Towards an Open-Source Modelica Compiler in Julia

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

[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

Recently the Julia language has become an option for scientific computing. As of 2020, efforts exist to provide libraries that emulate the equation-based modeling features provided by Modelica or otherwise provide such functionality in Julia. The issue with these approaches is that investment in standardization and libraries would be lost unless standard-compliance is guaranteed. We believe that it is possible to combine features from both by implementing such a compiler in Julia. We argue that this approach would open additional opportunities. One such being the handling of variable structure systems (VSS) within the framework of a Modelica standard-compliant compiler. The other being a proposed compiler architecture reminiscent of LLVM for equation-based object-oriented languages. Using the OpenModelica Compiler as a baseline, we verified the fidelity of our implementation by simulating a selected set of models. While there are performance penalties, we argue that improvements to the frontend would mitigate these issues. *Keywords: Modelica, OpenModelica, Compilers, Applied computing, Julia, Variable Structure Systems*

1 Introduction

Cyber-physical Systems (CPS) are becoming increasingly complex every year. This trend increases the workload for modelers, and subsequently, requirements on associated tooling. Thus, tools need to scale and be flexible enough to handle increased complexity. These new requirements call for interdisciplinary cooperation between applied mathematics and computer science. One open problem in the area of modeling and simulation is how to handle specific categories of VSS (Utkin, 1977) in a way that adheres to the requirements of standard compliance and performance required by industry.

To our knowledge[1], Modelica only supports a limited subset of VSS through the usage of if-equations. Thus the set of VSS Modelica supports are limited to scenarios that can be described *a priori*. However, when modeling a system, it is not always possible to account for all changes with sufficient fidelity. An important but not a necessary component for a computational framework to support VSS is a JIT (Just-in-time) compiler.

[1]As of August 7, 2020

Tinnerholm (2019) demonstrated that there is room for improving both the compile-time and runtime performance of OMC (the OpenModelica Compiler) and the possibility of JIT integration. Due to the similarities between Julia and MetaModelica (Fritzson et al., 2019b), a MetaModelica to Julia translator was developed (Tinnerholm et al., 2019) to examine alternatives to achieve JIT functionality. One such option is to use LLVM (Lattner and Adve, 2004), which was also investigated in (Tinnerholm, 2019). The other option was instead to generate Julia code. While automatically generated, Julia code in some cases had subpar performance compared to generating the LLVM intermediate representation (IR)[2] directly, the ease of code generation and the possibility of software reuse was promising. The problem of providing a standard-compliant compiler to deal with complex future requirements from industry motivated our continued investigation of Modelica-Julia integration.

The structure of this paper is as follows: First, we present the background of VSS in section 2. This section is followed by a discussion on providing a standard-compliant computational framework for VSS support section 3. This is followed by a technical overview together with initial verification of our Julia based Modelica implementation in section 4 and section 5. Related work is presented in section 6 and conclusions followed by future research directions in section 7.

2 Variable Structure Systems

The definition of VSS varies in literature since it defines a rather large class of systems. When the term system is used to discuss situations in which a model has a separate mode of functionality, *Multi-Mode DAE Models* (Benveniste et al., 2019) is sometimes used as a more precise description. The term used in (Höger, 2017) names variable structured systems with possible infinite sets of modes *Implicit Hybrid Automata*. The rationale being that there exist both explicit hybrid systems, which reefers to systems where the variability is specified *a priori* explicitly. When this behavior is not specified, the variability is implicit. Another term that captures the dynamics of these systems is *Structurally Dynamic Systems* (Giorgidze and Nilsson, 2009).

[2]LLVM IR, the common currency used between components that constitute the LLVM Compiler Infrastructure.

In this paper, the term VSS is encompassing all classes of variable structured systems. We will refer to VSS with fixed variability (Pepper et al., 2011) as FVSS. It follows that support for general VSS includes FVSS. The compiler architecture we are proposing in this paper is meant to support the subset of VSS that may change without *a priori* information. We are thus attempting to enable future support for implicit hybrid automata through our proposed computational framework.

An early treatment of VSS was given by Utkin (Utkin, 1977). Utkin illustrates a simple variable structured system with the set of equations: eq. (1), eq. (2) and eq. (3).

$$\ddot{x} = -\psi x \tag{1}$$

$$\psi = \begin{cases} \alpha_1^2 \text{ where } x\dot{x} > 0 \\ \alpha_2^2 \text{ where } x\dot{x} < 0 \end{cases} \tag{2}$$

$$\alpha_1^2 > \alpha_2^2 \tag{3}$$

Systems such as these can be modeled in Modelica by the use of if-equations, see listing 1. However, although the structure varies, it is an FVSS. The behavior is statically encoded. Changes to the structure of the equations by means such as conditional declarations are, to our knowledge, not yet possible in standard Modelica (Zimmer, 2010; Höger, 2019).

Listing 1. The Modelica if-equation an example of a FVSS.

```
model M
  equation
  if <Condition> then
    <Equation 1>
  else
    <Equation 2>
  end if;
end M;
```

The handling of VSS is an active area of research and has been treated in (Neumayr and Otter, 2019; Elmqvist et al., 2017; Pepper et al., 2011; Höger, 2014; Mattsson et al., 2015; Höger, 2017; Giorgidze and Nilsson, 2009). A further summary is presented in section 6.

3 Towards a standard-compliant computational framework for Variable Structured System support

We have previously investigated the possibility of providing JIT support via the LLVM compiler infrastructure and compared that approach with using Julia generated code. In this section, we will discuss our initial findings when experimenting with LLVM in section 3.2. We will also present recent advances using the Julia programming language in section 3.3.

3.1 Architectural principles

There exists several computational frameworks to handle VSS (Zimmer, 2010; Höger, 2019; Elmqvist et al., 2017). While they differ somewhat in implementation, they all allow the compiler to be self-recursive, either through interpretation or JIT. In this section we briefly elaborate on two previously investigated alternatives, LLVM in section 3.2 and Julia in section 3.3

3.2 LLVM in the OpenModelica Compiler

LLVM (Lattner and Adve, 2004) is a compiler infrastructure currently employed both in the development of tools for static analysis and as a component for full-fledged compilers such as the Rust Compiler and Clang. The Julia project uses it for efficient code generation and JIT compilation. Recently, OpenModelica was extended with a new LLVM backend (OMLB) using an LLVM based JIT (Tinnerholm, 2019). While this effort did not cover all aspects of the Modelica language, it demonstrated possible benefits the OpenModelica environment would achieve by targeting LLVM, both in terms of compile-time and runtime performance.

3.3 The Julia language and extensions for equation-based modeling

The Julia language (Bezanson et al., 2012) is a new language for numerical computing. The Julia ecosystem supports many features required by a Modelica compiler, such as symbolic processing and interaction with other programming languages via its foreign function interface.

Modia is an example of a language extension to provide equation-based modeling in Julia to support VSS (Elmqvist et al., 2017). As noted by Fritzson et al. (2019b), an automatic translation of the OMC would provide the OpenModelica environment with these facilities without the need of re-implementing them. The disadvantage being that the development of MetaModelica would stall. Thus it would reduce the future capabilities of using Modelica to model the syntax and semantics of programming languages, a feature that is provided to Modelica with the MetaModelica extension (Fritzson et al., 2019a).

However, a Julia implementation would allow developers of OpenModelica access to the Julia ecosystem. We believe that such access would increase cooperation between the Modelica and Julia communities, aiding the Modelica effort by profiting from contributions to numerical computing from the Julia community and vice versa.

4 A Modelica frontend in Julia

The MetaModelica to Julia translator (Tinnerholm et al., 2019) was implemented in Susan (Fritzson et al., 2009). The translator (Tinnerholm et al., 2019) has been extended and has now been used to translate the oldest compiler frontend in OMC. In addition, a Julia based runtime has been developed along with a parser capable of parsing the

Modelica Standard Library[3].

The parser uses the ANTLR3[4] based Modelica parser from OpenModelica and the external foreign function interface (FFI) that Julia provides to create and call Julia values and functions from C. The Julia representation of the Modelica abstract syntax tree[5] can be instantiated and translated to the existing representation of differential-algebraic equations.

The runtime consists of preprocessing constructs to ease translation, using Julia MetaProgramming, which is further expanded upon in section 4.1. It also re-implements the external runtime library that is used in OMC.

4.1 MetaModelica constructs via Julia-MetaProgramming

When writing a translator, it is key to make sure that the translated code is still readable. It is also desirable to handle specific language constructs existing in the target language.

Thus, for such a venture to be successful, it is vital that the languages are similar in paradigm and possess the same constructs. While there are some differences between MetaModelica and Julia (Fritzson et al., 2019b), Julia possesses constructs to mitigate these issues. The Julia macro system provides the users with the necessary tools to introduce new syntactical constructs to the language during compilation time. This means that even though Julia does not support inheritance[6], it is possible to implement it.

This property simplifies the process of automatically translating from MetaModelica to Julia since less logic is needed in the translator. The reason is that we can extend the Julia language with MetaModelica specific constructs and thus avoid generating extra code during the translation process. An example of how metaprogramming is employed for these purposes can be seen in listing 2. The listing depicts a translation of MetaModelica uniontype inside OMC see listing 3.

4.2 Performance of the existing frontend compared with the translated

The work described in this paper is a capable, but still an experimental prototype. Thus performance is not yet on par with MetaModelica based compiler even if the functionality (for the frontend) is the same. Currently, the memory usage of the abstract data structures (ADS's) remains approximately the same. However, translation time

[3]Used MSL v3.2.3
On Github: github.com/modelica/ModelicaStandardLibrary

[4]ANTLR3: ANother Tool for Language Recognition
www.antlr3.org

[5]The abstract syntax tree in the OMC is a hybrid between a concrete and abstract syntax tree. Absyn preserves parts of the original textual representation, which the SCode-representation lack.

[6]As to our knowledge; Julia version v1.3.1 (Dec 30, 2019)

Listing 2. Julia representation of Modelica equations in the Absyn IR.

```
@Uniontype Equation begin
 @Record EQ_IF begin
   ifExp::Exp # conditional expression
   equationTrueItems::IList # then
   elseIfBranches::IList # elseif
   equationElseItems::IList # else
 end

 @Record EQ_EQUALS begin
   leftSide::Exp # left hand side
   rightSide::Exp # right hand side
 end
 ...
end
```

Listing 3. MetaModelica metamodel of equations in OMC

```
uniontype Equation
  record EQ_IF
    Exp ifExp "Conditional expression" ;
    list<EquationItem> equationTrueItems "
       true branch" ;
    list<tuple<Exp, list<EquationItem>>>
       elseIfBranches "elseIfBranches" ;
    list<EquationItem> equationElseItems "
       equationElseItems Standard 2-side
       eqn" ;
  end EQ_IF;

  record EQ_EQUALS
    Exp leftSide "leftSide" ;
    Exp rightSide "rightSide Connect stmt"
       ;
  end EQ_EQUALS;
...
end Equation
```

when translating models from Absyn to SCode[7] into what is sometimes called flat Modelica[8] is currently far from optimal.

The reason for performance issues is due to differences between the covariant type system of MetaModelica and the invariant type system of Julia. This difference results in redundant creation and deletion of abstract data structures such as *list* and *arrays*.

Another reason is due to exception handling. The reason being that certain parts of the control flow are directed by the *matchcontinue* construct (Pop et al., 2019). In the compiler runtime[9], this construct is implemented as a macro that wraps the match macro in an exception handling block.

In MetaModelica, this is implemented with

[7]The SCode preferred internal representation before DAE IR generation.

[8]Flat Modelica is an intermediate stage in the compilation process, before symbolic manipulation of the equation system occurs. It is a set of DAE's with additional hybrid language constructs also remaining.

[9]MetaModelica.jl

`setjmp`/`longjmp` and a custom pass that removes them in contexts where they are not needed, or replaces a `longjmp` with a `goto` when jumping in a local context. At the time of writing, these kinds of optimizations are not provided.

4.3 Verification of OMFrontend.jl

To verify the fidelity of our Julia-based translated frontend, we simulated three Modelica models using the same settings. The same models were then simulated using the original OpenModelica environment (using the existing frontend, which the translated Julia one was based on). The result of these experiments is elaborated on further in section 5.2. From these experiments, we have reasons to believe that the translated frontend behaves correctly. We have also manually compared the DAE produced by the original OMC frontend and the Julia based one for the small HelloWorld model.

5 OMCompiler.jl

In section 4, we discussed how a Modelica frontend in Julia by means of automatic translation was provided. We briefly discussed the current capabilities and reasoned about the fidelity to OMC, which will be further elaborated upon in this section. While the performance of the new Julia based frontend is not on par with the existing frontend, we believe that these issues are possible to mitigate when the issues we brought up in section 4 are addressed. In this section, we will present the new open-source Modelica compiler `OMCompiler.jl`[10]. It consists of the various components listed in table 1, one of these being the previously discussed `OMFrontend.jl`. It is also supported by additional components such as `MetaModelica.jl` [11], which provides the compiler runtime. An overview of the inter-dependencies between them is illustrated in fig. 1.

In addition to the work that is presented here, we have already started the process of separating the different components that make up the compiler into different packages. In that way, we hope that future users of this work are able to utilize parts of our compiler as software components tailored for their specific use case. Further implications of this component-based design are discussed in section 5.4. An example of how a separate package can be used to simulate the exported output of the compiler design proposed here is provided in section 5.3.

5.1 An initial Modelica backend in Julia, OMBackend.jl

The compiler design proposed has the implication that a possible backend can be developed and used in separation. To assess the initial simulation characteristics of our proposed framework, a proof of concept backend `OMBackend.jl`[12] was developed, which is capable of simulating rudimentary DAE's. In this section, we will give a short introduction to a typical Modelica backend. We will also present our initial experiments to verify the results of both frontend and backend operations in section 5.2.

A Modelica compiler performs the following four steps on a flattened DAE provided by the frontend (Cellier and Kofman, 2006).

1. Pre-Optimization
2. Causalization
3. Post-Optimization
4. Code generation

5.1.1 Layout of OMBackend.jl

The purpose of the backend is to transform a given IR into a form suitable for simulation. The procedure to transform the given DAE IR into this form consists of:

1. Lower frontend into backend DAE IR:
    - Translate equations and variables to backend structure.
    - Detect and mark state variables.
    - Convert equations to residual equations.
2. Lower backend to simulation code IR:
    - Order variables into three categories:
        1) Derivatives of states
        2) States and algebraic variables
        3) Time-invariant parameters and constants
    - Index variables and store them in a hash table.
3. Generate Julia code by targeting `DifferentialEquations.jl`

To verify the correctness of the DAE IR generated by the frontend, three examples are presented, which can be studied in sections 5.2.1 to 5.2.3. While Julia currently remains the sole target language, it is possible to generalize and aim for different languages such as C (necessary for FMI).

5.2 Verification

To verify the behavior of `OMFrontend.jl` and `OMBackend.jl`, three examples are provided. The simulations of these examples where run both using the Julia based Modelica Compiler, `OMCompiler.jl` and using OMC. The result of simulating them are presented in sections 5.2.1 to 5.2.3.

[10]OMCompiler.jl
On Github: https://github.com/JKRT/OMCompiler.jl

[11]MetaModelica.jl
On Github: https://github.com/OpenModelica/MetaModelica.jl

[12]OMBackend.jl
On Github: https://github.com/JKRT/OMBackend.jl

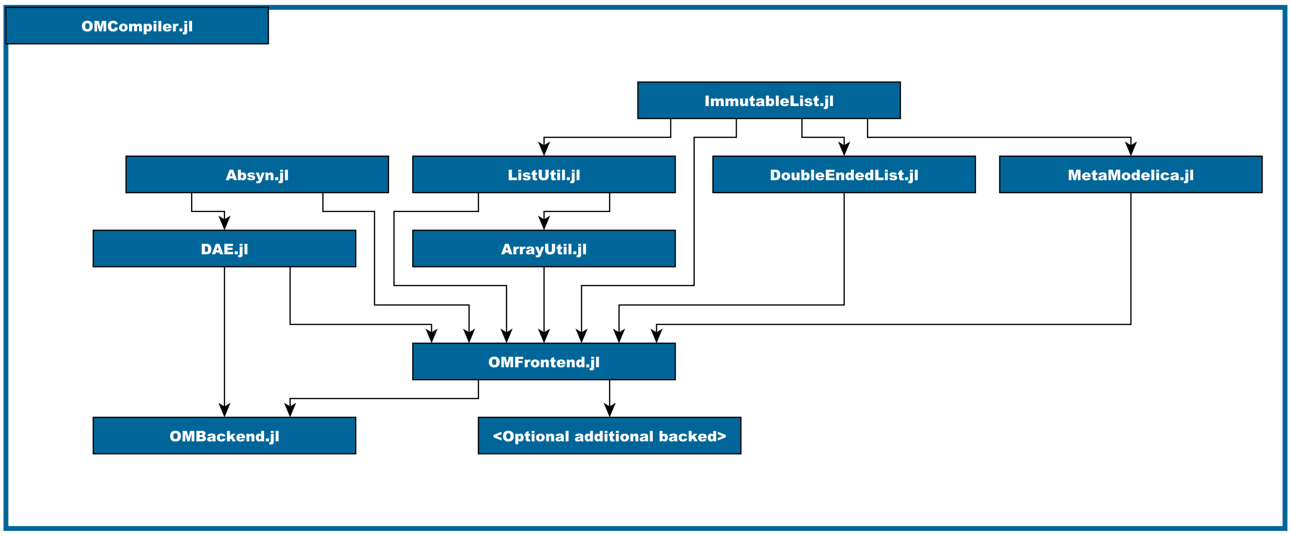

Figure 1. Overview of the inter-dependencies between packages that make up the proposed Julia based Modelica compiler.

Table 1. A brief description of the different components that make up `OMCompiler.jl`

Julia packages	Description
`Absyn.jl`	The concrete syntax tree
`ArrayUtil.jl`	Various utility functions for arrays
`DoubleEnded.jl`	Mutable linked list
`ImmutableList.jl`	Linked list support
`ListUtil.jl`	Various utility functions for lists
`MetaModelica.jl`	Compiler runtime
`Mutable.jl`	Mutable support
`OMBackend.jl`	Simulation and code generation
`OMFrontend.jl`	Lowers Modelica into DAE IR
`OpenModelicaParser.jl`	Standard compatible parser
`SCode.jl`	Implements SCode IR

5.2.1 Example: Hello World in Modelica

Our first example is the HelloWorld model presented in (Fritzson, 2015d) see listing 4 and listing 5 for the generated Julia code.

Listing 4. The Hello World model in Modelica (Fritzson, 2015a).

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

Listing 5. Code generated for listing 4 by `OMCompiler.jl`.

```
  function HelloWorldDAE_equations(res, dx,
       x, p, t)
    res[1] = ((dx[1]) -
      (((- (p[1] ))) * (x[1])))
  end
```

5.2.2 Example: The Van der Pol Oscillator

This example demonstrates the results when compiling the Van der Pol oscillator (Fritzson, 2015d). The model shown in listing 6 was compiled using the different components of `OMCompiler.jl` with final code generation being conducted by `OMBackend.jl` and generates Julia code for simulation, as shown in listing 7. A plot illustrating the fidelity of the simulation with respect to OMC can be studied in fig. 3.

Listing 6. The Van der Pol oscillator model (Fritzson, 2015c)

```
model VanDerPol
  Real x(start = 2);
  Real y(start = 0);
  parameter Real lambda = 1;
equation
  der(x) = y;
  der(y) = lambda*(1 - x*x)*y - x;
end VanDerPol;
```

5.2.3 Example: The Lotka-Volterra equations

In this example, we present the result of generating code and simulating the Lotka-Volterra predator and prey model (Fritzson, 2015b). It models the relationship between prey and predators with respect to time. The result of simulating this using `OMCompiler.jl` and OMC can be seen in fig. 4.

5.2.4 Summary of experiments

As illustrated in figs. 2 to 4 the frontend together with the backend, can compile and simulate standard Modelica within the computational framework of Julia. Although

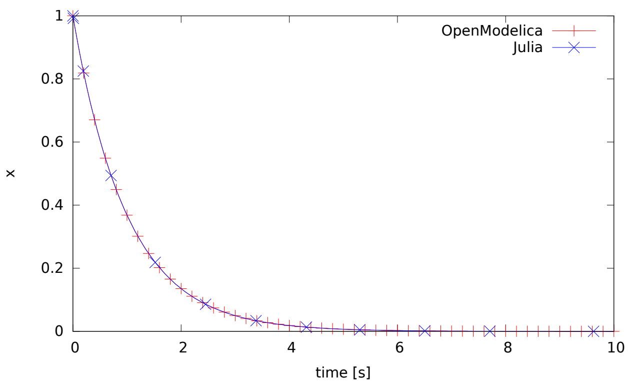

Figure 2. Julia and the corresponding OpenModelica simulation of listing 4 for 10 seconds using IDA.

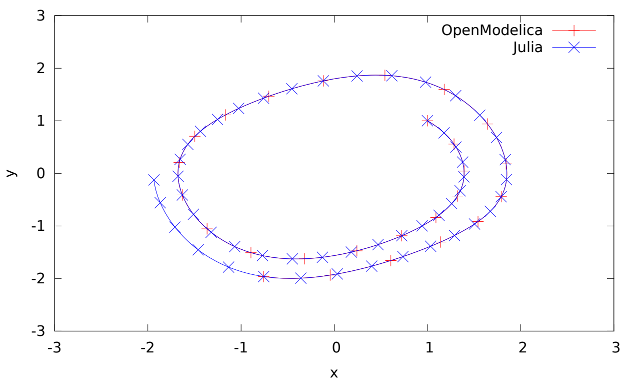

Figure 3. Julia and the corresponding OpenModelica simulation of listing 6 during 10 seconds using IDA.

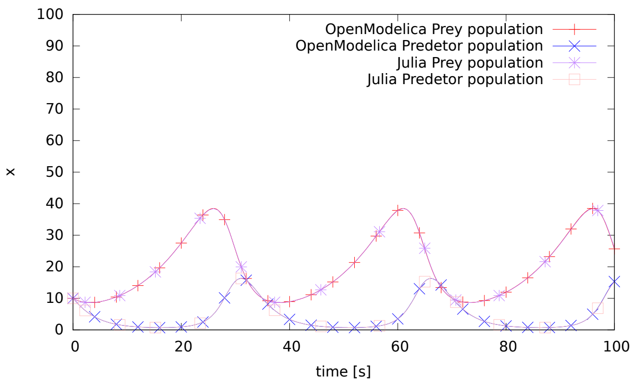

Figure 4. Julia and the corresponding OpenModelica simulation of the Lotka-Volterra model (Fritzson, 2015b) during 100 seconds using IDA

Listing 7. DAE residual equation in Julia

```
function VanDerPolDAE_equations(
    res #= Residual vector =#,
    dx  #= State derivatives =#,
    x   #= States & alg. variables =#,
    p   #= Parameters =#,
    t   #= time =#)
  res[1] = dx[1] - x[2]
  res[2] = dx[2] - (p[1] * ((1.0
      - x[1] ^ 2.0 ) * x[2]) - x[1])
end
```

not all elements of Modelica, such as hybrid system support is as of this writing not supported, we believe that the flattened DAE generated by the frontend presented here is correct. This can be seen by inspecting the resulting graphs. We conclude that the models are simulating within acceptable numeric error tolerances.

Providing hybrid system support along with VSS support, however, would require extensions to the backend. The possibility of adding such support in Julia has already been demonstrated in the work concerning *Modia* (Elmqvist et al., 2017) and within *Haskell* (Giorgidze and Nilsson, 2009).

5.3 Performance of OMBackend.jl

Comparing the current state of `OMBackend.jl` with regards to memory and time during translation is not feasible. The reason being that the current capabilities of the backend are rudimentary. Thus, fewer operations are performed compared to more complete backends such as the one present in OMC. The same holds for the simulation runtime. While both OpenModelica and `OMBackend.jl` are using IDA, the event handling and capabilities to catch asserts of OpenModelica C runtime are more advanced and will, therefore, consume more memory and time. It would thus be hard to draw general conclusions from such an experiment, and it would, in some respect, give our work an unrealistic advantage.

5.4 A non-monolithic compiler

One of the major benefits of the LLVM compiler framework (Lattner and Adve, 2004) is that it is possible for developers to make use of the different components on their own.

Inspired by the LLVM effort, we decided to follow a design approach reminiscent of LLVM by instead of opting for a monolith compiler, to separate the different parts including intermediate representations such as `Absyn.jl` [13] and `SCode.jl` [14] into different components. We elaborate on the consequences of this design in section 5.5.

[13] Absyn.jl
On Github: github.com/OpenModelica/Absyn.jl

[14] SCode.jl
On Github: github.com/OpenModelica/SCode.jl

5.5 Benefits of component-based compiler

The benefits of providing a component-based Compiler are many. First and foremost the intermediate representations of the OMC along with the parser will be available as separate packages. This means that developers can use the parser and the different intermediate representations of the OMC for experimentation without having to change and modify the current monolithic compiler. This opens up many possibilities for increased cooperation and improved tool support. For instance, the backend is not necessarily restricted to be used for one language. By utilizing the DAE IR, it would be possible to devise a specification for another equation-based language and utilize the same backend for simulation purposes, see fig. 5. Analogously it would be possible to support several backends using the frontend presented here.

6 Related work

Several authors have proposed computational frameworks for the treatment of VSS within the context of equation-based languages. One of the first systems to successfully deal with structural variability was Mosilab (Nytsch-Geusen et al., 2005).

However, according to (Höger, 2019), Mosilab did not get traction and had certain drawbacks such as a combinatorial explosion of modes. Zimmer proposed a language *Sol* inspired by Modelica that handles structural changes to the model through symbolic processing at runtime via interpretation (Zimmer, 2010). Höger has a similar approach a few years later but additionally provides a method for dynamic index reduction that improves performance during structural changes (Höger, 2014). Höger (2017) claims that a few language extensions are sufficient to add VSS support to Modelica successfully. Höger (2019) presents a compiler prototype that handles variable structured systems with many modes.

What differs between the framework proposed in this paper and the computational framework suggested by Höger is two things. We base our prototype on an existing compiler, OMC, and we propose a component-based approach. The implication being that the compiler itself is not necessarily coupled to one language.

The Modia language extension to Julia also supports structural changes to the model at runtime (Elmqvist et al., 2017; Benveniste et al., 2019). The main difference between Modia and our approach is that Modia does not aim to be compliant with the Modelica standard nor evolving with it being an embedded domain-specific language for Julia rather than a compiler constructed using Julia.

An approach similar to Modia from a few years earlier is Hydra by Giorgidze and Nilsson (2009). Like Modia Hydra is embedded in a host language, in the case of Hydra the language is Haskell. Similarly to Modia Hydra does not implement Modelica. While both Hydra and the work presented in this paper internally make use of both IDA and LLVM. Hydra is a host language realized within

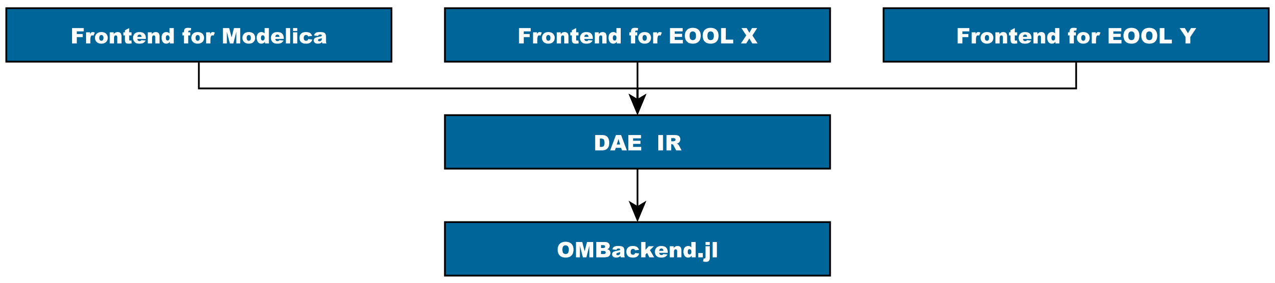

Figure 5. Separate frontends for equation-based languages utilizing the same backend.

Haskell and not a compiler, thus not as flexible in terms of providing support for new syntax and semantics. Still, Hydra provides support for dynamic hybrid systems, whereas at the time of writing `OMCompiler.jl` does not. A similar approach has also been proposed in Broman and Siek (2012). Consequently, the work presented in this paper can be seen as a complement to previous research, see how techniques pioneered before would fare when applied to existing Modelica models used in the industry.

At the time of this writing, the most extensive handling of differential equations in Julia is provided by `DifferentialEquations.jl` (Rackauckas and Nie, 2017). While not a standardized programming language comparable to Modelica, it is a capable package for simulating differential equations. `DifferentialEquations.jl` was selected as our final backend target.

7 Conclusions

In this paper, we presented an initial prototype of a Julia based Modelica compiler. The purpose was to provide flexible foundations for a computational framework capable of VSS support for Modelica while also adhering to existing standards. Thus, we propose an architecture reminiscent of LLVM for equation-based languages utilizing the Julia environment in combination with an automatically translated frontend to support this effort.

While there are performance issues, especially during the frontend phase of compilation, we believe that it is possible to overcome these issues. What we propose is an automatic translation of the new high-performance Modelica frontend in OMC Pop et al. (2019), which relies less on backtracking and outperforms the old frontend on which the frontend here is based.

We plan to investigate the results of Höger (2017), Benveniste et al. (2019), Giorgidze and Nilsson (2009), and Zimmer (2010) to evaluate how VSS support could be incorporated into this framework and consequently how VSS support would work in practice in a Modelica compiler. This raises further directions for future work such as examining what simplifications are possible in existing industrial-grade models.

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, engineers have contributed to the OpenModelica system. There is no room here to mention all these people, but we gratefully acknowledge their contributions. Additionally, we would like to thank Mahder Gebremedhin for providing feedback and advice concerning the backend design and the anonymous reviewers that provided additional feedback to this paper.

References

Albert Benveniste, Benoît Caillaud, Hilding Elmqvist, Khalil Ghorbal, Martin Otter, and Marc Pouzet. Multi-mode dae models-challenges, theory and implementation. In *Computing and Software Science*, pages 283–310. Springer, 2019.

Jeff Bezanson, Stefan Karpinski, Viral B Shah, and Alan Edelman. Julia: A fast dynamic language for technical computing. *arXiv preprint arXiv:1209.5145*, 2012.

David Broman and Jeremy G Siek. Modelyze: a gradually typed host language for embedding equation-based modeling languages. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2012-173*, 2012.

François E Cellier and Ernesto Kofman. *Continuous system simulation*. Springer Science & Business Media, 2006.

Hilding Elmqvist, Toivo Henningsson, and Martin Otter. Innovations for future modelica. In *Proceedings of the 12th International Modelica Conference, Prague, Czech Republic, May 15-17, 2017*, number 132, pages 693–702. Linköping University Electronic Press, 2017.

Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 3.3: A Cyber-Physical Approach*. Wiley-IEEE Press, 2 edition, April 2015a. ISBN 978-1-118-85912-4. pg. 20.

Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 3.3: A Cyber-Physical Approach*.

Wiley-IEEE Press, 2 edition, April 2015b. ISBN 978-1-118-85912-4. pg. 841-842.

Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 3.3: A Cyber-Physical Approach*. Wiley-IEEE Press, 2 edition, April 2015c. ISBN 978-1-118-85912-4. pg. 22-23.

Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 3.3: A Cyber-Physical Approach*. Wiley-IEEE Press, 2 edition, April 2015d. ISBN 978-1-118-85912-4.

Peter Fritzson, Pavol Privitzer, Martin Sjölund, and Adrian Pop. Towards a text generation template language for Modelica. In Francesco Casella, editor, *Proceedings of the 7th International Modelica Conference*, pages 193–207. Linköping University Electronic Press, September 2009. ISBN 978-91-7393-513-5. doi:10.3384/ecp09430124.

Peter Fritzson, Adrian Pop, Martin Sjölund, and Adeel Asghar. MetaModelica – A Symbolic-Numeric Modelica Language and Comparison to Julia. In Modelica'2019. doi:10.3384/ecp19157289.

Peter Fritzson, Adrian Pop, Martin Sjölund, and Adeel Asghar. Metamodelica–a symbolic-numeric modelica language and comparison to julia. In *Proceedings of the 13th International Modelica Conference, Regensburg, Germany, March 4–6, 2019*, number 157. Linköping University Electronic Press, 2019b.

George Giorgidze and Henrik Nilsson. Higher-order non-causal modelling and simulation of structurally dynamic systems. In *Proceedings of the 7th International Modelica Conference; Como; Italy; 20-22 September 2009*, number 043, pages 208–218. Linköping University Electronic Press, 2009.

Christoph Höger. *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, 2019. URL `http://dx.doi.org/10.14279/depositonce-8354`.

Christoph Höger. Dynamic structural analysis for daes. In *Proceedings of the 2014 Summer Simulation Multiconference*, SummerSim '14, pages 12:1–12:8, San Diego, CA, USA, 2014. Society for Computer Simulation International. URL `http://dl.acm.org/citation.cfm?id=2685617.2685629`.

Christoph Höger. Elaborate control: variable-structure modeling from an operational perspective. In *Proceedings of the 8th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*, pages 51–60, 2017.

Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

Sven Erik Mattsson, Martin Otter, and Hilding Elmqvist. Multi-mode dae systems with varying index. In *Proceedings of the 11th International Modelica Conference, Versailles, France, September 21-23, 2015*, number 118, pages 89–98. Linköping University Electronic Press, Linköpings universitet, 2015. doi:10.3384/ecp1511889.

Modelica'2019. *Proceedings of the 13th International Modelica Conference*, March 2019. Modelica Association and Linköping University Electronic Press.

Andrea Neumayr and Martin Otter. Algorithms for component-based 3d modeling. In *Proceedings of the 13th International Modelica Conference, Regensburg, Germany, March 4–6, 2019*, number 157, page 10. Linköping University Electronic Press, Linköpings universitet, 2019.

Christoph Nytsch-Geusen, Thilo Ernst, André Nordwig, Peter Schneider, Peter Schwarz, Matthias Vetter, Christof Wittwer, Andreas Holm, Thierry Nouidui, Jürgen Leopold, et al. Mosilab: Development of a modelica based generic simulation tool supporting model structural dynamics. In *Proceedings of the 4th International Modelica Conference TU Hamburg-Harburg*, volume 2, 2005.

Peter Pepper, Alexandra Mehlhase, Ch Höger, and Lena Scholz. A compositional semantics for modelica-style variable-structuremodeling. In *Proceedings of the 4th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools; Zurich; Switzerland; September 5; 2011*, number 056, pages 45–54. Linköping University Electronic Press, 2011.

Adrian Pop, Per Östlund, Francesco Casella, Martin Sjölund, and Rüdiger Franke. A New OpenModelica Compiler High Performance Frontend. In Modelica'2019. doi:10.3384/ecp19157689.

Christopher Rackauckas and Qing Nie. Differentialequations. jl–a performant and feature-rich ecosystem for solving differential equations in julia. *Journal of Open Research Software*, 5(1), 2017.

John Tinnerholm. An LLVM backend for the Open Modelica Compiler. Master's thesis, Linköping University, Department of Computer and Information Science, 2019. URL `http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-154291`.

John Tinnerholm, Martin Sjölund, and Adrian Pop. 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*, pages 11–19, 2019.

Vadim Utkin. Variable structure systems with sliding modes. *IEEE Transactions on Automatic control*, 22(2):212–222, 1977.

Dirk Zimmer. *Equation-based modeling of variable-structure systems*. ETH Zurich, 2010.