FMI-Based Distributed Multi-Simulation with DACCOSIM

Virginie Galtier
Stephane Vialle, Cherifa Dad
CentraleSupélec &
UMI GT-CNRS 2958
Metz, France
first.last@centralesupelec.fr

Jean-Philippe Tavella
Jean-Philippe Lam-Yee-Mui
EDF R&D, MIRE dept.
Clamart, France
first.last@edf.fr

Gilles Plessis
EDF R&D, ENERBAT dept.
Les Renardières, France
first.last@edf.fr

ABSTRACT

Our research project aims at enabling multi-simulation based on the FMI 2.0 standard and the cooperation of multiple FMUs (FMI simulation units). In order to support large scale multi-simulations, our solution (DACCOSIM) runs on multi-core and distributed architectures. To support variable step size, the necessary error control and rollbacks are achieved through a hierarchical and distributed control architecture. At each step, simulation data communications also occur, but directly between FMU pairs in a fully decentralized fashion. Moreover, DACCOSIM implements an algorithm to perform the complex initialization of the various components of the multi-simulation. DACCOSIM comes as a graphical framework to easily design a multi-simulation and to automatically generate associated code, and as a multithreaded and distributed library to execute it. We evaluated DACCOSIM on an industrial use case provided by EDF (leading French utility company), run on multi-core PCs and PC clusters. Preliminary performance measurements on a 4-physical-core PC exhibit a speedup compared to monothreaded Dymola execution using the same FMUs. On multi-core PC clusters we face overhead communication times due to frequent small communications but this distribution allows to process large co-simulations.

Author Keywords

FMI 2.0; multi-simulation; distributed architecture; framework; error control

ACM Classification Keywords

I.6.7 Simulation Support Systems; I.6.8 Distributed Simulation; D.1.3 Distributed programming; G.1.0 Error analysis

INTRODUCTION

Smart Grids, a Typical Complex System

We define complex systems as architectures involving a large number of heterogeneous components in interaction. We consider Smart Grids as an illustration: they are composed of many, various electrical equipments scattered across a wide geographical area from power plants to home meters, connected across electrical networks and along with telecommunication networks used to operate the electrical grid.

Pitfalls of Monolithic Simulation

Simulation is a valuable tool to study complex systems. But it is difficult to achieve for two main reasons. First, the model of the system inherits its complexity; it involves expertise in many different areas such as high-voltage systems, wireless network protocols, etc. but each community is used to different tools, and might use different time models. Second, since there are many components to simulate, a single computing machine might not have enough RAM, or might take too long to execute. What is needed is a simulation that is both distributed and component-based.

Overview of the Functional Mockup Interface for Co-Simulation

The automotive industry studied the multi-simulation challenge a few years ago in the MODELISAR European project, and came up with a standard to ease the exchange of models across modelling tools: the FMI (Functional Mockup Interface) [1]. A model is packaged as a black box called FMU (Functional Mockup Unit) which can be plugged as a component into a larger model thanks to the standard interface that provides access to states and derivatives of the included equations. A solver might also be included in the FMU along with the model. This is the type of FMUs (so-called ”CoSimulation FMUs”) we are considering in our work. A multi-simulation can be built using FMU blocks: during a ”macro-step” of the simulation, each FMU independently simulates part of the system; and at the end of each macro-step, the outputs from some FMUs provide new initial values (or inputs) to other FMUs.

Objective

As implied by the name, the FMI only provides an interface to the simulation units. It is up to the user to perform (in a so-called ”master code”) the required output-input transfers among FMUs, to choose the macro-step size and to order the execution of each FMU. Some tools, such as Dymola, are able to import FMUs, then let the user draw connections between them and execute the resulting multi-simulation. Yet, they are not multithreaded and consequently take no advantage of multi-core computation nodes to speed-up the simulation; nor do they support FMUs distributed among nodes to cope with large simulations; besides, they are not open-source. Our objective is to provide an open-source solution that is both easy to use and efficient.

Structure of this Article

Section ”Related Work” presents how DACCOSIM compares to other distributed multi-simulation solutions. Section

TMS/DEVS 2015 April 12-15, 2015, Alexandria, VA

”Principles and Architecture” sketches the structure of DACCOSIM and explains how it operates the distributed, controlled multi-simulation of FMUs. Section ”Framework” describes how DACCOSIM empowers the user to transparently exploit the architecture (functionalities and implementation choices). Section ”Experimental Results” presents our test case and reports the achieved performances. Lastly, the conclusion recaps our realization and outlines future directions for our work.

RELATED WORK

Multi-simulation Error Control Challenges

At the opposite of a classical large monolithic simulator, a FMI based co-simulation runs a graph of smaller simulators encapsulated into FMUs, controlled by a Master process and communicating only at the end of their time steps. This division can lead to numerical errors on the output values, and [2] and [3] propose different mathematical solutions to correct the input values at the end of each time step. However, these approaches lead to significant extra amounts of computations and communications, and as of today the multi-simulations achieved with DACCOSIM have not required this kind of regular corrections.

Solvers used by a FMU usually break one FMU time step into numerous smaller internal time steps. But the FMU input values do not evolve during a FMU time step, because new values of the connected outputs are only routed at the end of each FMU time step. In 2014 [4] proposed to achieve a contextual extrapolation, function of past input values, polynomial approximations and of some knowledge about the simulation evolution. We also implemented some degree of input extrapolation computations in DACCOSIM, but in order to be generic and easy to use, they do not require any particular knowledge about the simulation and only use some facilities of FMI 2.0 (see subsection ”Input Extrapolation”).

When connected FMUs form complex chains and include algebraic equations, some order has to be identified and respected when propagating values in the FMU graph, as pointed out by [5]. We reused this approach in a more complex algorithm (see subsection ”Co-Initialization”) to compute all the initial input and output values in our FMU graph.

Speedup and Scalability Challenges

Independently from these numerical issues, some research works focus on the parallelization and distribution of a FMU graph, in order to speedup and/or to scale up the co-simulations. A multithreaded FMI based co-simulation is introduced in [4] and allows exploiting a multicore machine. It achieves a supra-linear speedup greater than 8.9, running 5 threads on a 8-core machine including 2 processors (probably using both more cores and more cache memory). However, this solution is limited to multi-core machines and can’t be deployed on a cluster.

A distribution of a FMU graph on a cluster can be achieved by interfacing each FMU with a distributed HLA bus (the ”RTI”), initially designed to manage ”events” and event based co-simulations [6]. The FMUs become components of the HLA federation (the ”federates”) and the RTI is used as a distributed middleware to communicate output values at each step. This strategy has been experimented with in 2013 at the Austrian Institute of Technology, where several distributed versions of the Master algorithm controlling the FMUs were developed. They focus on different objectives: parallel execution of the distributed FMUs, or accuracy of the co-simulation, or possibility to support different adaptive step sizes in the different FMUs. This last objective allows reducing the number of events transmitted on the HLA bus limiting in turn the extra cost of this distributed FMI co-simulation, see [7]. These solutions rely on a fine usage of the numerous HLA functionalities, and on more or less complex algorithms achieving local controls of each FMU encapsulated in each HLA federate [8, 9]. However, no performance measurements of these FMI+HLA solutions were reported.

In 2014 the C2WT co-simulation framework developed at the Vanderbilt University (USA), integrated some FMUs in a HLA based co-simulation composed of event based simulators, in order to develop hybrid co-simulations [10]. However, the distributed control of the FMUs supported only constant time steps and delayed some event processing at the end of the FMU time steps which would not be acceptable in our Smart Grid use cases.

In contrast, DACCOSIM has both a multithreaded and distributed architecture, in order to achieve speedup and scalability.

PRINCIPLES AND ARCHITECTURE

The FMI standard calls ”master” the component in charge of orchestrating the multi-simulation and it provides a sketch for a simple, centralized master, with local FMUs. This architecture is not suited for large simulations though, as it does not scale (all the data exchanges going through the master result in a bottleneck, and the execution machine can be overloaded). In DACCOSIM, the master functionality is carried out by a combination of three kinds of components distributed across the various computation nodes. The architecture is pictured in figure 1.

Coordinated Variable Step

Each *FMUWrapper* is in charge of direct interactions with a FMU and of data exchange with other FMUWrappers. With a simple co-simulation schema, all the FMUs in the co-simulation would perform a simulation step of size h; h would be the same for all FMUs and would not vary during the simulation. After step i, FMUWrappers would exchange data according to the user-defined connection graph and once a FMU would have received all of its updated inputs, it would perform step $i+1$. While this constant-step strategy is easy to implement in a fully decentralized fashion, it rarely presents a good computation/accuracy ratio: the choice of a small value for h results in a large number of computation steps, while a large value might fail to capture some variations in the simulated variables.

For that reason, we choose to implement a variable-step strategy. After the simulation of step i, each FMU examines its outputs and estimates how far they are from the exact value.

DACCOSIM implements two algorithms which do that: one is based on the Euler's method and a second one is based on Richardson's method. Other low-cost solutions (calculation-wise) relying on variable Adam-Bashforth methods are also suited for FMUs. Their principle is to store the values of the derivatives at consecutive communication points to infer an estimation at the next iteration. The explicit Euler scheme implemented in DACCOSIM is merely a particular case of Adam-Bashforth at order 1. If the (local) error is found to be tolerable, the FMUWrapper 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 by the version 2.0 of FMI which introduced the notion of FMU state, allowing the serialization 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 t_{10}, *A* decides it should redo the step with 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 t_{10} to t_{20}). When *A* reaches t_6, it could send its outputs to *B* but they would not make much sense since *B* already advanced to t_{10} (and the next available outputs from *A* could be time stamped t_{12}, which is not satisfactory either). To avoid this situation, we introduce additional coordination in our architecture: if one FMU decides to rollback, all the FMUs will do the same 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. To reduce the number of messages involved in this global decision making process, we introduce two additional components to the architecture: on each computation node, a local master collects the proposed step size from each FMU on the same node and sends the local minimum value to the global master. The global master in turn collects the proposed step sizes from each local master and sends them back the global minimum value, which they propagate back to the FMUWrappers.

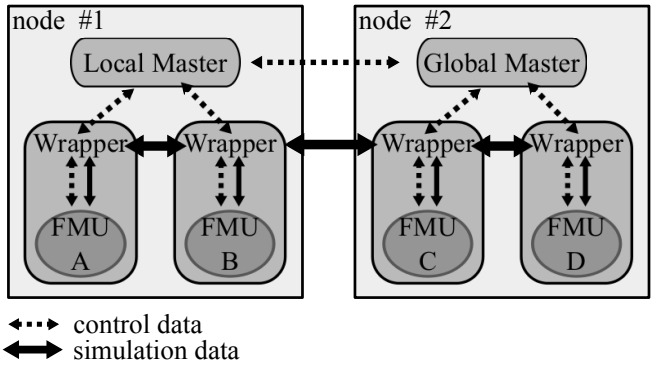

Figure 1. Co-simulation components architecture.

A drawback of this strategy is that the system tends to align its step size value to the most sensitive component, while some components with larger time constants could easily provide acceptable results using a larger step size. This results in additional, unnecessary computations. Researchers at LORIA-INRIA are currently working on an alternate, Multi-Agent System based architecture [11], which supports variable step size, with different step sizes among the FMUs but with the hypothesis that no rollbacks are necessary. We plan to jointly investigate the differences it will cause both on performance and on result accuracy.

Our architecture proposes 2 modes of operation regarding the synchronization:

- In the *optimistic mode*, a FMU starts to send its outputs before it receives the global decision (data communications and control operations are executed in parallel); since interrupting data communications in the case a rollback happens to be necessary would be too costly, this scheme is of course more efficient in the event there are not too many rollbacks.

- In the *prudent mode*, a FMU waits for the global decision reception to transmit its outputs if necessary. Figure 2 illustrates this option.

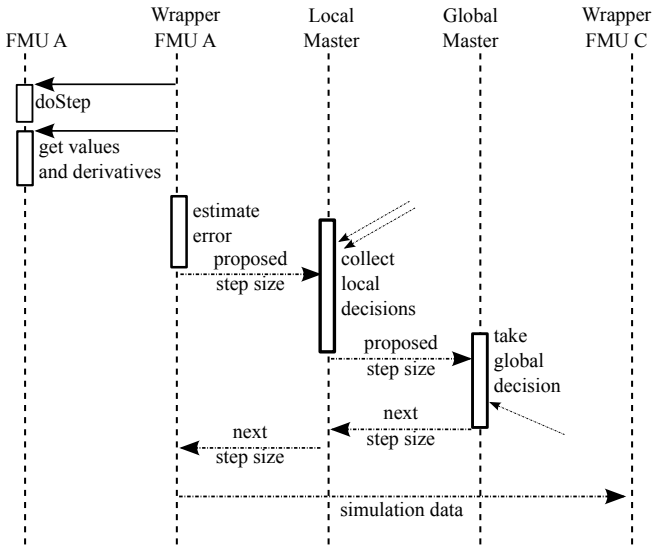

Figure 2. Co-Simulation "prudent" synchronization.

Simulation data are not transmitted to the master hierarchy, rather data communications take place in a point-to-point fashion, directly between the source and recipient FMUWrappers. Subsection "Communications" presents how this is implemented, depending on whether communications are local to a node or take place between separate nodes.

Input Extrapolation

FMI 2.0 offers the possibility to provide not only input values for a FMU, but also derivatives for those values. It can be used by the FMU internal solver to perform extrapolation of input values and hopefully to provide more accurate results. This possibility was included in DACCOSIM which transfers not only output values from source FMUs to inputs of sink FMUs but also first order derivatives. Of course the gain in accuracy comes at the cost of increased computation time and larger communications.

Co-initialization

One of the difficulties of the kind of distributed multi-simulations we are considering is the setting of consistent system-wide initial values for all the components. Our co-initialization 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, and it uses dependency information from the ModelDescription.xml file of a FMU to find internal dependencies.

The key idea is that a topological sorting of the directed acyclic graph (DAG) naturally gives the order in which the variables must be initialized. 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). The resulting graph in which each SCC has been contracted into a single vertex is a DAG[1]. We use Tarjan's SCC algorithm [12] 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 initialization 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 calculation involves in this case the following steps:
   (a) For each connection it has with others FMUs, the FMUWrapper computes the residual value and transmits these values to the local master.
   (b) Each local master builds the local Jacobian matrix of all the connections involved in the cyclic dependency being initialized, then transmits it to the global master.
   (c) The global master aggregates the Jacobians, computes the correction to apply to the inputs, and transmits this correction value back to the local masters which in turn propagate it towards the FMUs.

   Those steps are repeated until the residual value is below a given threshold.

In the most general case, a global dependency graph contains several cyclic areas that may be seen as super-nodes acyclically connected. A prototype algorithm is being developed mixing local JNRA resolution combined with an acyclic output propagation.

DACCOSIM allows the concurrent initialization of concurrent independent cyclic dependencies.

[1]The proof follows from the definition of SCC: a SCC is maximal in that no additional edges or vertices can be included in the SCC without breaking its property of being strongly connected.

State-Events Detection and Handling

We call "state-event" the fact that a boolean or integer output value changes[2]. In the current FMI version (2.0), the notion of event is absent for Co-Simulation FMUs (it only exists for Model-Exchange FMUs). As a result, events are captured only at the end of a simulation step, which can have detrimental consequences on the simulation if the step size is large. Future FMI versions (2.1) are expected to handle hybrid simulations (mixing continuous time and event-based models). Meanwhile, DACCOSIM offers a solution to detect state-events and to capture their instant of occurrence as precisely as possible. DACCOSIM examines the value of boolean and integer outputs and if a change is detected (compared to the previous step), a rollback procedure is triggered. Then the simulation is resumed with the smallest step size allowed by the user, until the event is detected. Of course if the boolean value changes an even number of times during the step, the events get undetected. To avoid that situation (until FMI 2.1 is released), the user must carefully choose the maximum step size.

FRAMEWORK FUNCTIONALITIES AND BASES

Remember our objective is to propose an architecture which can exploit distributed resources to perform efficient and accurate multi-simulation of FMUs. In addition, it should be easy to use and to maintain. This section gives an overview of the tools offered to the user by our framework, and the underlying technologies used.

User Modelling Interface

The user-visible side of DACCOSIM comes as an Eclipse plug-in which allows the user to build its multi-simulation in a user-friendly way. Using a form, the user describes his/her computation nodes (OS, architecture, number of cores...). Next the user can pick the different FMU archive files from the file system he wants to see loaded in his palette. He can then drop FMUs from the palette to his main design area where he organizes the FMUs to model his complex system. With simple mouse operations, the user connects an output from a FMU to the input of another one. The user may also discover the initial value for a variable (read from the ModelDescription.xml file), and he can easily set a different value. For now, the user also needs to associate each FMU with a computation node. Figure 3 gives a glimpse of the interface offered to the user. It is rather similar to commercial tools such as Dymola for instance and, while we have not tested that yet, we expect a "regular" user wouldn't feel lost.

We used EMF (Eclipse Modeling Framework) to create the metamodels, GMF (Graphical Modeling Project) to visualize them as graphs, and GEF (Graphical Editing Framework) to create the associated editors.

The framework performs a number of validations on the graph. Next it generates the code for the various FMUWrappers, local masters and for the global master. This step uses

[2]If a real output value reaching a given threshold needs to be considered as an event, we assume that this condition will be tested by the model designer and that he will provide a boolean output to signal that event.

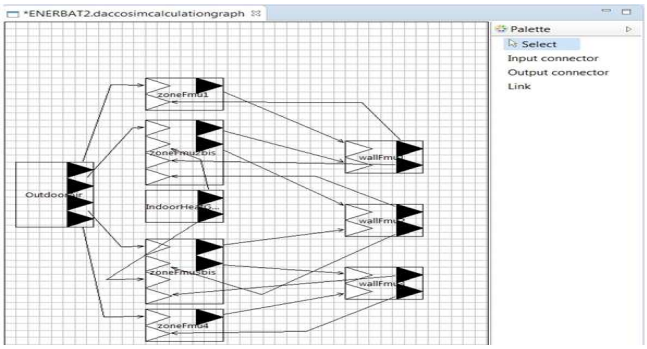

Figure 3. Screenshot of the simulation configuration GUI.

a code generator based on Acceleo[3]. For now, the user is responsible for moving the generated files onto the appropriate computation nodes and for launching the executions.

The framework also contains a tool which displays the dependency graph and highlights cycles (if any) (see Figure 4). It can also be used to identify dependency chains in order to avoid useless initializations (it is pointless to initialize two variables in an algebraic chain as the value of the upstream variable will lead the chain).

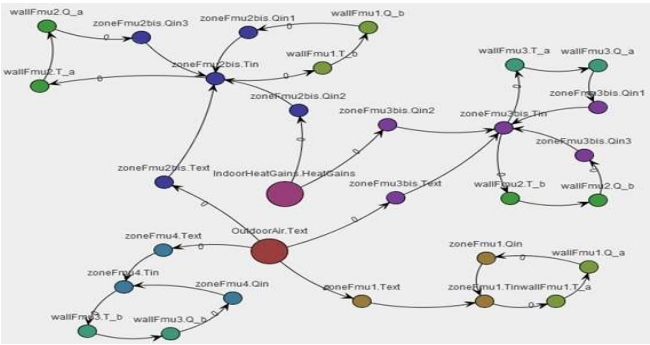

Figure 4. Screenshot of the dependency graph visualization.

DACCOSIM Library

Multithreaded Architecture

Each component in Figure 1 runs on its own thread. Additional threads are introduced to allow for some emission and reception of data and control messages concurrently with some computations, and to allow adaptive step methods to compute auxiliary steps in parallel with the computation of the current step.

Interacting with FMUs

We developed two versions of the library. In the first one, to load the FMUs and interact with them, the FMUWrapper code uses JavaFMI [13][4]. Using java makes it easier to provide a solution for both Linux and Windows.

Along the last few months of this project, the FMI standard was evolving from version 1 to version 2, and we exploit some of the new functionalities. Tools used to generate FMUs and to exploit them were not exempt of bugs and we've sometime had difficulties to validate our code. That is one of the reasons why we've also developed a C++ version of the library. Another reason is that C++ executes faster than Java. This version is built upon the QTronic SDK [14] and only runs on Windows.

Our preliminary tests first verified that both versions provide the same simulation results. Since the Java version spends some time in JNA calls (which of course are not necessary with the C version, it can natively call the DLL embedded inside the FMU), it is slower than the C version (by about 30%). As our FMUs are not computational intensive, the relative amount of time spent in JNA calls might not be as detrimental with heavier FMUs.

Communications

The technology used to exchange data between FMUWrappers depends on their respective location. If 2 FMUs are not on the same node, a TCP connection is used. If they are on the same node, it is more efficient to use an *inproc* communication. To provide a rather similar interface for the developer in both cases, the ØMQ middleware is used. We have found than the inproc implementation from JZMQ [15] performs sometimes faster than a shared queue, and sometimes not. We will further investigate that matter, and for the time being, both intra-node communication modes are available in DACCOSIM and it's up to the developer to pick whichever he prefers.

The communications from FMUWrappers to their local master use a shared queue, and the communications between the local masters and the global master use TCP connections (via ØMQ); a local master holds references onto its FMUWrappers so communications from local masters to FMUWrappers are simply Java method calls.

Code Availability

The code is available for Windows and Linux, both 32 and 64-bit. It will be distributed under an open source license on January 2016. It will be downloadable from `https://daccosim.foundry.supelec.fr`.

EXPERIMENTAL RESULTS

Use Case and Testbeds Description

The test case used for this benchmark is a first, simplified version of an industrial case and is provided by EDF R&D. It represents heat transfers in a building composed of four rooms, as a future component of a Smart Grid. The building envelope is subject to simple boundary conditions corresponding to the outdoor air temperature (for the four zones) and the internal heat gains (only for zones 2 and 3), see figure 5.

The building envelope is described in nodal form and, for the sake of simplicity, only the conductive and convective heat transfers are considered. The walls are discretized in one dimension, along their depth, and represented by a set of heat capacitors and thermal conductors in Modelica language. The thermal zones are represented by a single heat

[3]Eclipse Foundation, implementation of the OMG Model to Text Language (MTL) standard.

[4]JavaFMI has been updated multiple times and is now compliant with FMI v2-RC2. We use a slightly modified version for now.

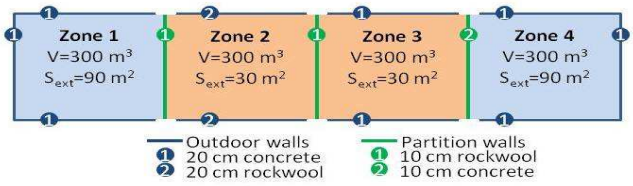

Figure 5. Heat transfers use-case.

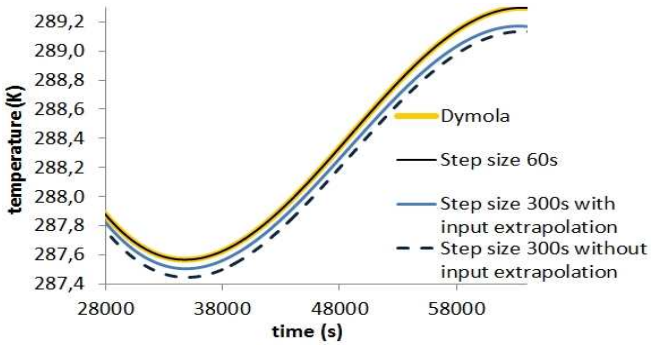

Figure 6. Simulation accuracy.

capacitor, meaning that the air temperature is considered uniform, using the so-called well-mixing assumption. It leads to an ODE system composed of 33 state variables. The whole system is divided into components, which will be distributed on threads: 4 models of rooms, 3 models of partition walls, and 2 models for the boundary conditions (outdoor air temperature and internal heat gains).

In order to generate FMUs, the Modelica models should be "oriented", the acausal connectors are broken down into a set of causal connectors (inputs and outputs) carrying temperature and heat flows. The FMUs were generated with Dymola 2015 for both Windows and Linux OS. The whole building model can be assembled in DACCOSIM by connecting inputs and outputs of corresponding FMUs.

We conducted 2 kinds of benchmarks on 2 different testbeds. First, we run some Dymola sequential simulations and some multithreaded DACCOSIM multi-simulations on a laptop PC at EDF with an Intel Core i7-3840QM processor at 2.8 GHz, including 4 hyperthreaded physical cores (8 logical cores), and with 32 GBytes of global DDR3 RAM on a 932 MHz memory bus. This PC was running under Windows-7 64 bits. Second, we run some distributed benchmarks of DACCOSIM on the experimentation cluster *Cameron* of SUPELEC, with 16 nodes that are interconnected across a 10-Gbit Ethernet switch, an OmniSwitch Alcatel 6900-X20-F, with up to twenty 10-Gbit/s ports. Each node has an Intel Xeon E5-1650 processor at 3.2 GHz, composed of 6 physical hyperthreaded CPU cores (12 logic cores), and is equipped with 8 GBytes of global DDR3 RAM on a 1600MHz memory bus. This cluster is operated under Linux 64 bits, fedora core 16.

Simulation Results

Figure 6 compares the simulation results obtained with different configurations. The values used as a reference for evaluating the simulation accuracy are computed by Dymola using the Modelica model. Using a constant step size of 60 seconds gives the same values as the Dymola reference. As expected, increasing the step size to 300 seconds leads to slight inaccuracy. The graph also demonstrates the positive influence of input extrapolation over the simulation accuracy.

The co-initialization algorithm implemented in DACCOSIM was validated against the results of a load-flow (numerical analysis of the power flow in an electrical grid) performed with Dymola. We obtain the same results with our Newton-Raphson-based algorithm, see figure 7.

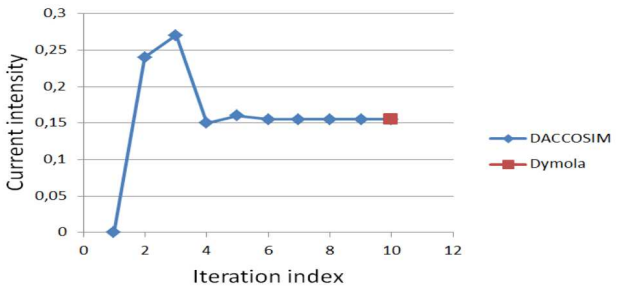

Figure 7. Co-initialization accuracy.

The event detection mechanism has been tested on a Medium Voltage network equipped with EDF PWH protections. The circuit breaker opening is triggered at $1.08273\ s$ according to the Dymola reference. Table 1 shows that DACCOSIM using a variable step size is able to detect this event as precisely as when using a small constant step size, and as expected with a better simulation time.

	constant step size	variable step size
Breaker opening	1.0830 s	1.0835 s
Simulation time	1.90 s	1.53 s

Table 1. Adaptive step for event detection.

Performances

This section evaluates our solution from the execution time perspective.

Performances on Multi-core Machine

Dymola is able to perform multi-simulations using FMUs. In a first set of experiments, we've compared the execution time of DACCOSIM C++ version on a single 8-core machine with Dymola execution time (and a constant step size configuration in each case). We've checked that the operational results are similar. DACCOSIM exhibits a speedup of 3.5 (see "multithreaded execution time" in figure 8). The main explanation is that as opposed to Dymola[5], DACCOSIM is multithreaded and thus it better exploits the 8-core architecture.

Performances on Cluster

We've also distributed our use case on up to 9 computation nodes and compared how long it took compared to the execution on a single node. We have tried several preliminary

[5]2 Dymola instances can be run on the same machine and communicate with one another but each instance is monothreaded.

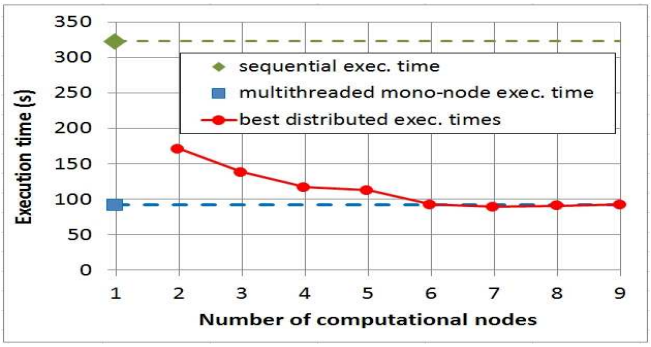

Figure 8. DACCOSIM execution time on cluster.

	w/o inputs interpol.	w inputs interpol.
constant step (300s)	196.6	302.4

Table 2. Influence of input extrapolation on execution time.

distribution strategies: among others, we have tried to balance the computation load, or to allocate the same node to FMUs which communicate a lot, or to balance the number of connections established by the FMUs on each node. We obtain the following results:

- Figure 8 shows the best execution time obtained on each number of nodes for a constant step size configuration. The FMUs do not perform huge computations (average doStep lasts less than 4 ms) and in the chosen scenario they exchange a very large number of very small messages (data part is no more than 34 bytes long). So, communication times are mainly constrained by the latency of our cluster network. As a result, we could not expect an interesting speed-up. Only the 6, 7 and 8-node configurations exhibit a speed-up slightly above 1. But at least if the simulation was using too much RAM to run on a single PC, DACCOSIM could be used to execute it anyway.

- Since in this use case the communications account for a large portion of the simulation time (because of their number), the strategy which leads to the best results is the one which balances the number of connections (but this strategy is only slightly better than the others).

- Richardson's method doubles the number of computations performed at each step, without impacting the communications. As a consequence, this case is more in favour of an execution on a cluster. And indeed, we observe a speedup close to 2 when increasing the number of computation nodes from 1 to 7. This advocates for better speedup than the ones currently reported with constant step once more computational intensive FMUs are available.

Figure 9 summarizes the best speed-up achieved on multi-nodes and multi-cores platforms.

In a second set of experiments, we've repeated the experiments from subsection "Simulation Accuracy" and we've measured the execution time. Table 2 presents the execution times obtained when turning on or off the input extrapolation.

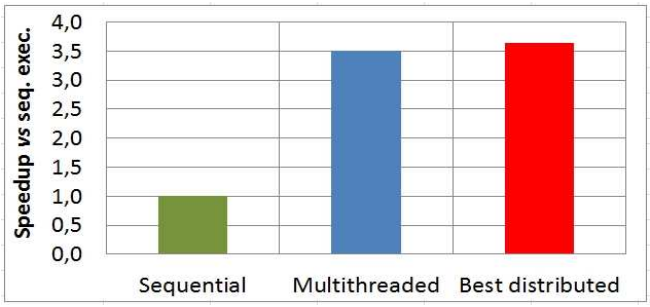

Figure 9. Parallel and distributed speedup.

CONCLUSION AND PERSPECTIVES

Functional Mockup Units are an interesting opportunity to build simulation of complex systems by assembling models (and associated solvers) from various fields of expertise. In addition, breaking a large model into smaller components (and resulting FMUs) makes it more easy to distribute the simulation across a set of computation nodes to scale-up and/or speed-up the simulation.

DACCOSIM offers a GUI for the user to describe how to interconnect FMUs in order to simulate his system. Once the multi-simulation is configured, all the code required to orchestrate the FMUs is generated and ready to be deployed on a cluster. The main contribution of this work is the ability to perform parallel and distributed multi-simulation (as opposed to the monothreaded simulation carried out by Dymola for instance). Data exchange between FMUs is fully decentralized. Our system supports both constant and variable step simulation, and a light hierarchical control architecture was introduced to coordinate the evolution of the step. DACCOSIM is also able to perform the co-initialization phase in some cases. Our tests show that this co-initialization phase, when applicable, followed by the simulation of multiple FMUs using DACCOSIM provides the same results as the Modelica model in Dymola. Our benchmarks also confirm that our multi-threaded architecture leads to shorter execution times than the same FMU-based Dymola simulation (3.5 time slower on our use-case). We achieved a speedup slightly above 1 on a cluster, because the problem was not computation intensive enough to be worth the distribution.

Future works are envisioned along multiple directions:

- Currently, the user needs to specify on which computation resources each FMU should execute. Given the description of available resources, the connection graph, and information about each FMU, we could automatically compute a deployment plan. This plan will ensure than FMUs available for Windows only will get allocated a Windows host for instance, and that a pair of FMUs with a lot of connections will sit on the same host rather than being separated on different machines. In addition, once the code is generated, we could provide a deployment tool to replace the current user-designed shell scripts.

- Due to difficulties in obtaining FMUs, our test case was reduced to 9 "small" FMUs. In the future we will experiment with real, large, Smart-Grid centric use-cases.

- While FMI is suited for discrete time simulations, the High Level Architecture (HLA [6]) is a reference for distributed event-based multi-simulations. Some parts of a complex system might be simulated using HLA-compliant components while others would be provided as FMUs. We are currently investigating how to efficiently assemble both classes. [16] proposes a solution to automatically integrate FMUs as federates. We expect our approach to differ in 3 ways: (1) a group of FMUs will be connected to the RTI as a single federate within which rollbacks can occur, (2) we will rely on JavaFMI to access the FMU data structures, and (3) user action will be requested during preprocessing in order to be able to semantically match all HLA events and interactions to actions applied to the FMU.
- Once FMI 2.1 is released, we will modify our mechanism to handle state-events.
- We are aware of on-going work using a multi-agent approach to perform a decentralized co-initialization at the French IRIT lab. We intend to closely assess if it could provide an interesting alternative to the centralized Newton-Raphson approach implemented in DACCOSIM.
- DACCOSIM lacks easy-to-use and efficient tools for online and/or post-mortem results retrieval, storage and analysis. Designing and implementing such tools to perform across distributed architectures require careful consideration about distributed I/O. We plan to investigate this issue in the medium term.

ACKNOWLEDGMENTS

This study has been carried out in the RISEGrid Institute (**`http://www.supelec.fr/342_p_38091/risegrid-en.html`**), joint scientific program between CentraleSupélec and EDF (*Electricité de France*) on smarter electric grids. The first version of the modeling and GUI parts of DACCOSIM were developed by Thomas Molines, student at Supélec. The authors also wish to thank José Juan Hernández's team at SIANI for the efforts they put in providing us JavaFMI.

REFERENCES

1. Modelica Association Project FMI. Functional Mock-up Interface for Model Exchange and Co-Simulation, version 2.0, July 2014. https://www.fmi-standard.org.
2. Sicklinger, S. and Belsky, V. and Engelmann, B. and Elmqvist, H. and Olsson, H. and Wüchner, R. and Bletzinger, K.-U. Interface Jacobian-based Co-Simulation. *International Journal for numerical methods in engineering*, March 2014.
3. A. Viel. Implementing stabilized co-simulation of strongly coupled systems using the Functional Mock-up Interface 2.0. In *10th International Modelica Conference 2014*, Lund, Sweden, March 2014. Modelica Association and Linköping University Electronic Press.
4. A. Ben Khaled, L. Duval, B. Gaïd, M. El Mongi, and D. Simon. Context-based polynomial extrapolation and slackened synchronization for fast multi-core simulation using FMI. In *10th International Modelica Conference 2014*, Lund, Sweden, March 2014. Modelica Association and Linköping University Electronic Press.
5. D. Broman, C. Brooks, L. Greenberg, E. Lee, M. Masin, S. Tripakis, and M. Wetter. Determinate Composition of FMUs for Co-simulation. In *Proceedings of the Eleventh ACM International Conference on Embedded Software (EMSOFT'13)*. IEEE Press, September 2013.
6. 1516-2000 IEEE Std. for Modeling and Simulation (M&S) High Level Architecture (HLA), 2000. http://standards.ieee.org/findstds/standard/1516-2000.html.
7. M. U. Awais, P. Palensky, W. Mueller, E. Widi, and A. Elsheikh. Distributed hybrid simulation using the HLA and the Functional Mock-up Interface. In *39th Annual Conference of the IEEE Industrial Electronics Society (IECON)*, Vienna, Austria, November 2013.
8. M. U. Awais, P. Palensky, A. Elsheikh, E. Widi, and S. Matthias. The high level architecture RTI as a master to the functional mock-up interface components. In *International Conference on Computing, Networking and Communications (ICNC)*, San Diego, USA, Jan. 2013.
9. M. U. Awais, W. Mueller, A. Elsheikh, P. Palensky, and E. Widi. Using the HLA for distributed continuous simulations. In *The 8th EUROSIM Congress on Modelling and Simulation*, Cardiff, UK, Sept. 2013.
10. H. Neema, J. Gohl, Z. Lattmann, J. Sztipanovits, G. Karsai, S. Neema, T. Bapty, J. Batteh, H. Tummescheit, and C. Sureshkumar. Model-Based Integration Platform for FMI Co-Simulation and Heterogeneous Simulations of Cyber-Physical Systems. In *10th International Modelica Conference*, Lund, Sweden, March 2014. Modelica Association and Linkoping University Electronic Press.
11. J. Siebert, L. Ciarletta, and V. Chevrier. Agents and artefacts for multiple models co-evolution. Building complex system simulation as a set of interacting models. In *Autonomous Agents and Multiagent Systems - AAMAS 2010*, volume 1, Toronto, Canada, May 2010. ACM.
12. R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2), 1972.
13. SIANI, University of Las Palmas, Spain. JavaFMI, 2014. https://bitbucket.org/siani/javafmi/wiki/Home.
14. QTronic. FMU SDK 2.0.3, 2014. http://www.qtronic.de/en/fmusdk.html.
15. JZMQ. Java binding for ØMQ, 2014. https://github.com/zeromq/jzmq.
16. Faruk Yilmaz, Umut Durak, Koray Taylan, and Halit Oguztüzün. Adapting Functional Mockup Units for HLA-compliant Distributed Simulation. In *International Modelica Conference*, Lund, Sweden, March 2014.