Daccosim NG: co-simulation made simpler and faster

José Évora Gómez[1] José Juan Hernández Cabrera[2] Jean-Philippe Tavella[3] Stéphane Vialle[4]
Enrique Kremers[5] Loïc Frayssinet[6]

[1]Monentia SL, Spain, jose.evora@monentia.com
[2]SIANI, Spain, josejuanhernandez@siani.es
[3]EDF Lab Paris-Saclay, France, jean-philippe.tavella@edf.fr
[4]CentraleSupélec - University Paris-Saclay & LRI, France, {Stephane.Vialle}@centralesupelec.fr
[5]EIFER, Germany, enrique.kremers@eifer.org
[6]CETHIL - BHEE, France, loic.frayssinet@insa-lyon.fr

Abstract

This paper introduces the last evolution of Daccosim co-simulation environment, with Daccosim NG developed in 2018. Main features of Daccosim NG are described: enhanced Graphic User Interface and Command-Line Interface, algorithm and mechanism of co-simulation, co-execution shell, software architecture designed for both centralised and distributed architectures, aggregation of a co-simulation graph into a Matryoshka FMU, and declarative language to design large scale co-simulation graphs. A new industrial use case in simulation of energetic systems is also introduced, and first performances of Daccosim NG on multi-core architectures are analysed.

Keywords: co-simulation tool, multithreaded execution, master algorithm, FMI standard, energy system, runtime performance

1 Introduction

The study of Smart Grids, which are intelligent energy systems enhanced by additional communication means and modern IT features, requires a complex analysis of many components considering different aspects. These aspects are amongst others, the demand, production (including renewable), stability of the power grid and flexibility assessment. This is the case for Electricité de France (EDF) and the European Institute For Energy Research (EIFER), where Smart Grids and, more in general, Multi-Energy System analysis are performed through simulations representing the power grids considering multiple aspects. To this end, EDF and EIFER are working in the development of simulation models.

For instance, there are teams working in the modelling and simulation of customers by representing how devices consume energy at their homes: fridges, stoves, washing machines, etc. The analysis of the energy demand of these devices also requires to study thermal dynamics, since many of these devices produce heat or cold. Besides of thermal dynamics, the sociotechnical behaviour of the customers must also be represented as they are the ones who operate the devices. There are also teams developing models for representing thermal gains and loses for houses, buildings, districts, etc. Other teams are dedicated to optimise the grid operation with massive renewable energy and storage units.

Some examples of these kinds of business models are ThermoSysPro, BuildSysPro, PlantSysPro, TelSysPro and EPSL. ThermoSysPro (Hefni et al., 2011) is a library devoted to the modelling and simulation of power plants and energy systems. BuildSysPro (Plessis et al., 2014) is designed to be used in several contexts including building physics research, global performance evaluation, technology development and impact assessment. PlantSysPro is devoted to industrial processes like hot water system. TelSysPro is a new Modelica library able to model the impact of telecommunication networks on complex systems from failure/repair rate of components and stochastic latency.

These teams develop their models using the tool that is the most appropriate according to their work habit or affiliation. There are many tools or programming languages that can be used for developing these models: Anylogic (Borshchev, 2013), Dymola (Elmqvist et al., 1996), Matlab (Guide, 1998), Java (Gosling et al., 2014), Python (Rossum and al., 2007), etc. So, it happens very often that teams want to collaborate by making their models interoperable with others. This is challenging since models are developed in different tools. At this level, the interoperability challenge is double: syntactic and semantic (Hernandez et al., 2016).

The syntax challenge consists in being able to technically communicate models that are developed in different tools. For instance, this problem is equivalent to two people trying to speak when they do not have a common language. The semantic problem has several axis when talking about data exchange between two models: meaning of the words, units that are used, data types, etc. The most common semantic problem in models communication is to have different words to express the refer to the same concept.

The syntactic problem is addressed in FMI (Blochwitz et al., 2011). FMI, the Functional Mock-Up Interface, is a tool-independent standard that supports both model exchange and co-simulation of dynamic models using a combination of XML-files and compiled C-code. In this way, every model that is exported following this standard can be inter-operated with other exported models (FMU - Functional Mock-Up Unit). In the case of co-simulation, an FMU contains a definition of the model expressed in a standard format using XML and some binaries depending on the platforms to which the FMU is compatible with. These binaries are dynamic libraries that can be loaded by a Master Algorithm (MA) and have a standard interface that the MA knows. In this context, FMUs are considered as slave components that are commanded by MAs. A MA is a piece of software that coordinates the execution of several FMUs (slaves). This coordination mainly regards the data exchange between the different FMU models and their scheduling (the way time is advanced).

The construction of the MA to engineer co-simulations is the main challenge this paper addresses. To this end, we present a new version of Daccosim (Distributed Architecture for Controlled CO-SIMulation) (Galtier et al., 2015; Tavella et al., 2016). This new version is called New Generation (NG). Daccosim NG is a tool oriented to facilitate the construction of co-simulations. To this end, the MA behaviour can be easily defined using a very simple interface. Through the Daccosim graphical user interface (GUI), the user can drag-and-drop the different FMUs that are desired to use and, through arrows, the data exchanges among FMUs are defined. Data exchanges are made from an FMU output with a given name to an FMU input which may have a different name, addressing in this way the semantic interoperability challenge of having different wording to refer concepts of the reality. Additionally, Daccosim NG provides mechanisms to deal with other semantic problems as data types or units mismatch.

The new version, NG, is based on the same concept as previous Daccosim versions but re-written from scratch to get an industrialized code knowing that the first Daccosim experience was achieved without professional software developers. A new dramatically simplified installation procedure, a redesigned smarter interface and better performances are the strongest points of this new version.

This paper firstly introduces some co-simulation use cases that have been addressed using Daccosim NG. Then, Dacossim NG is presented showing its capabilities and explaining how co-simulations are executed. After presenting Daccosim NG, the novelties with respect to the old versions are highlighted. The performances of the new version are then compared with Daccosim 2017 and the conclusions and roadmap are discussed.

2 Co-simulation use cases

2.1 Available use-cases in Daccosim NG

From 2018, several demonstration cases are supplied in Daccosim NG deliveries. Some are trivial examples while others are business-oriented use cases. All the referenced FMUs are license-free and have been built using the most recent version of Dymola both for 32-bit or 64-bit machines. In addition all the Modelica source models are supplied for a better understanding.

These demonstration use cases can be downloaded from the Daccosim website (Evora et al., b) regardless Daccosim NG releases. They are organised in three folders named 1-coinit-only, 2-academic, and 3-industrial.

Cases located in 1-coinit-only are co-initialization examples where only a starting point is calculated (no time integration). They illustrate how a system composed with two or more coupled FMUs can be initialised solving algebraic loops between FMUs using a Newton-Raphson algorithm. Incidentally, some cases also show how operators can be used as objects dropped in co-simulation graphs in addition to FMUs (section 3.3). One of these cases is more deeply detailed in section 3.2.1.

Cases located in 2-academic are academic examples illustrating different non-stiff and stiff cases, sometimes including internal events in FMUs. Among other interesting cases, we can emphasise on:

- A case defining a co-simulation graph with FMUs exported from ControlBuild and Papyrus coupled with Dymola FMUs (non-Modelica source models are not supplied)
- Theoretical cases illustrating the capability to define thousands of connections between two FMUs or to instantiate hundreds of times the same FMU
- Two other cases showing how a stochastic behaviour implemented in Modelica models can be useful at a system level

Lastly, 3-industrial folder is dedicated to industrial cases. At the time being, only one case is included in this folder but we intend to enrich it next with for example a distributed power flow.

The supplied use case is related to district heating and cooling energy in buildings. The system represents a district composed with 23 buildings (with only 2 adjoining walls) with inter-building long-wave radiation coupling and solar flux pre-processed per facade to account for shadings and reflections. These FMUs have been built with public components from the EDF BuildSysPro library (Plessis et al., 2014). It represents one of the four variants of a business case more deeply described in section 2.2.

2.2 Building heating and cooling power load at district scale use cases

The heating and cooling energy consumption of buildings is a critical target for current energy issues as its contribution to the overall energy consumption and related greenhouse gases emissions is dominant, while its saving potential is high (IPCC, 2014). Furthermore, district scale implementation offers advantage for the integration of renewable energy sources, particularly in buildings, and notably via Smart Grids.

Therefore, the modelling of the building heating and cooling power load at district scale is essential. However, it faces two main challenges: the computational cost, that becoming prohibitive when using detailed model, decreasing the temporal scale and increasing the spatial scale; and the lack of data at the district scale. To cope with these constraints, the level of detail of the models has to be adapted.

In order to quantify the adaptation suitability, the platform MoDEM (standing from Modular District Energy Model (Frayssinet, 2018)) has specifically been developed. This platform is able to generate building energy model at district scale, with different level of detail, automatically from geometrical data. The district scale model is made of Modelica building models[1], that are coupled (common wall and long-wave radiative heat exchanges), depending on the modelling variant, and co-simulated with Daccosim NG after being converted in FMUs.

The use cases correspond to a district of Paris, France, made of 23 buildings (with 2 adjoining walls)[2] for the following variants:

1. Model–the most detailed–considering inter-building long-wave radiation coupling and pre-processed solar fluxes computed per facade to account for shadings and reflections.
2. Model (1) but with a lower discretisation of the conductive heat problem (fewer equations).
3. Model (2) but without long-wave coupling and and specific solar fluxes (less external resources and no connection between FMUs, excepted for the adjoining buildings).
4. Model (3) but with a simplified model for the conductive heat problem (less equations).

These models were simulated for one year and a month, with a constant time step of 900 s.

The present models focus on heating and cooling but the FMI offers further opportunities to couple these models with energy system, occupant behaviour and energy network models, toward integrated district energy model.

2.3 Urban energy planning use cases

Urban planning use cases are dedicated to the simulation of the multi-energy system of one or more districts, up to a whole town or city, consisting usually in several hundreds or thousands of buildings. This use case is considered as a prospective evolution and application for Daccosim NG, and will thus not be directly analysed in this work. However, even if the time resolution of these models is rather low in comparison to the previous ones (1h-15min), it is a use case that has high requirements for scalability and thus parallelisation, as it replicates the number of buildings of the use case 2.2, "Building heating and cooling power load at district scale use cases", by a factor of 10-500. Therefore it has been identified as relevant, and the requirements and lessons learned by the prototypes being done in this direction on the Anylogic platform by EIFER are gently provided as inputs to drive and inspire the Daccosim NG development to support such applications.

[1]Using the BuildSysPro library.

[2]More information about the characteristics of the district can be found in previous reference.

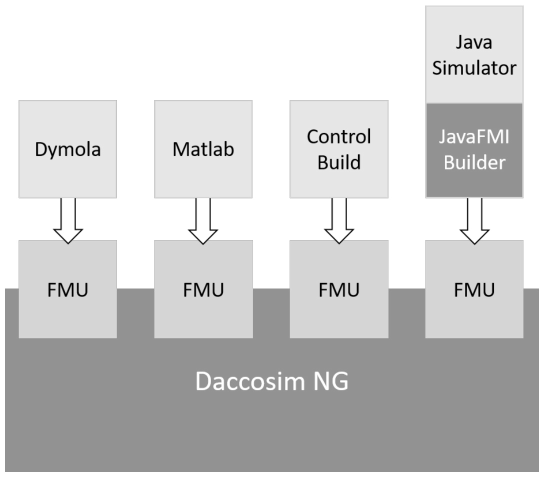

Figure 1. Daccosim interoperability example

3 Daccosim NG

Daccosim NG is an environment to develop co-simulation use cases supported by JavaFMI, a suite of tools for interoperability using the FMI standard (Evora et al., 2013). Daccosim allows the design, development and execution of co-simulation graphs, providing mechanisms to represent co-simulation graphs.

Daccosim NG is able to integrate different simulators exported as executable FMUs from various FMI-compatible tools. An exported FMU is a simulator contained in a FMU file, according to what is understood in the FMI standard. This way, any simulation developed in any programming language and deployed in any computer could be imported in Daccosim NG.

In figure 1, an example of a co-simulation integrating simulators from different sources is shown. On the upper part of the figure, it can be seen how Dymola, Matlab or ControlBuild simulators are integrated as FMUs. Same way Java or C++ codes can be exported as FMU thanks to the JavaFMI Builder tool (Evora et al., 2013).

Daccosim NG can be used through a graphical user interface (GUI) or a command-line interface (CLI). In subsequent sections, it is described how co-simulation graphs are defined, initialised and executed. Besides, some features are presented: the GUI, the CLI, FMI exposition, the matryoshka FMUs construction, the Daccosim graph declarative language. In a user's guide available with the tool distribution (Evora et al., a), more detailed information about the usage of Daccosim NG is available.

3.1 Co-simulation graph design

A co-simulation graph is composed of nodes and arrows that connect nodes. A connection defines which output variables of a source node are connected to which input variables of a target node. There are different types of nodes that can be included in a co-simulation graph:

FMU, operators, external inputs or external outputs:

- FMU: this node represents an FMU and it holds the file path, the variables used as input and output, and the initial values for variables and parameters.
- Operators: there are four operators: adder, multiplier, offset and gain. These operators allow to make calculations using outputs of other nodes providing the result in an output to be used. The adder and multiplier have two or more inputs and one output. The offset has only a fixed value and one input that are summed. The gain is defined with a fixed value that will multiply a given input. All of these nodes can work with Reals, Integers and Booleans.
- External inputs/outputs: these nodes allow to provide fixed values as input for other nodes (external input) or to store values provided by an output (external output). Both kind of nodes can have several variables. For instance, an external input will hold several outputs that can be used for other nodes.

Once nodes are defined, arrows can be established to define how variables are exchanged among them.

3.2 Co-simulation algorithm

Once the co-simulation graph is defined, the execution of the co-simulation can be done. The execution consists in the following steps: loading, co-initialisation (section 3.2.1), co-execution and exporting the results. The Daccosim engine executes each step of this method in parallel so that all cores of a machine are used, improving the performance (section 6).

In listings 1 the co-simulation algorithm is described. The procedure starts by opening the file in which the co-simulation is defined and loading the graph in memory. After opening this file and processing it, every FMU that is used is also loaded. In this way, co-simulation graph is ready for next steps: co-initialisation and co-simulation.

Listing 1. Co-simulation algorithm

```
public void execute() {
  loadGraph();
  coInit();
  export(currentTime);
  while (currentTime < stopTime) {
    currentTime += doStep();
    export(currentTime);
  }
  terminate();
}
```

After the co-initialisation process is executed (check section 3.2.1 for more information about this process), the initial values of the variables selected by the user for exportation are written into the output file for being analysed afterwards. Then the co-execution process starts. For this, the doStep method is called as many times as necessary until the stop time of the simulation is reached. The way the co-execution works is further described in section 3.2.2. After each successfully performed step, the values to be exported are once more written in the output file. Once the stop time is reached, the simulation is finished by terminating all FMUs and closing the exportation file.

3.2.1 Co-initialisation

One of the difficulties of the kind of co-simulations we are considering is the setting of consistent system-wide initial values for all the components. The Daccosim co-initialisation algorithm starts by building a global dependency directed graph for the connected variables of the FMUs. It uses the connections established by the user to find external dependencies between the outputs from source FMUs and the inputs from sink FMUs.

The key idea is that a topological sorting of the directed acyclic graph (DAG) naturally gives the order in which the variables must be initialised. Therefore, this led to study how to convert a generic directed graph into a DAG. The solution found is to build the graph of strongly connected components (SCC) corresponding to cyclic dependencies. The resulting graph in which each SCC has been contracted into a single vertex is a DAG. We use Tarjan's SCC algorithm (Tarjan, 1972) (used in many Modelica tools) to identify each SCC in the dependency graph (runs in linear time). Following the order obtained with a topological sorting on the contracted SCC graph:

1. for nodes which were not contracted, simply propagate their values
2. for nodes which were contracted (they correspond to cyclic dependencies), we solve the initialisation problem using an iterative algorithm called JNRA (Jacobian based Newton-Raphson Algorithm) inspired by traditional Newton-Raphson algorithms often used for electric load flow computation.

The example in figure 2 illustrates the co-initialisation of a system composed with two equations and two unknowns:

- equation1 model calculates x_2 from x_1 according to the equation: $2x_1^2 + 5x_2 = 42$
- equation2 model calculates x_1 from x_2 according to the equation: $x_1 - 6x_2 = 4$.

As it can be seen in figure 2, equation1.x2 depends on equation2.x1 while equation2.x2 depends on equation1.x2. In dotted lines it can be seen an algebraic loop where modifications on equation1.x1 affects equation1.x2 and modifications in equation2.x2 affects equation2.x1. Then, the co-initialisation procedure will then compute this graph to provide a consistent initial value to all variables. To do this, it will detect one SCC and, after several iterations, x1 and x2 will reach following values (x1 = 4.56, x2 = 0.09).

3.2.2 Co-execution

In Daccosim, it is possible to choose how the time is advanced when executing the co-simulation graph. There are two main categories of time steppers: constant and variable.

The constant stepper advances the simulation using a fixed step size. That is, when the simulation is stepped,

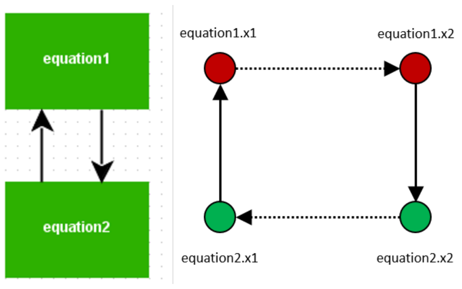

Figure 2. Calculation graph and dependency graph

the time that will be advanced is every time the same one. Constant stepper may not present a good computation/accuracy ratio: the choice of a small value for the step size results in a large number of computation steps, while a large value might fail to capture some variations in the simulated variables.

Depending on FMU capabilities, Daccosim NG also implements several variable stepping strategies. After the simulation of step i, each FMU examines its outputs and estimates how far they are from the exact value. Daccosim NG implements two algorithms which do that: one is based on the Euler's method and a second one is based on Adams Bashforth's method. Their principle is to store the values of the derivatives at consecutive communication points to infer an estimation at the next iteration. If the error is found to be tolerable, the engine will propose to perform the next step with a bigger step size. Otherwise, the last step would be cancelled and redone with a smaller step size value. The rollback is made possible since version 2.0 of FMI which introduced the notion of FMU state, allowing the serialisation of the FMU state before performing a simulation step, and the restoration of the saved state if necessary.

Suppose FMU A provides inputs for FMU B and initial step value is 10. At $t10$, it is decided that the step must be redone with a step size of 6 and it does not send its outputs to B. B, on the other hand, is satisfied with its outputs and only awaits updated inputs from A to perform its next step (from t10 to t20). When A reaches t6, it could send its outputs to B but they would not make much sense since B already advanced to t10 (and the next available outputs from A could be time stamped t12, which is not satisfactory either). To avoid this situation, all the FMUs adopt the same pace and they will all redo the cancelled step with the same new (smaller) step size. Conversely, if all the FMUs agree on a bigger step size, it will be used for the next steps.

It is also intended to implement another variable stepping method based on the concept of state quantization used in the Quantized State Systems (QSS) methods which are non-stiff QSS solvers of different orders described by Ernesto Kofman in many publications (Kofman and Junco, 2001; Kofman et al., 2001; Kofman, 2002). QSS adaptations for FMI standard are being designed in one work package of the French national project Modeliscale (2018-2020) leaded by Dassault Systèmes and whose

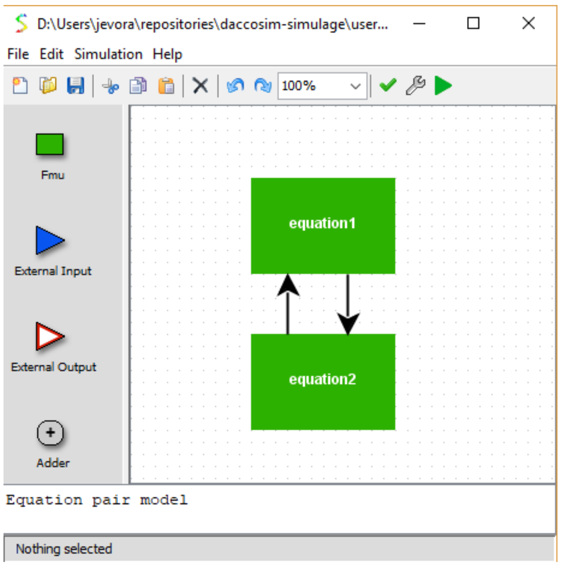

Figure 3. Daccosim NG GUI

goal is to provide the ability to model and simulate with Modelica and FMI the behavior of very large energy systems.

3.3 Editor

This module provides a GUI to facilitate the design, development and execution of co-simulation graphs. The editor can be downloaded from (Evora et al., b). The editor is distributed in three different formats: exe files for running in Windows (32 & 64 bits exe files) and a jar file for execution in any operating system. All of these version require to have a JVM installed in the system. In any of these three formats, the editor can be launched by just double-clicking it (it does not require any installation or configuration).

In the GUI (figure 3), aside from the menu and the toolbar with the options to deal with the co-simulation graph, the palette and the canvas are the main components supporting the definition of co-simulation graphs (Evora et al., a). The palette contains all the possible nodes that can be set in the co-simulation graph (note that for space constraints, not all blocks are visible in the figure). These nodes can be dragged and dropped into the canvas.

Then, by dragging out from the centre of a node (source node), an arrow can be created when dropping the mouse in the target node. The variables that are to be exchanged in that arrow can be configured by double-clicking it or in the contextual menu: properties option. Each kind of node has also its own configuration which can be accessed in the same way: double-click on the node. In the case of an FMU node, the label of the node and the initial values of the FMU variables can be configured. For external inputs and external outputs, the variables to be connected to other nodes can be defined and their initial value set. In the case of the operators, the data type to work with (either Real, Integer or Boolean) and in the case of the adder and multiplier, the amount of inputs to receive can be chosen. In the case of the offset and the gain, the fixed value that will be added or multiplied to the input can be set.

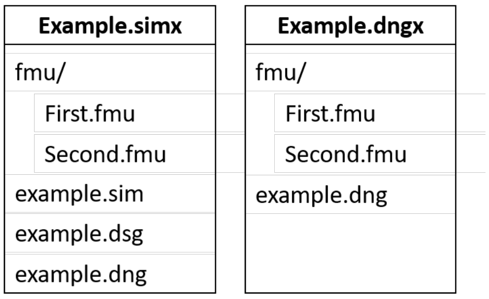

Figure 4. simx and dngx files

Below the canvas and the palette, there is a blank empty box in which the user can write a message to describe what purpose is the model and any other extra information. And below this box, there is a bar state in which some information is contextually displayed to interactions made by the user.

Up in the toolbar, commands are defined in menus to create a new co-simulation graph, to open it (see section 4.1 to know more about file formats) or to save it. It is also possible to cut, copy or paste parts of the graph and to undo or redo some of the actions made. Then, once the graph is defined, it can be validated, configured (start and stop times, variables to export, etc) and executed.

For more information about how to use the editor, please read the user's guide (Evora et al., a).

4 Other Features

4.1 Shell

This module wraps the core and provides a command line interface (CLI) to run co-simulation graphs (Evora et al., a). The shell can be downloaded from (Evora et al., b). This utility allows users to develop script files to run co-simulation graphs in a batch mode. The main argument to provide to this CLI is the path to the file in which the co-simulation graph is stored. With Daccosim NG there are two main file formats:

- simx: this is an archive file (zip) containing a folder named fmu with the fmu files used in the co-simulation graph as well as the sim, dng and dsg files (figure 4). The sim, dng and dsg files contain representations of the co-simulation graph in different formats. sim file contains the graph including the visualisation information to be presented in the Editor. dng contains the graph in the declarative language (section 4.4). dsg is a serialisation of the graph in json format. This is the one that is effectively used as a graph representation for starting the co-simulation execution.
- dngx: this archive has the same structure and content than the simx but sim and dsg files are not present (figure 4). It allows to create a runnable file in which the graph is defined using the declarative language (section 4.4). The idea of this format is to allow other tools or users with a text editor to create a co-simulation graph compatible with Daccosim NG by describing it in a dng file. This is especially interesting to develop large-scale co-simulation graphs.

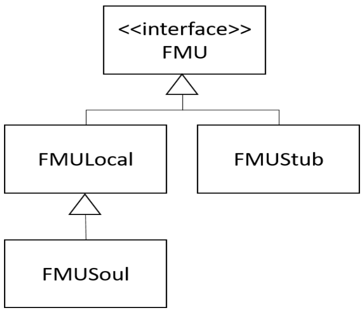

Figure 5. The FMU interface is used by the execution engine. This way the engine works independently from where FMUs are being executed

This CLI has also the possibility of parameterizing some of the features of the execution. The idea is to make simpler the GUI (editor) by avoiding this kind of parameterization and let for advanced users to play with them using the CLI. For instance, one of the parameters that can be changed in CLI is to run the co-simulation in singlethread or multithread. In the editor, this option is not available and all co-simulations are run in multithread by default.

4.2 Designed for distributed executions

FMU nodes are normally performing costly processes each time they are called in the doStep method (Blochwitz et al., 2011). For this reason, the execution engine is also prepared to be run in a distributed environment allowing the execution of large-scale co-simulation scenarios. This is made thanks to the use of abstraction mechanisms so that the execution engine does not need to be aware of where each FMU is being physically executed. To this end, every time the engine interacts with an FMU, it uses an abstracted interface. Based on this interface, there are three implementations: FMULocal, FMUStub and FMUSoul. First one uses the FMU files from the filesystem (normal case in a single machine). Second one uses a connection to a Java Message Service (JMS) to interact with an FMU that is being remotely executed. Third, and last one, is the representation in the remote computer of the FMU. This receives the queries from the FMUStub and acts accordingly (figure 5).

In figure 6, the communication between different machines running a distributed simulation is exemplified. In this example, there are three machines. In the first one, an instance of Daccosim core is responsible for coordinating a distributed execution. The execution engine of this instance uses the FMU interface to communicate with the three FMUs to be coordinated. Two of them are being executed remotely and one locally. However, as the engine only depends on the interface, these details of where they are being executed are not important for its execution. Whenever a command is asked by the engine, the FMUStub will communicate to the FMUSoul to perform

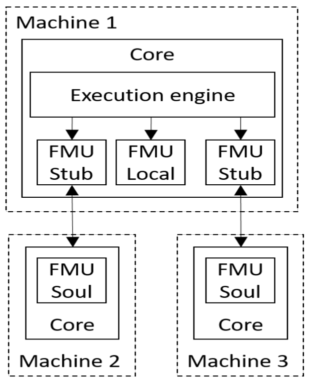

Figure 6. The communication of the execution engine with the remote fmu (soul) is made through the stub

the command providing the answer back to the engine. Note that, despite it is not represented, communications are made through a JMS. The use of JMS gives flexibility to design different distribution architectures to support large scale co-simulations.

4.3 Matryoshka

This feature(Galtier et al., 2017) allows to wrap a co-simulation graph into an FMU for being used either in other co-simulation environments like Dymola or in Daccosim NG itself (Evora et al., a). The co-simulation graph that is stored inside the FMU is seen as a single FMU when opened by other tools. Every time a master algorithm uses this FMU for any purpose, the Matryoshka FMU will dispatch the command to the corresponding inner FMU or FMUs.

For instance, if the master commands a simulation advancement through a doStep to the Matryoshka FMU, this FMU will perform the doStep for all the FMUs contained and will exchange the values between the FMUs as described in the co-simulation graph. At the same time, it is possible to embed one Matryoshka FMU inside another co-simulation graph and export this graph into a Matryoshka FMU having in this way several levels of FMUs embedded. Exported FMUs will be beneficial (Galtier et al., 2017):

- FMU can be imported into any FMI compliant simulation tool also able to handle non-FMI components with which Daccosim NG is not able to directly interact.
- Taking advantage of Daccosim NG efficient, multi-threaded, step-size control solution helps simulating faster larger models within traditional monothreaded simulation tools.
- Initialization of complex graphs is taken care of within the Matryoshka thanks to Daccosim NG generalized co-initialization algorithm.
- A complex simulation graph can be reused directly without having to re-write anything and with no risk of disclosing industrial and intellectual property.
- The co-simulation process can be finely tuned: when typically a solver only uses one accuracy objective for the whole model, Daccosim NG allows the user to define different tolerance values for every output and internal variable of each FMU.

Besides, new features have been developed:

- Added information in the modelDescription file about dependencies of external outputs to external inputs.
- Continuous inputs extrapolation and output derivatives provision.

These improvements have been accompanied by a significant FMU size reduction of the order of 3MB. They will be completed next year with two new features: improvement in the performance when loading multiple instances and the capability to make rollbacks.

4.4 Declarative language

The declarative language implemented in Daccosim NG allows the user to define a co-simulation graph on a text editor or to automatically generate it through a program (Evora et al., a). To do so, a domain-specific language has been designed to simply define a co-simulation graph. Enjoying a feedback from user experiences, this language has been dramatically simplified regarding the previous version named DSL in Daccosim 2017. As it can be seen in listing 2, this language is very simple and can be easily understood. Its purpose is to create very wide graphs that cannot be modelled in the GUI, in which there are hundreds of interconnected nodes exchanging thousands of variables. This feature allows pre-processing tools to develop compatible models to be executed in Daccosim NG. Note a textual form of a co-simulation is automatically generated from the GUI once a valid graph is defined. Conversely, a valid textual form of a co-simulation can be opened in the GUI and the corresponding graph is automatically drawn.

Listing 2. Declarative language example

```
FMU equation1 "fmu/equation1win3264.fmu"
Output equation1 x2 Real
Input equation1 x1 Real
FMU equation2 "fmu/equation2win3264.fmu"
Output equation2 x1 Real
Input equation2 x2 Real
Connection equation1.x2 equation2.x2
Connection equation2.x1 equation1.x1
CoInit 100 1.0E-5
ConstantStepper 1.0
Simulation 0.0 10.0
```

Listing 2 expresses the co-simulation graph defined in figure 2 using this language. There are two FMUs declarations followed by the label and the path to the file. Then the outputs and inputs to be connected are described for each of the FMUs and the connections defined. Finally, the co-init, stepper method (constant step with step size 1.0) and simulation start and stop times are defined.

5 Why NG?

In this last year and a half, we have rebuilt Daccosim from the scratch. To this end, we have gotten rid of some strong dependencies that made difficult the evolution and use of the software. Prior version of Daccosim (2017) required the installation of the Eclipse IDE (Eclipse, 2007) in a specific version with specific plugins. Then, the source code of the project had to be imported into it and compiled to be run. This was a tedious process that made harder to the users their initial steps in Daccosim. In the new version, just by downloading it and making a double click, the user can start working in the design of a co-simulation graph using a very comfortable GUI.

Since the GUI is detached from Eclipse IDE, the interface is not contaminated by the style in which Eclipse displays buttons, views, etc. This GUI is tailor made to focus on the design of the co-simulation graph and its execution. This way, the result is a very clean interface with two main components, palette and canvas, in which the graph is designed. The projects view has also been deleted as Daccosim NG can be opened as many times as necessary holding a project on each instance.

The performance has also been improved in many aspects. Previous Daccosim versions used the co-simulation graph designed by the user to generate tailor made Java code for executing the graph. For this reason, co-simulations made by the user were conceived as ”projects”, as they had the files containing the graph definition, the fmus and the generated code for execution. In this new version, co-simulation graphs are read an interpreted so that no Java code is generated to execute a specific graph. This makes the process much faster and lighter, specially for wide co-simulations. This way, all the information that concerns a co-simulation is stored in a single simx file (section 4.1). The performance in runtime is also compared in the following section 6.

From the point of view of the development, the most important achievements are the removal of strong dependencies (Eclipse and its plugins) and the code maintainability which will make easier the correction of bugs and the evolution of the software.

Finally, to make easier the experience of the user, a daccosim-windows-installer has been built for a complete and simple installation on windows 32-bit or 64-bit machines. This is available at (Evora et al., b).

6 Performance on parallel machines

6.1 Comparing previous and new Daccosim

In order to exhibit the performances of Daccosim NG, we run the 4 variants of the case building energy system with 23 FMUs described in section 2.2. The runs have been done on the same 4-core Windows machine both with Daccosim 2017 (the previous version of Daccosim) and with the new Daccosim NG (also called Daccosim 2018).

In order to get reliable results (summarized in table 1), each variant has been done 3 times with strictly the same co-simulation conditions (start time, stop time, stepping parameters,...). Based on the average of gotten time durations, a speedup has been calculated ($Speedup = T_{2017}/T_{2018}$) showing the performance improvement of Daccosim NG (2018).

Variant	Model 1	Model 2	Model 3	Model 4
Dac. 2017	6150 s	1666 s	1660 s	2075 s
Dac. 2018	5217 s	1574 s	1562 s	1515 s
Speedup	1.18	1.06	1.06	1.37

Table 1. Performances of Daccosim 2018 *vs* Dacossim 2017

6.2 Benchmark on dual-processor Linux machines

In order to evaluate the performances of Daccosim NG on a parallel multi-core machine, we needed a test application with a large number of FMUs to spread on computing cores, and with a significant total amount of computations and disk IO. But most of our business use cases involve FMUs handling resource files (e.g. temperature time series), and unhappily when exported from Dymola Linux these FMUs are not working correctly unlike with export from Dymola Windows. We go on investigations to identify the issue either at the Dymola side or as a side effect in Daccosim NG. For this reason, we fell back to the demonstration case *multiFMU* supplied beside the Daccosim NG distribution which was originally composed of 1000 instances of the well-known *stairBouncingBall* model (Kofman, 2004). In order to mix logical instances and physical FMUs, we have duplicated the original FMU 10 times to get 10 different FMUs and then we have instantiates each of them 100 times. However, all these FMUs model independent balls, and do not consume inter-FMU communication times. About IO, we have defined two variants, the first one without any FMU output saved on disk and the second one with 2 outputs per FMU written after each step integration.

This test case has been run on a dual 10-core Intel Xeon Silver 4114 at 2.2 GHz (*Skylake* architecture), with 96 GBytes of RAM. This machine is part of a PC cluster of CentraleSupelec, managed with the OAR[3] environnement. OAR allows to allocate an entire PC or only a required number of its cores, and runs all threads of an application only on the allocated cores. We used this mechanism to test our application from 1 up to 20 physical cores, and then up to 40 logical cores on our dual 10-core Xeon machine.

When running our test case for 5000 steps on one core of our test machine, the variant saving 2 outputs per FMU generates an output file of 110 *MBytes* and elapses on 160 *s* We have chosen this configuration with easy to measure execution times (not too long but significant times).

Figure 7 left shows the co-simulation execution times as a function of the number of allocated cores, in logarith-

[3] s://oar.imag.fr/

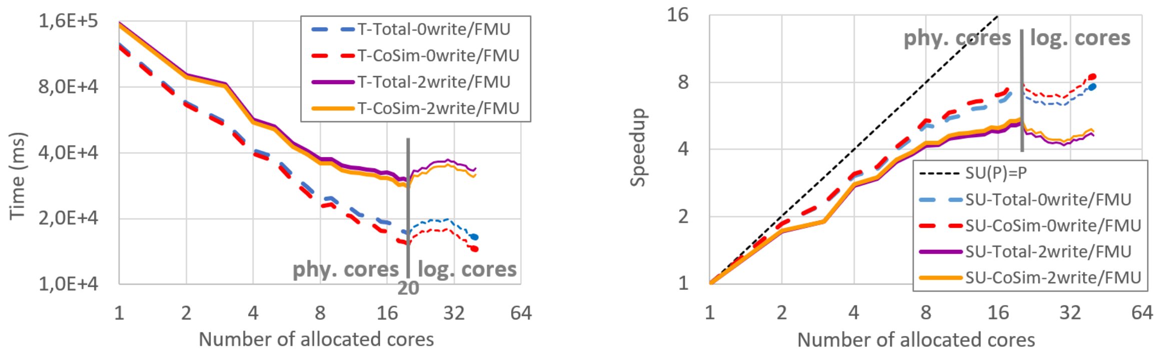

Figure 7. Execution times and speedup of *multiFMU* Daccosim NG benchmark on *dual 10-core Xeon Silver 4114* machine

mic scale. A straight line with a -1 slope would mean a perfect decrease of the execution time. Full lines illustrate performances of the benchmark with 2 FMU output writing per time step, while dashed lines are related to co-simulation runs without any IO (no FMU output was saved on disk). We can observe a very regular and very good decrease of co-simulation and total execution times from 1 up to 10 physical cores, and a little bit less good decrease from 10 up to 20 physical cores when using the second CPU of the machine. As expected, execution time is lower and exhibit better decrease when no FMU output are written on disk (no IO). When writing all FMU outputs on disk at each time step, execution time is higher but still exhibits a significant and almost regular decrease.

Beyond the 20 physical cores of our machine, the threads are distributed also on the *logical* cores. As each physical core hosts two logical cores, when allocating $20+n$ cores (with OAR), n physical cores host two threads and $20-n$ host only one thread. Beyond 20 cores this load unbalance leads to an execution time increase, as illustrated on figure 7 left. However, when allocating 40 cores (all virtual ones) load balancing is achieved again and performances appear a little bit better than on 20 physical cores when no FMU output is written on disk (dashed lines). At the opposite, when writing 2 outputs per FMU on disk (full lines) it appears better to use only the 20 physical cores. These IO remain sequential and partially overlapped with the computations (depending on the OS), but disturb parallel computations.

A single-threaded version of Daccosim NG (running on one core), has exhibited execution times very close to our multithreaded version run on one core. Then we can define the speedup achieved by Daccosim NG running on several cores: $SU(p) = T_{single}/T_{multi}(p) \approx T_{multi}(1)/T_{multi}(p)$. Figure 7 right shows this speedup, and we get:

$$\begin{aligned} SU^{max}_{0write/FMU} &= SU_{0write/FMU}(40) = 7.7 \\ SU^{max}_{2write/FMU} &= SU_{2write/FMU}(20) = 5.2 \end{aligned}$$

Considering only the experiments with FMU output saving (more realistic use case), experiments have shown the execution time of the multithreaded implementation of Daccosim NG has *scaled* on our benchmark. An almost regular decrease of the execution time has been measured up to all physical cores of our dual 10-core Xeon machine. Moreover a significant speedup close to 5.2 has been achieved compared to a sequential execution. The current multithreaded implementation of Daccosim NG appears ready to be the kernel of a distributed version on PC clusters and clouds.

7 Conclusion and roadmap

In 2018, Daccosim NG is more robust, faster and simpler to use than the previous version that was simply a proof of concept for EDF to make sure that co-simulation is helpful for the simulation of wide energetic systems.

To further improve it, the Daccosim NG team intends to implement before the end of 2018 a new major version including some Matryoshka evolutions (section 4.3) and QSS-inspired variable stepping implementation (section 3.2.2). In addition, as EDF is participating to the Modelica Association Project FMI, Daccosim NG is candidate to implement the new hybrid co-simulation feature under discussion in the WG "clock hybrid co-simulation" in order to accurately detect internal FMU events from a proposal pushed by EDF in 2017.

We will also think about an implementation of the System Structure and Parameterization Standard (SSP)(Köhler et al., 2016) as a future standard way to export/import co-simulation graphs in Daccosim NG. Additionally, and as mentioned in section 2.3, urban energy planning requires large scale simulations of hundreds of buildings, which can deliver valuable simulation results taken as decision aid for large infrastructure investments by municipalities. The following challenges arise in this context:

1. Multiple layers: different energy vectors addressed (heating, cooling, electricity, gas).
2. Bottom up simulation: large number of buildings that on its own have an individual behaviour.
3. Connection through networks: energy flows are distributed via networks which have to be included and are a model in itself to couple other models.
4. Data uncertainty and unavailability: in early planning stages, many parameters have not yet been fixed, and furthermore, projections over several decades allow for important assumptions in the development of environmental parameters (future evo-

lution of energy or fuel prices, etc).

The representation of such a complex system requires and efficient coupling of different models, to avoid constructing unmanageable complicated model couplings. Daccosim NG shows excellent prerequisites to support these kind of simulations, especially on points 1 & 2. A use case in which Dymola and Anylogic energy system models are co-simulated, is thus envisaged to proof the feasibility of Daccosim NG towards requirement 3 & 4.

Finally, to go further than previous experiments in 2017 (Vialle et al., 2017), we also intend to co-simulate again very wide complex systems to illustrate the new constraints EDF has to cope with in the context of the energy transition and the renewal of the energy market landscape. This will be done as soon as the problem we have encountered with Dymola FMUs exported from Linux is solved.

Acknowledgement. This work was partly supported by the Paris-Saclay region through the FUI research project “Modeliscale”.

References

T. Blochwitz, M. Otter, M. Arnold, C. Bausch, H. Elmqvist, A. Junghanns, J. Mauß, M. Monteiro, T. Neidhold, D. Neumerkel, and al. The functional mockup interface for tool independent exchange of simulation models. In *Proceedings of the 8th International Modelica Conference; March 20th-22nd; Technical Univeristy; Dresden; Germany*, 2011.

A. Borshchev. *The big book of simulation modeling: multimethod modeling with AnyLogic 6*. AnyLogic North America Chicago, 2013.

IDE Eclipse. Eclipse foundation, 2007.

H. Elmqvist, D. Brück, and M. Otter. Dymola-user’s manual. *Dynasim AB, Research Park Ideon, Lund, Sweden*, 1996.

J. Evora, J-Ph. Tavella, JJ. Hernandez, and S. Vialle. *Daccosim NG User’s Guide*. EDF, Monentia, CentraleSupelec, a.

J. Evora, J-Ph. Tavella, JJ. Hernandez, S. Vialle, and E. Kremers. Daccosim ng web page, b. URL `https://bitbucket.org/simulage/daccosim`.

J. Evora, JJ. Hernandez, and O. Roncal. Javafmi. *URL https://bitbucket. org/siani/javafmi*, 2013.

L. Frayssinet. *Adapting building heating and cooling power need models at the district scale*. PhD thesis, INSA de Lyon, 2018.

V. Galtier, S. Vialle, C. Dad, J-Ph. Tavella, J-Ph. Lam-Yee-Mui, and G. Plessis. Fmi-based distributed multi-simulation with daccosim. In *Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*. Society for Computer Simulation International, 2015.

V. Galtier, M. Ianotto, M. Caujolle, R. Corniglion, J-Ph. Tavella, J.E. Gómez, JJ. Hernandez, V. Reinbold, and E. Kremers. Experimenting with matryoshka co-simulation: Building parallel and hierarchical fmus. In *12th International Modelica Conference*, 2017.

J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley. The java language specification, java se 8 edition (java series), 2014.

MATLAB User’s Guide. The mathworks. *Inc., Natick, MA*, 5, 1998.

B. El Hefni, D. Bouskela, and G. Lebreton. Dynamic modelling of a combined cycle power plant with thermosyspro. In *Proceedings of the 8th International Modelica Conference; March 20th-22nd; Technical University; Dresden; Germany*, 2011.

JJ. Hernandez, J. Evora, and J-Ph. Tavella. Semantic interoperability in co-simulation: use cases and requirements. *European Simulation and Modelling Conference 2016 at Las Palmas de Gran Canaria, Spain*, 2016.

IPCC. *Climate change 2014: mitigation of climate change: Working Group III contribution to the Fifth Assessment Report of the Intergovernmental Panel on Climate Change*. Cambridge University Press, 2014.

E. Kofman. A second-order approximation for devs simulation of continuous systems. *Simulation*, 78(2), 2002.

E. Kofman. Discrete event simulation of hybrid systems. *SIAM J. Sci. Comput.*, 25(5), May 2004. ISSN 1064-8275.

E. Kofman and S. Junco. Quantized-state systems: a devs approach for continuous system simulation. *Transactions of The Society for Modeling and Simulation International*, 18 (3), 2001.

E. Kofman, J. S. Lee, and B. P. Zeigler. Devs representation of differential equation systems. review of recent advances. *Proceedings of ESS’01*, 2001.

Jochen Köhler, Hans-Martin Heinkel, Pierre Mai, Jürgen Krasser, Markus Deppe, and Mikio Nagasawa. Modelica-association-project “system structure and parameterization”– early insights. In *The First Japanese Modelica Conferences, May 23-24, Tokyo, Japan*, number 124, pages 35–42, 2016.

G. Plessis, A. Kaemmerlen, and A. Lindsay. Buildsyspro: a modelica library for modelling buildings and energy systems. In *Proceedings of the 10 th International Modelica Conference; March 10-12; 2014; Lund; Sweden*, 2014.

G. Van Rossum and al. Python programming language. In *USENIX Annual Technical Conference*, volume 41, 2007.

R. Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2), 1972.

J-Ph. Tavella, M. Caujolle, S. Vialle, C. Dad, Ch. Tan, G. Plessis, M. Schumann, A. Cuccuru, and S. Revol. Toward an accurate and fast hybrid multi-simulation with the fmi-cs standard. In *Emerging Technologies and Factory Automation (ETFA), 2016 IEEE 21st International Conference on*. IEEE, 2016.

S. Vialle, J-Ph. Tavella, C. Dad, R. Corniglion, M. Caujolle, and V. Reinbold. Scaling FMI-CS Based Multi-Simulation Beyond Thousand FMUs on Infiniband Cluster. In Modelica Association, editor, *12th International Modelica Conference 2017*, Czech Republic, May 2017.