International Journal of Modeling, Simulation,
and Scientific Computing
Vol. 8, No. 4 (2017) 1750055 (21 pages)

DOI: 10.1142/S1793962317500556

World Scientific
www.worldscientific.com

Hybrid simulation using implicit solver coupling with HLA and FMI

Muhammad Usman Awais*
National University of Computing and Emerging Science, Lahore, Pakistan
usman.awais@hotmail.com

Milos Cvetkovic and Peter Palensky
Department of Electrical Engineering, Mathematics and Computer Science
TU Delft Building 36, Room B36-LB 03.230
Mekelweg 4, 2628, CD Delft, Netherlands

Received 2 September 2016
Accepted 25 April 2017
Published 1 June 2017

Hybrid systems such as Cyber Physical System (CPS) are becoming increasingly popular, mainly due to the involvement of information technology in different aspects of life. For analysis and verification of hybrid system models, simulation is used extensively. As parts of a common hybrid system may belong to different domains of study, it is sometimes beneficial to use specialized simulation packages (SPs) for each domain. In this case, parts of a system are simulated in different SPs. The idea may seem simple, but coupling more than one simulation component presents challenges related to numerical stability. The presented article suggests an implicit solver coupling technique enhanced to facilitate simulation of hybrid models using multiple simulation components. The technique is developed using two of the most popular simulation interoperability standards, namely, the High Level Architecture and the Functional Mock-up Interface. By using these standards, the developed algorithm will be useful for a large number of practitioners and researchers. The developed algorithm is described using a generic distributed computation model, which makes it reproducible even without using the standards. For the verification of results, the algorithm is tested on two case studies. The results are compared to a monolithic simulator and the proximity of results initiates the validity of the developed algorithm.

Keywords: Distributed simulation algorithms; co-simulation; distributed simulation; parallel simulation; continuous simulation; simulation interoperability; DEVS; OpenModelica; Modelica.

*Corresponding author.

1. Introduction

Modeling and simulation have become cornerstones for verification of mathematical phenomenon found in nature. Development of simulation packages (SPs) follows the same pattern of fragmentation which is usual to the study of natural phenomena. Physicists, chemists, mathematicians and mechanical engineers have developed simulation packages suitable for their own domains. Most popular SPs either focus on one domain of problems or on a specific type of system specification. The differences in approach and domain imply algorithmic and mathematical challenges, when coupling more than one simulation component.

Hybrid systems are prevalent in every day life. For example, Cyber Physical System (CPS) is a class of systems where a physical system is augmented by information technology. When analyzing such systems, mostly cyber and physical parts are simulated using separate SPs. For example, the simulation of an optimized model-predictive control of an urban power system is likely to have a power grid simulator along with an ICT simulator.

Using already verified models to construct a simulation of a larger system is very beneficial. However, the benefits cannot be fully utilized if the developed technique itself requires modification of the SPs. Intuitively, the most useful technique could only be the one which follows already developed standards of simulation interoperability. Using standards can make the technique acceptable and accessible in both industry and research.

This paper suggests a distributed algorithm which can be used to couple more than one simulation component. The coupled simulation components are allowed to be hybrid in nature, meaning some simulation components can contain discrete events and others be purely continuous. The algorithm is implemented using the High Level Architecture (HLA) and the Functional Mock-up Interface (FMI), in order to make it operable for a wide range of SPs (for details on HLA and FMI, see supplemental material Sec. S-1). According to the knowledge of the authors, there has been no algorithm presented before which allows the hybrid simulation of more than two independent simulation components. The word independent means that each simulation component has its own solver. Previously, an implicit solver coupling algorithm was proposed in Ref. 1, which only allowed continuous simulation components to be coupled, no discrete event component was allowed. In other efforts, only one continuous time and one discrete event-based simulation components were coupled.[2] The presented algorithm does not put any restriction on the number or types of the simulation components. Any number of continuous time and discrete event-based simulation components can be used.

In the next section, first the state of the art with respect to the presented work is discussed. Later, a brief overview of only the relevant parts of the HLA and the FMI standards are discussed. After describing the distributed computation model applicable to the algorithm, the algorithm itself is discussed in detail. Although the algorithm is developed using the HLA and the FMI, it is described in a generic way,

so it can be implemented without even knowing the details of these standards. For verification of algorithm, results from two case studies are presented. Finally, the paper ends with conclusions.

2. Related Work

Different attempts to tackle the problem of simulation interoperability have been made in the past. Some addressed the problem from a mathematical point of view, while others devised standards to interface the components. Here we present a brief overview of the already proposed solutions with their shortcomings.

On the mathematical side, there are quite a few scientists who suggested formal methods for coupling simulation components. To name a few, Tseng and Hulbert suggested a simulation coupling technique for multibody simulation.[3] Kübler and Schiehlen, in their paper,[4] discussed different mathematical issues related to simulator coupling. Recently, Schweizer *et al.* have discussed issues like numerical stability and step size control of implicit,[5] explicit[6] and semi-implicit[7] coupling techniques. None of the articles mentioned above discusses the interfacing portion of the problem.

Discrete EVent Specification (DEVS) is a recognized modeling and simulation technique. DEVS provides a systematic way of converting a continuous simulation into a discrete event simulation,[8] in effect, the DEVS professes that any hybrid system can be simulated using DEVS. Zeiglar *et al.*[9] have mentioned the challenges faced while simulating DEVS specific simulation components over the HLA, but the approach used for some solutions is less than correct, as argued in Ref. 10. Besides other mathematical issues,[11] the biggest drawback of DEVS is the difficulty in coupling arbitrary SPs with the DEVS-specific simulation package. Only those SPs are favorable for coupling that produce DEVS-specific simulation components. The numerical stability of different solvers used with DEVS paradigm is discussed in Ref. 11.

Ptolemy II[12] is another attempt to make development of heterogeneous simulations easier and faster. There have been efforts to interface it with the FMI compliant components.[13] By adding support for FMI and enabling FMUs to be executed in different processes, Ptolemy II may be considered as a tool which supports distributed simulation. The biggest disadvantage of using Ptolemy II is its lack of flexibility. Although the methods for implementing a new solver or incorporating a new "director" are well documented, yet there are some limitations imposed by the Ptolemy II kernel. For example, how it treats the events in the queue, and when and how it processes them. The behavior can only be changed by changing the kernel of Ptolemy II.

Due to the introduction of ICT into the management of power grids, it has become vital to simulate models of power grids in conjunction with a communication technology. In the recent past, there have been quite a number of efforts to couple ICT network simulators with power system simulators[14] to analyze models of cyber

physical energy systems, or in other words smart grids.[15] There are a few limitations in the proposed simulation systems, as discussed below.

Most smart grid simulators do not try to couple more than one continuous system. Often, only one continuous power system simulator is coupled with another discrete event-based network simulator. In this case, the model does not have any "algebraic" relationship among the simulation components, which makes things much easier and manageable, but this is a limitation too. For example, consider the scenario of a large city having a thermal supply and a power supply connected to each other. To simulate such a scenario, a thermal energy simulator has to be included in the coupled simulation of power grid and the ICT infrastructure. Current smart grid simulators will not be able to simulate such a scenario due to their incapacity of coupling more than one continuous simulator.

Many of smart grid simulators are built on implicit assumptions about the underlying mechanism of distributed simulation. For example, many smart grid simulators seem to assume that the time synchronization is a solved problem, so they do not mention how they addressed the problem. It is not mentioned whether there is any level of parallelism involved or not. If yes, then how the problems of numerical stability, data sharing and time synchronization were addressed?

In many solutions, it is not mentioned clearly how the simulation time is progressed, in a fixed time stepped fashion, with the help of discrete events, or with a step size control mechanism? Some power system SPs used in aforementioned solutions only support fixed time stepped execution, which is problematic when a discrete event does not occur right at the boundary of a fixed time step.[11]

3. Model of Distributed Computation

To explain the forthcoming algorithm, it is necessary to describe the relevant distributed computation model. As mentioned in Ref. 16, a system having different processes connected via a networking medium is best described as an "*asynchronous message passing system*". A "*message passing system*" is a system where different processes communicate with each other with the help of messages. This is in contrast to a "*shared memory system*", where processes do not send messages, rather they communicate using a shared memory space. One of the benefits of a message passing system is their lesser dependency on concurrency control structures, as compared to shared memory systems. The model of the HLA RTI is close to the representation of an asynchronous message passing system. The only difference with a traditional asynchronous message passing system described in textbooks[16] like the "Time Stamped Ordered delivery" or TSO delivery. In an asynchronous model, there is no relation between the order of messages sent and received. Contrarily, in TSO delivery, there is always a "time stamp" on each and every message sent, and the same order is retained when the messages are received on the receiver end. The messages sent with the same time stamp need not follow any order.

A system is considered to be an "asynchronous message passing system" when there is no fixed upper bound on the messages to reach at the destination, or there is no fixed time limit on how much time should be spent on any step. Due to these conditions, if there are two messages m_1 and m_2 sent from processors p_1 and p_2 to a destination process p_d, then there is no guarantee which message will reach the destination first. In the proposed modified model with TSO delivery, m_1 must reach before m_2, if it has a time stamp less than m_2. If both were sent on the same time stamp, then there is no order enforced.

For a formal description of the algorithm, the RTI is considered as a separate process rather than merely a simulation message bus. The process for the RTI is referred to as p_{rti}. Describing the complete procedure of the RTI is far beyond the scope of this paper, but it is useful to have a small description of the p_{rti} for the algorithm under consideration. The description contains the information how the RTI is assumed to react on certain messages under consideration, according to the HLA standard, without going into the details of their implementation.

The RTI follows two different models of communication which correspond to the two main communication protocols namely, TCP and UDP. When "reliable" mode of communication is desired, the TCP protocol is used, while in case of the "best-effort" mode, the UDP protocol is used.[17] Here it is assumed that the "reliable" mode is being used for the implementation of the described algorithm.

Since the proposed algorithm assumes a message passing system, it must be kept in mind that all the processes are executing in separate binaries, remote or local, and they take actions on receiving a message similar to an interrupt driven system. As the distributed processors assume an infinite execution of the processes,[16] a special state **terminate** is introduced in order to represent the state where they do not respond to any new messages. Any distributed process acts in a specified manner on receiving a message. In the algorithm, a message is identified by ⟨Message⟩. A message may contain different parameters, like the simulation time associated to the message, and the values of some attributes. Sometimes, all these values are not mentioned in the algorithm to avoid cluttering of information. In such a case, the text and the supplemental material contain the required explanation separately.

The algorithm is developed using a master-slave configuration. Figure 2 shows how the master is connected to all slaves via the RTI. The RTI works as a medium for data and time synchronization, but the master is the real orchestrator. The master drives all the slaves, guides them through different states, and commands them to reach a common goal. Slaves, on the other hand, are the work horses. Each one of them contains an FMU inside, so they are called FMU-Federates. The master, on the contrary, does not contain any FMU, it just executes the algorithm. Each slave fulfills the commands sent from the master, some of these commands require to take action on the FMU, like setting or getting state variables and input variables, and setting or getting time of the FMU. The internal integration of each FMU takes place at the slave level. At certain points in time (called as communication points),

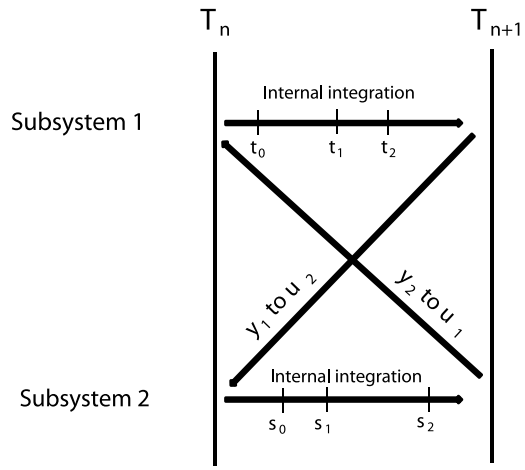


Fig. 1. Figure shows how the master, slaves and the RTI are connected together. The dotted lines for responses show that for some commands master may not be expecting a response.

the slaves share the data according to the directions imposed by the algorithm executed at the master level.

4. Hybrid Simulation Using Implicit Solver Coupling

The waveform relaxation method uses implicit solver coupling. It was first proposed in Ref. 1. The idea is to use repetitive evaluation and sharing of coupling state variables, until they converge to one set of values. For each time step, the loop to achieve convergence is executed. Figure 1 shows the concept of Waveform Relaxation (WR) scheme. The WR algorithm proposed in Ref. 1 is not capable of handling discrete events, it is only proposed for continuous systems. The algorithm proposed in the presented article introduces how discrete event simulators can be coupled with continuous time simulators. There is no boundary on the number of continuous time or discrete event simulators. Moreover, the presented technique uses the FMI and the HLA standards to make it applicable for a wide range of real life problems.

4.1. *Algorithm description*

Referring back to Fig. 2, there are three main components needed to implement the proposed scheme, according to the distributed computation model discussed in Sec. 3. A master, an RTI, and different FMU-Federates act as slaves. The relevant code for all three components is included in supplemental material, Section S-2. Algorithm S.A-1 contains the implementation details for the master, Algorithm S.A-2 is the code for slave, and Algorithm S.A-3 shows how the RTI behaves according to HLA specifications. The outputs of the FMUs loaded in slave

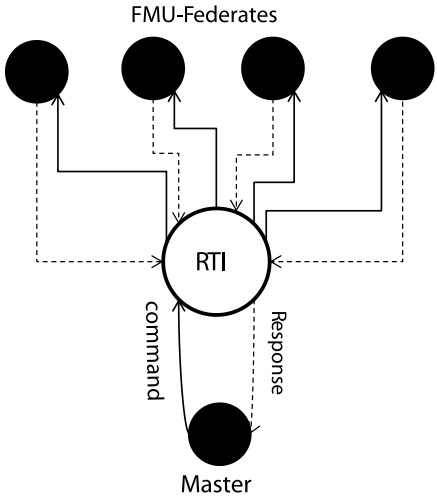


Fig. 2. Figure shows how WR method proposes to repetitively solve the coupled simulation components while sharing the coupling state variables. The position of internal integration points t_0, t_1, t_1 and s_0, s_1 is independent of the communication points T_n and $T_n + 1$. Here y_1 and y_2 represent outputs and u_1 and u_2 represent inputs to the system.

processes are sent to the master in form of tuples. A typical tuple has three elements, a **name**, a **value** and the **time**. The **time** element stores the information when the value of an output variable was changed, with respect to the simulation time. Its value is also used to identify a discrete event. The **value** element is used by the master to decide whether the iteration has converged or not. To access elements of a tuple, the following notation is adopted. The time is accessed as $time(x)$, value as $value(x)$ and name as $name(x)$, where x represents a single tuple or an element in the set of inputs or outputs. Generally, when an element from the set of inputs or outputs is sent to the master using a message, it has all three elements. Federates also communicate among themselves using tuples, but in that case, the element **time** is not present in the tuple.

Inputs and outputs are only communicated through the ⟨Update⟩ message. The second parameter of the ⟨Update⟩ message is the name of the class, and the third is the name of the attribute of the class to be updated. When an update is destined for the master, the second parameter of ⟨Update⟩ is "Master". When slaves communicate the updates among themselves, the second parameter is "Slave". In a Federation Object Model (FOM) the "Master" and the "Slave" represent classes. Although there are many different ways to create a FOM for the presented algorithm, it is assumed for a better understanding that each output of the system has a unique name which is represented as an attribute in classes "Master" and "Slave". The "Master" is used for the updates destined for the master and class "Slave" is used for slaves. When slaves communicate updates among themselves, there is no need to associate **time** of update with the value because in this context,

the information is meaningless. The representation of update messages as described above serves well to comply with the implementation. Still, the presented model is abstract enough to remove the HLA (or HLA RTI) from the picture, if needed. Such an implementation is only possible if the constraints imposed by the distributed computation model are kept intact.

The main idea of hybrid simulation algorithm revolves around the WR method of implicit solver coupling.[1] If the lines from 27 to 34 are replaced by one single line 33 in Algorithm S.A-1, then the algorithm will only be able to solve coupled continuous systems. Lines from 27 to 31 constitute the portion which is responsible for detecting and taking care of a discrete event. When a discrete variable changes its state, the respective slave FMU-Federate sends a negative value as the update time of the state variable. The statement is listed at line 27. With the negative value of update time, the master identifies that a discrete event has occurred.

From Listing S.A-1, it is clear that there are two main loops in the algorithm. One accounts for the simulation, in which the time is advanced. The second, inner loop, is for the convergence checking. During the execution of the inner loop, if there is a discrete event detected by the if statement at line 27, then the execution branches out to a procedure called $ProcessDiscreteEvent$. The procedure is responsible for finding the exact time of the discrete event and then handling the discrete event.

The easiest way to handle a discrete event is to "abort" the processing of the communication step and rollback to the previous state as soon as a discrete event is detected. Then reduce the step size to the minimum, and start simulating that communication time step once again. When the discrete event is detected for the second time, then exchange the discrete state variable among all other components, and proceed as normal. Reducing the communication time step to the minimum can cause performance issues. The procedure can be made faster by introducing stage-wise finding of the precise time of discrete events. In the first stage, a point t_l on the time axis is searched, such that t_l is very close to the point t_d and no discrete event occurs at t_l. Here t_d is the point on time axis where the discrete event occurs. In stage 2, the time step is reduced to the minimum and discrete state variable is propagated through the federation only once, just as described earlier. The process of handling the discrete event in a stage-wise manner is not part of Listing S.A-1. It is omitted to let the listing remain easily understandable.

Each command in the Algorithms S.A-1, S.A-2 and S.A-3 is accomplished by passing messages among processes. The messages used in hybrid simulation algorithm are described in detail in Sec. S-3 of supplemental material. Only the important messages are enumerated here.

- ⟨**Rewind**, *time*⟩: On receiving this command, a "slave" process sets its state variables back to the values attained at the end of the last time step.
- ⟨**Abort Iteration**, *time*⟩: The command is similar to ⟨Rewind⟩, here a slave rewinds inputs to the previous values along with the state variables.

- ⟨**Advance Time**, *time*⟩: The command asks the "slave" processes to integrate the FMU to the *time* parameter sent with the command. After the integration, a slave sends the updated state variables in the form of ⟨Update⟩ message(s).
- ⟨**Synch**, *time*⟩: On receiving this command, a "slave" process sends the updated state variables in the form of ⟨Update⟩ message(s), back to the master.
- ⟨**Update**, *cls*, *attribute*, *output*, *time*⟩: The messages are used to communicate the update of outputs. The first parameter can be either "Master" or "Slave", identifying the type of message. Second parameter is the name of the shared variable to be updated. The third parameter takes the value based on the value of the first parameter. For details, see Sec. S-3.
- ⟨**Share Data**, *time*⟩: The command asks all the slave processes to interchange their dependent variables among them. Each slave updates its outputs which it has published earlier, and then waits for the updates of all subscribed variables. ⟨**Share Data Non-discrete**⟩ and ⟨**Share Data Only discrete**⟩ do the same thing, except that the former works only for continuous variables and the later only for discrete ones. To communicate among themselves, slaves use the ⟨Update⟩ message, with parameter *cls* equal to "Slave".

An important procedure used in the implementation is *GetUpdates*. The purpose of this procedure is to make the master wait for all the updates destined for it. Its description start from line 2 in Algorithm S.A-1. The procedure sets the execution state *ExecState* of the master as $WaitForUpdates$. In this state, the master does not do anything except wait for the updates. During this time, it keeps sending ⟨TARA⟩ messages to allow the RTI to process and provide any messages it has received from other federates destined for it. The master changes its state during the processing of ⟨Time Grant⟩ message. The sequence of commands executed in result of receiving ⟨Time Grant⟩ message is listed at line 77 in Algorithm S.A-1. Here, it is checked whether all the updates which were destined to arrive, have arrived or not. Once all the updates arrive, the *ExecState* is changed from $WaitForUpdates$ to $UpdatesArrived$.

Both master and slave are state machines. The only difference is that the master changes its state based on some conditions, and a slave changes its state on receiving a certain message. During its simulation loop, the master keeps checking the values of different variables and changes its state based on the conditions applied on the variables. A slave on the other hand, changes its state only when it receives a command from the master to do so. In other words, the master and slave form the same state machine as shown in Fig. 3, the difference being the master is the leader in changing the states and slave is its follower.

To provide a background for Sec. 4.2, the working of algorithm is explained here in terms of message exchanges and state transformations, as shown in Fig. 3. The master reaches state S_1 after initialization. State S_1 represents that the execution has entered into the simulation loop. The first message originated by the master is ⟨Rewind⟩ message. When a slave receives this message and its simulation time

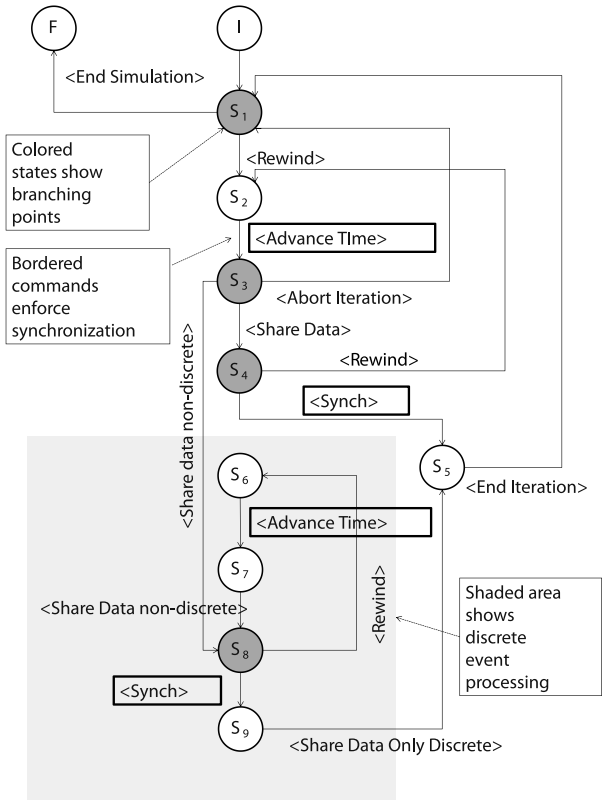


Fig. 3. The slave state machine of hybrid simulation algorithm. The area having dotted background shows the portion which tackles discrete event.

is at 0.0, it does nothing, except changing its state from S_1 to S_2. As the master also has changed its state to S_2 immediately after sending the ⟨Rewind⟩ message, it sends the ⟨Advance Time⟩ message, which leads it to state S_3. The slaves follow the master and move to state S_3. If the condition at line 27 in Algorithm S.A-1 is true, then this means that a discrete event has occurred. In that case, the master moves back to state S_1 after sending the ⟨Abort Iteration⟩ message. The slaves follow and move to state S_1 on reception of ⟨Abort Iteration⟩ message. If the condition at line 27 is false, the master moves to state S_4 and sends ⟨Share Data⟩ message. As before, the slaves follow the master. If there is no discrete event, then the loop continues until the values of state variables converge. The condition of convergence is checked at line 17 in Algorithm S.A-1. This leads the master from state S_4 to S_5, where it first sends the ⟨Synch⟩ message before moving to S_5. Then the master sends ⟨End Iteration⟩ message and moves back to state S_1 for the evaluation of the next time step.

If a discrete event occurs, then from state S_3 master moves to state S_8. The change is represented in algorithm by the call to $ProcessDiscreteEvent()$ at line 31

in Algorithm S.A-1. A similar loop is executed there. After the proper handling of the discrete event, master comes back at state S_1.

During the execution, the master keeps on sending the commands (messages) to slaves and slaves only wait for them to reach. Once they get the command allowed in a state, they execute the action and change the state. The master is the leader in its state transformations, while slaves experience the same state transformations by following the master.

In the description of algorithm most variables are named in a self-descriptive manner, though for more clarity, description of some variables is given here. Variable *discEvent* is a *boolean* variable which indicates whether a discrete event has occurred during the iteration or not. The variable *iterationOPs* stores the outputs of all FMUs at previous iteration, while *currOPs* stores their values at the current iteration. The variable *step* stores the length of communication time step. The constant TOL stores the value of error tolerance. The constant $\mathcal{N}_o$ stores the total number of outputs of all the FMUs which are expected to be received by the master. The constant FMU_{in} stores the number of inputs an FMU has subscribed to.

4.2. *Proof of the correct synchronized execution*

In order to prove that a slave always follows the correct execution path induced by the master, we denote the master state graph as $\mathcal{M}$ and slave state graph as $\mathcal{S}$, then

$$\mathcal{M} \preceq \mathcal{S}. \tag{1}$$

Equation (1) means that $\mathcal{S}$ simulates $\mathcal{M}$, or in other words $\mathcal{M}$ and $\mathcal{S}$ have simulation preorder relationship. Each move generated by the master $\mathcal{M}$ can be simulated by slave $\mathcal{S}$. From this fact, it can be concluded that a slave always follows the master correctly until and unless the order of commands sent from the master to a slave is changed during network communication.

Change in order only affects the execution when the slave is in a branching state. Branching states are highlighted in Fig. 3. At a branching state (or branching point), there can be cases when a slave can go into a direction not intended by the master. Before proving that by using synchronization points such a situation can be avoided, few structures must be defined

- Σ is the set of all commands $\{\mu_1, \mu_2, \mu_3, \ldots, \mu_n\}$.
- Γ_j is the list of all commands sent to a slave slv_j. A slave slv_j removes a command from Γ_j when it executes it. At any time, it may contain a limited number of commands induced by the master algorithm. As Σ is the set of all commands, so $x \in \Gamma_j \Rightarrow x \in \Sigma$. At any moment of execution $|\Gamma_j| \geq 0$.
- Λ is the set of all states $\{s_1, s_2, s_3, \ldots, s_n\}$ in the slave state graph.
- Λ^m is the set of all states $\{s_1^m, s_2^m, s_3^m, \ldots, s_n^m\}$ in the master state graph.
- $\Lambda_f \subset \Lambda$, contains all branching states in Λ.
- $\Lambda_f^m \subset \Lambda^m$, contains all branching states in Λ^m.

- Π is the set containing elements $\{\Pi_1, \Pi_2, \Pi_3, \ldots, \Pi_n\}$. An element Π_i is the set of all commands allowed at state s_i. So each $\Pi_i \subset \Sigma$, with $|\Pi_i| \geq 1$.

The slave algorithm works in a way that it keeps accepting the commands and saves them in Γ_j. At each state s_i, it follows the first command it finds in $\Gamma_j \cap \Pi_i$. In order to prove that the synchronization algorithm works perfectly, it is sufficient to prove that at any time during the execution of the algorithm, for the state s_i active at that time, and the slave slv_j, the following is true

$$|\Gamma_j \cap \Pi_i| \leq 1. \tag{2}$$

In order to prove the above statement, it should be noted that there are certain commands which work like synchronization commands for the slave and the master. The state where the slave ends up as a result of executing any of these commands is called as a "synchronization point" or synchronization state. The master also ends up at the similar state, in its own state graph, with a difference that it first gets into the state and then issues the command. At these points, the master waits for a response from all the slaves and it does not issue any more commands until it has received a response from all of them. It is easy to observe that the condition given in Eq. (2) can only be violated at a branching state where $|\Pi_i| > 1$.

Lemma 1. *Condition given in Eq.* (2) *remains valid, if starting from any state in* Λ_f^m *and* Λ_f *respectively, master and slave have to go through a synchronization point in their state graphs, in order to reach any state — same or different — in* Λ_f^m *and* Λ_f *again.*

Proof. Suppose that the master has passed through a branching state s_f^m. The corresponding state of s_f^m in slave is s_f. By this, the master sends a command in Π_f to the slave. As synchronization point s_s^m must follow it by definition, so the master must wait for a response from all the slaves at s_s^m. The corresponding state to s_s^m in slave is s_s. At s_s^m, the master cannot send any more commands until it receives all the responses. On the side of slave slv_j, Γ_j now may or may not contain a command present in Π_f. In order to send a response back to the master, the slave has to pass through s_f and reach s_s, because by Eq. (1), a slave simulates the master. To do this, it must consume the command sent from the master. So essentially when a slave reaches at s_s, it must have consumed the command in Π_f sent from the master. This means that when a slave slv_j sends a response back to the master, the set Γ_j does not contain any command in Π_f. The property holds for all branching states s_f and their respective set of commands Π_f, which proves that lemma 1 is true. □

The proof means that the condition given in Eq. (2) is entailed provided:

(1) All slaves simulate the master and the master sends commands in the correct order.

(2) Conditions imposed by lemma 1 are valid in the state graphs of master and slave.

The above discussion proves that if the condition given in Eq. (2) remains valid at all times during the execution of the simulation, then there will be no problem of synchronization. The property is enforced by lemma 1. Looking at the slave state machine in Fig. 3, it is clear that there are no two branching states reachable from each other without passing through a synchronization point. The only exception are S_4 and S_8 states, that can be reached from state S_3 without passing through a synchronization point. An ambiguity is possible if list Γ has both ⟨Share Data⟩ and ⟨Share Data Non-Discrete⟩ in it and slave is at state S_3. Looking at the master Algorithm S.A-1, it is clear that this is not possible because the master does not generate ⟨Share Data⟩ and ⟨Share Data Non-Discrete⟩ consecutively without issuing an ⟨Advance Time⟩ command, which enforces synchronization. It is important to mention that the proof of synchronized execution is sufficient for any number of slaves, because for the sake of synchronization each slave is only dependent on the master.

4.3. *Communication step size control*

$$\dot{y} = f(y, p), \tag{3}$$

$$\begin{aligned} \dot{\hat{y}} &= \hat{f}(\hat{y}, \hat{p}) \\ \dot{\tilde{y}} &= \tilde{f}(\tilde{y}, \tilde{p}), \end{aligned} \tag{4}$$

$$e_d = |y_f - y_0|_2. \tag{5}$$

Step size control offers many advantages in any numerical integration algorithm. Implemented correctly, it can significantly enhance the performance of an algorithm. Here too, the communication step size control offers many advantages. In the presented algorithm, each communication step is also followed by a sequence of messages among processes and the RTI. This means that more communication steps result in more messages to be communicated, which means more network traffic. Increasing network traffic not only increases the load on the network resources, but also increases the probability of unwanted network delays. So increasing the communication step size to the maximum, where the solution remains valid, is very beneficial.

Looking at Fig. 1, it is easy to understand that separating the ODEs means that some or all of the state variables in a subsystem are going to evolve without the knowledge of state variables in other subsystems. Mathematically speaking, suppose there is a system given in Eq. (3).

The state vector y contains n state variables $y = (y_1, y_2, y_3, \ldots, y_n)$. To perform the numerical integration of the system, if an implicit method is used, then the Jacobian of the system will be an $n \times n$ matrix, containing partial derivatives of all the state variables with respect to each of them. Partitioning the system in two (Eq. (4)), means that the Jacobian of each subsystem is also reduced to some

degree. If $\hat{y} = (y_1, y_2, y_3, \ldots, y_i)$ and $\tilde{y} = (y_{i+1}, y_{i+2}, y_{i+3}, \ldots, y_n)$, then this means that state variables in $\hat{y}$ are being evaluated without their partial derivatives with respect to $y_{i+1}, y_{i+2}, y_{i+3}, \ldots, y_n$. Similar is the case of $\tilde{y}$. This causes divergence in the solution. If the divergence remains in a realm where the system remains defined, then it is possible to recover the error through fixed point iteration. If not, then this means that the gap between two communication steps is too large.

Following the idea of divergence, apart from error tolerance, there is an additional parameter introduced, which is called "divergence tolerance" tol_d. This is tolerance for the error caused by the divergence. If the state variable vector, as a result of initial guess at the start of WR iteration, is y_0, and at the end of fixed point iteration, after the convergence, it is y_f, then an estimate of the error e_d caused by the divergence is given in Eq. (5).

At the end of each fixed point iteration, the communication step size is either increased or decreased by some percent based on the fact that $e_d + \tau_0 \|y_f - y_i\|_\text{max} < \text{tol}_\text{d}$ or $e_d + \tau_0 \|y_f - y_i\|_\text{max} > \text{tol}_\text{d}$. Here τ_0 is a small positive value used for normalization. During processing of a discrete event, the communication step size is intermediately reduced to minimum. After the discrete event, the communication step size takes some time to recover its value. At that moment, the mechanics of communication step size control becomes evident. Figure 5(a) shows the phenomenon by zooming into that situation for one of the test cases described in Sec. 5.

5. Test Cases

Two examples are chosen to demonstrate the correctness of the algorithm. First example in Sec. 5.1 examines a problem in which determining the time of discrete event is difficult. In this test case, the decision whether a discrete event has occurred or not depends on the values of input variables of a component. Changing these values by a very small fraction can invert the decision about the discrete event. So finding the precise time of a discrete event becomes very challenging.

The second case study examines a relatively bigger problem, with more than 50 state variables. The system is quite sensitive and comes from a real life problem in the domain of smart grids.

Results of both test cases are compared with the results of a monolithic simulator, to verify the correctness. Albeit, it should be kept in mind that the applicable domain of distributed simulation and monolithic simulation is almost disjoint. To simulate a system in a monolithic simulator, a modeler must have the complete mathematical description of the system. With the distributed simulation, on the other hand, a modeler may co-simulate partially known models in conjunction.

One purpose of developing complex simulations is to understand any phenomenon which is difficult to experiment in real life. In this way the modeler tries to verify the theory by "simulating" the real life phenomenon rather than going in the labor to perform a physical experiment. In some situations, there is no "complete" mathematical description of the phenomenon in the first place. The only thing a

Algorithm 1 Discrete Part CS-1

begin
  if $(y < h_{\text{stair}})$ **then**
    $\delta_{\text{contact}} \leftarrow 1$
  else if $(y > h_{\text{stair}})$ **then**
    $\delta_{\text{contact}} \leftarrow 0$
  end
  if $(x - \mathcal{N} + 1 + h_{\text{stair}} > 0)$ **then**
    $h_{\text{stair}} \leftarrow h_{\text{stair}} - 1$
  end
end

modeler knows is the mathematical models of "parts" of the system. To be able to see how these parts interact and evolve with each other, a modeler takes the help of a distributed simulation.

Further, there are situations where a modular approach is always more cost effective because independent models of individual components are easily usable in different scenarios. Case study 2 is one such example. In this case study, it is beneficial to create models of different components separately and let them co-simulate. Later, when it would be felt that more components should be added into the scenario, or behavior of a component has to be changed, a complete redesign of the scenario would not be necessary.

5.1. *Case Study* — 1

The first case study is a very popular hybrid system i.e., a ball being dropped from a height on stairs, namely, a "bouncing ball on stairs". The model is given by the system of System of equations 6. The discrete part is given by Algorithm 1. Figure 4 shows how different FMU-Federates are associated with each other via their state variables.

Here g is the gravitational constant, while c_0, c_1, c_2 and c_3 are constants that facilitate the phenomena offriction, air resistance, damping and mass of the ball. The variables h_{stair} and δ_{contact} represent discrete variables. The variable δ_{contact} shows whether the ball is in contact with the floor or not. When $\delta_{\text{contact}} = 1$, the system shifts its behavior immediately at that point. The variable h_{stair} shows the step of the stair that the ball is currently bouncing on. Initially, its value is $\mathcal{N}$ in Algorithm 1.

$$\begin{aligned} \dot{x} &= v_x, \\ \dot{y} &= v_y, \\ \dot{v}_x &= -c_0 v_x, \\ \dot{v}_y &= -g - c_1 v_y - \delta_{\text{contact}}((y - h_{\text{stair}})c_2 + c_3 v_y). \end{aligned} \tag{6}$$

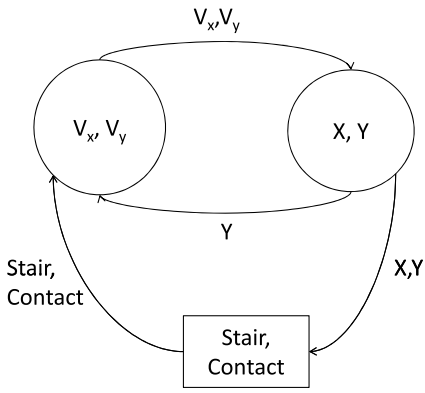
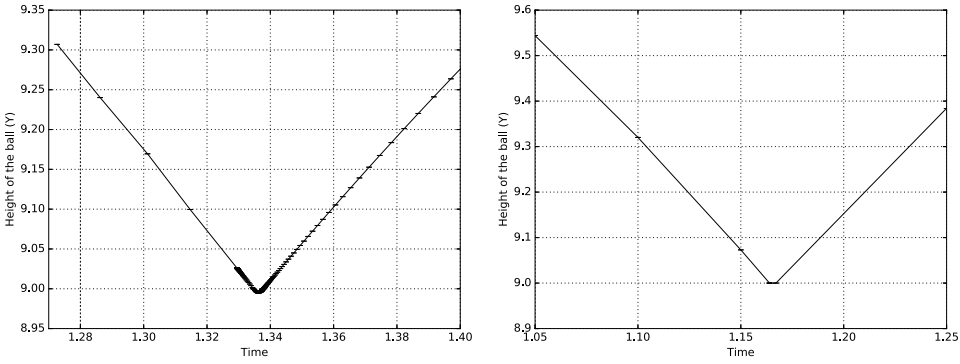


Fig. 4. Different treatment of discrete events by both solvers.



(a) Variation of communication step size during processing of a discrete event.

(b) OpenModelica's treatment of discrete event.

Fig. 5. Division and interdependence of different subsystems in the bouncing ball problem. The arrows show information flow. The square element shows discrete component, while the circular elements show continuous components.

For the presented run, the value of "divergence tolerance" was $\text{tol}_\text{d} = 1 \times 10^{-3}$. The value is relatively large. Using a smaller value makes results more accurate, but that results in more communication steps and performance deterioration.

Figure 6 shows the results produced by OpenModelica[18] and the hybrid simulation algorithm (more results can be seen in Sec. S-4 of supplemental material). It is clear that there are little differences in the results. The difference between results is obvious due to the completely different treatment of events in OpenModelica DASSL algorithm. Figure 5(b) shows how OpenModelica cuts the contact dynamics out, and converts the system into a piece-wise continuous system.

Lundvall *et al.*[19] describe how OpenModelica changes a hybrid system into a hybrid system of DAEs, separating it into a continuous part and a discrete part.

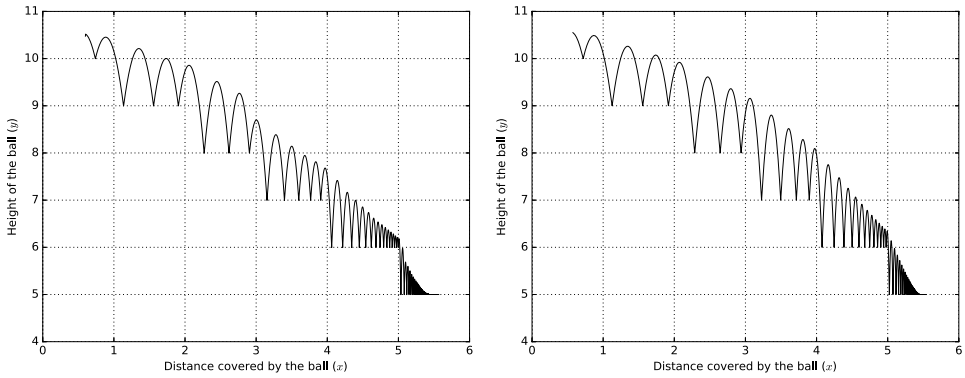


Fig. 6. Comparison between proposed algorithm (left) and OpenModelica (right).

This and many other simplification methods are examples of those advantages which monolithic SPs have. A distributed algorithm with current state of technologies cannot simplify the system as such. Most importantly, this type of simplification is something which a modeler may not wish to apply in complex simulations, as discussed at the start of this section.

5.2. *Significance of the approach*

Focusing on the specific type of simulation methodology, the presented hybrid simulation algorithm does not simplify the phenomenon of bouncing ball on stairs. It models parts of the system, discrete and continuous separately, and lets them evolve with each other governed by the algorithm. The results presented here show the success of the algorithm, as they are so close to the results obtained by a monolithic simulator (OpenModelica). From this, it can be easily deduced that the algorithm is able to simulate a partially modeled physical phenomenon successfully. Here a partial model means that the "complete" model of physical (or cyber physical) phenomenon is not known, and the modeler, by providing mathematical description of the parts of the system, is relying on the algorithm to accurately simulate the parts of the system as a whole.

5.3. *Case Study* — 2

The second case study is taken from the power system domain. In this case study, the performance of the secondary voltage control in a power system is investigated. The goal of the secondary voltage control is to keep the voltage of a pilot bus (a bus of interest in a large grid) at a predefined reference value.

The power system model chosen for this study is the IEEE 14 bus system model.[20] This system is used in many studies of power system transients due to its manageable size and complexity, while still being able to demonstrate the most typical transient phenomena.[21] The system is composed of 14 nodes (some of which

have demand connected to them) and five generators. Each of the generators is composed of a synchronous machine governed by an Automatic Voltage Control (AVR) system. The AVR controls the voltage level at the terminal bus. The terminal bus of a generator is the node at which that generator is connected to the grid. The secondary voltage control modifies the set-points of AVRs. The secondary voltage control receives these set-points and the set-point of the pilot bus from the grid operator.

$$\begin{aligned} q &= q_1 + K_p(V_{\text{pref}} - V_p) \\ \dot{q}_1 &= K_0(V_{\text{pref}} - V_p) \\ \dot{V}_{sg_i} &= \frac{1}{T_{g_i}}(X_{tg_i} + X_{eqg_i})(q.Q_{gr} - Q_{g_i} + Q_{\text{refg}_i}), \quad \text{where } i = 1, 2, 3, 6, 8 \\ V_{\text{gref}_i} &= V_{sg_i} + V_{gref_{0_i}}, \quad \text{where } i = 1, 2, 3, 6, 8. \end{aligned} \tag{7}$$

The simulation use case is composed of three FMUs. The first FMU is a continuous module which contains the model of IEEE 14 bus system. The second module is the secondary voltage controller module. The third module is a grid operator module that is implemented as a discrete event component.

The model for the IEEE 14 bus system together with AVRs is derived from the IPSL library.[22] The library is developed for Modelica use under the ITesla project. The mathematical model of the secondary voltage control is taken from,[23] the FMU is implemented in Modelica by the authors. System of Eq. (7) represents the secondary voltage controller in mathematical terms.

Among the parameters of the System of equations (7), K_0 and K_p represent the gains of the controller. In the presented simulation run, their values are 0.1 and 1.0, respectively. The inputs coming from the IEEE 14 bus system module are represented by V_p and Q_{g_i} for $i = 1, 2, 3, 6, 8$. The first five inputs are the reactive power components of the generation buses. The input V_p is the voltage magnitude of the pilot bus. The input from the discrete event module is V_{pref}, which is the set point for the pilot bus. All other variables are different parameters of the controller that can be changed according to the controller deployed at the site. The outputs from the controller are V_{gref_i} where $i = 1, 2, 3, 6, 8$. These are the calculated set-points for the AVRs of the generation buses. Naturally, they are the inputs to IEEE 14 bus module.

In the case study, the response of the grid voltage to the change in the voltage set-points is investigated. Bus 5 is chosen as the pilot bus and voltage set-point of this bus is stepped-up on every 60 sec to mimic the action of the system operator who is observing dangerously low level of voltage at this bus and is working on restoring it to nominal levels. The reference for the voltage on the pilot bus changes from 1 to 1.02, and then to 1.04, and finally to 1.06 per unit in a 3 min time span. Then the voltage is set back to 1.04. After 500 sec, the operator decides to set the voltage reference point even lower, i.e., 1.02, and after 60 more sec, it is set to 1.00. The pilot bus voltage behavior is observed in Fig. 7 (more results can be seen in

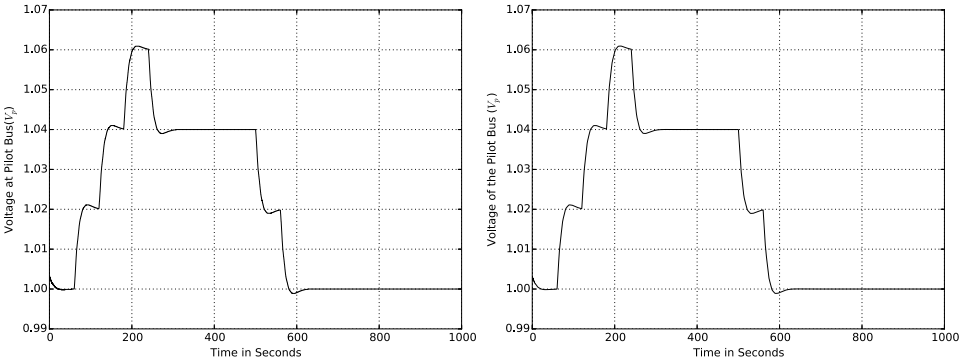


Fig. 7. Comparison between proposed algorithm (left) and OpenModelica (right).

Section S-4 of supplemental material). We see that the voltage level of the pilot bus increases with the increase in the reference voltage. This speaks in favor of a quality design of the secondary voltage loop.

In Fig. 7, comparison of responses is presented between hybrid simulation algorithm and OpenModelica. The right-hand side of Fig. 7 represents the simulation of the same system as a monolithic Modelica model. Only a small difference between the two is observed, at the order of 10^{-8}. Based on this observation, it can be concluded that the simulation of the power system using the hybrid simulation algorithm is as good as the monolithic one for this particular application (testing of secondary voltage control). Since power systems are complex in nature, further investigation is needed in order to assess the adequacy and accuracy of the hybrid simulation algorithm for other power system applications.

5.4. *Advantages of the modular approach*

There are considerable advantages of this approach. First, it allows the modular development of the simulation. Already tested modules can be coupled with each other for testing and verification. Different controllers with different power grid systems can be coupled and the results can be examined. Introducing many types of faults during the execution is possible. Introduction of noise in the sensory data of controller is possible. Most importantly, the discrete event module may be a lot more complicated than it is in this prototype. A well-formed artificial intelligence module can replace it, which collects the sensor values of the power grid and takes actions based on its expert knowledge. Implementing such an intelligent agent is much more a difficult task to do in Modelica, than doing this in a language specially designed for intelligent agents, like Prologue. Any other intelligence mechanism becomes far more usable, for example, artificial neural networks or support vector machines.

6. Conclusion

The article presents a distributed simulation algorithm for hybrid systems. The algorithm does not put any condition on the number or type of simulation components. Any numbers of continuous and discrete components are permissible. For modern simulation applications, many different SPs have to be used simultaneously. The algorithm presented here enables the modeler to develop different parts of the complete system in different SPs and then simulate them together. However, the components must adhere to the FMI specifications. The algorithm also uses an RTI specified by the HLA standard. The conformance to the HLA is not needed though. The article describes how an FMU can be changed into a component adhering to HLA specifications. Such a component is named FMU-Federate.

For verification, two hybrid systems are simulated, and their results are compared to the results produced by a monolithic solver, OpenModelica. The results show that there are some discrepancies in the details, still the overall behaviors of both the systems are identical to the behaviors presented by OpenModelica. It is argued that performance comparison between a distributed technique and a monolithic one is not justified, as both address their own spectrum of problems. A monolithic solver simply cannot be used in situations where there are more than one simulation components and each component is executed by a completely independent simulation package. Only a distributed algorithm can be used in such situations. One such algorithm is presented here. Using the algorithm, purely continuous components can be coupled with discrete event-based components.

References

1. Lelarasmee E., Ruehli A. E., Sangiovanni-Vincentelli A. L., The waveform relaxation method for time-domain analysis of large scale integrated circuits, *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **1**(3):131–145, 1982.
2. Mets K., Verschueren T., Develder C., Vandoorn T. L., Vandevelde L., Integrated simulation of power and communication networks for smart grid applications, in *2011 IEEE 16th Int. Workshop on Computer Aided Modeling and Design of Communication Links and Networks* (*CAMAD*), pp. 61–65, Kyoto Research Park, Kyoto, Japan, 2011, IEEE.
3. Tseng F.-C., Hulbert G., A gluing algorithm for network-distributed multibody dynamics simulation, *Multibody Syst. Dyn.* **6**(4):377–396, 2001.
4. Kübler R., Schiehlen W., Two methods of simulator coupling, *Math. Comput. Model. Dyn. Syst.* **6**(2):93–113, 2000.
5. Schweizer B., Li P., Lu D., Implicit co-simulation methods: Stability and convergence analysis for solver coupling approaches with algebraic constraints, *ZAMM-Journal of Applied Mathematics and Mechanics* **96**(8):986–1012, 2016.
6. Busch M., Schweizer B., Explicit and implicit solver coupling: Stability analysis based on an eight-parameter test model, *Proc. Appl. Math. Mech.* **10**(1):61–62, 2010.
7. Schweizer B., Lu D., Semi-implicit co-simulation approach for solver coupling, *Arch. Appl. Mech.* **84**(12):1739–1769, 2014.
8. D'Abreu M. C., Wainer G. A., Models for continuous and hybrid system simulation, in *Proc. 2003 Winter Simul. Conf. 2003*, Vol. 1, pp. 641–649, New Orleans, LA, USA, December 2003.

9. Zeigler B. P., Ball G., Cho H., Lee J. S., Sarjoughian H., Implementation of the DEVS formalism over the HLA/RTI: Problems and solutions, in *Simulation Interoperation Workshop* (*SIW*), pp. 65–73, Orlando, FL, USA, 1999.
10. Awais M. U., Palensky P., Mueller W., Widl E., Elsheikh A., Distributed hybrid simulation using the hla and the functional mock-up interface, in *IECON 2013 — 39th Annual Conf. IEEE Industrial Electronics Society*, pp. 7564–7569, 2013.
11. Cellier F. E., Kofman E., *Continuous System Simulation*, Springer, US, 2006.
12. Eker J., Janneck J. W., Lee E. A., Liu J., Liu X., Ludvig J., Neuendorffer S., Sachs S., Xiong Y., Taming heterogeneity-the Ptolemy approach, *Proc. IEEE.* **91** (1):127–144, 2003.
13. Müller W., Widl E., Linking FMI-based components with discrete event systems, in *2013 Proc. IEEE Int. Syst. Conf.* IEEE Conference Publications, pp. 5, IEEE Conference Publications, 2013.
14. Mets K., Ojea J., Develder C., Combining power and communication network simulation for cost-effective smart grid analysis, *IEEE Commun. Surveys Tut.* **16**(3):1771–1796, 2014.
15. Palensky P., Kupzog F., Smart grids, *Annu. Rev. Environ. Res.* **38**(1):201–226, 2013.
16. Attiya H., Welch J., *Distributed Computing*: *Fundamentals*, *Simulations*, *and Advanced Topics*, Vol. 19, Wiley series on parallel and distributed computing, John Wiley & Sons, 2nd edn., 2004.
17. Kuhl F., Weatherly R., Dahmann J., *Creating Computer Simulation Systems*: *An Introduction to the High Level Architecture*, Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edn., 1999.
18. Fritzson P., *Introduction to Modeling and Simulation of Technical and Physical Systems with Modelica*, Wiley-IEEE Press, 1st edn., 2011.
19. Lundvall H., Fritzson P., Bachmann B., *Event handling in the openmodelica compiler and runtime system*, Technical reports in Computer and Information Science. Linköping University Electronic Press, 1st edn., 2008.
20. Information Trust Institute, The IEEE 14-bus system, 1962.
21. Hashim N., Hamzah N., Abdul Latip M. F., Sallehhudin A. A., Transient stability analysis of the IEEE 14-bus test system using dynamic computation for power systems (dcps), in *2012 Third Int. Conf. Intell. Syst. Model. Simul.*, pp. 481–486, IEEE, 2012.
22. Vanfretti L., Rabuzin T., Baudette M., Murad M., iTesla power systems library (iPSL): A modelica library for phasor time-domain simulations, *Software X* **5**:84–88, 2016.
23. Milano F., An open source power system analysis toolbox, *IEEE Trans. Power Syst.* **20**(3):1199–1206, 2005.