Testing Large Scale System Simulation using Linear Implicit Equilibrium Dynamics

Dirk Zimmer

Institute of System Dynamics and Control,
German Aerospace Center (DLR), dirk.zimmer@dlr.de

Abstract

The concept of flattening where all model equations are collected in a single set is deeply hardwired into the Modelica language. While flattening enables effective symbolic manipulation such as the reduction of systems with a higher-index, it also imposes limitations. Two of these limitations are that the code generation for very large systems may not scale very well and that a statement on the regularity of the system often cannot be made before the flattening took place. Whereas it is difficult to overcome these limits in the general case, there is a sub-class within Modelica models that is comparatively easy to precompile while still enabling the modeling of complex systems. This paper explores this path and presents first experiments on the scalability for larger systems.

Keywords: Compilation, Modelling Methodology, Large Scale Systems

1 Introduction

1.1 Motivation

This work has its origins in the quest for robust modeling methodologies. The goal of such a methodology is to enable a sufficient statement for object-oriented modeling:

Any valid combination of components (under rules of limited complexity) shall have a solution representing a physical system.

In practice that means, that robust components will lead to robust overall system simulation. One particular solution for this task is the class of Linear Implicit Equilibrium Dynamics or short: LIED.

Because solvability can be guaranteed upfront, also the compilation of this class of models can be simplified, even to such a degree that the pre-compilation of components can be enabled, making it attractive for large system simulation. However, before explaining LIED we shall revisit the current approaches for precompiling or simulating larger systems in Modelica.

1.2 State of the Art

The problems that occur with the compilation of very large Modelica models has been noted more than 15 years ago (Zimmer2009) with suggestions how to mitigate this issue with a post flattening analysis. While this never came to any fruitful results, multiple other approaches have been conceived for this problem. Most notably a full test suite has been provided (Casella2015) to help the simulation of large system in OpenModelica (Braun2017). This work is now supported by a new research compiler MARCO (Agosta2023).

One (albeit limited) approach is to find a particular solution for large arrays or vectors. Per default, these elements will all be scalarized, leading to a large number of variables. It would be very helpful keeping them as one entity.

A specific solution for this approach can be found for instance in (Neumayr2023). For a general Modelica compiler, it is however not always clear which vector or matrices can avoid scalarization so that casualization still can take place. This has been investigated by (Abdelhak2023) and is also experimented on in MARCO.

Other attempts originate from a different perspective. The pre-analysis or even pre-compilation of components is also useful for variable-structure or multi-modal systems (Benveniste2019). One elaborate proposal in this direction for the general case stems from (Benveniste2023).

Pre-compilation is of course not only useful for multi-modal systems but (as shown in this paper as well) is a promising technique for large scale systems as well. Indeed, the paper (Benveniste2023) provides a scalarization example of the casualization algorithm.

One common challenge of all of these approaches is that they add complexity to existing Modelica compilers and a full Modelica compiler is already very complex. Some of the approaches hence switch to other experimental modeling languages.

Also, it shall be noted that the support for large models has meanwhile been improved in existing tools. An order of magnitude could be gained in the last decade for certain commercial tools, seemingly by improving the quality of implementation and the availability of more memory.

This paper now focusses not on a general solution but on a special class of models. Let us hence study how this class is being defined.

2 Linear Implicit Equilibrium Dynamics

Linear Implicit Equilibrium Dynamics (LIED) is technically defined as a special class of Differential Algebraic Equation (DAE) Systems.

2.1 Formal Definition

A DAE system with potential state derivatives $\dot{\boldsymbol{x}}$, time t and algebraic variables $\mathbf{w}$

$$\mathbf{0} = \boldsymbol{F}(\dot{\boldsymbol{x}}, \boldsymbol{x}, \mathbf{w}, t)$$

is defined as LIED system when it can be transformed into the following form:

$$\begin{bmatrix} \mathbf{w}_E \\ \dot{\boldsymbol{x}}_E \end{bmatrix} = \mathrm{g}(\boldsymbol{x}_I, \boldsymbol{x}_E, t)$$

$$\mathbf{A}(\boldsymbol{x}_I, \boldsymbol{x}_E, \mathbf{w}_E) \begin{bmatrix} \mathbf{w}_I \\ \dot{\boldsymbol{x}}_I \end{bmatrix} = \mathrm{f}(\boldsymbol{x}_I, \boldsymbol{x}_E, \mathbf{w}_E, t)$$

We see that both the algebraic variables as well as the state derivatives can be split into a fully explicit part ($\dot{\boldsymbol{x}}_E$; $\mathbf{w}_E$) and a part ($\dot{\boldsymbol{x}}_I$; $\mathbf{w}_I$) with a linear system in implicit form expressed by the regular matrix $\mathbf{A}$. Furthermore, the following conditions shall hold true:

$$\dot{\boldsymbol{x}}_E \cap \dot{\boldsymbol{x}}_I \subseteq \dot{\boldsymbol{x}}$$
$$\mathbf{w}_E \cap \mathbf{w}_I \supseteq \mathbf{w}$$
$$\dot{\boldsymbol{x}}_E \cap \dot{\boldsymbol{x}}_I \cap \mathbf{w}_I \supseteq \dot{\boldsymbol{x}}$$
$$\dot{\boldsymbol{x}}_E, \dot{\boldsymbol{x}}_I, \mathbf{w}_E, \mathbf{w}_I \text{ are all disjoint}$$

These conditions essentially mean that it is allowed to perform certain symbolic mechanism of index reduction such as the dummy derivative method (Mattsson1993) originating from (Pantelides1988). Using this method, states variables of $\boldsymbol{x}$ can be transformed to algebraic variables in $\mathbf{w}_I$ and further derivatives may be added to $\mathbf{w}_I$ or $\mathbf{w}_E$. In practice, this is important because it means that the linear implicit dynamics can be expressed by far fewer states than suggested by the vector $\boldsymbol{x}$ of the original DAE formulation.

2.2 Informal Explanation

The formal definition above may be primarily perceived as a relatively strong restriction on the model equations and not many systems may be intuitively expected to fall into this category. Surprisingly, LIED can be applied successfully for the object-oriented modelling of complex thermo-fluid architectures (Zimmer2020,2022) or to mechanical systems with stiff contacts (Zimmer2023).

The idea is that the non-linear behavior of the slow mode is explicitly expressed whereas the fast dynamics that typically is needed to uphold non-linear constraints is expressed by a linear implicit system that fulfils the constraint in its equilibrium. Hence the name: linear implicit equilibrium dynamics. The equilibrium dynamics is thereby often a replacement dynamic and only an approximation of reality (as all modelling is).

2.3 Use Cases for LIED

As the above references demonstrate, LIED has been applied using Modelica for the object-oriented modeling of thermo-fluid or mechanical systems. To this end, it is necessary to use triplets as interface of the model components that consist in a signal for the explicit non-linear part and a pair of potential and a pair of potential and flow for the implicit part as presented in Table 1.

Table 1. Connection triplets for the object-oriented modelling of LIED Systems.

Domain	Signal	Potential	Flow
trans. mechanics	position: r [m]	velocity: v [m/s]	force: f [N]
rotational mechanics	angle: φ [rad]	angular velocity: ω [rad/s]	torque: τ [Nm]
Thermo-fluid streams	Thermo-dynamic state: $\widehat{\Theta}$	inertial pressure r [Pa]	mass-flow rate: $\dot{m}$ [kg/s]

More background on the derivation of these triplets can be found in (Zimmer2024). It goes beyond the scope of this paper how the equations are formulated in detail but the two Modelica model diagrams shown in Figure 1 and Figure 2 may illustrate the practical usefulness.

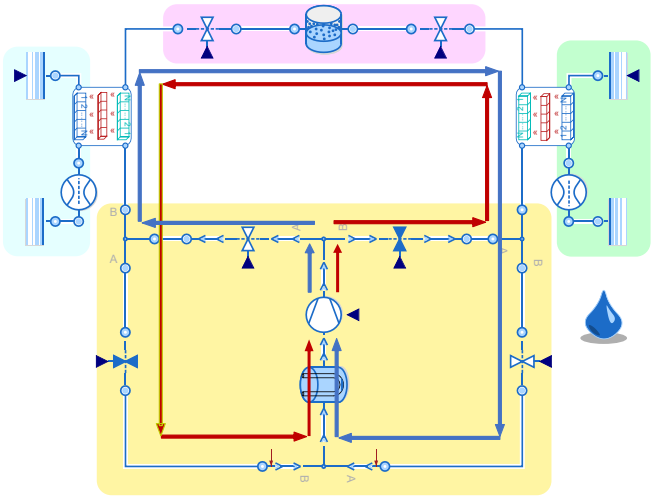

Figure 1. Model diagram of a reversible heat pump systems using the ThermoFluid Stream Library.

DOI
10.3384/ecp21785

Especially the ThermoFluid Stream Library has meanwhile become a popular OpenSource library used by academia (Junglas2023) and by industry.

LIED systems have very benevolent characteristics for object-oriented modelling. Following simple connection rules, the resulting matrix $\mathbf{A}(\boldsymbol{x}_I, \boldsymbol{x}_E, \mathbf{w}_E)$ will be regular and an a-priori statement on solvability can be given (Zimmer2020). This makes this class of modeling very robust and prevents many computational simulation errors.

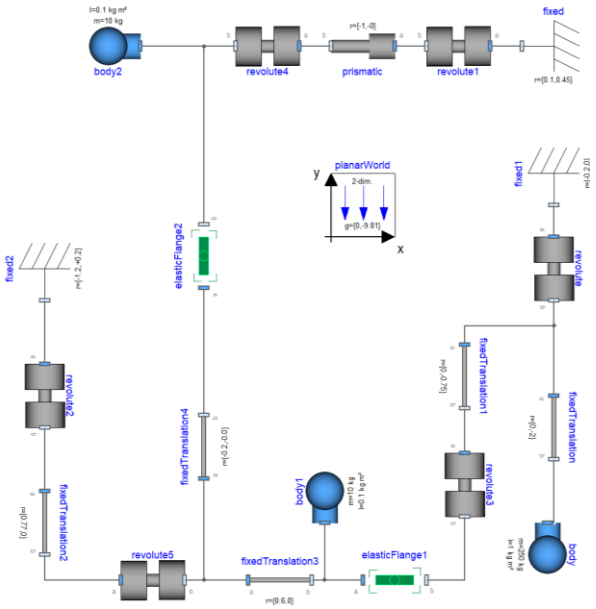

Figure 2. Model diagram of a reversible heat pump systems using the ThermoFluid Stream Library.

New libraries are currently under development that follow this methodology.

3 Potential Compilation of LIED Systems

It turns out that components modeled according to the LIED Methodology have a predictable computational structure. As outlined in (Zimmer 2024), suitable variables for states, their derivatives, for tearing variables and residuals are known a-priori on the level of individual components. This means that flattening is not necessary for LIED systems anymore and that instead the compilation could be performed on the component level. Such a compiler has not yet been developed. So far only a sketch exists on how Modelica models would need to be prepared to enable a component wise compilation.

From a dedicated compiler for LIED systems, we would expect the following advantages:

- Better scalability in terms of code generation speed and memory consumption
- Better error messages based on the connection rules on component level instead on equation level

There may be also disadvantages to be expected:

- Slower simulation code (at least for smaller systems) due to lack of symbolical optimization
- Of course: such a compiler is only applicable of the subset of LIED system and not a full Modelica compiler.

Hence, before making the effort of developing a dedicated compiler for LIED systems, it makes sense to study these benefits and drawbacks of the corresponding compile target. To do so, a dedicated LIED simulator was implemented in C++.

4 Building a Simulator for the LIED Compilation Target.

When compiling a LIED component from Modelica into C++, how would the corresponding C++ code look like? Essentially each LIED component can be represented by a C++ class. The class may thereby define the following elements:

- Interfaces for the signals
- Member functions for the blocks that process these signals
- Local parameters and variables
- Special member objects of type `ContinuousState` that define state variables and their derivatives
- Special member objects of type `Tearing` that define tearing variables and their linear dependent residuals.
- A mandatory virtual member function `metainfo` that provides meta information for the simulation engine.

To better understand these elements, let us look at a concrete example. Figure 3 presents a simple planar mechanical system of a crane crab (a pendulum attached to a slider). To each component of the system, 3 computational blocks are assigned of different color: blue, green, and orange. The blue and green blocks form thereby a computational sequence directed from the root whereas the orange signal leads to the root. For this particular domain of LIED systems, the blue signal contains the positional state and undergoes non-linear transformations. The green signal represents the velocity. Its derivative may be used as tearing variable for which the orange signal that represents the force provides a linear response.

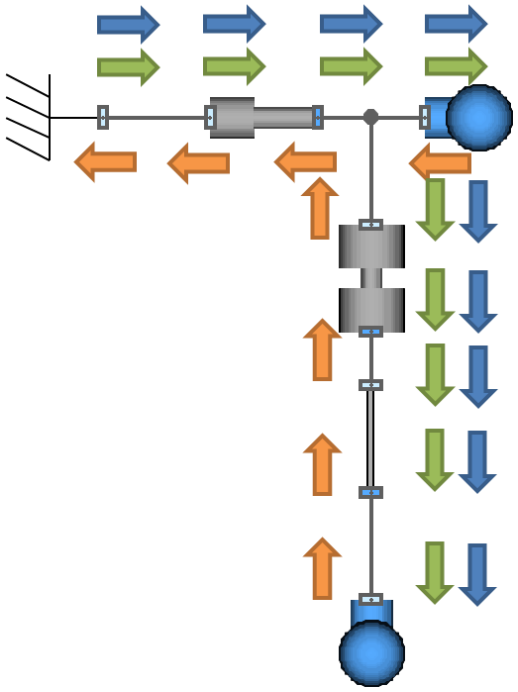

Figure 3. Crane crab modelling diagram

Let us look at one example component that contains all of the elements described above: the revolute joint. In Figure 4, the 3 member functions are marked with the colors to match Figure 3.

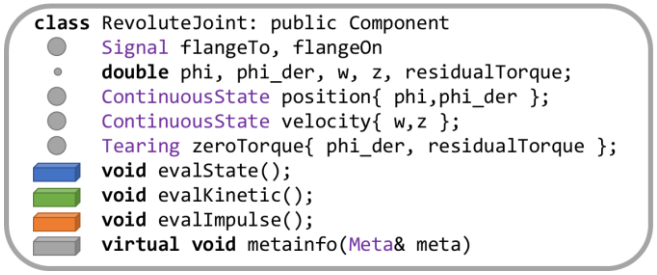

Figure 4. C++ class diagram of a model component.

The virtual member function `metainfo` is crucial to understand how the simulator works. Here is the corresponding code for the `RevoluteJoint`.

Listing 1. Function for metainformation on the revolute joint.

```
virtual void metainfo(Meta& meta) override {
  meta.regComp(&flangeTo,"flangeA");
  meta.regComp(&flangeOn,"flangeB");
  meta.regComp(&position, "position");
  meta.regComp(&velocity, "velocity");
  meta.regComp(&zeroTorque, "zeroTorque");

  meta.regVariable(&phi,"phi: [rad]");
  meta.regVariable(&phi_der,"phi_der:[1/s]");
  meta.regVariable(&w, "w_kin: [1/s]");
  meta.regVariable(&z, "z: [1/s2] ");

  meta.regVariable(&residualTorque,"…");
  meta.addBlock(this,
    [](Component* c) {
      return ((RevoluteJoint*)c)->evalState();
      },
      Signals{&flangeTo.state,
              &position.state },
      Signals{&flangeOn.state}
  );
  meta.addBlock(this,
   [](Component* c) {
     return ((RevoluteJ…*)c)->evalKinetic();
     },
     Signals{&flangeTo.kinetic,
             &velocity.state,
             &zeroTorque.tearing },
     Signals{&flangeOn.kinetic}
   );
  meta.addBlock(this,
   [](Component* c) {
     return ((RevoluteJoint*)c)->evalForce();
     },
    Signals{&flangeOn.j},
    Signals{&flangeTo.j,
            &position.derivative,
            &velocity.derivative,
            &zeroTorque.residual}
  );
};
```

The `Meta` object, whose reference is passed to the `metainfo` function, crawls through the entire model in a recursive way and collects all information of interest. There are different child classes of the abstract `Meta` class: for instance, the `DiagnosisMeta` class collects meta information about the variable names. In order to construct the simulation code, the `StructuralMeta` class is most relevant to us. It collects all the information which block depends on which signals and what variables are states, derivatives, tearing variables or residuals. Given this information callback objects can be created. The final simulation program is then simply a list of these call backs. Sections for dynamic evaluation and tearing of linear equation systems are thereby separately marked.

The following listing shows the top-level representation of the crane crap model in C++ as potential compile target. As you can see, it is possible to maintain the original structure of the model. The reader may use these examples to trace the recursive calls of the `metainfo` function.

Listing 2. CraneCrab Example in the compile target.

```
class CraneCrab : public Component  {
public:
  Fixed fixed{};
  PrismaticJoint  prismatic1{Vector2d{1.0,0}};
  Body body1{1.0,0.1};
  RevoluteJoint revolute2{};
  FixedTranslation rod2{Vector2d{0.5,1.5}};
  Body body2{0.5,0.05};
```


DOI
10.3384/ecp21785

```
Connections con {
   Connection{&fixed.flangeOn,
              &prismatic1.flangeTo},
   Connection{&prismatic1.flangeOn,
       &body1.flangeTo},
   Connection{&prismatic1.flangeOn,
              &revolute2.flangeTo},
   Connection{&revolute2.flangeOn,
              &rod2.flangeTo},
   Connection{&rod2.flangeOn,
              &body2.flangeTo}
 };

 virtual void metainfo(Meta& meta) override {
   meta.regComp(&fixed, "fixed");
   meta.regComp(&prismatic1, "prismatic1");
   meta.regComp(&body1, "body1");
   meta.regComp(&revolute2, "revolute2");
   meta.regComp(&rod2, "rod2");
   meta.regComp(&body2, "body2");
 };
};
```

Figure 5 presents the overall architecture of the simulation program. We can go through its main elements from the left to the right.

Models consist of Components, Signals and Connections, whereby both signals as well as components may have member functions that represent blocks.

When a model gets instantiated, it gets prepared for its evaluation. This is the function of the `ModelEvaluation` class. It will use the `StructuralMeta` crawler to collect all necessary meta information. Then all blocks will be sorted and the sections representing torn linear equation systems will be compressed and marked. Compression hereby means that all blocks not belonging to the linear system are placed outside this section.

The Simulator class will now apply a numerical ODE solver. Currently `zimsim` features a generalized solver for explicit Runge-Kutta Fix Step Methods with Butcher Tableaus for order 1 to 4, Backward Euler, and ESDIRK23 (Jørgensen2018) as solver algorithms.

When the simulator generates results, it calls the Recorder class. The recorder then decides on its own what to store at which frequency. The recorder may use a separate crawler on the simulated model to retrieve information on variable names, model hierarchy etc. Also, it may trigger the model evaluation if separate evaluations are needed for the generation of output variables. The recorder class is defined in a very generic manner and can be used to multiplex the output to other recorders.

We see that the design of a simulator for pre-compiled components may differ significantly from the monolithic generation of simulation code.

The structure of the model is maintained for the simulation code and instead of having to decide which meta-information to provide for the output before translation, this decision can be done at simulation time or even after simulation took place.

What are the benefits and drawbacks of such an architecture? We would expect a much lower memory consumption because meta-information can be generated on demand. On the other hand, we would expect a higher computational overhead: alias variables are not eliminated and linear equation systems need to be numerically solved instead of symbolically.

To better quantify these benefits and drawbacks, a scaling experiment is being performed where memory consumption and computational speed is being measured for models of different size.

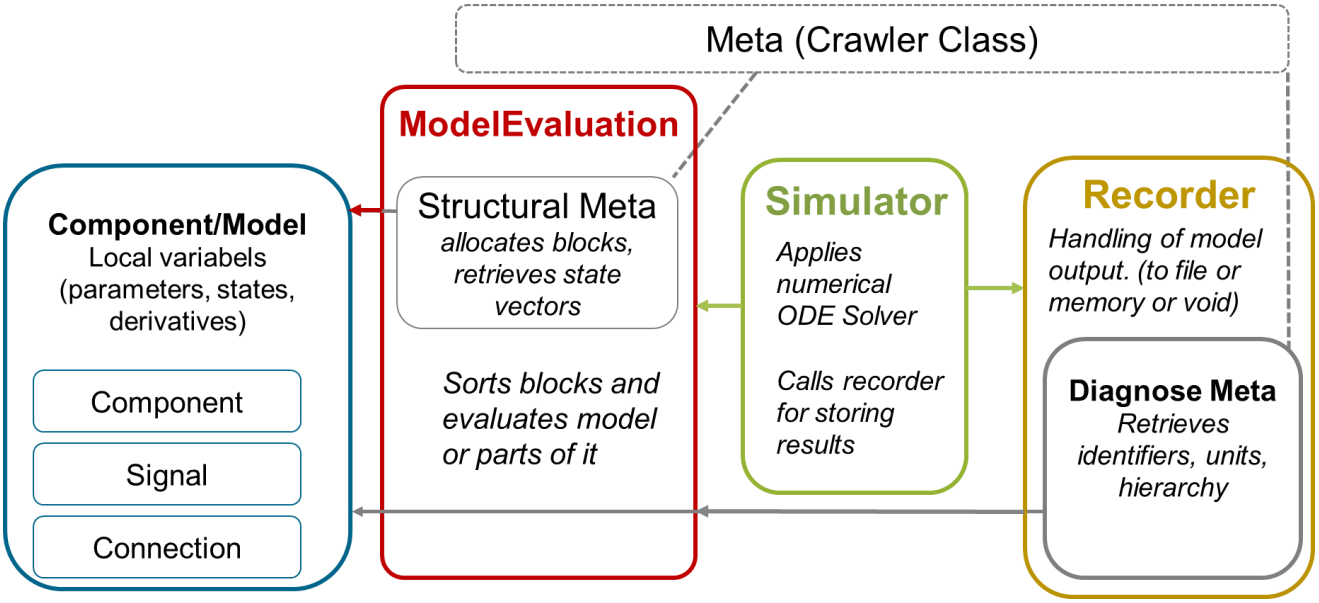

Figure 5. Architecture of the object-oriented simulator program

5 Evaluation of the Scaling Properties.

The crane crab model of Figure 3 has been used to perform a simple scaling experiment. Using a logarithmic grid of factor 4, 1 to 4,194,304 crane crabs have been instantiated and simulated using `zimsim`. The same exercise has been performed within an equivalent Modelica model using the same equations. OpenModelica (Fritzson2019) (abbreviated by OM) and a commercial Modelica tool have been used for the simulation tests. Here, this generates models ranging from 206 equations up to 792 million equations. In Figure 6 to Figure 8, we use the equation number since this quantity is more familiar to Modelica users.

Comparing simulation speed and memory consumption across tools is not trivial. For instance, the simulation speed may be influenced by the generation of output. For this reason, output generation was largely suppressed. Memory consumption refers to the estimated peak consumption of the whole program.

In general, one should be very careful when comparing absolute numbers for a specific case. What matters to a scaling test though is the slope in the log-log diagram.

All tests have been performed on a machine with Intel(R) Core(TM) i7 at 1.80 GHz and 32GB of RAM using Windows 10 as operating system.

All tools have been tested to their maximum extent. There are however orders of magnitude in difference between the tools. In the end, `zimsim` was limited by available memory. Also the memory consumption of the commercial tool caused disk swapping slowing down the whole process to such an extent that it did not finish within several hours. In OpenModelica, a clang linker error occurred for 1024 crane crabs. Simulation performance could thus not be estimated. Memory consumption and compile time for this data point were replaced by values that exclude the linking.

5.1 Memory Consumption

`zimsim` has been written with efficient use of memory in mind: Meta information is generated only on-demand and also the object-oriented formulation enables to do more computation on the steak than on the heap.

Peak memory usage for model translation, compilation and simulation is orders of magnitudes lower using `zimsim`. This is shown in Figure 7. The likely reasons are:

- Avoidance of flattening leading to much less instruction code.
- On demand generation of meta-information
- More variables on stack than on heap

For small models, comparing the memory usage hardly makes any sense and is also not fair because we compare the memory usage of a dedicated simulator (LIED) with the one of a whole modeling and simulation environment. The two orders of magnitude for small models is thus not surprising. What is surprising that this gap does not significantly close for larger models. Peak memory consumption seems to appear in the tools at model translation for the Modelica tools but this is also true for `zimsim.` The last data point of Figure 7 is misleading. The model was assigned 29GB of RAM by the operating system while the estimated remaining 15 GB of memory demand where provided by disk swapping. This also shows in the performance of model instantiation but not in simulation because this can be performed within the provided 29GB.

5.2 Computation Time

Pure simulation time is compared here. Time for translation, compilation and instantiation is ignored. Generation of output has been reduced to a negligible amount.

Figure 8 reveals that simulation speed of `zimsim` is competitive to OpenModelica but slower for small models than a commercial tool. The likely reason is the numerical solution of linear equation systems in `zimsim` instead of symbolical transformation. This primarily reveals one thing: flattening for the sake of flattening is not worth it. Structured code is also fast. Flattening is then a good idea when you can replace an iterative solver with a symbolic solution. (however, there might be ways to generate structured code that reduces iterations as well)

We can observe that for larger systems, the performance penalty is smaller, the reason is unknown.

A side remark: The performance study has only been performed using explicit solvers, ignoring the computation of the Jacobian which is very relevant for implicit solvers. However, implicit solvers scale badly for very large systems.

5.3 Time for Translation + Compilation or Instantiation

One expected advantage of pre-compilation would be that a model could be instantiated almost immediately. Hence, the time for model instantiation in `zimsim` is displayed in Figure 6.

The instantiation essentially uses the famous Tarjan algorithm for strong component analysis that provides a partial order of the dependence graph as side effect. It can be performed very efficiently. For it to take longer than a single second, models of more than 1 million equations are needed.

Time for Translation and Compilation in the Modelica tool is added to the chart. This measurement is not very precise (at least 20% error bar) and for all smaller models completely dominated by the C compiler. The comparison is hence not fair for smaller models.

DOI
10.3384/ecp21785

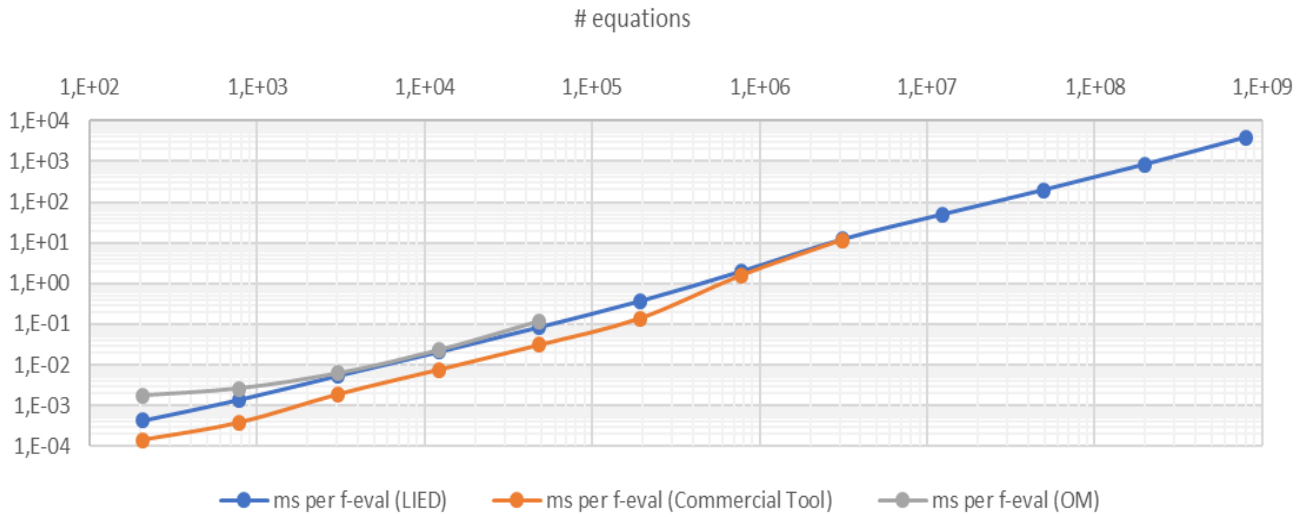

Figure 8: Computation speed: milliseconds per model evaluation over number of equations

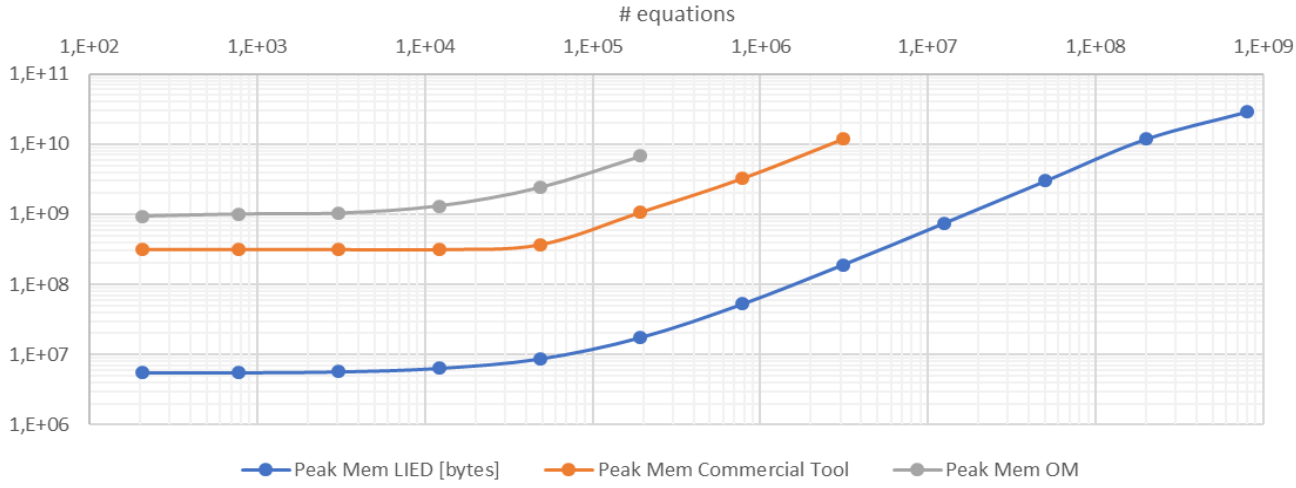

Figure 7. Peak memory usage in bytes over number of equations

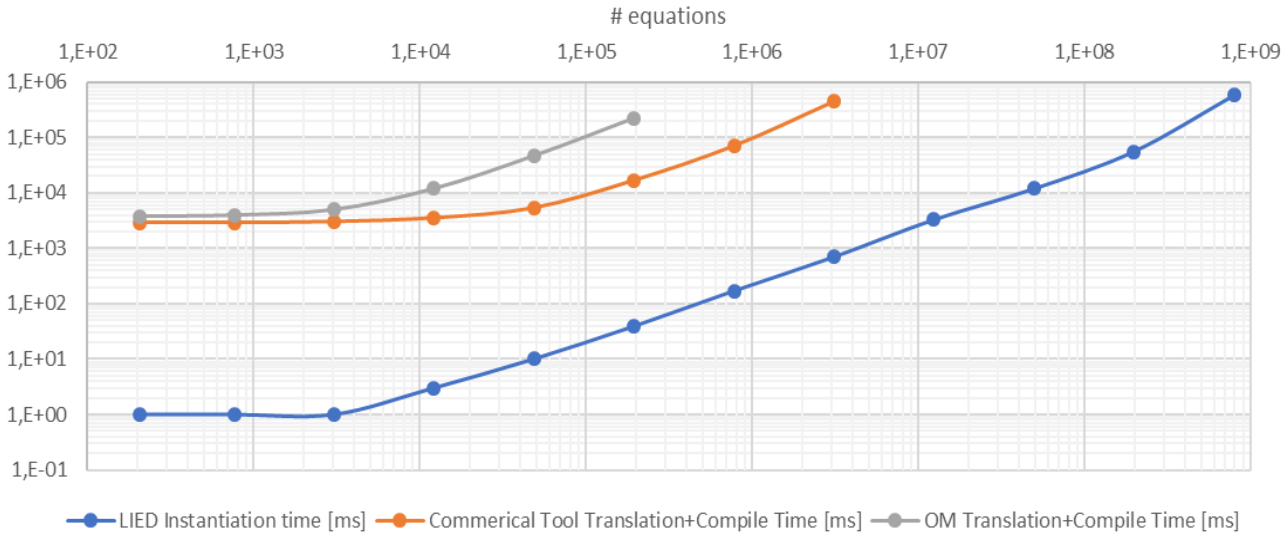

Figure 6. Instantiation Time or Translation+Compile time in ms over number of equations

6 Conclusion

The goal of this paper is to explore and compare alternative pathways for the compilation of simulation code. Naturally flattening and the generation of monolithic rather unstructured simulation code are procedures inherent to compiling Modelica. These procedures enable the effective symbolical manipulations and the removal of alias variables.

How strong these advantages are compared to a more structured generation of simulation code that compiles on the component level was however examined sparsely.

The preliminary examination presented in this paper is not sufficient to provide conclusive answers yet. Improvements in the algorithm and simulation code are needed because many potentials for code optimization have not yet been exploited. Furthermore, a much wider set of examples is needed. At the current stage, it merely indicates that the performance gain by flattening per se is not very significant, symbolic solution are preferable over iterative ones and it also shows that pre-compilation has a great potential to be way more memory efficient enabling the simulation of much larger models.

We also have to recall that the component-wise compilation could not be performed on arbitrary models but only on models following the LIED methodology. Albeit this is a remarkably powerful subclass of Modelica, it is still only a subclass.

Acknowledgement

This work was supported by the ITEA4 research project OpenScaling with financial support of the German federal ministry of education and research referring to the grant number: 01IS23062C

SPONSORED BY THE

Federal Ministry of Education and Research

References

Abdelhak, K., F. Casella, B. Bachmann (2023) Pseudo Array Causalization. Proceedings of the 15th International Modelica Conference, Aachen

Agosta F., et al. (2023) MARCO: An Experimental High-Performance Compiler forLarge-Scale Modelica Models Proceedings of the 15th International Modelica Conference, Aachen.

Benveniste, Albert, Benoît Caillaud, Hilding Elmqvist, et al. (2019). “Multi-Mode DAE Models - Challenges, Theory and Implementation”. In: Computing and Software Science – State of the Art and Perspectives. Vol. 10000. Lecture Notes in Computer Science. Springer, pp. 283–310.

Benveniste, A., B. Caillaud, M. Malandain, J. Thibault (2023) Towards the separate compilation of Modelica: modularity and interfaces for the index reduction of incomplete DAE systems Proceedings of the 15th International Modelica Conference, Aachen.

Braun, Willi, Francesco Casella, and Bernhard Bachmann (2017). “Solving large-scale Modelica models: new approaches and experimental results using OpenModelica”. In: Proceedings of the 12th International Modelica Conference.

Casella, Francesco (2015). "Simulation of Large-Scale Models in Modelica: State of the Art and Future Perspectives", Proceedings of the 11th International Modelica Conference, doi:10.3384/ecp15118459

Fritzson, Peter A. et al. (2019) “The OpenModelica Integrated Modeling, Simulation, and Optimization Environment.” Proceedings of The American Modelica Conference 2018, October 9-10, USA

Jørgensen, J.B., Kristensen M. R. and Grove, P. (2018) A Family of ESDIRK Integration Methods. arXiv Numerical Analysis eprint :1803.01613

Junglas, P. (2023) Implementing Thermodynamic Cyclic Processes Using the DLR Thermofluid Stream Library. Simulation News Europe E 33(4)

Mattsson, S.E., Gustaf Söderlind (1993). “Index Reduction in Differential-Algebraic Equations Using Dummy Derivatives” In: SIAM Journal on Scientific Computing 1993 14:3, 677-692

Neumayr, A.; Otter, (2023) M. Modelling and Simulation of Physical Systems with Dynamically Changing Degrees of Freedom. *Electronics*, doi.org/10.3390/electronics12030500

Pantelides, C. (1988), The consistent initialization of differential-algebraic systems, SIAM J. Sci. Statist. Comput., 9, 213–231

Zimmer, Dirk (2009) Module-Preserving Compilation of Modelica Models. In: *Proceedings 7th Modelica Conference*, Como, Italy, Sep. 20-22

Zimmer, D. (2020), Robust Object-Oriented Formulation of Directed Thermofluid Stream Networks. Mathematical and Computer Modelling of Dynamic Systems, Vol 26, Issue 3.

Zimmer, D., N. Weber, M. Meißner (2022) The DLR ThermoFluid Stream Library. MDPI Electronics - Special Issue.

Zimmer, D., C. Oldemeyer (2023). “Introducing Dialectic Mechanics”. Proceedings of the 15th International Modelica Conference, Aachen.

Zimmer D. (2024) Object-Oriented Modeling of Classic Physical Systems using Linear Implicit Equilibrium Dynamics Preprints 2024, 2024031139