Speeding Up Design Space Exploration through Compiled Master Algorithms

Ken Pierce[1], Kenneth Lausdahl[2], and Mirgita Frasheri[2]

[1] School of Computing, Newcastle University, United Kingdom
kenneth.pierce@ncl.ac.uk
[2] Dept. of Electrical and Computer Engineering, DIGIT, Aarhus University, Aarhus, Denmark
kenneth@lausdahl.com,mirgita.frasheri@ece.au.dk

Abstract. The design of Cyber-Physical Systems (CPSs) is complex, but can be mitigated through Model-Based Engineering. The models representing the components of a CPS are often heterogeneous, combining cyber and physical elements, and can be produced in different formalisms by engineers from diverse disciplines. To assess system-level properties of a specific design, the joint behaviour of such components can be analysed through co-simulation, whereas different design alternatives can be compared through Design Space Exploration (DSE). Due to its combinatorial nature, the DSE can suffer from state-space explosion, where the number of combinations rapidly increases the time required to analyse them. While careful experiment design and Genetic Algorithms can be used to reduce the number of simulations, the benefit of speeding up co-simulations is clear. In this paper, we present an extension of the FMI-compliant Maestro co-simulation engine that generates a custom co-simulation Master algorithm as C code that can be compiled and run natively, reducing the overheads from the existing Java-based version. We apply this to a standard water tank case study and show an initial speed up of five times over the existing Maestro implementation.

1 Introduction

Cyber-Physical Systems (CPSs) are systems constructed of interacting hardware and software elements, with components networked together and distributed geographically [6]. The development, deployment and maintenance of CPSs requires multiple disciplines to work together, which is often hampered by the organisational, social and technical barriers between engineering disciplines, including use of different terminologies, techniques and tools. Modelling such systems, and analysing these models with techniques such as simulation, are increasingly used in the development of CPSs. Combining models from different disciplines to create system-level "multi-models" has been shown as one way to bring together these disciplines and to permit better analysis of CPSs [1].

INTO-CPS [5] is a tool chain for model-based design of CPSs based around this concept of multi-models, using the Functional Mock-up Interface (FMI) standard [8] (FMI). The FMI standard provides a way for individual models to be packaged as Functional Mock-up Units (FMUs), with a standard interface for inputs and outputs. A multi-model therefore represents a CPS through a combination of FMUs representing the various components or parts of the system of interest, and the connections between them

(inputs and outputs that influence other FMUs in the multi-model). A multi-model can be analysed in various ways including static checking and model checking. The most common way is through *co-simulation*, where the FMUs are executed simultaneously under the control of a *Master Algorithm (MA)* that is responsible for advancing (simulated) time and passing inputs and outputs between FMUs. The INTO-CPS tool chain is centered around a co-simulation engine called Maestro [11] that provides a variety of Master algorithms.

One main benefit of modelling CPSs is that it enables investigation of different designs before committing time and resources to physical prototypes. A design is characterised by its *design parameters*, which are properties of the CPS that affect its behaviour (both physical properties and those of software). The set of all possible design parameters defines a *design space*. *Design Space Exploration* (DSE) is the act of assessing one or more areas of the design space by analysing and ranking a set of designs automatically, with the aim of helping the engineering team to make better informed decisions about which designs are most promising.

DSE is one feature of the INTO-CPS tool chain. INTO-CPS includes a set of Python scripts that allow an engineer to define a design space and to perform DSE through either exhaustive search (analysing all combinations of designs defined by a set of parameters) or genetic search (iteratively selecting promising designs from an initial set). Even with trivial multi-models that co-simulate quickly, large DSEs with many combinations of parameters rapidly increase the required number of co-simulations making DSE time and resource intensive.

Given the pressures of industry, it is important that decisions on designs be made quickly and effectively. While DSE can give a lot of useful information, examining combinations of even a few parameters can take a long time. This paper describes work on improving the speed of co-simulation by generating an MA to native C++ code, which in turn speeds up co-simulation and allows a given design space to be explored faster, or a larger design space to be explored in a similar time.

The remainder of this paper is structured as follows. Section 2 describes the existing DSE framework in INTO-CPS. Section 3 introduces the new code-generation feature. Section 4 provides a refresher on the standard water tank case study. Section 5 shows the performance of the new feature. Finally, Section 6 presents some conclusions and future work.

2 Design Space Exploration with INTO-CPS

Design Space Exploration (DSE) involves running multiple co-simulations, each of which represents a different design, and analysing the results in some form to provide the engineer with evidence for making design decisions about which designs are most promising [3]. A common way to perform a DSE is to run multiple simulations and sweep across one or more design parameters. Each design is characterised by a set of design parameters, which do not change during a co-simulation, but take different values for each design [1].

Defining a DSE then involves selecting which parameters will be changed, what values will be explored, and in what combination. Design parameters could be numer-

ical, in which case could be defined through minimum, maximum and step size, or they could be defined as an enumerated set of possible values. Combinations can be constrained, such as ensuring that one parameter is always greater than another. The number of possibilities defined by a given combination defines the size of the design space for a given DSE, and hence the number of co-simulations that must be run.

After a co-simulation has run, the results are saved as Comma-Separated Values (CSV) files, and can be analysed. This is accomplished by defining one or more objective functions, which represent metrics by which the design is judged. An objective function can be defined programmatically, for example by providing a Python script that computes a score from the CSV and any additional data, such as details of the scenario. Once all co-simulations in a DSE have run, the designs can be ranked based on the objectives.

A clear issue with DSE is state space, or in this case design space, explosion. As the number of parameters and values increases, the number of combinations increases exponentially. As mentioned below, the main ways to explore design spaces further using the same compute and time resources are:

1. To perform careful experiment design in order to focus on the most likely outcomes, for example by selecting combinations that demonstrate the most important parameters [3];
2. To explore the design space iteratively by running only some co-simulations initially, then deciding based on those results which are the most promising designs. This can be achieved automatically through genetic algorithms, for example [9]; or
3. To speed up the co-simulation itself by optimising the FMUs and/or Master algorithm.

The DSE features of the INTO-CPS tool chain take the form of a set of Python scripts written to use the Maestro co-simulation engine to carry out experiments with multiple co-simulation runs. These scripts were originally developed in the INTO-CPS project [4] using Python 2.7. These original Python scripts included an exhaustive search which computes all possible combinations of parameters given (excluding those that don't meet the constraints). The scripts also contained a basic genetic algorithm to select which designs to run after performing analysis at each step in order to search down and reduce the overall number of simulations

Since the INTO-CPS project has finished, a number of people have improved upon them. The most recent version now works with Python 3 and includes support for parallelization [9]. Other recent papers have also looked at using libraries to achieve similar results with particle swarm optimization and simulated annealing [10]. While different designs could also include different FMUs, the current scripts do not support sweeping across different FMUs, since care must be taken to understand what should happen if those have diverse design parameters.

3 Code Generation Extension

Speeding up co-simulation speed is crucial to the efficiency of DSE as it is directly linked to the size of the design space that can be explored in a feasible manner, as described in Section 2. In general there are a number of ways to speed up a co-simulation:

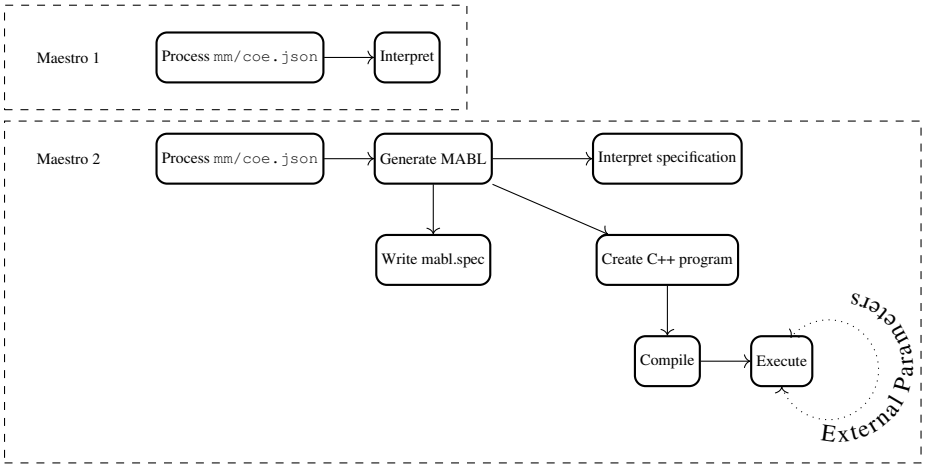


Fig. 1: Maestro 1 and 2 execution flow

1. Increase the step size (so fewer co-simulation steps are taken);
2. Reduce the complexity of the simulation (simplify the FMUs and their connections); or
3. Optimise the Master Algorithm

While the first two options have the largest potential for speeding up co-simulations, they both have the drawback that they require changes to the work already carried out, which may not be possible while retaining the required fidelity. Other circumstances, such as a need to use legacy models, can also limit the ability of the engineer to optimise the individual FMUs. Therefore, this paper focuses on the third option, which is a generic approach that significantly improves the co-simulation speed of the MA itself.

Maestro[3] [12] is a tool which implements a co-simulation MA as a Java application with support of multiple orchestration strategies. The earlier versions (maestro1) had good performance at the time, however as shown in Figure 2, performance degrades over time and incurs a high initial cost for each co-simulation run. Therefore a new internal architecture of Maestro was introduced by Thule et al. [13] (maestro2). This new version separates the construction of the MA from the specific co-simulation execution as shown in Figure 1. This was implemented as a extensible Java application that could construct an MA and includes an interpreter for running co-simulations. This version matches the behaviour of the initial Maestro implementation, but with better performance without degraded performance over time (seen also in Figure 2). A main factor for the improved performance is that all relations between signals are resolved during specification generation and not during run-time. Listing 1.1 illustrates how Maestro can be used to produce the intermediate representation (mabl) of MA and how that can be interpreted.

[3] Maestro was formally known as the INTO-CPS Co-simulation Orchestration Engine or COE.

```
# Generate the mabl spec
java -jar $mabl import sg1 -fmu-search-path FMUs -output . \
    Multi-models/mm/co-sim-51/mm.json \
    Multi-models/mm/co-sim-51/co-sim-51.coe.json

# Interpret the spec using JAVA
java -jar $mabl interpret -runtime spec.runtime.json spec.mabl
```

Listing 1.1: Creation of MA and interpretation

```
# Generate the mabl spec
java -jar $mabl import sg1 -fmu-search-path FMUs -output . \
    Multi-models/mm/co-sim-51/co-sim-51.coe.json \
    sim-dse/mm.json

# Interpret the spec using JAVA
java -jar $mabl interpret -runtime spec.runtime.json spec.mabl

# Generate native cpp simulator
java -jar $mabl export cpp -output cpp \
     -runtime spec.runtime.json \
     spec.mabl

# CMake
cmake -Bcpp/program -Scpp

# Compiling
make -Ccpp/program -j9

# Run native simulation
./cpp/program/sim -runtime spec.runtime.json
```

Listing 1.2: Generation and execution of native simulation from of MA

The INTO-CPS project defines two types of files to configure a co-simulation, `mm` and `coe`. These are JSON (Javascript Object Notation) files that define the FMUs, their connections, and co-simulation properties. The `spec.runtime.json` defines properties external to the co-simulation, such as the path to the output file.

In this paper we extend this work with the ability to make parameters external to the MA. This allows reuse of the MA across DSE runs by storing the parameters in the `spec.runtime.json` configuration. This enables the MA to be converted to a C++ application, thus removing the overhead of a Java interpreter and only having the compilation overhead once per exploration by effectively generating a custom, native MA for a given co-simulation. This significantly improves the execution time but adds a one-time extra compilation overhead which on average takes longer than starting the Java process used for interpretation. The compilation overhead is directly related to the systems CMake performance capabilities and the download of external libraries, where the latter can be avoided if preinstalled. Once the make files for the native MA are ready

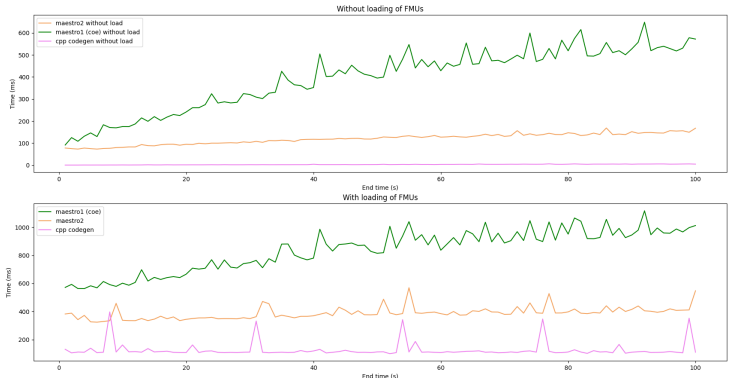


Fig. 2: Performance comparison between Meastro (COE), the new Meastro and the native C++ program.

for compilation, it is very fast to compile the native MA source file or any further MA source files. Because of this it is clear that the benefit of this native MA increases with the design space size.

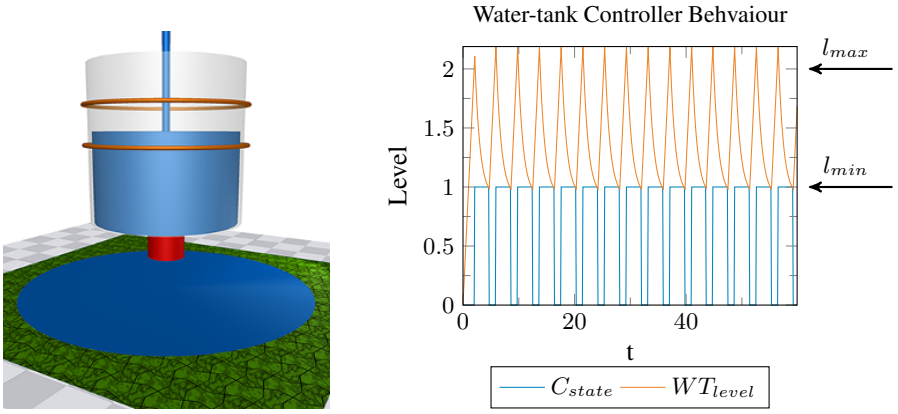
The new converter is implemented as an extension to Maestro and thus runs in Java. The converter outputs a CMake project. The project includes a Maestro-specific library for FMI and external dependencies to `libzip` and `rapidjson` and a single generated `co-sim.cxx` file representing the current MA. The CMake initialisation will fetch all dependencies and the subsequent compilation will compile all into a single executable `sim`. This process can be seen in Listing 1.2. It consumes the same input files as the interpreter on line 7 from Listing 1.1. This corresponds the lower right part of Maestro 2 in Figure 1.

The CMake project supports all major FMI target platforms Linux, MacOS and Windows using MSYS with mingw. For DSE purposes the process can be further improved by reusing a CMake generated project for subsequent explorations to avoid running the slow CMake initialisation and compilation of the external zip library. The only change for a new generation with the same version of the tool will be `co-sim.cxx`. Re-compilation of `co-sim.cxx` is significantly faster than re-running CMake.

4 Case Study: Single-tank Water Tank

The single-tank water tank example [7] is used as a case study in this paper to demonstrate the speedup of the proposed approach. The system, is composed of two FMUs, namely a water tank and a controller FMU. The water tank FMU models a physical water tank component, composed of a valve with two states (open/close) and a sensor that outputs the level of the water in the tank. The controller FMU represents the digital component that controls the behaviour of the water tank through the valve, based on the current level of water.

The logic of the controller is straightforward: it attempts to keep the water level between a defined minimum and maximum at all times (visible in Figure 3a). Should the water level go below a user defined minimum l_{min}, the controller closes the valve.



(a) (b)

Fig. 3: Visualisation of the water tank (left) and controller behaviour (right) over co-simulation time t (adapted from [2]), where C_{state} refers to the state of the controller, 1 for open, 50 for closed, and WT_{level} refers to the water level in the tank)

In case the level of water goes above a user defined maximum l_{max}, the controller opens the valve. The behaviour of the system is depicted in Figure 3b for $l_{min} = 1$ and $l_{max} = 2$. The minimum and maximum levels are used as design parameters in the DSE, with the constraint that minimum must be strictly below the maximum, $l_{min} < l_{max}$.

5 Results

To investigate the speed up with the native MA over the existing Java version of Maestro, a set of DSEs were run with differing design space sizes (i.e. number of combinations and hence co-simulations) and simulated durations (i.e. number of co-simulation steps needed to complete the co-simulation). The water tank multi-model introduced above served as the design, using native C FMUs for both the water tank (singlewatertank-20sim.fmu) and controller (watertankcontroller-c.fmu). Version 0.4.1[4] of the DSE scripts were used as the baseline. For the native MA run, the scripts were modified to call the compiled executable instead of calling the COE. This required a simple change to the `Common.py` scripts, shown in Listing 1.3. Additional small changes were needed in `Output_CSV.py` and `Output_HTML.py` files to handle different naming conventions in the results generated by the experimental native MA.

The changes made however are not robust and did not include calls to generate or compile the code. To fully integrate the option for native compilation, the scripts need to be refactored, ideally with a layer of abstraction to handle the two paradigms transparently to the higher-level DSE algorithm scripts. For example, various options for the co-simulation are passed at run-time in the existing version (such as the co-simulation end time) but are required at compile time in the native C++ version.

[4] `github.com/INTO-CPS-Association/dse_scripts/releases/tag/0.4.1` (January 2021)

Listing 1.3: Calling of the native co-simulation in the DSE scripts

```
runTimeJson = {}
runTimeJson["environment_variables"] =
  parsedMultiModelJson["parameters"]
runTimeJson["DataWriter"] = [
  {
    "filename": os.path.join(simFolderPath, "results.csv"),
     "type": "CSV"
  }
]
jsonOutput = json.dumps(runTimeJson,
  sort_keys=True, indent=4, separators=(",", ":"))
jsonOutputFile = open(filePath, 'w')
jsonOutputFile.write(jsonOutput)
jsonOutputFile.close()
subprocess.run(
  ["../sim-dse/cpp/program/sim.exe", "-runtime", filePath])
```

To try and estimate the scalability of both approaches, a range of orders of magnitude were chosen for both the size of the design space and end time of the co-simulations:

- Design space size: 1, 10, 100, and 500 and 1000 combinations; and
- Simulated duration (s): 1, 10, 100, 1000 and 10000 simulated seconds.

While 1000 combinations was attempted for the Java version, this taxed the memory and disk capacity of the test computer and was abandoned after several attempts and crashes. The test machine was a laptop with an Intel® Core™ i7-5600U Processor (4M Cache, up to 3.20 GHz), 8GB DDR3L-12800 1600 MHz and an M2 solid state drive running Windows 10 Pro (21H2). The native MA was compiled under the MSYS2 environment (a collection of tools for building native Windows app from a Bash-like environment) using gcc 11.2.0-10, make 4.3-1 and cmake 3.23.0-1.

Table 1 shows the overall execution times as recorded by the DSE scripts for each combination of design space and co-simulation length. Generation, compilation and re-compilation time are not included in the timings since these are not incorporated into the DSE scripts during these experiments. These incur the following penalties:

- Generate code from Maestro: 6 seconds
- Configure compilation with cmake: 2 minutes 34 seconds
- Compilation with make: 50 seconds
- Re-compilation after changing simulation duration: 6 seconds

Therefore, approximately 160 seconds should be added to the times in the Native columns. This means that for a one-off DSE with a design space of less than 100, the existing Java implementation is faster (indicated by the red colour in the columns Table 1). In practice, most engineers will run multiple DSEs on a single multi-model. For example, running a small DSE to check the objectives are being calculated correctly,

	Design space size (# of co-simulations)								
	1		10		100		500		1000
End time (s)	Java	Native	Java	Native	Java	Native	Java	Native	Native
1	5.00	0.46	35.60	4.14	264.64	43.50	1239.41	238.85	487.42
10	5.47	0.57	37.10	4.13	289.33	44.42	1290.90	237.45	493.73
100	6.50	0.48	39.48	4.21	271.24	45.86	1361.09	251.42	507.28
1000	10.37	0.61	41.84	5.88	329.07	63.00	1499.83	347.70	709.81
10000	15.17	2.16	113.59	22.48	885.56	239.38	3459.32	1181.84	2406.97

Table 1: Total time (s) to explore design spaces with increasing numbers of designs and simulated duration for the existing Java implementation and native version. The shaded columns indicate where a Native run is slower when accounting for the one-time cost of generating and compiling the code.

	Design space size			
End time (s)	1	10	100	500
1	11x	9x	6x	5x
10	10x	9x	7x	5x
100	13x	9x	6x	5x
1000	17x	7x	5x	4x
10000	7x	5x	4x	3x

Table 2: Relative speed-up of native MA over the existing Java implementation.

then running again with different parameter sets and sweeps. In such cases, due to the negligible re-compilation time, the initial cost of generation and compilation is amortised as more DSEs are run.

While the raw data is useful, it is easier to interpret after some processing. Table 2 shows the relative speed up of the native MA over the Java by dividing the Java execution time by the native execution time, and rounding to the nearest whole number. This shows that the native MA is significantly faster on single co-simulations, though the benefit is less on the 'longer' co-simulations. It is not immediately clear why this is. Similarly, as the design space increases, the speed up settles to around five times faster for the native MA. Given however that it was not possible to complete a 1000-design DSE using the Java version on the test machine, there may well be other factors at play, since memory usage was not examined in this test.

To further explore the data, they can be plotted on graphs. Given that there are two dependent variables, two graphs are plotted. The first is the effect of design space size on the time to complete the DSE, with the various end times as different series, shown in Figure 4. The second shows effect of end time on the time to complete a DSE, with

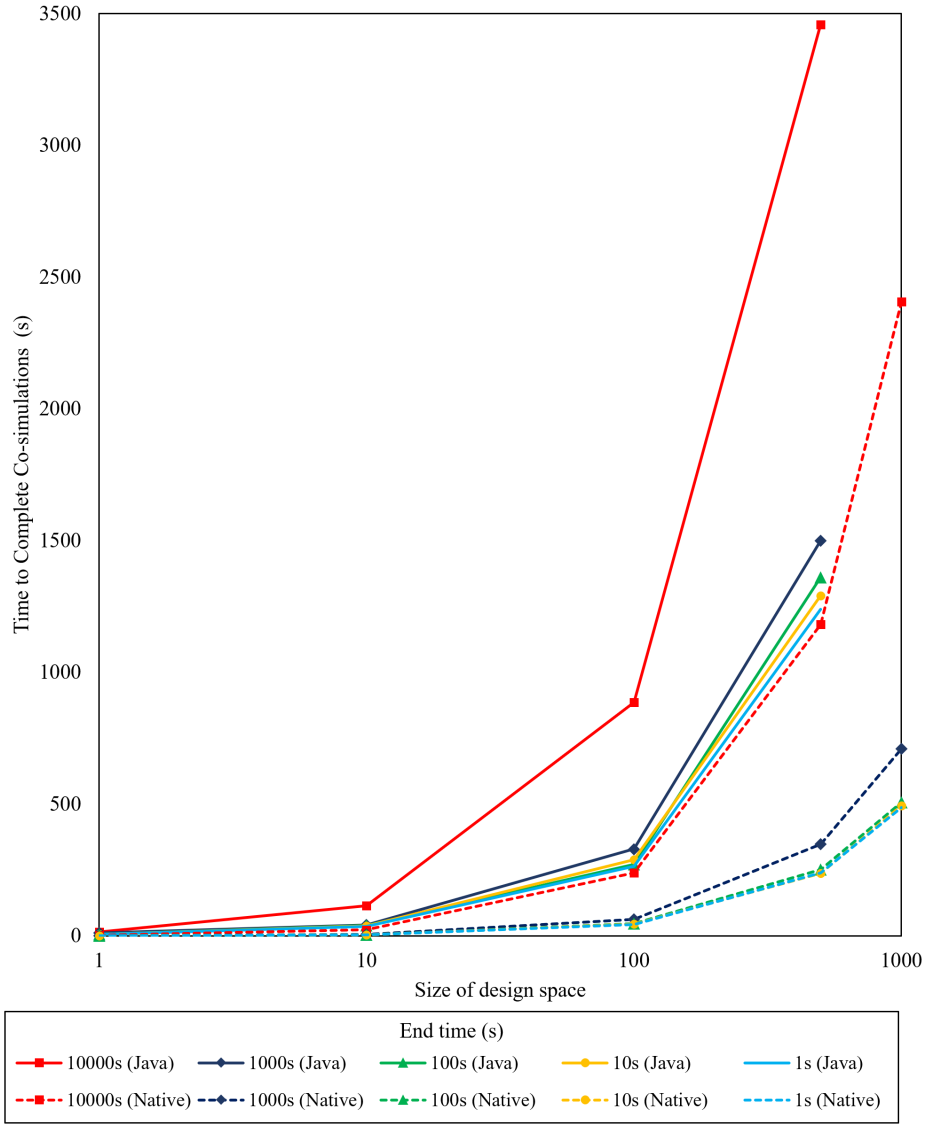


Fig. 4: Effect of design space size on time to complete DSE (using logarithmic x-axis).

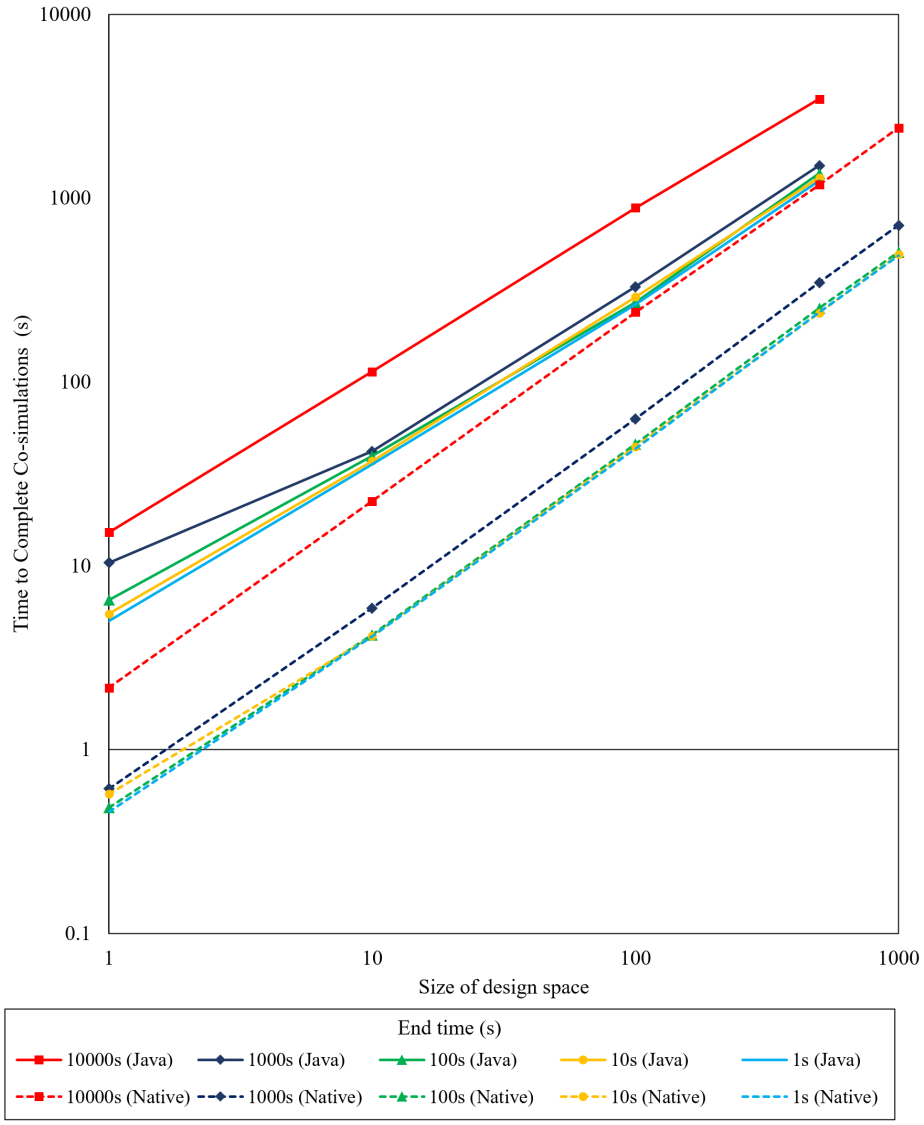


Fig. 5: Log-log plot showing effect of design space size on time to complete DSE.

the various sizes of design space as different series, shown in Figure 6. Both figures use a logarithmic x-axis. Similarly, in both figures, the Java version uses a solid line while the native MA uses a dashed line, then each pair has the same tick marks and colours.

Figure 4 confirms that for any given design space, the native version is faster. It also highlights the growth of the Java version for the largest design space, and suggests that were a 1000-design DSE completed it would reach into the hours range. The log-log plot in Figure 5 suggesting a clear power law relationship, however the data is not robust enough to draw any conclusions. The main takeaway from Figure 6 is the significant uptick in time taken for the longest co-simulation. The log-log plot in Figure 7 this is still apparent, so it would be useful to characterise this with more data.

Since this was only a single example with two FMUs, it would be important to try a range of examples and to run each multiple times to remove noise from the results. It is likely that there are some effects of the DSE scripts not being optimised for the native MA as well, since the changes made were experimental, there may well be superfluous files written to disk, for example.

When considering the various caveats from these initial results however, a cautious conclusion is that the native MA improves DSE by an order of magnitude. That is to say, by switching to the native MA, a DSE can be 10-times larger and the co-simulations can be 10-times longer for the a given time and compute cost. This is certainly of benefit and warrants further work to both characterise the speed up more robustly, and to create DSE scripts that can handle generation, compilation and use of the native MA functionality.

6 Conclusions and Future Work

In this paper we propose an approach that optimises the speed of execution of co-simulations. This is extremely useful in the context of Design Space Exploration (DSE). In such a context, the number of co-simulations to be explored can be rather high, making the speed of execution critical for time-bounded projects. In our work, we consider the Maestro co-simulation engine, an extensible Java application, able to construct an Master Algorithm (MA), and thereafter interpret it to execute the actual co-simulation. Our approach consists in converting the MA into a custom, native C++ application, thus reducing the overhead of the Java interpreter. The presented results show the benefit of such approach, however there are still some limitations related to the non-trivial setup, and the need for refactoring of the DSE scripts to allow for both types of execution, as well as open questions about the nature of the speedup achieved and relative scalability of the approach. Nevertheless, the achieved speedup creates opportunities that extend beyond DSEs defined by design parameter sweeps. Consider the case where for each FMU, there are a number of variants, and it is of interest to compare the behaviour of such variants in co-simulation. This means that there will be several multi-models, on which individuals DSEs should be performed, while also comparing the results across alternatives. In such scenarios, the number of co-simulations to be executed would increase rapidly and the approach presented in this paper may make such wider DSEs a realistic prospect. In future work, we will investigate such scenarios and validate the benefits from the approach proposed in this paper.

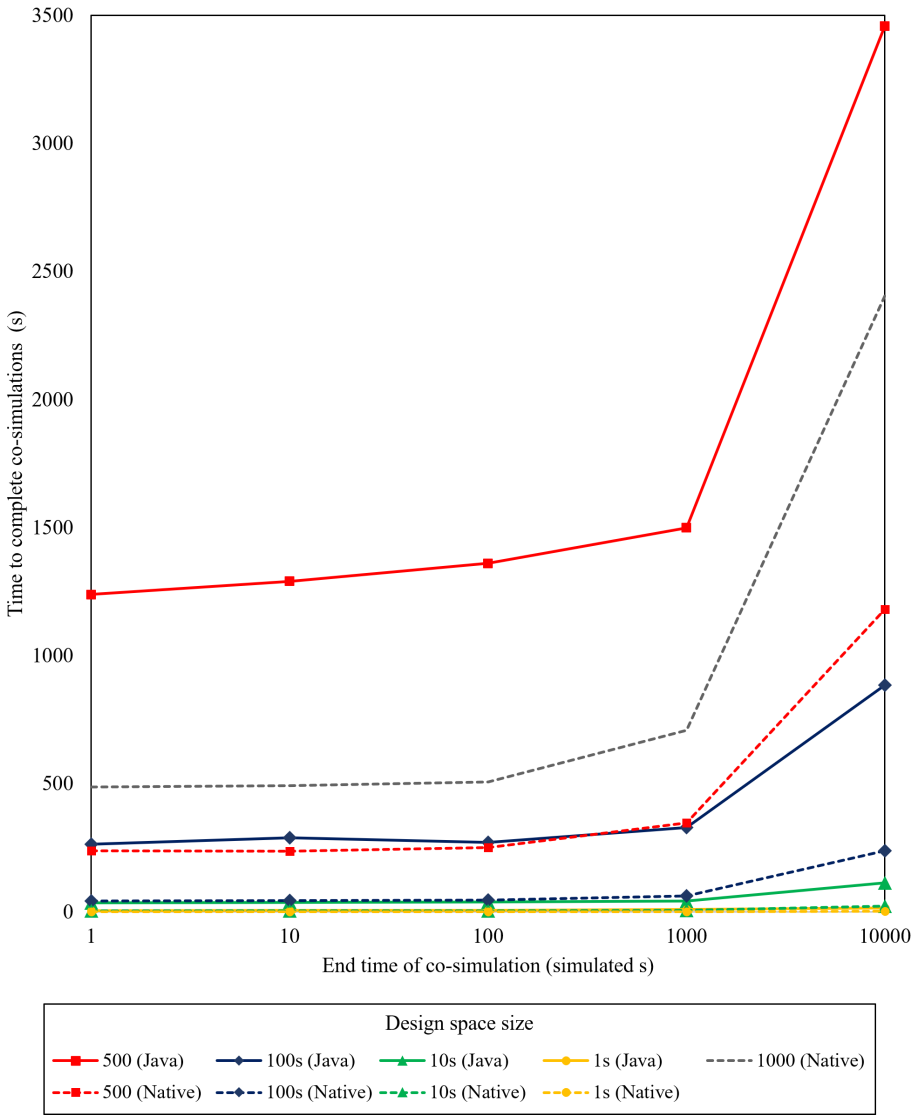


Fig. 6: Effect of end time on time to complete DSE (using logarithmic x-axis).

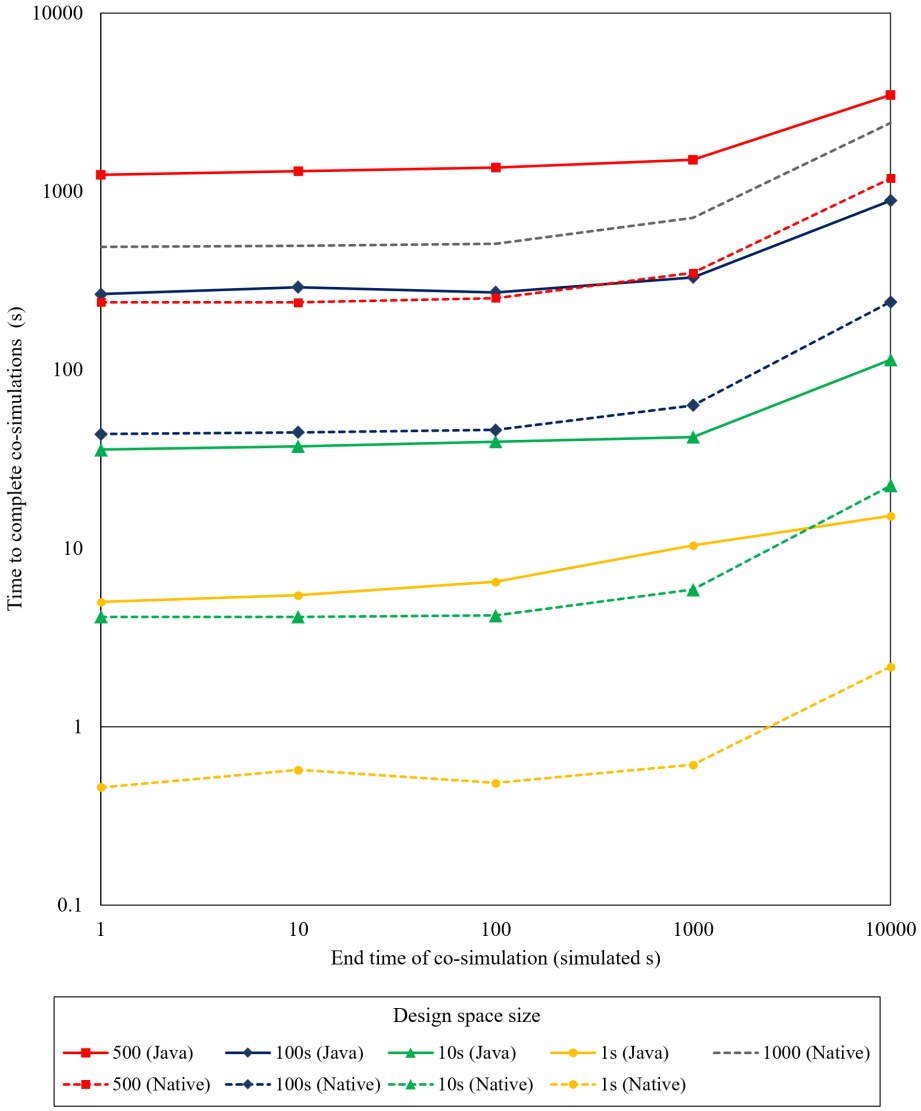


Fig. 7: Log-log plot showing effect of end time on time to complete DSE (using logarithmic x-axis).

Acknowledgements

We acknowledge the European Union's support for the INTO-CPS and HUBCAP projects (Grant Agreements 644047 and 872698). In addition we would like to thank Innovation Foundation Denmark for funding the AgroRobottiFleet project, and the Poul Due Jensen Foundation that funded our basic research for engineering of digital twins.

References

1. Fitzgerald, J., Larsen, P.G., Verhoef, M. (eds.): Collaborative Design for Embedded Systems – Co-modelling and Co-simulation. Springer (2014), http://link.springer.com/book/10.1007/978-3-642-54118-6
2. Frasheri, M., Thule, C., Macedo, H., Lausdahl, K., Larsen, P., Esterle, L.: Fault injecting co-simulations for safety (Nov 2021), http://icsrs.org/index.html, 5th International Conference on System Reliability and Safety, ICSRS 2021
3. Gamble, C., Pierce, K.: Design space exploration for embedded systems using co-simulation. In: Fitzgerald, J., Larsen, P.G., Verhoef, M. (eds.) Collaborative Design for Embedded Systems, pp. 199–222. Springer Berlin Heidelberg (2014)
4. König, C., Lausdahl, K., Niermann, P., Höll, J., Gamble, C., Mölle, O., Brosse, E., Bokhove, T., Couto, L.D., Pop, A.: INTO-CPS Traceability Implementation. Tech. rep., INTO-CPS Deliverable, D4.3d (December 2017)
5. Larsen, P.G., Fitzgerald, J., Woodcock, J., Gamble, C., Payne, R., Pierce, K.: Features of integrated model-based co-modelling and co-simulation technology. In: Bernardeschi, Masci, Larsen (eds.) 1st Workshop on Formal Co-Simulation of Cyber-Physical Systems. LNCS, Springer-Verlag, Trento, Italy (September 2017)
6. Lee, E.A.: Cyber Physical Systems: Design Challenges. Tech. Rep. UCB/EECS-2008-8, EECS Department, University of California, Berkeley (Jan 2008), http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-8.html
7. Mansfield, M., Gamble, C., Pierce, K., Fitzgerald, J., Foster, S., Thule, C., Nilsson, R.: Examples Compendium 3. Tech. rep., INTO-CPS Deliverable, D3.6 (December 2017)
8. Modelica Association: Functional Mock-up Interface for Model Exchange and Co-Simulation. https://www.fmi-standard.org/downloads (October 2019)
9. Rose, M., Fitzgerald, J.: Genetic algorithms for design space exploration of cyber-physical systems: an implementation in into-cps. In: Hugo Daniel Macedo, C.T., (Editors), K.P. (eds.) Proceedings of the 19th Overture Workshop. p. 96 (October 2021)
10. Stanley, A., Pierce, K.: Multi-objective optimisation support for co-simulation. In: Hugo Daniel Macedo, C.T., (Editors), K.P. (eds.) Proceedings of the 19th Overture Workshop. p. 96 (October 2021)
11. Thule, C., Lausdahl, K., Gomes, C., Meisl, G., Larsen, P.G.: Maestro: The into-cps co-simulation framework. Simulation Modelling Practice and Theory 92, 45 – 61 (2019), http://www.sciencedirect.com/science/article/pii/S1569190X1830193X
12. Thule, C., Lausdahl, K., Larsen, P.G., Meisl, G.: Maestro: The INTO-CPS Co-Simulation Orchestration Engine (2018), submitted to Simulation Modelling Practice and Theory
13. Thule, C., Palmieri, M., Gomes, C., Lausdahl, K., Macedo, H.D., Battle, N., Larsen, P.G.: Towards reuse of synchronization algorithms in co-simulation frameworks. In: Software Engineering and Formal Methods: SEFM 2019 Collocated Workshops: CoSim-CPS, ASYDE, CIFMA, and FOCLASA, Oslo, Norway, September 16–20, 2019, Revised Selected Papers. p. 50–66. Springer-Verlag, Berlin, Heidelberg (2019), https://doi.org/10.1007/978-3-030-57506-9_5