Performance Benchmark of Modelica Time-Domain Power System Automated Simulations using Python

Sergio A. Dorado-Rojas Manuel Navarro Catalán Marcelo de Castro Fernandes Luigi Vanfretti

Department of Electrical, Systems and Computer Engineering
Rensselaer Polytechnic Institute
Troy, NY, USA
{dorads, navarm2, decasm3, vanfrl}@rpi.edu

Abstract

In this paper, a benchmark between solvers and Modelica tools for time-domain simulations of a power system model is presented. A Python-based approach is employed to automate Modelica simulations and compute performance metrics. This routine is employed to compare the performance of a commercial (Dymola) against an open-source (OpenModelica) simulation tool with different solver settings. Python scripts are developed to execute a dynamic simulation of a common model for power system studies with 49 states and 420 variables in three different scenarios. This degree of automation makes it easier to change solver settings and tools during execution. The performance of each of the tools is assessed through metrics such as execution time and CPU utilization. The quantitative comparison results provide a clear reference to the performance of the tools and solvers for the execution of time-domain simulations with a significant degree of complexity. The commercial tool offers better performance for variable-step solver, but the performance of the open-source software shows significantly faster results for fixed-step solvers.

Keywords: Modelica, Python-Dymola Interface, Python-OpenModelica Interface, CPU performance.

1 Introduction

List of Acronyms and Definitions

Definitions

ST · Simulation Time: the simulation time is understood as the time it takes for the program to translate, compile and integrate a model.

ET · Execution Time: the execution time is the elapsed time to complete the numerical integration of the compiled model. It also known as *integration time*. Note that execution time is included as a part of Simulation Time in the context of this paper.

NMT · Normalized Minimum Execution Time: performance metric taken as a function of the execution time of each of the tools. It is defined as

$$\mathrm{NMT}^{[\mathrm{solver}]} = \frac{\min(\mathrm{ET_D}, \mathrm{ET_{OM}})}{\mathrm{ET_{observed}}}$$

where $\mathrm{ET_D}$ are the execution times for Dymola and OM, respectively, and $\mathrm{ET_{observed}}$ is the corresponding integration time of each tool for a given solver obtained from the simulation log.

Acronyms

ET · Execution Time
MSE · Mean Square Error
NAE · Normalized Absolute Error
OM · OpenModelica
OMPython · OpenModelica Python Interface
OpenIPSL · Open-Instance Power System Library
PDI · Python-Dymola Interface
ST · Simulation Time

Motivation

Modeling and simulation of power systems have been a habitual practice in the energy industry since the 1960s. The complexity of a power system is steadily increasing to accommodate modern technologies into the existing grid. A more complex system leads to more elaborated models. High-complexity models are directly correlated with computationally expensive tasks (Milano, 2010). In this context, the Modelica language represents an accurate, equation-based, multi-domain solution modeling and simulation alternative. Numerous initiatives such as OpenIPSL have been taken to incorporate into the power system workflow the benefits of the Modelica language (Baudette et al., 2018).

On the other hand, the academic, scientific and industrial communities have come to acknowledge the intrinsic benefits of the Modelica language. An outcome of this trend is that the user base has increased significantly during the last years. This has led to the development of many libraries with users coming from a very wide domain spectrum. Nowadays, Modelica stakeholders include students, consulting firms, big laboratories, and industry agents.

Free tools such as OpenModelica are fundamental for learning the language at little to no cost and to set a reference for the Modelica language (Fritzson et al., 2006).

Commercial tools such as Dymola, SystemModeler or SimulationX provide advanced functionalities that satisfy particular requirements from the industry. However, there is no clear guidance for a user on how to select the tool-solver based on its simulation performance exclusively.

This paper intends to provide this guidance. It aims to compare the time-domain simulation performance of the solvers from both Dymola and OpenModelica when subjected to different solver settings (see (Braun et al., 2017) for a detailed analysis of the potential of OpenModelica to solve large-scale models). Since these tools do not have the same features and solvers, we have chosen some of the ones they have in common for benchmarking purposes.

Contribution

This work is relevant to any user of the Modelica language. The tool performance analysis is based on the simulation of a power system model (IEEE 14 bus system), that serves as a representation of a dynamic nonlinear system. We consider three simulation scenarios: an initialization, a line-opening (one discrete event) and two bus faults (two discrete events). The paper reveals the difference in performance within the tools and helps users make an educated choice about the tools to use. The main contributions of the paper are the following:

- Quantitative evaluation of Dymola and OpenModelica simulation performance for time-domain simulation of complex dynamic systems (power systems).
- Benchmark of different solvers in a dynamic simulation with discrete events.
- Implementation of simple Python routines to automate Dymola and OpenModelica time-domain simulations.

Paper Organization

The paper is broken down in the following sections: Section 2 describes the test system and the Modelica library employed to construct it. The experiment setup regarding hardware characteristics and software setup is described in Section 3. In Sections 4 and 5, we discuss performance results of each of the tools with respect to each solver and the corresponding performance metrics. Finally, Section 6 concludes the work.

2 Modelica Power System Model

The IEEE14 bus system[1] represents a part of the Midwestern USA American Electric Power System as of February of 1962. The single-line diagram of the system can be seen in Figure 1. This model was chosen because it is a widely used testing system for an initial assessment in power system dynamical studies since it has a significant number of variables and states (420 and 49, respectively) which makes it a common factor in such simulation-based studies (Milano, 2010). For this reason, its dynamic simulation a challenge to the tools and the CPU.

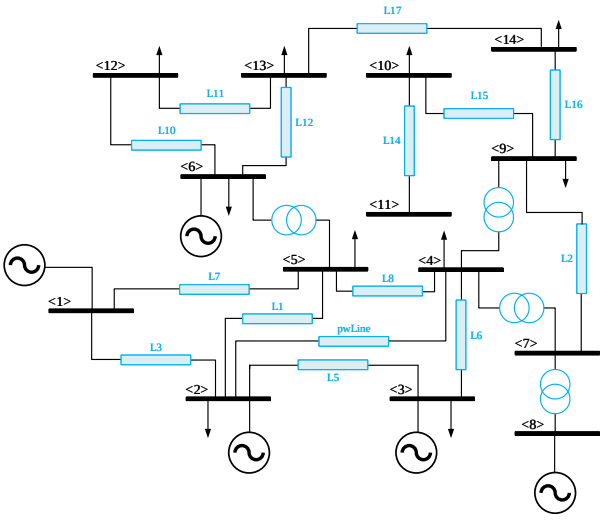

Figure 1. IEEE 14 bus model.

The test power system model (Figure 2) is built using the components from the open-source OpenIPSL library, a Modelica-based power system component library currently developed and maintained by ALSETLab at Rensselaer Polytechnic Institute. The library includes all the components to build a large power system model and perform dynamic analysis in time- and phasor-domain. The version of the library used in this paper is release 1.5.0[2].

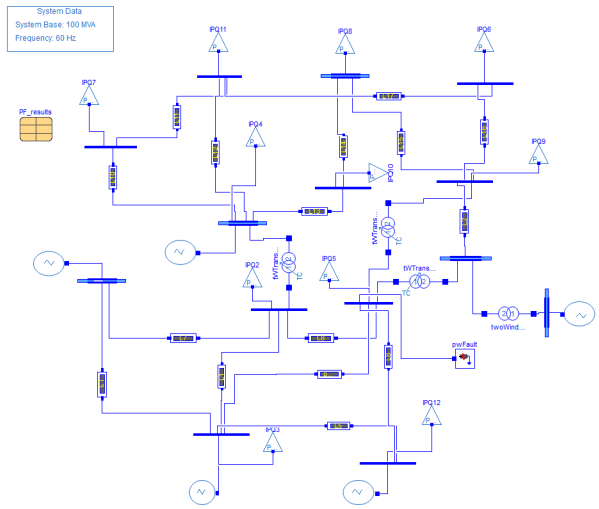

Figure 2. Implementation of the IEEE 14 bus model in Dymola using OpenIPSL.

3 Experiment Specifications

To make sure that the results are reproducible, this section details the conditions under which the experiments were performed regarding hardware setup and software characteristics.

[1] https://icseg.iti.illinois.edu/ieee-14-bus-system/

[2] The version of the library employed for this paper is included in the GitHub repository of the project. For the latest release of OpenIPSL, see: https://github.com/OpenIPSL/OpenIPSL

3.1 Hardware and Software Setup

The characteristics of the computer used to run the simulations are shown in Table 1.

	Characteristic
Operating System	Ubuntu Server 18.04 LTS
RAM	128 GB
Processor	Intel(R) Xeon(R) CPU E-1650 v4 12 Cores @ 3.60 GHz 15 MB Cache
Storage	1 TB SSD
Graphics Cards	4 x NVIDIA GTX 1080 Ti (CUDA Capable) 11 GB GDDR5X (each)
Dymola Distribution	Dymola 2020x
OpenModelica Distribution	1.14.0
Python Release	3.6.8
Dymola Compiler	MinGW CC
OpenModelica Compiler	MinGW CC

Table 1. Hardware characteristics and software specifications of the computer used to run the experiment.

To assess solver performance correctly, numerical integration must run in only one processor. While this is a default option in Dymola, we need to specify this option explicitly in OpenModelica before starting any simulation since it defaults to multi-core execution. This is done thanks to the flag `setCommandLineOptions("-n=1")`.

3.2 Simulation Scenarios

To properly measure solver performance in diverse dynamic conditions, we will consider the following three scenarios of the IEEE14 bus model: system initialization, time-domain simulation with one line opening, and system response with two faults.

IEEE14 System Initialization (S_1)

This scenario corresponds to a system with no disturbing events. The power flow of the model is modified so that the numerical simulation has problems during initialization. The provided initial conditions are such that the dynamic system is not initially at an equilibrium point, thus forcing the system to look for an acceptable steady-state condition at the beginning of the integration process. This increases the computational task and challenges the solver since the integration does not start with all state derivatives equal to zero.

Line Opening (S_2)

Besides the aforementioned bad initialization condition, we introduce a line opening to disturb the system from steady-state and excite nonlinear dynamics. This kind of scenario is used to study system-wide stability when two sub-areas are disconnected from each other. The line opening corresponds to the connection between buses 2 and 4 (B2 and B4). The line will open from both ends at time $t = 60$ s and will re close at $t = 61.5$ s.

Bus Faults (S_3)

In this case, the system will face two three-phase to ground faults at different times. This configuration is used to test the resiliency and stability of the system. By having two faults, the numerical complexity of the simulation increases, creating a more adverse scenario for the solvers to come up with a solution. Fault 1 occurs at bus 4 (B4) starting at $t = 20$ s and being removed at $t = 21.2$ s. Fault 2 takes place at bus 14 (B14) at $t = 80$ s, being cleared at $t = 81.2$ s. The parameters of the two faults are $R = 0$ pu and $X = 1 \times 10^{-5}$ pu.

3.3 Solver Selection

The performance of the time-domain simulation depends not only on the dynamic condition to be analyzed but also on the solver selection. In this regard, OpenModelica and Dymola contain a wide variety of different integration methods and three of them are going to be used and thus briefly described in this study. The Differential Algebraic System Solver (*dassl*) is an implicit, high-order, variable-step solver with time-step control. This solver is set as default solver in both OpenModelica and Dymola. The *Euler* method is another solver available in both software packages and it is an explicit (Forward Euler), first-order, fixed time-step solver. Finally, the last solver used in this study is the *runge kutta*. Dymola allows the user to chose between second, third and fourth order Runge-Kutta methods but in this work, only the fourth order is used since it is also available in OpenModelica. This solver is an explicit, fourth-order, fixed time-step solver. This paper will benchmark the performance of the tools with each of the mentioned solvers for the different scenarios of the test power system.

3.4 Time-step Selection

Since *dassl* is a variable-step solver with step-size control, there is no need to select a specific time-step for the solver. The selection of an adequate number of intervals is necessary to plot and analyze the results. For both tools, 5000 was found to be a reasonable number of simulation intervals. Moreover, to use the capabilities of a DAE solver to their full extent, we enable the newly incorporated DAEmode in Dymola by enabling the flag `Advanced.Define.DAEsolver = true` (Henningsson et al., 2019). In OpenModelica, to set similar settings we use the command `setCommandLineOptions("daeMode=true")`.

On the other hand, it is important to select an adequate step size T_s for fixed-step solvers in order to guarantee that the algorithm is operating in its region of convergence. To get an upper bound for T_s, we performed a linear analysis of the system in Dymola employing the library `Modelica_LinearSystems2`. After determining the time constant of the fastest mode ($\tau \approx 1$ ms), we found that $T_s = 0.5$ ms was a reasonable value to capture the effects of the fastest mode, guaranteeing numerical convergence for both solvers, *Euler* and *Runge Kutta*. The selected time-step size implies that 240,000 simulation intervals are going to be needed for a simulation time of 120 s.

3.5 Benchmark Metrics

In order to understand and accurately compare the two tools the paper focuses on two simulation features to compare:

- *Simulation Time* (ST) corresponds to the time it takes for a program to complete all of the routines for each scenario comprising model translation, compilation and execution. The discussion of the results of the simulation time are found in Section 4.1, with special remark on Execution Time (ET).
- *CPU Utilization* is the percentage of central processing unit (CPU) that is being used at any time during the execution. Results for CPU utilization can be found in Section 4.2.

3.6 Code Structure

The complete code to perform the experiments and analyze the resulting data can be found in GitHub[3]. The execution of the simulations is automated through Python using the Dymola API (Python-Dymola Interface) and the OpenModelica Python Interface (OMPython) (Lie et al., 2018). The details of the Dymola routine can be seen in the file `dymola_simulation.py`. Likewise, the OpenModelica commands are included in the file `om_simulation.py`.

To measure performance we execute the routine in the script `measurement_performance.py`. It measures each of the performance metrics every 0.2 s while the code is running in a different parallel process. The main program is contained in the file `01_modelica_tool_performance_benchmark.py`.

4 Performance Results

Before presenting the performance results, we validate the simulation outputs of the three scenarios for Dymola and OpenModelica for all solvers. We employed the Normalized Absolute Error (NAE) and the Mean Square Error (MSE) defined in Equation (1) to quantify the numerical difference between the outcomes of each tool.

[3]https://github.com/ALSETLab/Time-Domain-Simulation-Performance-Benchmark

$$\text{NAE} = \frac{|x_i - y_i|}{n}$$
$$\text{MSE} = \sum_{i=1}^{n} \frac{(x_i - y_i)^2}{n} \tag{1}$$

NAE shows how different the Dymola and OpenModelica results are throughout the simulation. MSE outputs a quantitative validation of the results of both tools (Devore and Berk, 2012). Full details can be seen in Table 3.

The numerical behavior of the simulation during initialization (*runge kutta* solver) can be observed in Figure 3 for the voltage magnitude signal at Buses 2 and 4. An initial transient behavior can be seen at the beginning of the integration time. This is not desired in a dynamic simulation since numerical convergence to a steady-state solution is not guaranteed given the fact that the solver starts from a guessing point with non-zero derivatives.

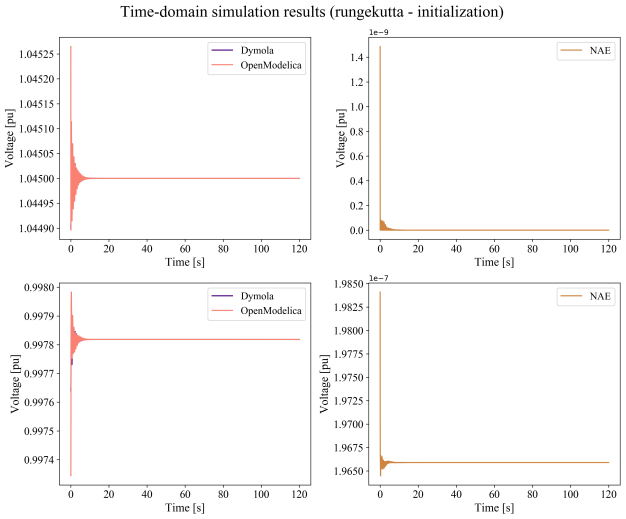

Figure 3. Comparison between Dymola and OpenModelica results for the initialization scenario using the *runge kutta* solver

The non-steady state behavior at the on-set of the simulation is due to the fact the initial guess used in the model (the so-called power flow) is not close enough to an equilibrium for the initialization routine to solve for a more precise set of initial values. A more complex initialization problem will better benchmark the capabilities of the tools. Despite this, Dymola and OM produce almost the same results, with an NAE in the order of 10^{-3}.

Likewise, for the *runge kutta* solver, Figures 4 and 5 show the simulation results for the line opening (voltage magnitude at buses 2 and 4) and the double bus fault (voltage at affected buses 4 and 14) scenarios, respectively. Both Figures reveal how there is a minimal error between the results of both tools. Based on these results, it is concluded that fixed-step solvers can be applied to reduce discrepancies between different Modelica tools.

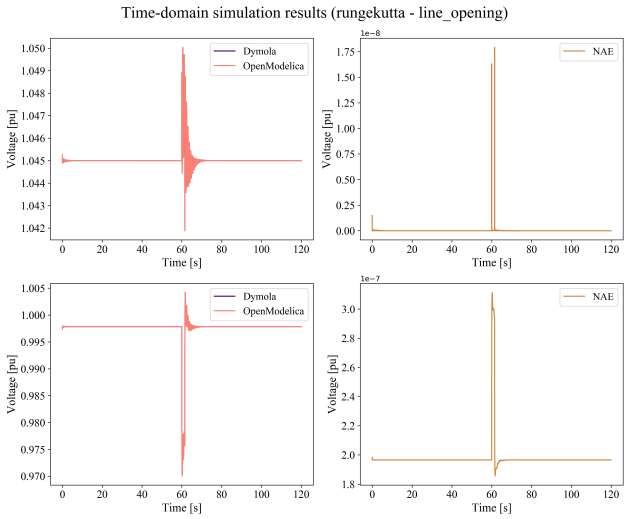

Figure 4. Comparison between Dymola and OpenModelica results for the line opening scenario using the *runge kutta* solver

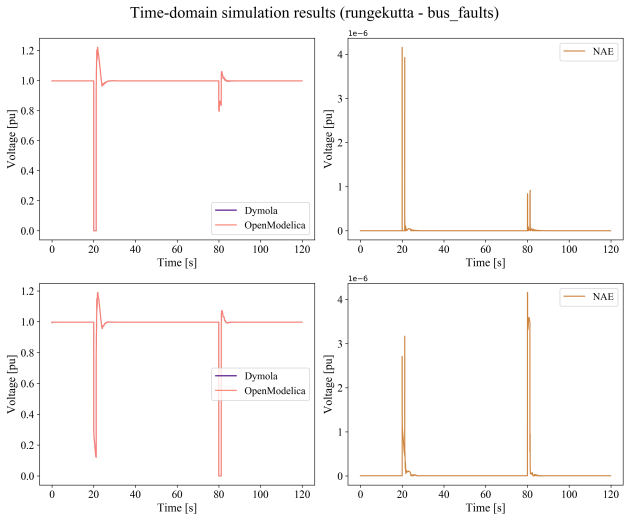

Figure 5. Comparison between Dymola and OpenModelica results for the double bus fault scenario using the *runge kutta* solver

The complete collection of plots for all solvers and simulation scenarios can be found online in the GitHub repository in the Notebook `02_Data_PostProcessing_SimulationResultPlotting.ipynb`.

4.1 Simulation Time

The information regarding simulation time is presented for all scenarios and solvers in Table 2. We must underline that simulation time includes compilation, translation and actual integration (execution time).

A clear conclusion from this information is that the variable-step solver is the most convenient for an initial analysis of the conditions of the system with an important amount of detail. Nevertheless, considering the information about MSE, a fixed-step solver shows advantages to reduce the numerical discrepancy between tools running the same model. The cost is a considerable increase in simulation time.

4.2 CPU Utilization

Since each instance of Dymola/OpenModelica was constrained to run only on one core, we expect exactly one processor to be responsible for numerical integration while a simulation is being carried out. The CPU usage of the assigned execution core is 100% due to the heavy numerical task of the simulation.

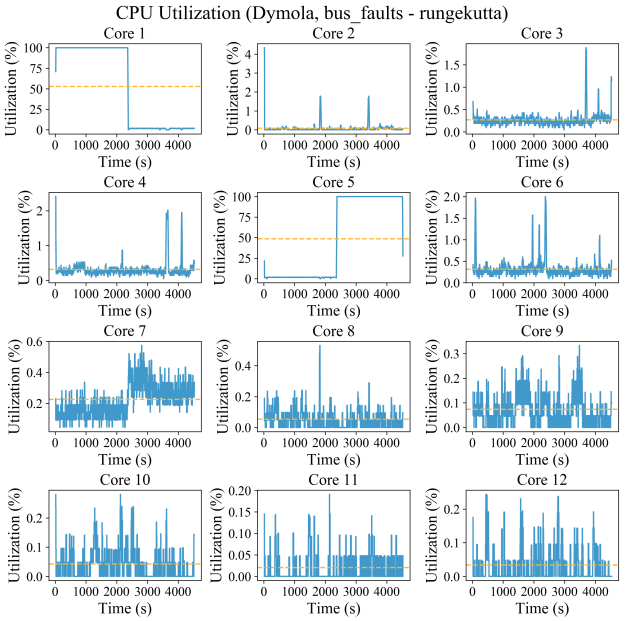

Figure 6. CPU Utilization for Dymola during bus fault scenario with *runge kutta* solver.

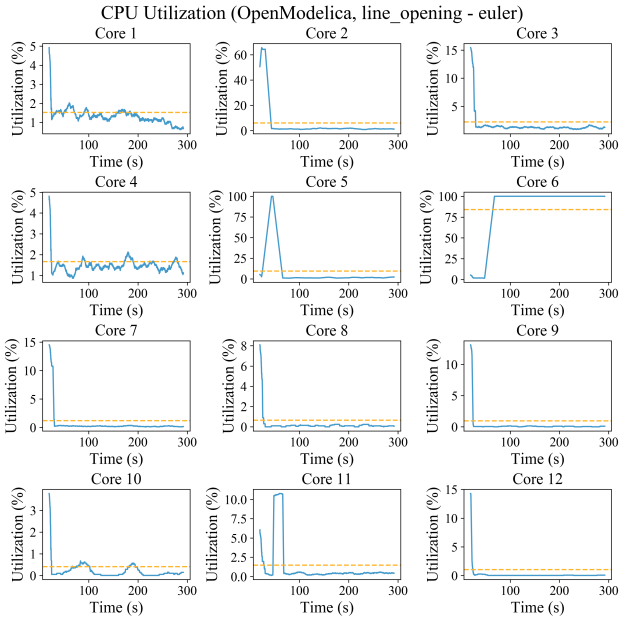

Figure 7. CPU Utilization for OpenModelica during line opening scenario with *euler* solver.

An interesting outcome of our experiments is that several CPUs are involved in the execution process but just one is performing the simulation tasks at a given time. We can detail this behavior in Figure 6 for a Dymola simulation using the *runge kutta* method of the bus fault sce-

nario. Simulation starts in Core 1 where the CPU usage is at a 100% at the beginning of the running time. Afterward, it is delegated to Core 5. Finally, Core 10 completes the execution of the program. This behavior is due to a task scheduling routine in the processor level that dispatches to different cores the compilation, translation, and integration sub tasks.

Similar behavior happens with another solver and OpenModelica (Figure 7) in which the simulation started in Core 2, then was briefly assigned to Core 5 and was finished in Core 6. All the graphics can be detailed in the GitHub repository inside the Jupyter Notebook called `03_DataPostprocessing_CPU_Usage.ipynb`.

5 Performance Evaluation Metrics

A score was proposed to quantify the performance differences between the tools and the solvers. The score is obtained from the data generated for all simulations and solvers. This single metric makes it simpler to directly compare the performance of Dymola versus OpenModelica. From Table 2 the Execution Time (ET) for each scenario and solver were employed. These metrics were obtained directly from the program logs and measured in Python. Notice that the time registered using OMPython is slightly larger than the reported by the simulation log due to the communication interface between Python and OM. The translation and compilation time were not taken into account since this information is only available in the Dymola developer version, not in the release version.

The Normalized Minimum Execution Time score (NMT) of each scenario per solver is computed as

$$\mathrm{NMT}^{[\mathrm{solver}]} = \frac{\min(\mathrm{ET_D}, \mathrm{ET_{OM}})}{\mathrm{ET_{observed}}} \tag{2}$$

where $\mathrm{ET_{observed}}$ is the ET for a particular solver in Dymola or OpenModelica), and $\min(\mathrm{ET_D}, \mathrm{ET_{OM}})$ is the minimum execution time between both tools for a specific solver. Clearly, $\mathrm{NMT}^{[\mathrm{solver}]}$ lies between 0 and 1. The higher the NMT is, the faster the simulation will run for a particular selected solver. At a first glance, this metric might be counter-intuitive since a better solver/tool combination would reduce execution time. However, we propose an increasing score metric due to the fact that users are more familiar to higher scores for better performance. Therefore, the larger the NMT is, the faster a particular solver will run.

The NMT metric results are presented in Table 4. The performance of Dymola is remarkably better using *dassl*. Nevertheless, OM shows a smaller execution time than Dymola for fixed-step solvers (as can be seen from Table 2, the NMT scores and 5). This conclusion can be further detailed in Table 5 where a direct comparison between the execution time for the tools with the different solvers for each scenario is presented.

The NMT scores highlight that the performance of Dymola in terms of execution time is remarkably better for variable-step solvers. The relative advantage of selecting one tool with respect to the other can be computed from the NMT directly. For instance, Dymola runs 47.3x faster than OM for the first scenario using *dassl* which can be computed by a direct comparison of the ET listed in Table 5. The NMT score of OM for $\mathrm{S_1}$ is 0.0211 which is $1/0.0211 = 47.3$ times smaller than the corresponding Dymola metric reflecting the relative difference in execution time.

For the variable-step solver, the discrepancy between the tools can be attributed to the performance of the *dassl* solver in all simulations thanks to the aforementioned improvements for `DAEMode` inside Dymola (Henningsson et al., 2019).

The execution time of OM is faster than the one of Dymola for all scenarios when a fixed-step solver is used. The NMT scores show a relative advantage between 3.4 and 6.8 times favoring OM. We have contacted Dassault Systèmes about the performance of the simulations of the IEEE 14 Bus System using *runge kutta* methods (including *euler*) as integrator and GCC for compilation. Dassault reports that bug fixes have been made in the GCC runtime libraries, leading to CPU times are about 3 – 4 times faster, on par with the run times given when compiling with Visual Studio under Windows 10. Dassault informs that the updated libraries will be part of Dymola 2021.

We should point out that the scope on the potential optimization features has been limited to the use of the flag `Evaluate = true` in Dymola and `-d=evaluateAllParameters` in OM, which is standard practice when attempting to improve simulation performance.

The detailed step-by-step computations of the scores can be found in GitHub in the Notebook `05_BenchmarkMetrics.ipynb`.

6 Conclusions and Future Work

The paper presents a concrete analysis of the time-domain simulation performance of Modelica-based tools for different solvers in the context of large-scale nonlinear dynamic systems. The presented results can help a user to choose a tool depending on the final application, and lead to improvements in Modelica tools. The methodology of this benchmark can be extended to virtually any platform or Modelica tool.

We benchmarked the time-domain simulation performance of two popular Modelica tools, Dymola and OpenModelica, for a dynamic power system simulation using the IEEE 14 bus system. We considered several scenarios that challenge numerical solvers differently. Thanks to Python scripting, we were able to change automatically the simulation settings while directly measuring the performance of the computer instead of relying on simulation logs. Python functions also made it quicker to analyze straightforwardly the big set of data regarding simulation

results and computer performance.

For the proposed heuristic score, we found out that OpenModelica performs better than Dymola in terms of execution time for fixed-step solvers while Dymola shows faster results when using a variable-step solver (see Table 5). Despite this, we must warn the reader that this conclusion is based upon only a particular system. Further research has to be done to include more test systems. Moreover, the use of a fixed-step solver has the main advantage of

The tool and solver benchmark results are expected to be reproduced in a larger system such as the Nordic 44 with $\approx$ 1300 states and 6300 variables. This system requires a considerable amount of RAM given its large number of states. Therefore, future work is related to the performance analysis in a 64-core machine with 512 GB of RAM for the N44 system considering different types of simulations and various initialization parameters.

Acknowledgments

This work was funded in part by New York Power Authority (NYPA) and the New York State Energy Research and Development Agency (NYSERDA) through the Electric Power Transmission and Distribution (EPTD) High Performing Grid Program under agreement 137940.

The authors would like to thank Erik Henningsson from Dassault Systèmes for his valuable contributions for the improvement of the results of this paper.

References

Maxime Baudette, Marcelo Castro, Tin Rabuzin, Jan Lavenius, Tetiana Bogodorova, and Luigi Vanfretti. OpenIPSL: Open-Instance Power System Library - Update 1.5 to iTesla Power Systems Library (iPSL): A Modelica library for phasor time-domain simulations. *SoftwareX*, 7:34–36, jan 2018. ISSN 23527110. doi:10.1016/j.softx.2018.01.002.

Willi Braun, Francesco Casella, and Bernhard Bachmann. Solving large-scale Modelica models: new approaches and experimental results using OpenModelica. In *Proceedings of the 12th International Modelica Conference, Prague, Czech Republic*, pages 557–563, jul 2017. doi:10.3384/ecp17132557. URL `http://www.ep.liu.se/ecp/article.asp?issue=132{%}26article=63`.

J. L Devore and K. N. Berk. *Modern Mathematical Statistics with Applications*. Springer-Verlag New York, 2012.

Peter Fritzson, Peter Aronsson, Adrian Pop, Hakan Lundvall, Kaj Nystrom, Levon Saldamli, David Broman, and Anders Sandholm. Openmodelica-a free open-source environment for system modeling, simulation, and teaching. In *2006 IEEE Conference on Computer Aided Control System Design, 2006 IEEE International Conference on Control Applications, 2006 IEEE International Symposium on Intelligent Control*, pages 1588–1595. IEEE, 2006.

Erik Henningsson, Hans Olsson, and Luigi Vanfretti. DAE Solvers for Large-Scale Hybrid Models. In *Proceedings of the 13th International Modelica Conference, Regensburg, Germany*, pages 491–502, feb 2019. doi:10.3384/ecp19157491. URL `http://www.ep.liu.se/ecp/article.asp?issue=157{%}26article=50`.

Bernt Lie, Sudeep Bajrachary, Alachew Mengist, Lena Buffoni, Arun Kumar, Martin Sjölund, Adeel Asghar, Adrian Pop, and Peter Fritzson. Api for accessing openmodelica models from python. In *Proceedings of The 9th EUROSIM Congress on Modelling and Simulation, EUROSIM 2016, The 57th SIMS Conference on Simulation and Modelling SIMS 2016*, pages 707–714. Linköping University Electronic Press, 2018.

Federico Milano. *Power System Modelling and Scripting*, volume 54 of *Power Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. ISBN 978-3-642-13668-9. doi:10.1007/978-3-642-13669-6. URL `http://link.springer.com/10.1007/978-3-642-13669-6`.

Appendix

In this appendix, all performance results of the different simulation experiments are presented. In Table 5, *runge-kutta* is abbreviated as rk.

		Simulation Time OpenModelica (OM)				
		Translation	Compilation	Execution	Total Time (OM log)	OMPython
S_1	dassl	2.3204 s	6.6270 s	7.8690 s	16.8164 s	19.3451 s
	euler	2.5432 s	6.5845 s	277.5495 s	286.6772 s	289.2878 s
	rk	2.3495 s	6.6213 s	783.0159 s	791.9867 s	794.6805 s
S_2	dassl	2.6079 s	6.6411 s	13.4004 s	22.6494 s	25.2542 s
	euler	2.4023 s	6.6437 s	310.1061 s	319.1521 s	321.7222 s
	rk	2.3489 s	6.6591 s	1086.3958 s	1095.4040 s	1098.0253 s
S_3	dassl	2.1952 s	6.7301 s	163.4884 s	172.4137 s	175.2962 s
	euler	2.3248 s	6.7801 s	378.6069 s	387.7118 s	390.3140 s
	rk	2.3960 s	6.7332 s	1344.6808 s	1353.8100 s	1356.2994 s

		Simulation Time Dymola		
		Translation + Compilation	Execution	Measured Python
S_1	dassl	20.186 s	0.1664 s	20.3524 s
	euler	24.7791 s	1880.0109 s	1904.7900 s
	rk	21.2389 s	4420.0125 s	4441.2514 s
S_2	dassl	20.2363 s	0.34082 s	20.5772 s
	euler	19.6561 s	1850.0129 s	1869.6690 s
	rk	24.6567 s	4410.0119 s	4434.6686 s

		Simulation Time Dymola		
		Translation + Compilation	Execution	Measured Python
S_3	dassl	20.2161 s	14.4098 s	34.6260 s
	euler	16.4581 s	1820.0119 s	1836.47 s
	rk	17.9956 s	4590.0129 s	4608.0085 s

Table 2. Execution time for Dymola and OpenModelica for each simulation scenario using different solvers.

		Mean Squared Error (MSE)	
		B2	B4
S_1	dassl	3.0011×10^{-11}	4.6482×10^{-11}
	euler	1.2894×10^{-11}	3.7950×10^{-11}
	rk	1.2853×10^{-11}	3.7828×10^{-11}
S_2	dassl	1.1728×10^{-8}	1.1267×10^{-7}
	euler	2.3598×10^{-10}	3.2470×10^{-9}
	rk	2.3579×10^{-10}	3.2473×10^{-9}

		Mean Squared Error (MSE)	
		B1	B4
S_3	dassl	0.0067	0.0002
	euler	0.0025	0.0002
	rk	0.0018	0.0002

Table 3. Mean Square Errors between voltage magnitude signals at different buses for each simulation scenario.

	Dymola			OpenModelica		
	$NMT^{[S_1]}$	$NMT^{[S_2]}$	$NMT^{[S_3]}$	$NMT^{[S_1]}$	$NMT^{[S_2]}$	$NMT^{[S_3]}$
dassl	1	1	1	0.0211	0.0254	0.0880
Euler	0.148	0.168	0.208	1	1	1
rk	0.177	0.296	0.293	1	1	1

Table 4. Normalized Minimum Execution Time scores.

		Execution Time (ET)		
		OM	Dymola	Result
S_1	dassl	7.869 s	0.1664 s	D > OM (47.3x)
	euler	277.54 s	4420.01 s	OM > D (6.8x)
	rk	783.01 s	1880.01 s	OM > D (5.6x)
S_2	dassl	13.40 s	0.3408 s	D > OM (39.3x)
	euler	310.10 s	1850.01 s	OM > D (6.0x)
	rk	1086.39 s	4410.01 s	OM > D (4.1x)

		Execution Time (ET)		
		OM	Dymola	Result
S_3	dassl	163.48 s	14.40 s	D > OM (11.3x)
	euler	378.60 s	1820.01 s	OM > D (4.8x)
	rk	1344.68 s	4590.01 s	OM > D (3.4x)

Table 5. Comparison between execution time in OM and Dymola for different solvers.