the
WINNOWER

MATHEMATICS

ModelingToolkit, Modelica, and Modia: The Composable Modeling Future in Julia

CHRISTOPHER RACKAUCKAS

READ REVIEWS

WRITE A REVIEW

CORRESPONDENCE:
me@chrisrackauckas.com

DATE RECEIVED:
May 18, 2021

DOI:
10.15200/winn.162133.39054

ARCHIVED:
May 18, 2021

CITATION:
Christopher Rackauckas, ModelingToolkit, Modelica, and Modia: The Composable Modeling Future in Julia, *The Winnower* 8:e162133.39054 , 2021 , DOI: 10.15200/winn.162133.39054

Let me take a bit of time here to write out a complete canonical answer toModelingToolkit and how it relates to Modia and Modelica. This question comes up a lot: why does ModelingToolkit exist instead of building on tooling for Modelica compilers? I'll start out by saying I am a huge fan of Martin and Hilding's work and we work very closely with them on the direction of Julia-based tooling for modeling and simulation. ModelingToolkit, being a new system, has some flexibility in the design space it explores, and while we are following a different foundational philosophy, we have many of the same goals.

COMPOSABLE ABSTRACTIONS FOR MODEL TRANSFORMATIONS

Everything in the SciML organization is built around a principle of confederated modular development: let other packages influence the capabilities of your own. This is highlighted in a paper about the package structure of DifferentialEquations.jl. The underlying principle is that not everyone wants or needs to be a developer of the package, but still may want to contribute. For example, it's not uncommon that a researcher in ODE solvers wants to build a package that adds one solver to the SciML\n ecosystem. Doing this in their own package for their own academic credit, but with the free bonus that now it exists in the multiple dispatch world. In the design of DifferentialEquations.jl, solve(prob,IRKGL16()) now exists because of their package, and so we add it to the documentation. Some of this work is not even inside the organization, but we still support it. The philosophy is to include every researcher as a budding artist in the space of computational research, including all of the possible methods, and building an infrastructure that promotes a free research atmosphere in the methods. Top level defaults and documentation may lead people to the most stable aspects of the ecosystem, but with a flip of a switch you can be testing out the latest research.

The Modelica ecosystem (open standard, OpenModelica, multiple commercial implementations), which started based on the simple idea of equation oriented modeling, has had a huge impact on industry and solved lots of difficult real industrial problems. The modern simulation system designer today, wants much more from their language and compiler stack. For example, in the Modelica language, there is no reference to what transformations are being done to your models in order to make them "simulatable". People know about Pantelides algorithm, and "singularity elimination", but this is outside the language. It's something that the compiler maybe gives you a few options for, but not something the user or the code actively interacts with. Every compiler is different, advances in one compiler do not help your model when you use another compiler, and the whole world is siloed. By this design, it is extremely difficult for an external user to write compiler passes in Modelica which effects this model lowering process. You can tweak knobs, or write a new compiler, or fork OpenModelica and hack on the whole compiler to just do the change you wanted. The barrier to entry can be significantly lowered, as the Julia compiler ecosystem has showed.

I think in many cases, the set of symbolic cases in Modelica may not be sufficient, and budding system designers might want to write their own. For example, on SDEs there's a Lamperti transformation which can exist which transforms general SDEs to SDEs with additive noise. It doesn't always apply, but when it does it can greatly enhance solver speed and stability. This is niche enough that it may never be in a commercial Modelica compiler (in fact, Modelica doesn't have SDEs at this moment), but it's something that some user might want to be able to add to the process.

MODELINGTOOLKIT: OPENING THE DEVELOPMENT PROCESS

So the starting goal of ModelingToolkit is to give an open and modular transformation system on which a whole \nmodeling ecosystem can thrive. My previous blog post exemplified how unfamiliar use cases for code transformations can solve many difficult\n mathematical problems, and my goal is to give this power to the whole development community. `structural_simplify` is something built into ModelingToolkit to do "the standard transformations" on the standard systems, but the world of transformations is so much larger. Log-transforming a few variables? Exponentiating a few to ensure positivity? Lamperti transforms of SDEs? Transforming to the sensitivity equations? And not just transformations, but functionality for inspecting and analyzing models. Are the equations linear? Which parameters are structurally identifiable?

ModelingToolkit is a deconstruction of what a modeling language is. It pulls it down to its component pieces and then makes it easy to build new modeling languages like Catalyst.jl which internally use ModelingToo\nlkit for all of the difficult transformations. The deconstructed form is a jumping point for building new domain-based languages, along with new transformations which optimize the compiler for specific models. And then in the end, everybody who builds off of it gets improved stability, performance, and parallelism as the core MTK passes improve.

BRINGING THE POWER TO THE PEOPLE

Now there's two major aspects that need to be handle to fully achieve such a vision though. If you want people to be able to reuse code between transformations, what you want is to expose how you are changing code. To achieve this goal, a new Computer Algebra System (CAS), Symbolics.jl, was created for ModelingToolkit.jl. The idea being, if we want everyone writing code transformations, they should all have easy access to a general mathematical toolset for doing such code transformations. We shouldn't have everyone building a new code for differentiation, simplify, and substitution. And we shouldn't have everyone relying on undocumented internals of ModelingToolkit.jl either: this should be something that is open, well-tested, documented, and a well-known system so that everyone can easily become a "ModelingToolkit compiler developer". By building a CAS and making it a Julia standard, we can bridge that developer gap because now everyone knows how to easily manipulate models: they are just Symbolics.jl expressions.

The second major aspect is to achieve a natural embedding into the host language. Modelica is not a language in which people can write compiler passes, which introduces a major gap between the modeler and the developer of extensions to the modeling language. If we want to bridge this gap, we need to ensure the whole modeling language is used from a host which is a complete imperative programming language. And you need to do so in a language that is interactive, high performance, and has a well-developed ecosystem for modeling and simulation. Martin and Hilding had seen this fact as the synthesis for Modia with how Julia uniquely satisfies this need, but I think we can take it a step further. To really make the embedding natural, you should be able to on the fly automatically convert code to and from the symbolic form. In the previous blog post I showcased how ModelingToolkit.jl could improve people's code by automatically parall\nelizing it and performing index reduction even if the code was not written in ModelingToolkit.jl. This grows the developer audience of the transformation language from "anyone who wants to transform models" to "anyone who wants to automate improving models and general code". This expansion of the audience is thus pulling in developers who are interested in things like automating parallelism and GPU codegen and bringing them into the MTK developer community.

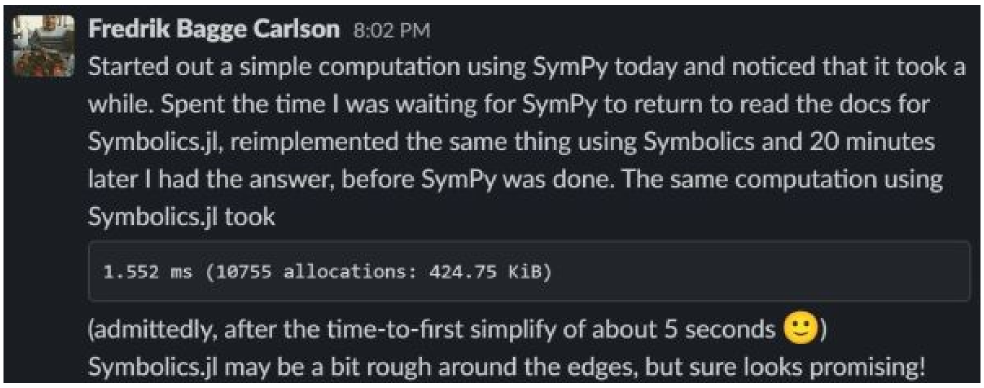
In turn, since all of these advances then apply to the MTK internals and code generation tools such as Symbolics.jl's build_function, new features are coming all of the time because of how the community is composed. The CTarget build_function was first created to transpile Julia code to C, and thus ModelingToolkit models can generate C outputs for compiling into embedded systems. This is serendipity when seeing one example, but it's design when you notice that this is how the entire system is growing so fast.

BUT CAN DISTRIBUTED DEVELOPMENT BE AS GOOD AS SPECIALIZED CODE?

Now one of the questions we received early on was, won't you not be able to match the performance of a specialized compiler which was only made to work on Modelica, right? While at face value it may seem like hyperspecialization could be beneficial, the true effect of hyperspecialization is that algorithms are simply less efficient because less work has been put into them. Symbolics.jl has become a phenomenon of its own, with multiple different hundred comment threads digging through many aspects of the pros and cons of its designs, and that's not even including the 200 person chat channel which has had tens of thousands of messages in the less than 2 months since the CAS was released. Tons of people are advising how to improve every single plus and multiply operation.

So it shouldn't be a surprise that there are many details that have quickly been added which are still years away from a Modelica implementation. It automatically multithreads tree traversals and rewrite rules. It automatically generates fast parallelized code, and can do so in a way that composes with tearing of nonlinear equations. It lets users define their own high-performance and parallelized functions, register them, and stick them into the right hand side. And that is even excluding the higher level results, like the fact that it is fully differentiable and thus allows training neural networks decomposed within the models, and automatically discover equations from data.

Just at the very basic level we can see that the CAS is transforming the workflows of scientists and engineers in many aspects of the modeling process. By distributing the work of improving symbolic computing, we have already taken examples which were essentially not obtainable and making them instant with Symbolics.jl:


Fredrik Bagge Carlson 8:02 PM
Started out a simple computation using SymPy today and noticed that it took a while. Spent the time I was waiting for SymPy to return to read the docs for Symbolics.jl, reimplemented the same thing using Symbolics and 20 minutes later I had the answer, before SymPy was done. The same computation using Symbolics.jl took

```
1.552 ms (10755 allocations: 424.75 KiB)
```

(admittedly, after the time-to-first simplify of about 5 seconds 🙂)
Symbolics.jl may be a bit rough around the edges, but sure looks promising!

We are building out a full benchmarking system for the symbolic ecosystem to track performance over time and ensure it reaches the top level. It's integrating pieces from The OSCAR project, getting lots of people tracking performance in their own work, and building a community. Each step is another major improvement and this ecosystem is making these steps fast.

BUT HOW DO YOU CONNECT TO MODELICA?

This is a rather good question because there are a lot of models already written in Modelica, and it would be a shame for us to not be able to connect with that ecosystem. I will hint that there is coming tooling as part of JuliaSim for connecting to many pre-existing model libraries. In addition, we hope to make use of tooling like Modia.jl and TinyModia.jl and collaboration with the Modelica community will help us make a bridge.

CONCLUSION: DESIGNING AROUND THE DEVELOPER COMMUNITY HAS MANY BENEFITS

The composability and distributed development nature of ModelingToolkit.jl is its catalyst. This is why ModelingToolkit.jl looks like it has rocket shoes on: it is fast and it is moving fast. And it's because of the thought put into the design. It's because ModelingToolkit.jl is including the entire research community as its asset instead of just its user. I plan to keep moving forward from here, looking back to learn from the greats, but building it in our own image. We're taking the idea of a modeling language, distributing it throughout one of the most active developer communities in modeling and simulation, in a language which is made to build fast and parallelized code. And you're invited.

PS: WHAT ABOUT SIMULINK?

I'm just going to post a self-explanatory recent talk by Jonathan at the NASA Launch Services Program who saw a 15,000x acceleration by moving from Simulink to ModelingToolkit.jl.

Enough said. While we don’t expect every application will see this kind of speedup (although we wish they will!), we would love to hear about your experiences with ModelingToolkit.