Proceedings of the 2019 Winter Simulation Conference
N. Mustafee, K.-H.G. Bae, S. Lazarova-Molnar, M. Rabe, C. Szabo, P. Haas, and Y.-J. Son, eds.

CO-SIMULATION OF CONTINUOUS SYSTEMS: A HANDS-ON APPROACH

Cláudio Gomes
Hans Vangheluwe

Department of Mathematics and Computer Science
University of Antwerp
Middelheimlaan 1
Antwerp, 2020, BELGIUM

ABSTRACT

Co-simulation consists of the theory and techniques to enable global simulation of a coupled system via the composition of simulators for each of the components of the model. Despite the large number of applications and growing interest in the challenges, practitioners still experience difficulties in configuring their co-simulations. This tutorial introduces co-simulation of continuous systems, targeted at researchers that want to develop their own co-simulation units and master algorithms, using the Functional Mock-up Interface (FMI) Standard. This document is complemented by online materials (Gomes, Cláudio 2019) that allow the reader to experiment with different co-simulation algorithms, applied to the examples introduced here.

1 INTRODUCTION

Truly complex engineered systems that integrate physical, software and network aspects are emerging (Nielsen et al. 2015), posing challenges in their design, operation, and maintenance. The design of such systems, due to market pressure, has to be concurrent and distributed. That is, divided between different teams and/or external suppliers, each in its own domain and each with its own tools (Vangheluwe et al. 2002). Each participant develops a partial solution, that needs to be integrated with all the other partial solutions. The later in the development process such integration is done, the higher its cost (Plateaux et al. 2009). Ideally, the solutions developed independently should be integrated sooner and more frequently, in so-called full system analysis (Van der Auweraer et al. 2013).

Modeling and simulation has improved the development of the partial solutions, but falls short in fostering this holistic development process (Blochwitz et al. 2011). To understand why, one has to observe that: (i) models of each partial solution cannot be exchanged or integrated easily, because these are likely developed by a specialized tool; (ii) externally supplied models may have Intellectual Property (IP) that cannot be cheaply disclosed to system integrators; (iii) as solutions are refined, the system should be evaluated by integrating physical prototypes, software components, and even human operators, in what are denoted as Model/Software/Hardware/Human-in-the-loop simulations (Alvarez Cabrera et al. 2011); and (iv) the models of each partial solution have different characteristics that can be exploited to more efficiently simulate them, making it difficult to find a technique that fits all kinds of models.

Co-simulation is a generalized form of simulation, where a coupled system is simulated through the composition of simulation units (Hafner and Popper 2017; Palensky et al. 2017; Gomes et al. 2018). Each unit is broadly defined as a *black box* capable of exhibiting behaviour, consuming inputs and producing outputs, over simulated time.

Since co-simulation is but a special kind of simulation, it shares the same major challenge: *can we trust the co-simulation results?* However, this challenge is aggravated due to the black-box nature of co-simulations, and the large number of configuration parameters. In fact, a recent empirical survey has shown that practitioners still experience difficulties in the configuration of co-simulations (Schweiger et al. 2018; Schweiger et al. 2019).

In this tutorial, we aim to provide the readers with a basic understanding of how to develop their own master algorithms and simulation units. Moreover, we highlight the strengths and weaknesses of the most common master algorithms. The concepts are presented in a technologically agnostic manner. However, the online materials (Gomes, Cláudio 2019) are implemented for the Functional Mock-up Interface (FMI) standard, a widely used standard for co-simulation.

Upon completion, the reader should know the most common co-simulation approaches, the main concepts involved, the main configuration parameters, and what their trade-offs are. Furthermore, the reader will be equipped to understand the more advanced concepts in the co-simulation literature, provided in the extended version of this tutorial (Gomes et al. 2018).

The next section gives a top-down overview of all the concepts that will be discussed here. In the subsequent sections, each concept will be discussed, in a bottom up manner, so as to increase the complexity gradually.

2 MAIN CONCEPTS

In this section, we provide an informal top-down overview on the concepts related to co-simulation. To that end, we use a *feature model* (Kang et al. 1990): an intuitive diagram that breaks down the main concepts in a domain. Some of these concepts will only become clear in later sections, as we delve into the details, so we recommend the reader to come back to this section to place these in context. More rigorous definitions are given in (Gomes et al. 2018).

First, we summarize the *objective* of running a co-simulation: to reproduce, as accurately as possible, the behavior of a *system under study*.

Figure 1 breaks down the main concepts in the co-simulation domain. To run a co-simulation, one needs a co-simulation scenario and an orchestrator algorithm.

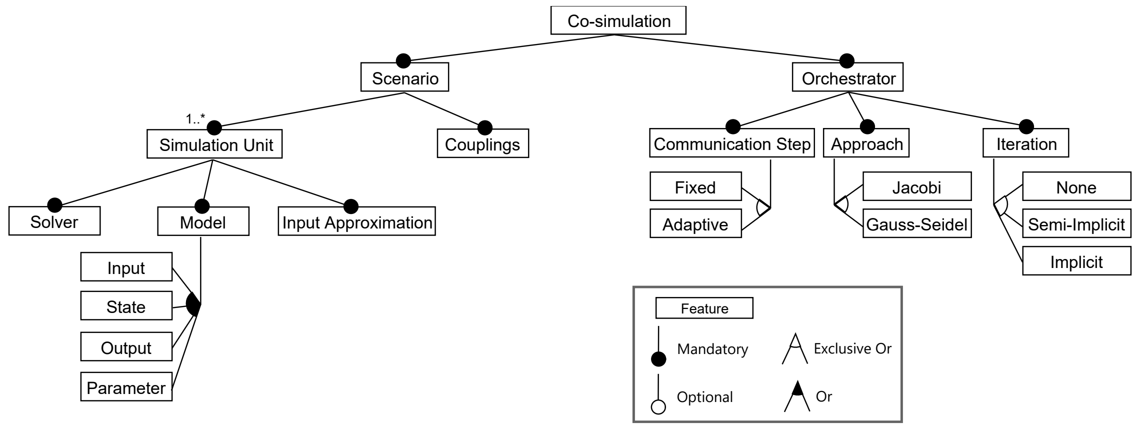

Figure 1: Co-simulation concept breakdown.

The *co-simulation scenario* points to one or more simulation units, describes how the inputs and outputs of their models are related, and includes the configuration of relevant parameters. Each *simulation unit* represents a black box capable of producing behavior. To produce behavior, the simulation unit needs to have a notion of:

- a *model*, created by the modeller based on his knowledge of the system under study;
- a *solver*, which is part of the modeling tool used by the modeller, that approximates the behavior of the model; and

- an *input approximation*, which approximates the inputs of the model over time, to be used by the solver;

The *orchestrator* is responsible for running the co-simulation. It initializes all the simulation units with the appropriate values, sets/gets their inputs/outputs, and coordinates their progression over the simulated time. To progress the co-simulation, the orchestrator, after setting the appropriate inputs to the simulation units (computed from their outputs according to the co-simulation scenario), asks them to simulate for a given interval of simulated time, by providing them with a *communication step*. The simulation units in turn will approximate the behavior of their model within the interval between the current simulated time and the next communication time, relying only on the inputs they have received at the previous communication times.

Figure 2 gives an illustration of these concepts. The figure in the left-hand side illustrates how the orchestrator coordinates the co-simulation by getting outputs, setting inputs, and requesting the simulation units S1 and S2 to progress in time. The figure in the top-right-hand side presents the co-simulation scenario, where S_1 receives input F_c and outputs $[x_1, v_1]$, and S_2 receives inputs $[x_1, v_1]$ and outputs F_c. The two plots in the bottom-right-hand side presents the internal behaviour of the simulation units. The large unfilled dots represent input values, and the smaller unfilled dots represent their extrapolations, as computed by the simulation units. One can see that there is a difference between the values calculated by the extrapolation functions and the actual input, due to the gap between the larger and smaller unfilled dots at $t+H$. The black dots represent outputs. As illustrated, S_1 and S_2 perform small steps of respectively h_1 and h_2 internally, until the time $t+H$ is reached.

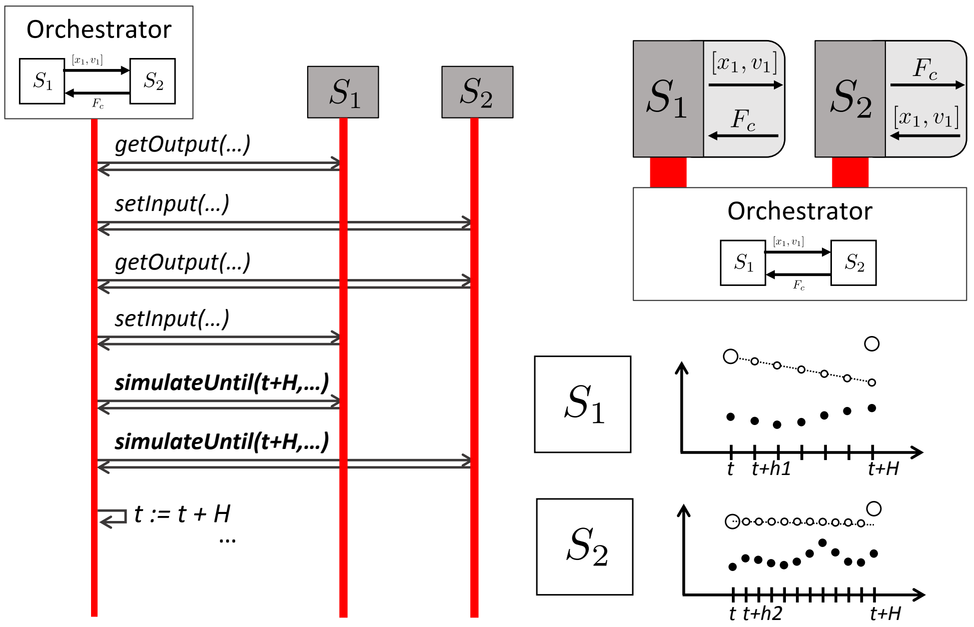

Figure 2: Example co-simulation coordination (left), co-simulation scenario (top right), and internal behavior of simulation units (bottom right).

Looking back at Figure 1, the communication step size can either be *fixed* (defined before the co-simulation starts and constant throughout its execution), or *adaptive* (the orchestrator determines the best value to be used whenever it asks the simulation units to compute). The *communication approach* encodes the order in which the simulation units are given inputs and instructed to compute the next interval.

Figure 3 summarizes the multiple types of orchestration algorithms using time diagrams. In the *Gauss-Seidel* approach, the orchestrator asks each simulation unit to compute the next interval, before asking it to produce outputs. These outputs are then fed into the next unit before asking it to compute the next interval. In the *Jacobi* approach, the orchestrator asks all units to compute the interval in parallel, setting their inputs at the end of the co-simulation step.

Finally, the orchestrator may retry the co-simulation step, using improved input estimates, computed from the most recent outputs. This process can be repeated until there is no improvement on the inputs (*fully implicit iteration*), or a fixed number of iterations has been done (*semi-implicit iteration*).

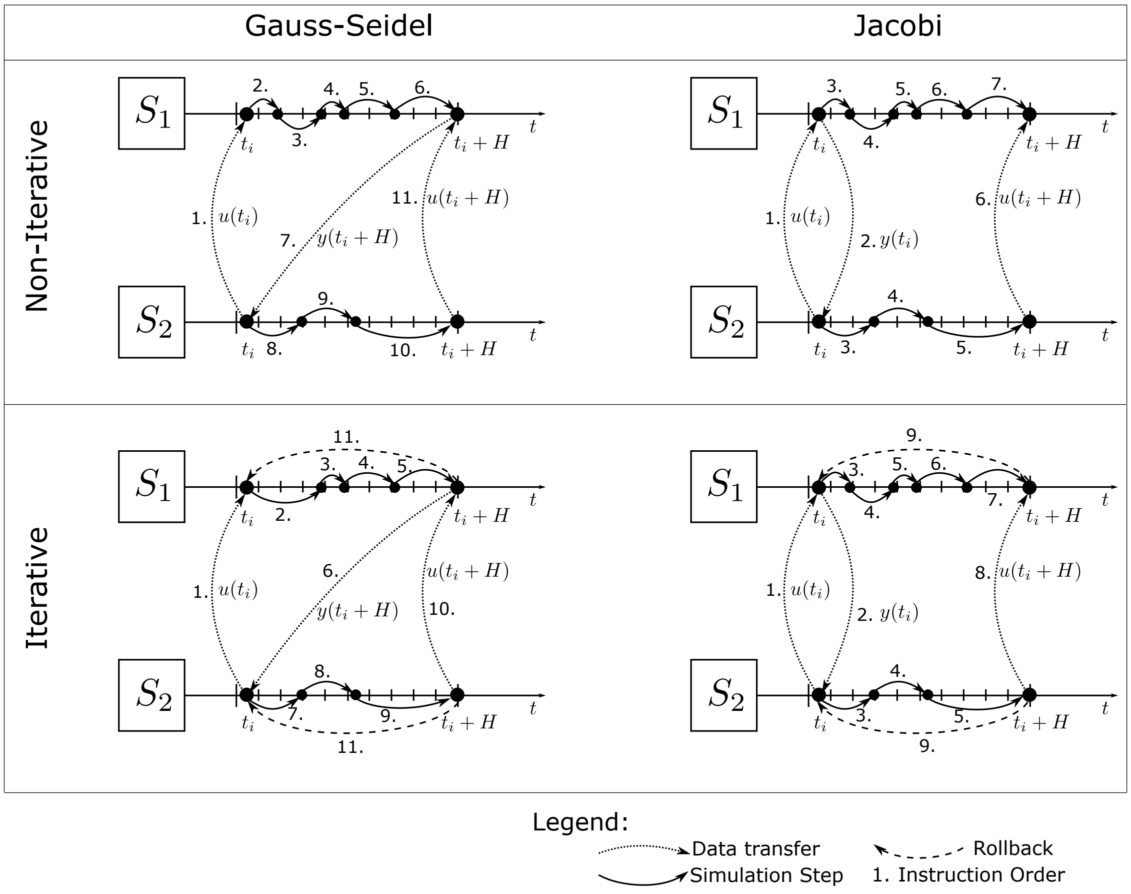

Figure 3: Overview of orchestration algorithms of two simulators using time diagrams. The number next to each edge denotes the order of execution of that operation.

2.1 Related Concepts and Standards

Using the nomenclature adopted in this tutorial, the High Level Architectures (HLAs) (IEEE. 2014), and FMI(FMI. 2014), are co-simulation standards. Table 1 summarizes the concepts described in this tutorial, and their relationships with the aforementioned standards. It's important to note that, compared to HLAs, the FMI standard does not specify the orchestration algorithm, but places a higher focus on the integration of continuous system models. For those reasons, this tutorial and accompanying materials are aligned with the FMI standard.

In the following sections, we follow a bottom up approach, starting with the simplest concepts in Figure 1 (Model, Solver, and Input Approximation), and building our way up to co-simulation.

Table 1: Main concepts and their relationship with existing co-simulation standards.

	FMI	HLA
Scenario	Federation	Co-simulation
Simulation Unit	Federate	Functional Mock-up Unit (FMU)
Orchestrator	Run Time Infrastructure (RTI)	Master Algorithm

3 BASICS OF CO-SIMULATION

Since co-simulation is a form of generalized simulation, it is paramount that simulation is well understood. In the following, we introduce based algorithms to approximate the solution, $x(t)$, of first order Ordinary Differential Equations (ODEs), $\dot{x} = f(x,u)$, having an initial condition, $x(0) = x_0$. We start with scalar differential equations and then move to vector equations. The relationship between the concepts learned in Sections 3.1.1 and 3.1.2, and the concept of simulation unit (recall Figure 1), is discussed in then Section 3.1.3.

3.1 Models, Solvers, and Input Approximations

3.1.1 Scalar Initial Value Problems

A *scalar Initial Value Problem (IVP)* is defined as a scalar ODE, with an initial condition. Formally, it has the form:

$$\dot{x} = f(x,u), \text{ with } x(0) = x_0, \tag{1}$$

where $x : \mathbb{R} \to \mathbb{R}$ denotes the (scalar) state function, $\dot{x}$ denotes the time derivative of x, $f : \mathbb{R}^2 \to \mathbb{R}$ is a scalar function, $u : \mathbb{R} \to \mathbb{R}$ is the input function, and $x_0 \in \mathbb{R}$ is a given initial value of $x(t)$.

Example 1. *Consider a car whose acceleration is set by a cruise controller, and moves in a straight line. Let $v(t)$ denote the speed of the car over time, m its mass, and v_d the desired speed (constant input); and assume that the car is initially moving at speed v_0. Then the scalar IVP is given by*

$$\dot{v} = \frac{1}{m}\left[k(v_d - v) - c_f v\right], \textit{ with } v(0) = v_0, \tag{2}$$

where $k(v_d - v)$ is the acceleration set by the cruise controller, $k > 0$ is the acceleration multiplier constant, and $c_f > 0$ is the friction coefficient. The code for this example is available online (Gomes, Cláudio 2019).

The solution of the scalar IVP (1) is a function $x(t) : \mathbb{R} \to \mathbb{R}$ whose derivative satisfies Equation (1). For example, the solution of the IVP posed in the car example (Example 1), and plotted in Figure 4, is:

$$v(t) = \frac{kv_d}{c_f + k} - \left(\frac{kv_d}{m} - v_0\right) e^{-\frac{t}{m}(c_f + k)}.$$

In general, it is not possible, nor feasible, to find an explicit solution to the IVP. Instead, an approximate solution can be computed using a numerical method.

To derive an approximation $\tilde{x}(t)$ of the solution to the scalar IVP in Equation (1), we start by noting that the initial point is given by the initial value, that is, $\tilde{x}(0) = x_0$, so at least one point is known. For a small $h > 0$, the limit definition of the derivative in the left hand side of Equation (1) can be replaced by its approximation $\dot{x} \approx (x(t+h) - x(t))/h$. By Equation (1), we have $(x(t+h) - x(t))/h \approx f(x(t),u(t))$, which can be solved for $x(t+h)$ to give the *Explicit Euler Method*:

$$x(t+h) \approx x(t) + f(x(t),u(t))h, \text{ with } x(0) = x_0. \tag{3}$$

Applying Equation (3) to the initial value, gives the point $\tilde{x}(h)$, which approximates $x(h)$. The procedure can then be repeated using $\tilde{x}(h)$ to compute $\tilde{x}(2h)$, and so on. The result of applying this procedure to Example 1 is shown in Figure 4.

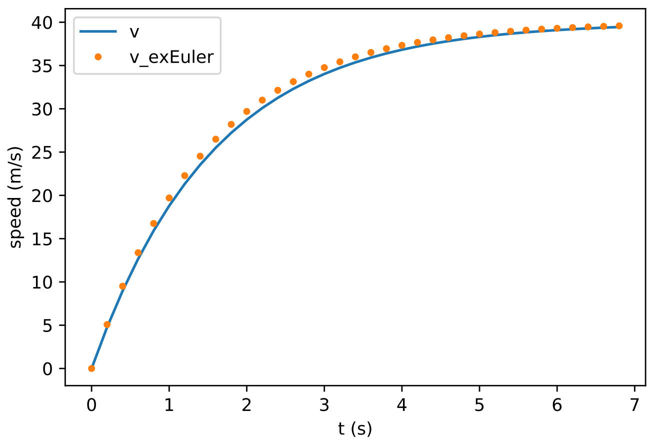

Figure 4: Analytical solution and approximation to the IVP in Example 1. Parameters are: $h = 0.2, m = 1576(kg), v_d = 40(m/s), v_0 = 0(m/s), k = 10^3, c_f = 0.5$.

3.1.2 Vector Initial Value Problems

In this sub-section, we generalize the numerical techniques introduced in Section 3.1.1 to vector IVPs. We will denote vectors with bold face, and we will use capital letters for matrices and vector valued functions. Given a vector $\boldsymbol{x}$, we denote its transpose as $\boldsymbol{x}^T$. Similarly, $F_i(\boldsymbol{x})$ denotes the i-th element of the vector returned by $F(\boldsymbol{x})$.

An *IVP* is the generalization of Equation (1), to vectors:

$$\dot{\boldsymbol{x}} = F(\boldsymbol{x}, \boldsymbol{u}(t)), \text{ with } \boldsymbol{x}(0) = \boldsymbol{x_0}, \tag{4}$$

where $\boldsymbol{x}$ and $\boldsymbol{u}$ are vector functions, and F is a vector valued function.

Example 2. *The mass-spring-damper system, illustrated in Figure 5, is modelled by the following second order ordinary differential equation:*

$$\ddot{x}_1 = \frac{1}{m_1}(-c_1 x_1 - d_1 \dot{x}_1 + F_c(t)),$$

where x_1 denotes the position of the mass, $\ddot{x}_1$ denotes the second time derivative of x_1, $c_1 > 0$ is the stiffness coefficient of the spring, $d_1 > 0$ is the damping constant of the damper, and $F_c(t)$ denotes an external force exerted on the mass.

The above equation can be put into the form of Equation (4) by introducing a new variable for velocity, $v_1 = \dot{x}_1$, and letting the vector $\boldsymbol{x}_1 = \begin{bmatrix} x_1 & v_1 \end{bmatrix}^T$. Given an initial position $x_1(0)$ and velocity $v_1(0)$, we obtain the following IVP:

$$\dot{\boldsymbol{x}}_1 = \begin{bmatrix} \dot{x}_1 \\ \dot{v}_1 \end{bmatrix} = F(\begin{bmatrix} x_1 \\ v_1 \end{bmatrix}, F_c(t)) = \begin{bmatrix} v_1 \\ (1/m_1)(-c_1 x_1 - d_1 v_1 + F_c(t)) \end{bmatrix}, \textit{ with } \boldsymbol{x}(0) = \begin{bmatrix} x_1(0) \\ v_1(0) \end{bmatrix} \textit{ given.}$$

The time derivative of a vector is the time derivative of each of its components, so the solution to Equation (4) is a vector valued function $\boldsymbol{x}(t)$ where each component $\boldsymbol{x}_i(t)$ obeys the equation $\dot{\boldsymbol{x}}_i(t) = F_i(\boldsymbol{x}(t), \boldsymbol{u}(t))$, with $\boldsymbol{x}_i(0)$ given. As an example, Figure 6 plots the solution of the position component of the mass-spring-damper IVP introduced in Example 2. The solution to the velocity component is omitted.

The Explicit Euler method, introduced in Section 3.1.1 can be generalized to vector IVPs:

$$\boldsymbol{x}(t+h) \approx \boldsymbol{x}(t) + F(\boldsymbol{x}(t), \boldsymbol{u}(t))h, \text{ with } \boldsymbol{x}(0) = \boldsymbol{x_0}. \tag{5}$$

An example application is shown in Figure 6.

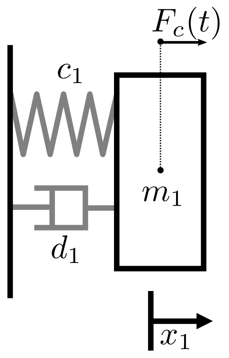

Figure 5: Mass-spring-damper system.

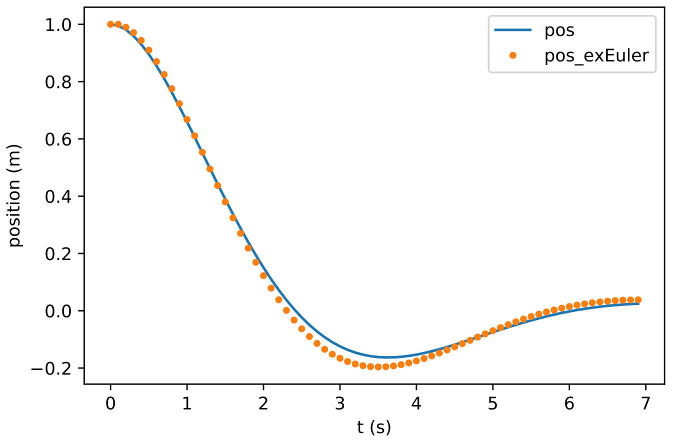

Figure 6: Position, and its approximation, over time, of the mass-spring-damper system. Parameters are: $h = 0.1, m_1 = c_1 = d_1 = 1, F_c(t) = 0, \boldsymbol{x_1}(0) = \begin{bmatrix} 1 & 0 \end{bmatrix}^T$.

3.1.3 Constructing Functional Mockup Units

This subsection describes how the concepts introduced in the previous subsection can be used to construct simulation units.

Models are vector IVPs with output:

$$\begin{aligned} \dot{\boldsymbol{x}} &= F(\boldsymbol{x}, \boldsymbol{u}), \text{ with } \boldsymbol{x}(0) = \boldsymbol{x_0}, \text{ and} \\ \boldsymbol{y} &= G(\boldsymbol{x}, \boldsymbol{u}), \end{aligned} \tag{6}$$

where $\boldsymbol{y}$ denotes the output vector, and G the output function. Solvers are numerical methods, such as the Euler method introduced in Equation (5).

To understand the role of input extrapolation functions, we need to recall the interactions between the orchestrator and each simulation unit (recall Figure 2). In order to facilitate the explanation, let us make the following assumptions: $H > 0$ denotes the communication step size, kept the same throughout the co-simulation; $t_i = iH$ denotes the simulated time at the i-th co-simulation step; and the orchestrator follows a Jacobi approach (see Figure 3). The other cases (i.e., H varies over simulated time, or the Gauss-Seidel orchestrator is used) should be easy to understand once this one is clear.

Under the above assumptions, the orchestrator, at time t_i, constructs the input to the unit, denoted as $u(t_i)$, and then asks the unit to compute until the time $t_{i+1} = t_i + H$. Between times t_i and t_{i+1}, the unit will iteratively approximate the state of the model, only taking into account the inputs $u(t_i), u(t_{i-1}), u(t_{i-2}), \ldots$ that it has received in the past. As such, the numerical solver employed in the simulation unit is actually solving a modified version of Equation (6):

$$\dot{\boldsymbol{x}} = F(\boldsymbol{x}, \tilde{\boldsymbol{u}}(t)), \text{ with } \boldsymbol{x}(t_i) = \boldsymbol{x_i}, \text{ and } t \in [t_i, t_{i+1}], \tag{7}$$

where $\tilde{\boldsymbol{u}}(t)$ is an approximation of $\boldsymbol{u}(t)$ in the interval $t \in [t_i, t_{i+1}]$, built from input samples computed by the orchestrator in the previous co-simulation steps: $\boldsymbol{u}(t_i)$, $\boldsymbol{u}(t_{i-1})$, $\boldsymbol{u}(t_{i-2})$, ... In this interval, the goal of the simulation unit is to estimate $\boldsymbol{x}(t_{i+1})$, so that the output $\boldsymbol{y}(t_{i+1})$ of the model (recall Equation (6)) can be computed and given to the orchestrator.

The accompanying online material (Gomes, Cláudio 2019) contains examples of simulation units implemented as FMUs.

3.2 Constructing Co-simulations

In this subsection, we describe how the mass-spring-damper system, introduced in Example 2, is coupled to second mass-spring-damper system (Example 3 below) in a co-simulation. Then we introduce multiple orchestration algorithms.

Example 3. *Consider the system in the right-hand-side of Figure 7. It is given by the following equations:*

$$\begin{aligned} \dot{x}_2 &= v_2 \\ m_2 \cdot \dot{v}_2 &= -c_2 \cdot x_2 - F_e \\ F_e &= c_c \cdot (x_2 - x_c) + d_c \cdot (v_2 - \dot{x}_c), \end{aligned} \tag{8}$$

where c_c and d_c denote the stiffness and damping coefficients of the spring and damper, respectively; x_c denotes the displacement of the left end of the spring-damper, and F_c denotes the force due to the relative displacement of the left end of the spring-damper and the mass. The variables x_2 and v_2 denote the position and velocity of the mass.

The physical coupling of the system in Example 3 and the system in Example 2 is illustrated in Figure 7. The combined equations can be obtained by replacing $F_c(t)$ in Example 2 by $F_e(t)$ as defined in Example 3, and replacing x_c, v_c in Example 3 by x_1, v_1 in Example 2. The resulting equation has an analytical solution, which can be used to obtain the time evolution of the the position of the left hand side mass in Figure 7, as is shown in Figure 8.

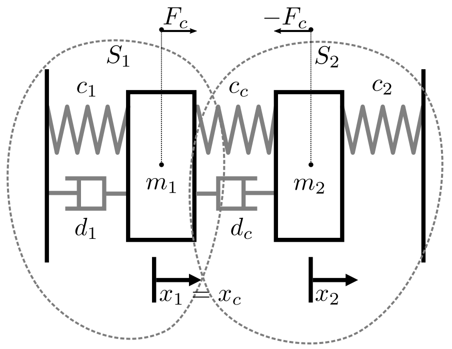

Figure 7: A multi-body system comprised of two mass-spring-damper subsystems.

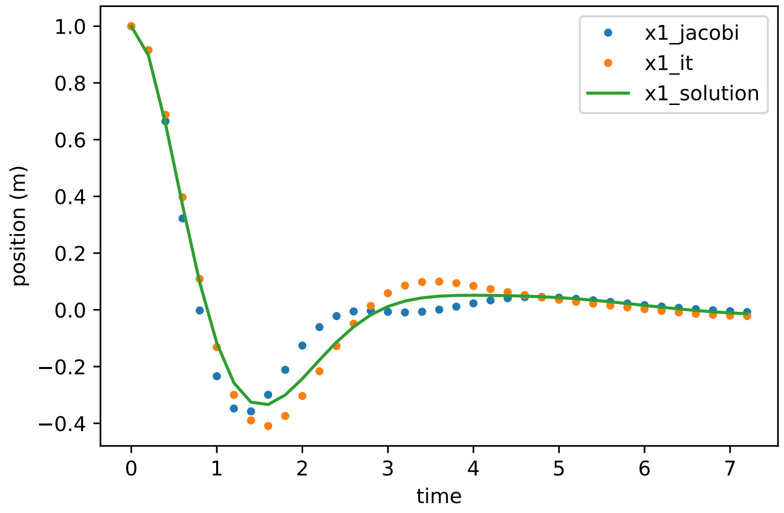

Figure 8: Coupled system behavior and corresponding co-simulation. Parameters are: $m_1 = m_2 = d_1 = c_c = d_c = 1, H = 0.2, c_1 = 5.0, c_2 = 0.1$.

3.2.1 Orchestration

In the following, we introduce the *Gauss-Seidel* and *Jacobi* orchestration algorithms, named after the analogous techniques to solve linear systems. To explain these methods, we need to first detail the elements that comprise a co-simulation scenario.

Let $H > 0$ denote the given communication time step. We denote the i-th communication time as $t_i = iH$. We say that the i-th step of the co-simulation is finished when all the numerical methods have computed their solutions up to, and including, time t_i.

Each model is associated with a reference $w \in D$, where D is a set of all model names. The model w is an IVP with output:

$$\begin{aligned} \dot{\boldsymbol{x}}_{[w]} &= F_{[w]}(\boldsymbol{x}_{[w]}, \boldsymbol{u}_{[w]}), \text{ with } \boldsymbol{x}_{[w]}(0) = \boldsymbol{x}_{0_{[w]}}, \text{ and} \\ \boldsymbol{y}_{[w]} &= G_{[w]}(\boldsymbol{x}_{[w]}, \boldsymbol{u}_{[w]}), \end{aligned} \tag{9}$$

where $\boldsymbol{y}_{[w]}$ denotes the output vector, and $G_{[w]}$ the output function.

As described in Section 3.1.3, the input function $\boldsymbol{u}_{[w]}(t)$ is an approximation constructed from samples of the outputs of other models. We will denote the set of models whose output is used to construct the input $\boldsymbol{u}_{[w]}(t)$, as $S_{[w]} \subseteq D$, standing for *Source models*. With this notation, for $t \in [t_i, t_{i+1}]$, the input $\boldsymbol{u}_{[w]}(t)$ is constructed from the samples of the outputs of every model $v \in S_{[w]}$ at the current and previous co-simulation steps. The number of samples needed depend on the concrete approximation technique.

We will use w to refer both to the model and the simulation unit, when there is no ambiguity. Roughly, the task of the orchestrator at time t_i is to provide the output samples that each unit w needs, and ask the unit to approximate the value of $\boldsymbol{y}_{[w]}(t_{i+1})$.

Example 4. *The co-simulation scenario corresponding to the coupled mass-spring-damper systems introduced in Examples 2 and 3 is illustrated in Figure 2. The input to the simulation unit constructed from Example 2 is F_c, as computed by the unit constructed from Example 3. The input to the unit of Example 3 is $\begin{bmatrix} x_1 \\ v_2 \end{bmatrix}^T$ of Example 2.*

3.2.2 Gauss-Seidel Orchestrator

Algorithm 1: Gauss-Seidel orchestrator. See Figure 3.

```
Data: The stop time $T$, a communication step size $H$, a co-simulation scenario with unit references $D$, and their
      order $\sigma$.
$t := 0$ ;                                                                    // Simulation time
// Initialize variables
for $w \in D$ do
    $\boldsymbol{uc}_{[w]} := \boldsymbol{y}_{[w]} := \mathbf{0}$ ;                                         // Current I/O variables.
    $\boldsymbol{up}_{[w]} := \mathbf{0}$ ;                                                  // Previous input variables.
end
// Compute initial outputs
for $j = 1, \ldots, |D|$ do
    $w := \sigma(j)$;
    $\boldsymbol{uc}_{[w]} := C_w\left(\left\{\boldsymbol{y}_{[v]} | v \in S_{[w]}\right\}\right)$;                 // Compute input from set of sources.
    $\boldsymbol{y}_{[w]} :=$ getOutput($w, \boldsymbol{uc}_{[w]}$);                                      // Compute output.
    $\boldsymbol{up}_{[w]} := \boldsymbol{uc}_{[w]}$;
end
while $t < T$ do
    for $j = 1, \ldots, |D|$ do
        $w := \sigma(j)$;
        $\boldsymbol{uc}_{[w]} := C_w\left(\left\{\boldsymbol{y}_{[v]} | v \in S_{[w]}\right\}\right)$;
        doStep($w, H, \boldsymbol{uc}_{[w]}, \boldsymbol{up}_{[w]}$);              // Compute $\boldsymbol{x}_{[w]}(t+H)$ from $\boldsymbol{x}_{[w]}(t)$ and inputs.
        $\boldsymbol{y}_{[w]} :=$ getOutput($w, \boldsymbol{uc}_{[w]}$);
    end
    for $w \in D$ do
        $\boldsymbol{up}_{[w]} := \boldsymbol{uc}_{[w]}$;                                          // Update previous input.
    end
    $t := t + H$;                                                                // Advance time
end
```

The Gauss-Seidel orchestrator requires an order between the units to be established. We denote the order with a map $\sigma : \mathbb{N} \to D$, that returns the unit reference $\sigma(j)$ that is the j-th in the order. For example, the unit $\sigma(1)$ is the first. A more detailed discussion regarding how to sort the simulation units is given in the extended version of this tutorial (Gomes et al. 2018).

With this notation, the Gauss-Seidel orchestrator is summarized in Algorithm 1. Function

$$C_w\left(\left\{\boldsymbol{y}_{[v]} | v \in S_{[w]}\right\}\right)$$

computes the input sample of unit w from the output samples of its sources. The function getOutput($w, \boldsymbol{uc}_{[w]}$) asks unit w to compute the output, optionally using the value in the variable $\boldsymbol{uc}_{[w]}$ Likewise, function doStep($w, H, \boldsymbol{uc}_{[w]}, \boldsymbol{up}_{[w]}$) asks unit w, assumed to be in state $\boldsymbol{x}_{[w]}(t)$, to compute the value $\boldsymbol{x}_{[w]}(t+H)$, using either one of the variables provided. Any other previous inputs the unit may require are assumed to be stored in its internal state (collected from previous calls to the doStep function.

The accompanying material (Gomes, Cláudio 2019) contains interactive example applications of Algorithm 1.

3.2.3 Jacobi Orchestrator

The main difference between the Jacobi and Gauss-Seidel orchestrator lies in the fact that the Jacobi orchestrator does not impose an order on the simulation units. This has a couple of consequences:

Algorithm 2: Jacobi orchestrator. See Figure 3.

```
Data: The stop time $T$, a communication step size $H$, a co-simulation scenario with unit references $D$, and the
      order $\sigma$ of their inputs.
$t := 0$ ;                                                                  // Simulation time
// Initialize variables
for $w \in D$ do
    $\boldsymbol{uc}_{[w]} := \boldsymbol{y}_{[w]} := \boldsymbol{0}$ ;                                            // Current I/O variables.
end
while $t < T$ do
    // Compute outputs in order
    for $j = 1, \ldots, |D|$ do
        $w := \sigma(j)$;
        $\boldsymbol{uc}_{[w]} := C_w\left(\left\{\boldsymbol{y}_{[v]} | v \in S_{[w]}\right\}\right)$;
        $\boldsymbol{y}_{[w]} :=$ getOutput($w, \boldsymbol{uc}_{[w]}$);
    end
    for $w \in D$ do
        doStep($w, H, \boldsymbol{uc}_{[w]}$);                // Compute $\boldsymbol{x}_{[w]}(t+H)$ from $\boldsymbol{x}_{[w]}(t)$ and inputs.
    end
    $t := t + H$;                                                           // Advance time
end
```

- The units can still be ordered for the invocations of the `getOutput` functions.
- There is no need to keep track of the previous inputs to each unit.

The Jacobi orchestrator is summarized in Algorithm 2. Compared to the Gauss-seidel orchestrator, the Jacobi is in general less accurate (due to the fact that units cannot use interpolation techniques), but can take advantage of parallelism.

Figure 8 shows the results of applying the Jacobi orchestration algorithm to Example 4.

3.2.4 Implicit and Semi-Implicit Orchestrators

The Jacobi and Gauss-Seidel orchestration algorithms have iterative counterparts (recall Figure 1). An iterative orchestration algorithm will retry each co-simulation step multiple times. If the number of repetitions is fixed, then we say that the orchestration is *semi-implicit*. If, on the other hand, the co-simulation step is repeated until some criteria is met, then the orchestration is *implicit*.

In general, iterative techniques are useful when the non-iterative techniques fail to preserve the stability of the original IVP, or when there are algebraic loops in the co-simulation scenario. When there are algebraic loops, then the units cannot be sorted, as assumed in Section 3.2.2. A more detailed discussion regarding algebraic loops is given in the extended version of this tutorial (Gomes et al. 2018).

Algorithm 3 illustrates the iterative version of the Gauss-Seidel orchestrator. Function `hasConverged` encodes the test for convergence, which can either count a fixed number of iterations (semi-implicit method), or check whether the output values have converged (implicit method). The `rollback` function reverts the state of the simulation unit to the one before the most recent call to the `doStep` function.

The iterative version of the Jacobi algorithm is similar, so we omit it. Figure 8 shows the results of applying the iterative Jacobi orchestration algorithm to Example 4. The iterative Gauss-Seidel algorithm produces similar results.

Comparing the results in Figure 8, one can see that the iterative version of the algorithm performs slightly better. However, it takes longer to execute.

Algorithm 3: Iterative Gauss-seidel orchestrator. See Figure 3.

```
Data: The stop time $T$, a communication step $H$, a scenario with unit references $D$, and their order $\sigma$.
$t := 0$ ;                                                        // Simulation time
// Initialize variables
for $w \in D$ do
    $\boldsymbol{uc}_{[w]} := \boldsymbol{y}_{[w]} := \boldsymbol{0}$ ;                                    // Current I/O variables.
    $\boldsymbol{up}_{[w]} := \boldsymbol{aux}_{[w]} := \boldsymbol{0}$ ;                       // Previous and auxiliary I/O variables.
end
// Compute initial outputs
$converged := \texttt{FALSE}$;
while $t < T$ do
    for $j = 1, \ldots, |D|$ do
        $w := \sigma(j)$;
        $\boldsymbol{uc}_{[w]} := C_w\left(\left\{\boldsymbol{y}_{[v]} | v \in S_{[w]}\right\}\right)$;              // Compute input from set of sources.
        $\boldsymbol{y}_{[w]} := \texttt{getOutput}(w, \boldsymbol{uc}_{[w]})$;                                    // Compute output.
        $\boldsymbol{up}_{[w]} := \boldsymbol{uc}_{[w]}$;
    end
    if $\mathit{hasConverged}\left(\left\{(\boldsymbol{uc}_{[w]}, \boldsymbol{aux}_{[w]}) | w \in D\right\}\right)$ then
        $converged := \texttt{TRUE}$;
    else
        $\boldsymbol{aux}_{[w]} := \boldsymbol{uc}_{[w]}$ for each $w \in D$;
    end
end
while $t < T$ do
    $converged := \texttt{FALSE}$;
    for $j = 1, \ldots, |D|$ do
        $w := \sigma(j)$;
        $\boldsymbol{uc}_{[w]} := C_w\left(\left\{\boldsymbol{y}_{[v]} | v \in S_{[w]}\right\}\right)$;
        $\texttt{doStep}(w, H, \boldsymbol{uc}_{[w]}, \boldsymbol{up}_{[w]})$;        // Compute $\boldsymbol{x}_{[w]}(t+H)$ from $\boldsymbol{x}_{[w]}(t)$ and inputs.
        $\boldsymbol{y}_{[w]} := \texttt{getOutput}(w, \boldsymbol{uc}_{[w]})$;
    end
    if $\mathit{hasConverged}\left(\left\{(\boldsymbol{uc}_{[w]}, \boldsymbol{aux}_{[w]}) | w \in D\right\}\right)$ then
        $converged := \texttt{TRUE}$;
        $\boldsymbol{up}_{[w]} := \boldsymbol{uc}_{[w]}$ for each $w \in D$;                              // Update previous input.
    else
        $\boldsymbol{aux}_{[w]} := \boldsymbol{uc}_{[w]}$ for each $w \in D$;
        $\texttt{rollback}(w)$ for each $w \in D$;                            // Cancel the effects of doStep.
    end
    $t := t + H$;                                                            // Advance time
end
```

4 SUMMARY

Co-simulation allows us to apply the best numerical method to each part of a given IVP. This is not the only benefit though. For example, each numerical method can use a different step size. This is an advantage because different models may evolve with derivatives that are orders of magnitude apart. Another benefit is that simulation units do not have to disclose the equations being solved internally. Instead, it is common to only disclose the outputs and inputs, capabilities such as the ability to rollback, and the derivatives of outputs with respect to time and inputs. The black box nature of the units makes it easier to standardize their interface, which in turn enables the coupling of mature modeling and simulation tools. Indeed, wide industrial adoption is one of the main drivers behind research into co-simulation (Schweiger et al. 2018).

This tutorial aimed at introducing the fundamental concepts in co-simulation, and providing researchers and practitioners with the basic knowledge to start developing their own simulation units and master algorithms. The accompanying online material (Gomes, Cláudio 2019) provides a starting point for users to run their own co-simulations and the extended version of this tutorial (Gomes et al. 2018) provides more theoretical background.

References

Alvarez Cabrera, A. A., K. Woestenenk, and T. Tomiyama. 2011. “An Architecture Model to Support Cooperative Design for Mechatronic Products: A Control Design Case”. *Mechatronics* 21(3):534–547.

Blochwitz, T., M. Otter, M. Arnold, C. Bausch, C. Clauss, H. Elmqvist, A. Junghanns, J. Mauss, M. Monteiro, T. Neidhold, D. Neumerkel, H. Olsson, J.-V. Peetz, and S. Wolf. 2011. “The Functional Mockup Interface for Tool Independent Exchange of Simulation Models”. In *Proceedings of the 8th International Modelica Conference*: Linköping, Sweden: Linköping University Electronic Press; Linköpings universitet. 105-114.

FMI. 2014. “Functional Mock-up Interface for Model Exchange and Co-Simulation”. https://fmi-standard.org/downloads/, accessed 15th September 2019.

Gomes, Cláudio 2019. “Online Materials”. https://msdl.uantwerpen.be/git/claudio/2019.WinterSim.CosimTutorialMaterials, accessed 15th September 2019.

Gomes, C., C. Thule, D. Broman, P. G. Larsen, and H. Vangheluwe. 2018. “Co-Simulation: A Survey”. *ACM Computing Surveys* 51(3):Article 49.

Gomes, C., C. Thule, P. G. Larsen, J. Denil, and H. Vangheluwe. 2018. “Co-Simulation of Continuous Systems: A Tutorial”. Technical Report arXiv:1809.08463, University of Antwerp, Belgium.

Hafner, I., and N. Popper. 2017. “On the Terminology and Structuring of Co-Simulation Methods”. In *Proceedings of the 8th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*, edited by D. Zimmer and B. Bachmann: New York, USA: Association for Computing Machinery Press. 67-76.

IEEE. 2014. “IEEE 1516 High Level Architecture”. https://standards.ieee.org/standard/1516-2010.html, accessed 15th September 2019.

Kang, K. C., S. Cohen, J. Hess, W. Novak, and A. Peterson. 1990. “Feature-Oriented Domain Analysis. Feasibility Study”. Technical Report CMU/SEI-90-TR-21, Carnegie Mellon University, Pittsburgh, PA, USA. https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=11231, accessed 15th September 2019.

Nielsen, C. B., P. G. Larsen, J. Fitzgerald, J. Woodcock, and J. Peleska. 2015. “Systems of Systems Engineering: Basic Concepts, Model-Based Techniques, and Research Directions”. *ACM Computing Surveys* 48(2):18:1–18:41.

Palensky, P., A. A. Van Der Meer, C. D. Lopez, A. Joseph, and K. Pan. 2017. “Cosimulation of Intelligent Power Systems: Fundamentals, Software Architecture, Numerics, and Coupling”. *IEEE Industrial Electronics Magazine* 11(1):34–50.

Plateaux, R., J. Choley, O. Penas, and A. Riviere. 2009. “Towards an Integrated Mechatronic Design Process”. In *Proceedings of the 2009 IEEE International Conference on Mechatronics*, Volume 00: Piscataway, New Jersey: Institute of Electrical and Electronics Engineers. 1-6.

Schweiger, G., C. Gomes, G. Engel, I. Hafner, J. Schoeggl, A. Posch, and T. Nouidui. 2018. “Functional Mock-up Interface: An Empirical Survey Identifies Research Challenges and Current Barriers”. In *Proceedings of the American Modelica Conference*: Linköping, Sweden: Linköping University Electronic Press, Linköpings Universitet. 138-146.

Schweiger, G., C. Gomes, G. Engel, I. Hafner, J.-P. Schoeggl, A. Posch, and T. Nouidui. 2019. “An Empirical Survey on Co-Simulation: Promising Standards, Challenges and Research Needs”. *Simulation Modelling Practice and Theory* 95:148–163.

Schweiger, G., C. Gomes, I. Hafner, G. Engel, T. S. Nouidui, N. Popper, and J.-P. Schoggl. 2018. “Co-Simulation: Leveraging the Potential of Urban Energy System Simulation”. *EuroHeat&Power* 15(I-II):13–16.

Van der Auweraer, H., J. Anthonis, S. De Bruyne, and J. Leuridan. 2013. “Virtual Engineering at Work: The Challenges for Designing Mechatronic Products”. *Engineering with Computers* 29(3):389–408.

Vangheluwe, H., J. De Lara, and P. J. Mosterman. 2002. “An Introduction to Multi-Paradigm Modelling and Simulation”. In *Proceedings of the AI, Simulation and Planning in High Autonomy Systems Conference*: San Diego, CA, USA: Society for Computer Simulation International. 9-20.

AUTHOR BIOGRAPHIES

CLÁUDIO GOMES is a PhD student at the University of Antwerp (Belgium). The topic of his PhD is the foundations of co-simulation. His email address is claudio.gomes@uantwerp.be and his web address is http://msdl.cs.mcgill.ca/people/claudio.

HANS VANGHELUWE is a Professor in the Department of Mathematics and Computer Science of the University of Antwerp (Belgium). He heads the Modelling, Simulation and Design (MSDL) research lab. His research interest is the multi-paradigm modeling of complex, software-intensive, cyber-physical systems. His email address is: hans.vangheluwe@uantwerp.be and his web address is http://msdl.cs.mcgill.ca/people/hv.