Available online at www.sciencedirect.com

ScienceDirect

Procedia CIRP 00 (2018) 000–000

www.elsevier.com/locate/procedia

51st CIRP Conference on Manufacturing Systems

Dynamic Co-Simulation of Internet-of-Things-Components using a Multi-Agent-System

Tobias Jung*, Payal Shah, Michael Weyrich

University of Stuttgart, Institute of Industrial Automation and Software Engineering, Pfaffenwaldring 47, 70550 Stuttgart, Germany

* Corresponding author. Tel.: +49-711-685-67299; fax: +49-711-685-67302 *E-mail address:* tobias.jung@ias.uni-stuttgart.de

Abstract

The heterogeneity and dynamic of IoT-systems pose new challenges for their simulation, which can be met by a modular co-simulation. Therefore several existing co-simulation approaches are presented and evaluated. A new concept for a dynamic co-simulation of IoT-systems utilizing a multi-agent-system is presented, wherein each IoT-component is simulated in a separate simulation tool. Each separate simulation is represented by an agent, and therefore able to enter a running co-simulation dynamically during runtime, which allows for a "Plug-and-Simulate" behavior. The connection between agents and simulation tools is realized by an interface concept. The presented concept is evaluated by a prototypical implementation.

Peer-review under responsibility of the scientific committee of the 51st CIRP Conference on Manufacturing Systems.

Keywords: Internet of Things; co-simulation; agent-based modelling; Digital Twin

1. Introduction

Traditionally simulation is used only during development and virtual commissioning of production systems, but recently, with the introduction of the Digital Twin, the use of simulation during operation has gained importance and can be used for decision-making support, system optimization and predictive maintenance. With the introduction of cyber-physical systems, components are able to communicate with each other and thereby shape an Internet of Things (IoT) system. Such systems consist of heterogeneous devices of different domains, manufacturers, etc., which enter and leave the system during runtime. The concept of "Plug-and-Produce" was introduced to enable a seamless and effortless operation in such systems. The challenges of dynamic and heterogeneity also affect the simulation of such systems and hence the Digital Twin. Therefore, a similar concept, like "Plug-and-Simulate", is needed for the Digital Twin.

Reference [1] researched 15 approaches to simulate IoT-systems and concluded, that it will not be possible to simulate an IoT-system with every relevant aspect in a single simulation tool. Therefore a co-simulation is needed to simulate heterogeneous IoT-systems. To realize "Plug-and-Simulate" for dynamic, heterogeneous IoT-systems several thesis have to be met:

T1: It has to be possible for models of IoT-components to enter and leave the simulation during run-time, to simulate the dynamic behaviour of IoT-systems, where components also enter and leave during run-time [1].

T2: It also has to be possible to use various simulation tools to model the IoT-components. IoT-systems consist of heterogeneous components, which could be modelled with different simulation tools, so that each relevant aspect of the component can be considered [1].

T3: The simulation concept has to be applicable during every phase of the life-cycle of a product or system. Simulation will gain in importance and will be used during every phase of the life-cycle [2].

T4: The simulation concept has to be domain independent. With the introduction of IoT-technologies, systems are connected across different domains, so in the simulation of an IoT-system several domains have to be simulated.

Peer-review under responsibility of the scientific committee of the 51st CIRP Conference on Manufacturing Systems.

T5: It has to be possible to add intelligence and autonomy on top of the models of the IoT-components. The Digital Twin can be more than just an ultra-realistic simulation of a component or a system, it can also be used for decision-making support, system optimization and predictive maintenance. Therefore the simulation concept also has to take adding additional intelligence into account.

The rest of the paper is organized as follows: Section 2 introduces and compares existing co-simulation approaches with regard to the mentioned requirements. Section 3 presents a new concept for an agent-based co-simulation, for which a prototypical implementation is given in Section 4. In Section 5 the presented concept is evaluated with regard to the requirements and in the end a conclusion and an outlook is given.

2. Existing co-simulation approaches

Domain-independent and domain-specific co-simulation standards were researched as well as co-simulation approaches, which use other technologies to couple simulation tools.

2.1. Functional Mock-Up Interface (FMI)

The Functional Mock-Up Interface (FMI) provides beside a standard for co-simulation the possibility to exchange models between simulation tools [3]. A drawback of FMI is, that the used simulation tools have to support FMI, therefore it is not possible to include simulation tools in the co-simulation, which do not support FMI [4]. Additionally the communication is limited to discrete points in time, in between the tools run independently of each other. A master-algorithm coordinates the data exchange and synchronises the slave-simulations, where FMI allows for variable time steps between two synchronization steps. For a co-simulation each simulation tool has to be represented by a Functional Mock-up Unit (FMU), which implements the interface to the simulation tool [3].

In literature many examples exist for an implementation of a co-simulation with FMI, including [5], where the capabilities of FMI are discussed. The discussion shows, that the master-algorithm needs information of each slave-simulation, before starting the simulation, which is why FMI is not useful for a dynamic co-simulation with “Plug-and-Simulate”-capabilities.

2.2. High Level Architecture (HLA)

High Level Architecture (HLA) is an architecture developed by the United States Department of Defense for distributed and parallel simulation [6]. A co-simulation with HLA, called federation, consists of federates, the different simulators, and the Run-Time-Infrastructure (RTI), a central unit for coordinating the federates. In an Interface-Specification the interfaces between the federates and the RTI are defined and an Object-Model-Template specifies the information, which can be exchanged between the federates. Additionally a set of HLA-rules exist, which have to be met by simulator for being HLA-conform. The RTI can be seen as the simulation-master, responsible for the synchronisation of the federates [7]. Even though HLA allows for a dynamic entering of federates during run-time [8] for each federate a new Federation Agreement has to be written, which is domain- and use-case-specific. Several implementations of a RTI exist, both commercial and freely useable [9].

2.3. Co-Simulation with OPC UA

OPC UA is a machine-to-machine communication standard developed by the OPC-Foundation, which is service-oriented and enables a transmission of process data and their machine-readable description [10].

Reference [11] realises a co-simulation with OPC UA, where each simulator is connected via an interface to a generic adapter, which contains an OPC-UA-server and an OPC-UA-client. This adapter communicates via OPC UA with a central server, which also consists of an OPC-UA-server and an OPC-UA-client. Each simulator has to register itself at the central server and the first registered simulator the simulation-master, the following simulators are simulation-slaves. The master coordinates and synchronises the co-simulation. If the master leaves the co-simulation, another simulator takes over the role of the co-simulation-master.

2.4. Co-Simulation with OSGi

OSGi is a framework of the Open-Services-Gateway-initiative, which enables a dynamic component system, based on Java and was developed for the development of application, which consist of dynamic combinable and reusable components. To reduce complexity, in OSGi the implementation of the components is encapsulated to other components and the components interact via services with each other [12]. The components are represented by so-called Bundles, which can be loaded, removed, exchanged or updated by the framework during run-time Those Bundles can either be on a single computer or distributed over several computers [13].

The dynamic exchanging of Bundles allows for a dynamic co-simulation realized in [14]. Each simulation is represented by a Bundle, which is connected via a simulator-coupler to the simulation. The simulation-coupler utilizes OPC and also enables a synchronisation and date exchange between the simulations. It is also possible to integrate a model directly into an OSGi-Bundle. An additional Bundle is required for exchanging simulators during run-time, in which the states of the removed simulators are saved, so that a re-entry of those simulators is possible.

2.5. Domaine-specific approaches

Beside the presented domain-independent approaches many domain-specific approaches exist. One of the more

common standards is CAPE-OPEN (Computer-Aided Process Engineering) which is used to enable co-simulation in process industry [15]. However it is not possible for simulation tools to be coupled during run-time [8].

Besides CAPE-OPEN a huge variety of domain-specific co-simulation standards and approaches exist, like EPOCHS [16], Mosaik [17] and ADEVS [18], which are used for the simulation of electric power grids. As all of those standards and approaches are domain-specific, they do not meet the requirement of a domain-independent co-simulation.

2.6. Comparison of the approaches

Table 1 gives a comparison of the co-simulation approaches with regard to the requirements defined in Section 1. If a requirement is fully met by an approach, it is marked with "+", if it is only partially met it is marked with "0" and if not met at all it is marked with "-".

Table 1. Comparison of the co-simulation approaches

Co-simulation approach	T1	T2	T3	T4	T5
FMI	-	0	+	0	-
HLA	+	-	+	0	-
OPC UA	+	0	+	+	0
OSGi	+	0	+	+	0
Domain-specific	0	-	+	-	-

As can be seen in Table 1, no approach fully realizes every thesis. Only for HLA, OPC UA and OSGi it is possible to add new models during run-time. With FMI, OPC UA and OSGi it is possible to use different simulation tools, but for every simulation tool an interface has to be implemented, with HLA it is not possible to add new simulation tools during run-time, as for a new simulation tool a new Federation Agreement has to be written. For the domain-specific standards only domain-specific simulation tools are useable. Every introduced approach is usable during every phase of the life-cycle, but only OPC UA and OSGi are completely domain-independent. FMI and HLA support a huge variety but are not completely domain-independent, as FMI is mainly used in automotive and HLA in the military domain. No approach has the capability to add intelligence on top of the models, especially FMI, HLA and the domain-specific approaches, which are all co-simulation standards, would need major modifications for adding additional intelligence on top of the models. With OPC UA and OSGi it would be possible to add intelligence on top of the models but here also modifications of the concept would be necessary.

3. Dynamic, agent-based Co-Simulation

As none of the presented co-simulation approaches realizes more than 3 of the defined thesis, a new approach for a dynamic co-simulation is needed.

3.1. Multi-agent-system

The concept of multi-agent-systems for coupling the simulation tools was chosen, as software agents are capable of entering and leaving a multi-agent-system during run-time, are domain-independent, have no restrictions in any phase of the life-cycle and have the capability of adding intelligence on top of the models [19]. The concept of using agents to represent simulations of IoT-components was already presented in [1]. Each IoT-component is simulated in a separate simulation, possibly also in different simulation tools, and each simulation is connected to and represented by an agent, see Fig. 1. By simulating each IoT-component in a separate simulation and connecting them to agents, it is possible to exchange the models during run-time, as the agents are able to enter and leave the multi-agent-system during run-time, just like the IoT-components enter and leave the IoT-system. The interaction between the models are forwarded by the agents via communication between the agents. For example if the model of a heating unit requests the temperature value of the model of a temperature sensor, the agent connected to the model of the heating unit forwards this request to every other agent, which forward the request to their respective models. The models then decide themselves, whether the received message is useful and then reply respectively. To enable the connection between the agents and their respective simulation tools a concept for an interface has to be developed.

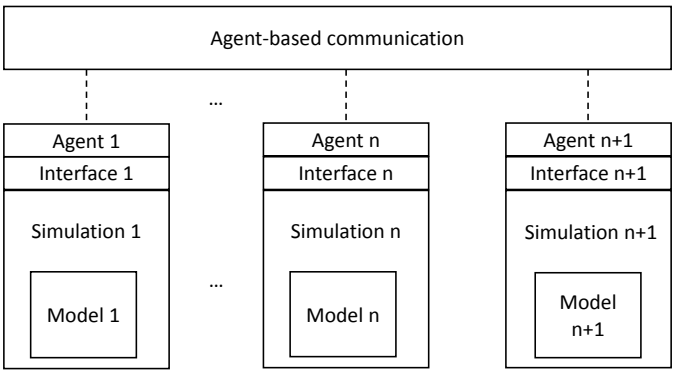

Fig. 1. Multi-agent-based co-simulation.

3.2. Interface between agents and simulation tools

To enable a connection between the agents and the simulation tools an interface is required. In IoT-systems, the components communicate with each other, and therefore the models also have to communicate with each other. Hence, the models have to be able to send messages to other models, which has to be enabled by the interface. This part is named communication interface and can be seen in Fig. 2.

Additionally IoT-components have a process-oriented interaction, meaning, that besides communication, they also can interact physically with each other. If a good requests a forklift to be transported, it is an interaction by communication and if the forklift transports the good, it is a physical interaction, further called process-oriented

interaction. Therefore in addition to the communication interface between agent and model, an interface for the process-oriented interaction is required, see Fig. 2.

Another required interface between the agents and the models is an interface for the synchronisation of the simulations (Fig. 2.). Problems in the simulation of the whole IoT-system can occur, if one simulation of an IoT-component runs faster than another simulation of another component, as messages are not delivered at the right time.

All of those interfaces have a connection to an agent and to a model. The connections to the agents will always be the same, as the agents do not differ, only the connection to the models have to be implemented individually, as each simulation tool has different interfaces. Therefore every interface is divided into a generic part, which is connected to the agent and is the same for every agent and a specific part, which is connected to the simulation tool. The specific part has to be developed for each simulation tool and can be used for every model in the respective simulation tool, as can be seen in Fig. 2.

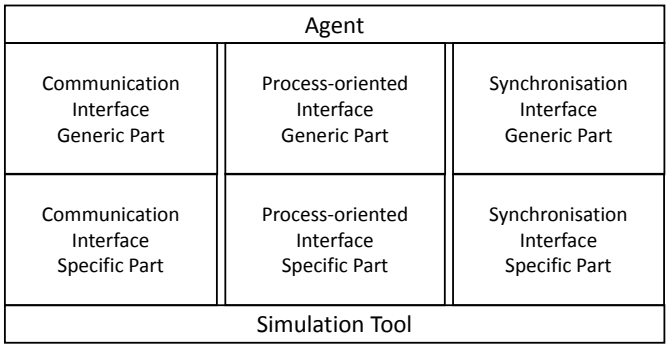

Fig. 2. Agent-simulation-tool-interface.

3.3. Reduction of message traffic

So far, we have seen that each model is able to communicate with every other model in the system via agents. However, this is quite inefficient in a sense that there would be a lot of unnecessary messages being sent. In reality, not all models would want to interact with every other model in the system. Therefore, instead of sending messages to all the models, if we could send the message to group of relevant models then we can significantly reduce the message traffic in the agent system. The concept of model type identification was thus developed. A generic way to group the models (IoT-components simulated) is to group them by the communication protocols the IoT-components and therefore the models support. For example, all the models communicating via “Bluetooth” or “Wi-Fi” communication protocol will belong to one group. Models will now send messages only to other models of the same type (in the same group) and not every other model present in the system. The communication interface forwards the message it receives from the model to the agent, but the agent doesn’t receive any information as to which communication protocol this message belongs to and thus cannot decide which type of agents it should search in order to forward the message. To solve this problem the agent, on its creation, needs to create as many communication interfaces as the number of communication protocols the model supports. Thus, the agent then creates multiple generic and specific components, as seen in grey in Fig. 3.

However just grouping of agents will not help much in the worst case scenarios, where a message sent by a model was received by many models but was useful for only one or none of them. After getting a message to be forwarded to other agents, the sender agent does not semantically analyse the message and it has no knowledge about the usefulness of the message to the recipient agent. However, it can learn that from the recipient agent. The idea is that the sender agent learns which type of message is not useful for which recipient agent. The recipient agent can get this information from its model. Therefore, if the receiver agent gives a feedback to the sender agent which types of messages were not useful for it, the sender agent won’t send them to it anymore, further helping in reducing the message traffic in the agent system. For this a back-channel is needed in the interface, as can be seen in grey in Fig. 3.

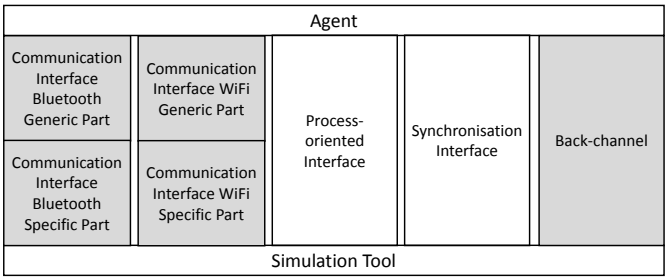

Fig. 3. Message reduction.

4. Prototype Description

This section explains the scenario considered for the prototype and then the implementation details of the prototype.

4.1. Scenario

A smart warehouse scenario was implemented to demonstrate the concept in which the goods, storage rack, sensors and the forklift communicate with one another and take decisions for the storage of the goods. The warehouse has a storage rack which has a temperature sensor installed on it to measure its temperature. This is because only a predefined temperature range is suitable for the storage of goods.

When the goods enter the warehouse, they send a request to the storage rack if they can be stored. Upon getting the request the storage rack gets the current temperature value from the sensor and checks if the value is suitable for the good storage. If yes, then it sends a signal to the forklift to place the goods in it otherwise it tells the goods to wait. After placing the goods in the rack, the forklift sends an acknowledgement signal to the goods which meant that the storage is completed.

The temperature sensor and the storage rack communicated with each other using Bluetooth while the goods, storage rack and the forklift communicated over WIFI.

4.2. Implementation

Simulation tools namely, MATLAB Simulink and OpenModelica were used to simulate the scenario explained above. Goods and storage rack were simulated in OpenModelica and temperature sensor and forklift were modelled in MATLAB Simulink. Jadex, an agent framework, was used for facilitating dynamic co-simulation of the developed models. For the user to select the models and plug or unplug them at run-time a graphical user interface (GUI) was developed in the prototype. User could select the simulation tool and the model to be connected to plug it.

On plugging-in the model, a BDI agent is created for that model. The agent in turn creates communication interface(s) as well as a back-channel for the model. If the input of a model was not acceptable, an exception would be thrown by the model which is handled by the back-channel of that model. The specific component of the communication interface and the back-channel then establish a connection with the simulation tool (i.e. the model).

This connection had to be handled differently for MATLAB Simulink and OpenModelica. Since MATLAB Simulink allows for only one connection per instance we had to:

- Run one instance of Simulink per model.
- Include the functionality of back-channel also in the specific component of the communication interface establishing the connection with the model and generic component of the communication interface.

On the other hand, OpenModelica allows for multiple connections per instance. We were able to:

- Run a single instance of OpenModelica for all the models simulated in it.
- Have separate connection for specific component of the communication interface and the back-channel for each model simulated in OpenModelica.

On unplugging the model all the related components were destroyed. Agents can be created and destructed dynamically using the component management service, a central service offered by Jadex platform.

Jadex uses service component architecture. Therefore, the agents interact with one another using Services. Agents ‘provide’ and ‘require’ services. The provided service interface and the required service interface are used to make service calls i.e. to communicate with one another. In the prototype agents provided (and required) services based on the communication type the model supports to have model type identification in the system as described in Section 3.3 For example, if temperature sensor supports the Bluetooth communication protocol then its agent would provide and require ’Bluetooth’ service. So whenever an agent providing ‘Bluetooth’ service wants to send a message, it will search for agents in the system providing ‘Bluetooth’ service and send them the message.

If a model supports multiple communication protocols, then its agent would provide (and require) multiple services. In case of MATLAB Simulink, since it permits only one connection per instance, we can have multiple generic communication interface and one specific communication interface.

All the agents in the system also provided a common service used to give feedback. To realize the reduction in message traffic as discussed in section 3.3, the recipient agent would use this service provided by the sender agent to give feedback about the messages which are not useful. Also, note that no feedback is given when the messages are useful, because that is simply going to add on the message traffic which we are trying to reduce.

The processing and use of the feedback is implemented in a simple way in the first version of our prototype. The feedback i.e. the message and the agent name to which the message was not useful is maintained in a list at the sender agent. The list will gradually grow as the agent learns from feedback by agents it has previously contacted. These messages stored in the list will serve as message patterns rejected by the agents. Before it sends a message to another agent, it will check in the list if such a message pattern exists for the receiver agent. If yes, then it will not send the message and proceed to check for the next relevant agent, else it will send. When such a message pattern is present in the list, it means that the sender agent has already gotten a feedback saying that such a message pattern is not useful for the receiver agent under consideration. The pattern matching of the message to be sent and the messages maintained by the agent in the list is done by checking the Levenshtein distance between the two strings. If the Levenshtein distance between the two strings is less, then the two strings under consideration are of the similar pattern as one of the rejected strings, else the string (message) to be sent is of a different pattern than them.

The way in which the message is passed and fetched from the simulation tool differs in case of MATLAB Simulink and OpenModelica. The specific communication interface handles those differences. For example, one can pass and fetch messages from MATLAB Simulink using MATLAB commands (to fetch the output, one can poll the output port). While for OpenModelica, polling is not possible, as the outputs are written to a text file which then has to be read from the specific communication interface. Also, a MATLAB Simulink instance can be shared and accessed from Java using the com.mathworks.engine java class. Whereas for OpenModelica, whenever the application starts, a CORBA server is started and the specific communication interface being the java client can connect to the server using the omc_java_api library.

5. Evaluation

Fig. 4 shows a screenshot of the prototype. The models of the temperature sensor and the forklift were executed in MATLAB Simulink, the models of the goods and the storage rack in OpenModelica and all of the models were connected via the multi-agent-system. The prototype, was tested to make sure the presented concept realizes all the defined thesis.

- T1: We were able to plug and unplug models during run-time.
- T2: We used MATLAB Simulink and OpenModelica to implement the models and therefore different simulation tools.
- T3 and T4: As the presented concept is neither domain-specific (it is also possible to simulate different domains than logistic scenarios) nor life-phase-specific (it is possible to simulate a system during all phases of the lifecycle), both requirements are met.
- T5: We presented the reduction of message traffic by recognizing patterns in the send messages and therefore added intelligence on top of the models.

Thus, the prototype has satisfactorily attempted to solve the challenges of heterogeneity and dynamicity in the simulation of IoT systems.

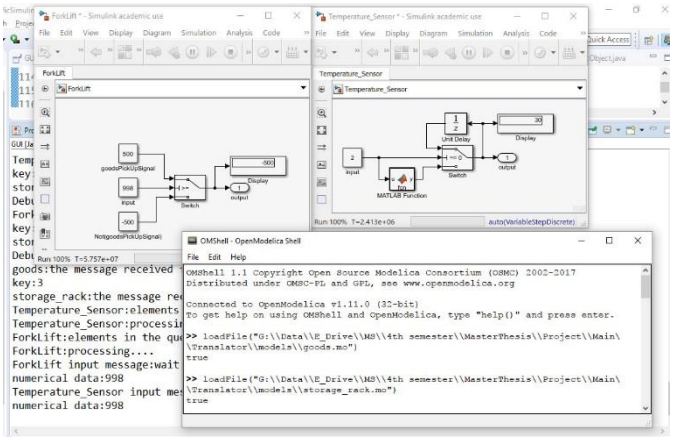

Fig. 4. Prototype with MATLAB and OpenModelica

6. Conclusion and Future Work

The presented concept of a dynamic co-simulation of IoT-systems utilizing a multi-agent-system allows for an entering and leaving of heterogeneous simulation tools. By using the capabilities of the software agents it is possible to add intelligence to the models of the IoT-components, shown at the example of message traffic reduction. As the reduction of messages is only an optimization of the simulation and not of the simulated system, a task for future work will be, to add intelligence to the agents, which benefits the simulated system, such as support for self-configuration of the components. The concept of the message reduction itself also can be optimized, as it could be possible, that through changes in the system a rejected message type gets relevant for the receiving model again. The concept of the synchronization and the process oriented interaction of the models also has to be detailed and implemented.

References

[1] Jung, T.; Jazdi, N.; Weyrich, M.: A Survey on Dynamic Simulation of Automation Systems and Components in the Internet of Things. 22nd IEEE ETFA, Limassol, Cyprus, 2017.
[2] Mathias Oppelt, Mike Barth, and Leon Urbas. 2015. The Role of Simulation within the Life-Cycle of a Process Plant - Results of a global online survey, 2015.
[3] Blockwitz, T., Otter, M., Akesson, J., Arnold, M., Clauss, C., Elmqvist, H., Friedrich, M., Junghanns, A., Mauss, J., Neumerkel, D., Olsson, H., and Viel, A. 2012. Functional Mockup Interface 2.0: The Standard for Tool independent Exchange of Simulation Models. In Linköping Electronic Conference Proceedings. Linköping University Electronic Press, 173–184.
[4] Bertsch, C., Ahle, E., and Schulmeister, U. 2014. The Functional Mockup Interface - seen from an industrial perspective. In Linköping Electronic Conference Proceedings. Linköping University Electronic Press, 27–33.
[5] Bastian, J., Clauß, C., Wolf, S., and Schneider, P. 2011. Master for Co-Simulation Using FMI. In . Linköping Electronic Conference Proceedings. Linköping University Electronic Press, 115–120.
[6] Fujimoto, R. M. op. 2000. Parallel and distributed simulation systems. Wiley series on parallel and distributed computing. John Wiley & Sons.
[7] 2010. IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)-- Framework and Rules. IEEE, Piscataway, NJ, USA.
[8] Oppelt, M., Wolf, G., and Urbas, L. 2014. Capability-analysis of co-simulation approach-es for process industries. In ETFA'2014. 19th IEEE ETFA : September 16-19, 2014 : Barcelona, Spain. IEEE,
[9] Möller, B. 2012. The HLA Tutorial: A practical guide for developing distributed simulations, 2012.
[10] OPC Foundation. 2008. OPC-UA Specification.
[11] Hensel, S., Graube, M., Urbas, L., Heinzerling, T., and Oppelt, M. 2016. Co-simulation with OPC UA. In Proceedings, 2016 IEEE 14th INDIN. Palais des Congrès du Futuroscope, Futuroscope - Poitiers, France, 19-21 July, 2016. IEEE, Piscataway, NJ, 20–25.
[12] The OSGi Alliance. 2014. OSGi Core.
[13] McAffer, J., VanderLei, P., and Archer, S. J. op. 2010. OSGi and Equinox. Creating highly modular Java systems. The eclipse series. Addison-Wesley, Upper Saddle River.
[14] Oppelt, M., Drumm, O., Lutz, B., and Gerrit Wolf Siemens, A. G. 2013. Approach for in-tegrated simulation based on plant engineering data. In ETFA 2013. September 10-13, 2013, Cagliari, Italy. IEEE, Piscataway.
[15] Peshev, D. and Livingston, A. G. 2013. OSN Designer, a tool for predicting organic solvent nanofiltration technology performance using Aspen One, MATLAB and CAPE OPEN. Chemical Engineering Science.
[16] Hopkinson, K., Wang, X., Giovanini, R., Thorp, J., Birman, K., and Coury, D. 2006. EPOCHS. A Platform for Agent-Based Electric Power and Communication Simulation Built From Commercial Off-the-Shelf Components. IEEE Trans. Power Syst. 21, 2, 548–558.
[17] Schutte, S., Scherfke, S., and Troschel, M. 2011. Mosaik: A framework for modular simulation of active components in Smart Grids. In SGMS 2011. 2011 IEEE First Inter-national Workshop on Smart Grid Modeling and Simulation : [17 Oct. 2011, Brussels, Belgium]. IEEE.
[18] Nutaro, J., Kuruganti, P. T., Miller, L., Mullen, S., and Shankar, M. 2007. Integrated Hy-brid-Simulation of Electric Power and Communications Systems. In 2007 IEEE Power Engineering Society General Meeting, Tampa, FL 24-28 June. IEEE Xplore, Pisca-taway, N.J.
[19] Weiss, G. 2001, ©1999. Multiagent systems. A modern approach to distributed artificial intelligence. MIT Press, Cambridge.