Experimenting with Matryoshka Co-Simulation: Building Parallel and Hierarchical FMUs

Virginie Galtier[1] Michel Ianotto[1] Mathieu Caujolle[2] Rémi Corniglion[2] Jean-Philippe Tavella[2]
José Évora Gómez[3] José Juan Hernández Cabrera[3] Vincent Reinbold[4] Enrique Kremers[5]

[1]CentraleSupélec, France, `{first.last}@centralesupelec.fr`
[2]EDF R&D, France, `first.last@edf.fr`
[3]SIANI, Spain, `jose.evora@siani.es, josejuanhernandez@siani.es`
[4]University of Leuven, Belgium, `vincent.reinbold@kuleuven.be`
[5]EIFER, Germany, `enrique.kremers@eifer.uni-karlsruhe.de`

Abstract

The development of complex multi-domain and multi-physic systems, such as Smart Electric Grids, have given rise to new challenges in the simulation domain. These challenges concern the capability to couple multiple domain-specific simulators, and the FMI standard is an answer to this. But they also concern the scalability and the accuracy of the simulation within an heterogenous system. We propose and implement here the concept of a Matryoshka FMU, i.e. a first of its kind FMU compliant with the version 2.0 of the FMI standard. It encapsulates DACCOSIM – our distributed and parallel master architecture – and the FMUs it controls. The Matryoshka automatically adapts its internal time steps to ensure the required accuracy while it is controlled by an external FMU-compliant simulator. We present the JavaFMI tools and the DACCOSIM middleware used in the automatic building process of such Matryoshka FMUs. This approach is then applied on a real-life Distributed Energy System scenario. Regarding the Modelica system simulated in Dymola, improvements up to 250% in terms of computational performance are achieved while preserving the simulation accuracy and enhancing its integration capability.

Keywords: co-simulation tool, multi-threaded execution, master algorithm, FMU, FMI standard

1 Introduction

Complex systems can be characterized by a great number of heterogeneous entities in interaction. The Smart Grids provide a typical example: over a large territory a multitude of devices produce, transport, store and consume electricity, while some are being monitored and controlled in order to best adjust the dynamic configuration of the electric network to the current and forecasted weather conditions and client needs. Co-simulation is essential to design and study such complex systems.

In this context, the FMI (Functional Mockup Interface) standard (Blochwitz and Otter, 2011) allows users to share and combine their models across simulation tools by wrapping them with a native solver in a package, called an *FMU for Co-Simulation*, that is composed of an XML model description and a compiled C code. But the orchestration of the execution of the multiple FMUs forming the co-simulation of a complex system is up to the user. DACCOSIM, as an FMU-based co-simulation platform able to define and simulate complex calculation graphs, proposes an answer to this matter.

Furthermore, solvers usually used to simulate multi-physics systems are single-threaded. They may thus encounter scaling problems when simulating larger systems. This is the same for those included in FMUs. DACCOSIM provides a master code orchestrating the execution of FMUs in parallel, synchronizing their data exchanges and adjusting the internal step size to ensure accuracy.

Our objective is to get the best of both worlds by wrapping a DACCOSIM co-simulation in an FMU. We refer to this englobing FMU as a "Matryoshka" FMU.

This article is organized as follow: Section 2 provides a quick overview of DACCOSIM features and inner architecture. Section 3 lists the benefits of encapsulating DACCOSIM within an FMU. Section 4 presents JavaFMI, a tool which greatly facilitates the construction of FMUs from Java code. Section 5 explains how a Matryoshka FMU is built with and by DACCOSIM. A real-life Distributed Energy System is then considered and the results obtained in terms of accuracy and computation efficiency are presented in Section 6. Finally Section 7 points out a few directions we would like to explore in the future.

2 DACCOSIM, a Powerful FMI for Co-Simulation Platform

DACCOSIM (Galtier et al., 2015) is a Java co-simulation middleware able to define and simulate complex calculation graphs consisting of multiple FMUs compliant with the FMI 2.0 standard for Co-Simulation. It relies on JavaFMI (see Section 4) and is available[1] under AGPL for both Windows and Linux operating systems, whether 32-bit or 64-bit.

[1]`https://daccosim.foundry.supelec.fr`

It consists of two complementary parts:

- A **user-friendly Graphical User Interface** (GUI) that facilitates the definition of multi-domain studies (Figure 1). It enables to easily design the calculation graph of the simulation case, i.e. the FMUs involved and the variables exchanged in-between. It also allows the user to set the resources used for the simulation case (local multi-threaded machine or HPC cluster) as well as its setup (simulation duration, co-initialization method, time step control strategy, tolerance allowed to internal solver and variables...). Results can be displayed *a posteriori* or in real-time during the simulation. In addition, a Domain Specific Language allows the user to write scripts to define, configure and run parametric studies on large co-simulation cases involving hundreds of FMUs and thousands of variable exchanges.

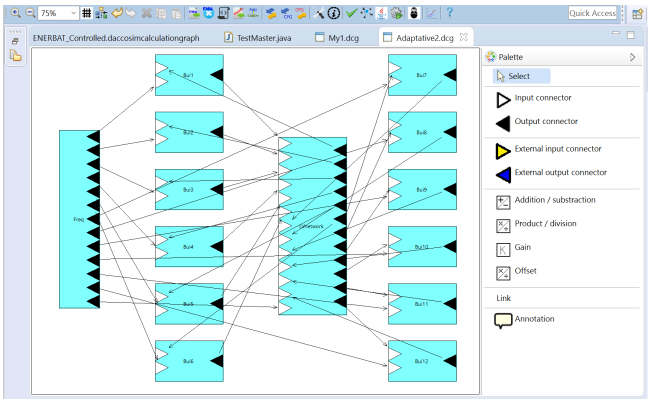


Figure 1. Screenshot of DACCOSIM GUI for a system of 14 FMUs with 110 variables exchanged

- A **parallel and distributed execution architecture** which manages the initialization and the execution of the involved FMUs. To maximize performance and scalability, DACCOSIM runs the FMUs involved in the co-simulation in parallel, using multiple threads on a node, and using multiple nodes when a cluster is available. Each FMU is executed by a wrapper directly connected to other wrappers to import and export variable values at each communication step. To provide the best trade-off between precision and computational speed, DACCOSIM integrates fixed and adaptive time step control strategies to dynamically adjust the simulation step size of all FMUs to the estimated error. In order to perform this coordinated step-size adjustment, DACCOSIM relies on a hierarchy of "masters", one on each computation node, controlling the set of FMU wrappers executing on this node. This architecture (Figure 2) is used during both co-initialization and co-simulation stages.

The transition from the calculation graph designed with DACCOSIM GUI to its execution with DACCOSIM calculation engine relies on Acceleo[2]: the graph is translated

[2] https://www.eclipse.org/acceleo/

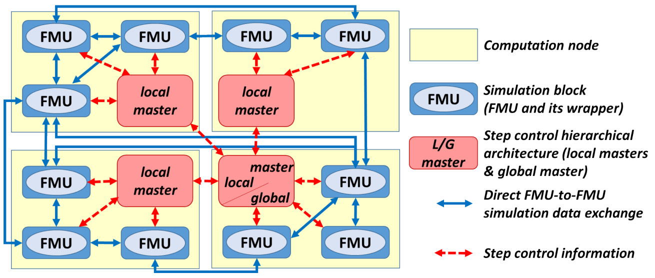

FMU
local master
master
local
global
L/G master
Computation node
Simulation block (FMU and its wrapper)
Step control hierarchical architecture (local masters & global master)
Direct FMU-to-FMU simulation data exchange
Step control information

Figure 2. DACCOSIM distributed architecture

into one or more DACCOSIM masters depending on the resources considered. These masters launch the simulation and run concurrently till the end of the simulation case.

If it is above all a robust and scalable co-simulation middleware able to simulate large and complex use cases, **DACCOSIM is also an experimental playground for the FMI standard** where innovative features are tested, such as the ahead implementation of proposed FMI primitives (Tavella et al., 2016), or the Matryoshka FMU approach presented in this paper.

3 The Benefits of Encapsulating DACCOSIM within an FMU

DACCOSIM itself is a powerful FMI for co-simulation middleware able to perform fully parallel and distributed co-initialisation and co-simulation tasks. But as a standalone tool, its scope remains limited:

- Only FMUs compliant with the FMI 2.0 for CS standard are supported. Consequently simulators such as NS-2 (a communication networks simulator), or HLA federates with no FMI interface cannot be included into its co-simulation graph.
- It cannot be integrated within domain-specific tools able to import FMUs, tools which become more and more widespread nowadays.

Designing a specific control API for DACCOSIM would help to meet these needs, but encapsulating it all into a Matryoshka FMU fulfills even more of them:

- Such an FMU can be imported into any FMI compliant simulation tool or platform such as Dymola or MECSYCO (Vaubourg et al., 2015). This opens new perspectives since some of these tools might as well handle non-FMI components with which DACCOSIM is not able to directly interact.
- Taking advantage of DACCOSIM efficient, multi-threaded, step-size control solution helps simulating faster larger models within traditional mono-threaded simulation tools. It makes particular sense for domains where few parallel solvers are available.
- Initialization of complex graphs is taken care of within the Matryoshka thanks to DACCOSIM 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 a solver typically uses only one accuracy objective for the whole model, DACCOSIM allows the user to define different tolerance values for every output and internal variable of each FMU.

4 JavaFMI Tools to Generate and Execute FMUs

JavaFMI is a software project devoted to provide a toolbox that allows to import and export Functional Mock-up Units (FMU) to/from Java in conformance with the FMI-CS 1.0 and 2.0 standards. This project is developed by SIANI[3] university institute and its license is LGPL. Main contributors of this project are EDF Lab, EIFER, and CentraleSupélec. This project is composed of two main tools: a wrapper and a builder.

4.1 FMI Wrapper

The **FMI wrapper** allows to import FMUs into a Java application supporting the creation of Master Algorithms (Evora et al.). It provides **two types of interface**: simulation (simplified interface) and access (full interface).

The **simulation class** (*FmiSimulation*) provides a very simplified access to the FMU. This way, the user of the wrapper can load FMUs without having a deep knowledge of the FMI standard. Its methods are init, doStep, terminate, read and write variable, getSimulationTime, isTerminated and reset.

The **access class** allows invoking all available methods of the standard depending on the version that is being used. This way, the simulation class can be wrapped by the access class allowing for an advanced usage. Methods like get, set and free state can be invoked among others. Basically, all the primitives specified in the FMI-CS standard can be found as methods in this class.

4.2 FMI Builder

The **FMI builder** allows to **create an FMU based on a Java application** or any program that can be controlled by a simple Java code. That is, any Java simulation can be exported to an FMU. This tool provides an automated solution to create an FMU covering the development of the dynamic libraries, the generation of a model description file and the packaging of the needed resources.

The builder provides a framework to convert a Java simulation into an FMU. It is required to extend the *FmiSimulation* class where, at least, the following methods should be implemented:

- **define.** It returns a model that contains the information to be rendered in the modelDescription.xml
- **init.** It is called in the instantiation process of the FMU. It should register all input and output variables with their corresponding getter and/or setter methods so that the framework can later get and set the FMU variables during the initialization and simulation stages.
- **doStep.** It advances the simulation according to the given step size.
- **reset.** It resets the simulation to its initial state.
- **terminate.** If needed, it should be filled with a termination code.

[3]`http://www.siani.es`

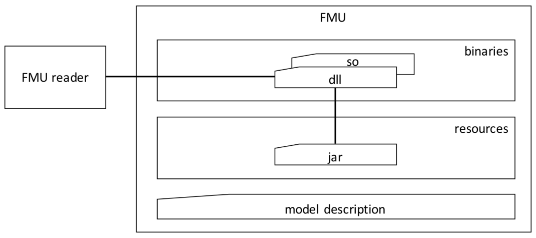


Figure 3. Communication between the JavaFMI wrapper and the FMU JAR through the libraries (dll, so)

Once these methods are implemented, the *FmiSimulation* class is packaged into a JAR (Java ARchive) file and processed by the builder so that an FMU is created. The builder creates an FMU file containing:

- Dynamic libraries (dll and so) in the binaries folder.
- Model description.
- JAR file tuned to the model in the resources folder.
- Additional FMU resources in the resources folder if any are defined by the user.

The resulting FMU is compliant with the version 2.0 of the FMI standard which makes it applicable within any FMI compliant tool. Basic primitives like init, doStep, terminate, etc. are available as well as advanced ones like get, set and free state. For these advanced methods, the FMI builder has a default implementation that can be overridden in case a custom implementation is needed.

At runtime, the FMU dynamic libraries are programmed so that an instance of a Java Virtual Machine (JVM) is created in order to load the FMU JAR file. Once this happens, all functions invoked by the user of the library are directly bridged to the Java application by using pipes. Associated data flows are explained in Figure 3.

When using JavaFMI wrapper to load the FMU, this data flow can be shortened: if the JavaFMI wrapper detects that the FMU has been built with the JavaFMI builder, it takes the JAR located in the resources folder, loads it in the JVM in which the wrapper is, and communicates directly with the FMU methods through Java (Figure 4). This yields a significant improvement in the communication speed.

The JavaFMI project also contributes to make the FMI standard evolve. New co-simulation concepts (Tavella et al., 2016) are being trialed and validated by implementing newly defined primitives linking compliant FMU-generating tools to master algorithms.

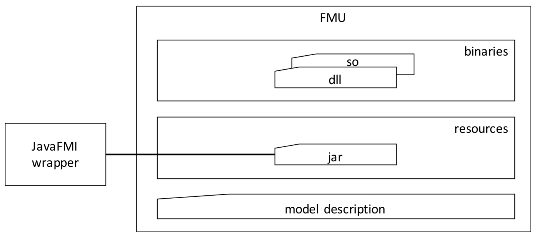


Figure 4. Direct communication between the JavaFMI wrapper and the FMU JAR created by the builder

5 Matryoshka FMU Building Process

In this section we first describe the steps taken by a DACCOSIM user to build a Matryoshka FMU for his co-simulation test case. Next we expose the behind the scene mechanisms, i.e. how DACCOSIM 2017 was augmented to support the construction of a Matryoshka FMU and which operations it performs during the building process. Last we present the result of the building process.

5.1 The User's Perspective: How to Build a Matryoshka FMU from DACCOSIM

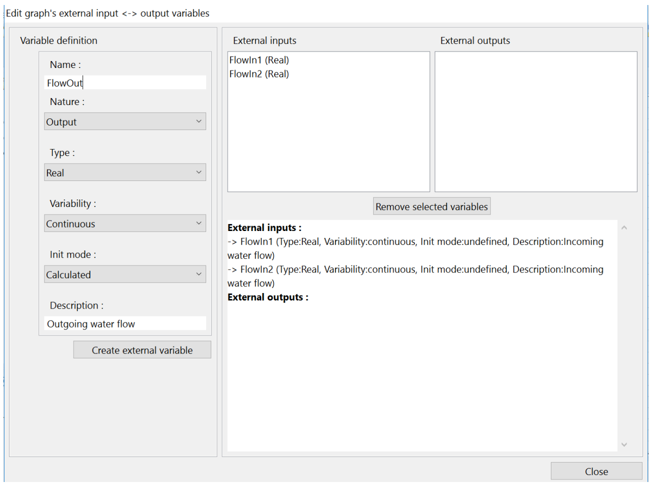
Exporting a Matryoshka FMU is quite simple for DACCOSIM users. Only a few additional steps are required after having designed the co-simulation graph.

During this initial stage, the user sets the simulation configuration as he would do for any co-simulation test case. These settings determine the internal behavior of the Matryoshka, with in particular:

- the **co-initialization mode** (none, sequential output propagation, Newton or a mix of both),
- the **step size control method** (constant step size or the adaptative Euler, Richardson or Adams-Bashforth methods) and its step size characteristics (initial, minimum and maximum step size),
- the **event detection method** (bisectional approach (Camus et al., 2016) or minimum step-size).

The user's only task is then to define the inputs and outputs of the Matryoshka FMU and link them to the variables of its internal FMUs:

1. The user uses a specific interface (Figure 5) to **create the external variables** of the graph and set for each **its name**, **causality** (input or output), **type** (real, integer, boolean, string, enumeration), **variability** (constant, discrete or continuous) and **initialization mode** (exact, approximated or calculated). Adding a **description** of the variables is also possible.
2. He **defines default initial values** for each external input variable.
3. He **adds external connectors**, connects them to the FMUs own connectors in the graph and associates their variables as depicted in Figure 6.
4. Finally he **generates the Matryoshka FMU** by clicking the toolbar export button of the GUI.


Edit graph's external input <-> output variables

Variable definition

Name : FlowOut

Nature : Output

Type : Real

Variability : Continuous

Init mode : Calculated

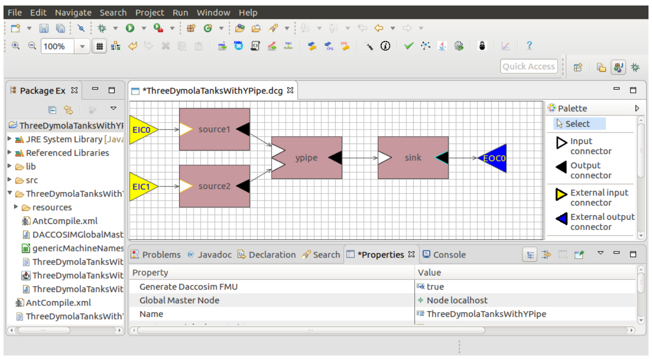
Description : Outgoing water flow

External inputs
FlowIn1 (Real)
FlowIn2 (Real)

External outputs

External inputs :
-> FlowIn1 (Type:Real, Variability:continuous, Init mode:undefined, Description:Incoming water flow)
-> FlowIn2 (Type:Real, Variability:continuous, Init mode:undefined, Description:Incoming water flow)
External outputs :

Figure 5. DACCOSIM interface enabling external IO definition



Property	Value
Generate Daccosim FMU	true
Global Master Node	Node localhost
Name	ThreeDymolaTanksWithYPipe

Figure 6. DACCOSIM graph with external connectors

5.2 Behind the Scene: the Steps Towards the Matryoshka FMU

We describe in the following subsections the sequence of actions that are automatically performed by DACCOSIM and result in the Matryoshka FMU generation when clicking the "Generate DACCOSIM Matryoshka FMU" button of DACCOSIM GUI toolbar.

5.2.1 Creating DACCOSIM master external API

DACCOSIM 2017 was augmented to be controlled from the outside when executed on a local machine. The result is a **DACCOSIM*GlobalMaster* class that is tailored to a particular co-simulation configuration**, and can be instantiated from another Java program. The obtained master class retains its internal mechanism specificities (multi-threaded architecture, adaptive step size control...) while adapting to the constraints imposed by the control program (external step size, input values...): if the internal step size leads to exceed the external step bound, the internal step size is truncated to meet this limit. It is afterward restored to its non-truncated value at the beginning of the following external step. Only DACCOSIM cluster features, i.e. its distributed architecture, are for now not exploited in the context of the Matryoshka FMU.

The master class is generated with Acceleo. It integrates a set of basic functions enabling external interactions with the master, among which:

- **instantiating** DACCOSIM global master;
- **setting the start and stop times** of the simulation;
- **setting and getting the value of the external variables** of the Matryoshka. Inner variables are not accessible for now;
- **changing the state of the master**, i.e. initialization or simulation mode;
- **performing the co-initialization of the internal graph** considering imposed input variable values;
- **performing a co-simulation step** whose size is imposed by the control program;
- **terminating** the co-initialization and/or co-simulation process.

5.2.2 Specifying Matryoshka interface to JavaFMI

All these functions are called by a **Java interface code extending the *FMISimulation* class** defined in the JavaFMI tools. This interface is used to perform the mapping between the primitives defined by the FMI standard and DACCOSIM master's own interaction functions. It is automatically generated with Acceleo. The modelDescription.xml file of the Matryoshka FMU is later generated based on the information specified in this interface class, and especially the list and characteristics of the external input and output variables of the DACCOSIM co-simulation graph.

One characteristic of the Matryoshka that has to be calculated prior to the interface generation is the *dependency of the external output variables regarding its external input variables*. This information is important when performing the co-initialization of a co-simulation scheme involving the Matryoshka to ensure that the variables are initialized in the correct order.

The calculation of the Matryoshka output dependencies is automated by DACCOSIM. It first computes the *oriented acyclical causality graph* of the Matryoshka co-simulation scheme (Figure 7). The graph is then reversibly parsed from the external outputs (blue dots) until its reaches the *graph seeds* that include the external inputs (large yellow dots). Optionally, this process can be disabled to let the user define the dependencies manually.

5.2.3 JavaFMI interface compilation with Ant

The two Java files (Master and Interface) are put into Java packages and compiled into a JAR using Ant. The Ant command file is tuned to each use-case and generated with Acceleo. The resulting JAR is an essential input component for the FMU builder.

5.2.4 Building the FMU using JavaFMI builder

The Matryoshka FMU is finally created by using JavaFMI builder (see Section 4). The following components are assembled as the super FMU resources:

- the **previously constituted Jar file**;
- the **resources required by DACCOSIM master**, i.e. the inner FMUs, the csv files defining the variables to log and the variables exchanged, a modelDescription file generated by DACCOSIM (different from Matryoshka's own modelDescription file generated by the builder, even though they're quite similar);
- the **library files required by DACCOSIM calculation engine**. If most are platform independent, a few such as 0MQ require OS specific components. This results in an OS specific Matryoshka that can be used either on Windows 64 bit or Linux 64 bit systems.

A simple call to the FMU builder command line pointing to these resources is then sufficient to automatically create the Matryoshka FMU.

5.3 What is a Matryoshka FMU like

The result is an FMU embedding DACCOSIM with the following capabilities:

- Can manage variable simulation time step.
- Can be instantiated several times.
- Cannot get and set FMU state, serialize its state or provide directional derivatives.

Generated Matryoshka FMUs have been successfully tested with the FMU checker, as well as imported and run in FMI 2.0 compliant tools (Dymola, DACCOSIM...).

6 Application to an Industrial Simulation Use Case

6.1 Presentation of the District Energy System Use Case

A District Energy System (DES) consists of components that enable the delivery of energy services in a district. This includes all possible carriers, most frequently electricy, heating, cooling and gas networks. Research interests mainly focus on the modelling of electrical and heat grids on a neighbourhood scale to optimize the topology and sizing of the electrical network, as well as to design the energy management system (Baggi et al., 2014; Zucker et al., 2016; Wetter et al., 2015).

6.1.1 Problematics

One of the main issues of such models is their lack of scalability, i.e. the inability to study a growing number of buildings connected to real size distribution networks in an appropriate amount of time: long time-scale simulation of a DES can thus easily reach limits in terms of memory and simulation time when using one generic solver since most of them are mono-threaded. As a result, the simulation of large and complex DES usually leads to simplifications either on the building side or on the network side. The alternative is to distribute the simulation by decomposing the problem into smaller interconnected sub-problems. DACCOSIM is then a suitable candidate tool.

6.1.2 Model Description

We consider a fixed district model written in Modelica (Baetens et al., 2015) involving 12 grid-connected Smart Buildings in a heterogeneous district (Figure 8).

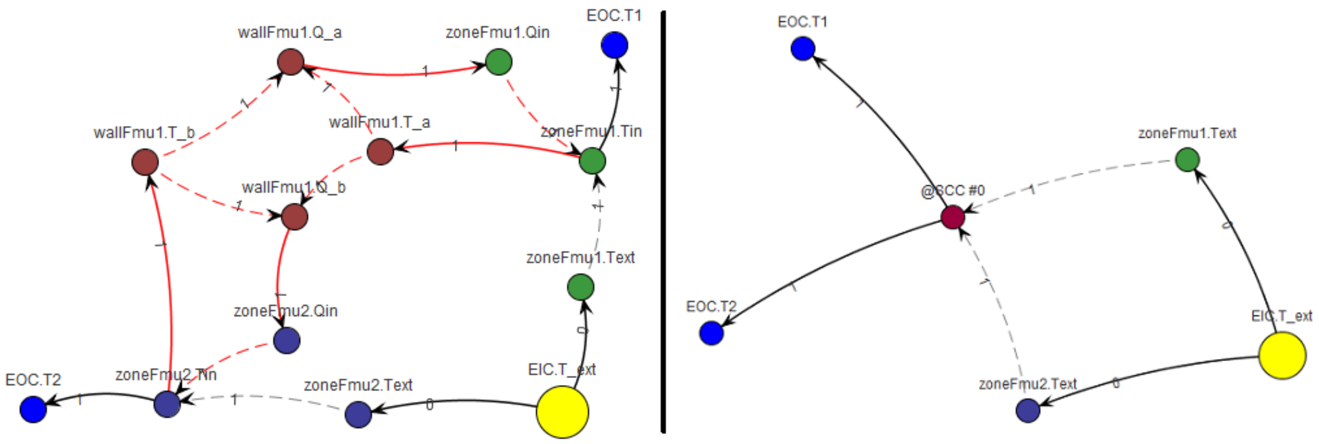


Figure 7. Complete causality graph (on the left) and its acyclic view (on the right) of a simple co-simulation graph.

For each building, we consider 3 thermal zones and one heating-pump (HP) connected to a 3-phases linear feeder. Thermal, electrical, ventilation, hot-water demand and occupancy profiles are heterogeneous and derive from a stochastic model (Baetens and Saelens, 2015). We employ complex quasi-stationary equations of the grid in order to study the influence of the load demand on the maximal/minimal tension of the grid (Protopapadaki et al., 2015). The impact of the MV network is also considered: it is modeled by a voltage source following real unbalanced LV busbar measurements. No hot water network is considered here.

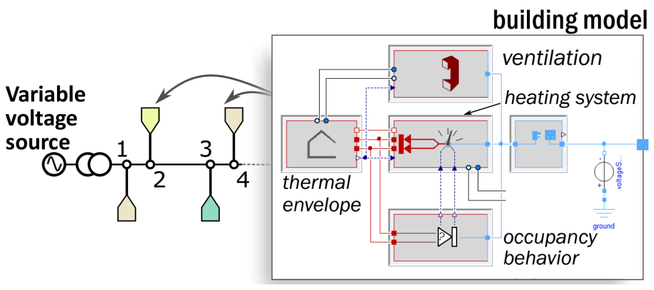


Figure 8. Illustration of the District Energy System use case

This basic scenario is already complex enough to exhibit scalability issues when using a standard solver. For illustration, it takes about 1 full day for 2 weeks of simulated time on our standard PC with Dymola 2016.

To distribute this use-case, the global model must be divided into multiple FMUs. The use of component-oriented modeling languages like Modelica usually makes the cutting decisions easy. Moreover, DES usually offer a lot of similarities, thus, one could consider creating communications between clusters of buildings, buildings or, even deeper, between heating systems, thermal envelope, and the network. In this section, we consider each building as one FMU, and the electrical network as another one, in order to simplify the understanding of the results. A smart handling of occupancy profiles has been implemented in the building models: the identity key, noted $idOcc \in \{1,..,12\}$, is related to resources profiles, *i.e.* electrical energy and hot water demands, occupancy and reference temperatures. This allows to keep the FMU general so that only a single profile needs to be loaded. The frequency of the network propagated to each of its components is also represented as an FMU, as is the LV voltage information imposed on the busbar.

This results in a system composed of 15 FMUs. All of them except for the LV voltage FMU are included within the Matryoshka FMU (Figure 9). This split allows to make the Matryoshka sensitive to its electrotechnical environment, i.e. to the MV network behavior. This would also allow to easily connect several DES Matryoshka FMUs to a MV network and see how they interact.

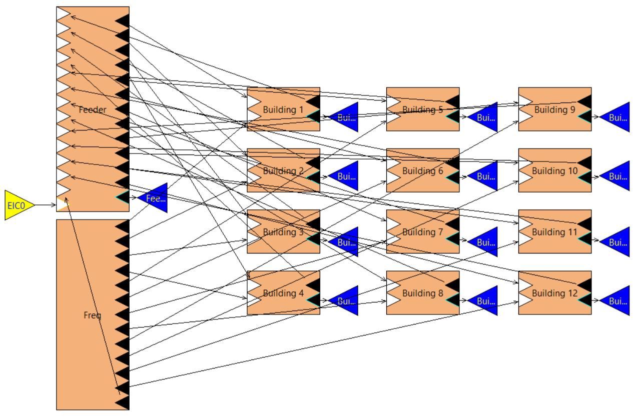


Figure 9. Screenshot of DACCOSIM showing the Matryoshka of the 12 Smart Buildings as a system of 14 FMUs with its external inputs and outputs

6.2 Description of the Experiments

The pure Modelica model simulated under Dymola is used as the reference for all the simulations performed in this article. This allows us to assess the performance of the other solutions in both their accuracy and computational time. The Matryoshka FMU is compared to this reference. This FMU is built by DACCOSIM integrating the electrical network and its 12 Smart Buildings as FMUs. Each inner FMU is generated using Dymola 2016.

Similar tests are performed in DACCOSIM and Dymola environments: considering a constant step size, a voltage is imposed to the DES system FMUs. This input comes from a Modelica block in Dymola and from its FMU counterpart in DACCOSIM.

The comparisons are carried out for a simulated time from one to five days. Realistic demand and occupancy data as well as weather data are used, therefore creating a variability in the calculations between each simulated day. The accuracy of several variables is relevant regarding the validation of the design of the electric grid with Smart Buildings, among them:

- the inner temperature of the buildings;
- the norm of the voltage and the current;
- the correct capture of the extrema of these quantities.

The Dymola model was simulated with tolerances of 10^{-4}, 10^{-5} and 10^{-6}. Output points are saved every 60 s. The Matryoshka is set up with a relative tolerance on each FMU internal solver of 10^{-4}, as well as a relative tolerance on the outputs of the FMUs of 10^{-3} for temperature and voltage, and 10^{-2} for currents. Euler algorithm is used inside the Matryoshka to manage the step size, with a minimum step size of 1 s and a maximum of 40 s. The Matryoshka is then simulated along the voltage data FMU with a constant step size of 60 s. The results obtained for these four configurations for a one and five days of continuous simulation are shown in Table 1. It is clear that the results of the Matryoshka co-simulations are closer to the ones of the Dymola model with a tolerance of 10^{-6} than to the other Dymola setups. Thus in the following, the performance of the Matryoshka will be compared with the Dymola model with a tolerance of 10^{-6} on longer simulations.

All the simulations are performed on a laptop computer with 4 physical cores and 8 logical threads with a maximum speed of 2.50 GHz (Intel i7-4710MQ) and 8 Gb of RAM running under Windows 8.1 64 bit.

6.3 Results

6.3.1 Matryoshka Accuracy

Providing sufficient accuracy is a key-aspect of a co-simulation. Splitting a model such a DES into smaller subparts exported and then interconnected as FMUs creates propagation delays: they depend on the largest number of linked FMUs separating the start from the end of the co-simulation graph and the sum of the varying time step sizes observed for each propagation sequence. It is thus important to use time step adaptive strategies to shorten the time steps when model dynamics are important and enlarge them when they stabilize.

Using Euler adaptive approach in the Matryoshka, we obtained the cumulated distribution displayed in Figure 10. It illustrates the repartition of the absolute error of the Matryoshka model to the pure Modelica model chosen as reference, respectively for voltage, current and temperature norms for the five days simulation case. 95.0 % and 99.8 % of the measurement points have an absolute error lower than 10^{-1} for respectively current and voltage. This is to be compared with the current Smart Meter capabilities: on voltage an accuracy of 0.5 % of the nominal voltage, i.e. about 1.15 V, is expected.

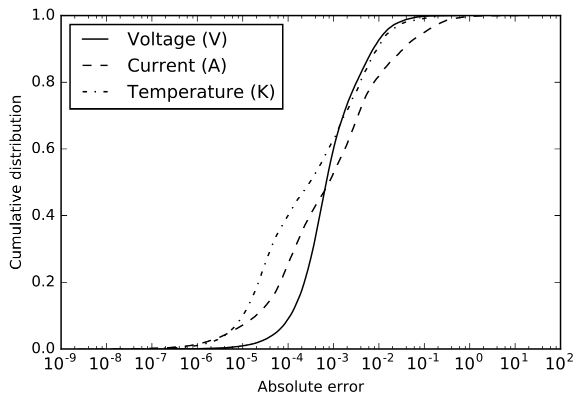


Figure 10. Cumulated distribution of the absolute error on voltage, current and temperature in one building zone: Matryoshka compared to pure Modelica simulation in the 5 days case

It is especially important to correctly capture the minimum and maximum values of both grid voltages and currents in order to properly design the grid. The extrema of the Matryoshka simulation should be close to the ones computed with the pure Modelica model so that we can use DACCOSIM results with as much trust as Dymola ones. Table 2 shows the minimum, mean and maximum error over the 12 buildings on the maximum values of current and voltage. The error is kept low with maximum errors of 5 mA, 0.7 mV and 1.6 mV.

With such error levels, using a Matryoshka in a co-simulation is relevant for distribution grid design. The representation of the use case dynamics as well as its accuracy are sufficient for a correct simulation of the network and its usages.

6.3.2 Matryoshka Computational Performance

The computation time of the pure Modelica model simulated under Dymola should not be lower than the one of the FMU co-simulation under DACCOSIM to make it relevant to use Matryoshka FMUs.

The results of the execution time measurement can be seen on Figure 11. The speed up starts around 1.5 for one day, grows and stabilizes itself around 3.5. This performance is quite interesting when doing simulation on long time scales. The changes of the speed up might be due to the variable calculation load induced by the different occupancy and weather profiles considered for every day.

Using a Matryoshka co-simulation also enables to tune the tolerance on the relevant variables when doing simulations for design purposes. The user can thus have the accuracy he needs in a shorter time.

Table 1. Performances of the different configurations for one and five days simulations

Model	Mean RMSE over the 12 buildings					
	Computation time (s)		*On current (A)*		*On voltage (V)*	
	1 Day	*5 Days*	*1 Day*	*5 Days*	*1 Day*	*5 Days*
Dymola (10^{-4} tolerance)	191	2633	1.34	8.69×10^{-1}	1.17×10^{-1}	7.10×10^{-2}
Dymola (10^{-5} tolerance)	169	2329	3.25×10^{-1}	2.52×10^{-1}	3.13×10^{-2}	2.04×10^{-2}
Dymola (10^{-6} tolerance)	139	2375	reference	reference	reference	reference
Matryoshka	79	666	7.99×10^{-2}	1.39×10^{-1}	7.08×10^{-3}	1.18×10^{-2}

Table 2. Errors on extrema aggregated on the 12 buildings

Error type	*Absolute Error*
Max. error on max. voltage	6.97×10^{-1} *mV*
Max. error on min. voltage	1.62 *mV*
Max. error on max. current	5.04 *mA*

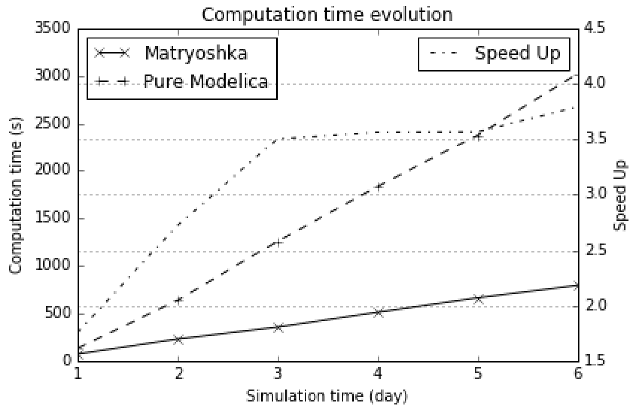


Figure 11. Computation time of the pure Modelica and FMU simulations with speed up

7 Conclusions and Future Work

The ***Matryoshka FMU*** we have presented in this paper and successfully implemented on a real-life test case is a ***first of its kind*** that is compliant with the latest version of the FMI 2.0 standard and built with an open-source solution DACCOSIM. The FMIBench commercial tool can also build hierarchical FMUs but supports fully only the version 1.0 of the FMI standard and we have no knowledge about the co-initialization and co-simulation features implemented within the embedded master.

By taking advantage of the FMI standard capabilities, a Matryoshka FMU can be easily ***integrated within any FMI-CS compliant simulator*** on any Windows or Linux 64 bits system. Such FMU could even be easily deployed on a node of a HPC-cluster environment. The use of DACCOSIM parallel master architecture allows to ***achieve both computational efficiency and accuracy*** thanks to its internal adaptive time step mechanisms and its capability to finely tune the tolerance on its variables. The JavaFMI builder makes its ***generation automatic***: once the simulated use-case is set, a single click in the DACCOSIM user interface generates such an FMU.

With DACCOSIM Matryoshka FMUs, complex real-life systems can thus be easily simulated, finely tuned, and improved in their computation efficiency while allowing an easy implementation within any FMI-CS compliant simulation environment.

Work is currently being carried out to further improve their capabilities. Some can be performed with the current FMI standard, while others require new attributes :

- When the user chooses the target platform and architecture (Linux, Windows or both) for the Matryoshka FMU we will check that the choice is conform with the platform(s) targeted by the inner FMUs.
- We are working to allow the Matryoshka FMU to save and restore its state (if all its inner FMUs have this capability). So, the Matryoshka could be included into any co-simulation which might require FMUs to rollback.
- We are investigating a way to build a Matryoshka which distributes its simulation on multiple cluster nodes. We also wish to include new information about the number of inner FMUs in its modelDescription.xml file. This would provide useful information about the number of created threads in order to automate its placement on HPC cluster nodes.

8 Acknowledgment

Authors thank Region Grand Est and RISEGrid institute for their support to this research. The modeling of the electrical network and the smart buildings was conducted within the EFRO-SALK project, which receives the support of the European Union, the European Regional Development Fund, Flanders Innovation & Entrepreneurship and the Province of Limburg.

References

R. Baetens and D. Saelens. Modelling uncertainty in district energy simulations by stochastic residential occupant behaviour. *Journal of Building Performance Simulation*, (September), 2015.

R. Baetens, R. De Coninck, F. Jorissen, D. Picard, L. Helsen, and D. Saelens. OPENIDEAS - An Open Framework for Integrated District Energy Simulations. In *Proceedings of Building Simulation 2015*, 2015.

S. Baggi, D. Rivola, V. Medici, G. Corbellini, D. Strepparava, and R. Rudel. Modeling and Simimulation of a residential Neighborhood with Photovoltaic System Coupled to Energy Storage Systems. In *29th European Photovoltaic Solar Energy Conference and Exhibition*, 2014.

T. Blochwitz and M. Otter. The Functional Mockup Interface for Tool independent Exchange of Simulation Models. *8th International Modelica Conference*, 2011.

B. Camus, V. Galtier, and M. Caujolle. Hybrid Co-simulation of FMUs using DEV and DESS in MECSYCO. In *Symposium on Theory of Modeling and Simulation*, 2016.

J. Evora, J. J. Hernandez, and O. Roncal. JavaFmi. URL `https://bitbucket.org/siani/javafmi/`.

V. Galtier, S. Vialle, C. Dad, J-P. Tavella, J-P. Lam-Yee-Mui, and G. Plessis. FMI-Based Distributed Multi-Simulation with DACCOSIM. In *Symposium on Theory of Modeling and Simulation - TMS'15*, 2015.

C. Protopapadaki, R. Baetens, and D. Saelens. Exploring the impact of heat pump-based dwelling design on the low-voltage distribution grid. In *14th Conference of International Building Performance Simulation Association*, 2015.

J-Ph. Tavella, M. Caujolle, S. Vialle, C. Dad, C. Tan, G. Plessis, M. Schumann, A. Cuccuru, and S. Revol. Toward an Accurate and Fast Hybrid Multi-Simulation with the FMI-CS Standard. *IEEE ETFA Track 9 - Information and Communication Technology in Energy Systems*, 2016.

J. Vaubourg, Y. Presse, B. Camus, C. Bourjot, L. Ciarletta, V. Chevrier, J-P. Tavella, and H. Morais. Multi-agent Multi-Model Simulation of Smart Grids in the MS4SG Project. In *PAAMS'15*, 2015.

M. Wetter, M. Bonvini, and T. Nouidui. Equation-based languages - A new paradigm for building energy modeling, simulation and optimization. *Energy and Buildings*, 2015.

G. Zucker, F. Judex, M. Blöchle, M. Köstl, E. Widl, S. Hauer, A. Bres, and J. Zeilinger. A new method for optimizing operation of large neighborhoods of buildings using thermal simulation. *Energy and Buildings*, 2016.