memory during simulation. With this proposed extension, we can reformulate this model as shown in Listing 15, where only one of the mode is active during simulation and therefore present in memory. Simulating this system results in the plot seen in Figure 9.

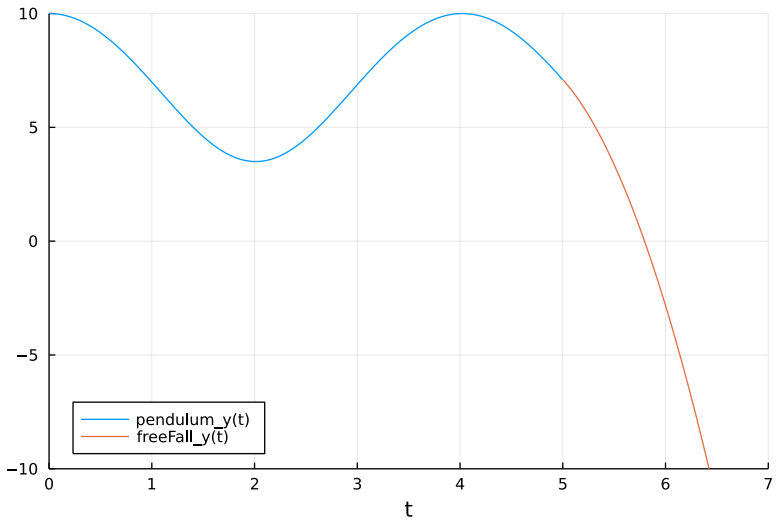

Figure 9. The result of simulating the breaking pendulum using recompilation during simulation. The code to obtain this plot is presented in Listing A2.

Listing 15. The breaking pendulum model using the new recompilation keyword to activate and deactivate components via just-in-time compilation during simulation.

```
model BreakingPendulum

model FreeFall
  parameter Real e=0.7;
  parameter Real g=9.81;
  Real x;
  Real y;
  Real vx;
  Real vy;
equation
  der(x) = vx;
  der(y) = vy;
  der(vy) = -g;
  der(vx) = 0.0;
end FreeFall;

model Pendulum
  parameter Real x0 = 10;
  parameter Real y0 = 10;
  parameter Real g = 9.81;
  parameter Real L = sqrt(x0^2 + y0^2);
  // Common variables
  Real x(start = x0);
  Real y(start = y0);
  Real vx;
  Real vy;
  // Model specific variables
  Real phi(start = 1., fixed = true);
  Real phid;
equation
  der(phi) = phid;
  der(x) = vx;
  der(y) = vy;
  x = L * sin(phi);
  y = -L * cos(phi);
  der(phid) =  -g / L * sin(phi);
end Pendulum;

  parameter Boolean breaks = false;
  FreeFall freeFall if breaks;
  Pendulum pendulum if not breaks;
equation
  when 5.0 <= time then
    recompilation(breaks, true);
  end when;
end BreakingPendulum;
```

In this and previous sections we presented *OpenModelica.jl*, a Modelica environment in Julia, and discussed some of the implications of having an equation-based compiler written in this language. While we have illustrated the possibilities of our framework, we have not yet presented its feasibility. The next two sections describe the evaluation of our framework and how it performs in comparison to an existing state-of-the-art Modelica

compiler, the Open Modelica Compiler (OMC). We tested it by using several benchmarks from the ScalableTestSuite [19].

7. Materials and Methods

In this section, we describe the software and hardware used in our experiments.

Instrumentation

The experiments were run with on a system with the following specifications. The operative system was Ubuntu 20.04.4 LTS on a AMD Ryzen (Threadripper 1950X 16-Core Processor) with 130 GB of RAM. The versions of OpenModelica, ModelingToolkit and Julia were 1.18.1, 8.5.0 and 1.7.2, respectively. Concerning the models used in our experiments, the transmission line model is located in Section A.2. In the appendix, we have also included some of the compiler-generated models. The results of our experiments are presented in Section 8.

8. Results and Benchmarking

The previous section presented a novel Modelica framework. In this section, we evaluate the performance of this framework to establish its feasibility. We start by analyzing the simulation performance of the framework and how it compares to an existing state-of-the-art framework with a similar aim, the OpenModelica environment, in Section 8.1, using a model from the ScalableTestSuite library.

This is followed by Section 8.2, where we examine the frontend performance when translating a set of Modelica models using the Modelica Standard Library (MSL) to flat DAE. We have previously explained how we extended the Modelica language to allow the simulation of systems with variable structure via JIT compilation. To examine the overhead and possible advantages of this method, we compare the costs induced by this method in Section 8.2. We end with Section 8.3, where we summarize the results and the implication of our experiments.

8.1. Simulation of Large Modelica Models

Here, we evaluate the simulation time performance of simulating large Modelica models using our proposed compiler. The model selected for this experiment was the CascadingFirstOrder system from the scalable testsuite; see Listing A4. In our experiment, we gradually increased N from 10 to 25,600 and simulated the system using the MTK backend of our proposed compiler with the TSIT5 [46] solver. The resulting simulation time was evaluated using BenchmarkTools.jl [47]. We also performed the same experiments using OMC with the IDA solver; the IDA solver was selected since OMC, at this time, does not support TSIT5. (At this time, there are issues in setting up the IDA solver within the MTK framework in the same way as it is done in OMC). The benchmarking program was set to use 1000 samples for each level of N. The timeout over all samples for each N was set to 500 seconds.

OMC was used with the standard settings and the IDA solver.

The resulting simulation time performance is presented in Figure 10. From this experiment, we can see that the simulation time performance of our proposed compiler is on par with at least one state-of-the-art Modelica compiler. Furthermore, since the MTK environment supports more solvers compared to OMC, we can also leverage this difference and achieve better performance than OMC. The feasibility of the MTK framework has also been discussed in other literature, such as [13], where MTK outperformed the commercial Dymola compiler in a specific case. However, due to the high memory requirements of Julia and MTK, we were currently unable to go further than 25,600 equations in this benchmark.

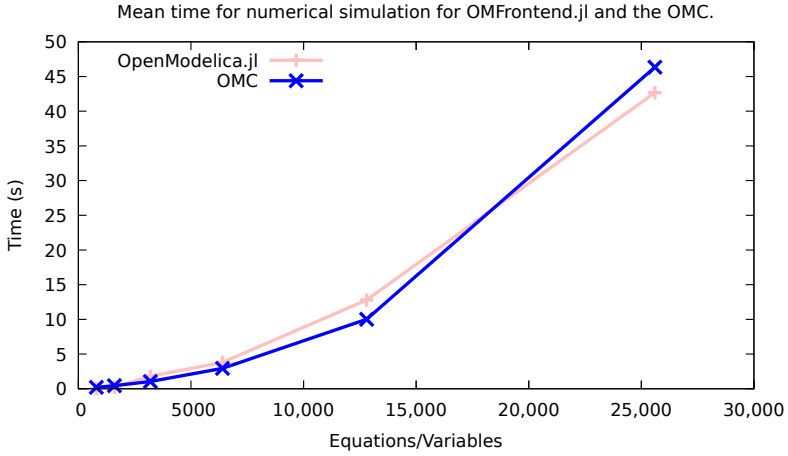

Figure 10. Time spent during numerical simulation for OpenModelica.jl (pink) and for OMC (blue). Lower is better.

8.2. Evaluating Compile-Time Overhead

In this section, we present current numbers concerning the compile-time overhead when flattening large Modelica models. We selected the transmission line mode from the ScalableTestSuite for this experiment since, it represents a typical Modelica model and makes use of the Modelica Standard Library. The full model is presented in Listing A5. In this experiment, we generated scalarized flat Modelica code by gradually increasing N in the transmission line model, starting with $N = 10$ and ending with $N = 1280$. For each value of N, we ran the experiments for 500 seconds with the maximum sample size of 100. The results of this experiment are presented in Figure 11.

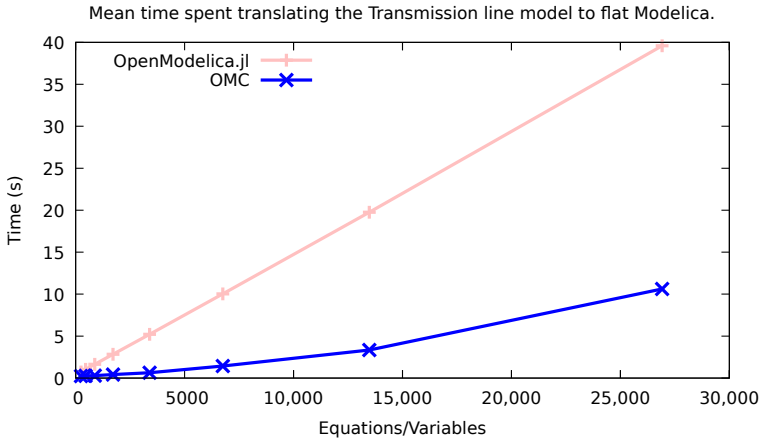

Figure 11. Time spent translating the transmission line model in Listing A5 to flat Modelica for OMFrontend.jl (pink) and for OMC (blue). Lower is better.

Evaluating the Cost of Structural Changes and JIT Compilation

When introducing JIT compilation in a Modelica compiler, it is important to examine the overhead imposed by this technique. In the previous section, we presented the current performance characteristics of our proposed Modelica frontend. In this section, we evaluate the cost of compilation during simulation using a modified variant of the previously described *ArrayGrow* model, *SimpleClockArrayGrow*; see Listing 16.

Listing 16. SimpleClockArrayGrow. This model initially starts out with N equations and variables; however, every 15 s, the structure of the model changes and K new equations and variables are added to the system.

```
// This model a exhibits the same behavior as array grow,
// except that it resizes several times
model SimpleClockArrayGrow
  parameter Integer N = 1000;
  parameter Integer K = 2000;
  Real x[N](start = {i for i in 1:N});
equation
  when sample(0.0, 15.0) then
    recompilation(N, N + K);
  end when;
  for i in 1:N loop
    x[i] = der(x[i]);
  end for;
end SimpleClockArrayGrow;
```

This model differs from the previous *ArrayGrow* in that it gradually grows the system of equations during simulation every 15 s instead of just once. A more realistic model with similar behavior could be a nuclear power plant where different reactors are scheduled to be active at specific times, or some other system where the dynamics of said system might change abruptly during regular intervals.

In this experiment, we simulated *SimpleClockArrayGrow* for 60 s. Initially, the model consisted of 1000 equations and variables, but after 15 s, the dynamics of the system changed, and the number of equations and variables increased by 2000. This process was repeated continuously until the system reached 7000 equations.

The median value was computed by running the model 5 times. The solver used in this experiment was Rodas5, a Rosenbrock method for stiff problems, with the tolerance set to 10^{-6}. This solver was selected to emulate computationally expensive simulation.

The reason for not using a standardized benchmark suite for this example was that it was not possible to configure *BenchmarkTools.jl* with the granularity necessary to estimate the cost of the various phases. If we examine the data in Figure 12, we can see that the main cost of recompiling during the simulation was caused by the Julia compiler and the following machine code generation done by LLVM.

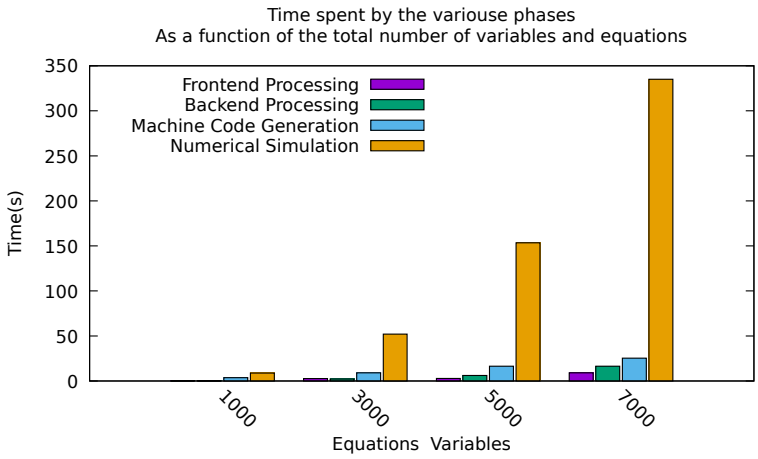

Figure 12. *Cont.*

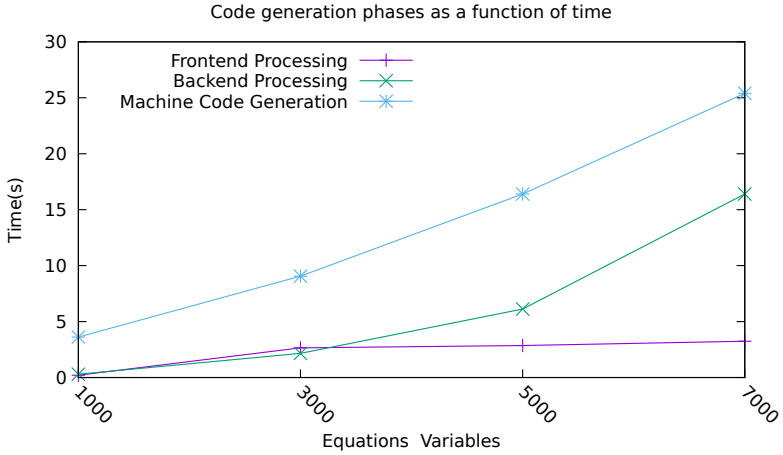

Figure 12. Median time spent in seconds in frontend, backend and code generation phases when simulating the model in Listing 16.

8.3. Summary

To summarize, our experiments indicate that the compiler can process medium to large models. As can be seen in Figure 11, the compilation performance is not yet on par with OMC. We believe that with tuning of the automatically translated frontend and with further work on OMBackend to be able to further improve the performance of our framework.

In Figure 12, we can see that the total compilation time is only a fraction of the total time spent when simulating this model. Furthermore, we can see that the process of translating Modelica to Julia code is only a small fraction of the total compilation time. The main bottleneck is compilation time and machine code generation to LLVM by Julia. Comparing the results, it can be seen that the feasibility of runtime compilation depends on how often the system undergoes structural changes. This experiment shows that a system that undergoes such changes every other time step would suffer from an extensive overhead caused by excessive recompilation. This issue, however, could be mitigated by relying on interpretation instead of machine code generation. The Julia compiler supports the *@interpret* command, combining the two approaches in such a way that for large systems with few changes, machine code is generated, and utilizing interpretation for the smaller system would be a possible technique to improve the simulation time performance of smaller systems. However, heuristics need to be developed to decide when to generate machine code and when to interpret the system under simulation.

9. Discussion

With the empirical results from our experiments presented in Section 8, we have demonstrated the following:

- Automatically translating MetaModelica to Julia is possible.
- A Modelica compiler written in the Julia language is possible, and can have reasonable performance.
- The Modelica language can be extended to simulate variable structured systems (VSS) with minor changes.
- The advantage concerning the expressive power of a modeling language when blurring the lines between compilation, modeling and simulation.

9.1. Comparison To Related Work

As discussed in Section 3.4 there are other frameworks capable of handling equation-based models with variable structure. However, none of these frameworks can handle standard Modelica to the same extent, and none of these frameworks can simulate variable Modelica models of the same size and scale. Furthermore, we have provided experimental

data on the practical performance of our framework in comparison to an existing state-of-the-art framework; see Section 8.

9.2. Future Work

While our proposed compiler's frontend can handle a subset of the MSL, full backend support for all needed Modelica constructs is not yet implemented. One recommendation for future research would be to investigate the performance of simulating such models in practice.

Still, while we have presented a Modelica compiler capable of JIT compilation and proposed extensions to the Modelica language, several aspects could be improved upon. Currently, the initialization of the new system during a structural change is based on the previous value of the simulation before the structural change. However, initialization might lead to errors for some transitions; hence, syntax and semantics for safer structural transitions should be investigated. One possibility would be to incorporate and investigate the techniques for safe re-initialization proposed by Benveniste et al. [17]. These techniques have been applied for Modelica [18], however, not in combination with JIT compilation. We are currently examining extending our naive re-initialization algorithm presented here in this regard.

During our experiments, we investigated the cost of JIT compilation when simulating a large system with a computationally expensive integration algorithm. However, for smaller models, it might be the case that compilation is more expensive than simulation. For such cases, it would be beneficial to use the interpreter to improve the overall simulation and compilation efficiency. Developing effective heuristics for equation-based models to efficiently select between interpretation and compilation during simulation should be investigated.

Furthermore, the performance of the final generated Julia code can be improved both in terms of compilation time and simulation time. One suggestion would be for MTK to introduce *descalarization* or avoid *scalarization* during symbolic processing. Recent techniques for unscalarized processing are described in [48]. Another alternative is MTK support for *DAE-Mode*, as presented in [43,49].

By enabling support for VSS, we provided the Modelica language with increased expressiveness. However, this also enables the possibility of formulating Modelica models more efficiently by using techniques from *Model Reduction* [8] applied to the context of object-oriented modeling languages [50]. One possibility could be to formulate an initial model with detailed granularity. However, if necessary conditions are met during the simulation, the model could simplify itself via the dynamic recompilation constructs presented in Section 6 and remove no longer needed equations. One such example is the engine model by Zimmer [29] illustrated in Listing 4. In the context of object-oriented modeling, Mikelsons et al. applied symbolic reduction techniques to optimize the simulation of a construction machine [9]. Furthermore, the effectiveness of model reduction has been exemplified by Davoudi et al. [10]. However, the practical implications of using model reduction on a large scale in a Modelica compiler have to our knowledge not been investigated in detail. We believe that the work presented could be extended to investigate model reductions techniques during simulations. Hence, further research is needed to fully understand the implications of VSS modeling in the context of model reduction.

Another current area of research in modeling and simulation is the use of machine learning techniques to speed up simulations. Similarly, to model reduction, the accuracy of the numerical simulation is reduced in order to simulate systems or parts of systems faster. A contemporary example of this technique in the context of equation-based languages is [13]. Bruder and Mikelsons [12] have also demonstrated the benefits of this technique.

In their paper, Bruder and Mikelsons stated that current Modelica compilers do not expose their AST, hindering the adaption of such techniques. Our framework, however, does expose the AST, and we have done some initial experiments concerning the replacement of algebraic loops [6]. An avenue for future research would be to use this framework to

compare the benefits and drawbacks of scientific machine learning, with related techniques, such as model reduction, on existing industrial grade models.

9.3. Conclusions

In this article, we have presented OpenModelica.jl, a novel compiler framework targeting the equation-based language Modelica. We have argued how the framework is extensible by providing a new backend, and composable by integrating several Julia libraries, such as *Graphs.jl* (https://github.com/JuliaGraphs/Graphs.jl/, accessed on 9 May 2022) and *Plots.jl* [44] to provide the capability of plotting solutions and interact with the wider Julia ecosystem. Such a framework can open new possibilities for pre/post processing of Modelica models and simulation results not existing in any other Modelica tools.

We have demonstrated that this compiler can be used as a basis to provide support for variable structure systems. Moreover, we have performed experiments comparing the performance characteristics of the framework presented here with that of an existing state-of-the-art Modelica compiler. The experiments showed that while OMC is currently more efficient at compiling Modelica models, the frontend presented here can handle models of a similar size and scale. We believe that modifications to the automatically translated frontend and incorporating techniques from Section 9.2 in the backend will improve our framework's performance. To the best of our knowledge, this is the first standard-compliant Modelica compiler written in Julia that is capable of handling parts of the MSL and variable structure systems using JIT compilation as the main technique.

Author Contributions: Conceptualization, J.T.; methodology, J.T.; software, J.T. and A.P.; validation, J.T.; formal analysis, J.T.; investigation, J.T.; writing—original draft preparation, J.T.; writing—review and editing, J.T., A.P. and M.S.; supervision, A.P. and M.S.; project administration, A.P. and M.S.; funding acquisition, A.P. All authors have read and agreed to the published version of the manuscript.

Funding: This work has been supported by the Swedish Government via the ELLIIT project and by Vinnova via the ITEA3 EMBRACE project. Support has also been received from the Swedish Strategic Research foundation (SSF) via the LargeDyn project.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Acknowledgments: The OpenModelica development is supported by the Open Source Modelica Consortium.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

AST	Abstract Syntax Tree
CPS	Cyber Physical System
DAE	Differential Algebraic Equations
EOOL	Equation-Based Object-Oriented Modeling Language
HIR	High-level Intermediate Representation
JIT	Just-In-Time
LIR	Low-Level Intermediate Representation
MIR	Mid-level Intermediate Representation
MSL	Modelica Standard Library
MTK	ModelingToolkit
OMC	The OpenModelica Compiler
VSS	Variable Structure Systems

Appendix A

Appendix A.1. Program to Simulate the Simple Two Modes Model

Listing A1. A program to simulate and plot *SimpleTwoModes* from Listing 12 using OpenModelica.jl with associated modules.

```
using Revise
import Absyn
import DAE
import OM
import OMBackend
import OMFrontend
import SCode
using MetaModelica
using Plots
function runModelMTK(model,
                     file;
                     timeSpan = (0.0, 1.0))
  @info "Running : " model
  @time OM.simulate(model,
                    file,
                    mode = OMBackend.MTK_MODE,
                    startTime = first(timeSpan),
                    stopTime = last(timeSpan))
end
res = runModelMTK("SimpleTwoModes",
                  "./Models/SimpleTwoModes.mo";
                  timeSpan=(0.0, 1.0))
p = plot(res; legend = :topleft)
Plots.pdf(p,
          "./Plots/SimpleTwoModesPlot")
```

Appendix A.2. Program to Simulate the Implicit Breaking Pendulum Model

Listing A2. Program to simulate the implicit breaking pendulum model, *ArrayGrow* and *ArrayShrink* models.

```
using Revise

import Absyn
import DAE
import OM
import OMBackend
import OMFrontend
import SCode
using DifferentialEquations
using MetaModelica
using Plots

function runModelMTK(model, file; timeSpan = (0.0, 1.0))
  @info "Running : " model
  @time OM.simulate(model, file,
                    mode = OMBackend.MTK_MODE,
```

```
                    startTime = first(timeSpan),
                    stopTime = last(timeSpan),
                    solver = :(Rodas5()))
end

function plotCombined(res, name, limX, limY)
  #= Plot array grow=#
  p1 = plot(res[1]; legend = false)
  p2 = plot(res[2]; legend = false)
  p3 = plot(p1, p2)
  #= Plot array grow change from 10 to 15 equations =#
  Plots.pdf(p3, "./Plots/$name")
  #= Construct a merged plot =#
  p1 = plot(res[1]; legend = false, xlim=limX, ylim = limY)
  p2 = plot!(res[2]; legend = false, xlim=limX, ylim = limY)
  Plots.pdf(p2, "./Plots/$(name)SinglePlot")
end

function plotPendulum(res, name, limX, limY)
  #= Plot array grow=#
  p1 = plot(res[1]; legend = true)
  p2 = plot(res[2]; legend = true)
  p3 = plot(p1, p2)
  #= Plot array grow change from 10 to 15 equations =#
  Plots.pdf(p3, "./Plots/$name")
  #= Construct a merged plot =#
  p1 = plot(res[1]; legend = :bottomleft, xlim=limX, ylim = limY,
   ↪  vars = [(0,3)])
  p2 = plot!(res[2]; legend = :bottomleft, xlim=limX, ylim = limY,
   ↪  vars = [(0,2)])
  Plots.pdf(p2, "./Plots/$(name)SinglePlot")
end

res = runModelMTK("BreakingPendulum",
                  "./Models/BreakingPendulumRecompilation.mo";
                   ↪  timeSpan=(0.0, 7.0))
plotPendulum(res, "BreakingPendulum", (0.0, 7.0), (-10, 10.0))

res = runModelMTK("ArrayGrow", "./Models/ArrayGrow.mo";
 ↪  timeSpan=(0.0, 1.0))
plotCombined(res, "ArrayGrow", (0.0, 1.0), (0.0, 20))
res = runModelMTK("ArrayShrink", "./Models/ArrayShrink.mo";
 ↪  timeSpan=(0.0, 1.0))
plotCombined(res, "ArrayShrink", (0.0, 1.0), (0.0, 20))
```

Appendix A.3. The Flat Model of the RLC Circuit Produced by OpenModelica.jl When Using the Modelica Standard Library

Listing A3. The flat Modelica model of the RLC Circuit.

```
class ElectricalTest.RLCCircuit
  parameter Real R1.R(start = 1.0, unit = "Ohm", quantity = "Resistance");
  parameter Real R1.T_ref(nominal = 300.0, start = 288.15, min = 0.0, displayUnit
   ↪  = "degC",
```

```
unit = "K", quantity = "ThermodynamicTemperature") = 300.15;
parameter Real R1.alpha(unit = "1/K", quantity = "LinearTemperatureCoefficient")
 ↪  = 0.0;
Real R1.v(unit = "V", quantity = "ElectricPotential");
Real R1.i(unit = "A", quantity = "ElectricCurrent");
Real R1.p.v(unit = "V", quantity = "ElectricPotential");
flow Real R1.p.i(unit = "A", quantity = "ElectricCurrent");
Real R1.n.v(unit = "V", quantity = "ElectricPotential");
flow Real R1.n.i(unit = "A", quantity = "ElectricCurrent");
parameter Boolean R1.useHeatPort = false;
parameter Real R1.T(nominal = 300.0, start = 288.15, min = 0.0, displayUnit =
 ↪  "degC",
unit = "K", quantity = "ThermodynamicTemperature") = R1.T_ref;
Real R1.LossPower(unit = "W", quantity = "Power");
Real R1.T_heatPort(nominal = 300.0, start = 288.15, min = 0.0, displayUnit =
 ↪  "degC",
unit = "K", quantity = "ThermodynamicTemperature");
Real R1.R_actual(unit = "Ohm", quantity = "Resistance");
Real C.v(start = 0.0, unit = "V", quantity = "ElectricPotential");
Real C.i(unit = "A", quantity = "ElectricCurrent");
Real C.p.v(unit = "V", quantity = "ElectricPotential");
flow Real C.p.i(unit = "A", quantity = "ElectricCurrent");
Real C.n.v(unit = "V", quantity = "ElectricPotential");
flow Real C.n.i(unit = "A", quantity = "ElectricCurrent");
parameter Real C.C(start = 1.0, min = 0.0, unit = "F", quantity =
 ↪  "Capacitance");
parameter Real R2.R(start = 1.0, unit = "Ohm", quantity = "Resistance");
parameter Real R2.T_ref(nominal = 300.0, start = 288.15, min = 0.0, displayUnit
 ↪  = "degC",
unit = "K", quantity = "ThermodynamicTemperature") = 300.15;
parameter Real R2.alpha(unit = "1/K", quantity = "LinearTemperatureCoefficient")
 ↪  = 0.0;
Real R2.v(unit = "V", quantity = "ElectricPotential");
Real R2.i(unit = "A", quantity = "ElectricCurrent");
Real R2.p.v(unit = "V", quantity = "ElectricPotential");
flow Real R2.p.i(unit = "A", quantity = "ElectricCurrent");
Real R2.n.v(unit = "V", quantity = "ElectricPotential");
flow Real R2.n.i(unit = "A", quantity = "ElectricCurrent");
parameter Boolean R2.useHeatPort = false;
parameter Real R2.T(nominal = 300.0, start = 288.15, min = 0.0, displayUnit =
 ↪  "degC",
unit = "K", quantity = "ThermodynamicTemperature") = R2.T_ref;
Real R2.LossPower(unit = "W", quantity = "Power");
Real R2.T_heatPort(nominal = 300.0, start = 288.15, min = 0.0, displayUnit =
 ↪  "degC",
unit = "K", quantity = "ThermodynamicTemperature");
Real R2.R_actual(unit = "Ohm", quantity = "Resistance");
Real L.v(unit = "V", quantity = "ElectricPotential");
Real L.i(start = 0.0, unit = "A", quantity = "ElectricCurrent");
Real L.p.v(unit = "V", quantity = "ElectricPotential");
flow Real L.p.i(unit = "A", quantity = "ElectricCurrent");
Real L.n.v(unit = "V", quantity = "ElectricPotential");
flow Real L.n.i(unit = "A", quantity = "ElectricCurrent");
parameter Real L.L(start = 1.0, unit = "H", quantity = "Inductance");
Real G.p.v(unit = "V", quantity = "ElectricPotential");
flow Real G.p.i(unit = "A", quantity = "ElectricCurrent");
parameter Real AC.V(start = 1.0, unit = "V", quantity = "ElectricPotential");
parameter Real AC.phase(displayUnit = "deg", unit = "rad", quantity = "Angle") =
 ↪  0.0;
parameter Real AC.freqHz(start = 1.0, unit = "Hz", quantity = "Frequency");
Real AC.v(unit = "V", quantity = "ElectricPotential");
Real AC.i(unit = "A", quantity = "ElectricCurrent");
Real AC.p.v(unit = "V", quantity = "ElectricPotential");
flow Real AC.p.i(unit = "A", quantity = "ElectricCurrent");
Real AC.n.v(unit = "V", quantity = "ElectricPotential");
flow Real AC.n.i(unit = "A", quantity = "ElectricCurrent");
parameter Real AC.offset(unit = "V", quantity = "ElectricPotential") = 0.0;
parameter Real AC.startTime(unit = "s", quantity = "Time") = 0.0;
parameter Real AC.signalSource.amplitude = AC.V;
```

```
  parameter Real AC.signalSource.freqHz(start = 1.0,
  unit = "Hz", quantity = "Frequency") = AC.freqHz;
  parameter Real AC.signalSource.phase(displayUnit = "deg", unit = "rad",
  quantity = "Angle") = AC.phase;
  final parameter Real AC.signalSource.offset = AC.offset;
  final parameter Real AC.signalSource.startTime(unit = "s", quantity = "Time") =
   ↪  AC.startTime;
  output Real AC.signalSource.y;
  protected constant Real AC.signalSource.pi = Modelica.Constants.pi;
  equation
  R2.p.v = AC.p.v;
    R2.p.v = R1.p.v;
    C.p.v = R1.n.v;
    R2.n.v = L.p.v;
    AC.n.v = C.n.v;
    AC.n.v = L.n.v;
    AC.n.v = G.p.v;
    R1.n.i + C.p.i = 0.0;
    L.p.i + R2.n.i = 0.0;
    G.p.i + C.n.i + L.n.i + AC.n.i = 0.0;
    R1.p.i + R2.p.i + AC.p.i = 0.0;
    assert(1.0 + R1.alpha * (R1.T_heatPort - R1.T_ref) >= Modelica.Constants.eps,
    "Temperature outside scope of model!", AssertionLevel.error);
    R1.R_actual = R1.R * (1.0 + R1.alpha * (R1.T_heatPort - R1.T_ref));
    R1.v = R1.R_actual * R1.i;
    R1.LossPower = R1.v * R1.i;
    R1.T_heatPort = R1.T;
    R1.v = R1.p.v - R1.n.v;
    0.0 = R1.p.i + R1.n.i;
    R1.i = R1.p.i;
    C.i = C.C * der(C.v);
    C.v = C.p.v - C.n.v;
    0.0 = C.p.i + C.n.i;
    C.i = C.p.i;
    assert(1.0 + R2.alpha * (R2.T_heatPort - R2.T_ref) >= Modelica.Constants.eps,
    "Temperature outside scope of model!", AssertionLevel.error);
    R2.R_actual = R2.R * (1.0 + R2.alpha * (R2.T_heatPort - R2.T_ref));
    R2.v = R2.R_actual * R2.i;
    R2.LossPower = R2.v * R2.i;
    R2.T_heatPort = R2.T;
    R2.v = R2.p.v - R2.n.v;
    0.0 = R2.p.i + R2.n.i;
    R2.i = R2.p.i;
    L.L * der(L.i) = L.v;
    L.v = L.p.v - L.n.v;
    0.0 = L.p.i + L.n.i;
    L.i = L.p.i;
    G.p.v = 0.0;
    AC.signalSource.y = AC.signalSource.offset + (if time <
     ↪  AC.signalSource.startTime then 0.0
                                                  else
                                                    AC.signalSource.amplitude *
                                                     ↪  sin(2.0 *
                                                     ↪  AC.signalSource.pi *
                                                     ↪  AC.signalSource.freqHz *
                                                     ↪  (time -
                                                     ↪  AC.signalSource.startTime)
                                                     ↪  + AC.signalSource.phase));
    AC.v = AC.signalSource.y;
    AC.v = AC.p.v - AC.n.v;
    0.0 = AC.p.i + AC.n.i;
    AC.i = AC.p.i;
end ElectricalTest.RLCCircuit;
```

Appendix A.4. The Cascading First Order System

Listing A4. The scalable Cascading first Order system from the ScalableTestSuite library.

```
package CascadingFirstOrder

  model Casc
    parameter Integer N = 100 "Order of the system";
    final parameter Real tau = T/N "Individual time constant";
    parameter Real T = 1 "System delay";
    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 Casc;

  model Casc10
    Casc(N = 10);
  end Casc10;

  model Casc100
    Casc(N = 100);
  end Casc100;

  model Casc200
    Casc(N = 200);
  end Casc200;

  model Casc400
    Casc(N = 400);
  end Casc400;

  model Casc800
    Casc(N = 800);
  end Casc800;
...
end CascadingFirstOrder;
```

Appendix A.5. The Transmission Line Model

Listing A5. A Modelica model representing an electrical transmission line.

```
// Transmission line model from the Scalable testsuite by Francesco Casella
 ↪  Politecnico Milano
model TransmissionLine "Modular model of an electrical transmission line"
  import Modelica.SIunits;
  import Modelica.Electrical.Analog;
  SIunits.Voltage vpg "voltage of pin p of the transmission line";
  SIunits.Voltage vng "voltage of pin n of the transmission line";
  SIunits.Current ipin_p
    "current flows through pin p of the transmission line";
  SIunits.Current ipin_n
    "current flows through pin n of the transmission line";
  Analog.Interfaces.Pin pin_p;
  Analog.Interfaces.Pin pin_n;
  Analog.Interfaces.Pin pin_ground "pin of the ground";
  Analog.Basic.Ground ground "ground of the transmission line";
  parameter Integer N = 1 "number of segments";
  parameter Real r "resistance per meter";
  parameter Real l "inductance per meter";
  parameter Real c "capacitance per meter";
  parameter Real length "length of tranmission line";
  Analog.Basic.Inductor L[N](L = fill(l * length / N, N)) "N inductors";
```

```
  Analog.Basic.Capacitor C[N](C = fill(c * length / N, N)) "N capacitors";
  Analog.Basic.Resistor R[N](R = fill(r * length / N, N)) "N resistors";
initial equation
  for i in 1:N loop
    C[i].v = 0;
    L[i].i = 0;
  end for;
equation
  vpg = pin_p.v - pin_ground.v;
  vng = pin_n.v - pin_ground.v;
  ipin_p = pin_p.i;
  ipin_n = pin_n.i;
  connect(pin_p, R[1].p);
  for i in 1:N loop
    connect(R[i].n, L[i].p);
    connect(C[i].p, L[i].n);
    connect(C[i].n, pin_ground);
  end for;
  for i in 1:N - 1 loop
    connect(L[i].n, R[i + 1].p);
  end for;
  connect(L[N].n, pin_n);
  connect(pin_ground, ground.p);
end TransmissionLine;
```

References

1. Bezanson, J.; Edelman, A.; Karpinski, S.; Shah, V.B. Julia: A fresh approach to numerical computing. *SIAM Rev.* **2017**, *59*, 65–98. [CrossRef]
2. Elmqvist, H.; Otter, M. Innovations for future Modelica. In Proceedings of the 12th International Modelica Conference, Prague, Czech, 15–17 May 2017; Linköping University Electronic Press: Linköping, Sweden, 2017. [CrossRef]
3. Ma, Y.; Gowda, S.; Anantharaman, R.; Laughman, C.; Shah, V.; Rackauckas, C. ModelingToolkit: A Composable Graph Transformation System For Equation-Based Modeling. *arXiv* **2021**. [CrossRef]
4. Fritzson, P.; Pop, A.; Abdelhak, K.; Asghar, A.; Bachmann, B.; Braun, W.; Bouskela, D.; Braun, R.; Buffoni, L.; Casella, F.; et al. The OpenModelica integrated environment for modeling, simulation, and model-based development. *Model. Identif. Control* **2020**, *41*, 241–295. [CrossRef]
5. Tinnerholm, J.; Pop, A.; Sjölund, M.; Heuermann, A.; Abdelhak, K. Towards an Open-Source Modelica Compiler in Julia. In Proceedings of the Asian Modelica Conference, Tokyo, Japan, 8–9 October 2020. [CrossRef]
6. Tinnerholm, J.; Pop, A.; Heuermann, A.; Sjölund, M. OpenModelica.jl: A modular and extensible Modelica compiler framework in Julia targeting ModelingToolkit.jl. In Proceedings of the 14th International Modelica Conference, Linköping, Sweden, 20–24 September 2021; pp. 109–117. [CrossRef]
7. Pop, A.; Östlund, P.; Casella, F.; Sjölund, M.; Franke, R. A new openmodelica compiler high performance frontend. In Proceedings of the 13th International Modelica Conference, Regensburg, Germany, 4–6 March 2019; pp. 689–698. [CrossRef]
8. Antoulas, A.C.; Sorensen, D.C.; Gugercin, S. *A Survey of Model Reduction Methods for Large-Scale Systems*; Technical report; RICE University: Houston, TX, USA, 2000.
9. Mikelsons, L.; Ji, H.; Brandt, T.; Lenord, O. Symbolic model reduction applied to realtime simulation of a construction Machine. In Proceedings of the 7th International Modelica Conference, Como, Italy, 20–22 September 2009; Casella, F., Ed.; Linköping University Electronic Press: Linköping, Sweden, 2009. [CrossRef]
10. Davoudi, F.; Lenord, O.; Worschech, N.; Durak, U.; Hartmann, S. Redesign and evaluation of an equation-based model reduction method in OpenModelica. *Int. J. Eng. Syst. Model. Simul.* **2019**, *11*, 91–101. [CrossRef]
11. Rai, R.; Sahu, C.K. Driven by data or derived through physics? a review of hybrid physics guided machine learning techniques with cyber-physical system (cps) focus. *IEEE Access* **2020**, *8*, 71050–71073. [CrossRef]
12. Bruder, F.; Mikelsons, L. Modia and Julia for Grey Box Modeling. In Proceedings of the 14th International Modelica Conference, Linköping, Sweden, 20–24 September 2021; pp. 87–95. [CrossRef]
13. Rackauckas, C.; Anantharaman, R.; Edelman, A.; Gowda, S.; Gwozdz, M.; Jain, A.; Laughman, C.; Ma, Y.; Martinuzzi, F.; Pal, A.; et al. Composing Modeling and Simulation with Machine Learning in Julia. In Proceedings of the 14th International Modelica Conference, Linköping, Sweden, 20–24 September 2021; pp. 97–107. [CrossRef]
14. Minhas, R.; De Kleer, J.; Matei, I.; Saha, B.; Janssen, B.; Bobrow, D.G.; Kurtoglu, T. Using Fault Augmented Modelica Models for Diagnostics. In Proceedings of the 10th International Modelica Conference, Lund, Sweden, 10–12 March 2014; pp. 437–445. [CrossRef]

15. Bäck, O. *Modelling for Diagnosis in Modelica: Implementation and Analysis*; Institutionen för Systemteknik: Linköping, Sweden, 2008; p. 65.
16. Benveniste, A.; Caillaud, B.; Elmqvist, H.; Ghorbal, K.; Otter, M.; Pouzet, M. Multi-Mode DAE models-challenges, theory and implementation. In *Computing and Software Science*; Springer: Berlin/Heidelberg, Germany, 2019; pp. 283–310. [CrossRef]
17. Benveniste, A.; Caillaud, B.; Malandain, M. The mathematical foundations of physical systems modeling languages. *Annu. Rev. Control* **2020**, *50*, 72–118. [CrossRef]
18. Benveniste, A.; Caillaud, B.; Malandain, M. Handling Multimode Models and Mode Changes in Modelica. In Proceedings of the 14th International Modelica Conference, Linköping, Sweden, 20–24 September 2021; pp. 507–517. [CrossRef]
19. Casella, F. Simulation of Large-Scale Models in Modelica: State of the Art and Future Perspectives. In Proceedings of the 11th International Modelica Conference, Palais des Congrès de Versailles, France, 21–23 September 2015; pp. 459–468. [CrossRef]
20. Broman, D. Interactive Programmatic Modeling. *ACM Trans. Embed. Comput. Syst. (TECS)* **2021**, *20*, 1–26. [CrossRef]
21. Peffers, K.; Tuunanen, T.; Rothenberger, M.A.; Chatterjee, S. A design science research methodology for information systems research. *J. Manag. Inf. Syst.* **2007**, *24*, 45–77. [CrossRef]
22. Tinnerholm, J.; Sjölund, M.; Pop, A. Towards introducing just-in-time compilation in a Modelica compiler. In Proceedings of the 9th International Workshop on Equation-based Object-oriented Modeling Languages and Tools, Berlin, Germany, 5 November 2019; pp. 11–19. [CrossRef]
23. Fritzson, P. *Principles of Object-oriented Modeling and Simulation with Modelica 3.3: A Cyber-Physical Approach*; John Wiley & Sons: Hoboken, NJ, USA, 2014.
24. Pop, A.; Fritzson, P. MetaModelica: A Unified Equation-Based Semantical and Mathematical Modeling Language. In Proceedings of the Joint Modular Languages Conference, Oxford, UK, 13–15 September 2006; Springer: Berlin/Heidelberg, Germany, 2006; pp. 211–229. [CrossRef]
25. Rackauckas, C.; Nie, Q. DifferentialEquations.jl - A Performant and Feature-Rich Ecosystem for Solving Differential Equations in Julia. *J. Open Res. Softw.* **2017**, *5*, 15. [CrossRef]
26. Elmqvist, H.; Otter, M.; Neumayr, A.; Hippmann, G. Modia-Equation Based Modeling and Domain Specific Algorithms. In Proceedings of the 14th International Modelica Conference, Linköping, Sweden, 20–24 September 2021; pp. 73–86. [CrossRef]
27. Cellier, F.E.; Kofman, E. *Continuous System Simulation*; Springer Science & Business Media: New York, NY, USA, 2006.
28. Nytsch-Geusen, C.; Ernst, T.; Nordwig, A.; Schneider, P.; Schwarz, P.; Vetter, M.; Wittwer, C.; Holm, A.; Nouidui, T.; Leopold, J.; et al. MOSILAB: Development of a Modelica based generic simulation tool supporting model structural dynamics. In Proceedings of the 4th International Modelica Conference, Hamburg, Germany, 7–8 March 2005.
29. Zimmer, D. Equation-Based Modeling of Variable-Structure Systems. Ph.D. Thesis, ETH Zürich, Zurich, Switzerland, 2010. [CrossRef]
30. Giorgidze, G. First-Class Models: On A Noncausal Language for Higher-Order and Structurally Dynamic Modelling and Simulation. Ph.D. Thesis, University of Nottingham, Nottingham, UK, 2012.
31. Nilsson, H.; Peterson, J.; Hudak, P. Functional hybrid modeling. In Proceedings of the International Symposium on Practical Aspects of Declarative Languages, New Orleans, LA, USA, 13–14 January 2003; Springer: Berlin/Heidelberg, Germany, 2003; pp. 376–390.
32. Höger, C. Compiling Modelica : About the Separate Translation of Models from Modelica to OCaml and Its Impact on Variable-Structure Modeling. Doctoral Thesis, Technische Universität Berlin, Berlin, Germany, 2019. [CrossRef]
33. Broman, D. Meta-Languages and Semantics for Equation-Based Modeling and Simulation. Ph.D. Thesis, Linköping University Electronic Press, Linköping, Sweden, 2010.
34. Elmqvist, H.; Matsson, S.E.; Otter, M. Modelica extensions for multi-mode DAE systems. In Proceedings of the 10th International Modelica Conference, Lund, Sweden, 10–12 March 2014; Linköping University Electronic Press: Linköping, Sweden, 2014; pp. 183–193. [CrossRef]
35. Mehlhase, A. A Python framework to create and simulate models with variable structure in common simulation environments. *Math. Comput. Model. Dyn. Syst.* **2014**, *20*, 566–583. [CrossRef]
36. Lattner, C.; Adve, V. LLVM: A compilation framework for lifelong program analysis & transformation. In Proceedings of the International Symposium on Code Generation and Optimizatio, CGO, San Jose, CA, USA, 20–24 March 2004; IEEE: Piscataway, NJ, USA, 2004; pp. 75–86. [CrossRef]
37. Nystrom, N.; Clarkson, M.R.; Myers, A.C. Polyglot: An extensible compiler framework for Java. In Proceedings of the International Conference on Compiler Construction, Warsaw, Poland, 7–11 April 2003; Springer: Berlin/Heidelberg, Germany, 2003; pp. 138–152. [CrossRef]
38. Lee, S.; Min, S.J.; Eigenmann, R. OpenMP to GPGPU: A compiler framework for automatic translation and optimization. *ACM Sigplan Not.* **2009**, *44*, 101–110. [CrossRef]
39. Fritzson, P.; Pop, A.; Sjölund, M. *Towards Modelica 4 Meta-Programming and Language Modeling with MetaModelica 2.0*; Technical Report 2011:10; Linköping University, PELAB–Programming Environment Laboratory: Linköping, Sweden, 2011.
40. Fritzson, P.; Pop, A.; Sjölund, M.; Asghar, A. MetaModelica—A Symbolic-Numeric Modelica Language and Comparison to Julia. In Proceedings of the 13th International Modelica Conference, Regensburg, Germany, 4–6 March 2019. [CrossRef]
41. Parr, T.J.; Quong, R.W. ANTLR: A predicated-LL (k) parser generator. *Softw. Pract. Exp.* **1995**, *25*, 789–810. [CrossRef]

42. Hindmarsh, A.C.; Brown, P.N.; Grant, K.E.; Lee, S.L.; Serban, R.; Shumaker, D.E.; Woodward, C.S. SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. *ACM Trans. Math. Softw. (TOMS)* **2005**, *31*, 363–396. [CrossRef]
43. Braun, W.; Casella, F.; Bachmann, B. Solving large-scale Modelica models: New approaches and experimental results using OpenModelica. In Proceedings of the 12th International Modelica Conference, Prague, Czech, 15–17 May 2017; Linköping University Electronic Press: Linköping, Sweden, 2017; pp. 557–563. [CrossRef]
44. Christ, S.; Schwabeneder, D.; Rackauckas, C. Plots.jl—A user extendable plotting API for the julia programming language. *arXiv* **2022**, arXiv:2204.08775.
45. Muchnick, S. *Advanced Compiler Design & Implementation*; Morgan Kaufmann: San Mateo, CA, USA, 1997.
46. Tsitouras, C. Runge–Kutta pairs of order 5 (4) satisfying only the first column simplifying assumption. *Comput. Math. Appl.* **2011**, *62*, 770–775. [CrossRef]
47. Chen, J.; Revels, J. Robust benchmarking in noisy environments. *arXiv* **2016**, arXiv:1608.04295.
48. Marzorati, D.; Fernández, J.; Kofman, E. Efficient connection processing in equation–based object–oriented models. *Appl. Math. Comput.* **2022**, *418*, 126842. [CrossRef]
49. Henningsson, E.; Olsson, H.; Vanfretti, L. DAE Solvers for Large-Scale Hybrid Models. In Proceedings of the 13th International Modelica Conference, Regensburg, Germany, 4–6 March 2019. [CrossRef]
50. Donida, F.; Casella, F.; Ferretti, G. Model order reduction for object-oriented models: A control systems perspective. *Math. Comput. Model. Dyn. Syst.* **2010**, *16*, 269–284. [CrossRef]