# Towards a High-Performance Modelica Compiler

Giovanni Agosta[1] Emanuele Baldino[1] Francesco Casella[1] Stefano Cherubin[1] Alberto Leva[1] Federico Terraneo[1]

[1]DEIB, Politecnico di Milano, Italy, given_name.family_name@polimi.it

## Abstract

The use of Modelica as a modelling and simulation language is progressively replacing hand-tuned simulation code written in traditional imperative programming languages. This adoption is fuelled by the availability of libraries to target different application domains, as well as optimizations in Modelica implementations that allow to address larger problems. However, the effort required to extend existing Modelica implementations to support large scale models may not be economically sustainable by the Modelica community. To overcome this barrier, we believe a perspective change is required. Instead of developing, maintaining and optimizing a dedicated codebase, we propose to develop a Modelica implementation by relying on the LLVM state-of-the-art compiler framework. Although this approach will require a higher initial development effort, we believe that it will lead to significantly improved performance as well as lower overall cost. The paper discusses a possible roadmap for such a development, and presents a very early prototype implementation that exploits array structures by avoiding scalar expansion during the code generation process.

*Keywords: Modelica Tools, Large-scale model simulation, Compilers,* LLVM

## 1 Introduction

The high-level, declarative nature of the Modelica language has secured it a widespread adoption across industry and academia alike, bringing DAE-based modeling to many fields where custom simulation codebases had to be developed and maintained.

The performance of mainstream Modelica tools when handling large models has recently improved, mainly thanks to the introduction of sparse solvers (see, e.g. (Braun et al., 2017)). However, for systems approaching or exceeding the one-million equation target the code generation time is unacceptably large, as well as the memory footprint of the generated simulation code, which also has an impact on simulation speed due to CPU cache misses. Efficient simulation of large scale systems, with hundred of thousands to millions of equations, can today only be done with an acceptable compilation and execution performance through hand-written and hand-tuned simulation code. Large-scale models are typical of – and increasingly common in – a variety of relevant application fields: *smart grids* (Vialle et al., 2017) where there is the need to simulate the stability of an electrical network, *detailed thermal simulations* (Leva et al., 2016) that require to partition physical objects in a large number of finite volumes, *coarse-scale fluid dynamics* models for simulation studies targeted to energy efficiency (Bonvini and Leva, 2011). Several more examples could be reported that we omit for brevity.

In crafting hand-written simulation codes optimized to scale to millions of equations, the human designer follows an integrated approach, by coordinating optimizations that are specific of the simulation domain (such as exploiting sparsity in the model, causalization and tearing) and optimizations specific of the computer architecture domain (such as loop optimizations, cache optimizations, vectorization and parallelization).

Although the need to extend existing Modelica implementations to support large models is recognized by the Modelica community (Frenkel et al., 2011; Casella, 2015), significant effort is still required to effectively support large-scale systems. Existing Modelica toolchains are mainly targeted at medium-sized models, and therefore perform heavy structural analysis optimization passes along the translation process. These operations scale poorly for large-scale system. Furthermore, the C-code generation phase does not take into account architectural optimizations, and simply generates unoptimized C code. This approach passes the burden of optimization to the C compiler, to the detriment to both the overall translation efficiency and runtime performance.

A major issue concerning the generation of C code (or any other imperative language, for that matter) is that it is structurally impossible to make the compiler aware of structural properties of the code that could allow further optimizations. Such properties are an obvious consequence of the structural properties of the Modelica code. They can only be preserved by skipping the generation of an intermediate imperative code and by using an intermediate representation instead. One such property, for example, is guaranteeing the absence of pointer aliasing. A C compiler could in principle infer some of such properties from the generated C code, but there is no guarantee that such inference is complete and a lot of time would be wasted recovering information that was already known in the beginning. Moreover, existing Modelica workflows lose additional information during the flattening phase, such as arrays and looping constructs, and as a consequence the generated C code does not exploit exist-

ing CPU architectures effectively.

Significant performance improvements of Modelica tools could thus be achieved if an integrated approach is adopted, where high-level information from the Modelica source, instead of being transferred to an imperative language compiler, is used directly to produce architecture-optimized machine code, effectively resulting in a Modelica-to-binary-code workflow.

Summarizing, in this paper we argue that to scale the Modelica language to large-scale problems, a change of perspective is required, where a Modelica *compiler* – not just a translator – can perform model-specific and architectural-specific optimizations in an integrated way. Our proposal aims at improving the code generation process without any impact on the Modelica syntax and semantics. Thus, being fully compatible with existing Modelica models and libraries. However, our vision involves the re-design of portions of the existing compilation-related language specifications – e.g. the flattening.

We argue that to achieve this result in a cost-effective way, and without redesigning from scratch a complex code generation infrastructure and porting it to existing and future CPU architectures, said Modelica compiler has to be integrated in an existing compiler framework. For this reason, we propose to design a Modelica compiler integrated in LLVM (Lattner and Adve, 2004), which is a state of the art compiler framework, designed with the explicit goals of modularity and extensibility. The authors form an inter-disciplinary research group within the Dipartimento di Elettronica, Informazione e Bioingegneria of Politecnico di Milano, which includes strong competences in the areas of Modelica and object-oriented modelling and simulation, Computer Architectures, and Compiler Design.

This on-going work is today at a very early stage of development. The main goal of this paper is thus to present this group's vision and roadmap, as well as to present some initial results of a very early prototype.

This paper is organized as follows. Section 2 summarizes the state of the art of the support of large-scale models in Modelica tools. Section 3 shows a motivating example for the proposal, while Section 4 presents in detail our roadmap toward the development of a highly optimized Modelica compiler for large-scale systems. Section 5 illustrates the activities we carried out so far, and finally Section 6 ends the paper with some concluding remarks.

# 2 State of The Art

We now briefly describes the state of the art in order to motivate the presented research. Section 2.1 looks at the matter from the Modelica side, evidencing in particular some emerging application domains that require a technological evolution on the part of Modelica tools. Section 2.2 conversely takes the compiler technology standpoint, in a view to sketching out how recent developments in that domain could help realize the mentioned evolution.

## 2.1 The Modelica Side

As discussed in (Casella, 2015), the architecture of current mainstream Modelica tools was designed with individual systems in mind: one robot (possibly two cooperating robots), one hybrid car, one power plant, one air conditioning system, etc., which could be handled by expanding the system model all the way down to its scalar equations, performing optimization on them, and eventually generating code to solve them with ODE solvers using dense matrix algebra. In fact, the very same Modelica Language specification (The Modelica Association, 2017) describes the flattening process with reference to individual scalar variables.

Unfortunately, this approach does not scale well when large-scale systems and systems of systems are modelled. The potential application domains include power generation and transmission systems, smart grids, smart district heating systems with heat pumps (possibly integrated with smart grids), simulation of large fleets of interacting autonomous cars, building energy management simulation (BEMS), and all kinds of future internet-of-things and cyber-physical systems, whose behaviour is the results of the interaction of a large number of physical entities, interacting through a communication network and controlled by centralized and distributed control systems.

The availability of high-quality, open-source, general-purpose sparse solvers such as IDA, Kinsol, and KLU has recently triggered an effort to include support of sparse solvers in Modelica tools, as well as alternative approaches to the simulation of Modelica models that do not rely on the causalization of the system equations but use direct DAE solvers once the system has been symbolically brought to index 1, see (Braun et al., 2017). However, the structural analysis of the system equations, and the consequent code generation, is still carried out on a fully flattened and expanded system.

Some work has been carried out in the past on methods to carry out the structural analysis of the system while keeping repetitive structures such as arrays of variables and loop equations as atomic entities, see (Arzt et al., 2014), possibly also considering issues such as CPU cache misses in the generated code, see (Schuchart et al., 2015). (Zimmer, 2009) proposed methods to exploit the object-oriented structure of large system models, rather than going through full flattening of the equations, in order to come up with more efficient code generation strategies. Unfortunately, all these attempts have remained confined to the stage of concept or prototype implementation, but never made it into mainstream Modelica compiler technology.

## 2.2 The Compiler Technology Side

Compiler technology, while being from several points of view a mature research field, is still evolving. Modern compilers are very costly to develop, ranging in the tens to hundreds of person-years to reach full maturity when


DOI 10.3384/ecp19157313

starting from scratch [1]. As a result, the ability to translate to and from multiple source and target languages is a highly desirable feature, as it allows to pool resources in the development of the large portion of a compiler that is neither target-dependent nor source-dependent.

Since many compiler transformations are fairly general (e.g., loop transformations (Bacon et al., 1994; Grosser et al., 2011) such a *loop unrolling* or *loop tiling* apply in the same way to all loops with the same induction variable evolution), re-implementing them for a new language is unlikely to provide any beneficial effect, and is instead likely to cost additional time in development, optimization, and bug fixing. Actually, the benefits of pooling development resources in this manner are so massive that it is preferrable to abstract some target and language properties (e.g., the size of C integer types for a given target machine) into codified data structures in order to maximise the fraction of the compilation that can be handled by otherwise target-independent, source-independent tools. Thus, a modern compiler is usually implemented on top of a *compiler framework*, a collection of libraries for manipulating and storing an *intermediate representation*, that is a set of data structures that are semantically equivalent to the original program.

As a result of this trend, the GNU Compiler Collection (GCC) dominated the compiler market for decades. However, advanced software does not always age well, and adding more and more optimisation passes forced GCC to stretch the limits of its original design, for instance by adding multiple intermediate representations to supplement the original RTL, which was deemed too low-level to allow certain optimisations.

Nowadays, the industry is increasingly supporting the LLVM compiler framework (Lattner and Adve, 2004) as a more streamlined and modern alternative, leveraging a single, low-level intermediate representation, but capitalising an improved ability to perform loop transformations on lower lever representations. Thanks to the support from multiple large companies such as Google, Apple, Arm, and Sony, LLVM was able to catch up a 20+ years development gap, reaching a position of industry standard in a mere decade from its introduction in 2004. As anticipated and better detailed in Section 4 later on, we propose to develop a Modelica compiler based on LLVM. This choice will in our opinion entail advantages as for both simulation code efficiency and compiler maintainability.

Regarding efficiency, some optimizations were already mentioned, namely operating under the guarantee of no pointer aliasing. Others are for example loop optimizations, on which some words are spent later on. In addition, not going through an imperative language allows the compiler to preserve the non-ordered character of the model (equations in Modelica) as opposite to the ordered nature e.g. of C vectors. When vectors are created to host variables in current mainstream Modelica tools, the order in which these occupy a vector is chosen without any conscience of the consequences on the final machine code. For example, on two different architectures, the same vector order can result in very different cache management efficiencies. A C compiler is inherently incapable of swapping two elements in a vector, as this would alter the semantics of the C program. The same operation however does not alter the semantics of the model, for which the order of variables in vectors is irrelevant.

Coming to the creation and maintenance of the envisaged compiler, a distinctive feature of this research is that LLVM-based compiler development mostly concerns imperative languages, while Modelica is *declarative*. Despite the problems that will surely be encountered, adopting the LLVM framework is keen to produce benefits also from this viewpoint. When considering a highly specialized declarative language such as Modelica, one may come to the wrong conclusion that the majority of transformations required by the code generation process will be strictly language-dependent, and thus to be developed from scratch. In fact, this is actually not the case, as most of the primitives that are provided in an optimized way by the LLVM framework can readily be used in the Modelica context. For example, the well-known equation-variable matching phase of the code generation process starting from Modelica models corresponds to a graph manipulation problem for which LLVM provides the basic data structures (nodes, arcs). In fact, it turns outs that the standard matching algorithm is already implemented efficiently in LLVM [2], because it represents the basic foundation of other types of data-flow analysis, so it can be readily re-used.

# 3 Motivating Example

As a motivating example for our research, in this section we show and briefly comment an experiment that was performed to understand the current scalability gap between Modelica toolchains and optimized handwritten code.

Consider the Modelica benchmark code in Listing 1. The code represents a simple 1D thermal model, describing thermal conduction in a solid copper wire divided in *nx* sections of equal length. Just as in the ScalableTestSuite (Casella, 2015), this model can be simulated with a progressively large *nx*, to observe the scalability of a Modelica toolchain.

The model was tested with a number of equations ranging from 10 to 1 million, using OpenModelica 1.13.0 and Dymola 2018. In both toolchains, an explicit Euler integration algorithm was used. The platform used to run the experiments is a NUMA node with two Intel Xeon E5-2630 V3 CPUs (@3.2 GHz), and 128 GB of DDR4 mem-

[1] See `https://news.ycombinator.com/item?id=16469218` for the development cost of GCC and related tools by Cygnus, estimated in 250 M$ over 10 years by founder D. Henkel-Wallace, and `http://www.ace.nl/compiler/cosy.html` for the development cost of CoSy, estimated in 200 person/years.

[2] See `http://llvm.org/doxygen/SCCIterator_8h_source.html`

**Listing 1.** Thermal conduction benchmark.

```
model Thermal1D
parameter Integer nx = 1000;
parameter Real area = 0.0005^2*3.14;//m^2
parameter Real nlength = 0.1;       //m
parameter Real conductivity = 401; //W/m.K
parameter Real specheatcap = 385;  //J/Kg.K
parameter Real density = 8960;     //Kg/m^3
parameter Real Thi = 400+273.15;   //K
parameter Real Tlo = 20+273.15;    //K
parameter Real g =
  conductivity * area / nlength;
parameter Real c =
  specheatcap * density * area * nlength;
Real T[nx];
initial equation
  for x in 1 : nx loop
    T[x] = Tlo;
  end for;
equation
  c * der(T[1])  =   g * (T[2]    - T[1])
                 + 2*g * (Thi     - T[1]);
  c * der(T[nx]) =   g * (T[nx-1] - T[nx])
                 + 2*g * (Tlo     - T[nx]);
  for x in 2 : nx-1 loop
    c * der(T[x]) = g * (T[x-1] - T[x])
                  + g * (T[x+1] - T[x]);
  end for;
annotation(experiment(StartTime = 0,
StopTime = 100000, Tolerance = 1e-6,
    Interval = 20));
end Thermal1D;
```

**Listing 2.** Optimized C++ implementation.

```
#include <cstdio>
#include <cstring>
#include <string>
#include <chrono>
#include <algorithm>

using namespace std::chrono;

// #define PRINT

int main(int argc, char *argv[])
{
    if(argc<2) return 1;
    int N = std::stoi(argv[1]);
    const int Nsteps = 5000;
    const double h = 20.0;
    const double g = 0.00314785;
    const double c = 0.2707936;
    const double Thi = 400.0 + 273.15;
    const double Tlo = 20.0 + 273.15;
    double *x=new double[N];
    double *xo=new double[N];
    for(int i = 0; i < N; i++) xo[i] = Tlo;
    FILE *fh=fopen("log.csv","w");

    auto a = steady_clock::now();
    for(int j = 0; j < Nsteps; j++)
    {
        x[0] = (1.0-3.0*g*h/c) * xo[0]
             + g*h/c * xo[1]
             + 2.0*g*h/c*Thi;
        for(int i = 1; i < N-1; i++)
            x[i] = g*h/c * xo[i-1]
                 + (1.0-2.0*g*h/c) * xo[i]
                 + g*h/c * xo[i+1];
        x[N-1] = (1.0-3.0*g*h/c) * xo[N-1]
               + g*h/c * xo[N-2]
               + 2.0*g*h/c*Tlo;
        std::swap(x,xo);
        #ifdef PRINT
        for(int i = 0; i < N; i++)
            fprintf(fh,"%e,",xo[i]);
        fprintf(fh,"\n");
        #endif //PRINT
    }
    auto b = steady_clock::now();
    auto e = duration_cast<
             duration<double>>(b-a)
             .count();
    printf("Simulation time %f\n",e);
    fclose(fh);
    delete[] x;
    delete[] xo;
}
```

ory (@1866 MHz) on a dual channel memory configuration. The operating system is Ubuntu 16.04 with version 4.4.0 of the Linux kernel. The compiler used is CLANG version 3.8.0 for OpenModelica, and GCC 7.3.0 for Dymola. The model was also manually translated in optimized C++ code, using an explicit Euler integration algorithm, exploiting the sparsity in the model, and preserving the contained loop constructs. The optimized version can be found in Listing 2.

Tables 1-3 show the results. The simulation column reports only the time for the integration of the differential equations, excluding initialization time and the time required to save results to disk. The binary code size column is only the part of the executable file containing assembly instructions (the .text section), in order to not take into account other metadata such as debug symbols that could be present in the executable. The source code size column is the sum of the size of all C and header files produced by the translator.

From the tables, two main facts can be noted. First, the current generation of Modelica translators is, at least in this simple example, around two orders of magnitude slower than hand-tuned code. Second, the tables evidence the effects of the loss of model structure in current Modelica translators. The flattening of looping constructs results in a source and binary code size that grows linearly with respect to the number of equations in the system, while this does not happen in the handwritten code. As a smaller code size translates to better cache locality, this difference can at least partially explain the improved simulation performance of hand tuned code.

However, there is no theoretical reason why an optimizing Modelica compiler could not generate as efficient


DOI 10.3384/ecp19157313

**Table 1.** OpenModelica Thermal1D simulation time, binary and source code size

| Equations | Simulation | Binary | Source |
|---|---|---|---|
| 10 | 40 ms | 34.6 KByte | 46.1 KByte |
| 100 | 43 ms | 101 KByte | 183 KByte |
| 1000 | 331 ms | 775 KByte | 1.55 MByte |
| 10000 | 2.49 s | 7.37 MByte | 15.6 MByte |
| 100000 | 69.0 s | 73.7 MByte | 160 MByte |
| 1000000 | Stopped after 2 hours | | |

**Table 2.** Dymola Thermal1D simulation time, binary and source code size

| Equations | Simulation | Binary | Source |
|---|---|---|---|
| 10 | 37.6 ms | 193 KByte | 8665 Byte |
| 100 | 49.1 ms | 238 KByte | 53.7 KByte |
| 1000 | 274 ms | 690 KByte | 527 KByte |
| 10000 | 2.64 s | 5.17 MByte | 5.35 MByte |
| 100000 | 61.9 s | 50.9 MByte | 55.7 MByte |
| 1000000 | Stopped after 2 hours | | |

**Table 3.** Handwritten C++ Thermal1D simulation time, binary and source code size

| Equations | Simulation | Binary | Source |
|---|---|---|---|
| 10 | 46 $\mu$s | 4922 Byte | 1258 Byte |
| 100 | 257 $\mu$s | 4922 Byte | 1258 Byte |
| 1000 | 3.72 ms | 4922 Byte | 1258 Byte |
| 10000 | 34.2 ms | 4922 Byte | 1258 Byte |
| 100000 | 401 ms | 4922 Byte | 1258 Byte |
| 1000000 | 3.62 s | 4922 Byte | 1258 Byte |

code as the handwritten one—a remark that in our opinion motivates our research path.

Finally, it is also worth noticing that the time required by the Modelica translators to produce the C code and compile it can significantly exceed the simulation time. Considering the 100000 equations benchmark, OpenModelica took 55 minutes, while Dymola took 4 minutes and 20 seconds. Furthermore, the 1 million equations benchmark was stopped for both Dymola and OpenModelica after two hours, and the simulation had not yet started. Compiling the handwritten C++ code took only 183ms, as the code size is independent on the model size, although a fair comparison cannot be made due to the time required to manually translate the Modelica code to C++.

# 4 Roadmap

Given the considerations laid out so far, it is the authors' opinion that producing highly optimized binary code from a Modelica model is possible. The process we envision first translates the Modelica code into an LLVM intermediate representation (LLVM-IR), and then turns that directly into architecture-optimized machine code. Such an approach exploits all the structural information and metadata that comes from the original Modelica model to the fullest extent.

We also believe that, given the functionality offered by the LLVM framework, this objective can be achieved with an effort that will be abundantly rewarded in terms of efficiency, scalability and, last but not least, maintainability. The performance of the LLVM framework will further improve over time thanks to the efforts of the very active community working on it, which is much wider than the community of Modelica tool developers.

The roadmap laid out here is based on three main assumptions:

- in the future there will be a growing interest in the simulation of large-scale, modular Modelica models of ever-increasing size;
- such large-scale models are built by connecting a very large number of instances of a relatively small number of models, which only differ by the numerical values of their parameters – this is possibly (but not necessarily!) done via arrays of variables and models;
- virtually all modular models are characterized by local interaction, i.e., most (if not all) the components in the system interact with a small number of neighbours only, which means that the corresponding DAE system has a very high degree of sparsity and O($N$) non-zero entries in the incidence matrix, $N$ being the number of instantiated models.

The Modelica compiler we are aiming to build will exploit these features to achieve highly efficient and optimized simulation code generation and execution. This will be obtained by working on two lines of development, Line A and Line B, which are orthogonal and can be carried out simultaneously. Line B is partitioned into two subsequent phases, as shown in Figure 1.

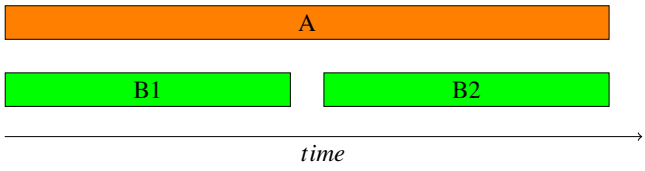


**Figure 1.** The development of our proposed compiler will follow two lines, Line A (compiler backend) and Line B (compiler frontend), partitioned into Phase B1 and Phase B2.

## 4.1 Line A

Line A focuses on the improvement of the Modelica compiler backend by directly integrating with the LLVM compiler framework. The Modelica code will be translated directly into an LLVM-IR, which retains all the structural information that can be extracted from the original Modelica code. This will allow the generation of machine code which is optimized thanks to this information, as well as all the available information about the target hardware architecture.

The traditional intermediate C (or C++) code generation will thus be skipped, drastically reducing the code generation time while at the same time allowing more optimizations to be performed faster.

## 4.2 Line B

Line B focuses on the Modelica compiler frontend, which extends from the Modelica source code parsing to the the transformation of the DAE equations in a form that can be passed to a DAE solver.

### 4.2.1 Phase B1

The traditional Modelica code generation toolchains are based on the complete flattening of the object-oriented features and on the expansion of arrays and unrolling of `for` loops. In fact, the very same Modelica Language Specification is written with this assumption in mind.

The goal of this line is to preserve arrays, `for` loops, and in general the object-oriented structure of the models as much as possible, in order to factor out common behaviour (= equations) in large-scale models. Thus it is possible to achieve much faster code generation, a much smaller memory fooprint, and hence much faster code execution thanks to the vastly reduced chances of cache misses, among other optimizations sought after in Line A. Of course the generated code should eventually lead to the solution of a system of equations which is equivalent to the one that would be obtained by applying the full flattening and expansion mentioned in the Modelica Specification.

The main idea is that the machine-code function to compute the residuals of DAE (and their directional derivatives) in an object which is instantiated many times in the system should only be generated once and then called many times using different inputs and outputs corresponding to the specific variables of each instance.

The concept should be then extended to cover hybrid systems, involving the equations in when clauses, the clocked equations and the zero-crossing functions which are also repeated many times in the large-scale model. The very nice feature of this approach is that the code generation time and the generated machine code footprint scales as $O(1)$ for a large system with $N$ components.

This approach requires the use of direct sparse DAE solvers, such as IDA, which avoids the need of causalization and allows to preserve an N:1 mapping (possibly with some optimizations such as alias elimination) between each equation in `for` loops or model arrays and the corresponding function computing the residual of the equation in the DAE system. This would not possible if the system were causalized, turning it into a set of ODEs, because in general the causalization destroys such N:1 correspondence, depending on the specific causality relationships in the overall system model.

In fact, such an N:1 mapping can be applied to the vast majority of the DAE equations a typical large-scale system model. However, a small set of equations remains that needs a special handling, that will be carried out following the traditional approach for simplicity.

The first sub-set in this set of equations is given by the equations corresponding to *flow* variables in connection sets. Assuming there are no redundant connection statements in a connection set, a statement such as `connect(a,b)` can be directly mapped into the equations $0 = a.v_{nf} - b.v_{nf}$ with an N:1 mapping only for the non-flow variables $v_{nf}$. The equations for flow variables, instead, can only be generated once the connection sets have been computed, a task that can only be performed by analyzing the fully assembled system; only one flow equation per connection set is eventually generated.

The second sub-set is given by the auxiliary equations needed to generate the results of `inStream()` operators. Also in this case it is necessary to analyze the connection sets of the full system model, since the expression of the results of the `inStream()` operator depends on the cardinality of the set and on the `min` attribute of the flow variables of each involved connector.

The third sub-set involves DAE systems of index greater than one. If the system is known a-priori to have index one, as it is e.g. the case of phasor-based power generation and transmission system models, then this set is empty and there is no need of further processing. The a-priori assumption of structural index one could be declared by a suitable annotation of the model. Otherwise it is necessary to flatten and expand the system model all the way down to scalar components, run the matching algorithm and in case of failure due to structural high index, run Pantelides' algorithm, which will identify algebraic constraint equations between variables that appear differentiated in the model, differentiate them and add them to the original set of DAEs. The dummy-derivatives algorithm will also require to select the state set (statically or dynamically), and to demote some derivatives to dummy derivatives, hence a provision must be made to identify as dummy derivatives some elements of un-expanded arrays of derivatives, that are handled by the $O(1)$ efficient code described above. Eventually, the DAE solver will be passed the residuals and Jacobian of a reduced-order index-1 system.

Note that in Phase B1 we still need to expand all non index-1 systems, as in the current state-of-the-art Modelica compilers. We postpone the exploitation of optimization opportunities for higher-index system to the B2 phase. This milestone partitioning allows us to reach a working compiler in shorter time by prioritizing the optimization of index-1 systems.

Summing up, a straightforward implementation of the tool requires to fully flatten and expand the model to scalar components, build the connection sets on it, and then run the standard structural analysis algorithms on it. Note that this fully expanded model *will not be used directly for code generation*, but only to perform structural analysis. The actual code generation process will start from an un-expanded version of the model, in order to achieve $O(1)$ performance as much as possible.


DOI 10.3384/ecp19157313

This processing phase on the fully expanded model requires $O(N)$ time, so it doesn't scale as well as the generation of equations that have an N:1 mapping, which scales as $O(1)$ as discussed above. However, experience carried out by some of the authors with the OpenModelica compiler shows clearly that the time and memory resources involved in these specific phases of the processing of a fully flattened model are a tiny fraction (5–10% at most) of the total. Hence, a performance improvement of at least one order of magnitude is expected by following this approach, compared to the traditional approach of running all the code generation phases on the fully flattened system.

As to the runtime performance of the executable simulation code, it has to be noted that the cardinality of the additional set of equations that need to be generated from the fully expanded model (connection equations for flow variables, equations defining `inStream()` outputs and equations differentiated by Pantelides' algorithm) is again a very tiny fraction (a few percentage point at most) of the total number of equations of the system. This means that the penalty on the runtime performance and memory footprint of these equations not being handled in an array- and object-oriented-structure-preserving way will be very small, compared to what happens when a traditional full flattening and expansion approach is followed.

#### 4.2.2 Phase B2

Once the development of Phase B1 is complete, it would be possible to focus on the modularization of the structural analysis algorithms. The current state-of-the-art approaches work only on scalar variables. In this phase, a generalized versions of such algorithms will be designed, which can handle entire arrays and sets of equations from `for` loops, or from arrays of models as individual `E` and `V` nodes.

On one hand, this modularization would further improve the performance and the scalability of the tool. On the other hand, developing such generalized algorithms that work efficiently in all cases could turn out to be quite a hard task. Therefore, the ratio between the development effort and the performance gains is probably going to be much less spectacular than the one that can be achieved by completing Phase B1. It is the authors' opinion that this kind of optimization is worth considering only after the optimizations described in Phase B1 have been fully exploited.

## 5 On-Going Work

Since the second half of 2018, we started the implementation of a compiler prototype to materialize the effort discussed in Section 4. The development of such prototype is being addressed in a master's thesis work that aims at demonstrating the benefits of the efficient exploitation of arrays and equation loops. Due to time and resource limitations, the current prototype is still in a very preliminary state.

The focus of this thesis work is avoiding the generation of redundant code that is obtained when the conventional flattening-based approach based is followed. At the moment authors are writing, our prototype is limited to the handling of flat models with no object-oriented structures. Structural analysis is still not implemented, so that only the dense version of the IDA DAE solver can be used. Last, but not least, the direct LLVM-IR code generation is not yet in place, and the prototype still generates C code that is then compiled by clang into executable code.

As a consequence, the performance of our compiler prototype on large-scale models is still very far from the objectives stated in the roadmap. That said, our prototype can handle simple Modelica models with arrays and for loops, producing correct simulation results. The improvements currently supported by our prototype lie in the code generation stage. We aim at the preservation of the data- and code-structures concepts as they are written in the Modelica source code. In particular, we avoid to perform the *vector expansion* whenever it is possible. Thus, we generate a residual function that features loops over variables. Our compiler generates a compact code that better exploits instruction locality with respect to the code generated by OpenModelica.

Another improvement over the OpenModelica code generation consists in the reduced modularity of the residual function. OpenModelica generates a single C function for each equation to be described. This approach is fairly convenient for debugging purposes, as it allows to trace the effects of the single equation from the source code to the executable binary. However, it implies a non-trivial overhead due to function call instructions at runtime, and this overhead is not paid back through code reuse, as these functions are only called once. We reduce this overhead during the code generation stage in two ways. The first optimization is a direct consequence of the loop preservation: whenever there is an equation within a loop body, we reuse the same code at each iteration of the loop. The second optimization consists in the inlining of functions, to be performed before the code generation. Instead of generating an independent C function and respective call instructions to invoke it, we directly place the code of that function in the residual function. This second optimization can be performed at almost zero cost during the code generation stage, as opposed to later forcing the compiler to analyze the emitted code as a whole.

Although preliminary results on small dense systems supported by our prototype are promising, actual direct performance comparisons on meaningful test cases against mainstream Modelica translators will become significant as soon as the support for the sparse version of the IDA solver is implemented. This effort and the replacement of the intermediate C code generation pass with the LLVM-IR one are planned to be carried out in 2019.

# 6 Conclusions

In this paper, we are proposing to realise a Modelica compiler based on a compiler framework, namely LLVM. The presented research is carried out by a group at the Politecnico di Milano, putting together knowledge and experience about Modelica and its use for a wide variety of applications, about computer architectures, and about compiler science and technology.

We have motivated our proposal based on current trends observed in the problems that Modelica models need to address, with particular (yet not exclusive) reference to large-scale systems.

We have argued that not passing through the generation of source code in an imperative language can yield improvements in terms of wider optimization possibilities, as the semantics of a language like C inherently causes a loss of information about the semantics of the original model, that could be exploited to tailor the code to its target architecture.

We have also noticed that the huge effort spent, and the vast community involved in compiler frameworks, quite certainly entail future benefits in terms of compiler standardization and maintainability.

We have shown a motivating example to support our statements, defined a roadmap for future activities, and briefly described what we carried out so far.

We hope that this paper fosters a discussion in the Modelica community, and that our proposal can be a basis for a future generation of efficient, architecturally flexible and easily maintainable Modelica compilers.

# References


Matthias Arzt, Volker Waurich, and Jörg Wensch. Towards utilizing repeating structures for constant time compilation of large Modelica models. In David Broman and Peter Pepper, editors, *Proceedings of the 6th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*, pages 35–38, Berlin, Germany, Oct 10 2014. ACM. ISBN 978-1-4503-2953-8. doi:10.1145/2666202.2666207.

David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, December 1994. ISSN 0360-0300. doi:10.1145/197405.197406. URL `http://doi.acm.org/10.1145/197405.197406`.

M. Bonvini and A. Leva. Object-oriented sub-zonal modelling for efficient energy-related building simulation. *Mathematical and Computer Modelling of Dynamical Systems*, 17(6): 543–559, 2011. doi:10.1080/13873954.2011.592143.

W. Braun, F. Casella, and B. Bachmann. Solving large-scale Modelica models: new approaches and experimental results using OpenModelica. In *Proc. 12th International Modelica Conference*, pages 557–563, Prague, Czech Republic, 2017. doi:10.3384/ecp17132557.

F. Casella. Simulation of large-scale models in Modelica: State of the art and future perspectives. In *Proc. 11th International Modelica Conference*, pages 459–468, Versailles, France, 2015. doi:10.3384/ecp15118459.

J. Frenkel, C. Schubert, G. Kunze, P. Fritzson, M. Sjölund, and A. Pop. Towards a benchmark suite for Modelica compilers: Large models. In *Proc. 8th International Modelica Conference*, pages 143–152, Dresden, Germany, 2011. doi:10.3384/ecp11063143.

Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Größlinger, and Louis-Noël Pouchet. Polly-polyhedral optimization in llvm. In *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, volume 2011, page 1, 2011.

C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. 2004 International Symposium on Code Generation and Optimization*, pages 75–86, Palo Alto, CA, USA, 2004. doi:10.1109/cgo.2004.1281665.

A. Leva, F. Terraneo, and W. Fornaciari. Event-based control as an enabler for high power density processors. In *Proc. 2nd International Conference on Event-based Control, Communication, and Signal Processing*, pages 1–8, Krakow, Poland, 2016. doi:10.1109/EBCCSP.2016.7605253.

Joseph Schuchart, Volker Waurich, Martin Flehmig, Marcus Walther, Wolfgang E. Nagel, and Ines Gubsch. Exploiting repeated structures and vectorization in modelica. In *Proc. 11th International Modelica Conference*, pages 265–272, Versailles, France, Sep 21–23 2015. doi:10.3384/ecp15118265.

The Modelica Association. Modelica - A unified object-oriented language for physical systems modeling - Language specification version 3.4. Online, 4 2017. URL `https://www.modelica.org/documents/ModelicaSpec34.pdf`.

S. Vialle, J.P. Tavella, D. Cherifa, R. Corniglion, M. Caujolle, and V. Reinbold. Scaling FMI-CS based multi-simulation beyond thousand FMUs on Infiniband cluster. In *Proc. 12th International Modelica Conference*, pages 673–682, Prague, Czech Republic, 2017. doi:10.13140/RG.2.2.14481.63847.

D. Zimmer. Module-preserving compilation of Modelica models. In *Proc. 7th International Modelica Conference*, pages 880–889, Como, Italy, 2009. doi:10.3384/ecp09430028.