Testing the verification and validation capability of a DCP based interface for distributed real-time applications

Mikel Segura[1] Alejandro J. Calderón[1] Tomaso Poggi[2] Rafael Barcena[3]

[1]Dependable Embedded Systems Team, IKERLAN, José María Arizmendiarrieta, 2, Arrasate 20500, Basque Country, Spain, {msegura,ajcalderon}@ikerlan.es

[2]Mondragon Unibertsitatea, Loramendi Kalea, 4, Arrasate, 20500, Basque Country, Spain, tpoggi@mondragon.edu

[3]Department of Electronic Technology, University of the Basque Country (UPV-EHU), Torres Quevedo Ingeniariaren Enparantza, 1, Bilbao, 48013, Basque Country, Spain, rafa.barcena@ehu.es

Abstract

Cyber-physical systems are composed of a variety of elements developed by different vendors that are often geographically distributed. Therefore, its development process presents a double challenge: each element has to be developed individually and, at the same time, a correct interaction with the rest of the elements has to be ensured. In a previous work, we proposed and developed an interface, based on the non-proprietary Distributed Co-simulation Protocol standard, to ease the interaction between these elements. In this paper, we improve it to be applicable in a variety of hardware platforms and we test its applicability for the verification and validation process. To do so, firstly, we prove that our interface is hardware agnostic, demonstrating its easy implementation on different platforms. Secondly, we test its applicability in different X-in-the-Loop simulations. Finally, we also test its behaviour in distributed real-time executions, a necessary requirement for linking elements from different suppliers and helping to preserve their Intellectual Property.

Keywords: Simulation interface, Real-time, Intellectual Property protection, Distributed Co-Simulation Protocol, Verification and Validation

1 Introduction

Model-based design (MBD) is a commonly used practice for the development of cyber-physical systems (CPS) (Böhm et al. 2021). This process consists of developing virtual models that reproduce certain behaviours of a real system, thus avoiding the need to create costly physical prototypes and facilitating the process of validation and verification (Marwedel 2021). It is common that these models are located in different modelling and simulation (M&S) environments, either because they have been developed by different vendors and they want to preserve confidentiality (Falcone and Garro 2019), or because it is wanted to implement part of the model on a specific hardware platform in order to verify its performance in a specific environment (Alfalouji et al. 2023). Testing the correct interaction between these elements at an early stage of the development phase facilitates the process of validation and verification of them. However, as stated in (Attarzadeh-Niaki and Sander 2020), it is common to solve the challenge of linking such elements using ad-hoc methods. In (Segura, Poggi, and Barcena 2021) we argue the lack of a language and platform independent co-simulation architecture to address this problem and in (Segura, Poggi, and Barcena 2023) we propose a solution for it, presenting an architecture based on the non-proprietary Distributed Co-Simulation Protocol (DCP) standard. Nevertheless, we did not dive into how it could be deployed on different hardware platforms and thus, demonstrate how the verification and validation process can be simplified. It is worth mentioning that this implementation would save time, resources and money, as it facilitates the coupling of elements, saves displacements for integration testing and helps to preserve Intellectual Property.

Accordingly, this article discusses three points. First, we demonstrate the easy implementation of this interface on a variety of hardware platforms, deploying it in generic but different targeted hardware platforms such as, the Xilinx Zynq UltraScale+, Xilinx Zynq-7000 SoC ZC702, NVIDIA Jetson Nano, and Raspberry-Pi. Second, taking into account that the use of X-in-the-Loop (XIL) simulations is widely extended in the development of CPSs, we analyse the limitations of our interface in a real-time communication between a development software such as Simulink and the platforms mentioned above. Finally, in order to show how our interface can solve the challenge of communicating systems developed by geographically distributed suppliers, we link the Xilinx Zynq-7000 SoC ZC702 with the Raspberry-Pi in a real-time simulation using our interface via UDP communication.

The paper is structured as follows. Section 2 addresses the need for a generic architecture for co-simulation. Section 3 introduces the generic interface that enables performing co-simulations between a variety of simulation environments. Section 4 explains the tests that we executed to demonstrate the applicability of the interface. Section 5 exposes the results of the conducted tests. Section 6 analyses the results of the previous section. Finally,

Section 7 presents the conclusions and future work.

2 Background and Related Work

Co-simulation is used to couple different simulation environments (e.g. a continuous and an event driven simulation environment) in order to use an appropriate simulation environment for each part of the system (Köhler 2011). Co-simulation can also be used to link spatially distributed models (Baumann et al. 2019). Additionally, in the development and verification process of control systems, different co-simulation techniques referred to as X-in-the-Loop (XIL) (Ivanov et al. 2019) are used, encompassing the well-known Model-in-the-Loop (MIL), Software-in-the-Loop (SIL), Processor-in-the-Loop (PIL), or Hardware-in-the-Loop (HIL) techniques. Nevertheless, despite being a widely used technique, coupling problems often arise and there is no a generic methodology for linking different simulation environments. This problem is detected in several works, where different co-simulation architectures are proposed to solve it. For instance, (Hatledal et al. 2019) presents a language- and platform-independent co-simulation framework based on the Functional Mock-up Interface (FMI). Its major drawback is that it has not considered the integration of real-time systems or hardware-in-the-loop simulations. In (Ivanov et al. 2019) the authors propose an architecture applicable in a variety of XIL approaches and it is intended to perform real-time and distributed co-simulations in different geographical locations. However, they use a proprietary architecture that is not enforced by any standard and there is no mention of how it could be implemented in different modelling and simulation environments, which suggests that its integration is not straightforward. The architecture proposed in (Attarzadeh-Niaki and Sander 2020) attempts to avoid ad-hoc approaches and it is focused on guaranteeing IP protection, however, no mention is made on real-time applications.

Some standards also aim to manage distributed co-simulations, such as Distributed Co-Simulation Protocol (DCP) (Modelica Association 2023b). DCP is a non-proprietary standard that aims to integrate real-time systems into co-simulation environments. It follows the master-slave principle and it is independent of the communication medium, as it works over common transport protocols such as Bluetooth, UDP, or CAN. However, as the DCP is a relatively new standard, it has a limited applicability in terms of simulation environments. Furthermore, its operation resides in encapsulating the systems to be communicated, thus a particular DCP slave must be created for each application.

In previous works, we propose (Segura, Poggi, and Barcena 2021) and present (Segura, Poggi, and Barcena 2023), a co-simulation architecture, based on non-proprietary standards, that facilitates coupling between different M&S environments and hardware platforms. By relying on a non-proprietary standard such as DCP, the architecture is implementable without any intellectual property restrictions and on top of that, it is compatible with any other DCP slave. In comparison with the DCP, we propose a generic co-simulation interface, i.e., the system to be communicated is independent to the interface and there is no need to develop a specific slave for each application. In (Segura, Poggi, and Barcena 2023) we extended the scope of the DCP by creating a Simulink library, allowing Simulink to be easily integrated not only into our architecture, but also into any DCP application. Moreover, our architecture is agnostic to the simulation platform and to the communication medium, which facilitates cross-platform migration, which is what we will demonstrate in this article. Additionally, it enables real-time co-simulation. This is boosted by the Simulink implementation, that is, thanks to the automatic code generation capability of Simulink, we can convert our interface into source code (e.g. C or C++) and execute it on a wide variety of platforms, facilitating the validation of the system to be developed. In this work we take advantage of this capability by adapting the Simulink code developed previously, so it can be directly implemented on different platforms. Consequently, we provide a means to perform real-time communication between the models executed on such platforms.

As this paper deals with real-time (RT) simulations, it is worth having a little background on the characteristics of these systems. First, it is worth mentioning that, in real-time systems, the instant in which the response occurs is as important as the response itself (Kopetz 2011). If the response does not arrive at a predefined time, called deadline, the response may be unusable and may have adverse consequences for the system. Another characteristic of these systems is the so-called wall-clock. All computational elements involved in a real-time simulation must have a common clock reference and be synchronised to it. The higher the accuracy of this synchronisation, the better the system will be able to carry out temporally more constrained simulations. Determinism is another characteristic of these systems, it indicates the reproducibility of the system. That is, if we run several simulations of the same system, where all its components start at the same time and with the same starting conditions, the determinism means ability of the system to replicate the results at the same time instants.

On the other hand there are non real-time (NRT) simulations. These are controlled simulations that usually repeat a read-compute-write sequence, where they first wait for receiving data, then process it, and finally write the result at the output port. Once this cycle is finished, the next cycle starts following the same sequence. Comparing with real-time systems, they do not have to provide a temporally accurate response. It is to say, the message transport latency can vary without affecting the behaviour of the system. Simulink, for example, is a tool that by default runs in NRT, however, it also has a tool called Simulink Desktop Real-Time (MathWorks 2023), which allows us

DOI 10.3384/ecp204245

to synchronise the simulation with the wall-clock. This tool has two modes of use: I/O mode and kernel mode. The first one synchronises the I/O drivers with the real-time clock and allows us to perform real-time executions up to 1 kHz (1 ms sampling time), this is the one that we use in this work.

3 Proposed interface

The interface we propose is based on the non-proprietary DCP standard, thus it must be configured as a DCP slave. Nevertheless, as depicted in Figure 1, its behaviour is not that of a conventional DCP slave. Our implementation focuses on transmitting information from one environment to another and it is completely model-independent. Whereas in conventional usage, the slave wraps the model (Krammer et al. 2020), having to link them internally by hand. From a practical point of view, there is a big difference, since in the original paradigm a specific DCP slave has to be developed for each application, whereas our proposal is designed to indicate only the number of input/outputs plus an easy configuration of them. To achieve this independence between the model and the DCP slave, apart from implementing a specific DCP slave, we also developed a series of peripheral modules, which are explained in (Segura, Poggi, and Barcena 2023). Thus, our interface is composed of these modules and a DCP slave. Our goal in developing this interface as a Simulink library, was to take advantage of its tools so that we could generate C/C++ code for our interface and implement it on a variety of hardware platforms without additional modifications.

3.1 Configuration of the interface

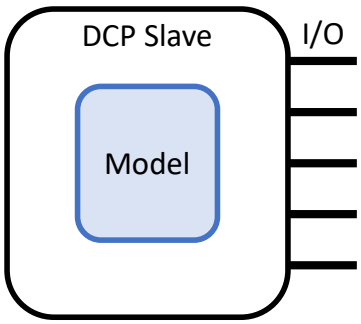
We have advanced that the configuration of the interface is based on the DCP standard, therefore, we will use the DCP standard specification document (Modelica Association 2023a) for the explanations of this section. The references to specific clauses of the standard will be in italics to better guide the reader. In this section we will only focus on the essential parameters (in monospace font) to define our interface, however, there are also other optionally modifiable parameters whose explanation can be found in the standard specification document. Figure 2 will be helpful to understand certain concepts.

Some parameters are set in each slave, while others are set in the master. The slaves are limited to set their internal parameters (see *5.4 Definition of dcpSlaveDescription Element, pp. 80-82*), of which, the essential parameters for configuring the interface are defined by the following elements:

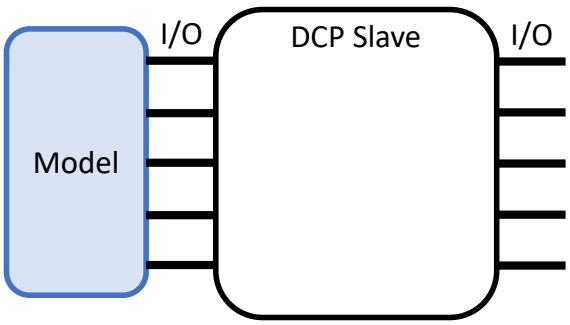
- Time resolution (*5.9 Definition of TimeRes Element, pp. 87-88*). It defines one atomic step of the slave and it is represented by an element that contains a list of permissible single time resolutions or a list of resolution ranges. To set a single resolution, that is what we are going to do next, we have to set two sub-parameters: `numerator` and `denominator`. Where the `numerator` divided by the `denominator` represents the time resolution of the slave.

- Transport protocol (*5.11 Definition of TransportProtocols Element, p. 88*). The DCP supports multiple transport protocols and this element is used to store their specific settings. For instance, if UDP transport protocol is used, this element must indicate it and contain the `host` and `port` data of the slave.

- Variables (*5.13 Definition of Variables Element, pp. 92-98*). This element contains the information about the variables of the slave, they can be either an Input, an Output, a Parameter, or a Structural Parameter. Among its subparameters, the indispensable ones to configure our interface are: `valueReference`, `dataType`, and `declaredType`. `valueReference` is the identifier of each variable and its value must be unique, in Figure 2 it can be seen how each I/O of each slave has different `valueReference` (vr) values. With `dataType` we declare the data type of the variable,



(a) Original DCP Slave design representation. Internally linked DCP Slave and Model. One specific DCP Slave for each Model.



(b) Proposed interface. New DCP Slave design. Externally linked DCP Slave and Model. The same slave with easily configurable Input/Outputs.

Figure 1. Comparison between original DCP Slave design and our interface.

see *Table 174: Data type elements* to know the accepted data types by the DCP standard. Finally, with `declaredType` we indicate whether the variable is used as an input/output that connects to another slave or as an input/output that connects to an external model. For this purpose, we have two predefined options: *default*, for communication between slaves, and *interface* for communication with the models.

As the task of the interface is to communicate different co-simulation environments, inputs and outputs will always have to be declared in pairs. Each pair will be internally linked in an automatic way as long as their `valueReferences` are consecutive. In other words, the `valueReference` parameters of an input-output pair must be consecutive. These values shall consist of the pairs 1-2, 3-4, ..., regardless of which of the two is the value of the input and which of the output. Additionally, if an output of one interface communicates with an input of another interface, both must have the same `valueReference`. This is shown in Figure 2. In this way we determine the link between interfaces and certify a correct communication.

The master, on the other hand, configures how the inputs and outputs of the slaves should communicate with each other. That is to say, it indicates to each slave where the inputs corresponding to its outputs are and vice versa; in addition, it establishes the sending frequency of each output. To do this, the following parameters must be configured:

- Step Size. The master defines the step size of each output of all slaves. The step size is a multiple of the time resolution parameter mentioned in the slave configuration. That is, the step size of each output is defined by the time resolution of its slave multiplied by the `step` parameter.
- Data Identifier (`data_id`). When exchanging information between slaves, Outputs are communicated to Inputs via DAT_input_output PDUs. Thanks to the `data_id` parameter, the values of several outputs of a slave can be grouped in a single DAT_input_output PDU. Only outputs that have the same configuration, i.e. sender, receiver and step size, can be grouped together.

4 Methodology

In this section we explain the experiments that we performed to show how the architecture presented in (Segura, Poggi, and Barcena 2021) is applicable on several hardware platforms, and how it is applicable on a distributed real-time co-simulation application. To do so, we apply it on four different platforms, which are introduced in subsection 4.1. As proof-of-concept use case, we use a closed-loop control model explained in subsection 4.2. In subsection 4.3 we present the co-simulation scenarios we use to demonstrate the applicability of the interface. Finally, in subsection 4.4, we explain how to configure our generic co-simulation interface for this particular use case.

4.1 Hardware platforms

In order to test the applicability of our architecture, and thus of our interface, it has been decided to work with hardware platforms designed for different purposes, concretely we used:

- Hardware platforms with integrated FPGA, such as the Xilinx Zynq UltraScale+ and the Xilinx Zynq-7000 SoC ZC702.
- Hardware platforms with integrated GPU, such as the NVIDIA Jetson Nano.
- Generic hardware platforms such as the Raspberry Pi 3B, which is very accessible and widely used.

By working with platforms that integrate FPGAs or GPUs, we are able to introduce these technologies into co-simulations, expanding our design to new applications, such as simulation accelerators, artificial intelligence, or image processing.

However, for now our interface has two limitations. Firstly, it is only implemented to run on soft real-time (SRT) and hard real-time (HRT) operation modes, the non-real-time (NRT) operation mode has not yet been implemented. Therefore, as both implemented modes require a common clock reference shared by all computing elements, the interface must be implemented in an environment that can provide it. Secondly, for the moment, we have only implemented UDP communication, so the boards must have an Ethernet port.

4.2 Use case: control of a closed-loop system

As was done in (Segura, Poggi, and Barcena 2023), as a proof of concept we use a closed-loop control system. This system consists of two parts: a plant, which is modeled as a discrete time first-order system, represented by Equation 1; and a PI control algorithm, represented by Equation 2. The simulation analysis is done by observing the time evolution of the closed-loop system response to a step input, comparing both the transient and steady-state parts. It is worth pointing out that this work is focused on testing the capability of the interface to communicate distributed systems in real-time. Therefore, in order to facilitate the demonstration process, we have chosen this simple use case.

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

Where:

- u is the control signal.
- y is the plant output or feedback signal.

DOI 10.3384/ecp204245

- a is a constant parameter, and it was permanently set to $a = 0.99$.
- b is a constant parameter, and it was permanently set to $b = 0.01$.

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

Where:

- u is the control signal.
- K_p is the proportional gain coefficient.
- K_i is the integral gain coefficient.
- T_s is the sampling period.
- $e(k) = r(k) - y(k)$ is the error signal.
- $r(k)$ is the target reference signal.
- z is the unit delay operator.

4.3 Co-simulation scenarios

In (Segura, Poggi, and Barcena 2023) we presented different scenarios to explain the development process of the interface. We started with a scenario composed only of Simulink and ended up with a scenario where the control algorithm was running in an UltraScale+ and the plant in Simulink. However, in order to implement the control subsystem on the UltraScale+ board, we manually created a DCP slave using C++ code that was specifically adapted to work on this board and be compatible with the control algorithm. Now we want to progress in the development of the interface and test its applicability. To do so, following the MBD methodology, we have automatically generated the interface code from the Simulink model, using the Embedded Coder tool. In order to be able to generate code correctly, we adapted the Simulink model by adding blocks and creating functions compatible with this generation. After that, we were able to generate directly implementable code, without the need for any changes, in any of the platforms presented in subsection 4.1.

Figure 2 represents the co-simulation scenario. In it, we can see the two models that compose the use case, defined in the subsection 4.2, located in different simulation environments and communicated by two entities/slaves of our interface. On the left we can see the *Model 1*, where the control algorithm is located. On the right, we can see the *Model 2*, which is composed of the plant and a synchronisation mechanism. The latter has the function of ensuring a controlled start of closed-loop control applications. Guaranteeing identical starts in all executions help us analyze the interface behaviour. For a detailed description of its operation refer to (Segura, Poggi, and Barcena 2023). The communication medium used to link both systems is UDP.

To demonstrate the easy implementation capability of the interface on different hardware platforms and, at the same time, to analyse its scope for performing real-time XIL simulations, we have considered the following scenarios:

- Scenario 1.A: Control algorithm and interface in the ARM-based processor of the ZC702 and Plant in Simulink.
- Scenario 1.B: Control algorithm and interface in the ARM-based processor of the UltraScale+ and Plant in Simulink.

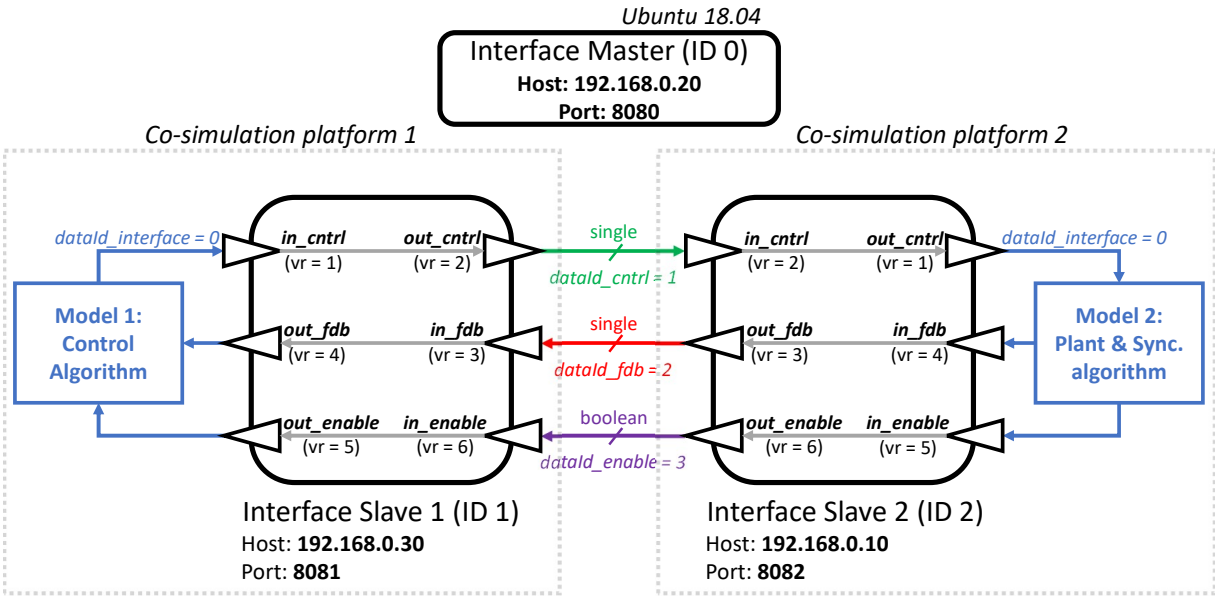


Figure 2. Interface configuration of the employed use case.

- Scenario 2: Control algorithm and interface in the ARM-based processor of the Jetson Nano and Plant in Simulink.
- Scenario 3: Control algorithm and interface in the CPU of the Raspberry-Pi and Plant in Simulink.
- Scenario 4.A: Control algorithm in the FPGA of the ZC702, interface in the ARM-based processor of the ZC702, and Plant in Simulink.
- Scenario 4.B: Control algorithm in the FPGA of the UltraScale+, interface in the ARM-based processor of the UltraScale+, and Plant in Simulink.

Additionally, to demonstrate its applicability in distributed real-time executions, we have proposed the following scenario:

- Scenario 5: Control algorithm in the FPGA of the ZC702, interface in the ARM-based processor of the ZC702, and Plant in the CPU of the Raspberry-Pi, see Figure 3.

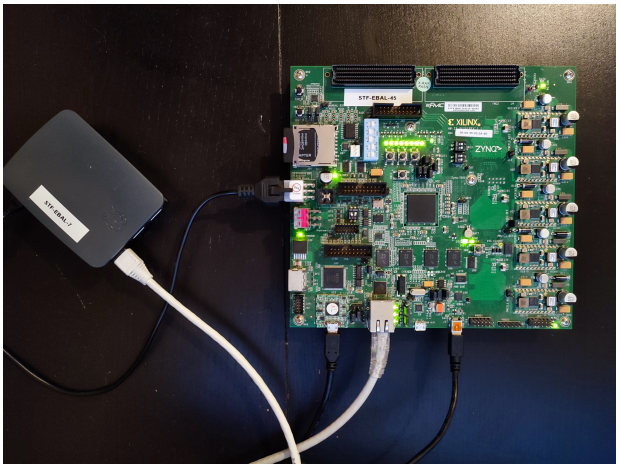


Figure 3. Scenario 5.

It is worth to mention that in all scenarios we have kept the same interface configuration (see subsection 4.4), thus facilitating the interoperability between simulation tools.

As we mentioned in subsection 4.1, our interface is limited to work in real-time operation mode, therefore all the scenarios have to be executed in real-time. The DCP standard defines that in its real-time mode, all components within the simulation must be synchronised with POSIX time. That is to say, they have to synchronise their clock by reference to 1 January 1970, 00:00:00 UTC. Consequently, we have to make all the components run in an environment that supports it and make them synchronised with each other. To do this possible on the platforms, we have installed an Ubuntu operating system on the CPUs of all of them, whose clock will be synchronised with the POSIX time. Therefore, the interface and the system (plant or controller) will run on it. Regarding Simulink, as explained above, by default it works in non-real-time mode. However, we will use its Simulink Desktop Real-Time tool in I/O mode, which allows us to synchronise the UDP ports with the wall-clock. This way, it will be also be synchronised to the POSIX time. It should be noted that we will run Simulink on a conventional PC that contains a 6 core Intel Core i7 CPU processor and using a Windows 10 operating system.

The use of the interface must not alter the behaviour of the closed-loop system in any of the scenarios. Therefore, we need a reference in order to be compared with the scenarios. To this end, using the Embedded Coder and HDL Coder tools provided by Simulink, we conducted Processor-in-the-Loop (PIL) and FPGA-in-the-Loop (FIL) simulations equivalent to the scenarios. In this way, we obtained a reference response for each of the scenarios. In other words, for scenarios 1.A, 1.B, 2, and 3 we performed PIL simulations, one with each hardware platform. While for scenarios 4.A and 4.B we performed FIL simulations. Each of these simulations are used as a reference for their respective scenario. Regarding scenario 5, we compare its responses to those of the system running entirely in Simulink.

4.4 Interface and simulations configuration

Two types of configurations were applied to conduct the experiment: the configuration of the models to be simulated and the configuration of the interface.

Regarding the configuration of the models, in (Segura, Poggi, and Barcena 2023) we saw that the behaviour of the interface varied depending on the execution time, therefore we performed the tests using six different configurations. In the current article, instead, we are going to focus on working only with the most limiting configuration that we encountered, which is shown at Table 1.

Configuration I	$T_s = 10$ ms	$K_p = 10$ $K_i = 10$

Table 1. Configuration for the simulation

Regarding the configuration of the co-simulation interface. In subsection 3.1 we explain the indispensable parameters to be configured. Now, we specify which values we chose for our particular use case:

- Time resolution. With this parameter we indicate under which step-size the state machine of the interface is executed. We set it to the lowest resolution that Simulink Desktop Real-Time allows when working in I/O mode. Therefore, we set it to $1ms$: $numerator = 1$ and $denominator = 1000$. It is also worth mentioning that the step-size of the interface must be lower than that of the model (Segura, Poggi, and Barcena 2023).
- Transport protocol. As mentioned, we use UDP. In

DOI 10.3384/ecp204245

the Figure 2 we can see the chosen host and port values.

- Variables. We declared 3 inputs and 3 outputs to each interface. In the Figure 2 we can see the data types and the value reference of each one.
- Data identifier. In the Figure 2 we can see that each slave has been assigned four *data_id*. This means that each variable that is transmitted between slaves has grouped independently, while the outputs that go from the interfaces to each of the models have been grouped together.
- Step size. We assigned a step value of 3 to the three *data_id*s that are transmitted between slaves (i.e. dataId_cntrl, dataId_fdb, and dataId_enable) and a step value of 10 to dataId_interface. With this configuration we could have assigned the same data_id to the three signals that are transmitted between slaves, but this makes it easier in case of future modifications.

5 Results

In this section, we present the simulation results of the scenarios explained in subsection 4.3. They will be discussed in section 6. In order to prove that the system behaves identically every execution, as done in (Segura, Poggi, and Barcena 2023), we perform 25 executions of each scenario. Between each run, the system is reset in order to ensure identical starting conditions in each of them. Subsequently, we compare the time response of the $y(k)$ output of each scenario with the respective reference. From this comparison we obtain the error, which is calculated by means of Equation 3.

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

where N represent the steps executed in a simulation, $y(k)$ is the response of the closed-loop system at step k under a specific scenario, and $y^{ref}(k)$ is the response of the closed-loop system under the corresponding reference. Table 2 reports the maximum, the minimum, the mean and the standard deviation of the error over the 25 repetitions.

Figure 4 helps us understand the results of the table. It comprises two graphs, each comparing the response $y(k)$ of a scenario (red line) with its corresponding reference (green line). There are 25 red lines in each graph, corresponding to the 25 repetitions that were executed for each configuration. We have decided to show only these two scenarios because they are sufficient to explain the behavior of the rest.

6 Results Analysis

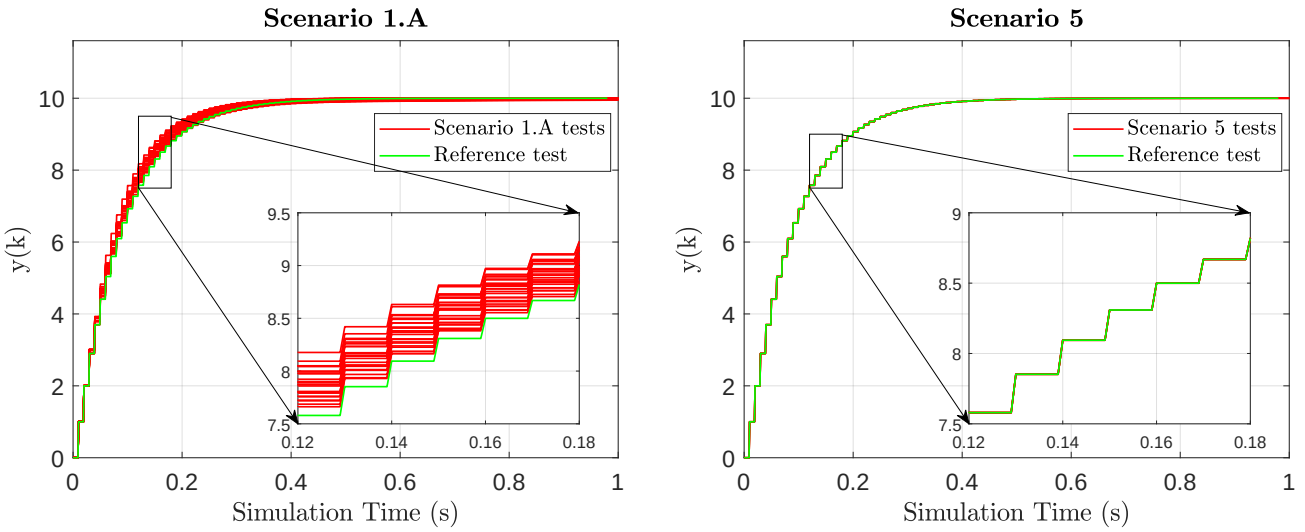
There are three topics we have discussed in this article, so this analysis will also be divided into three parts.

Scenario 1.A - PIL in ZC702	
max = 0.23479	min = 0.042779
mean = 0.12957	sd = 0.05131
Scenario 1.B - PIL in UltraScale+	
max = 0.21266	min = 0.024876
mean = 0.13098	sd = 0.048035
Scenario 2 - PIL in Jetson Nano	
max = 0.20944	min = 0.030067
mean = 0.11274	sd = 0.043965
Scenario 3 - PIL in RaspberryPi	
max = 0.20592	min = 0.050682
mean = 0.13517	sd = 0.04197
Scenario 4.A - FIL in ZC702	
max = 0.14277	min = 0
mean = 0.04963	sd = 0.0404
Scenario 4.B - FIL in UltraScale+	
max = 0.12031	min = 0
mean = 0.051518	sd = 0.036303
Scenario 5 - Distributed co-simulation	
max = 0	min = 0
mean = 0	sd = 0

Table 2. Comparison between MathWorks PIL/FIL solution and our Interface.

The first point focuses on evaluate the viability and ease of implementation of the interface in a variety of hardware platforms. Since we developed it in Simulink and adapted all its functions to be compatible with the generation of generic C/C++ code, we were able to implement it easily on all the platforms mentioned in subsection 4.1. This way we demonstrate that the interface can be migrated between platforms without any extra effort.

As for our second point, we evaluated the viability of the interface to perform real-time executions between Simulink and the platforms described in subsection 4.1. In Table 2, from Scenario 1A to Scenario 4.B, we find the results of the tests carried out for this point. Additionally, Figure 4a displays graphically the results obtained in Scenario 1A. As the graphs obtained in Scenarios 1A to 4B are very similar, we decided to omit the rest and display only this one to assist in the interpretation of Table 2. Analysing this table, we observe that there is an error in every scenario. Simplifying the results, we can say that the mean error of PIL simulations (from Scenarios 1A to 3) is in the order of 0.215 units ($\pm$0.02), whereas the mean error of FIL simulations (Scenarios 4A and 4B) is about 0.13 units ($\pm$0.01). This error occurs because the PIL and FIL simulations made with MathWorks tools exhibit identical behaviour in each run, while our simulations vary in each of the 25 runs. This can be seen in Figure 4a, where there is single green line (corresponding to the MathWorks re-



(a) Scenario 1A.

(b) Scenario 5. The responses overlap.

Figure 4. Comparison of the responses of Scenarios 1A and 5 with their respective References.

sponse) and multiple red lines (each corresponding to one of the 25 tests). Nevertheless, it can also be seen that this error is focused in the transient state, while in the steady state it is minimum. In fact, in the steady state, from second 0.4 onwards, the mean error is of the order of 0.019 units ($\pm$0.008), with a standard deviation of another 0.019 units ($\pm$0.007).

The appearance of this error means that our solution is not deterministic, which is an indispensable quality in the real-time executions for the verification and validation processes of the CPS. Therefore, we can say that our interface is not suitable for linking Simulink and hardware platforms in real-time. However, we did not test how our interface would behave with these scenarios working in non real-time mode, that is, following the controlled read-compute-write sequence explained previously. In fact, this is the way that MathWorks perform PIL and FIL simulations. Nevertheless, as explained before, this is an operation mode that we have yet to implement.

Analysing the cause of this non-deterministic response, we have not been able to link Simulink simulation time with POSIX time. In other words, POSIX time is constantly moving forward, and in a period of time, the simulation time get blocked and it does not advance. Figure 5 demonstrates this behaviour, where the graph instead of showing a perfect diagonal line shows "jumps", which are sometimes more pronounced. As we discussed previously, real-time systems must guarantee a response every predefined period and this breaks do not allow it. This can be caused, for instance, due to an interruption in the computing platform. This is the reason why we have not been able to benefit from all the power that the Simulink Desktop Real-Time tool provides.

The effect of this desynchronization in our scenarios is that the plant, which is executing in Simulink, stops running for an indefinite period; while the control algorithm

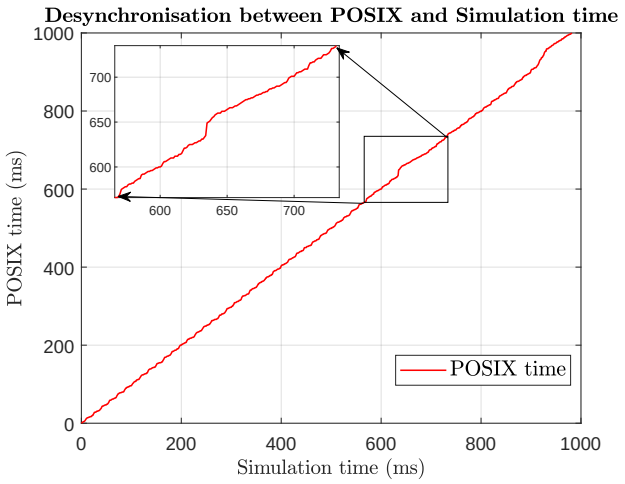


Figure 5. Desynchronisation between POSIX and Simulation time

on the hardware platform continues to run. During this period, the control will not receive updated input values from the plant, however they will be processed, provoking incorrect output control signals. When the plant resumes running, it reads the last of this unwanted values, resulting in an incorrect feedback value being sent to the controller. This way, a single desynchronization in the execution can significantly impact the system's behavior, leading to non-deterministic behavior. It is worth mentioning that the faster the system runs, the smaller its execution steps will be. As a result, a pause in the simulation time will involve more simulation steps, creating a more adverse effect on the system's response.

Finally, it remains to analyse the response of our implementation in real-time distributed applications, i.e., the Figure 4b. Contrary to what happens in the previous tests,

DOI 10.3384/ecp204245

we can see how the 25 red lines are overlapped. On top of that, they have the same behaviour as the reference (green line). The fact that we obtained identical results in all executions means a deterministic simulation. Therefore, we can be assured that our interface is suitable for real-time distributed simulations. At the same time, these results demonstrate that the problem we had in linking Simulink with hardware platforms lies in the fact we were not able to link Simulink with the wall-clock correctly.

7 Conclusions and future work

In this paper we present an empirical demonstration of the applicability of the previously developed generic co-simulation interface. Specifically, we demonstrated i) its easy implementation on a variety of hardware platforms, and ii) how it can be used in real-time distributed simulations. Both capabilities are very useful in the process of verification and validation of cyber-physical systems, especially in those whose components are developed by different suppliers; in those where the system is split into smaller modules to spread the computational load across different processors; or in those where the integration of different simulators into a single system is required. Therefore our interface could save time, resources and money, as it facilitates coupling of elements, saves displacements for integration testing and helps to preserve Intellectual Property.

To test the interface we deployed it in different hardware boards and performed a closed-loop simulation between them, obtaining reliable responses. Consistent with the MBD methodology, as we have developed it in Simulink and take advantage of its code generation capabilities, our interface is easily implementable in a wide variety of simulation environments. Additionally, as our interface is based on the non-proprietary DCP standard, it is fully compatible with any other DCP slave.

Looking to extend our work to the future, in order to improve the linking capability to Simulink, we want to develop the non-real-time simulation mode. Additionally we have planned to test the applicability of this interface in a more complex use case involving hardware-in-the-loop simulations.

Acknowledgements

This work was supported by Basque Government through the ELKARTEK programme under the AUTOEV@L project (KK-2021/00123).

References

Alfalouji, Qamar et al. (2023). "Co-simulation for buildings and smart energy systems — A taxonomic review". In: *Simulation Modelling Practice and Theory* 126, p. 102770. ISSN: 1569-190X. DOI: https://doi.org/10.1016/j.simpat.2023.102770.

Attarzadeh-Niaki, Seyed Hosein and Ingo Sander (2020). "Heterogeneous co-simulation for embedded and cyber-physical systems design". In: *Simulation: Transactions of the Society for Modeling and Simulation International* 96, pp. 753–765. DOI: 10.1177/0037549720921945.

Baumann, Peter et al. (2019). "Using the Distributed Co-Simulation Protocol for a Mixed Real-Virtual Prototype". In: *Proceedings - 2019 IEEE International Conference on Mechatronics, ICM 2019*. IEEE, pp. 440–445. ISBN: 9781538669594. DOI: 10.1109/ICMECH.2019.8722844.

Böhm, Wolfgang et al. (2021). *Model-Based Engineering of Collaborative Embedded Systems*. 1st ed. Springer. Chap. 12 & 13. ISBN: 978-3-030-62135-3. DOI: https://doi.org/10.1007/978-3-030-62136-0.

Falcone, Alberto and Alfredo Garro (2019). "Distributed Co-Simulation of Complex Engineered Systems by Combining the High Level Architecture and Functional Mock-up Interface". In: *Simulation Modelling Practice and Theory* 97. ISSN: 1569-190X. DOI: https://doi.org/10.1016/j.simpat.2019.101967.

Hatledal, Lars Ivar et al. (2019). "A Language and Platform Independent Co-Simulation Framework Based on the Functional Mock-Up Interface". In: *IEEE Access* 7, pp. 109328–109339. DOI: 10.1109/ACCESS.2019.2933275.

Ivanov, Valentin et al. (2019). "Connected and shared x-in-the-loop technologies for electric vehicle design". In: *World Electric Vehicle Journal* 10, pp. 1–13. DOI: 10.3390/wevj10040083.

Köhler, Christian (2011). *Enhancing Embedded Systems Simulation*. 1st ed. Vieweg+Teubner. Chap. 2. ISBN: 978-3-8348-1475-3. DOI: https://doi-org.ehu.idm.oclc.org/10.1007/978-3-8348-9916-3.

Kopetz, Hermann (2011). *Real-Time Systems*. 2nd ed. Real-Time Systems Series. New York: Springer, pp. XVIII, 378. DOI: https://doi.org/10.1007/978-1-4419-8237-7.

Krammer, Martin et al. (2020). "A Protocol-Based Verification Approach for Standard-Compliant Distributed A Protocol-Based Verification Approach for Standard-Compliant Distributed Co-Simulation". In: *Asian Modelica Conference 2020*. DOI: 10.3384/ecp20174133.

Marwedel, Peter (2021). *Embedded System Design - Embedded Systems Foundations of Cyber-Physical Systems, and the Internet of Things*. 4th ed. Springer. Chap. 1. ISBN: 978-3-030-60909-2. DOI: https://doi.org/10.1007/978-3-030-60910-8.

MathWorks (2023). *Simulink Desktop Real-Time*. https://es.mathworks.com/help/sldrt/low-sample-rate-simulation.html.

Modelica Association (2023a). *DCP standard specification*. https://github.com/modelica/dcp-standard.

Modelica Association (2023b). *Distributed Co-Simulation Protocol (DCP) website*. https://dcp-standard.org/.

Segura, Mikel, Tomaso Poggi, and Rafael Barcena (2021). "Towards the implementation of a real-time co-simulation architecture based on distributed co-simulation protocol". In: *35th Annual European Simulation and Modelling Conference 2021, ESM 2021*. EUROSIS-ETI, pp. 155–162. ISBN: 978-9-492-85918-1.

Segura, Mikel, Tomaso Poggi, and Rafael Barcena (2023). "A Generic Interface for x-in-the-Loop Simulations Based on Distributed Co-Simulation Protocol". In: *IEEE Access* 11, pp. 5578–5595. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2023.3237075.