IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received 25 October 2022, accepted 4 January 2023, date of publication 16 January 2023, date of current version 19 January 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3237075

RESEARCH ARTICLE

A Generic Interface for x-in-the-Loop Simulations Based on Distributed Co-Simulation Protocol

MIKEL SEGURA[1], TOMASO POGGI[2], AND RAFAEL BARCENA[3], (Member, IEEE)
[1]Embedded Systems Group, IKERLAN, Arrasate, 20500 Basque Country, Spain
[2]Mondragon Unibertsitatea, Arrasate, 20500 Basque Country, Spain
[3]Department of Electronic Technology, University of the Basque Country (UPV-EHU), Bilbao, 48013 Basque Country, Spain

Corresponding author: Mikel Segura (msegura@ikerlan.es)

This work was supported by Basque Government through the ELKARTEK Program under the AUTOEV@L Project under Grant KK-2021/00123.

ABSTRACT Co-simulation is a key step in the development of today's complex cyber-physical systems (CPS), specially in the integration and validation activities. However, performing a co-simulation involving models developed in different environments and possibly deployed in different platforms with mixed real-time and non real-time constraints is a challenging engineering task. A promising technology that could help overcome communication and synchronisation difficulties is the non-proprietary standard Distributed Co-simulation Protocol (DCP). This standard defines an application-level communication protocol, independent of the platform and the communication medium, that regulates the exchange of information between the co-simulation entities. This paper presents a co-simulation interface based on the DCP standard. It offers a novel approach to apply the DCP standard. Instead of using it as a model encapsulation mechanism, having to develop an specific DCP slave for each application, it is proposed to use it as a generic co-simulation interface. To this end, a Simulink library has been developed, allowing to connect models developed in Simulink with the outside world in an standardised way. Moreover, by exploiting the code generation potential of Simulink, a wide variety of devices become accessible, thus enabling x-in-the-loop simulations, which are commonly used tests in the verification and validation process of CPSs. This library has been tested in a soft real-time co-simulation application between a Simulink instance and an application running on a Xilinx Zynq Ultrascale+ System-on-Chip. As an additional contribution, an analysis of DCP synchronisation problems when simulating closed-loop systems composed of two slaves is performed. Finding that the main causes are the occurrence of random delays and that the simulations of the two slaves start at an arbitrary time. A possible solution to this problem is also presented.

INDEX TERMS Control systems, co-simulation interface, distributed co-simulation protocol, model-based design, model testing, simulink, x-in-the-loop.

I. INTRODUCTION

The concept of Cyber-Physical System (CPS) is defined in multiple ways throughout the literature, however, two characteristics are always present [1]: i) they are distributed systems where computational processes interact with the physical world; and ii) they are strongly influenced by communication aspects, such as interconnection or collaboration. Commonly mentioned examples include autonomous automobile systems, smart grid systems, process monitoring and control systems, or automatic aeronautical pilot systems. All of them are composed of multiple components, designed separately and integrated once their operation has been tested [2]. This variety of elements of which the CPS are composed, make them complex systems, in which it is common that both discrete and continuous dynamic elements converge [3]. Therefore, the process of development and subsequent verification and validation of a CPS is not exempt from challenges. However, there is a methodology that helps to address them, namely model-based design (MBD) [4]. This process is based on

The associate editor coordinating the review of this manuscript and approving it for publication was Mohsin Jamil.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

the creation of a model of each component, where only its relevant characteristics to perform a specific task are considered [1].

CPS are characterised by a heterogeneous nature, since different abstraction levels are required for their physical and computational components, as well as for their interactions. Therefore, each component is usually modelled in a different modelling and simulation (M&S) environment [5]. This can cause problems when integrating and simulating them for testing, either because of the communication interface between environments or because errors have arisen when porting a model created in one tool to another [6]. Therefore, to carry out the tasks involved in the whole design and validation process, interoperability between the tools plays a significant role [7]. To tackle this issue, the concept of co-simulation has emerged, where a framework is used to couple different execution environments, thus enabling the simulation of such complex systems. The concept of co-simulation is not limited to the interaction between simulation environments, but also to the interaction between a model simulated in some M&S environment and a physical element [2].

Nowadays, to deal with the development of the model-based systems, the market offers numerous M&S environments, such as MATLAB/Simulink by Mathworks or Dymola by Dassalt Systèmes, that allow to generate code from the developed models and to deploy it directly in a hardware platform. If exploited correctly, this workflow helps to reduce the design time and to eliminate errors caused by hand coding. On a parallel level, there exist associations dedicated to the development of standards to facilitate the analysis, management and (co-)simulation of models. Of particular interest is the Modelica Association, which promotes and maintains open access standards, such as: Modelica Language, Functional Mock-up Interface (FMI), and Distributed Co-Simulation Protocol (DCP) [8]. Modelica Language is a proposal for standardizing the modelling language. FMI is oriented to the exchange of models between different M&S tools. It was created with the aim of fostering the collaboration and innovation between suppliers and developers of the final products. DCP aims at defining a non-proprietary co-simulation protocol for integrating real-time systems into co-simulation environments, specially for use in distributed HIL simulations.

Despite the availability of these tools, performing a co-simulation between a M&S environment and a hardware platform remains difficult, mostly due to interfacing issues [6]. As it is proposed in [9], the first step in tackling this specific problem is to establish a real-time co-simulation architecture. The need for a real-time multidisciplinary communication framework is also mentioned in [10]. At the same time, [11] claims that a standardised solution is needed for the coupling of simulation tools and/or test benches. After a literature review on different simulation standards, we concluded that DCP can solve this problem, as it was specifically designed to integrate real-time systems into (co-)simulation environments. Additionally, as it is a non-proprietary standard, its usage is opened to the whole community. At the same time, we observed that DCP is a relatively new standard, thus, a few implementation of it are available. Moreover, the few mentions on Matlab/Simulink are particularly remarkable, as it is one of the most popular MBD tool in the field of CPS design. Indeed there is not a direct DCP plugin for Simulink and previous integrations of a Simulink model into a DCP co-simulation scenario were performed by using additional tools such as xMOD [12] or Model.Connect [13]. Therefore, we decided to implement the DCP standard in MATLAB/Simulink to reach a large sector of the engineering community. Another aspect that we have detected in the DCP literature is that it is used as a wrapper for the model to be communicated [14]. Thus, in order to communicate different models/systems, it is necessary to create a particular DCP slave for each one. In this article, instead, we propose to use the DCP as a generic communication interface, where several instances of the same slave can be used to perform the communication. In this way, we split the modelling and co-simulation tasks, allowing the user to focus on the modelling activities.

Even though our solution is based on Simulink, which is a proprietary environment, the possibility to generate C/C++ allows us to overcome this limitation. Our implementation will bring clear benefits to the state of the art of co-simulation. First of all, it constitutes an interface between Simulink and other M&S environments. Thus, it will be possible to co-simulate Simulink models with other simulators, provided that they implement a DCP interface. This interface could be implemented either by adapting the pre-existent C++ code [15] or by generating C code from our solution and than integrating it in another simulator. Moreover, the availability of a model implemented in DCP together with the possibility to generate code for embedded systems open the doors to perform Processor-in-the-loop (PIL) or FPGA-in-the-loop (FIL) simulations on real-time platforms. Indeed, in these type of simulations our model of DCP could be deployed on a remote embedded platform.

Summing up, the main contribution of this paper is a Simulink library based on the non-proprietary DCP standard. The aim of this implementation is to provide a generic co-simulation interface, allowing to link Simulink models with the outside world in an standardised way. On this basis, two further interesting contributions are also made. First, to test the library, we present an use case in which a closed-loop is simulated, linking a plant modelled in Simulink with a control implemented in an embedded device, concretely in a Xilinx Zynq Ultrascale+ development board. Second, we conduct an analysis of timing and synchronisation issues when using DCP for closed-loop systems simulation. From this analysis we propose a DCP synchronisation mechanism. It is important to remark that DCP does not include any synchronisation mean between models and simulations.

The paper is structured as follows. Section II establishes the bases of the research explaining three core concepts: MBD, co-simulation, and review of standards for model analysis and simulation. Section III explains the Simulink implementation of DCP. Section IV focuses on timing and synchronisation issues. Section V mentions the limitations of the aforementioned implementation. Section VI introduces the use case. Finally, Section VII resumes the main achievements of this work.

II. BACKGROUND AND RELATED WORK

The work behind this paper is based on three pillars: MBD, co-simulation, and simulation standards. Therefore, for a correct understanding of the paper, it is convenient to understand these concepts and to analyse related works on these topics.

A. MODEL BASED DESIGN

MBD is a widely used strategy in the development of engineering systems [7]. It is intended to reduce development time and to improve the quality of the system. To this end, the reuse of models and tests is promoted, thus avoiding iterative manual work and optimising the testing phase [16]. With this methodology instead of using physical prototypes and textual specifications, a model of the system to be developed is created and, through a series of simulations, its behaviour is verified in order to check its veracity. A model is a virtual representation of the system that includes every component relevant to reproduce its desired behaviour. Following the standard MBD approach, the model of each part of the target system is modified and tested while undergoing various simulation processes until the required behaviour is achieved. Once this incremental process is terminated, it is possible to generate the software code (e.g. C, C++, or HDL) that define the model to deploy it directly in a specific hardware platform [17]. Therefore, the use of this methodology not only implies a reduction of the development time, but also lowers costs and, by avoiding the use of real prototypes, also prevents physical damage, either material or human [18]. Some examples where MBD is used to develop complex systems can be found in [16], [18], and [19].

B. CO-SIMULATION

Co-simulation has emerged as the solution for the development of the complex CPSs that are being created nowadays. It is common for the elements that compose these systems to be developed separately, not only with different tools, but also by different vendors. Therefore, all these partial solutions have to be integrated and this is where co-simulation comes into play [20]. Within co-simulation, a growing practice is distributed co-simulation [21], [22]. It refers to performing an online co-simulation, linking systems that are located in different geographical locations. It is especially useful for integrating externally provided elements that have Intellectual Property (IP) [23].

Hence, in short, co-simulation allows communication between models developed in different M&S tools, thus enabling each part of the system to be developed in a suitable environment. Furthermore, it promotes modularity, which means that it is possible to perform independent testing of the models that compose the system. This way, it ease the replacement of modules/models that compose the system, while at the same time permitting the reusability of models. For these reasons, it can be said that co-simulation is consistent with the MBD paradigm.

In order to boost this potential, several projects have emerged, such as MODELISAR, ACOSAR or EMPHYSIS; which have led to the creation of different standards, such as FMI, DCP or eFMI respectively. In Section II-C some of these standards will be explained, with special emphasis on DCP.

Standard simulations and co-simulations are common practice in MBD, since it is usual to pass through different simulation phases, known as X-in-the-loop (XIL), to test and verify the developed models, e.g. in [22], [24], and [25]. Figure 1 shows the main simulations that compose the XIL concept in the field of control system design: Model-in-the-loop (MIL), Software-in-the-loop (SIL), Processor-in-the-loop (PIL) and Hardware-in-the-loop (HIL). It is important to remark that a HIL simulation requires real-time conditions, which implies that the response of the system should be consistent with the timing of its environment [26]. That is to say, rather than being fast, the system must react within the precise time it is predicted to work. This and other features that define HIL systems are discussed in [27]. Finally, FPGA-in-the-loop simulations are special subset of PIL, where the processing unit is an FPGA instead of a micro-controller.

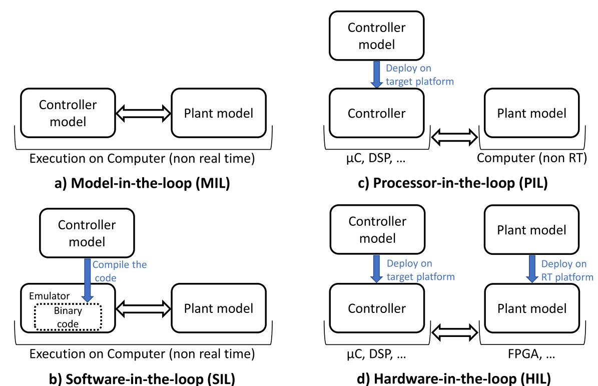


FIGURE 1. X-in-the-loop simulations.

The market offers various solutions to perform these kind of simulations, especially for HIL [28], [29]. However, these tools lack of interoperability, i.e. it is usually hard or impossible to exchange models between them or to perform simulations involving models developed in different environments. This paper aims at addressing this interoperability problem through the use of non-proprietary standards, fostering the development of a co-simulation tool that can be integrated in different environments and thus, to find a cost effective

solution. Moreover, a generic SoC with an integrated FPGA will be used, which also allows to perform such simulations [24], [30].

C. STANDARDS FOR MODEL ANALYSIS AND SIMULATION

There exist three association specially focused in the development of these kind of standards, namely, Modelica Association, Institute of Electrical and Electronics Engineers Standards Association (IEEE-SA), and Association for Standardization of Automation and Measuring Systems(ASAM). Each of them has developed a number of standards, those of which result of special interest:

- Standards of Modelica Association: Modelica Language, Functional Mock-up Interface (FMI) and Distributed Co-Simulation Protocol (DCP).
- Standards of IEEE-SA: High-Level Architecture (HLA) and Standard SystemC Language Reference Manual.
- Standards of ASAM: ASAM XIL, ASAM XIL-MA.

Within the literature, it is common to find works in which these standards are used collaboratively. For instance, FMI and HLA are combined in [31] and [32]. On the other hand, there are works that compare them. This is the case of [33], where the co-simulation potential of FMI and HLA, among others, is discussed. The authors of [34] analyse the extent in which some of these standards are used in continuous, discrete and hybrid time simulations. After analysing these standards, it clearly emerged that DCP is a non-proprietary standard oriented to the integration of real-time systems into co-simulation environments caught our attention. The fact that it is a relatively new standard is also interesting, so its potential is yet to be discovered. Seeing this, together with the fact that it had no direct application in Simulink, a widely used tool for modeling and simulation, we had the idea of using DCP slaves as a generic interface that links Simulink models with the outside world. This is the reason why this paper will only focus on DCP standard, therefore, this is the only standard that will be discussed in depth in this section.

The DCP standard originated from an ITEA project called ACOSAR with the aim of introducing real-time devices into co-simulation environments. It was later adopted and maintained by the Modelica Association. Regarding the documentation about DCP, the Modelica Association provides the specification document [35] and an example of the DCP library implemented in C++ [15]. For the reader convenience the main characteristics of the protocol are summarised in the remainder of this section.

DCP is an application-level communication protocol, independent of the platform and the communication medium. To enable the exchange of information between the co-simulation entities, underlying transport protocols such as Bluetooth, UDP or CAN can be used. It operates on a master-slave architecture. Despite of this, its specification only covers the definition of the slave, which includes the data model, a finite state machine, and the communication protocol. The particular description of the slaves is represented by XML files (DCPX), which includes information such as supported transport protocol, operation mode (i.e., Non Real Time (NRT), Soft Real Time (SRT), or Hard Real Time (HRT)), time resolution, inputs, outputs, etc. If SRT or HRT is chosen as operation mode, the communication will be done according to absolute real time, i.e., the DCP entities must work synchronised with the Newtonian time represented by a UNIX time stamp in UTC format, which is defined in seconds with reference to January 1st, 1970, 00:00:00 UTC. The present work is focused on SRT and HRT modes only.

The information exchange is done though packages called protocol data unit (PDU). They are categorized in families, namely: Control, Notification and Data. Within Control families there are other two families, named Request and Response. The functionality of each family is explained in Table 1. All of them are structured by predefined fields, the first one, called *type_id*, is common to all PDUs and is where their type is specified. Thanks to these families, DCP can perform different actions: exchange of simulation data using DAT PDUs; transmit notifications through NTF PDUs; and perform a configuration and control mechanism based on a Request and Response pattern, where the master is in charge of sending the PDUs of the Request family (CFG, STC and INF), while the slaves confirm the reception through RSP PDUs.

TABLE 1. Protocol Data Unit (PDU).

<table>
<tr><td rowspan="4">Control</td><td rowspan="3">Request</td><td>Configuration (CFG)</td><td>They can only be sent by the DCP master. They are used to request certain configuration setting from DCP slaves.</td></tr>
<tr><td>State Change (STC)</td><td>They can only be sent by the DCP master. They are used to trigger the transition between the states of the slave's FSM.</td></tr>
<tr><td>Information (INF)</td><td>They can only be sent by the DCP master. They are used to query state, error, or log information to DCP slaves.</td></tr>
<tr><td colspan="2">Response (RSP)</td><td>They are only sent by DCP slaves. They are used to acknowledge received Request family PDUs.</td></tr>
<tr><td colspan="3">Notification (NTF)</td><td>They are only sent by DCP slaves. They are used to notify the master a successful state transition or a log entry.</td></tr>
<tr><td colspan="3">Data (DAT)</td><td>They can used by all DCP entities to exchange input, output or parameter data.</td></tr>
</table>

Regarding the provided library [15], it consists of several source (.cpp) and header (.hpp) files that describe the behaviour of a generic slave, i.e., they implement the description indicated in the specification document. Additionally, an example which consist of a communication between a master and a slave is also provided. In this example, the definition of both entities is provided in different header (.hpp) files. The particular functionality and the configuration of the slave are hard coded in one of this files. Therefore in

order to modify the functionality, this file must be manually modified. The slave configuration affects different aspects, such as the number of inputs and outputs, time resolution, etc., and must be set in accordance with the previously mentioned DCPX file. It is worth to mention that in this example, the DCP master reads the slave's information from the DCPX file, however, the slave do not use it. In the configuration process, the master sends this information to the slave. The latter compares it to see if the configuration that the master is sending corresponds to the one programmed in the source files.

Additionally, there are several works that help understanding the standard and provide interesting contributions to DCP. This is the case of [36], where the specification is summarised and different examples for its application are also given. The paper [37] provides a configuration mechanism based on a bin packing algorithm. Specifically, the effect of changing the distribution of the sizes of the PDUs is analysed with respect to the variation of network load and to the robustness against packet loss. In [38] a DCP master state machine is proposed; it should be remembered that the specification only covers the definition of the slave. In [39] time synchronisation is discussed, which is one of the points that is not defined in this protocol. This issue is also addressed in the current work, as it has been one of the problems encountered when simulating closed-loop dynamical systems. In [40] a working methodology based on the IEEE1730 standard that is compatible with the DCP standard is presented. In the presented use case, it is of particular interest that one of the slaves is encapsulated in an FMU. To accomplish this, the authors use the C++ library provided in [15] and map the states of the DCP state machine with the states of the FMI state machine. In [41] a proposal to simplify the slave configuration process is made based on the modelling of the co-simulation scenario.

There are also other publications that show interesting use cases. In [13], for instance, a distributed co-simulation is performed, i.e. the DCP is used to link two slaves located in different countries. One of the slaves takes data from a co-simulation platform called Model.CONNECT, where models developed in different environments (e.g. MATLAB/Simulink, FMU from Dymola and CarMaker) are running, while the second slave is a small scale test bed. Of particular interest is the use case shown in [12], where DCP is used to couple on one hand, a DCP master in xMOD co-simulation platform and, on the other hand, a DCP slave that works as a client of CARLA software to control the simulation of a 3D virtual vehicle.

Summing up, DCP is a suitable tool to carry out distributed co-simulations, both in real-time and in non-real-time. Additionally, the fact that it is independent of the communication medium (e.g. UDP or CAN), i.e. it works in a higher abstraction layer, provides a high degree of versatility and independence form the target platform. This point is very interesting in order to develop a generic communication interface. Observing the applications mentioned above, a particular slave has been created for each of them. Contrarily, we propose to develop a generic DCP slave that acts as a general communication interface between two or more models running in different M&S environments. This, together with the development in Simulink, which apart from being a widely used tool in the development of models has a direct application on various hardware platforms, we obtain a multi-purpose tool that can greatly facilitate co-simulation.

III. DCP IMPLEMENTATION IN SIMULINK

This section presents how the DCP standard has been implemented in Simulink. As a guide for this development, two elements have been consulted: the DCP specification document [35] and the example developed in C++ provided by the Modelica Association [15]. The proposed implementation is based on the DCP standard version V1.0.0 released in March 2019, which is the latest release at the time of writing this paper. In the description of the implementation the references to specific clauses of the standard will be in italics to better guide the reader. For a deeper understanding of the protocol the reader is referred to the standard itself, which is available publicly.

For the implementation of the DCP in Simulink, four aspects were taken into account:

- **DCP slave:** We only implemented the DCP slave, whereas we reused the existing master provided in [15] to perform tests. Indeed, as mentioned in Section II-C, the protocol only covers the definition of the slave, whereas the design of the master is left open. This is possible thanks to how the communication between slave and master has been designed, i.e, though Control and Notification PDUs (see Table 1). Therefore, the master must be able to: i) communicate at least with one DCP slave; ii) manage the states of the slave(s) by sending STC PDUs and checking the reception of the corresponding RSP state acknowledgement PDUs; iii) manage the configuration of the slave(s) by sending CFG PDUs and checking the reception of RSP acknowledgement PDUs.
- **Limit of arrays size:** Both the specification and the C++ example contemplate that the arrays size is not limited. This is not possible in Simulink, so it has been necessary to limit the size of all the arrays. As a general rule, it has been chosen to limit arrays to 32 elements, with a few exceptions that will be discussed below. Nevertheless, the size is an arbitrary upper limit and it could be increased in future implementations.
- **DCP slave functionality:** Usually the system modelled by the slave is embedded into the DCP implementation. This means that the communication functionality is coded together with the model in single software entity. On the contrary, the main idea of the proposed implementation is to provide a way to link an arbitrary Simulink model with another simulation process performed in an external M&S tool or even in another device, such as a Xilinx Zynq Ultrascale+. To do this, a mechanism must be conceived to separate the

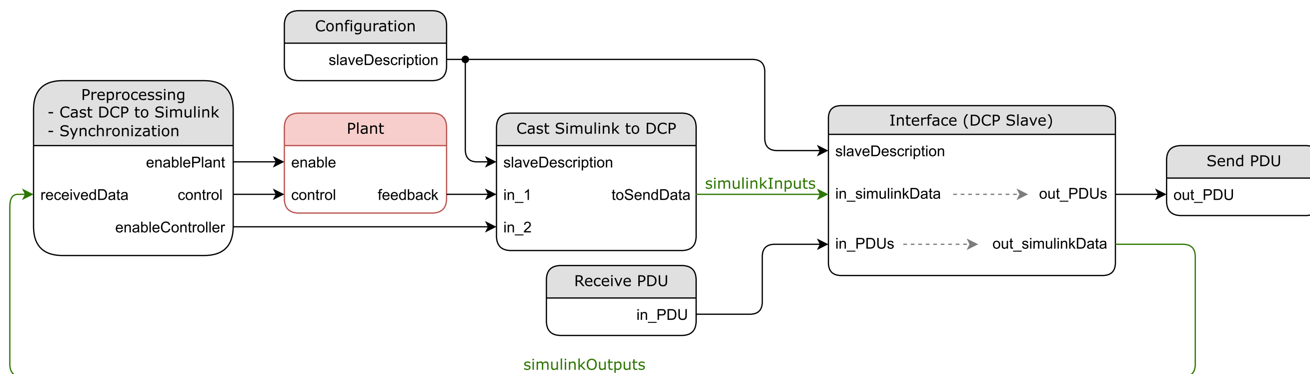


FIGURE 2. Diagram of the implemented example to test the developed DCP library in simulink.

communication functionality from the model. This will allow the user to focus on the development of the Simulink model rather than on communication aspects.

- **Fixed-step simulation:** Due to the nature of DCP only fixed-step simulations are taken into account in this work.

We decided to develop a block library to implement the DCP slave and the accessory functionalities (e.g. configuration, type cast, or UDP communication). Figure 2 shows a diagram where the main blocks of the library are instantiated. Notice that this figure represent one of the experimental setups that will be used in Section VI to verify the correctness of the implementation. In the following subsections these blocks will be explained in details.

A. INTERFACE BLOCK (DCP SLAVE)

This is the main block, which contains the main functionalities of our implementation of DCP. That is, on the one hand, a common functionality for all DCP slaves which consist of the state machine and the elements described in the standard specification. On the other hand, it implements an interface compatible with any Simulink model.

To develop the state machine, the Stateflow tool of Simulink has been used to code the scheme described in the DCP specification document, concretely in the *Figure 1: DCP slave state machine, p. 21*. The result can be observed in Figure 3. As can be appreciated, thanks to the formalism employed by Stateflow [42], it is possible to make a direct graphical implementation of the state machine. This provides a valuable visual aid to understand the slave's operation, which would be hard to get from a C++ implementation.

In order to represent the data structures described in the standard, it has been chosen to employ Simulink buses, the equivalent of C structures. The most relevant data structures are:

- The features of the slave and all of its sub-elements (number of inputs and outputs, time resolution, underlying communication protocol, etc.). See the point *5 DCP Slave Description, pp. 78-99,* of the specification document.
- PDUs. PDUs are the messages that DCP slaves and masters send to each others. In *Table 62: Generic PDU structure, p. 41,* of the specification document.
- Enumerations such as data types (see *Table 2: Supported data types of the DCP, p. 12*), the ID of each state machine state (see *Table 13: State IDs, p. 23*), and the error codes (see *3.4.7.2 List of Error Codes, pp. 55-57*).

Additionally, other buses that are not considered in the specification have been created, for instance to describe the input and output variables to the Simulink model. These buses are named *simulinkInputs* and *simulinkOutputs*. This names are given from the perspective of the DCP slave, i.e. *simulinkInputs* contains the data that a Simulink model transmits to the DCP slave for transmission to another DCP slave. Whereas *simulinkOutputs* contains the data that the DCP slave receives from another DCP slave to input into a Simulink model. The use of these buses can be seen in the model of Figure 2 and a description of their fields is provided in Table 2, Table 3, and Table 4. As it is reported in the tables, *inVars* and *outVars* fields can contain up to 8 elements. It is worth to mention that when using DCP, each variable is assigned a unique identifier called value-reference (see *3.1.18 Variables, p. 15*).

As mentioned above, the main idea behind the proposed implementation is to allow a Simulink model to communicate with the outside. This is achieved by the interface provided by the *simulinkInputs* and *simulinkOutputs* buses, that takes simulation data from Simulink and transmit them through PDUs, and vice versa. The main advantage of this solution is that, unlike the example provided in C++ [15], the functionality of the model can be coded separately from the communication, that it is handled by the slave, so that the user does not have to modify the code. In this way, anyone can design a functionality and link it to the slave via inputs and outputs without the need for in-depth knowledge of the protocol.

The interface of the *DCP slave* block is composed of 5 inputs, 4 outputs and 2 sets of parameters:

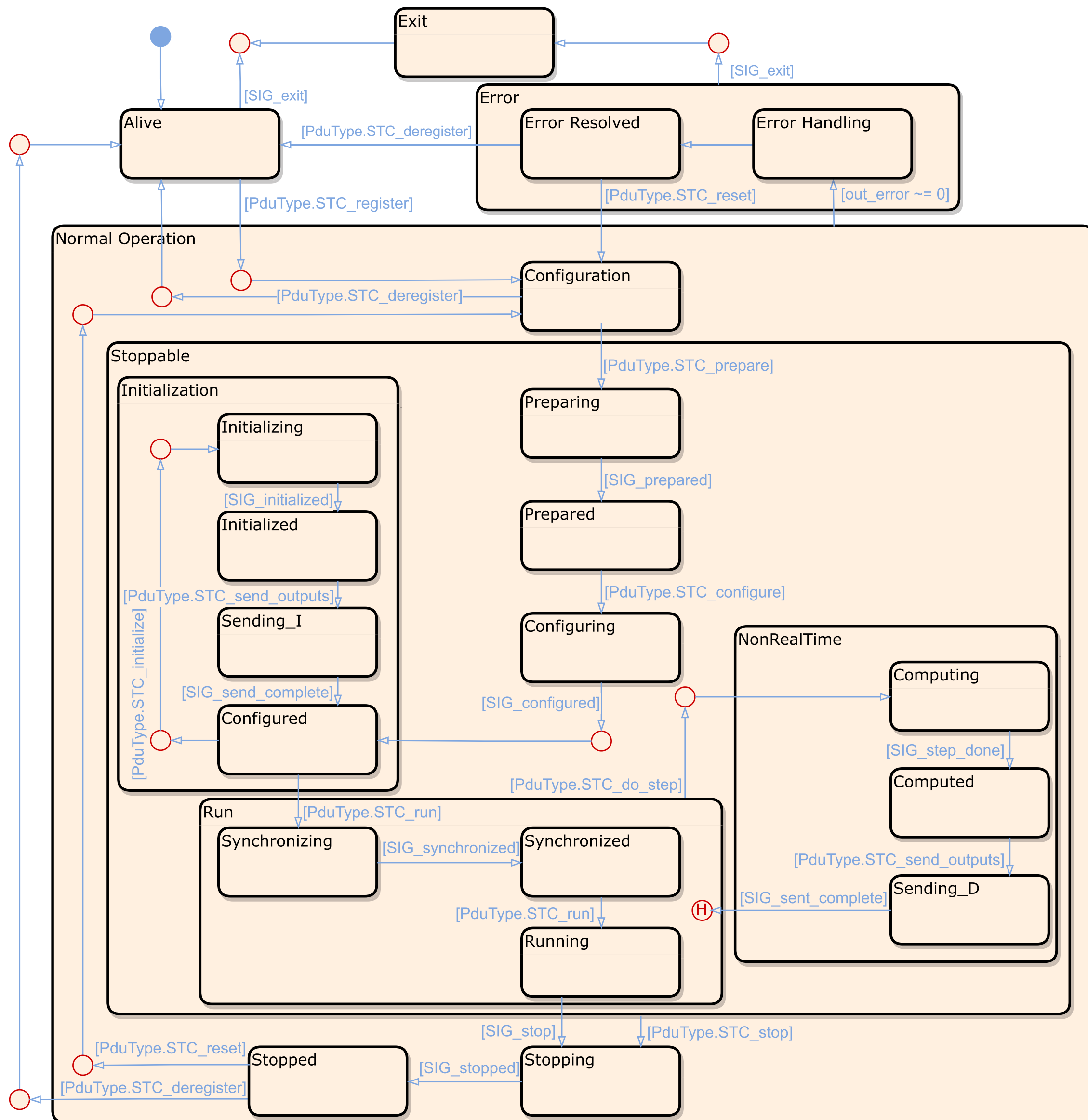


FIGURE 3. DCP slave state machine done with Stateflow tool.

1) INPUTS

- *in_DcpSlaveDescription*. This input is a bus that defines the particular slave description in accordance with the DCPX file. This description is configured with the library block explained in Section III-B.
- *SIG_exit*. Signal to stop the execution of the state machine. Normally it will not be used but it is contemplated in the standard.
- *in_Simulink*. This is the port where the bus containing the Simulink input variables arrives. That is, the data that is to be passed to other slaves. The data in the bus is filled by *castToDcpIn* block; see Section III-D.
- *in_PDU*. This port receive the PDUs received by other DCP entities, i.e. by other slaves and by the master. The PDUs are received by the *ReceivePDU* block which is described in Section III-C.
- *in_PDU_length*. In this entry the length of each received PDU must be entered. This value is also obtained from the Transport Protocol Blocks described in Section III-C.

2) OUTPUTS

- *out_Simulink*. The PDUs that have been converted into Simulink data are transferred via this port. In order to use this information in a Simulink model, it is necessary to use *castToSimData* block (see Section III-D).
- *out_logVaiables*. The use of this port is optional, as its use is limited to monitoring the variables (both inputs and outputs) handled by the DCP slave.
- *out_PDU*. This port outputs the PDUs to be sent to other DCP entities through the *SendPDU* block, described in Section III-C.

TABLE 2. SimulinkOutputs bus attributes.

Attribute name	Data type	Description
simulationTime	double	Indicates the running time of the current simulation.
currentStep	uint64	Indicates the steps that have been executed in the current simulation.
inNumber	uint8	The number of variables that are passed from the DCP slave to the Simulink environment.
outVars[8]	struct: varData uint8 uint32	A sub-bus composed of three fields: - *varData* data type field contains the specific data of the variables that are passed from the DCP slave to the Simulink environment. - *uint8* data type field is an enable signal which indicates if the data from *varData* can be read. - *uint32* data type field indicates the step size of the variable described in *varData* field.

- *out_PDU_length.* This port indicates the length of the message to be transmitted, its use is optional, depending on whether it is needed in the *SendPDU* block (Section III-C).

3) PARAMETERS

- *Info logging parameters.* Parameters that enable the logging of debug information and that specify the name of the log file.
- *Error logging parameters.* Parameters that enable the logging of the errors defined in the DCP protocol and that specify the name of the log file.

It is worth to mention that each of these slaves can support a variable number of inputs and outputs, both limited to 8 units. In other words, currently, a maximum of 16 variables can be used: 8 input and 8 output variables. These variables are the ones that compose the aforementioned *simulinkInputs* and *simulinkOutputs* buses.

B. DCP SLAVE CONFIGURATION BLOCK

This block is used to set the bus that contains the slave configuration. The bus is defined in section *5 DCP Slave Description, pp. 78-99,* of the standard and a schematic description can be found in *Figure 16: DCP slave description root level XSD schema, p. 80,* and *Figure 17: dcpSlaveDescription element attributes, p. 81*. This block is a template that must

TABLE 3. SimulinkInputs bus attributes.

Attribute name	Data type	Description
displayPeriod	uint32	Input/output log time in simulation steps.
inNumber	uint8	The number of variables that are passed from the Simulink environment to the DCP slave.
inVars[8]	varData	A sub-bus that contains the specific data of the variables that are passed from the Simulink environment to the DCP slave.

TABLE 4. VarData bus attributes.

Attribute name	Data type	Description
valueReference	uint64	The value reference of the variable, it must coincide with the set in the slave's description.
dataType	uint8	The data type of the variable.
dimensions	uint64	The dimensions of the variable.
payload	uint8	Contains the variable data.

be filled manually by modifying the values in the constant blocks it contains. This procedure is similar to the C++ implementation [15], as it is configured by editing a header (.hpp) file. As a future development it could be possible to automate this process by implementing a parser of the DCPX file.

C. TRANSPORT PROTOCOL BLOCKS

SendPDU and *ReceivePDU* blocks are in charge of sending and receiving PDUs, respectively. As mentioned in Section II-C, DCP is a layer that is placed on top of a communication protocol, either UDP, TCP/IP or CAN. Once a message is received, it must be formatted into a PDU. Also, when a PDU is being sent is must be formatted to the data structures of the communication protocols. As it can be seen, at low level these operations depends on the specific communication protocol.

For the moment, the only transport protocol implemented in our library is UDP. Nevertheless, being aware that in the future other communication modes may be used, these blocks have been created so that they could be easily updated. In particular we employed a "Variant Subsystem", that allows to include several different functionalities in it, leaving to the

user the option to choose which of these functionalities has to be executed during the simulation.

To implement the UDP communication functionality, the blocks provided by the Simulink Real-Time toolbox have been used as a baseline. Some extra logic has been added to handle outbound messages queue and enables, accordingly to the FSM in the DCP Slave Block. This is necessary since more than one outbound PDU can be generated at the same time. Moreover, apart from sending and receiving messages, the possibility to save all sent and received PDUs to a file has been implemented too. The latter is done by using another block of the library, namely dataLogging block, which is explained in Section III-F.

D. CAST BLOCKS

As mentioned in Section II-C, DCP uses DAT type PDUs to exchange data. These PDUs contain a field called *payload*, which consists of an array of uint8. The section *3.1.12 Data Type Encoding, pp. 12-14,* of the specification explains how the data must be encoded in the *payload*. To comply with these requirements, two blocks have been created, the first of them, called *castToDcpIn*, converts the Simulink data to *payload* field data. The second, called *castToSimData*, performs the opposite operation, converting the *payload* field data to a Simulink data type. Both blocks are based on the aforementioned *simulinkInputs* and *simulinkOutputs* buses (Section III-A). In this way, *castToDcpIn* block outputs a *simulinkInputs* bus to pass data from a Simulink model to the DCP slave. The maximum number of data that each DCP slave can transmit from a Simulink model is set to 8. For this reason, up to 8 DCP inputs can be configured in this block. To use this block the user only have to select the amount of data to be transmitted and to set the value-reference of each input. In this way, the DCP slave knows to which output it has to connect each of the inputs. In the same way, *castToSimData* block transforms *simulinkOutputs* bus to Simulink data types.

E. SLAVE SYNCHRONISATION BLOCK

This block had to be created because the DCP does not include mechanisms for time synchronisation (see *3.1.13 Timing, p. 14*, of the specification). The need for this block emerged from the analysis of synchronisation problems in the simulation of closed-loop systems that will be presented in Section IV. In brief, the analysis concludes that although the DCP worked correctly and no data is lost, there is a desynchronisation that prevented the system from behaving exactly the same in all simulations. In order to avoid such a situation, a synchronisation system has been developed. Both analysis of the synchronisation issues and the algorithm used to solve them are presented in Section IV.

F. ADDITIONAL BLOCKS

Additional blocks that perform sub-activities have also been created. For instance, with the *Data Logging* block, error and log messages can be stored. Instances of this block are used, for example, in *Transport Protocol* blocks (Section III-C). We developed also Debugging blocks, which can be used to monitor some of the data buses that are defined in the specification.

IV. TIME SYNCHRONISATION

As it will be explained in Section VI, to test the developed DCP library, we decided to implement a closed-loop system operating in SRT mode. This system is composed of two subsystems: a plant (first order system) and a control algorithm. As mentioned in Section III-E, when performing the first simulations with this system, we realised that the response was not deterministic and that it varied in each simulation.

To find the reasons causing behind this behaviour, we conducted an analysis of the synchronisation issues between these two subsystems (Section IV-A). To do so, we assume, for simplicity, that the co-simulation involves just two subsystems, *Subsystem 1* and *Subsystem 2*, each running in a different simulation environment. As depicted in Figure 4, each subsystem is composed of the models under tests and the DCP library blocks that allow the co-simulation. Both simulation environment are configured with the same timing parameters. The step size at which the simulation environment runs will be referred to as "simulation step size" and it will be denoted by S_{ss}. The state machine of the DCP slave is also synchronous with this step-size, i.e. the states are executed every S_{ss}. Accordingly, the models will be executed at bigger step sizes, that will be termed "model step size" and denoted by M_{ss}. We assume that M_{ss} is a multiple of S_{ss}, that is: $M_{ss} = \alpha \cdot S_{ss}$, $\alpha \in \mathbb{N}$, $\alpha > 0$. Notice that this is required by the fact that the simulation includes the DCP framework, that requires two simulation steps to perform the communication tasks between the simulators. In other words, the state machine that compose the DCP slave and the communication blocks needs two S_{ss} to receive, process, and send a data. This was discovered empirically, as running the model and the slave with the same step sizes, i.e. if $M_{ss} = S_{ss}$, the transmitted data take always a null value. Contrarily when increasing the value of M_{ss} to the double of S_{ss}, i.e. when $M_{ss} = 2 * S_{ss}$, the data was transmitted correctly. It is also important to remark that S_{ss} is locked to the clock of the platform where each simulator is running and that both clocks must be synchronised, as a requirement of DCP.

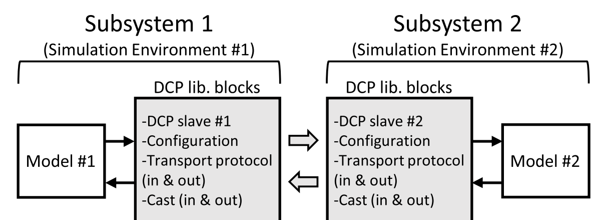


FIGURE 4. Schematic concept of a system to which the synchronisation algorithm is applied. We assumed that both subsystems.

In order to clarify the sequence of actions that take place in the simulation of this system and to be able to explain the subsequent analysis and algorithm, we will establish the

following terms. At each M_{ss}, *Model 1* generates a set of outputs that should be sent over the DCP channel (though the developed DCP library blocks) to *Model 2*, running into the other simulator. We call e_{M1} this event. Similarly, we call e_{M2} the dual event that is the generation of data by *Model 2* that is to be sent to *Model 1*. We call e_{M2_def} the event corresponding to the generation of outputs in *Model 2* if no synchronisation mechanism is applied. As it explained later in Section IV-B, this event is delayed in order to achieve synchronisation. The transmission time between the two simulators is a random variable τ_T. We denote by t_T the transmission time expressed as multiple of S_{ss}, that is $t_T = \lceil \tau_T / S_{ss} \rceil$. The fact that the communication takes an aleatory, although finite, time together with the fact that the two simulations starts at arbitrary times are the causes of the lack of synchronisation between the two simulations.

A. SYNCHRONISATION ISSUES ANALYSIS

After multiple executions of a system that meets aforementioned assumptions, we saw that the synchronization error came from the fact that, for a correct operation of the system, each of the subsystems must receive an input value before generating the output value. If at some point it happens that an output is generated without having received a new input value, the whole system is destabilised and do not behave identically. Concretely, we noticed that the simulation of the system do not always start in the same way, i.e, randomly, one of the subsystems is started before the other. Additionally, the time from the start of one, to the start of the other, was also variable. This was particularly destabilising when the PI subsystem started running first, as the time response to a step of a PI control tends towards infinity [43].

It is worth mentioning that this problem will only arise on DCP's SRT and HRT operation modes, as NRT operates independently from absolute time and requires the use of external edge signals, i.e. STC_do_step kind PDUs, to work. Therefore this analysis will be carried out considering that both subsystems work in SRT or HRT operation modes. Additionally, we have to consider that if one of the systems cannot work on HRT, even if it works on SRT, the events may not arrive periodically, which would cause an unpredictable variation in the response of the system.

Therefore, in order to synchronise the simulation of a closed-loop system, we have identified three issues to be addressed:

- The simulation step sizes in Simulink should be tied to the CPU clock time.
- The arrival period of PDUs is not deterministic, but it varies randomly.
- The slaves are not initialised at the same time. Randomly, one starts to simulate before the other.

To tackle the first issue, we developed a Simulink block that forces simulation steps to coincide with computer time. This block is implemented in C as an S-Function that calls the underlying OS API. An alternative solution consists in either using the Simulink Desktop Real-Time or in running the simulation on a Simulink Real-Time target (e.g. a Speedgoat). Anyway, both solutions require extra licensing or specific hardware. This block cannot operate at times shorter than 1 ms. Thus the lower bound on S_{ss} that we can achieve with the proposed DCP library is 1 ms. Notice that locking the simulation time to the computer time is a requirement of DCP when it is configured to work in SRT or HRT operation modes.

The second problem is related to the UDP communication protocol, that is not a real-time protocol. Since we do not assume the availability of any mechanism to enforce real-time communication over UDP, it is not possible to ensure that messages are received at constant intervals. Figure 5 shows an example of the effects of the random communication time t_T on the exchange of events e_{M1} and e_{M2} during a simulation. In this example, e_{M1} takes a time $t_T = 2$ to pass from *system 1* to *system 2*, while the dual communication, that is e_{M2} from *system 2* to *system 1*, takes $t_T = 3$. Therefore, as the precise time of receiving cannot be known, we concluded that to overcome this problem, we should set two step sizes, one for the model (M_{ss}) and the other for the simulation (S_{ss}), making S_{ss} significantly smaller than M_{ss}. With this in mind, in the simulation process we have tested several different step size configurations, mentioning the most relevant ones in Section VI-B.

Finally, the third issue is related to the fact that the DCP does not have a mechanism to synchronise the beginning of the simulations; see *3.1.13 Timing, p. 14,* of the specification. Concretely, the problem is that even if the simulation step sizes (S_{ss}) of both simulations are synchronised, the execution steps of the models are not. For instance, if we look at the example in Figure 5, between the events e_{M1} and e_{M2_def} there are 8 steps, however this value changes from one simulation to another. That is to say, in each simulation, the difference in steps between the execution of one model and the other varies. This prevent the simulations from starting in the same way, thus the transient response of the system varied significantly from run to run.

B. SYNCHRONISATION MECHANISM

In order to ensure a deterministic start, the synchronisation block presented in Section III-E has been developed. This block was created on the basis of the following assumptions:

- The models to be synchronised must operate at the same step size M_{ss}.
- The simulation step size S_{ss} must be smaller than that of the model M_{ss}, i.e. $S_{ss} < M_{ss}$.
- The *Slave synchronisation* block manages the simulation period of each model. Therefore, any model connected to the *DCP slave* block must have an enable input signal, which will be connected to an enable output of this block.
- The models under co-simulation compose a closed-loop system, where only two slaves are involved. For instance, an example case would be the simulation of a plant and a control system. Although this point limits

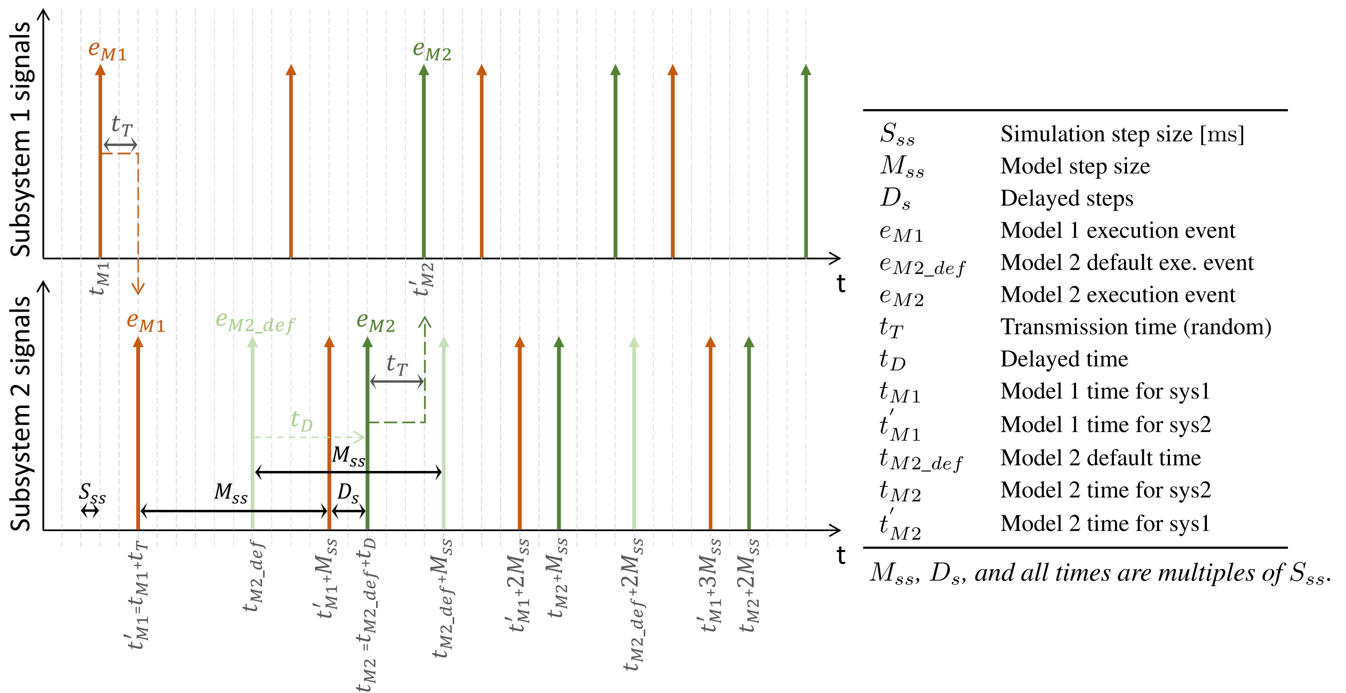


FIGURE 5. **Developed synchronisation mechanism.**

the scope of the synchronisation block, it is required to make the synchronisation problem treatable.

It should be noted that *system 1* refers to the control system and *system 2* to the plant, and that the *Slave synchronisation* block is located in the last one.

With this in mind, this block was designed to be placed only in one of the subsystems, in this case in the plant modelling environment, i.e. Simulink. Its interface consists of one input, two outputs and three parameters. The input is the structure *outVars* received through DCP (see Table 2), that contains the output data coming from the other subsystem and an enable signal that indicates when a new data is available. The outputs consist of both models enable signals. Finally, the parameters are the simulation step size (S_{ss}), the model step size (M_{ss}), and the step delay (D_s). Both step sizes (S_{ss} and M_{ss}) are known parameters of the simulation, whereas D_s is set by the user to control the synchronisation. D_s is used to set the number of steps to delay e_{M2} with respect to e_{M1} and it is determined by trial and error. The synchronization mechanism is based on delaying the execution of the subsystem where this block is placed. The block works as follows:

1) It disables the simulation of the plant model, while enabling the control model.
2) After receiving the first control signal, based on the configured D_s value, it calculates how many steps the plant's response needs to be delayed.
3) It modifies the plant's simulation steps to be synchronised to the control's ones.

This functionality is graphically depicted in the example of Figure 5. In order to explain the synchronisation mechanism in more detail, we will analyse this figure from the *system 2* perspective. S_{ss} and M_{ss} are known from the beginning, while the time at which the input signals from the other system (e_{M1}) will be received is unknown. Taking into account that each system expects to receive a data before generating a response, in the situation described, it is possible that *System 2* receives e_{M1} at a moment when it does not have enough time to process it and send the response $e_{M2_{def}}$ before *System 1* starts a new e_{M1} execution step. Thanks to this block, it is possible to delay the execution instant of the *Model 2* by D_s number of steps after the reception of e_{M1}, i.e., delay it from $E_{M2_{def}}$ to E_{M2}. This synchronisation mechanism implies that the *System 1* will execute the two first steps consecutively without receiving any input. Therefore, it is necessary to assess whether this is a problem for the specific use case.

V. COMPLIANCE WITH DCP

This section comments on the identified limitations, unfinished parts and other untested aspects of the proposed implementation.

Regarding the specification, the functionality of the Non-RealTime (NRT) superstate has not been developed, however, as it can be seen in Figure 3, dummy states have been created to ease its future implementation. For this reason the DCP will only be able to work in SoftRealTime (SRT) or HardRealTime (HRT) modes. Automatic simulation restart after an error has not been implemented either. However, to perform this task, the involvement of the master is required, and the

version provided in the C++ example [15] that we used in this work does not support it. Therefore, in order to run a simulation after an error occurred, the slave has to be stopped and restarted. Contrarily, it is possible to restart a simulation after the previous one has been successfully completed.

On the other hand, there are aspects included in the specification that have been modelled but not tested, such as the correct processing of some configuration PDUs (CFG): CFG_parameter, CFG_param_network, CFG_logging and CFG_clear; and the information PDUs (INF): INF_log and INF_state.

Concerning the connectivity, at the moment, a slave can only communicate with a second slave. This is because the PDU transmission block that it was created (see Section III-C) is not prepared for more. Moreover, the Simulink library blocks do not allow to configure the IP addresses during the simulation, but they have to be configured manually before running the simulation. Anyway, although the use of the network configuration PDUs (CFG_source_network_information and CFG_target_network_information) is not necessary at this time, they have been been implemented and tested to comply with the standard.

Apart from these limitations, we checked that our implementation of the standard is correct by testing it in a co-simulation with the DCP slave and the DCP master provided by the Modelica Association.

VI. PROOF-OF-CONCEPT

To test this implementation, we decided to use the interface in a simulation of a closed-loop system, composed of two subsystems: a plant and a PID control algorithm, as shown in Figure 6. The main idea is to partition the system by simulating each subsystem in a simulation environment and check that the use of the interface does not affect the behaviour of the overall system. For this purpose, it has been decided to perform five different experimental setups of the simulation with increased complexity in terms of subsystem partitioning, that are detailed in Section VI-A. After that, the results of simulating each of these experimental setups with different step sizes will be shown in Section VI-B. It is worth mentioning that, as our objective is to test the data flow between different systems, we did not consider necessary to use a more complex use case.

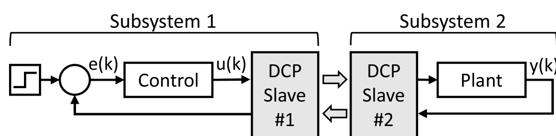


FIGURE 6. Use case: a closed-loop system composed of a control system and a plant, interfaced by the DCP slaves that have been developed.

Regarding the closed-loop system, we employed a simple PI algorithm to control the plant, that it is modeled as a first-order time discrete dynamical system (see Equation 1). The PI control was modeled by a *Discrete PID Controller* Simulink block (see Equation 2). It is worth to mention that, the configuration of the plant model has been maintained constant in all experimental setups. In contrast, the tuning of the PI has been modified accordingly to the variation of the time step, in order to obtain similar dynamical behaviour of the closed-loop system in all the test setups. The analysis of the simulation is done by observing the time evolution of the response of the closed-loop system to a step input, comparing both the transient and the steady-state part of the response.

$$y(k) = a \cdot y(k-1) + b \cdot u(k) \tag{1}$$

where:

- u is the control signal (see Figure 6).
- y is the plant output (see Figure 6).
- a is a constant parameter, and it was permanently set to $b = 0.99$.
- b is a constant parameter, and it was permanently set to $a = 0.01$.

$$u(k) = \left[K_p + K_i T_s \frac{1}{z-1}\right] e(k) \tag{2}$$

where:

- u is the control signal (see Figure 6).
- K_p is the proportional gain coefficient.
- K_i is the integral gain coefficient.
- T_s is the sampling period, which must coincide with the model sample size (i.e.: $T_s = M_{ss}$) introduced in Section III-E.
- $e(k) = r(k) - y(k)$ is the error signal (see Figure 6).
- $r(k)$ is the target reference signal.
- z is the unit delay operator.

The experimental setups differs for simulation environments, as will be explained in Section VI-A. In particular, we employed three environments: simulations in Simulink on a standard PC, C++ program running directly on an Ubuntu 18.04 operating system (OS) and simulations in an embedded platform. Indeed, the control was deployed in a Xilinx Zynq Ultrascale+ MPSoC ZCU102 Evaluation Kit. The general architecture of this device consists of an ARM-based CPU, called Processing System (PS), and a FPGA-like Programmable Logic (PL). Both the PS and the PL can be used independently or together. The PS features the ARM Cortex-A53 64-bit quad-core and ARM Cortex-R5 dual-core real-time processing units, which are capable of supporting an operating system such as Ubuntu. Whereas the PL, is based on Xilinx's programmable logic fabric, which has been designed to implement high-speed logic, solve arithmetic operations and perform data exchange between subsystems. In this work we only exploited the ARM Cortex-A53, were we deployed an Ubuntu 18.04 OS.

To perform the (co)-simulations between these subsystems, three DCP entities have been used: a master and two slaves. They have been configured to work in SRT operation mode. The master is an adaptation of the one provided in the C++ example [15] and will be in charge of coordinating the (co)-simulations. While the slaves, depending on their

location, will be those developed in this work (Section III-A), if the subsystem is implemented in Simulink; or a modification of the one provided in the C++ example, if the subsystem is implemented in an Ubuntu operating system. These slaves will take care of transmitting the data between the two subsystems we have just described.

A. EXPERIMENTAL SETUPS

This section provides an explanation about the developed experimental setups. From a general point of view, the three entities that compose the co-simulation environment are distributed as follows. The plant and the DCP slave linked to it (the subsystem 2 of Figure 6) have been placed, in all setups, in Simulink. The master has been placed, in all setups, in a virtual machine where an Ubuntu 18.04 is deployed. Finally, the control algorithm and the DCP slave linked to it (the subsystem 1 of Figure 6) have been placed in a different environment in each experimental setup. The experimental setups are built incrementally, by increasing the complexity of the topology of the simulations.

Concerning the physical communication network, Ethernet is employed to link the three entities. Simulink and the Ubuntu virtual machine are located in the same PC, so they communicate each other via virtual network ports. Whereas a direct Ethernet cable is used to connect the PC and the Xilinx Ultrascale board.

1) REFERENCE SETUP - REFERENCE CLOSED-LOOP SYSTEM

The closed-loop system is implemented entirely in Simulink without using any DCP slaves. The results obtained from this setup will be used as a baseline reference to be compared with the other experimental setups. Ideally, all the experimental setup should provide results identical to the baseline. See Figure 8 for a schematic description of this setup.

2) EXPERIMENTAL SETUP I - PLANT AND CONTROL ALGORITHM IN THE SAME SIMULINK MODEL

A DCP communication is introduced into the Reference Setup while keeping the plant and the control system in the same Simulink model. To do this, the two subsystems are separated and two DCP slaves are introduced in the Simulink model, each connected to the input and output ports of each subsystem. To coordinate the simulation, the DCP master is configured in the Ubuntu virtual machine. In this way, the communication between the two subsystems is handled by the DCP. See Figure 9 for a schematic description of this experimental setup.

3) EXPERIMENTAL SETUP II - PLANT AND CONTROL ALGORITHM IN DIFFERENT SIMULINK MODELS

Each subsystem is isolated in a separate Simulink simulation. That is the same environment execute two simulations in parallel. This experimental setup uses the same master as the previous one. In order to run the complete system, it is necessary to initialize two instances of Simulink and launch two separate simulation at the same time. See Figure 10 for a schematic description of this experimental setup.

4) EXPERIMENTAL SETUP III - PLANT IN SIMULINK AND CONTROL ALGORITHM IN AN UBUNTU VIRTUAL MACHINE

The plant model is kept in Simulink and the control system is deployed into the same virtual machine where the master is located. As the virtual machine uses the DCP library developed in [15], we developed a C++ version of the control algorithm that behaves in the same way as the Simulink algorithm. To do this, we used the Simulink Coder tool to automatically generate the C++ code of the control algorithm from the control system model. This code was then embedded in the slave provided by the library, making for this purpose additional modifications in the slave configuration. Finally, the master was modified to indicate the new location of the slave 1. See Figure 11 for a schematic description of this experimental setup.

5) EXPERIMENTAL SETUP IV - PLANT IN SIMULINK AND CONTROL ALGORITHM IN THE PS OF AN UltraScale+

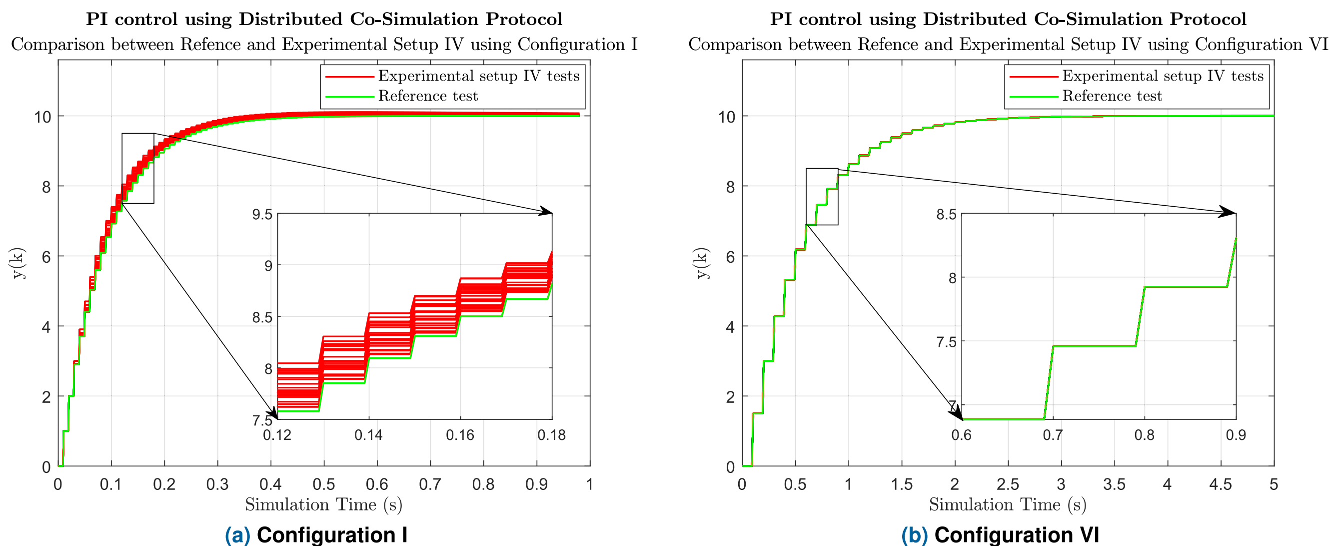
The master is kept in the virtual machine and the plant subsystem in Simulink, whereas, the control subsystem is moved to a Xilinx Zynq Ultrascale+ board. Concretely, it has been located in the PS part of the board, where an embedded Ubuntu system has been deployed. The only limitation found in this experimental setup is that it was not possible to work with 64-bit pointers. Therefore, some minor adjustments have had to be made to both the provided C++ DCP library [15] and the control algorithm. The master has also had to be modified to indicate the new location of slave 1. See Figure 12 for a schematic description of this experimental setup.

This experimental setup is the most interesting, as it opens up a way to communicate in the future with models that are deployed on SoC platforms. Either to speed up a simulation or because they are computationally expensive to run on conventional PC hardware. It also offers the possibility to perform a PIL (or FIL) without the need for an external tool.

B. SIMULATION CONFIGURATIONS AND OBTAINED RESULTS

In order to know the maximum speed at which it is possible to communicate using the DCP, it has been decided to run each of the previous experimental setups with different step sizes. As explained in Section IV, before setting the configurations we have to take into account that: the simulation step of the simulation run (S_{ss}) must be lower than that of the model (M_{ss}), and that S_{ss} cannot be smaller than 1ms. This section presents and analyses the results obtained after simulating the experimental setups with the configurations in Table 5. It should be noted that the reference value for $y(k)$ is set to 10 for all simulations, i.e., $r(k) = 10$.

The same procedure has been followed for all experimental setups with each configuration. First, the step size of the model and of the simulation environment were set, and the PI control was tuned. Second, 25 simulations were run. Finally,



(a) Configuration I

(b) Configuration VI

FIGURE 7. Comparison between reference setup and experimental Setup IV.

TABLE 5. Configurations for the simulations.

Configuration I	$S_{ss} = 1ms$ $M_{ss} = 10ms$ $D_s = 2$ steps	$K_p = 10$ $K_i = 10$
Configuration II	$S_{ss} = 1ms$ $M_{ss} = 20ms$ $D_s = 2$ steps	$K_p = 20$ $K_i = 10$
Configuration III	$S_{ss} = 1ms$ $M_{ss} = 20ms$ $D_s = 2$ steps	$K_p = 20$ $K_i = 9.2$
Configuration IV	$S_{ss} = 2ms$ $M_{ss} = 20ms$ $D_s = 2$ steps	$K_p = 20$ $K_i = 10$
Configuration V	$S_{ss} = 4ms$ $M_{ss} = 40ms$ $D_s = 2$ steps	$K_p = 24$ $K_i = 6$
Configuration VI	$S_{ss} = 10ms$ $M_{ss} = 100ms$ $D_s = 2$ steps	$K_p = 15$ $K_i = 1.5$

the time behaviour of the output $y(k)$ of the closed-loop system was obtained from the log files. Figure 7 compares the plots of $y(k)$ in the reference setup (green lines) with the experimental setup IV (red lines). The first graph Figure 7a shows this comparison when Configuration I is set, while the second Figure 7b corresponds to Configuration VI. For the sake of compactness, we decided to show only these results graphically, indeed, notice that experimental setup IV is the most interesting with a view to the future and also the most challenging to implement. There are 25 red lines in each graph, corresponding to the 25 repetitions that were executed for each configuration. In Figure 7b this lines are not appreciated since all simulations behaved identically. Therefore, from Figure 7 two main observations can be made: i) the higher S_{ss}, the closer the tests are to the reference response; and ii) the response of the system varies only in the transient state, as the error in the steady state seems insignificant.

To explain these findings thoroughly, a more detailed analysis of the response $y(k)$ can be seen in Table 6, where the error between the reference setup and each of the other experimental setups is shown. The error is calculated as the mean square deviation from the response of the reference setup $y^{ref}(k)$, as shown in Equation 3.

$$err = \sqrt{\frac{1}{N}\sum_{k=1}^{N}\left[y(k) - y^{ref}(k)\right]^2} \quad (3)$$

where N is the number of simulations steps, $y(k)$ is the response of the closed-loop system at step k under a particular setup, $y^{ref}(k)$ is the response of the closed-loop system under the reference setup. The data for this analysis were obtained by running 25 repeated simulations for each combination of experimental setup and configuration. Table 6 reports the mean, the minimum, the maximum and the standard deviation of the error over the 25 repetitions.

Before discussing the results in Table 6, it should be noted that each configuration represents a slightly different dynamical system, and so comparisons between them should be made with care. Moreover, in close-loop systems, synchronisation errors may cause the response to overshoot the setpoint value due to the computational differences they induce. This results in a significant increase in the mean, maximum and standard deviation of the error. In order to keep the difference between configurations to a minimum, the PI has been tuned to obtain a robust closed-loop behaviour. Indeed, since we are working with discrete time, changing the execution times

(S_{ss} and M_{ss}) also changes the dynamics of the system. Therefore, the controller must be retuned. Additionally, as a synchronisation failure can also produce an overshoot, instead of looking for the fastest possible response, we have chosen to tune the control in such a way that it produces a slightly slower response, thus ensuring more similar results. In this way, we can ensure a correct qualitative comparison between the different configurations. On the contrary, when comparing the experimental setups of the same configuration, as the system remains identical (same control tunning and same step-sizes), both a quantitative and a qualitative comparison can be made. Additionally, it is worth to point out some obvious but important aspects about the mean, maximum, minimum, and standard deviation indicators. The lower the mean value, the more similar the responses will be to those of the reference setup. The differences registered in the time evolution of $y(k)$ are greater in the transitory than in the steady-state; therefore, an error appearing in the firsts simulation steps causes a greater deviation from the reference. If the minimum value is 0, it means that at least the response of one of the simulations is identical to that of the reference setup. If the maximum value is 0, it means that all simulations produced exactly the same results as the reference setup.

By looking at Table 6 and by taking each configuration individually, comparing the mean errors of the experimental setups, we can notice a main common behaviour: the mean error in experimental setup II is significantly larger than that of the other setups. Moreover, if we compare the mean error of experimental setup II across all the configurations (i.e., the second column of Table 6), we cannot obtain any logical tendency. This behaviour may be caused by the fact that running two instances of Simulink, and a virtual machine, may require too many computational resources from the PC. Therefore, for the rest of the analysis we will not take into account the results of the experimental setup II.

TABLE 6. Error analysis.

Configuration	Experimental Setup I	II	III	IV
I	mean: 0.0368 max: 0.1810 min: 0 sd: 0.0736	mean: 0.1674 max: 0.3263 min: 0.0737 sd: 0.0585	mean: 0.0854 max: 0.1466 min: 0.0399 sd: 0.0328	mean: 0.0951 max: 0.1739 min: 0.0117 sd: 0.0421
II	mean: 0.0032 max: 0.0774 min: 0 sd: 0.0155	mean: 0.1905 max: 0.3872 min: 0.0338 sd: 0.0895	mean: 0.0533 max: 0.1655 min: 0 sd: 0.0557	mean: 0.0709 max: 0.2189 min: 0 sd: 0.0634
III	mean: 0.0032 max: 0.0788 min: 0 sd: 0.0157	mean: 0.1541 max: 0.3579 min: 0.0127 sd: 0.1005	mean: 0.0558 max: 0.2177 min: 0 sd: 0.0690	mean: 0.0670 max: 0.2441 min: 0 sd: 0.0740
IV	mean: 0.0044 max: 0.1087 min: 0 sd: 0.0217	mean: 0.2295 max: 0.3919 min: 0.0524 sd: 0.0981	mean: 0.0241 max: 0.1412 min: 0 sd: 0.0381	mean: 0.0231 max: 0.1976 min: 0 sd: 0.0493
V	mean: 0.0001 max: 0.0021 min: 0 sd: 0.0004	mean: 0.1404 max: 0.3298 min: 0 sd: 0.1248	mean: 0.0027 max: 0.0668 min: 0 sd: 0.0134	mean: 0.0028 max: 0.0690 min: 0 sd: 0.0138
VI	mean: 0.0033 max: 0.0813 min: 0 sd: 0.0163	mean: 0.1302 max: 0.3382 min: 0 sd: 0.1489	mean: 0.0033 max: 0.0664 min: 0 sd: 0.0132	mean: 0 max: 0 min: 0 sd: 0

By analysing the results for each configuration, in other words, by comparing the values of each row independently, we can observe three different behaviours:

- In **configurations I, II, and III**, the mean error of experimental setup I is the closest to the reference setup, while the mean error of experimental setup IV is the largest one. It should be noted that although the mean error of experimental setup III is a value between the previous ones, it is only slightly lower to that of experimental setup IV. Additionally, it can be observed that the mean error of experimental setup I in configurations I and II is almost null, which means that their response $y(k)$ is almost identical to that of the reference setup.
- In **configuration IV**, the mean error of experimental setup I is again the lowest. Moreover, as it remains almost null, we can also say that the response of experimental setup I is almost identical to that of the reference setup. On the other hand, the mean errors of experimental setups III and IV become similar. As this error is quite low, this means that the response of both experimental setups is also similar.
- In **configurations V and VI**, the mean error of all experimental setups is very similar and almost null. Therefore we can conclude that the response in almost all simulations will be identical to that of the reference.

From these tests we can extrapolate some general consideration about the DCP when simulating closed-loop systems. Generally speaking, Experimental Setup I shows better performances in terms of error compared to setups III and IV in almost all configurations. This is reasonable, since setups III and IV are more complex from a communication topology point of view. In other words, since in experimental setup I the entire system is simulated in a single Simulink instance, there are few factors that can cause communication errors. Indeed, the main factor is that Simulink is not a real-time environment, thus an exactly periodic exchange of data cannot be guaranteed. However, if we deploy one of the subsystems to another simulation environment, such as the Ubuntu O.S or the Xilinx Zynq Ultrascale+, the communication becomes more error prone.

Observing the experimental setups independently, in other words comparing the values of each column, the following is evidenced:

- In **experimental setup I**, if $M_{ss} > 20ms$ the mean error is almost null.
- In **experimental setups III and IV**, the higher S_{ss}, the closer the tests are to the reference response. Moreover, with the same S_{ss}, the higher M_{ss}, the closer the tests are

to the reference response. Additionally, if S_{ss} and M_{ss} are maintained and the system is modified, by changing the parameters K_p and K_i of the PI (see configurations II and III), the mean errors remain similar. This implies that the behaviour of the communication will be the same no matter of the system that is being simulated.

- For **all experimental setups**, if $S_{ss} \geq 4ms$, the mean error is almost null.

We see that if S_{ss} is increased, the difference between the responses of the experimental setups is reduced. In fact, when $S_{ss} > 4ms$ the mean error is almost null, hence it can be said that the response of all simulations are identical to that of the reference. To be more precise, the difference lies in the fact that some independent step arrives a little late, however this is not something that affects the system's behaviour. All in all, as we already mentioned commenting Figure 7, almost all of the error is due to differences in the transient state, while the error in the steady state is zero or minimal.

VII. CONCLUSION AND FUTURE DEVELOPMENTS

This paper presents an implementation of DCP developed in Simulink, which allows to perform a co-simulation between a Simulink model and other M&S environment. Until now, the connection to Simulink was only available through third party tools, so in addition to offering direct integration into Simulink, we have also extended the scope of the DCP. Moreover, thanks to the code generation tools provided by Simulink, the proposed implementation of DCP open the gates to deployments in other environments and hardware devices, thus offering new possibilities to perform distributed co-simulations. That is, by putting hardware elements into the co-simulation, we provide the possibility to speed up simulations or to perform x-in-the-loop simulations. Furthermore, the DCP was designed with the aim of encapsulating a real-time model/system and to link it with another device; however, we have shown that it is possible to use it as a general communication mechanism, independent from the model/system to be simulated. This provide extra degrees of flexibility to perform co-simulations since it separates the modelling activities from the technical details of the co-simulation. All of this has been demonstrated in a use case, especially in the experimental setup IV.

Apart from these benefits, we have also shown that the lack of synchronisation can cause undesired behaviours in some simulations. We have proposed a synchronisation method that allows all subsystems that are involved in the simulation to start at the same instant.

Looking to extend our work in the future, it would be interesting to add additional slaves to the co-simulation, to read the configuration from an XML file (DCPX), and to implement the NRT usage mode. Regarding the synchronisation mechanism, we intend to apply some improvements in order to be able to work with slaves that work different step sizes. In addition, it would also be interesting to compare it with other synchronisation mechanisms, thus analysing if there is any way to improve the results obtained in Table 6. In relation to this table, it would also be interesting to know the reliability of the results, e.g. how many simulation runs of the same scenario are necessary to be confident enough. On the other hand, we would like to port DCP to other hardware platforms, for instance based on the RISC-V architecture, as it is an emerging option in the research community. Finally, it would be interesting to use DCP as middle layer to implement FIL simulations, possibly comparing it to proprietary solutions, such as the one provided by Mathworks. Indeed, we have implemented the control algorithm on a platform such as the Xilinx Ultrascale with the intention of a future deployment of the plant model on the PL, that is the FPGA-like programmable logic.

APPENDIX A
GRAPHICAL DESCRIPTION OF PROOF-OF-CONCEPT EXPERIMENTAL SETUPS

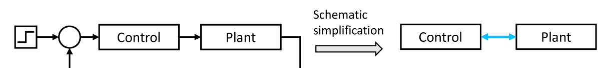


FIGURE 8. Reference Setup: Reference closed-loop system.

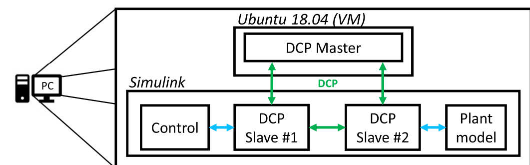


FIGURE 9. Experimental Setup I: Plant and control algorithm in the same Simulink model.

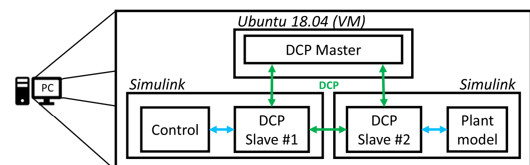


FIGURE 10. Experimental Setup II: Plant and control algorithm in different Simulink models.

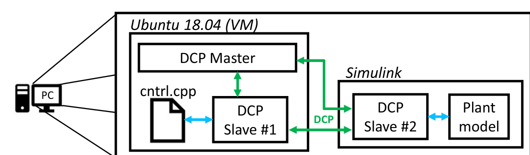


FIGURE 11. Experimental Setup III: Plant in Simulink and control algorithm in an Ubuntu virtual machine.

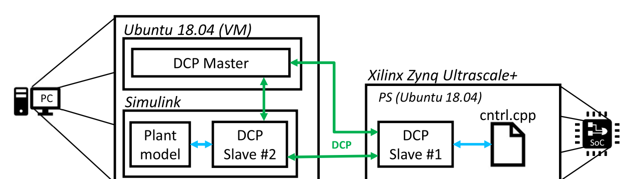


FIGURE 12. Experimental Setup IV: Plant in Simulink and control algorithm in the PS of an UltraScale+.

REFERENCES

[1] P. Marwedel, *Embedded System Design-Embedded Systems Foundations of Cyber-Physical Systems, and the Internet of Things*, 4th ed. Berlin, Germany: Springer, 2021, ch. 1.

[2] T. Pieper, ''Distributed co-simulation framework for hardware- and software-in-the-loop testing of networked embedded real-time systems,'' Ph. D. thesis, Faculty Natural Sci. Eng., Institut für Praktische und Technische Informatik, Univ. Siegen, Siegen, Germany, 2020.

[3] C. Köhler, *Enhancing Embedded Systems Simulation*, 1st ed. Germany: Vieweg+Teubner Verlag, 2011, ch. 2.

[4] I. Graja, S. Kallel, N. Guermouche, S. Cheikhrouhou, and A. H. Kacem, ''A comprehensive survey on modeling of cyber-physical systems,'' *Concurrency Comput., Pract. Exper.*, vol. 32, no. 15, pp. 1–18, 2020.

[5] S. K. Khaitan and J. D. McCalley, ''Design techniques and applications of cyberphysical systems: A survey,'' *IEEE Syst. J.*, vol. 9, no. 2, pp. 350–365, Jun. 2014.

[6] M. Faruque, V. Dinavahi, M. Steurer, A. Monti, K. Strunz, J. A. Martinez, G. W. Chang, J. Jatskevich, R. Iravani, and A. Davoudi, ''Interfacing issues in multi-domain simulation tools,'' *IEEE Trans. Power Del.*, vol. 27, no. 1, pp. 439–448, Jan. 2012.

[7] W. Böhm, M. Broy, C. Klein, K. Pohl, B. Rumpe, and S. Schröck, *Model-Based Engineering of Collaborative Embedded Systems*, 1st ed. Berlin, Germany: Springer, 2021, ch 12&13.

[8] Modelica Association. *Distributed Co-Simulation Protocol (DCP) Website*. Accessed: Sep. 2022. [Online]. Available: https://dcp-standard.org/

[9] M. Segura, T. Poggi, and R. Barcena, ''Towards the implementation of a real-time co-simulation architecture based on distributed co-simulation protocol,'' in *Proc. 35th Annu. Eur. Simul. Model. Conf.*, 2021, pp. 155–162.

[10] N. Li, L. Zhao, C. Bao, G. Gong, X. Song, and C. Tian, ''A real-time information integration framework for multidisciplinary coupling of complex aircrafts: An application of IIIE,'' *J. Ind. Inf. Integr.*, vol. 22, Jun. 2021, Art. no. 100203.

[11] S. Klein, F. Xia, K. Etzold, J. Andert, N. Amringer, S. Walter, T. Blochwitz, and C. Bellanger, ''Testing and calibration of hybrid power testing and calibration of hybrid power trains,'' *IFAC-PapersOnLine*, vol. 51, pp. 240–245, Jan. 2018.

[12] M.-A. Meyer, L. Sauter, C. Granrath, H. Hadj-Amor, and J. Andert, ''Simulator coupled with distributed co-simulation protocol for automated driving tests,'' *Automot. Innov.*, vol. 4, no. 4, pp. 373–389, Nov. 2021.

[13] P. Baumann, M. Krammer, M. Driussi, L. Mikelsons, J. Zehetner, W. Mair, and D. Schramm, ''Using the distributed co-simulation protocol for a mixed real-virtual prototype,'' in *Proc. IEEE Int. Conf. Mechatronics (ICM)*, Mar. 2019, pp. 440–445.

[14] M. Krammer, C. Kater, C. Schiffer, and M. Benedikt, ''A protocol-based verification approach for standard-compliant distributed a protocol-based verification approach for standard-compliant distributed co-simulation,'' in *Proc. Asian Modelica Conf.*, 2020, pp. 1–10.

[15] Modelica Association. *DCP Library*. Accessed: Sep. 2022. [Online]. Available: https://github.com/modelica/DCPLib

[16] K. Wang, Z. Gong, Y. Hou, M. Zhang, C. Liu, and R. Chen, ''Model based design and procedure of flight control system for unmanned aerial vehicle,'' in *Proc. 3rd Int. Conf. Unmanned Syst. (ICUS)*, Nov. 2020, pp. 763–768.

[17] R. Aarenstrup, *Managing Model-Based Design*. Natick, MA, USA: The MathWorks, 2015.

[18] S. Gautham, A. V. Jayakumar, A. Rajagopala, and C. Elks, ''Realization of a model-based DevOps process for industrial safety critical cyber physical systems,'' in *Proc. 4th IEEE Int. Conf. Ind. Cyber-Phys. Syst. (ICPS)*, May 2021, pp. 597–604.

[19] A. Talaeizadeh, E. Najafi, H. N. Pishkenari, and A. Alasty, ''Deployment of model-based design approach for a mini—Quadcopter,'' in *Proc. 7th Int. Conf. Robot. Mechatronics (ICRoM)*, Nov. 2019, pp. 291–296.

[20] C. Gomes, C. Thule, D. Broman, P. G. Larsen, and H. Vangheluwe, ''Co-simulation: State of the art,'' 2017, *arXiv:1702.00686*.

[21] V. Schreiber, V. Ivanov, K. Augsburg, M. Noack, B. Shyrokau, C. Sandu, and P. S. Els, ''Shared and distributed X-in-the-loop tests for automotive systems: Feasibility study,'' *IEEE Access*, vol. 6, pp. 4017–4026, 2018.

[22] C. Shum, W.-H. Lau, T. Mao, H. S.-H. Chung, K.-F. Tsang, N. C.-F. Tse, and L. L. Lai, ''Co-simulation of distributed smart grid software using direct-execution simulation,'' *IEEE Access*, vol. 6, pp. 20531–20544, 2018.

[23] L. I. Hatledal, A. Styve, G. Hovland, and H. Zhang, ''A language and platform independent co-simulation framework based on the functional mock-up interface,'' *IEEE Access*, vol. 7, pp. 109328–109339, 2019.

[24] J. Mina, Z. Flores, E. Lopez, A. Perez, and J.-H. Calleja, ''Processor-in-the-loop and hardware-in-the-loop simulation of electric systems based in FPGA,'' in *Proc. 13th Int. Conf. Power Electron. (CIEP)*, Jun. 2016, pp. 172–177.

[25] A. Etxebarria, R. Barcena, and I. Mancisidor, ''Active control of regenerative chatter in turning by compensating the variable cutting force,'' *IEEE Access*, vol. 8, pp. 224006–224019, 2020.

[26] G. C. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Alphen aan den Rijn, The Netherlands: Kluwer Academic, 1997.

[27] N. Brayanov and A. Stoynova, ''Review of hardware-in-the-loop—A hundred years progress in the pseudo-real testing,'' *Electrotechnica Electronica*, vol. 54, nos. 3–4, pp. 70–84, 2019.

[28] L. Ibarra, A. Rosales, P. Ponce, A. Molina, and R. Ayyanar, ''Overview of real-time simulation as a supporting effort to smart-grid attainment,'' *Energies*, vol. 10, no. 6, pp. 1–24, 2017.

[29] S. S. Noureen, N. Shamim, V. Roy, and S. B. Bayne, ''Real-time digital simulators: A comprehensive study on system overview, application, and importance,'' *Int. J. Res. Eng.*, vol. 4, no. 11, pp. 266–277, Dec. 2017.

[30] A. Rothstein, T. Stoetzel, V. Staudt, and J. Wiesemann, ''Asymmetric multiprocessing: A promising option for SoC-based real-time control in power electronics,'' in *Proc. 11th IEEE Int. Conf. Compat., Power Electron. Power Eng. (CPE-POWERENG)*, Apr. 2017, pp. 666–670.

[31] A. Falcone and A. Garro, ''Distributed co-simulation of complex engineered systems by combining the high level architecture and functional mock-up interface,'' *Simul. Model. Pract. Theory*, vol. 97, Dec. 2019, Art. no. 101967.

[32] H. Neema, J. Gohl, Z. Lattmann, J. Sztipanovits, G. Karsai, S. Neema, T. Bapty, J. Batteh, H. Tummescheit, and C. Sureshkumar, ''Model-based integration platform for FMI co-simulation and heterogeneous simulations of cyber-physical systems,'' in *Proc. Linköping Electron. Conf.*, Mar. 2014, pp. 235–245.

[33] T. Jung, P. Shah, and M. Weyrich, ''Dynamic co-simulation of Internet-of-Things-components using a multi-agent-system,'' *Proc. CIRP*, vol. 72, pp. 874–879, Jan. 2018.

[34] G. Schweiger, C. Gomes, G. Engel, I. Hafner, J. Schoeggl, A. Posch, and T. Nouidui, ''An empirical survey on co-simulation: Promising standards, challenges and research needs,'' *Simul. Model. Pract. Theory*, vol. 95, pp. 148–163, Sep. 2019.

[35] Modelica Association. *DCP Standard Specification*. Accessed: Sep. 2022. [Online]. Available: https://github.com/modelica/dcp-standard

[36] M. Krammer, M. B. Msc, T. Blochwitz, K. Alekeish, N. Amringer, C. Kater, S. Materne, R. Ruvalcaba, K. Schuch, J. Zehetner, M. Damm-Norwig, V. Schreiber, N. Nagarajan, I. Corral, T. Sparber, S. Klein, and J. Andert, ''The distributed co-simulation protocol for the integration of real-time systems and simulation environments,'' in *Proc. 50th Comput. Simul. Conf.*, vol. 50, 2018, pp. 1–14.

[37] M. Krammer and M. Benedikt, ''Configuration of slaves based on the distributed co-simulation protocol,'' in *Proc. IEEE 23rd Int. Conf. Emerg. Technol. Factory Autom. (ETFA)*, Sep. 2018, pp. 195–202.

[38] M. Krammer and M. Benedikt, ''Master for simulation control using the distributed co-simulation protocol,'' in *Proc. IEEE 16th Int. Conf. Ind. Informat. (INDIN)*, Jul. 2018, pp. 329–334.

[39] M. Krammer, P. Ferner, and D. Watzenig, ''Clock synchronization in context of the distributed co-simulation protocol,'' in *Proc. IEEE Int. Conf. Connected Vehicles Expo (ICCVE)*, Nov. 2019, pp. 1–6.

[40] M. Krammer, C. Schiffer, and M. Benedikt, ''ProMECoS: A process model for efficient standard-driven distributed co-simulation,'' *Electronics*, vol. 10, no. 5, p.633, 2021.

[41] M. Krammer and M. Benedikt, ''Design and application of a domain specific modeling language for distributed co-simulation,'' in *Proc. IEEE 17th Int. Conf. Ind. Informat. (INDIN)*, Jul. 2019, pp. 677–682.

[42] D. Buck and A. Rau, ''On modelling guidelines: Flowchart patterns for stateflow,'' *Softwaretechnik-Trends*, vol. 21, no. 2, pp. 7–12, 2001.

[43] L. Keviczky, R. Bars, J. Hetthéssy, and C. Bányász, *Control Engineering*. Berlin, Germany: Springer, 2019, ch. 8.

MIKEL SEGURA received the M.S. degree in embedded systems from the University of the Basque Country (UPV-EHU), in 2020, where he is currently pursuing the Ph.D. degree in engineering physics.

In 2019, he joined IKERLAN to develop his master's thesis, where he is still researching as a Ph.D. student. The research focuses on how to perform co-simulations between modeling and simulation environments (e.g. Simulink and Open Modelica) and high-performance hardware platforms using non-proprietary standards. The aim is to ease the communication between the simulation entities and to accelerate simulations through the use of FPGAs.

TOMASO POGGI was born in Varazze, Italy, in 1982. He received the M.Sc. degree in electronic engineering and the Ph.D. degree in mathematical and simulation engineering from the University of Genoa, Italy, in November 2006 and April 2010, respectively.

In 2011, he collaborated with the Department of Biophysical and Electronic Engineering, University of Genoa, as a Research Assistant. From 2011 to 2016, he worked as an Embedded Systems Engineer at the RF Group, ESS-Bilbao, Spain, in the field of light ion linear accelerator. From 2016 to 2022, he worked at IKERLAN, Spain, in the field of dependable embedded systems. He is currently a Professor with the Automation Department, Faculty of Engineering, University of Mondragon, Spain. He is the author or coauthor of 21 scientific papers, published in indexed international journals and conferences. His main research interests include digital signal processing and control and analysis of nonlinear dynamical systems.

RAFAEL BARCENA (Member, IEEE) received the M.S. and Ph.D. degrees in physics from the University of the Basque Country (UPV/EHU), in 1994 and 2001, respectively.

Since 1998, he has been a Research Professor with the Department of Electronic Technology, UPV/EHU. His research interests include control theory, hybrid control, and sampled-data systems.

Dr. Barcena has been a member of the IEEE Control Systems Society and the IEEE Industrial Electronics Society, since 2005.