Cossembler - Rapid Prototyping Tool for Energy System Co-simulations

Miloš Cvetković, Digvijay Gusain, Peter Palensky
Faculty of Electrical Engineering Mathematics and Computer Science
Delft University of Technology
Delft, The Netherlands
Email: {m.cvetkovic,d.gusain,p.palensky}@tudelft.nl

***Abstract*—Cossembler (co-simulation assembler) is a rapid prototyping tool for co-simulation. The tool is created to expedite the process of co-simulation development for power and energy system studies targeting user groups of power engineers, energy consultants and grid operators. Instead of focusing on message encoding, transportation and synchronization, as many other co-simulation tools do, Cossembler emphasizes application-level functionalities which are of interest to the intended users (such as power flow studies, stability simulations, market simulations, etc.). Cossembler is a block modeling tool whose blocks reflect these main functionalities in power and energy sector. In this paper, we show the main characteristics of Cossembler architecture, discuss some of its advantages and disadvantages, and finally, show examples of its use.**

I. Introduction

Co-simulation stands for combined simulation and typically involves combination of two or more simulators or software tools. The purpose of this simulation modeling methodology is to extend the capabilities of existing tools and simulators in order to increase the scope of their modeling range and enlarge the set of potential applications that could be addressed using off-the-shelf products. The need for such greater capabilities is elicited by challenges of energy transition, from enabling flexibility in other forms of energy, to deployment of new technologies described by specialized models.

The main benefit of co-simulation is that it relies on simulation tools and models developed by domain experts, and hence, it expands on the existing knowledge and expertise. A plenty of specific examples exist. References [1], [2] use co-simulation for multi-domain modeling of EV charging infrastructure and hybrid energy grids, respectively. In [3], a summary of communication networks and power systems co-simulation is given. Another domain where co-simulation comes handy is for integration of converter-based technologies with the grid, as described in references on EMT and RMS co-simulation [4]. Finally, it is shown that it can be used for laboratory coupling [5], [6] and as an alternative to model exchange for dynamic stability studies [7].

Although co-simulation can provide many opportunities, yet many challenges exist until it can be easily and widely deployed to address energy-transition questions. As notable in the above examples, co-simulations are often custom designed targeting a particular application domain (ICT and power system simulation, multi-energy system simulation, etc.) and tailored for specific tools (e.g. PowerFactory, NS3, etc.). The main challenge to develop such co-simulation is that it requires high level of expertise by the modeler in simulation coupling concepts, such as simulator execution synchronization and message exchange. For a typical user from energy domain (such as power engineers, energy consultants, etc.) it is a steep learning curve to develop such intuition. Even if the modeler is proficient in co-simulation concepts, they still on average need quite some time to develop the co-simulation, including design process and programming. Finally, such custom solutions are often insufficiently flexible to the change in study objectives.

Another challenge in developing a co-simulation comes from the variety of simulators and their interfacing capabilities. Simulators sometimes have poorly documented API which is not always provided for all platforms. Further difficulties arise if the tools do not support all operating systems (e.g. PowerFactory is available only for Windows). Hence, even with a highly experience modeler, the co-simulation solutions end-up custom-tailored for a few tools of interest.

Cossembler simplifies these difficulties by enabling rapid prototyping of co-simulations. As the rapid prototyping tool, Cossembler is intended to have a gentle learning curve for its user. In addition, its modular architecture allows easy deployment for different types of energy co-simulations. In this paper, we review its architectural design and highlight its current capabilities.

The rest of the paper is organized as follows. Section II briefly reviews existing co-simulation architectures which inspired Cossembler and outlines the added value of Cossembler with respect to these architectures. In Section III we introduce Cossembler architecture and explain some of its main design choices. Section IV gives examples of Cossembler usage, while Section V provides a summary discussion.

II. Preliminaries

The approaches to co-simulation design can be roughly divided into two groups: those whose execution is governed by a co-simulation master, and those who operate without a dedicated master (i.e. masterless). A co-simulation master is a functional piece of code that handles synchronization among simulators and takes care of message propagation. In some instances, it is also responsible for scenario design, configuration and logging of simulation results.

In masterless architectures, one simulator assumes the role of a co-simulation master while other simulators synchronize with respect to it. Some examples are already mentioned references [1], [2]. Such approach typically makes development of simulator adapters easier, yet unstandardized. Simulator adapter is a piece of code that serves as an interface between the simulator API and the rest of the co-simulation. Since in this case a simulator acts as a master in addition to performing domain simulations, the scalability and flexibility of such approach are limited.

On the other hand, architectures based on a co-simulation master provide much higher scalability and flexibility. Domain agnostic co-simulation masters have been researched, starting with DEVS [8], primarily for use by the defense, automotive and airspace industries. Today, the energy simulation community is mostly centered around three solutions: HLA, Mosaik and FMU.

In order to show the added value of Cossembler, we briefly review these three approaches to co-simulation. In addition, each of these approaches served as an inspiration in respective ways when creating Cossembler.

HLA - High Level Architecture (HLA) is a general purpose architecture for distributed simulations developed with a particular intention to enable co-simulation in different application domains [9]. It originated as a natural extension of DEVS [8], and hence, it kept DEVS ideology regarding highly distributed implementation. HLA master, implemented as a Real-Time Infrastructure (RTI), offers an extensive set of services which are at the disposal to the user. This makes HLA highly flexible for various configurations, allowing the user precise control over the co-simulation execution.

Mosaik - Mosaik is a co-simulation framework particularly designed for simulations of cyber-physical energy systems and smart grids [10]. The special focus of Mosaik is on creation of large-scale system configurations which involve many smart grid controllers. Its minimalistic user interface makes its deployment easy when compared to HLA.

FMU - Functional Mockup Unit (FMU) is a standard firstly developed as Functional Mockup Interface (FMI) for model exchange for continuous time simulations [11]. In contrast to HLA and Mosaik who are agnostic to message content, FMU standard describes the semantics of the shared variables for co-simulation of time-domain models. Yet, FMU does not provide rules for synchronization or message exchange. Hence, FMU is complementary to HLA and Mosaik in its purpose. As it will be shown next, it is also complementary to Cossembler.

A. Added value of Cossembler

Figure 1 illustrates the main focus of each of the previously described co-simulation solutions. HLA provides an abundance of typical co-simulation services, while remaining application agnostic. Mosaik provides only essential co-simulation services while focusing instead on scenario scripting and simulator integration. FMU, as a standard for simulator interfacing, does not provide any co-simulation services, but instead introduces taxonomy for message exchange for continuous time simulations.

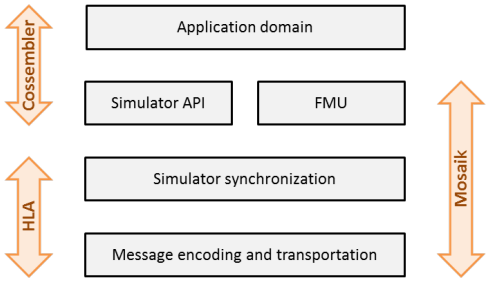

Fig. 1. Cossembler in relationship to other co-simulation architectures.

Cossembler can be viewed as a complement to HLA, Mosaik, and FMU, since its focus is placed on the exploration of the application domain. In its current version, Cossembler allows integration with HLA and FMU. Integration with Mosaik is planned as a part of future work.

III. Cossembler Architecture

The first premise of Cossembler architecture is a distributed ideology to co-simulation prototyping, as it is the case with HLA. Since the architectures such as HLA and Mosaik already exist, and are excellent in solving challenges of message propagation and synchronization, Cossembler directs attention to application domain co-simulation prototyping. In this regard, we focus on solving challenges that relate to coupling of energy domain models rather than simulators (see next sections for examples).

The second premise is the ease of use. Hence, Cossembler is developed as a block modeling tool in order to allow users who are less experienced with script programming to easily deploy it in their studies.

The third premise is deemphasize of performance. Although Cossembler does not intentionally sacrifice performance, it neither does actively try to improve it. As its name says, Cossembler is intended for rapid prototyping, and hence usability is given a higher priority to performance.

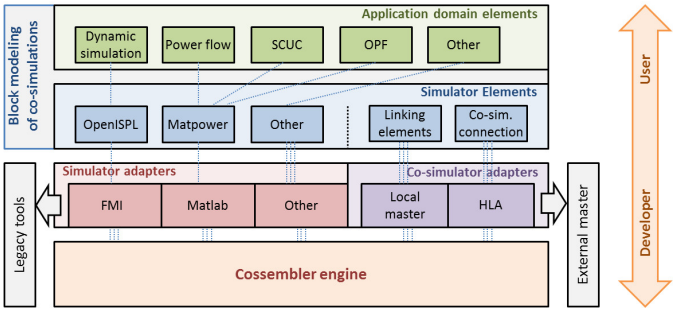

Fig. 2. Sketch of the Cossembler architecture including key components of the Power system operations application domain.

Figure 2 shows a conceptual illustration of Cossembler architecture. There are two basic categories of elements within Cossembler. The first one is Canvas and the second one is Element. Since Cossembler is object oriented and follows the rules of inheritance, Canvas is also an incarnation of Element.

Elements are used for two purposes. First, they are used to represent simulators. To accomplish this, a simulator adapter is created for each simulator as an Element of Cossembler.

Hence, we have a Matlab Element for example. Second purpose of Elements is to represent many basic data processing functionalities which could be needed when creating a co-simulation. For example, one might wish to extract only a part of the result returned by the simulator Element, or one might wish to plot the result or do other type of post-processing. These elements are called linking elements since their purpose is to link the simulator elements.

Canvas acts as a workspace in which prototyping takes place. A user creates Elements, adds them to a Canvas and connects them in desired configuration. Besides acting as a container for Elements, Canvas also makes sure that the data propagates from one Element to another in a form of a message. A more detailed representation of the canvas architecture is given in Figure 3.

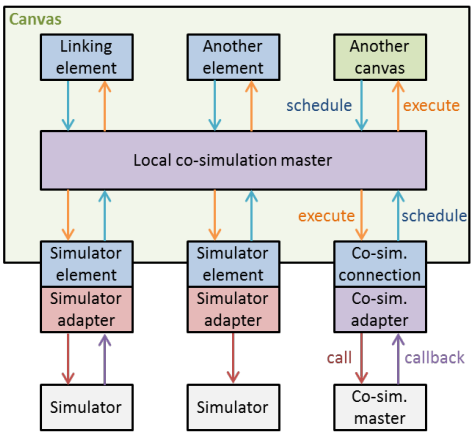

Fig. 3. Canvases and Elements in relation to each other.

It is important to emphasize that Cossembler can use its own local master or an external co-simulation master. In many simpler co-simulation prototypes, local master could be perfectly sufficient, while the external co-simulation master is particularly useful for large-scale co-simulations and for inter-process co-simulations, if and when these are needed.

With such minimalistic representation, the use of Cossembler becomes simple to master (see Figure 4 for a code snippet). In the next subsection, we discuss some of the specific design choices and characteristics of Cossembler. These are more important from developers' perspective than from users' perspective.

A. *Discussion of some important architectural choices*

Each Cossembler message contains a value that can either take one of the Python native types (Integer, Real, Boolean, String) or two types created for handling larger amounts of data: Vector which is implemented as a Python list, and Matrix which is implemented as a list of Python lists. Although there are more computationally efficient ways to represent vectors and matrices in Python (e.g. numpy), we intentionally opt in for lists since this choice eases the usage, avoids dependencies to third-party libraries, and discourages the user to implement computationally-heavy code using Cossembler environment. Cossembler is meant to ease interconnectivity of simulators and not to replace the functionality of these simulators. Since lists in Python are

```
from cossembler import Canvas
from cossembler import PowerFlow
from cossembler import DynamicSimulation

PFoptions = {...}
DSoptions = {...}

wrld = Canvas('world')
elemDS = DynamicSimulation("DSIPSL",DSoptions)
elemPF = PowerFlow("PF",PFoptions)

wrld.add_element(elemDS)
wrld.add_element(elemPF)

elemPF.connect(elemDS,'Pg','Pg')

wrld.start()
```

Fig. 4. Pseudo code showing the example of connecting two elements in Cossembler

copied by reference (and not by value), this design choice is perfectly suitable. The messages in Cossembler are time-stamped, and the user can choose if they wish to use this information.

```
elemLoop = ForLoop("FLoop", elemDS, 100)
```

Fig. 5. Example of creation of a ForLoop element. The element is initialized with another element whose execution is to be repeated N number of times (N=100 in this example).

Cossembler also contains Elements for easy creation of loops and conditions. These Elements are essential for control of co-simulation execution. For example, loops with conditions can be used for iterative engagement of one tool until a particular result is reached (see Figure 5 for a code sample for loop creation). Once such result is reached, the message is propagated to another tool and that tool is engaged in execution. Running an N-1 reliability assessment using an ordinary power flow tool is one such example. The power flow tool would be executed iteratively until all possible contingencies are checked. If the assessment returns unfavorable result, another tool can be engaged to compute new generation set-points.

To accomplish message propagation within a Canvas, each Canvas contains an Orchestrator that sets the rules for message propagation. This design choice was inspired by Ptolemy II [12]. Ptolemy II is a simulation tool developed for modeling of the heterogeneous mixture of models of computation. Although its intended application domain is embedded and computer architecture systems, its design is highly modular separating the simulation execution rules from its connectivity. Currently, Cossembler supports only one type of orchestrator. This is a Prioritized Discrete Event Scheduler (PDES) which aligns the execution of Elements according to their priority.

The nesting of Canvases is also supported, allowing more complex element creation and co-simulation design. One such element is the Dynamic simulation element, illustrated in Figure 6. This element uses an FMU adapter which engages Modelica OpenIPSL library to run dynamic (transient stability) simulations. The FMU adapter, implemented as a simulator element, is then interconnected with other linking elements and nested into two canvases in order to ensure accurate execution.

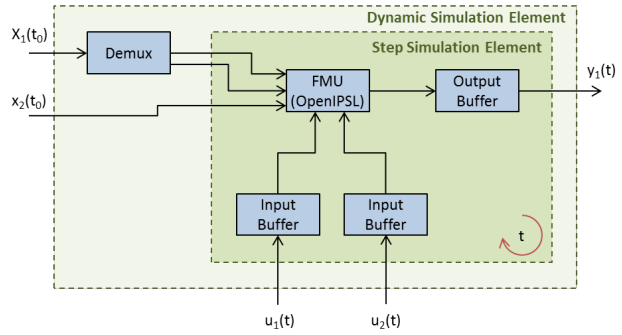

Fig. 6. Example of the dynamic simulation element that consists of other linking elements (buffers and demultiplexers) and the simulator adapter element (FMU adapter in this case running OpenIPSL elements in the background). Note that the input signals can be vectors (e.g. $X_1(t_0)$, $u_1(t)$, $u_2(t)$) and scalars (e.g. $X_2(t_0)$). The difference between $X_1(t_0)$ and $u_1(t)$ is that the former is a vector of variables at a single point in time, while the latter is a vector of a single variable over the period of dynamic simulation.

B. Simulators and co-simulation masters

Thus far, the adapters for Matlab (Matpower) and FMUs are available. FMU adapter is particularly handy since it provides inclusion of many power and energy related models. For example, IPSL-Tesla project has given birth to many power system models in Modelica modeling language, all of which are now available through the use of the FMU adapter. In addition, Modelica language is quite popular in other energy domains, such as heat, building and transportation modeling whose availability is particularly suitable for modeling of multi-energy systems.

On the side of co-simulation masters, an adapter for HLA is created and HLA is currently used as an external co-simulation master. In theory, any Run-Time Infrastructure (RTI) with Python API should be deployable through this adapter. However, the adapter has only been tested so far with CERTI RTI. CERTI is an open source RTI for HLA.

In addition to the external master, a local (internal) master is created as a PDES. This master is acts as the orchestrator of the main Canvas. To use such master, all simulators must be connected to one instance of Cossembler which would then act as the overall co-simulation master.

C. Application domains

Since the number and scope of elements of the tool could be quite large covering a wide and heterogeneous breadth, we establish the notion of application domains to confine elements to potential application domains. In other words, application domains are libraries of modeling blocks which could cover sufficient number of cases within a sub-area of power and energy field. The first application domain developed with Cossembler is the domain of Power System Operations. This domain contains the blocks typically used in power system operations, such as power flow, optimal power flow, SCUC, economic dispatch (with and without grid constraints), dynamic simulation (in particular transient stability simulation), and the blocks representing renewable generation including their uncertainty.

The block structure in Figure 7 is provided as a general template for use cases within power system operations domain. The blocks are arranged in its typical relative positioning within the power system operations cycle. This template can be changed by the user to match the exact case for illustration.

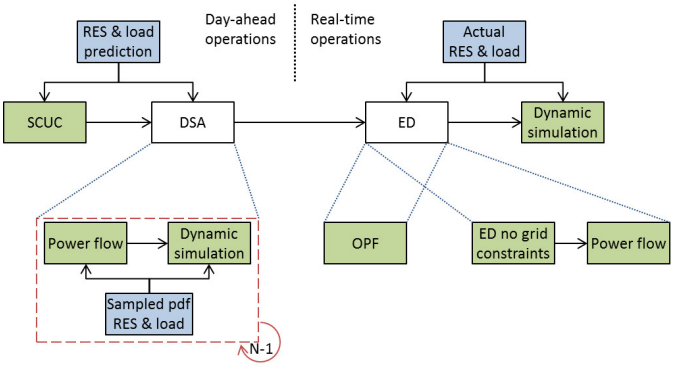

Fig. 7. Illustration of power system operations represented by Cossembler blocks. The day-ahead operations are represented using SCUC and DSA, while the real-time operations are represented using Economic Dispatch (ED) and grid dynamics simulations. In this sketch, DSA is composed of two blocks, while ED can be implemented in two common variants. Finally, both DSA and real-time operations are implemented in cycles to capture their repetitiveness.

The elements which use external simulators for execution are marked in green in Figure 7. These elements are at the moment provided by deploying Matpower (for steady-state computations) and Modelica OpenIPSL library (for dynamic simulations). Other popular legacy tools are to be added in the future. The blue blocks are created directly in Python as Cossembler elements. Note that an external co-simulation master is not used in the setup in Figure 7. The execution of co-simulation is controlled by the local master of the overarching Canvas.

IV. Use Cases

We illustrate the use of Cossembler in a use case of Dynamic Security Assessment (DSA). The first purpose of this use case is to show that Cossembler can be used to model existing or operational procedures under consideration. The blocks from the Power System Operations application domain are used in this use case.

The DSA model under simulation is illustrated in the left hand side of Figure 7. The power system is modeled as the IEEE 9 Bus system with an addition of one wind farm connected to the load Bus 9. In this use case, a grid operator runs a SCUC solver to create schedules for the next day. After the schedules have been created, a set of power flow and dynamic simulations are started assuming different levels of forecast errors and different realizations of wind fluctuations. The SCUC block engages Matpower for its computations while the dynamic simulation is performed using the model from the OpenIPSL library of Modelica. This model has been compiled into FMU by OpenModelica, and hence, Cossembler engages the model using the rules of the FMU standard.

The first part of analysis aims to investigate system frequency excursion for different realization of wind power. We use Monte Carlo simulation while sampling the probability density function of wind to generate different scenarios. The probability density function captures the distribution

of inter-hour wind fluctuations. If it is observed that the electrical grid frequency exceeds the allotted reliability band, the grid operator has to schedule sufficient level of frequency regulation reserves.

Figure 8 shows the histogram of frequency deviation obtained as the result of the simulation runs. If the band of 49.8Hz-50.2Hz is taken as the frequency regulation band in Continental Europe, we observe that the frequency excursion stays within the acceptable range.

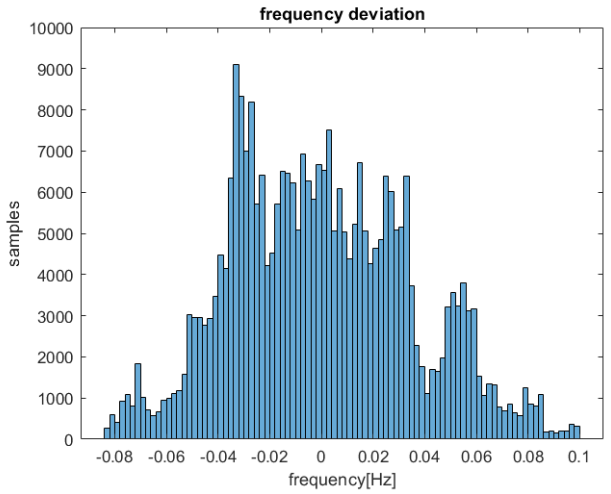

Fig. 8. Histogram of the frequency deviation for different realizations of wind power in the IEEE 9 bus system.

Next, we observe the dynamic behavior of the system frequency in response to the complete loss of wind generation. Figure 9 shows the path of grid frequency. We observe that the frequency nadir is 49.73Hz and that primary control restores the frequency to an acceptable range. Since our IEEE 9 bus model does not have secondary frequency control, the frequency remains wobbling around 49.91Hz.

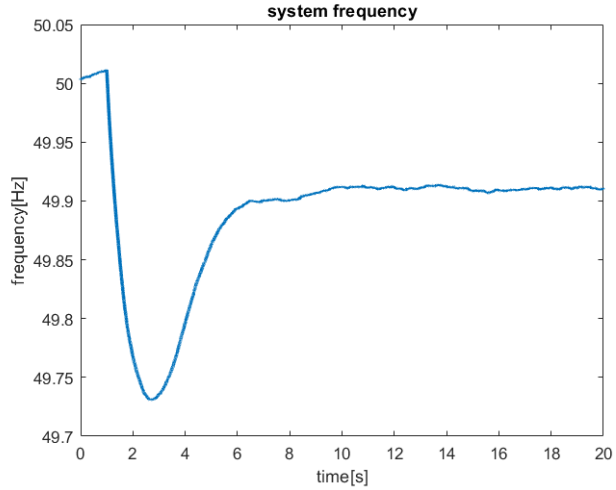

Fig. 9. Time response of grid frequency in the case of complete loss of wind generation.

V. Discussion

Use case flexibility - The use case presented in this paper is only an illustration. Other examples using these particular tools could also be explored, particularly in the direction of operations and market modeling (such as N-x reliability, co-optimization of day-ahead markets and frequency regulation markets, etc.). In the future, we plan to extend the set of models and simulators in the direction of multi-energy system modeling and in the direction of prosumer modeling (including EVs, DRES, heat pumps, home ESS, etc.).

Co-simulating two or more time-domain models - Although provided example does not model the coupling of the time-domain simulations, such coupling is possible since Cossembler propagates time-stamped messages. However, many practical challenges for such coupling still exist, ranging from stability and error propagation to scalability [13]. A proper choice of interface variables could improve any of these aspects [14]. Yet, further research and interface development is needed.

Real-time simulation - Cossembler was built with the focus on non-real-time co-simulations. The integration with real-time processes, which could be enabled through time-stamping as in [6], is a part of the future work.

Orchestrator variants - PDES is chosen as the default orchestrator implementation, due to its high flexibility and common usage (even many HLA RTIs use one). However, as seen on the example of Ptolomy II, there are many other variants for its implementation. Cossembler provides sufficient flexibility to implement different variations.

Configuration of simulation runs - This area requires particular attention going forward. Automatic configuration and scripting of simulation runs is increasingly important as the use cases grow in scale. As Cossembler has a neat programming structure, in terms of block models, such scripting should be possible to develop.

Interpretation and analysis of results - For now, Cossembler does not provide capabilities for post-processing of simulation results (besides saving of these results as CSV files). Stronger result analysis and visualization capabilities will be developed in the future.

VI. Conclusions

In this paper, we presented Cossembler - a rapid prototyping tool for co-simulations of power and energy systems. The main advantages of the tool are its high usability and potentially large set of application domains that could be covered, while the learning curve is rather light. Cossembler connects legacy tools and hence leverages on the existing domain expertise.

Yet, several aspects are to be improved in the future. First, block models for other application domains should be developed while including adapters for other legacy tools including commercial and open source software. Next, scripting and result post-processing capabilities will be improved to allow scalable deployment. Finally, further testing and example development will follow to ensure wide deployment of the tool.

References

[1] P. Palensky, E. Widl, M. Stifter, A. Elsheikh, *Modeling Intelligent Energy Systems: Co-Simulation Platform for Validating Flexible-Demand EV Charging Management*, IEEE Tran. on Smart Grid, vol. 4, no. 4, pp. 1939-1947, Dec. 2013.

[2] T. Jacobs et al., *Case Studies of Energy Grid Hybridization in a Northern European City*, IEEE Tran. on Sustainable Energy, doi: 10.1109/TSTE.2018.2867955

[3] IEEE Task Force on Interfacing Techniques for Simulation Tools et al., *Interfacing Power System and ICT Simulators: Challenges, State-of-the-Art, and Case Studies*, IEEE Tran. on Smart Grid, vol. 9, no. 1, pp. 14-24, Jan. 2018.

[4] V. Jalili-Marandi, V. Dinavahi, K. Strunz, J. A. Martinez, A. Ramirez, *Interfacing Techniques for Transient Stability and Electromagnetic Transient Programs IEEE Task Force on Interfacing Techniques for Simulation Tools*, IEEE Tran. on Power Delivery, vol. 24, no. 4, pp. 2385-2395, Oct. 2009.

[5] A. A. van der Meer, et al., *Cyber-Physical Energy Systems Modeling, Test Specification, and Co-Simulation Based Testing*, 2017 Workshop on Modeling and Simulation of Cyber-Physical Energy Systems (MSCPES), Pittsburgh, PA, 2017, pp. 1-6.

[6] M. Stevic et al., *Multi-site European framework for real-time co-simulation of power systems*, in IET Generation, Transmission & Distribution, vol. 11, no. 17, pp. 4126-4135, 2017.

[7] M. Cvetkovic, H. Krishnappa, C. Lopez, R. Bhandia, J. Rueda Torres, P. Palensky, *Co-simulation and Dynamic Model Exchange with Applications to Wind Projects*, Wind Integration Workshop, Berlin, Germany, October 2017.

[8] B. P. Zeigler, T. G. Kim, H. Praehofer, *Theory of Modeling and Simulation (2nd ed.)*, Academic Press, Inc., Orlando, FL, USA, 2000.

[9] IEEE standard for modeling and simulation high level architecture (HLA) - object model template (omt) specification (2010), IEEE Std 1516.2-2010, (Revision of IEEE Std 1516.2-2000)

[10] S. Rohjans, S. Lehnhoff, S. Schütte, S. Scherfke, S. Hussain, *mosaik: a modular platform for the evaluation of agent-based smart grid control*, IEEE/PES Innovative Smart Grid Technologies Europe, 2013.

[11] T. Blochwitz, et al., *The Functional Mockup Interface for Tool independent Exchange of Simulation Models*, In: 8th International Modelica Conference 2011, pp. 173-184 (2009).

[12] Claudius Ptolemaeus, Editor, *System Design, Modeling, and Simulation using Ptolemy II*, Published by ptolemy.org, 2014.

[13] C. D. Lopez, A. van der Meer, M. Cvetkovic, P. Palensky, *A variable-rate co-simulation environment for the dynamic analysis of multi-area power systems*, IEEE PowerTech, 2017.

[14] M. Cvetkovic, M. Ilic, *Interaction Variables for Distributed Numerical Integration of Nonlinear Power System Dynamics*, IEEE Allerton Conference on Communication, Control, and Computing, 2015.