Co-simulation at Different Levels of Expertise with Maestro2

Simon Thrane Hansen[a,*], Casper Thule[a], Cláudio Gomes[a], Kenneth Guldbrandt Lausdahl[a], Frederik Palludan Madsen[a], Giuseppe Abbiati[b], and Peter Gorm Larsen[a]

[a]*Department of Electrical and Computer Engineering, Aarhus University, Åbogade 34, Denmark*
[b]*Department of Civil and Architectural Engineering, Aarhus University, Inge Lehmanns Gade 10, Denmark*

Abstract

When different simulation units are coupled together there are different choices to take, in particular regarding the granularity of such a co-simulation. When prototyping systems, it is typically favourable to get an initial idea of how a collection of simulation units work together without spending too much time setting up the orchestration. However, the granularity of such a simulation may be far away from what is needed in relation to the purpose of the simulation. In order to enable more flexibility and control over the co-simulation it is necessary to be able to steer the orchestration in a more detailed manner. This paper presents an open source co-simulation orchestration engine based on the Functional Mockup Interface standard but with a Domain Specific Language (DSL) enabling detailed control between the individual simulation units. The same tool can thus be used right out of the box for low-granularity co-simulation, and for high-granularity simulation the DSL enable a significant flexibility.

Keywords: co-simulation framework, domain specific language, functional mockup interface standard

1. Introduction

Co-simulation is a technique for simulating complex systems by combining multiple simulation tools into a single simulation Kübler and Schiehlen (2000b); Gomes et al. (2018b)

Interoperability between simulation tools is achieved through the use of Functional Mock-up Units (FMUs) defined by the Functional Mock-up Interface (FMI) standard Committee (2014, 2021). An FMU encapsulates a Simulation Unit (SU) by providing a standardised interface of inputs, outputs, and functions to let a *co-simulation framework* control the simulation of a coupled system of FMUs, referred to as a *scenario*. A scenario is obtained by coupling inputs and outputs of the FMUs in the scenario, as illustrated in Figure 1. A coupling denotes that the output FMU's state influences the input FMU's state. Figure 1 depicts a scenario with two coupled FMUs of a coupled mass-spring-damper, similar to a later example in Section 4.

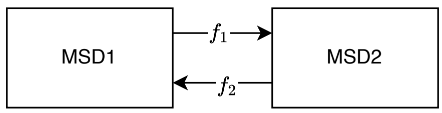

Figure 1: A co-simulation scenario with two SUs `MSD1` and `MSD2` representing a coupled mass-spring-damper system. The SUs are represented as rectangles, and the arrows f_1 and f_2 denote the connections between the SUs.

A co-simulation framework executes the scenario by computing the joint behaviour of the system by coordinating the execution of the FMUs in the scenario according to an orchestration algorithm (OA). The OA describes how stimuli are exchanged between the FMUs in the scenario and how the state of the FMUs evolves over the course of the co-simulation. Although the OA is not part of the FMI standard, it is a critical component of a co-simulation framework, as studies have shown that the OA can significantly affect the accuracy of co-simulation results Busch (2016); Kalmar-Nagy and Stanciulescu (2014); Schweizer et al. (2015); Arnold (2010); Gomes et al. (2018e); Schweizer et al. (2016); Andersson (2016); Hansen et al. (2021b). To obtain accurate co-simulation results, the OA must be tailored to the scenario and the characteristics of the FMUs it contains, as illustrated in Figure 2, which compares the results of two different OAs for the scenario presented in Figure 1 with its analytical solution. Both OAs are compliant with the FMI standard, but the results differ

*Corresponding author
Email address: `sth@ece.au.dk` (Simon Thrane Hansen)

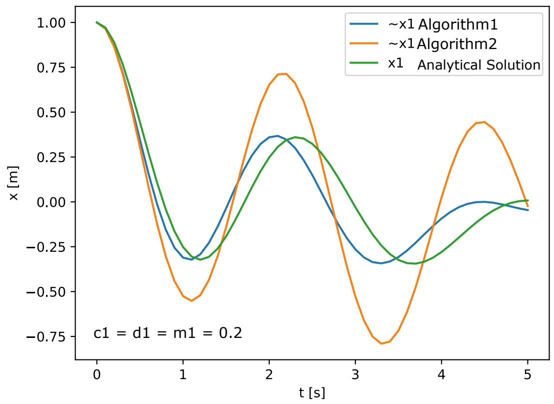

Figure 2: Comparison of the results of two different OAs for the scenario in Figure 1 with its analytical solution. The step size used in the co-simulation is 0.1 s.

significantly.

Figure 2 shows that the OA significantly affects the accuracy of the co-simulation results and suggests that the OA should be carefully designed to minimise the error in a co-simulation. Similar results have been reported in Busch (2016); Kalmar-Nagy and Stanciulescu (2014); Schweizer et al. (2015); Arnold (2010); Gomes et al. (2018e); Schweizer et al. (2016); Andersson (2016); Hansen et al. (2021b), where the authors show empirically that co-simulation results can be highly sensitive to the order in which the FMUs are simulated and how they exchange data.

The sensitivity can be attributed to several factors, but can be summarised as follows: The discrete nature of co-simulation, where data is exchanged between FMUs at discrete points in time, called communication points, can be challenging for FMUs representing continuous processes. Such FMUs typically rely on numerical solvers (variable step or fixed-step) to advance in simulated time and use a variety of approximation techniques, such as extrapolation and interpolation, to reason about the values of the FMU's inputs between communication points Kübler and Schiehlen (2000a); Gomes et al. (2018b). The approximation techniques used by the FMUs impose constraints on the OA interactions, dictating the order in which the FMUs are simulated and how they exchange data to account for the characteristics of the FMUs Gomes et al. (2019b); Hansen et al. (2022b).

To address these challenges and cater to a wide range of application domains, a co-simulation framework should strike a balance between allowing experts to customise the OA and providing synthesised OAs to new users, while trying to minimise the overhead of orchestrating the co-simulation. This balance empowers experts to fine-tune the OA to minimise co-simulation error, while enabling new users to quickly start co-simulating without having to delve into intricate details.

Unfortunately, to the best of our knowledge, there is a lack of open source co-simulation frameworks that offer such flexibility without compromising on usability and performance. As a result, users who wish to customise/fine-tune the OA to accommodate the idiosyncrasies of the SUs, or researchers who wish to experiment with novel co-simulation algorithms, are left with having to develop their own OA from scratch using low-level co-simulation libraries. Developing an OA for a large scenario is a time-consuming task that requires the user to spend many hours laying the groundwork before they even reach the point where they can tune their co-simulation, as the user must code all interactions with the SUs.

In summary, there is a need for a co-simulation framework that provides flexibility while maintaining customisability, expressiveness, verifiability, and speed. Such a framework should strike a balance between providing experts with customisable OAs and providing new users with synthesised OAs, allowing different levels of granularity for co-simulation practitioners at all levels of expertise.

Contribution. To address the above issues, we propose an open source co-simulation framework called Maestro2, which leverages the latest advances in OA synthesis Gomes et al. (2019b); Hansen et al. (2022b) to enable rapid development of customisable OAs and use code generation to enable high-performance co-simulations. Specifically, Maestro2 provides different levels of granularity to describe the co-simulation scenario and the OA, ranging from a high level of granularity suitable for prototyping and new users, to a lower level DSL that describes the OA in detail, suitable for experts fine-tuning the OA to accommodate the idiosyncrasies of the FMUs.

Thus, the main contribution of this manuscript is the co-simulation framework Maestro2, which enables the different levels of granularity to support co-simulation practitioners with different levels of expertise. The manuscript also presents the results of the application of Maestro2 in two case studies.

Prior Work. This manuscript represents the complete implementation of the idea originally proposed in Thule et al. (2020) to maximise reuse among co-simulation frameworks. Since then, Maestro2 has been used in several case studies and research projects (see Section 4).

Among other, Maestro2 has been used to experiment with different approaches for handling algebraic loops during the initialisation of a co-simulation Hansen et al. (2021c). While the above works introduce Maestro2, none of them introduces the Maestro2 approach in detail, showcasing how it empowers users of different levels of expertise and with different needs to perform co-simulations. The Maestro2 approach is the main contribution of this manuscript.

Structure. The remainder of this manuscript is structured as follows: The next section introduces the main concepts used throughout the manuscript, including the FMI standard, the concept of co-simulation, and the concept of orchestration algorithms. Then, Section 3 presents the Maestro2 approach and how it enables different levels of granularity. Section 4 summarises a case study where Maestro2 has been applied and Section 5 discusses related work. Finally, Section 6 details future work and concludes the manuscript.

2. Background

This section serves as an introduction to the fundamental concepts of co-simulation, the FMI standard, and OAs, and provides an overview of the problem we aim to address. However, due to the breadth and complexity of co-simulation and the FMI standard, and the interested reader is referred to Gomes et al. (2018b,d) for a comprehensive introduction to co-simulation and Blochwitz et al. (2011); Committee (2014); Gomes et al. (2021b) for the FMI standard. The notation and definitions introduced in this section are adopted from Gomes et al. (2019a); Hansen and Ölveczky (2022).

2.1. Co-simulation

Co-simulation is a technique that enables the global simulation of a system composed of several black-box SUs, typically developed individually or exported from different tools (Kübler and Schiehlen (2000b); Gomes et al. (2018b)). An SU models the behaviour of a *dynamic system* consisting of inputs and outputs, the state of the system, and a set of functions to provide stimuli to the system (by setting the inputs), to retrieve the state of the system (by getting the outputs), and to evolve the state of the system in simulated time (by stepping the model).

The evolution of an SU is governed by a set of evolution rules described by differential and algebraic equations that define how the state of the system changes in response to stimuli and the current state of the system. The behaviour trace of an SU is a function that maps time to state, and the evolution rules are typically obtained using a numerical solver that discretizes the continuous-time model of the SU into a discrete-time model (a trace that maps time to state), allowing the simulation of the SU in discrete time steps.

A formal definition of an SU is given in Definition 1, where the evolution rules are captured by the function $\texttt{doStep}_c$, which advances the state of the SU in simulated time. In addition, the function $\texttt{doStep}_c$ returns a step size to accommodate those SUs that use numerical solvers with variable step lengths or implement error estimation mechanisms. These SUs may conclude that a step size of H will result in an intolerable error.

Definition 1. *An SU with identifier c is represented by the tuple* $\langle S_c, U_c, Y_c, \texttt{set}_c, \texttt{get}_c, \texttt{doStep}_c \rangle$*, where:*

- S_c *represents the state space.*
- U_c *and* Y_c *the set of input and output variables, respectively.*
- $\texttt{set}_c : S_c \times U_c \times \mathcal{V} \to S_c$ *and* $\texttt{get}_c : S_c \times Y_c \to \mathcal{V}$ *are functions to set the inputs and get the outputs, respectively (we abstract the set of values that each input/output variable can take as* $\mathcal{V}$*).*
- $\texttt{doStep}_c : S_c \times \mathbb{R}_{>0} \to S_c \times \mathbb{R}_{>0}$ *is a function that instructs the SU to compute its state after a given time duration. If an SU is in state* $s_c^{(t)}$ *at time* t*,* $(s_c^{(t+h)}, h) = \texttt{doStep}_c(s_c^{(t)}, H)$ *approximates the state* $s_c^{(t+h)}$ *of the corresponding model at time* $t+h$*, with* $h \leq H$*.*

A collection of SUs can be coupled to form a co-simulation scenario by connecting the outputs of one SU to the inputs of another SU (see Definition 2). A coupling means that the state of one SU always depends on the state of another SU - this is called a *coupling restriction* and can be thought of as a system invariant, which says that the values of the coupled inputs and outputs must always be equal. Nevertheless, the coupling restrictions are only satisfied at specific points in time, called *communication points*, when the SUs exchange data. The SUs try to compensate for the inconsistency between the communication points by making assumptions about the evolution of the values on their inputs. Obviously, these assumptions can cause a significant error in the co-simulation for large intervals between the communication points. In fact, they can be the main source of error (Arnold et al. (2014)), so it is essential to fine-tune the OA to account for the characteristics of the SUs. The characteristics of these approximation functions are captured by the function R, see Definition 2.

The function R links each input to indicate whether the input SU expects the coupled output to be simulated before or after the input SU itself.

Definition 2 (Scenario). *A scenario is a structure* $\langle C, L, R\rangle$*, where*

- C *is a finite set (of SU identifiers).*
- L *is a function* $L : U \to Y$*, where* $U = \bigcup_{c\in C} U_c$ *and* $Y = \bigcup_{c\in C} Y_c$*, and where* $L(u) = y$ *means that the output* y *is coupled to the input* u*.*
- $R : U \to \mathbb{B}$ *is a predicate, which describes the SUs' input approximation functions.* $R(u) = true$ *means that SU* c *expect the SU* d *of the output* y *coupled to* u *to be simulated before* c*. similarly,* $R(u) = false$ *means that SU* c *expect the SU* d *of the output* y *coupled to* u *to be simulated after* c*.*

To illustrate the concepts introduced in Definitions 1 and 2, consider the scenario in Figure 3 and the corresponding behavioural trace shown in Figure 4. The scenario consists of two SUs, a `controller SU` and a `tank SU`, and two couplings, one from the output `valve state` of the controller SU to the input `valve state` of the tank SU, and one from the output `water level` of the tank SU to the input `water level` of the controller SU. Each SU has some parameters (e.g. `max level` and `min level`) which are used to configure the simulation. The function R is omitted from the scenario, as it is not part of the FMI standard, and is discussed in Section 2.2.

The controller SU is a simple controller that opens or closes a valve based on the current water level, and the tank SU is a simple tank that keeps track of the current water level based on the flow of water in and out of the tank through the valve. The behaviour trace of the scenario is in Figure 4 and is obtained by simulating the scenario from time 0 to 10 seconds. The trace is the function σ that maps time to the state of the scenario, i.e. $\sigma(t) = \left\langle s_c^{(t)} \mid c \in C\right\rangle$.

The *co-simulation error* is the difference between the simulated behaviour of the scenario and its ideal behaviour, i.e. the behaviour obtained by simulating the scenario with an infinitely small step size or by solving the differential equations analytically, which is generally not possible.

The simulation of a scenario is controlled by the OA, which is the algorithm coordinating the execution of the SUs in the scenario to obtain the joint behavioural trace of the system. The OA comprises multiple stages, including everything from loading the SUs to terminating the simulation and performing the co-simulation by invoking the $\texttt{set}_c$, $\texttt{get}_c$, and $\texttt{doStep}_c$ functions defined in Definition 1. The stages of a typical OA are summarised in Figure 5, the colours are used to visually distinguish the three overarching phases of the OA: *initialisation*, *simulation*, and *termination*. The initial set of phases *Start*, *Instantiate*, *Setup* and *Initialise* are executed once before the simulation starts and are responsible for loading the SUs, creating instances, setting initial parameters, and computing the initial state of the SUs, respectively. The initialisation is followed by the simulation loop, which consists of the *Step* and *Plotting* stages, which are responsible for advancing the SUs in simulated time and reporting results. Finally, the stage *Free* releases resources and terminates the simulation.

Many articles Broman et al. (2013a); Hansen et al. (2022b); Gomes et al. (2018a) on OAs do only consider the *Initialise* and *Step* stages, as these are the most critical for the correctness of the co-simulation results and constitutes the phases where the OA interacts with the

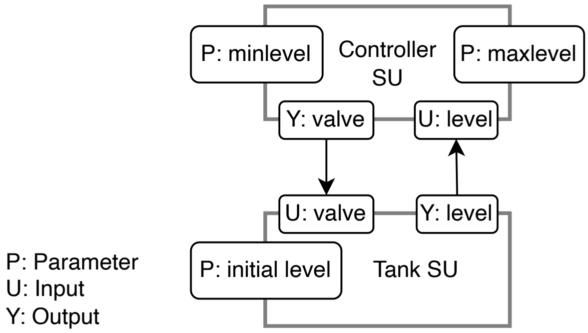

Figure 3: The the co-simulation scenario of the water tank example illustrated as a block diagram. The scenario is adapted from Mansfield et al. (2017).

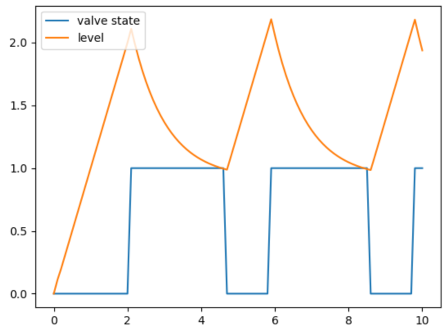

Figure 4: The behavioural trace of the Water Tank example shown as a plot of the water level and valve state over time for a simulation from 0 to 10 seconds with a step size of 0.1 seconds. The scenario is adapted from Mansfield et al. (2017).

SUs using the $\texttt{set}_c$, $\texttt{get}_c$ and $\texttt{doStep}_c$ functions defined in Definition 1. These phases are concerned with satisfying the coupling restrictions and ensuring that the SUs move in lockstep, i.e. that the SUs are synchronised with respect to simulated time.

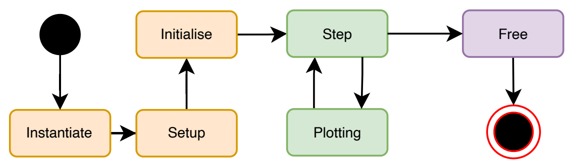

Figure 5: Generic OA structure. Each round rectangle represents a stage in the execution of the OA. For instance, the plotting stage will query the outputs of the SU and record them in a CSV file. Adapted from Thule et al. (2020).

For a better understanding of the OA stages *Initialise* and *Step*, consider the scenario in Figure 3. The parameters (`minimum water level`, `maximum water level` and `initial level`) of the SUs are set in the *Initialise* stage, after which the initial state of the SUs is calculated by calling the functions `get` and `set` to get the valve state output from the controller SU and set it to the valve state input on the water level SU. Then the water level output from the tank SU is assigned to the water level input on the controller SU. The simulation then begins. The outputs are retrieved and logged at time 0 in the *Plotting* step. The *Step* stage uses an algorithm similar to the algorithm shown in Algorithm 1 to compute new states for the SUs by calling the `doStep` function and to exchange data between the SUs by calling the `get` and `set` functions.

Algorithm 1 Step algorithm for the watertank scenario in Figure 3.

doStep(*tank*,0.1) ▷ Advance the tank SU by 0.1s
level′ ← getOut(*tank*,*level*) ▷ Get water level of tank
setIn(*ctr*,*level*,*level*′) ▷ Set water level on controller
doStep(*ctr*, 0.1) ▷ Advance the controller SU by 0.1s
valve′ ← getOut(*ctr*,*valve*) ▷ Get valve state of controller
setIn(*tank*,*valve*,*valve*′) ▷ Set valve state on tank

Algorithm 1 shows the order in which the SUs are simulated and how and when data is exchanged between the SUs. The algorithm advances the tank SU by 0.1 second, retrieves the water level from the tank SU, sets the water level on the controller SU, advances the controller SU by 0.1 second, retrieves the valve state from the controller SU, and sets the valve state on the tank SU. We have deliberately omitted the error handling and logging of the outputs for brevity, nevertheless a practical implementation of co-simulation will have to deal with such things. The *Plotting* stage is employed between each iteration of the *Step* stage to log the outputs of the SUs (`valve state` and `water level`) at the current communication point. This process is repeated until the simulation is terminated, which in this case is when the simulation time reaches 10 seconds. Once the simulation is finished, the *Free* stage is invoked to release resources and terminate the simulation.

Due to the numerous modelling and simulation tools that are capable of producing SUs, and the many *ad-hoc* co-simulation implementations (Gomes et al. (2018c)), the community has proposed a standard for the SU interface: the Functional Mockup Interface (FMI) Blochwitz et al. (2011); Committee (2014); Gomes et al. (2021b) standard to enable interoperability between SUs.

2.2. The FMI Standard

The Functional Mock-up Interface (FMI) standard is a tool-independent standard for the exchange of models and co-simulation, originally developed during the European ITEA2 project called MODELISAR (Blochwitz et al. (2011)). The standard provides and describes a complied C-interfaces, the structure of a static description file, called *ModelDescription*, and a way of packaging these into a zip file according to a predefined structure. Consequently, a component that implements the C-interfaces according to the rules of the FMI standard, compiles its model into a dynamic/shared library, provides a ModelDescription and packages these into a zip file according to a predefined structure, is called a Functional Mock-up Unit (FMU). FMUs can be exported from a variety of modelling and simulation tools, such as Dymola Brück et al. (2002), OpenModelica Fritzson (2015), and Simulink. They can be imported into a co-simulation framework, such as INTO-CPS Larsen et al. (2016), to be integrated with other FMUs in a co-simulation scenario. This summarises the main purpose of the FMI standard, which is to enable interoperability between modelling and simulation tools by providing a standardised interface for SUs, which is essential for the simulation of CPSs.

The *ModelDescription* file defines the interface of the FMU and contains, among other things, information about its inputs, outputs and parameters, called *ScalarVariables*. Each ScalarVariable has a type, an identifier, a causality, and a variability constraining how the value of the ScalarVariable can be obtained and changed during the simulation using a set of functions, defined by the FMI standard, analogous to the $\texttt{set}_c$, $\texttt{get}_c$, and $\texttt{doStep}_c$ functions defined in Definition 1. The FMI standard defines a set of functions for getting and setting the values of ScalarVariables, e.g. `fmi2GetReal` for `Real` and `fmi2GetInteger` for `Integer`. The FMI standard also defines a function to advance the state of the FMU in simulated time, e.g. `fmi2DoStep`.

Algorithm 1 shows how these FMI functions can be used to simulate the scenario in Figure 3. In Algorithm 1 we have deliberately chosen to simulate the tank SU before the controller SU to minimise the co-simulation error. However, the FMI standard does not provide any means to specify such constraints, so the definition of a scenario according to the FMI standard does not include the function R. Consequently, a co-simulation framework compliant with the FMI standard must either simulate the SUs in an arbitrary order or provide a mechanism for the user to specify the order in which the SUs are simulated. Our co-simulation framework provides the latter approach, thus providing the user with more control over the co-simulation, which is essential for fine-tuning the OA to minimise the co-simulation error.

The FMI standard for co-simulation aims to capture the common denominator of co-simulation and therefore strives for simplicity. However, this simplicity comes at a cost, as numerous studies (Gomes et al. (2019b); Oakes et al. (2021); Gomes et al. (2018e); Schweizer et al. (2015); Gomes et al. (2018a); Hansen et al. (2022b)) have shown how the accuracy of co-simulation results can be improved by tailoring the OA to the specific scenario by incorporating domain knowledge/implementation details of the SUs. For example, the OA can be adapted to take into account the implementation of the SUs, e.g. whether an SU interpolates or extrapolates an input, which is not captured by the standard.

By including such details, the OA can be adapted to improve the accuracy of the co-simulation results, as we show in Section 4. By incorporating such details, the OA can be customised to improve the accuracy of the co-simulation results, as we show in Section 4. Another limitation of the standard is in the context of network simulation, where the FMI export tool for the ns-3 network simulator supports simulation of purely discrete behaviour as it allows progression with 0 time (CES et al. (2021)), which is disallowed by the FMI standard[1].

Our goal is to provide a co-simulation framework that leverages both the strength of the FMI infrastructure and community while at the same time providing a framework more flexible than the FMI standard, which has to target a vast audience. We aim to provide a framework that can be customised to support advanced co-simulation based studies by incorporating experimental features and research results to support co-simulation practitioners at all levels of expertise. In the long term, we aim to improve the co-simulation support for the following simulation activities, each of which places specific requirements on the co-simulation frameworks.

Optimisation/DSE: Co-simulations are run as part of an optimisation loop, for example, in a Design Space Exploration (DSE) approach . This includes decision support systems, used, for example, in a digital twin (Glaessgen and Stargel (2012)) setting, where a modelled system is updated based on the operating system. Some of the specific requirements include: the ability to define co-simulation stop conditions, the ability to compute sensitivity, high performance, fully automated configuration, faster than real-time computation.

Certification: Co-simulation results can be used as a part of a certification endeavour. Requirements include fully transparent, and formally certified, synchronisation algorithms.

X-in-the-loop: Co-simulations include simulators that are constrained to progress in sync with the wall-clock time, because they represent human operators or physical subsystems.

Fault Injection: Co-simulations provide an additional test environment where all sorts of scenarios can be tested. Fault injection is a specific type of test where faults and other irregularities are injected into the system to investigate the system's behaviour under such conditions.

Last but not least, co-simulation tools have different audiences ranging between researchers, students, and industry from different domains. Each audience has different requirements and expectations from a co-simulation tool. At the same time, students and researchers are interested in transparency and customisation possibilities, while the industry is interested in plug-and-play, stable, scalable, and mature solutions with consumable interfaces. Consequently, we believe that the co-simulation framework, presented in the next section, meets the needs of all audiences by providing a low-level interface for students and researchers and a high-level interface for industry.

3. Co-simulation with Maestro2

This section describes the guiding principles behind Maestro2, namely the separation of concerns between the specification of a co-simulation and the execution of a co-simulation. The separation of concerns is achieved through the use of a Domain Specific Language (DSL) called MaBL, which specifies a co-simulation scenario

[1] Section 4.4.2, page 105 (FMI (2020)).

that can be analysed, verified and optimised prior to execution. The section begins with a brief introduction to the Maestro2 framework before introducing the MaBL DSL and how it is used to describe and analyse a co-simulation scenario. Finally, the section shows how Maestro2 uses code generation to provide a performant and flexible co-simulation engine.

3.1. Maestro2

Maestro2 is a co-simulation framework based on the FMI standard. It is written in Java and is available as open source software (`https://github.com/INTO-CPS-Association/maestro`). Co-simulation is performed by executing a MaBL specification, a "C-like" DSL for specifying co-simulation scenarios. The specification describes all the steps of the OA (see Figure 5), the FMUs involved and the connections between them. The general idea behind the invention and use of MaBL is to separate the specification of the OA from the execution of the OA. This need arises from the desire to analyse, verify and optimise the OA prior to execution, which was identified based on the experience with Maestro1 (Thule et al. (2019)).

The Maestro2 approach, illustrated in Figure 6, can be summarised as follows: The user can either write a MaBL specification manually or generate it using one of the approaches described in Section 3.3. The MaBL specification is then fine-tuned and analysed by a range of expansion plugins, which also can be used to expand the specification with additional functionality. Finally, the MaBL specification is executed using either the MaBL interpreter or a code generator, which generates a high-performance co-simulation engine in C++. Although Figure 6 depicts the Maestro2 approach as a linear process, it is possible to jump back and forth between the different phases to fine-tune the specification.

The following sections detail the different phases of the Maestro2 approach, starting with the specification phase. Nevertheless, before diving into these phases, we take a look at the MaBL DSL, the common denominator of the Maestro2 approach.

3.2. Maestro Base Language (MaBL)

The MaBL DSL is a "C-like" language that is used to specify a co-simulation scenario. A MaBL specification is a collection of modules, functions, and annotations that describe the OA. In the following, we give a brief introduction to the MaBL DSL using a series of small didactic examples. The reader can consult the online documentation for a more detailed description of the MaBL DSL (Association (d)).

Each MaBL specification must contain a entity called `simulation`, which is the entry point of the specification. The `simulation` block (line 8 in Listing 1) contains a set of imports (lines 9-10), which are used to import so-called runtime modules, and a set of annotations (lines 11-13), which are used to configure the simulation environment available to *expansion plugins* within the simulation, which are described below. Runtime modules are treated in Section 3.2.1, whereas expansion plugins are described in Section 3.3.1.

A module definition can also be part of a MaBL file, as exemplified by `module DataWriter` in Listing 1 line 1 – 6, which also includes yet another module (`DataWriterConfig`) in line 2.

Listing 1: MaBL Specification Structure

```
1  module DataWriter
2  import DataWriterConfig;
3  {
4    DataWriterConfig writeHeader( string  headers
       ↪  []) ;
5     ...
6  }
7
8  simulation
9  import FMI2;
10 import Logger;
11 @Framework( "FMI2");
12 @FrameworkConfig( "FMI2", "{...\"connections
       ↪  \":{\"{ crtl }. crtlInstance  . valve \":[\"{
       ↪  wt}. wtInstance . valvecontrol   \"],...}")  ;
13 {
14   //  Simulation  code  goes  here
15 }
```

MaBL is a statically and strongly typed language that incorporates the type system of the FMI 2.0 standard (`Real, Integer, Boolean, String`). It extends this type system with the `Array` type and introduces the `FMI2` type to represent an FMU. Additionally, runtime modules can introduce new types.

MaBL provides a range of built-in functions, including `load` and `unload`, which facilitate the loading and unloading of runtime modules, respectively. In terms of non-module functions, MaBL follows a minimalistic approach, offering basic arithmetic operations such as `+, -, *, /`, and fundamental Boolean operators such as `==, !=, !, >=, <=, <, >, &&, ||`. Furthermore, MaBL offers standard control flow constructs such as `if-else, while, try-finally` to describe the OA.

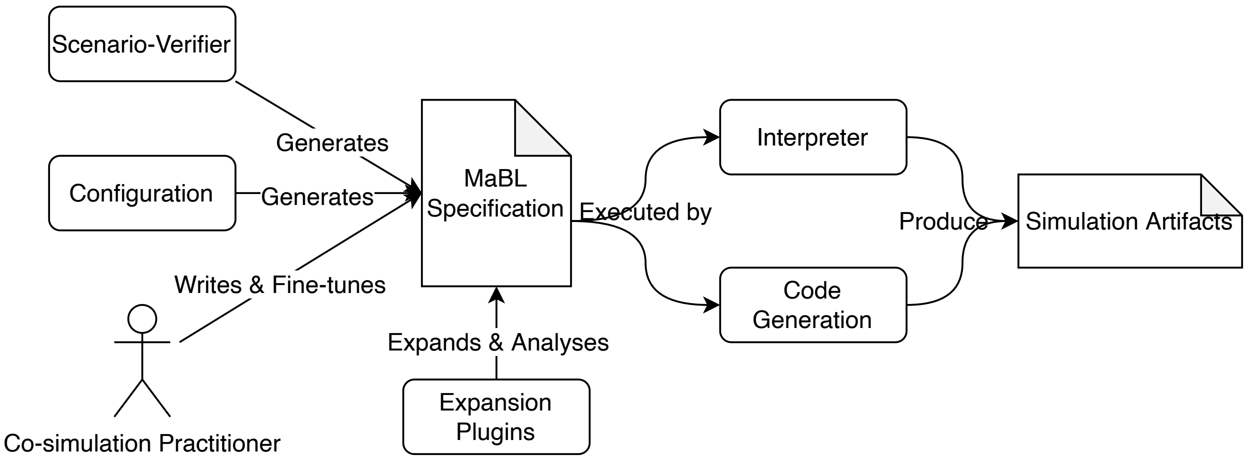

Figure 6: The Maestro2 is centred around the MaBL DSL. Maestro2 provides different approaches to generate a MaBL specification to enable co-simulation practitioners of different levels of expertise. A MaBL specification can be analysed by a range of expansion plugins. Finally, Maestro2 offers two approaches to execute a MaBL specification, namely the MaBL interpreter and a code generator.

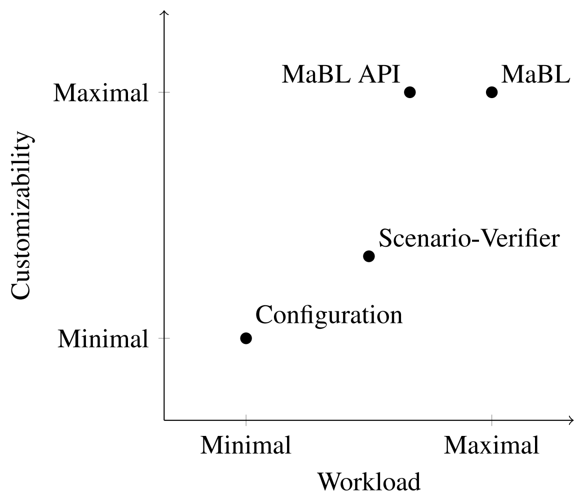

Figure 7: The different approaches to generate a MaBL specification in terms of workload and customizability.

3.2.1. Runtime Modules

More advanced features and functionality are typically implemented as runtime modules, which are loaded and executed during the execution of a MaBL specification through function calls.

A *runtime module* is a dynamically linked library that exposes a set of functions that can be called from MaBL to perform specific tasks that are not natively supported. For example, the FMI2 runtime module provides the FMI2 interface and offers various runtime module options. The "regular" runtime module unpacks an FMU, loads its dynamically linked library, and invokes its functions. On the other hand, the JFMI2 module allows loading an entity by specifying the class name instead of the FMU path. This is particularly useful for prototyping, as it enables development in other JVM-based languages such as Java, Kotlin, or Scala, bypassing the need for compilation into shared libraries and packages.

Another example of a runtime module is the Fault Injection module, which allows faults to be injected into the simulation with minimal effort. Specifically, the module wraps around an FMU and intercepts all data to and from the FMU, allowing it to modify the data before it is passed to the FMU based on a given configuration (Frasheri et al. (2021)).

3.3. Generation of Specifications

To cater for users with different levels of expertise, Maestro2 offers a variety of approaches to generating a MaBL specification as illustrated in Figure 6. The most basic approach is to manually write the specification in MaBL, which is a viable option for small specifications and experienced users. However, writing a large scenario manually is a tedious task, so it is desirable to generate the specification from other sources. Maestro2 offers the following approaches to generating a MaBL specification (based on the amount of work required (from most to least)): (i) Expansion Plugins, (ii) MaBL API, (iii) OA using the Scenario-Verifier, and (iv) Configuration.

These approaches vary in the degree of customisability and effort required to generate the specification, as illustrated in Figure 7. The expansion plugins are deliberately not included in the figure, as they are not a standalone approach, but rather a set of plugins that can be used to extend a MaBL specification generated using one of the other approaches. Nevertheless, all approaches are described below.

3.3.1. Expansion Plugins

MaBL specifications can be populated using an advanced set of expansion features called *expansion plugins*, which are community developed plugins that generate MaBL based on a given set of parameters provided by the user and the scenario at hand. A MaBL specification can contain `expand` constructs, which are used to invoke expansion plugins. Maestro2 will then invoke the expansion plugin and replace the `expand` construct with the MaBL generated by the corresponding expansion plugin. MaBL expansion plugins serve two purposes: 1. to generate MaBL based on a given set of parameters to reduce the amount of manual work required to create a MaBL specification, and 2. to extend the Maestro2 framework with new functionality such as fault injection and design space exploration Ejersbo et al. (2023); Pierce et al. (2022); Frasheri et al. (2021).

An example of the application of expansion plugins is the `Initializer` plugin (Hansen et al. (2021c)) shown in Listing 2. The `Initializer` defines a function called `initialize` which is called in line 3 using the `expand`. The plugin generates the necessary FMI calls to initialise the FMUs in the simulation in MaBL, based on the connections and dependencies between the FMUs. Listing 2 also shows the `@Config` annotation, which is used to provide additional information to the expansion plugin, which it can use to generate the MaBL. The configuration in Listing 2 specifies the values of the parameters `maxLevel` and `minLevel` of the `Controller` FMU.

Listing 2: MaBL Expansion

```
1 @Config("{\"parameters\":{\"{crtl}.crtlInstance
    ↪ .maxlevel\":1,\"{crtl}.crtlInstance.
    ↪ minlevel\":1}}");
2 Initializer.expand initialize(components,
    ↪ START_TIME, END_TIME);
```

Expansion plugins can be chained together, so the MaBL generated by the `Initializer` plugin can contain additional `expand` constructs, which are then expanded by other plugins. This feature, while powerful, should be used with care to avoid potential infinite loops. However, when used properly, it allows complex scenarios to be created with minimal effort, and allows expansion plugin authors to leverage existing functionality. To avoid invalid MaBL specifications, the Maestro2 framework performs a type check on the resulting MaBL specification after each extension plugin is applied. However, it is important to note that the expansion plugins are not limited to generating MaBL, but can also perform other tasks such as verifying the modelDescription of an FMU, as done by the `ModelDescriptionVerifier` plugin. Once a fully expanded MaBL specification without `expand` constructs has been generated, it can be fine-tuned manually if necessary before execution.

Commonly Used Expansion Plugins. Maestro2 is supplied with a number of commonly used extension plugins to minimise the amount of manual work involved in the creation of a MaBL specification.

`Initializer` (Hansen et al. (2021c)) generates the initialisation code for a co-simulation scenario in MaBL. The plugin leverages state of the art algorithms for synthesizing the initialisation algorithm of a co-simulation scenario potentially containing algebraic loops (Hansen et al. (2021c); Gomes et al. (2019a); Hansen et al. (2022b)). Concretely, the plugin builds a dependency graph of the FMUs in the scenario and employs graph-based reasoning to determine the order in which the FMUs should be initialised based on the interconnections between the FMUs and their contracts. The plugin contains an optional feature of verifying the calculated initialisation order against the verifier implemented in Gomes et al. (2019a).

`JacobianStepBuilder` produces the MaBL necessary to perform *Step* stage of the OA based on the Jacobian iteration method. Jacobian iteration performs a simulation step by prompting all FMUs to progress in time, retrieves the necessary outputs, and sets the necessary inputs (Gomes et al. (2018d)). The plugin can, similarly to Maestro1, be tailored to use different methods for dynamically determining the step size when performing a simulation with a variable step size (Thule et al. (2019)). The current implementation supports the following methods:

Zero Crossing: Synchronise the FMUs at a point in time where a given signal is zero or two signals intersect.

Bounded Difference: The bounded difference constraint bounds the difference between two signals or two consecutive observations of the same signal by a pre-defined value.

Sampling Rate: Ensures that all FMUs synchronise at pre-determined points in time.

FMU Max Step Size: First proposed in Broman et al. (2013a) this constraint attempts to avoid the need for rollbacks. It requires the FMUs to implement a non-FMI function `getMaxStepSize` that returns the maximum step that the given FMU can perform at the given point in time.

The main difference between this plugin and its counterpart in Maestro1 is that the functionality is now realised via MaBL and associated runtime modules, while it was previously implemented directly in Scala.

Stabilisation is another commonly used feature to ensure that the co-simulation converges to a steady state. This is accomplished by performing multiple iterations of a given step until convergence is achieved or the maximum attempts are reached. Concretely, the plugin performs a simulation step, checks if the simulation has converged, and if not, it rolls back the FMUs to their previous state and performs another simulation step with the output values from previous iteration.

`ModelDescriptionVerifier` is a verification plugin. A verification plugin does not generate MaBL. Instead, it has access to the AST of the MaBL specification and can perform various checks on the specification. The `ModelDescriptionVerifier` plugin verifies the ModelDescription files of the FMUs against a formal model of the FMI standard implemented in VDM-SL via the tool VDMCheck (Battle et al. (2020)). Moreover, the plugin verifies that all FMUs are properly unloaded after the simulation has finished by analysing the MaBL AST.

More information about the expansion plugins available in Maestro2 can be found in the online documentation (Association (d)). The online documentation also contains a guide for creating new expansion plugins to enable expert users to extend the Maestro2 framework with new functionality.

3.3.2. MaBL API

One approach to generating a MaBL specification is to use the MaBL API, a Java API that can be used to generate a MaBL specification programmatically. This approach is typically used by the expansion plugins (e.g. Initialization and JacobianStepBuilder) to generate MaBL, but users can also use it to generate a MaBL specification. Specifically, the MaBL API hides the complexity of the underlying MaBL syntax and allows the user to generate a MaBL specification using a high-level API, without having to worry about error handling, AST generation, and other complexities of the MaBL language.

An example of the MaBL API is shown in Listing 3, which is used by the `Initialization` plugin to change the mode of an FMU to `Initialisation Mode`. Although the example is simple, it still empowers the user as they do not have to worry about the complexities of handling all the possible return values of the FMI function `enterInitializationMode`. Instead, the user can concentrate on the task at hand, which is to change the mode of an FMU to `Initialization Mode`.

Listing 3: FMU instance function call using MaBL API

```
fmuInstanceVariable . enterInitializationMode ()
```

Another valuable feature of the MaBL API is the ability to programmatically link one FMU's output to another FMU's input. The MaBL API will, based on these couplings, generate the necessary MaBL code to exchange the data between the linked ports.

The MaBL API provides complete flexibility to the user, allowing them to create a MaBL specification tailored to their needs, such as the adaptive co-simulation scenario described in Section 4. However, this expressiveness comes at the cost of increased complexity, as the user has to deal with all the intricacies of the simulation.

3.3.3. Scenario-Verifier

The third approach to generating a MaBL specification is to use the Scenario-Verifier (Hansen et al. (2021b,a, 2022b)), which is a tool for synthesising and verifying OAs for co-simulation scenarios described in a high-level DSL similar to Definition 2.

The Scenario-Verifier uses the latest advances in OA synthesis (Hansen et al. (2021c); Gomes et al. (2019a); Hansen et al. (2022b)) and constraints declared by the *R* function to synthesise an OA tailored to a given co-simulation scenario, subject to both step rejections and algebraic loops Kübler and Schiehlen (2000b), while ensuring that the OA respects the implementation details of the scenario and correctly implements the FMI standard. The Scenario-Verifier synthesises an OA that employs a stabilisation algorithm to handle algebraic loops and a step size adjustment algorithm to handle step rejections. As a result, a co-simulation practitioner does not need to worry about the intricacies of the OA of such complex scenarios, as the Scenario-Verifier will synthesise an OA that ensures a co-simulation where all FMUs move in lockstep and algebraic loops are stabilised.

Furthermore, the tool can verify a given OA against the FMI standard and the scenario description. Concretely, the tool uses a symbolic formalisation of the FMI standard and the OA in the model checker UPPAAL (Behrmann et al. (2006)) to verify that the OA respects the implementation details of the scenario and correctly implements the FMI standard. For example, the tool verifies that the OA solves the FMUs in an optimal order that respects the implementations of the FMUs, that all FMUs move in lockstep, and that algebraic loops are stabilised. Errors in the OA are reported to the user in the form of a trace, which can be visually inspected to debug the OA.

However, it is essential to note that the Scenario-Verifier tool only synthesises the *Initialise* and *Step* stages of the OA in DSL format. Consequently, it is used in conjunction with the MaBL API and expansion plugins to generate the remaining stages of the OA and translate the DSL into MaBL.

This approach is recommended for users with little experience with MaBL and co-simulation in general, as it requires almost no effort to start a co-simulation. Nevertheless, the Scenario-Verifier also provides a number of advanced features for expert users to fine-tune the OA by providing additional constraints and expectations to the tool.

3.3.4. Configuration

The last approach to generating a MaBL specification is to use a configuration file called the multi-model (Larsen et al. (2016)). The multi-model is a JSON file that details the FMUs involved in the co-simulation, the connections between them, and the parameters of the FMUs. The configuration file is then used to generate a MaBL specification using the MaBL API, using both the `Initializer` and `JacobianStepBuilder` expansion plugins. Nevertheless, the configuration file can also be used to generate a MaBL specification using the Scenario-Verifier tool, which requires some minor modifications to the multi-model by Maestro2 to make it compatible with the Scenario-Verifier tool.

This approach is recommended for users with little experience with MaBL and co-simulation in general, as it requires the least effort to start a co-simulation. The configuration file can be generated by other tools, such as the INTO-CPS Application (Macedo et al. (2020)).

3.4. Execution

The final step in the Maestro2 approach is to execute the MaBL specification generated in the previous step. Maestro2 provides two execution modes: *interpretation* and *code generation* for executing a MaBL specification.

The interpretation mode is based on an interpreter written in Java, which is the default execution mode of Maestro2 as it is the most convenient for development and debugging purposes. Nevertheless, in order to minimise the overhead of the co-simulation framework, a code generator that translates a MaBL specification into C++ code has been implemented as well. The code generator is based on the Java interpreter, and thus the generated code is functionally equivalent to the interpreted code. The code generator offers a significant performance improvement over the Java interpreter as shown in Figure 8. The comparison in Figure 8 is based on 100 co-simulations of the water tank scenario described in Figure 3. Each simulation has a different end time, starting with an end time of 1s for simulation 1 and increasing by one for each simulation so that the last simulation (number 100) has an end time of 100s. All these 100 co-simulations were run on Maestro1 (the predecessor of Maestro2, see Section 5), Maestro2 with the Java interpreter, and the code-generated C++ version of Maestro2.

The execution time of the various co-simulations is shown in Figure 8. The figure shows two plots, one including the time taken to load FMUs and one without the time taken to load the FMUs, telling the same story. The time taken to generate the specification and compile the C++ code is not included in the figure, as it is a one-time cost.

The figure shows that the C++ code generated by Maestro2 is significantly faster than the interpreter (`maestro2`) and Maestro1. The optimisation of the generated code is amplified as the simulation time increases, as the simulation loop is the primary time expenditure as the simulation time grows. As the figure shows, the interpreter (`maestro2`) is faster than Maestro1. This is because Maestro1 performs various lookups during runtime, whereas Maestro2 performs these lookups during compilation.

However, the performance gain of the code generated does come at a cost in terms of expressivity, as it requires used runtime modules to be ported to C++, as only the MaBL specification is translated to C++. Nevertheless, some runtime modules have already been ported to C++, such as the FMI2 runtime module, the DataWriter module, and the MEnv module, to test and validate the approach.

Although the execution times in Figure 8 are small, and you might think that the performance gains it not worth the effort, it is essential to note that the performance gain is amplified when performing DSE. A DSE study consists of a series of co-simulations with different parameters to find the optimal solution. Thus the performance gain is amplified as the number of co-simulations increases.

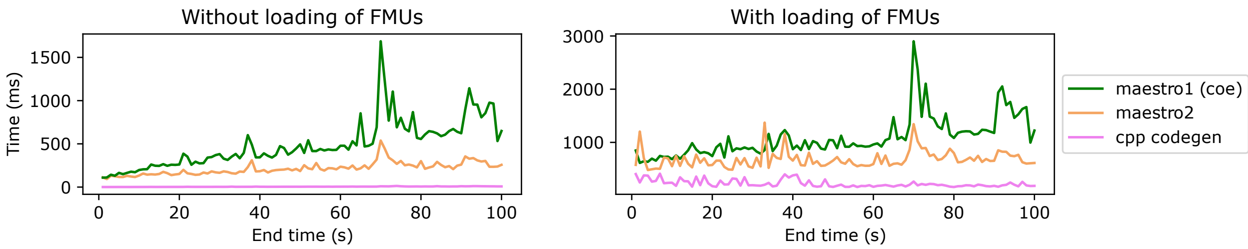

Figure 8: Performance comparison of Maestro, Maestro2 Interpreter and Maestro2 Code Generated version. In the left figure the time for loading the FMUs is subtracted from the total simulation time. In the right figure it is included.

3.5. Utilizing Maestro2

Maestro2 offers two interfaces for interacting with the framework: a Command Line Interface (CLI) and a REST interface. Both interfaces offer similar functionality and strike a balance between new functionality and compatibility with Maestro1 to ensure a smooth transition.

The CLI offers a minimalistic interface for interacting with Maestro2, which is useful for scripting and automation as it does not require running a web server. The CLI is used by the INTO-CPS DSE functionality (Bogomolov et al. (2020)).

The REST interface allows Maestro2 to function as a web server, enabling cloud support and remote access. It offers the flexibility of accessing Maestro2's functionality through HTTP requests. The REST interface is used by applications like the INTO-CPS Application Macedo et al. (2020). Additionally, the REST interface supports live-streaming of data via web sockets, enabling real-time data updates as the simulation progresses. This feature involves adding an extra listener to the DataWriter runtime module, as briefly demonstrated in Listing 1. It showcases the possibilities of combining MaBL with runtime modules.

The interfaces are not covered in detail here, but the interested reader can refer to the online documentation (Association (d)) for more information on their usage and capabilities.

4. Case Studies

This section presents two case studies that illustrate how Maestro2 can be used to tackle a broad variety of co-simulation scenarios. This section provides a brief overview of the case studies, the essential challenges that cannot be solved by a standard co-simulation framework, and the role of Maestro2 in tackling these challenges. More details about the case studies can be found in the corresponding references.

4.1. Adaptive Mass-Spring-Damper Co-simulation

The adaptive mass-spring-damper case study, detailed in Inci et al. (2021) and illustrated in Figure 9, consists of two linear mass-spring-damper subsystems, connected to rigid walls and coupled with a spring-damper. The system is simulated with two FMUs (`MSD1` and `MSD2`) as shown in Figure 11, one for each mass-spring-damper subsystem. Subsystem 1, `MSD1`, acts as an inert system by inputting the coupling force from Subsystem 2, `MSD2`, and outputting displacement and velocity to Subsystem 2.

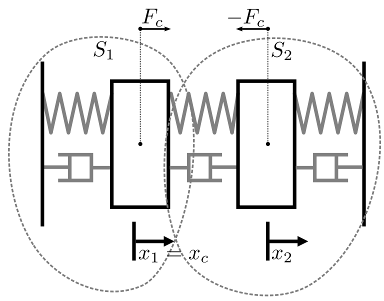

Figure 9: Double mass-spring-damper.

Although the system may initially appear simple, it poses challenges when accuracy is critical, as demonstrated in Inci et al. (2021). Their work demonstrates that achieving accurate results for this system requires the use of an adaptive co-simulation algorithm, which dynamically changes the order of actions in the algorithm employed in the `Step` stage of the co-simulation.

Changing the order of actions affects the sequence in which FMUs are simulated and how they communicate with each other. Specifically, the system depicted in Figure 9 can, according to the FMI standard, be simulated using the two algorithms described in Figure 10. Algorithm 2 simulates MSD1 first, followed by MSD2, while Algorithm 3 simulates MSD2 first, followed by MSD1. The two algorithms can, in fact, be synthesised by Maestro2 using the Scenario-Verifier tool.

In Inci et al. (2021) it is suggested that instead of choosing one of the algorithms a priori, it is better to choose the algorithm that minimises the error at each co-simulation step (i.e., adaptive co-simulation). The error is estimated by comparing the results of the two algorithms at each co-simulation step against a reference solution obtained by solving the system of equations analytically. To estimate the error and determine the algorithm, an additional FMU called `SwitchingDecision` is employed as shown in Figure 11.

The adaptive co-simulation algorithm uses the judgement on the output `best_order` of the `SwitchingDecision` FMU to decide which algorithm to use at each co-simulation step. Concretely, the adaptive co-simulation algorithm starts by employing Algorithm 2 to simulate the system for a single co-simulation step and then uses the `best_order` output of the `SwitchingDecision` FMU to decide which algorithm to use at the next co-simulation step to minimise the co-simulation error.

Role of Maestro2. Maestro2 provides the necessary functionality to implement adaptive co-simulation. Concretely, MaBL allows the user to implement the adaptive co-simulation algorithm in Figure 11 by using a conditional statement to decide between Algorithm 2 and Algorithm 3, in Figure 10 based on the `best_order` output of the `SwitchingDecision` FMU. Furthermore, MaBL support for declaring new variables to store variables between co-simulation steps, facilitating the implementation of the adaptive co-simulation algorithm. Finally, Maestro2's performance made the running time difference between the adaptive and static algorithms negligible.

The errors of the adaptive co-simulation algorithm are compared with those of the static algorithm in Figure 12. As can be seen, the adaptive algorithm attempts to follow the best sequence at any given time, thus freeing the user from determining which algorithm to employ at a specific time.

4.2. Hardware-in-the-loop Co-Simulation

The second case study showcases the use of Maestro2 in a hardware-in-the-loop co-simulation scenario, where the numerical simulation of a system is coupled with a physical system. Hardware-in-the-loop co-simulation is a common practice in seismic testing of civil engineering structures (McCrum and Williams (2016)).

The reported case study, detailed in Gomes et al. (2021a), consists of a physical cantilever beam coupled to a linear spring, as illustrated in Figure 13. The cantilever beam is excited by a sinusoidal loading applied via an electric linear actuator to study the dynamic response in terms of displacement (u) of the beam to seismic excitations provided by the linear spring. Figure 13 shows a picture of the experimental setup, along with a simplified version of the system, as depicted in the left part of the figure. Finally, the right part of the figure provides a schematic overview of the experimental setup.

Hardware-in-the-loop co-simulation is enabled by using a hybrid testing setup depicted in Figure 14. The hybrid testing (HT) setup comprises a three subsystems, modelled as distinct FMUs, as shown in Figure 14. The HT setup consists of a physical substructure (PS), and a numerical substructure (NS) simulated by a finite element (FE) software, and an FMU that couples the two substructures to enforce compatibility between the physical and numerical substructures (Coupling). The PS structure is equipped with several sensors and actuators, which are connected to a data acquisition system on an industrial PC via EtherCAT. The sensors and actuators are used to measure the response of the PS and to provide the excitation to the NS.

Role of Maestro2. There are two challenges in this case:

1. Mistakes in the co-simulation could lead to physical consequences on the connected hardware. Here Maestro2 support for static analysis plays a crucial role to prevent these.

2. One of the main challenges in this case study is the need to synchronise the numerical and physical substructures. This is achieved by using a custom orchestration algorithm implemented in MaBL. Concretely, the orchestration algorithm ensures that the Coupling FMU is simulated after the other two FMUs.

A video of the experiment is available online, see Association (2021).

5. Related Work

Co-simulation is a large field and challenging to cover thoroughly, with co-simulation frameworks being a moving target. For this reason, we introduce some of the existing co-simulation frameworks and compare them to our contribution in Table 1, on the item where Maestro2 is most novel: its capability for extensive customization while maintaining robust verification capabilities. For surveys on the co-simulation topic, we refer

Algorithm 2 MSD1 → MSD2

doStep(S_1,H)
$x_1' \leftarrow$ getOut(S_1,x_1)
$v_1' \leftarrow$ getOut(S_1,v_1)
setIn(S_2,x_1,x_1')
setIn(S_2,v_1,v_1')

doStep(S_2,H)
$F_k' \leftarrow$ getOut(S_2,F_k)
setIn(S_1,F_k,F_k')

Algorithm 3 MSD2 → MSD1

doStep(S_2,H)
$F_k' \leftarrow$ getOut(S_2,F_k)
setIn(S_1,F_k,F_k')

doStep(S_1,H)
$x_1' \leftarrow$ getOut(S_1,x_1)
$v_1' \leftarrow$ getOut(S_1,v_1)
setIn(S_2,x_1,x_1')
setIn(S_2,v_1,v_1')

At time t:	
doStep(S_1,H):	Advances the state of FMU S_1 by H
getOut(S_1,y):	Returns the output y of the FMU S_1
setIn(S_1,u,y):	Assigns the input u of the FMU S_1 to the value y

Figure 10: Possible algorithms. Adapted from Inci et al. (2021). Both algorithms are valid according to the FMI standard and can be used to simulate the system depicted in Figure 9.

Table 1: Overview of co-simulation frameworks and their customization options.

Tool	FMI	Compiled	Interpreted	License	Customizations
DACCOSIM NG Evora Gomez et al. (2019)	Yes	No	Yes	Open Source	Step size
Dymola Brück et al. (2002)	Yes	Yes	No	Proprietary	Step size
FIDE Cremona et al. (2016)	No[2]	Yes	No	Proprietary	Step size
VICO Hatledal et al. (2021)	Yes	No	Yes	Open Source	Step size, Runtime behavior through coded extensions.
C2WT Neema et al. (2014)	No[3]	No	Yes	Proprietary	Step size
FMPy	Yes	No	Yes	Open-Source	Full customization by coding interaction with FMUs in Python.
OMSimulator Ochel et al. (2019)	Yes	No	Yes	Open-Source	Step size, Algebraic loop solver.
Van Acker et al. (2015)	Yes	Yes	No	Open-Source	Step-size.
Maestro1	Yes	No	Yes	Open-Source	Step-size, Algebraic loop solver.
Maestro2	Yes	Yes	Yes	Open-Source	Full customization using domain specific language.

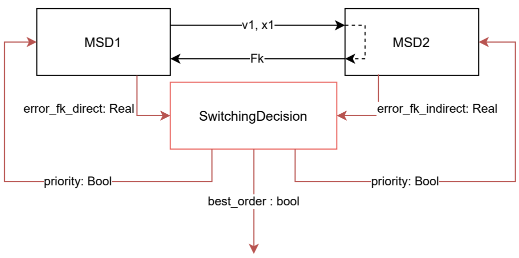

Figure 11: Co-simulation scenario. Adapted from Inci et al. (2021).

the reader to Gomes et al. (2018b); Hafner and Popper (2017); Palensky et al. (2017) for related surveys.

Maestro1 Thule et al. (2019), the predecessor of Maestro2, was developed during the INTO-CPS project (Association (c); Larsen et al. (2016)) and is an FMI-based co-simulation orchestration engine. It can perform co-simulations with a fixed algorithm and lacks the customisation and verification abilities of Maestro2. DACCOSIM Galtier et al. (2015); Evora Gomez et al. (2019) has been rebuilt as DACCOSIM NG, and extended with additional features, such as the capability of packaging multiple FMUs, including a scenario into a single FMU, referred to as Matryoshka FMU (Evora Gomez et al. (2019)). Another interesting FMI-based co-simulation framework is VICO (Hatledal et al. (2021)). VICO runs on the Java Virtual Machine and is thereby cross-platform. It supports the FMI companion standard System Structure and Parameterisation (Jochen Köhler et al. (2016); Association (e)) for defining the structure of a co-simulation. It strongly focuses on the possibility of composition through a clear separation of objects composed solely of data and systems that act on such objects. This is referred to as Entity-Component-System (Martin (2007)). The promise of this approach is supporting runtime behaviour change and simplicity of use. Furthermore, it features built-in 3D graphics and plotting capabilities. While this shares the goals of Maestro2, the approach is different. One could, for

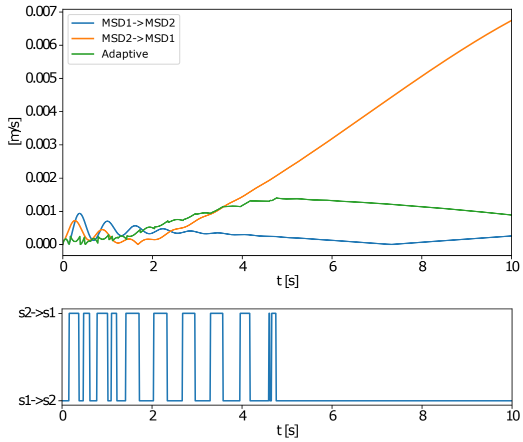

Figure 12: On the top, error in v_1 for adaptive and static co-simulation sequences. At the bottom, sequence changes of the adaptive co-simulation (corresponds to the `best_order` output in Figure 11). Reproduced from Inci et al. (2021).

example, consider composing a co-simulation through VICO and then use the MaBL API to generate the execution of the composed co-simulation. The compositional features could also be implemented as a runtime plugin. Van Acker et al. (2015) presents a modelling language to model a co-simulation setup and a related transformation to an optimised master algorithm ready for execution. Thus, it is similar in nature to some of the concepts of Maestro2 and the Scenario Verifier. One difference is, for example, that it does not contain an interpreter or a plugin structure. The AVL Model.CONNECT co-simulation tool is a professional tool focusing primarily on the automotive market. It does support FMI, and its strength seems to be its ability to carry out Hardware-In-the-Loop simulations. We have not been able to find indications that it supports the level of customizability we demonstrate with Maestro2. For more related FMI-based tools, the reader is referred to the tools page on the FMI-website (Committee), where several FMI-enabled importing tools are listed.

Crucially, Maestro2 also targets verification efforts, ideally both at the FMU and orchestration level, and as such, contributions within this area are of interest as well. The tool VDMCheck (Battle et al. (2020)), is an example of FMU verification. It verifies the ModelDescription file of an FMU, and has explicit support within Maestro2. This can be applied directly, as it does not require user interaction. Gomes et al. (2018a) considers adapting simulators to correct interaction assumptions based on a different environment. Their approach is to create a new FMU, referred to as *external FMU* that encloses one or more FMUs, referred to as *internal FMUs*. Via a DSL called *baseSA* it is possible to create rules for mapping actions applied to the external FMU to actions applied to the internal FMUs and interaction between the internal FMUs. This is applied through a sound definition of hierarchical simulators that leaves the internal FMUs unmodified and thus improves modularity and preserves transparency. MaBL is capable of representing a semantically equivalent set of operations, and as such, MaBL and Maestro could function as an execution engine for baseSA, if one was inclined to write such an expansion plugin. However, it does not feature the FMU-generation capabilities that generalise the approach in Gomes et al. (2018a) to all FMI-based orchestration engines. The Scenario Verifier (Hansen et al. (2021b)) described in Section 3.3.3 is an example of a tool that calculates and verifies an FMI-based co-simulation OA based on constraints and expectations of the enclosed FMUs. Enriching the environment of such an OA has been proven possible, see Section 3.3.3, to create an executable co-simulation in MaBL. The last example of tooling to be considered in this publication related to verifying the behaviour of a co-simulation and its constituents is within the domain of Test Automation in Ouy et al. (2017). Here, a Test FMU is created in order to stimulate the system under test according to system requirements, whereas other FMUs represent the system under test. An external tool then evaluates the outputs of the test FMU in order to determine whether the system under test expressed correct behaviour.

6. Concluding Remarks

This paper introduced Maestro2, a co-simulation framework designed for running FMI-based co-simulations. The key contribution of this work is the Maestro2 approach, which leverages the MaBL DSL to empower users in customizing the OA and minimizing co-simulation errors.

Maestro2 offers different levels of automation for describing the co-simulation scenario and the OA, catering to both prototyping needs and expert-level fine-tuning. The framework utilizes code generation techniques to ensure minimal overhead and high performance for co-simulation practitioners.

With its plugin architecture, Maestro2 enables users to extend the framework with new capabilities, such as incorporating new FMU types, supporting additional logging formats, and integrating new verification tools.

The framework's effectiveness has been demonstrated through multiple case studies and research

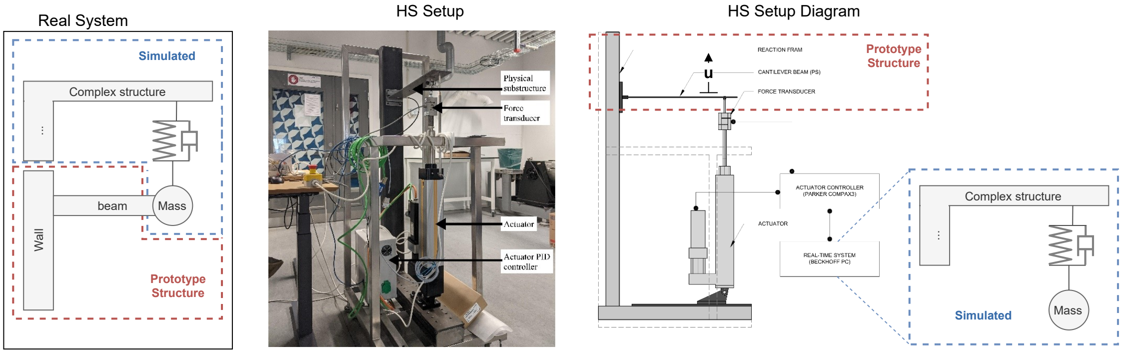

Figure 13: Experimental setup installed at the Dynamisk LAB of Aarhus University. Adapted from Gomes et al. (2021a).

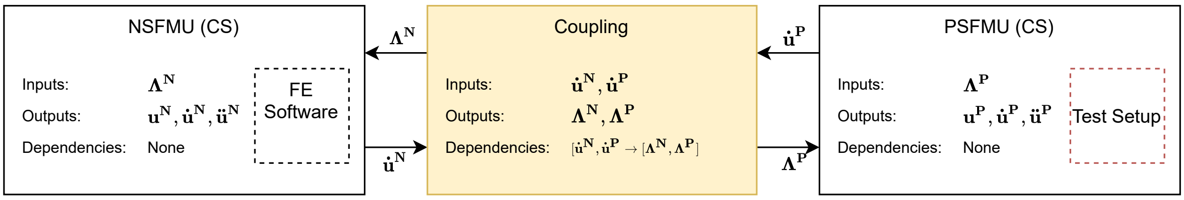

Figure 14: Co-simulation scenarios that implements the setup described in Figure 13. The dashed boxes represent the fact that NSFMU (respectively PSFMU) communicate with a FE Software (resp. the Test Setup), when the `fmi2DoStep` function is invoked.

projects (see Section 4), showcasing its capabilities in tackling real-world problems.

To our knowledge, Maestro2 is the only open-source co-simulation framework that offers a balance between flexibility, usability, and performance. However, we acknowledge the presence of open challenges highlighted by the case studies, including the need for a more user-friendly interface for the MaBL DSL and further enhancements to the verification capabilities of the framework. Addressing these challenges will be crucial for advancing the framework and improving its usability in practical applications.

Future work will focus on these challenges and explore promising directions to enhance Maestro2.

Future Work. Maestro2 provides, due to its plugin-based architecture, a solid foundation for future work. Some of the most promising directions for future work are: Supporting the newest version of the FMI standard (Junghanns et al. (2021); Hansen et al. (2022a)) to enable co-simulation of a broader range of systems, such as hybrid and reactive systems. This work has already been initiated, and MaBL is currently being extended to support the new features of the FMI standard. The work is expected to be completed during 2023, making Maestro2 one of the first co-simulation frameworks to support the new version of the FMI standard.

Furthermore, we plan to extend Maestro2 to be applicable in the context of digital twin engineering, more specifically, the incubator project (Feng et al. (2021a,b)), some initial results of which are presented in (Association (b,a)). We expect this will lead to new interfaces and functionality, such as the possibility of changing the simulation algorithm or replacing FMUs at runtime due to external changes while still preserving the benefits of calculating a specification before execution.

Finally, we plan to extend the verification capabilities of Maestro2 to permit verification of the final MaBL specification using the Scenario Verifier (Hansen et al. (2022b)). This is an appealing challenge, as it becomes available to all other tools using MaBL/Maestro2 as their execution target. Possible verification efforts include verifying that the MaBL specification is well-formed, deterministic, and adheres to the FMI standard. Nevertheless, the verification efforts must be carefully considered, as the verification capabilities are potentially at odds with the customizability of MaBL empowering experts to fine-tune the OA to minimise co-simulation error.

Acknowledgments

This research was funded by a number of externally funded research projects including DiT4CPS, UPSIM,

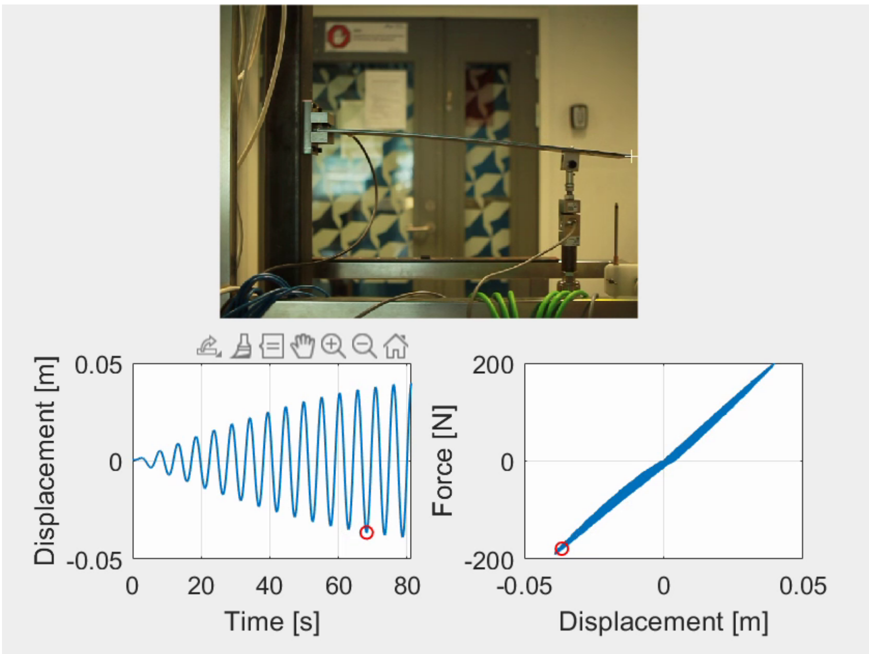

Figure 15: Numerical results as reported in Gomes et al. (2021a). The full video can be seen online Association (2021).

HUBCAP and AgroRobottiFleet. Furthermore, we are grateful to the Poul Due Jensen Foundation, which has supported the establishment of the Center for Digital Twin Technology at Aarhus University.

References

Andersson, C., 2016. Methods and Tools for Co-Simulation of Dynamic Systems with the Functional Mock-up Interface. Ph.D. thesis. Lund University.

Arnold, M., 2010. Stability of Sequential Modular Time Integration Methods for Coupled Multibody System Models. Journal of Computational and Nonlinear Dynamics 5, 9. doi:10.1115/1.4001389.

Arnold, M., Clauß, C., Schierz, T., 2014. Error Analysis and Error Estimates for Co-simulation in FMI for Model Exchange and Co-Simulation v2.0, in: Progress in Differential-Algebraic Equations, Springer Berlin Heidelberg, Berlin, Heidelberg. pp. 107–125. doi:10.1007/978-3-662-44926-4_6.

Association, I.C., a. Digital twin tutorial. URL: https://sites.google.com/view/fm2021tutorialdt/home. visited July 11th, 2023.

Association, I.C., b. Fm workshops and tutorials. URL: http://lcs.ios.ac.cn/fm2021/workshops-and-tutorials/. visited July 11th, 2023.

Association, I.C., c. Integrated tool chain for model-based design of cpss. URL: https://cordis.europa.eu/project/id/644047. visited July 11th, 2023.

Association, I.C., d. Maestro2 documentation. URL: https://into-cps-maestro.readthedocs.io/en/latest/user/index.html. visited July 11th, 2023.

Association, I.C., 2021. Hybrid testing experiment video. URL: https://youtu.be/-VkrQJaUo1o. visited July 11th, 2023.

Association, M., e. SSP standard website. URL: https://ssp-standard.org/. visited July 11th, 2023.

Battle, N., Thule, C., Gomes, C., Macedo, H.D., Larsen, P.G., 2020. Towards a Static Check of FMUs in VDM-SL, in: FM 2019 International Workshops, Springer International Publishing, Porto, Portugal. pp. 272–288. doi:10.1007/978-3-030-54997-8_18.

Behrmann, G., David, A., Larsen, K.G., Håkansson, J., Pettersson, P., Yi, W., Hendriks, M., 2006. UPPAAL 4.0, in: QEST 2006, IEEE Computer Society. pp. 125–126. doi:10.1109/QEST.2006.59.

Blochwitz, T., Otter, M., Arnold, M., Bausch, C., Clauss, C., Elmqvist, H., Junghanns, A., Mauss, J., Monteiro, M., Neidhold, T., Neumerkel, D., Olsson, H., Peetz, J.V., Wolf, S., 2011. The Functional Mockup Interface for Tool independent Exchange of Simulation Models, in: Proc. of the 8th International Modelica Conference, Linköping University Electronic Press; Linköpings universitet, Dresden, Germany. pp. 105–114. doi:10.3384/ecp11063105.

Bogomolov, S., Fitzgerald, J., Foldager, F., Larsen, P.G., Pierce, K., Stankaitis, P., Wooding, B., 2020. Tuning Robotti: the Machine-assisted Exploration of Parameter Spaces in Multi-Models of a Cyber-Physical System, in: Fitzgerald, J.S., Oda, T. (Eds.), Proc. of the 18th International Overture Workshop, Overture. pp. 50–64.

Broman, D., Brooks, C., Greenberg, L., Lee, E.A., Masin, M., Tripakis, S., Wetter, M., 2013a. Determinate composition of FMUs for co-simulation, in: 2013 Proc. (EMSOFT), IEEE. pp. 1–12. doi:10.1109/EMSOFT.2013.6658580.

Broman, D., Derler, P., Eidson, J.C., 2013b. Temporal Issues in Cyber-Physical Systems. J. Indian Inst. Sci. 93, 389–402.

Brück, D., Elmqvist, H., Mattsson, S.E., Olsson, H., 2002. Dymola for multi-engineering modeling and simulation, in: Proc. of modelica, Citeseer.

Busch, M., 2016. Continuous approximation techniques for co-simulation methods: Analysis of numerical stability and local error. Journal of Applied Mathematics and Mechanics 96, 1061–1081. doi:10.1002/zamm.201500196.

CES, A., Widl, E., Strasser, T.I., 2021. Erigrid/ns3-fmi-export: v1.1. URL: https://doi.org/10.5281/zenodo.4638103, doi:10.5281/zenodo.4638103. the time progression of 0 is mentioned in the readme.md file.

Committee, F.S., . Fmi website. URL: https://fmi-standard.org/tools/. visited July 11th, 2023.

Committee, F.S., 2014. Functional Mock-up Interface for Model Exchange and Co-Simulation. https://fmi-standard.org/downloads/.

Committee, F.S., 2021. Functional Mock-up Interface for Model Exchange, Co-Simulation, and Scheduled Execution. https://fmi-standard.org/downloads/.

Cremona, F., Lohstroh, M., Broman, D., Lee, E.A., Masin, M., Tripakis, S., 2017. Hybrid co-simulation: it's about time. Software & Systems Modeling doi:10.1007/s10270-017-0633-6.

Cremona, F., Lohstroh, M., Tripakis, S., Brooks, C., Lee, E.A., 2016. FIDE, in: Proc. of the 31st Annual ACM Symposium on Applied Computing, ACM. doi:10.1145/2851613.2851677.

Ejersbo, H., Lausdahl, K., Frasheri, M., Esterle, L., 2023. fmiSwap: Run-time Swapping of Models for Co-simulation and Digital Twins. arXiv preprint arXiv:2304.07328 .

Evora Gomez, J., Cabrera, J.J.H., Tavella, J.P., Vialle, S., Kremers, E., Frayssinet, L., 2019. Daccosim NG: co-simulation made simpler and faster, in: Linköping electronic conference proceedings, pp. 785–792. doi:10.3384/ecp19157785.

Feng, H., Gomes, C., Thule, C., Lausdahl, K., Iosifidis, A., Larsen, P.G., 2021a. Introduction to Digital Twin Engineering, in: Martin, C.R., Blas, M.J., Psijas, A.I. (Eds.), 2021 ANNSIM, Virginia, USA. pp. 19–22.

Feng, H., Gomes, C., Thule, C., Lausdahl, K., Sandberg, M., Larsen, P.G., 2021b. The incubator case study for digital twin engineering. arXiv:2102.10390.

FMI, 2020. Functional Mock-up Interface for Model Exchange and Co-Simulation. Standard 2.0.2. URL: https://fmi-standard.org/downloads/.

Frasheri, M., Thule, C., Macedo, H.D., Lausdahl, K.G., Larsen, P.G., Esterle, L., 2021. Fault injecting co-simulations for safety, in: Proceedings of the Fifth International Joint Conference on System Reliability and Safety, 2021. ICSRS 2021, pp. 24–26. Accepted for publication in 5th International Conference on System Reliability and Safety.

Fritzson, P., 2015. Principles of Object-Oriented Modeling and Simulation with Modelica 3.3: A Cyber-Physical Approach. IEEE Press. 2 ed., Wiley. doi:10.1002/9781118989166.

Galtier, V., Vialle, S., Dad, C., Tavella, J.P., Lam-Yee-Mui, J.P., Plessis, G., 2015. FMI-Based Distributed Multi-Simulation with DACCOSIM, in: Spring Simulation Multi-Conference, Society for Computer Simulation International, Alexandria, Virginia, USA. pp. 804–811.

Glaessgen, E., Stargel, D., 2012. The Digital Twin Paradigm for Future NASA and U.S. Air Force Vehicles, in: Structures, Structural Dynamics, and Materials Conference: Special Session on the Digital Twin, American Institute of Aeronautics and Astronautics, Reston, Virigina. pp. 1–14. doi:10.2514/6.2012-1818.

Gomes, C., Abbiati, G., Larsen, P.G., 2021a. Seismic Hybrid Testing using FMI-based Co-Simulation, in: Proc. of the 14th International Modelica Conference, Linköping University Electronic Press, Linköpings Universitet, online. pp. 287–295.

Gomes, C., Lucio, L., Vangheluwe, H., 2019a. Semantics of Co-simulation Algorithms with Simulator Contracts, in: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), IEEE, Munich, Germany. pp. 784–789. doi:10.1109/MODELS-C.2019.00124.

Gomes, C., Meyers, B., Denil, J., Thule, C., Lausdahl, K., Vangheluwe, H., De Meulenaere, P., 2018a. Semantic Adaptation for FMI Co-simulation with Hierarchical Simulators. SIMULATION 95, 1–29. doi:10.1177/0037549718759775.

Gomes, C., Najafi, M., Sommer, T., Blesken, M., Zacharias, I., Kotte, O., Mai, P., Schuch, K., Wernersson, K., Bertsch, C., Blochwitz, T., Junghanns, A., 2021b. The FMI 3.0 Standard Interface for Clocked and Scheduled Simulations, in: Proc. of the 14th International Modelica Conference, Linköping University Electronic Press, Linköpings Universitet, online. pp. 27–36. doi:10.3384/ecp2118127.

Gomes, C., Oakes, B.J., Moradi, M., Gamiz, A.T., Mendo, J.C., Dutre, S., Denil, J., Vangheluwe, H., 2019b. HintCO - Hint-Based Configuration of Co-Simulations, in: International Conference on Simulation and Modeling Methodologies, Technologies and Applications, Prague, Czech Republic. pp. 57–68. doi:10.5220/0007830000570068.

Gomes, C., Thule, C., Broman, D., Larsen, P.G., Vangheluwe, H., 2018b. Co-simulation: A Survey. ACM Computing Surveys 51, 49:1–49:33. doi:10.1145/3179993.

Gomes, C., Thule, C., DeAntoni, J., Larsen, P.G., Vangheluwe, H., 2018c. Co-simulation: The Past, Future, and Open Challenges, in: ISOLA 2018, Springer Verlag, Limassol, Cyprus. pp. 504–520. doi:10.1007/978-3-030-03424-5_34.

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

Gomes, C., Thule, C., Lausdahl, K., Larsen, P.G., Vangheluwe, H., 2018e. Stabilization Technique in INTO-CPS, in: 2nd Workshop on Formal Co-Simulation of Cyber-Physical Systems, Springer, Cham, Toulouse, France. pp. 45–51. doi:10.1007/978-3-030-04771-9_4.

Hafner, I., Popper, N., 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, ACM, New York, NY, USA. pp. 67–76. URL: https://doi.org/10.1145/3158191.3158203, doi:10.1145/3158191.3158203.

Hansen, S.T., Gomes, C., Larsen, P.G., Van de Pol, J., 2021a. Synthesizing co-simulation algorithms with step negotiation and algebraic loop handling, in: Martin, C.R., Blas, M.J., Psijas, A.I. (Eds.), 2021 ANNSIM, pp. 1–12. doi:10.23919/ANNSIM52504.2021.9552073.

Hansen, S.T., Gomes, C., Palmieri, M., Thule, C., van de Pol, J., Woodcock, J., 2021b. Verification of co-simulation algorithms subject to algebraic loops and adaptive steps, in: Lluch Lafuente, A., Mavridou, A. (Eds.), FMICS 2021, Springer International Publishing, Cham. pp. 3–20.

Hansen, S.T., Gomes, C.G., Najafi, M., Sommer, T., Blesken, M., Zacharias, I., Kotte, O., Mai, P.R., Schuch, K., Wernersson, K., Bertsch, C., Blochwitz, T., Junghanns, A., 2022a. The FMI 3.0 Standard Interface for Clocked and Scheduled Simulations. Electronics 11, 3635.

Hansen, S.T., Ölveczky, P.C., 2022. Modeling, algorithm synthesis, and instrumentation for co-simulation in maude, in: Bae, K. (Ed.), Rewriting Logic and Its Applications, Springer International Publishing, Cham. pp. 130–150.

Hansen, S.T., Thule, C., Gomes, C., 2021c. An FMI-Based Initialization Plugin for INTO-CPS Maestro 2, in: Cleophas, L., Massink, M. (Eds.), SEFM 2020 Collocated Workshops, Springer International Publishing, Cham. pp. 295–310.

Hansen, S.T., Thule, C., Gomes, C., van de Pol, J., Palmieri, M., Inci, E.O., Madsen, F., Alfonso, J., Castellanos, J.Á., Rodriguez, J.M., 2022b. Verification and synthesis of co-simulation algorithms subject to algebraic loops and adaptive steps. STTT 24, 999–1024.

Hatledal, L.I., Chu, Y., Styve, A., Zhang, H., 2021. Vico: An entity-component-system based co-simulation framework. Simulation Modelling Practice and Theory 108, 102243. doi:https://doi.org/10.1016/j.simpat.2020.102243.

Inci, E.O., Gomes, C., Croes, J., Thule, C., Lausdahl, K., Desmet, W., Larsen, P.G., 2021. The effect and selection of solution sequence in co-simulation, in: Martin, C.R., Blas, M.J., Psijas, A.I. (Eds.), 2021 ANNSIM, Virginia, USA. pp. 1–12.

Jochen Köhler, Hans-Martin Heinkel, Pierre Mai, Jürgen Krasser, Markus Deppe, Mikio Nagasawa, 2016. Modelica-Association-Project "System Structure and Parameterization" – Early Insights, Tokyo, Japan. pp. 35–42. doi:http://dx.doi.org/10.3384/ecp1612435.

Junghanns, A., Blochwitz, T., Bertsch, C., Sommer, T., Wernersson, K., Pillekeit, A., Zacharias, I., Blesken, M., Mai, P., Schuch, K., Schulze, C., Gomes, C., Najafi, M., 2021. The Functional Mock-up Interface 3.0 - New Features Enabling New Applications, in: Proc. of the 14th International Modelica Conference, Linköping University Electronic Press, Linköpings Universitet, online.

Kalmar-Nagy, T., Stanciulescu, I., 2014. Can complex systems really be simulated? Applied Mathematics and Computation 227, 199–211. doi:10.1016/j.amc.2013.11.037.

Kübler, R., Schiehlen, W., 2000a. Modular Simulation in Multibody System Dynamics. Multibody System Dynamics 4, 107–127. doi:10.1023/A:1009810318420.

Kübler, R., Schiehlen, W., 2000b. Two Methods of Simulator Coupling. Mathematical and Computer Modelling of Dynamical Systems 6, 93–113. doi:10.1076/1387-3954(200006)6:2;1-M;FT093.

Larsen, P.G., Fitzgerald, J., Woodcock, J., Fritzson, P., Brauer, J., Kleijn, C., Lecomte, T., Pfeil, M., Green, O., Basagiannis, S., Sadovykh, A., 2016. Integrated tool chain for model-based design of Cyber-Physical Systems: The INTO-CPS project, in: 2nd International Workshop on Modelling, Analysis, and Control of Complex CPS (CPS Data), IEEE, Vienna, Austria. pp. 1–6. doi:10.1109/CPSData.2016.7496424.

Macedo, H.D., Rasmussen, M.B., Thule, C., Larsen, P.G., 2020. Migrating the INTO-CPS Application to the Cloud, in: Sekerinski, E., Moreira, N., Oliveira, J.N., Ratiu, D., Guidotti, R., Farrell, M., Luckcuck, M., Marmsoler, D., Campos, J., Astarte, T., Gonnord, L., Cerone, A., Couto, L., Dongol, B., Kutrib, M., Monteiro, P., Delmas, D. (Eds.), FM 2019 International Workshops, Springer International Publishing, Cham. pp. 254–271.

Mansfield, M., Gamble, C., Pierce, K., Fitzgerald, J., Foster, S., Thule, C., Nilsson, R., 2017. Examples Compendium 3. Technical Report. INTO-CPS Deliverable, D3.6.

Martin, A., 2007. Entity systems are the future of mmog development - part 1. URL: http://t-machine.org/index.php/2007/09/03/entity-systems-are-the-future-of-mmog-development-part-1/. (Accessed on 06/23/2021).

McCrum, D.P., Williams, M.S., 2016. An overview of seismic hybrid testing of engineering structures. Engineering Structures 118, 240–261. doi:10.1016/j.engstruct.2016.03.039.

Neema, H., Gohl, J., Lattmann, Z., Sztipanovits, J., Karsai, G., Neema, S., Bapty, T., Batteh, J., Tummescheit, H., Sureshkumar, C., 2014. Model-based integration platform for fmi co-simulation and heterogeneous simulations of cyber-physical systems, in: The 10th International Modelica Conference 2014, Modelica Association, Lund, Sweden.

Oakes, B.J., Gomes, C., Holzinger, F.R., Benedikt, M., Denil, J., Vangheluwe, H., 2021. Hint-Based Configuration of Co-simulations with Algebraic Loops, in: Simulation and Modeling Methodologies, Technologies and Applications. Springer International Publishing, Cham. volume 1260, pp. 1–28. doi:10.1007/978-3-030-55867-3_1.

Ochel, L., Braun, R., Thiele, B., Asghar, A., Buffoni, L., Eek, M., Fritzson, P., Fritzson, D., Horkeby, S., Hällquist, R., Kinnander, Å., Palanisamy, A., Pop, A., Sjölund, M., 2019. OMSimulator - integrated FMI and TLM-based co-simulation with composite model editing and SSP, in: Linköping Electronic Conference Proc., Linköing University Electronic Press. doi:10.3384/ecp1915769.

Ouy, J., Lecomte, T., Foldager, F.F., Henriksen, A.V., Green, O., Hallerstede, S., Larsen, P.G., Couto, L.D., Antonante, P., Basagiannis, S., Falleni, S., Ridouane, H., Saada, H., Zavaglio, E., König, C., Balcu, N., 2017. Case Studies 3, Public Version. Technical Report. INTO-CPS Public Deliverable, D1.3a. URL: https://into-cps.org/fileadmin/into-cps.org/Filer/D1.3a_Case_Studies.pdf.

Palensky, P., Van Der Meer, A.A., Lopez, C.D., Joseph, A., Pan, K., 2017. Cosimulation of intelligent power systems: Fundamentals, software architecture, numerics, and coupling. IEEE Industrial Electronics Magazine , 34–50doi:10.1109/MIE.2016.2639825.

Pierce, K., Lausdahl, K., Frasheri, M., 2022. Speeding up design space exploration through compiled master algorithms, in: Macedo, H., Pierce, K. (Eds.), Proc. of the 20th International Overture Workshop, pp. 66–81. doi:10.48550/arXiv.2208.10233. 20th Overture Workshop ; Conference date: 05-07-2022 Through 05-07-2022.

Schweizer, B., Li, P., Lu, D., 2015. Explicit and Implicit Cosimulation Methods: Stability and Convergence Analysis for Different Solver Coupling Approaches. Journal of Computational and Nonlinear Dynamics 10, 051007. doi:10.1115/1.4028503.

Schweizer, B., Lu, D., Li, P., 2016. Co-simulation method for solver coupling with algebraic constraints incorporating relaxation techniques. Multibody System Dynamics 36, 1–36. doi:10.1007/s11044-015-9464-9.

Thule, C., Lausdahl, K., Gomes, C., Meisl, G., Larsen, P.G., 2019. Maestro: The INTO-CPS co-simulation framework. Simulation Modelling Practice and Theory 92, 45–61. doi:10.1016/j.simpat.2018.12.005.

Thule, C., Palmieri, M., Gomes, C., Lausdahl, K., Macedo, H.D., Battle, N., Larsen, P.G., 2020. Towards Reuse of Synchronization Algorithms in Co-simulation Frameworks, in: Software Engineering and Formal Methods, Springer International Publishing, Oslo, Norway. pp. 50–66. doi:10.1007/978-3-030-57506-9_5.

Van Acker, B., Denil, J., Meulenaere, P.D., Vangheluwe, H., 2015. Generation of an Optimised Master Algorithm for FMI Co-simulation, in: Symposium on Theory of Modeling & Simulation-DEVS Integrative, Society for Computer Simulation International San Diego, CA, USA, Alexandria, Virginia, USA. pp. 946–953.