Verification of Co-Simulation Algorithms

Simon Thrane Hansen
DIGIT, Department of Engineering, Aarhus University,
sth@eng.au.dk

Cláudio Gomes
DIGIT, Department of Engineering, Aarhus University,
claudio.gomes@eng.au.dk

Casper Thule
DIGIT, Department of Engineering, Aarhus University,
casper.thule@eng.au.dk

Maurizio Palmieri
DII, Department of Information Engineering, University of Pisa,
maurizio.palmieri@ing.unipi.it

***Abstract*—Obtaining correct co-simulation results requires a correct co-simulation algorithm to mediate between the simulation units. Such an algorithm can be complex to implement due to constraints dictated by the simulation units' implementation, like how a simulation unit uses the value set on the input ports to calculate its future state.**

Previous work has looked into how explicit knowledge of the implementation of a simulation unit can be used to generate and define contracts of deterministic co-simulation algorithms of trivial co-simulation scenarios. This paper extends their work by extending it with the treatment of co-simulation algorithms for non-trivial scenarios containing both algebraic loops and step rejection. This work has resulted in a co-simulation *Verifier* that has been successfully applied to an industrial case study and other more complex scenarios. The results and implementation of the *Verifier* are available online.

***Index Terms*—Formal Semantics, Co-Simulation, Model-checking, Formal Methods**

I. Introduction

Claudio: This introduction might have to be adapted to give more motivation to co-simulation in case we go for a venue that is not familiar with the topic.

Cyber-physical systems (CPS) are omnipresent and embody physical processes being controlled by cyber elements. A CPS is typically developed in a distributed fashion, where the different constituents are developed by different teams using different tools and techniques relevant to the given domain of the constituent. Such systems are becoming increasingly complex [1], which leads to the desire of techniques to assist in the development of such systems. One such technique is co-simulation, a simulation technique used to manage the increasing complexity of the development of CPSs. Co-simulation is the study of how to coordinate multiple black-box simulation units (SUs), each responsible for computing the behavior of a subsystem, in order to compute their combined behavior, and therefore produce the global behavior of a system, as a discrete trace (see, e.g., [2], [3]). The coupling of the SUs, often developed independently from each other, is performed by a master algorithm that interacts with each SU via its interface. Such interface comprises functions for setting/getting values of the simulation and computing the associated model behavior over a given time interval. An example of such an SU is a Functional Mock-up Unit (FMU), which is a SU adhering to the Functional Mock-up Interface (FMI) standard version 2.0 for Co-Simulation [4], [5]. The definition of SU in this manuscript is closely related to, but not restricted by, the definition of an FMU. An SU can be restricted to act as an FMU, if certain functionality is avoided. Thus, the results of this manuscript are also relevant to FMI-based co-simulation. The distinction between FMU and SU is highlighted when relevant.

The main challenge in co-simulation is ensuring correct result, while developing co-simulation interfaces that hide as many implementation details as possible, to preserve the intellectual property of sub-systems. Previous studies [6], [7], [8], [9] have shown that obtaining a correct co-simulation result requires a co-simulation algorithm that is aware of some implementation details of each SU. For instance, a SU's input approximation function may expect values whose timestamp does not match what the master algorithm can provide, leading to hard to debug errors in the co-simulation results (see Sect. II-B for more details). This fact was highlighted in the work by Gomes et al. [6], [7], which showed that contracts/constraints on the co-simulation algorithm could be constructed based on information provided by the user, leading to improved co-simulation results (see also Sect. III for more related work). From a numerical perspective, [8], [9] demonstrate how to analyze the numerical stability of the co-simulation, and show how the results are affected by these implementation details.

Claudio: A thought: because the reviewers are already expecting this work to be an extension of previous work, we have to be careful to make sure that the paper is mostly self-contained. This means that we should not force the reader to

We are grateful to the Poul Due Jensen Foundation, which has supported the establishment of a new Centre for Digital Twin Technology at Aarhus University. Maurizio Palmieri is also grateful to the Italian Ministry of Education and Research (MIUR) in the framework of the CrossLab project (Department of Excellence).

read the previous work. It's pretty annoying as a reviewer if you have to go and read the previous N papers of the author.

a) Contribution: This manuscript extends the formalization introduced by Gomes et al. in [10] with the formalization of correctness for master algorithms dealing with step size correction and handling of algebraic loops. The semantics are implemented in UPPAAL[11] resulting in a co-simulation algorithm verifier that allows users to check whether their co-simulation algorithm respects the contract of the SUs.

The verifier has been applied to several case studies among other the industrial case study from [10] and non-trivial co-simulation scenarios which shall be simulated using a master algorithm both capable of solving algebraic loops and finding a common step between the SUs.

b) Organization: The manuscript starts with an introduction to co-simulation and formalization of SU in Sect. II. Sect. III describes other approaches for obtaining trustworthy and deterministic co-simulation results. Sect. IV follows with a presentation of the approach and the implemented verifier. Finally, Sect. V shows some of the applications where the verifier has been applied.

II. Background

This section introduces the concept of co-simulation along with a formalization of co-simulation.

A. Co-simulation

This paragraph will briefly summarize the main concepts related to co-simulation, and the reader is referred to [3] for a more detailed introduction of each concept.

Co-simulation is a technique to enable global simulation of a system consisting of multiple black-box SUs ofter developed individually. A SU captures the behavior of a dynamical system. A dynamical system is a system of a model that has a state that evolves over time according to some defined rules from an initial state. The dynamical system can engage with the environment through inputs and outputs[10].

SUs can be coupled through their inputs and outputs, indicating that the state of one SU is reliant on the state of another SU at all times - this is referred to as a coupling restriction. In practice, the coupling restrictions can only be satisfied at certain points in time - communication points. The orchestrator and SU agree on the communication points since these depend on the dynamical system the SU represents.

An SU has its own solver/simulator that can calculate its behavior trace at any given time based on the couplings to other SUs.

A co-simulation is executed by an orchestrator (or master), an algorithm that takes a set of SUs and their couplings (this is called a co-simulation scenario) computes the behavior trace of all units, trying to satisfy the coupling restrictions. The combined behavior trace is called the co-simulation.

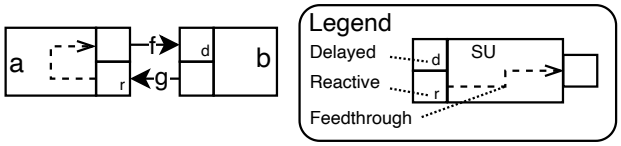

Figure 1. Simple Co-simulation scenario

B. Simulation Unit Definitions

The formalization of SUs is being done using the vocabulary from [10], and definitions adapted from [12].

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* $\mathcal{V}$*).*
- $\texttt{doStep}_c : S_c \times \mathbb{R}_{>0} \to S_c$ *is a function that instructs the SU to compute its state after a given time step* t*.*

Definition 2 (Scenario)**.** *A scenario is a structure* $\langle C, L \rangle$ *where each identifier* $c \in C$ *is associated with an SU, as defined in Definition 1, and* $L(u) = y$ *means that the output* y *is connected to input* u*. Let* $U = \bigcup_{c \in C} U_c$ *and* $Y = \bigcup_{c \in C} Y_c$*, then* $L : U \to Y$*.*

Note a single output can connect to multiple inputs, but a single input can only rely on a single output.

A co-simulation scenario is often presented graphically. In this paper the following semantics seen in Fig. 1 is adapted for presented scenarios graphically.

If an SU is in state $s_c^{(t)}$ at time t where $s_c^{(t)} \in S_c$, $\texttt{doStep}_c(s_c^{(t)}, H)$ approximates the state of the corresponding model at time $t + H$. The result of this approximation is encoded in state $s_c^{(t+H)}$.

Definition 3 (Run-time State)**.** *Given an SU c as defined in Definition 1, the run-time state of c at timestamp t is a member of the set* $S_c^R = \mathbb{R}_{\geq 0} \times S_{U_c}^R \times S_{Y_c}^R$*, where* $\mathbb{R}_{\geq 0}$ *is the time base,* $S_{U_c}^R = \prod_{u_c \in U_c} S_{u_c}^R$ *represents the aggregated state set of the input ports,* $S_{u_c}^R = \mathbb{R}_{\geq 0} \times \{defined, undefined\}$ *represents the set of states of an input port* $u_c \in U_c$*,* $S_{Y_c}^R = \prod_{y_c \in Y_c} S_{y_c}^R$ *represents the aggregated state set of the output ports, and* $S_{y_c}^R = \mathbb{R}_{\geq 0} \times \{defined, undefined\}$*.*

Definition 4 (Feed-through)**.** *The input* $u_c \in U_c$ *feeds through to output* $y_c \in Y_c$*, that is,* $(u_c, y_c) \in D_c$*, when there exists* $v_1, v_2 \in \mathcal{V}$ *and* $s_c \in S_c$*, such that* $\texttt{get}_c(\texttt{set}_c(s_c, u_c, v_1), y_c) \neq \texttt{get}_c(\texttt{set}_c(s_c, u_c, v_2), y_c)$*.*

Feed-through is an essential difference between the definition of an SU and the definition of an FMU. An FMU does not support feed-through; however, the work by Arnold [13], and Gomes [14] show the need for feed-through. Thus feed-through is being treated in this work.

Definition 5 (Reactivity)**.** *For a given SU c with input $u_c \in U_c$, $R_c(u_c) = true$ if the function $\texttt{doStep}_c$ assumes that the input u_c comes from a SU that has advanced forward relative to SU c.*

The following definitions correspond to the operations that are permitted in a co-simulation. Definitions 6 to 8 shows the effect of a given operation on the current runtime state $s_c^{(t)}$ of SU $c \in C$ at timestamp $t \in \mathbb{R}_{\geq 0}$. The operations in Definitions 6 to 8 are not normally pure functions, however it is chosen to make them pure in this formalization.

Definition 6 (Output Computation)**.** *The $\texttt{get}_c(s_c^{(t)}, y_c)$ represents the calculation of output y_c of $c \in C$. The operation is valid given a co-simulation state, if all inputs that feed-through to y_c are defined and have the same timestamp t.*

Definition 7 (Input Computation)**.** *The $\texttt{set}_c(s_c^{(t)}, u_c, v) \Rightarrow s_c^{(t)'}$ represents the setting of input u_c of $c \in C$. The operation is valid if given a co-simulation state, all outputs connected to u_c are defined and have the same timestamp t are defined. In that case, $s_c^{(t)'}$ is obtained by setting the run-time state of u_c of $s_c^{(t)}$ to defined with timestamp t.*

Definition 8 (Step Computation)**.** *The $\texttt{doStep}_c(s_c^{(t)}, H) \Rightarrow s_c^{(t+H)'}$ represents the progressing of SU c to time $t + H$. The success of this operation depends on satisfying all the following conditions:*

- *For every input port u_c that is non-reactive, $s_c^{(t)}$ must contain the state of u_c as defined at timestamp t.*
- *For every input port u_c that is reactive, $s_c^{(t)}$ must contain the state of u_c as defined at timestamp $t + H$.*
- *The SU c should have a maximal step $\texttt{h}_c \geq H$.*

If all these conditions hold, then $s_c^{(t+H)'}$ is obtained by setting all its output ports, to timestamp $t+H$, and setting all the output ports to undefined of $s_c^{(t)}$. In case the H is larger than $\texttt{h}_c$ the strategy described in Sect. II-C should be applied.

The state of an SU can be saved in an auxiliary variable and the present state of the SU can be overwritten by the state saved in an auxiliary variable.

Definition 9 (Step)**.** *Given a scenario $\langle C, L \rangle$, a co-simulation step is a finite ordered sequence of SU function calls $(f_i)_{i \in \mathbb{N}} = f_0, f_1, \ldots$ with $f_i \in F = \bigcup_{c \in C} \{\texttt{set}_c, \texttt{get}_c, \texttt{doStep}_c\}$, and i denoting the order of the function calls.*

As it can be seen from the conditions of operations defined in Definitions 6 to 8 the order of the operations inside a co-simulation step/algorithm is significant to obtain a correct co-simulation algorithm since an invalid algorithm will violate the constraints specified in the definitions.

An example of a co-simulation step algorithm of the scenario shown in Fig. 1 that satisfy the constraints of Definitions 6 to 8 can be seen in Algorithm 1.

Algorithm 1 Step function of scenario Fig. 1

1: $s_b^{(s+H)} \leftarrow \texttt{doStep}_b(s_b^{(s)}, H)$
2: $g_v \leftarrow \texttt{get}_b(s_b^{(s+H)}, y_b)$
3: $s_a^{(s)} \leftarrow \texttt{set}_a(s_a^{(s)}, u_g, g_v)$
4: $s_a^{(s+H)} \leftarrow \texttt{doStep}_a(s_a^{(s)}, H)$
5: $f_v \leftarrow \texttt{get}_a(s_a^{(s+H)}, y_f)$
6: $s_b^{(s+H)} \leftarrow \texttt{set}_b(s_b^{(s+H)}, u_f, f_v)$

Definition 10 (Initialization)**.** *Given a scenario $\langle C, L \rangle$, the initialization procedure is defined as sequence of SU function calls $(I_i)_{i \in \mathbb{N}} = I_0, I_1, \ldots$ with $I_i \in G = \bigcup_{c \in C} \{\texttt{set}_c, \texttt{get}_c\}$ and i denoting the order of the function calls.*

C. Step Finding

As described in Definition 8 each SU c at a specific state $s_c^{(t)}$ has a maximal step $\texttt{h}_c$ it is capable of performing on a $\texttt{doStep}_c$-action. The following assumptions by Broman [12] are made about a step operation.

Assumption 1. *For all $H \in \mathbb{R}_{>0}$ where $\texttt{h}_c \geq H \implies \texttt{doStep}_c(s_c^{(t)}, H) \Rightarrow s_c^{(t+H)'}$*

Assumption 2. *For all $H \in \mathbb{R}_{>0}$ where $\texttt{h}_c < H \implies \texttt{doStep}_c(s_c^{(t)}, H) \Rightarrow s_c^{(t+\texttt{h}_c)'}$*

These assumptions describe that an SU c at a given state maximal can advance with $\texttt{h}_c$ no matter the argument provided in the doStep-action. The size of $\texttt{h}_c$ is determined by the implementation of the SU and its current state $s_c^{(t)}$. A co-simulation scenario where not all SUs share the same maximal step for each co-simulation step will need a special strategy to ensure all SUs take a common step. A common step of the SUs is needed to satisfy the constraints of the actions specified in Definitions 6 to 8.

A capable master algorithm respects these constraints. It handles this problem by ensuring that a common step between all SUs is being performed at all co-simulation steps. A common step can be found by applying the Assumptions 1 and 2 that an SU c can take an arbitrary step greater than 0 and less or equal to $\texttt{h}_c$. A common step of all SUs is a member of the set defined in Eq. (1).

$$\{h \mid h \in \mathbb{R}_{>0} \text{ where } \texttt{h}_c \geq h \text{ for all } c \in C\} \tag{1}$$

The procedure for finding a common step will be described in more detail in Sect. IV where an algorithm is shown in Algorithm 4. This algorithm requires that all SUs in the step finding procedure support rollback to a previously saved state. If this is not the case, another strategy should be utilized where first the SUs capable of performing a rollback agree on a common step h using the

step finding routine. Afterward, the other SUs are called, and h is the size of the step that is being tried to perform on all other SUs in a single co-simulation step. It is asserted that all the other SUs can take a step of size h; if this is not the case, the simulation is aborted.

It is **important** to note that the $\mathtt{h}_c$ is dependent on $s_c^{(t)}$ so a common step from one iteration of the step procedure can not necessarily be used in other iterations.

D. Algebraic Loops

An algebraic loop in a co-simulation scenario occurs when the value retrieved on an output port depends on itself. The connections between the SUs can introduce such cyclic dependencies. An example of such a loop is shown in Fig. 2.

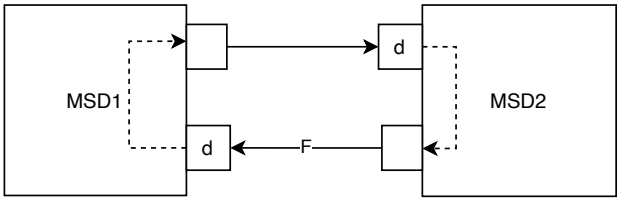

Figure 2. Co-simulation scenario with algebraic loop.

Algebraic loops are described and handled in more detail in ([14] see Definition 15). The dependency of a port to itself can reduce the chance of obtaining a deterministic co-simulation result [12], [15], [13] and lead to an unstable co-simulation. A master algorithm treating a scenario containing algebraic loops needs to apply a fixed-point iteration strategy. The fixed-point iteration strategy is where a finite ordered sequence of SU actions is applied repeatedly until convergence is established. Convergence refers to the value obtained on each output port in the loop is similar to the value obtained at the previous iteration. This algorithm is described in more detail in Sect. IV, where an algorithm is shown in Algorithm 5.

III. Related Work

The study of semantics and verification of co-simulation algorithms is done by Gomes et al. and Broman et al. in [14], [10], [12]. Gomes et al. describe a formalization of an FMI-based co-simulation scenario where several correctness criteria are placed on the co-simulation algorithm to generate and verify a co-simulation algorithm. This paper extends their work by treating co-simulation scenarios with algebraic loops and co-simulation with a variable step size. Thule et al. [16] studied how the characteristics of a co-simulation scenario can be used to choose the correct simulation strategy for a given co-simulation algorithm. They based their work on an implementation in Promela using Spin.

Broman et al. describe in [12] an approach to achieve deterministic co-simulation results by placing constraints on the co-simulation algorithm. Broman et al. also propose a generic master algorithm for handling co-simulation scenarios where a fixed-step master algorithm can not be applied. However, such generic algorithms do not consider other constraints on the simulation units like algebraic loops. This manuscript deals with all these constraints. Amálio et al.[15] study how connections between simulation units can be formalized. They use model checking and theorem proving to prove that the scenario adheres to certain healthiness properties so a deterministic co-simulation result can be obtained. Their work is limited to treating co-simulation scenarios without algebraic loops and where no simulation units that may reject a certain step.

The idea of using model checking to verify an FMI-based co-simulation scenario is also not new. Model checking and Uppaal have been used by Nyman et al. in [17]. Nyman et al. also look into how Uppaal can analyze controller systems with FMUs and a master algorithm modeled in Uppaal. Nyman et al. use Uppaal to verify properties about the controller used in the co-simulation. Also, Palmieri et at. in [18] have used Uppaal to provide sound guarantees on the interleaving between a graphical user interface and a generic FMI master algorithm. Our approach looks into the co-simulation semantics and applies to arbitrary co-simulation scenarios supporting both rollback and algebraic loops.

IV. Co-simulation Verifier

This section will describe the approach of the Verifier used to verify algorithms for non-trivial co-simulation scenarios. This will be presented by using a lot of different examples. Finally, the section will describe the implementation of the Verifier in Uppaal.

A. The Verifier

A user can verify a master algorithm of a given SU-based co-simulation scenario using the *Verifier*. Note that the verifier can also be used for FMI-based co-simulation scenarios if feed-through is avoided. The approach is graphical presented in Fig. 3.

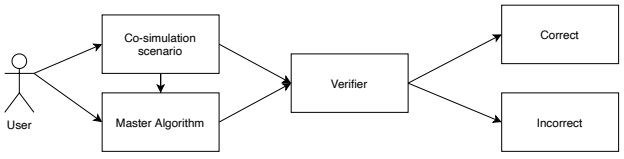

Figure 3. How the Interpreter is used to verify a co-simulation algorithm.

For example, if the user wants to verify the master algorithm, the user developed for the co-simulation scenario seen in Fig. 4. The scenario is non-trivial since it contains both an algebraic loop and some SUs that may reject a given step.

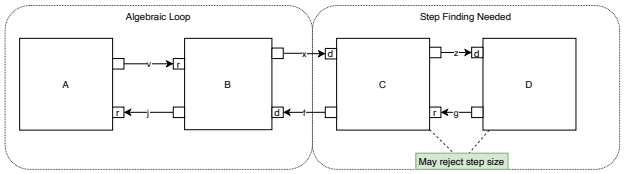

Figure 4. Co-simulation scenario with algebraic loop and need for Step size handling.

The master algorithm is a structure capturing all the operations performed on the SUs during the simulation. The operations defined in Definition 1 are denoted as *simple SU-actions* in the scope of the Verifier.

B. Initialization Procedure

The effect of the initialization procedure is to ensure all ports of the scenario are defined before the co-simulation starts. The initialization procedure consists only of the simple SU-actions *get* and *set* as defined in Definition 10. The initialization procedure of the scenario of Fig. 4 is shown in Algorithm 2.

Algorithm 2 Initialization procedure of the co-simulation scenario in Fig. 4.

$v_v \leftarrow \texttt{get}_A(s_A^{(0)}, y_v)$
$s_B^{(0)} \leftarrow \texttt{set}_B(s_B^{(0)}, u_v, v_v)$
$j_v \leftarrow \texttt{get}_B(s_B^{(0)}, y_j)$
$s_A^{(0)} \leftarrow \texttt{set}_A(s_A^{(0)}, u_j, j_v)$
$x_v \leftarrow \texttt{get}_B(s_B^{(0)}, y_x)$
$s_C^{(0)} \leftarrow \texttt{set}_C(s_C^{(0)}, u_x, x_v)$
$f_v \leftarrow \texttt{get}_C(s_C^{(0)}, y_f)$
$s_B^{(0)} \leftarrow \texttt{set}_B(s_B^{(0)}, u_f, f_v)$
$z_v \leftarrow \texttt{get}_C(s_C^{(0)}, y_z)$
$s_D^{(s)} \leftarrow \texttt{set}_D(s_D^{(0)}, u_z, z_v)$
$g_v \leftarrow \texttt{get}_D(s_D^{(0)}, y_g)$
$s_C^{(0)} \leftarrow \texttt{set}_C(s_C^{(0)}, u_g, g_v)$

Since the initialization procedure only contains *get* and *set*, the reactivity constraints do not apply in the initialization procedure. This means the co-simulation step of this scenario needs to handle an algebraic loop while the initialization procedure does not. However, scenarios where an algebraic loop should be solved in the initialization, do exist (see Fig. 2). The strategy for solving algebraic loops during the initialization is still to use a fixed-point strategy until convergence, as described later. The strategy for solving algebraic loops in the initialization is described in more detail in [19].

C. The Co-simulation Step

The co-simulation step defined in Definition 9 contains all the actions of a single co-simulation step. A co-simulation step of a simple co-simulation scenario (see Fig. 1) without any algebraic loops and where all SUs are capable of running a fixed step master algorithm is shown in Algorithm 1.

The co-simulation step of the scenario is shown in Algorithm 3. This scenario is considered non-trivial since it contains both an algebraic loop and cannot run a fixed-step co-simulation algorithm. The co-simulation step in Algorithm 3 consists of both simple SU actions, a step finding routine, and a fixed-point iteration procedure. The step finding routine and Fixed-Point Iteration procedure will be described in more detail in the later sections.

Algorithm 3 Co-simulation step of scenario in Fig. 4.

$s_{C_v}^{(s)} \leftarrow s_C^{(s)}$
$s_{D_v}^{(s)} \leftarrow s_D^{(s)}$
$h \leftarrow H\_max$
$StepFindingRoutine()$
$s_{A_v}^{(s)} \leftarrow s_A^{(s)}$
$s_{B_v}^{(s)} \leftarrow s_B^{(s)}$
$FixedPointIteration$
$x_v \leftarrow \texttt{get}_B(s_B^{(s+h)}, y_x)$
$s_C^{(s)} \leftarrow \texttt{set}_C(s_C^{(s+h)}, u_x, x_v)$
$f_v \leftarrow \texttt{get}_C(s_C^{(s+h)}, y_f)$
$s_B^{(s)} \leftarrow \texttt{set}_B(s_B^{(s+h)}, u_f, f_v)$

1) The Step Finding Routine: In case the co-simulation scenario does not allow a fixed step co-simulation algorithm to be executed, a special strategy should be used in each co-simulation step to finding the correct step size. An example of such a scenario is shown in Fig. 4, this example cannot be simulated using a fixed step algorithm since SU C and SU D may not be able to perform a step of arbitrary size. The strategy to be applied for such scenarios, to find a common step for each co-simulation step, is called a *Step Finding Routine*. Previous work of Broman et al.[12] looked into developing a procedure for finding a common step of all SUs of such scenarios. The step finding procedure is rather intuitive and works by performing a finite ordered sequence of SU-operations iteratively. The step size of all doStep-commands is shrunken in each iteration until all SUs take a step of the same size[1]. The first iteration performs a global maximal step h on all SUs and monitors how far each SU has progressed. The smallest step h_{min} of a single SU is identified, and it is checked if there exists an SU that has performed a greater step than h_{min}. An example of this approach is shown in Algorithm 4, where a common-step is found between the SUs C and D of Fig. 5.

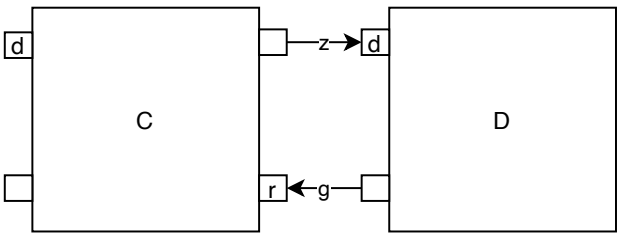

Figure 5. SUs of scenario in Fig. 4 that may reject a performing an arbitrary step.

[1]This is normally achieved after two iterations.

Algorithm 4 Step finding procedure of the example scenario.

1: **while** !$step\_found$ **do**
2: $s_D^{(s+h_D)} \leftarrow \texttt{doStep}_D(s_D^{(s)}, h)$
3: $g_v \leftarrow \texttt{get}_D(s_D^{(s+h_D)}, y_g)$
4: $s_C^{(s)} \leftarrow \texttt{set}_C(s_C^{(s)}, u_G, G_v)$
5: $s_C^{(s+h_C)} \leftarrow \texttt{doStep}_C(s_C^{(s)}, h_D)$
6: $z_v \leftarrow \texttt{get}_C(s_C^{(s+h_C)}, y_z)$
7: $s_D^{(s+h_C)} \leftarrow \texttt{set}_D(s_D^{(s+h_D)}, u_z, z_v)$
8: $h \leftarrow min(h_C, h_D)$
9: $step\_found \leftarrow h == h_C \wedge h == h_D$
10: **if** !$step\_found$ **then**
11: $s_C^{(s)} \leftarrow s_{C_v}^{(s)}$
12: $s_D^{(s)} \leftarrow s_{D_v}^{(s)}$
13: **end if**
14: **end while**

We want to stress that if the *doSteps*-actions on line 2 and line 5 of Algorithm 4 are not resulting in a step of the same size performed by both SU C and SU D: $s + H_D \neq s + H_C$. The conditions placed on *doStep* in Definition 8 is violated at the *doStep*-action of SU C on line 5 since the reactive input g is not defined at the timestamp $s + H_C$ which the SU C is progressing to. The approach taken in this work to handle this problem is to allow such violations inside the step finding procedure until a common step has been identified. Then all SUs are performing a step of the same size. There should be no violations of the constraints described in Definitions 6 to 8. This can be verified by running an extra iteration of the step finding procedure where all constraints are checked.

2) Solving algebraic loops: As stated in Sect. II-D co-simulation scenarios containing algebraic loops require a special strategy to obtain a trustworthy co-simulation result. The strategy to use is fixed-point iteration[2], where a finite ordered sequence of SU actions are repeated until a fixed-point is reached. The fixed-point refers to the values obtained on the output ports of the SUs in the algebraic loop. A fixed-point is obtained if the difference between the value retrieved on each output port between two successive iterations is less than a specified limit. If a fixed-point is not reached, another iteration of the procedure needs to be executed. However, the SUs should first be restored to their state before the iteration started. The last thing to do in the fixed-point procedure is to update all variables being set on reactive inputs in the next iteration. This ensures that the loop satisfies the reactivity constraints on the reactive inputs. Updating these variables ensures that each iteration of the loop is different from the previous iteration.

A concrete example of a fixed-point algorithm of the SUs shown in Fig. 6 is shown in Algorithm 5. The convergence of an algebraic loop is not guaranteed, and measures should be taken to catch such unstable scenarios.

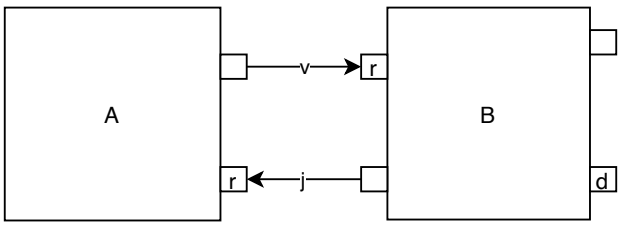

Figure 6. Algebraic loops of the scenario in Fig. 4.

Algorithm 5 Procedure for solving the algebraic loop of the example scenario.

1: **while** !$conv$ **do**
2: $s_A^{(s)} \leftarrow \texttt{set}_A(s_A^{(s)}, u_j, j_v)$
3: $s_B^{(s)} \leftarrow \texttt{set}_B(s_B^{(s)}, u_v, v_v)$
4: $s_A^{(s+h)}, \leftarrow \texttt{doStep}_C(s_A^{(s)}, h)$
5: $s_B^{(s+h)}, \leftarrow \texttt{doStep}_B(s_B^{(s)}, h)$
6: $v_a \leftarrow \texttt{get}_A(s_A^{(s+h)}, y_v)$
7: $j_a \leftarrow \texttt{get}_B(s_B^{(s+h)}, y_j)$
8: $conv \leftarrow CheckCon((j_a, j_v), (v_a, v_v))$
9: **if** !$conv$ **then**
10: $s_A^{(s)} \leftarrow s_{A_v}^{(s)}$
11: $s_B^{(s)} \leftarrow s_{B_v}^{(s)}$
12: **end if**
13: $v_v \leftarrow v_a$
14: $j_v \leftarrow j_a$
15: **end while**

The procedure for solving an algebraic loop will similarly with the step finding procedure violate some of the constraints placed on the SU actions described in Definitions 6 to 8. Algorithm 5 does, for example, violate the reactivity constraint of input v and input j on line 2 and 3 on the first iteration on the loop, since v_v on the first iteration has a timestamp s that is similar to the timestamp of SU B. The same applies to j_v and SU A. These violations are being treated similarly to the violations introduced in the step finding procedure. The violations are tolerated until a fixed-point is reached. Once convergence is established, all constraints on the actions should be respected. This is again checked by running an extra iteration of the loop where all constraints are checked.

The strategy for solving algebraic loops is only valid based upon the assumption that the convergence criteria will force the fixed-point variables (in Algorithm 5 it is: j_a, j_v, and v_a, v_v) to have a similar effect on the maximal step of the respective SU. This means the $\texttt{h}_A$ of SU A is affected in the same way by setting either J_v or J_a starting from the same state: $s_A^{(t)}$. This formulation can formally

be stated as:

$$CheckCon(B) \implies \forall (v,a) \in B.\mathtt{h}_v = \mathtt{h}_a$$
$$\text{where: } \mathtt{h}_v \text{ is obtained by: } \mathtt{set}_A(s_A^{(s)}, u_j, j_v)$$
$$\text{And: } \mathtt{h}_a \text{ is obtained by: } \mathtt{set}_A(s_A^{(s)}, u_j, j_a)$$
$$\text{And B: is the set of fixed-point variable pairs}$$

Note that an algebraic loop is solved differently in the initialization procedure compared to in the co-simulation step since there is no advancement of time in the initialization procedure. However, the underlying concept of using fixed-point iteration is similar.

3) Solving algebraic loops inside the step finding routine: Co-simulation scenarios exist where an algebraic loop needs to be solved inside a step-finding procedure**Simon: Do we have a real example of this: Claudio or Casper?** . An example of this is shown Fig. 8 where the algorithm also can be found in Algorithm 7. A master algorithm of such a scenario can be verified by the approach put forward here. The strategy is straightforward, using the procedures for solving algebraic loops and finding a common step as described earlier. However, as both the fixed-point iteration strategy and the step finding procedure break the constraints defined in Definitions 6 to 8 it is not enough to relax these constraints until convergence of the inner loop. The constraints should first be relaxed once convergence on the outer loop is established (in this case, a common step is taken by all). Once convergence is established on the outer loop, the strategy is similar to the other cases: An extra iteration of the outer loop is performed where all constraints are checked.

D. The implemented Model

This section introduces the Uppaal model of the *Verifier* developed in this project. The Verifier encapsulates the behavior and characteristics of a co-simulation Master-Algorithm. This means that a scenario and algorithm of co-simulation is passed to the *Verifier*. The Verifier will try to perform all the actions of the master algorithm. Since the Verifier is encapsulating all the constraints on the different actions, the verifier will ensure a master algorithm's correctness. The Verifier's implementation can be found at **Simon: Add URL to repository** together with the examples verified by the tool.

The co-simulation algorithm is divided into four distinct subsequent parts:

1) Instantiation: Instantiation and setting of Parameters on all each SU instance in the scenario
2) Initialization: Initialization of all ports described in Definition 10
3) Cosim-Step: The simulation part of the co-simulation described in Definition 9
4) Termination: Termination, Unloading on all each SU instance in the scenario

This monograph will only describe CoSim-Step since it is the most complex.

The model itself is divided into four different Uppaal templates - three of them (Interpreter, StepFinder, and LoopSolver) are responsible for the execution of the master algorithm by executing the operations on the SU, and the fourth template (SU) is encapsulating the state and rules of an SU. The relationship between the templates is shown in Fig. 7. As shown from Fig. 7 a typical model, this tool consists of multiple SU-templates and a single instance of the other templates.

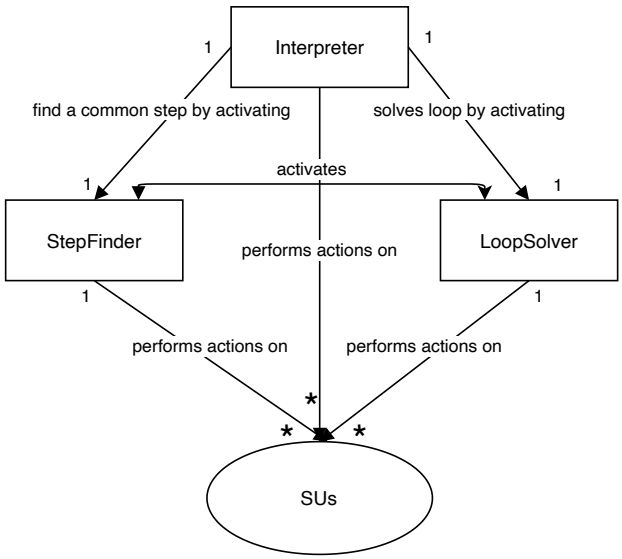

Figure 7. The relationship between the Uppaal templates.

The choice of dividing the execution of the master algorithm of a co-simulation into three parts allows nesting of the different template procedures - solving an algebraic loop inside the procedure of finding a common co-simulation step size or the other way around.

The Uppaal model is presented in a top-down approach where a co-simulation scenario's general encoding is described first. The Uppaal-templates will be introduced subsequently, starting with the Interpreter and ending with the SU.

1) Encoding of co-simulation scenario: A user specifies both the co-simulation scenario and a master algorithm for the *Verifier* to verify as shown in Fig. 3. The encoding of the co-simulation scenario is defined similarly as Definition 2 with several SUs, each with some input and output ports. The SUs are connected using connections and feed-through dependencies. The type (reactive/delayed) of each input port is also described as a part of the scenario. The encoding of the scenario also captures which SUs may reject a step of arbitrary size.

The user specifies the algorithm as a finite ordered sequence of function calls (operations) which the Interpreter invokes on the other Uppaal-templates as seen in Fig. 7. An action describes all the interactions between the Master Algorithm and the SUs, and the operations

are divided into two different categories:

- Simple actions: Actions performed by an SU.
- Complex actions: Actions not executable by an SU, but requires a special technique - implemented by another template.

The *Complex actions* is telling the *Verifier* that a special technique like fixed-point iteration or step finding routine should be performed on the scenario or a part of it.

E. The Interpreter

The Interpreter is the main entity of the *Verifier*. It is responsible for running the master algorithm from *Instantiation* to *Termination*. The Interpreter works by picking an action of the co-simulation scenario. The action being picked depends on the current state of the co-simulation. Once an action is selected, the Interpreter will try to perform it by in case of a *simple* action invoking it on a specified SU. If it is a *complex* action, the Interpreter will invoke it on either the StepFinder or LoopSolver. If the action is valid, it will be performed; otherwise, the model will reach a deadlock, and the user will be informed about the error.

An action is executable if none of the constraints specified in Definitions 6 to 8 are violated, and the current state of the co-simulation allows it. The co-simulation state criteria prevent getting a value from an SU that is not yet instantiated and other obvious violations.

F. StepFinder

The *StepFinder* is an optional part of the model that should be explicitly invoked by a complex action. It contains the logic described for verifying the Step finding routine.

The *StepFinder* routine is similar to the approach described earlier. It works by non-deterministically picking a $\mathtt{h}_c$ for each SU that may have a maximal step different for the global maximal step [2]. The *StepFinder* precedes by executing the step-finding routine described in the master algorithm as a finite order sequence of actions. When all actions have been performed, the co-simulation consistency is examined by looking at the step taken by all SUs involved in the routine. If the SUs did not take the same step, all SUs are rollback, and the step finding routine is performed again. In the next iteration, all SUs try to take the step of the size of the minimal step of the previous iteration. The routine is repeated until all SUs take a common step.

G. LoopSolver

The *LoopSolver* is another optional part of the model that should be explicitly invoked by a complex action by either the *StepFinder* or the *Interpreter*. It contains the fixed-point strategy described in Sect. II-D for solving an algebraic loop. The user should also supply the output ports on which convergence should be established.

[2]This is specified in the scenario.

The strategy for solving the loops is based on fixed-point iteration. A finite ordered sequence of specified actions is performed until convergence is established on all the convergence variables specified for each algebraic loop. Since convergence is not guaranteed to be established, the number of iterations is monitored to identify an unstable co-simulation scenario to avoid infinite iteration.

H. SU

The SU-templates encapsulates the behavior of an SU. The SU has all the different states described by the standard. The SU is a passive entity only active when the other templates activate it. The SU exposes an interface that the other entities invoke via a synchronization channel with a specified action to be performed. The SU will based on this synchronization and the value of some global variables, perform a simple SU-action.

Once an action has been performed, the SU informs the invoker of the action using another synchronization channel so the next operation of the master algorithm can be performed.

As described in Sect. II, all SU actions cannot always be performed due to the constraints described in Definitions 6 to 8. The SU-template is encapsulating these constraints using two different mechanisms - a state-based approach that disallows certain actions to be performed in a specific state. For example, a value cannot be obtained from a port if the SU is *Terminated*. The second mechanism uses preconditions on the actions encoded as state-invariants in Uppaal on the intermediate state of the SU actions.

As described earlier, the constraints from Definitions 6 to 8 are violated inside the step finding routine and inside a fixed-point iteration. These violations should be tolerated and are indeed necessary to perform the step finding routine. As a result of this, the preconditions of the actions can be turned on and off to allow verification of Master Algorithms of scenarios with algebraic loops and no natural simulation step. These scenarios are considered non-trivial.

1) Handling of Violations of the constraints of SU-action: Since both the *StepFinder* and *LoopSolver* unavoidably violate the constraints of Definitions 6 to 8 at the iterations where convergence is not established. An example of this is presented in Algorithms 4 and 5. The strategy taken to solve this problem is to tolerate the violations until convergence is established in both the *StepFinder* and the *LoopSolver*. Once convergence is established, it is important to ensure that no violation will happen at the final iteration where the converged parameters are used. The Verifier works by turning off the constraints until convergence is established. Then convergence is established, all SUs are restored, and an extra iteration is performed with the constraints turned on. This change of the co-simulation algorithm is shown in Algorithm 6 where the change has been applied to the fixed-point algorithm of Algorithm 5.

Algorithm 6 Changed Algebraic loop.

while $!conv \wedge !extraIteration$ **do**
  $s_A^{(s)} \leftarrow \texttt{set}_A(s_A^{(s)}, u_j, j_v)$
  $s_B^{(s)} \leftarrow \texttt{set}_B(s_B^{(s)}, u_v, v_v)$
  $s_A^{(s+h)}, \leftarrow \texttt{doStep}_C(s_A^{(s)}, h)$
  $s_B^{(s+h)}, \leftarrow \texttt{doStep}_B(s_B^{(s)}, h)$
  $v_a \leftarrow \texttt{get}_A(s_A^{(s+h)}, y_v)$
  $j_a \leftarrow \texttt{get}_B(s_B^{(s+h)}, y_j)$
  $conv \leftarrow CheckCon((j_a, j_v), (v_a, v_v))$
  if $!conv \wedge !extraIteration$ **then**
    $s_A^{(s)} \leftarrow s_{A_v}^{(s)}$
    $s_B^{(s)} \leftarrow s_{B_v}^{(s)}$
    if $conv$ **then**
      $extraIteration \leftarrow true$
    end if
  end if
  $v_v \leftarrow v_a$
  $j_v \leftarrow j_a$
end while

It is important to note that the real co-simulation algorithm should not be changed. The algorithm is only changed in the view of the *Verifier*.

I. Limitations of the model

The model is not capturing all valid SU-based co-simulation master algorithms, and this section will describe the shortcomings of the current model. The model will not be able to tell the difference between a wrong co-simulation algorithm and a co-simulation scenario that is not supported by the tool. A tool currently being developed to generate the encoding of a scenario will support this feature.

1) Hierarchical SUs: The model can only handle co-simulation scenarios where all the SUs have a common step for each iteration of the co-simulation stepping loop. This means that the Verifier is not capable of verifying simulations with hierarchical SUs, where the different SUs runs with a different frequency; for example, one SU takes one step of 10-time units while some other SUs take 10 steps of the one-time unit every time these different block synchronizes[20].

2) Assumption: The assumption that the convergence criteria will guarantee that a step of the same size in the next iteration can be taken is also a limitation of the tool. The authors of this paper do not currently possess the knowledge to judge the validity of this assumption. However, future work will be used to validate the assumption.

Simon: Maybe we should describe the scenarios what we have tested it on?

V. Case-study

The implemented Verifier has been tested using different co-simulation scenarios. The scenarios include both realistic scenarios and imaginary scenarios to explore the model's capabilities and limitations. Since an example of both the step-finding routine and solving algebraic loops has been shown in Sect. IV, these examples will be omitted from this section. The model was able to show the correctness of the presented algorithm of all the case studies. The model did also catch all invalid modification of the co-simulation algorithms.

The case study we present in this section is, unfortunately, a hypothetical scenario. It is chosen to show the capabilities of the Verifier. The scenario consists of three SUs, which all may have a unique maximal step. The SUs are connected so that all three SUs are a part of an algebraic loop. The scenario is shown in Fig. 8, and the algorithm of a valid co-simulation step is shown in Algorithm 7.

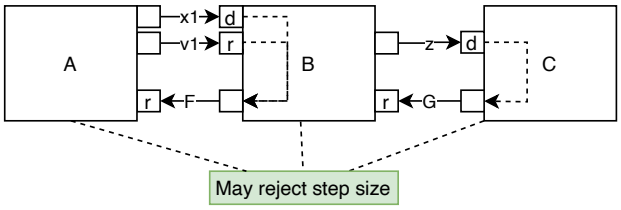

Figure 8. Case study scenario - Loop within Loop.

This scenario is indeed non-trivial due to the step rejection and algebraic loop. Therefore, it is clear that the co-simulation scenario should be simulated using a master algorithm containing both a fixed-point iteration for solving the algebraic loop inside the procedure for finding a common step size.

The algorithm of the scenario is shown in Algorithm 7. The fixed-point iteration (line 6-29) is nested inside the step finding routine (line 5-37). All ports of the scenario are a part of the algebraic loop due to their reactivity constraints. Thus all ports should be defined inside the fixed point iteration.

Algorithm 7 Co-simulation Step of Fixed-point Iteration inside Step finding procedure.

$s_{A_v}^{(S)} \leftarrow \texttt{Save}(s_A^{(s)})$
$s_{B_v}^{(S)} \leftarrow \texttt{Save}(s_B^{(s)})$
$s_{C_v}^{(S)} \leftarrow \texttt{Save}(s_C^{(s)})$
$h \leftarrow H\_max$
while !step_found **do**
  while !converged **do**
    $s_A^{(s)} \leftarrow \texttt{set}_A(s_A^{(s)}, u_F, F_v)$
    $s_B^{(s)} \leftarrow \texttt{set}_B(s_B^{(s)}, u_{v1}, v1_v)$
    $s_B^{(s)} \leftarrow \texttt{set}_B(s_B^{(s)}, u_G, G_v)$
    $s_C^{(s+h\_C)}, \leftarrow \texttt{doStep}_C(s_C^{(s)}, h)$
    $s_B^{(s+h\_B)}, \leftarrow \texttt{doStep}_B(s_B^{(s)}, h)$
    $s_A^{(s+h\_A)}, \leftarrow \texttt{doStep}_A(s_A^{(s)}, h)$
    $v1_a \leftarrow \texttt{get}_A(s_A^{(s+h\_A)}, y_{v1})$
    $z_v \leftarrow \texttt{get}_B(s_B^{(s+h\_B)}, y_z)$
    $s_C^{(s+h\_C)} \leftarrow \texttt{set}_C(s_C^{(s+h\_C)}, u_z, z_v)$
    $G_a \leftarrow \texttt{get}_C(s_C^{(s+h\_C)}, y_G)$
    $x1_v \leftarrow \texttt{get}_A(s_A^{(s+h\_A)}, y_{x1})$
    $s_B^{(s+h\_B)} \leftarrow \texttt{set}_B(s_B^{(s+h\_B)}, u_{x1}, x1_v)$
    $F_a \leftarrow \texttt{get}_B(s_B^{(s+h\_B)}, y_F)$
    $conv \leftarrow CheckCon((G_a G_v), (v1_a, v1_v), (F_a, F_v))$
    if !$conv$ **then**
      $s_A^{(s)} \leftarrow \texttt{Restore}_A(s_{A_v}^{(s)})$
      $s_B^{(s)} \leftarrow \texttt{Restore}_B(s_{B_v}^{(s)})$
      $s_C^{(s)} \leftarrow \texttt{Restore}_C(s_{C_v}^{(s)})$
    end if
    $G_v \leftarrow G_a$
    $v1_v \leftarrow v1_a$
    $F_v \leftarrow F_a$
  end while
  $h \leftarrow min(h\_A, h\_B, h\_C)$
  $step\_found \leftarrow$
  if !$step\_found$ **then**
    $s_A^{(s)} \leftarrow \texttt{Restore}_A(s_{A_v}^{(s)})$
    $s_B^{(s)} \leftarrow \texttt{Restore}_B(s_{B_v}^{(s)})$
    $s_C^{(s)} \leftarrow \texttt{Restore}_C(s_{C_v}^{(s)})$
  end if
end while

Maurizio: I think it should be useful to show a counterexample: a list of operations that looks correct, but with the Verifier, we can prove that it leads to a violation of the definition (i.e., a deadlock)

A. Industrial case study

The Industrial case study of the paper [10] has also been verified. Since this example is far more trivial than the example above, it will not be presented in detail in this paper.

VI. Concluding Remarks

This paper looked at how model-checking could be used to verify a co-simulation algorithm for an FMI/SU-based co-simulation formally. The approach can handle non-trivial co-simulation scenarios containing both algebraic loops and co-simulation algorithms with a variable step size.

Future work will look into how this modeling approach can be extended to cover a broader class of co-simulation scenarios, for example, scenarios with hierarchical SUs which already have been studied by Gomes et al. [20]. An interesting project could be to see if the Uppaal *Verifier* could be turned into a co-simulation algorithm synthesizer. Knowledge about the SU semantics has also been gathered, which could be explored in another formal system like an assistant theorem prover like Isabelle/HOL. A potentially broader class of co-simulation algorithms could be verified in an assistant theorem prover.

It is also extended to use the model as a foundation to explore the use cases of experimental features of the FMI 3 standard like clocks.

Acknowledgements: We would like to thank Stefan Hallerstede, Tomas Kulik, and Jalil Boudjadar for providing valuable input to this paper and the developed Uppaal model.

References

[1] E. A. Lee, "Cyber physical systems: Design challenges," in *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, 2008, pp. 363–369.

[2] R. Kübler and W. Schiehlen, "Two methods of simulator coupling," *Mathematical and Computer Modelling of Dynamical Systems*, vol. 6, no. 2, pp. 93–113, 2000.

[3] C. Gomes, D. Broman, H. Vangheluwe, C. Thule, and P. G. Larsen, "Co-simulation: A survey," *ACM Computing Surveys*, vol. 51, no. 3, pp. 49–49, 2018. [Online]. Available: https://doi.org/10.1145/3179993

[4] T. Blockwitz, M. Otter, J. Akesson, M. Arnold, C. Clauss, H. Elmqvist, M. Friedrich, A. Junghanns, J. Mauss, D. Neumerkel, H. Olsson, and A. Viel, "Functional mockup interface 2.0: The standard for tool independent exchange of simulation models," in *Proceedings of the 9th International MODELICA Conference, September 3-5, 2012, Munich, Germany*, vol. 76. Linköping University Electronic Press, 2012, pp. 173–184.

[5] FMI, *Functional Mock-up Interface for Model Exchange and Co-Simulation*. FMI development group, 2014. [Online]. Available: https://fmi-standard.org/downloads/

[6] C. Gomes, B. Oakes, M. Moradi, A. Gámiz, J. Mendo, S. Dutré, J. Denil, and H. Vangheluwe, "HintCO – hint-based configuration of co-simulations:," in *Proceedings of the 9th International Conference on Simulation and Modeling Methodologies, Technologies and Applications*. SCITEPRESS - Science and Technology Publications, 2019, pp. 57–68. [Online]. Available: http://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0007830000570068

[7] B. J. Oakes, C. Gomes, F. R. Holzinger, M. Benedikt, J. Denil, and H. Vangheluwe, "Hint-based configuration of co-simulations with algebraic loops," in *Simulation and Modeling Methodologies, Technologies and Applications*. Springer International Publishing, 2020, vol. 1260, pp. 1–28.

[8] C. Gomes, C. Thule, K. Lausdahl, P. G. Larsen, and H. Vangheluwe, "Stabilization technique in INTO-CPS," in *2nd Workshop on Formal Co-Simulation of Cyber-Physical Systems*, vol. 11176. Springer, Cham, 2018.

[9] B. Schweizer, P. Li, and D. Lu, "Explicit and implicit cosimulation methods: Stability and convergence analysis for different solver coupling approaches," *Journal of Computational and Nonlinear Dynamics*, vol. 10, no. 5, p. 051007, 2015, publisher: ASME.

[10] C. Gomes, L. Lucio, and H. Vangheluwe, "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, 2019, pp. 784–789.

[11] G. Behrmann, A. David, K. Larsen, J. Hakansson, P. Petterson, W. Yi, and M. Hendriks, "UPPAAL 4.0," in *Third International Conference on Quantitative Evaluation of Systems (QEST 2006)*, 2006, pp. 125–126.

[12] D. Broman, C. Brooks, L. Greenberg, E. A. Lee, M. Masin, S. Tripakis, and M. Wetter, "Determinate composition of FMUs for co-simulation," in *Eleventh ACM International Conference on Embedded Software*. IEEE Press Piscataway, NJ, USA, 2013, p. Article No. 2.

[13] M. Arnold, C. Clauß, and T. Schierz, "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, 2014, pp. 107–125, event-place: Berlin, Heidelberg.

[14] C. Gomes, C. Thule, L. Lúcio, H. Vangheluwe, and P. G. Larsen, "Generation of co-simulation algorithms subject to simulator contracts," in *Software Engineering and Formal Methods*, ser. Lecture Notes in Computer Science, J. Camara and M. Steffen, Eds. Springer International Publishing, 2020, pp. 34–49.

[15] N. Amálio, R. Payne, A. Cavalcanti, and J. Woodcock, "Checking sysML models for co-simulation," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 10009 LNCS. Springer Verlag, 2016, pp. 450–465, ISSN: 16113349. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-47846-3_28

[16] C. Thule, C. Gomes, J. Deantoni, P. G. Larsen, J. Brauer, and H. Vangheluwe, "Towards the verification of hybrid co-simulation algorithms," in *Software Technologies: Applications and Foundations*, M. Mazzara, I. Ober, and G. Salaün, Eds. Springer International Publishing, vol. 11176, pp. 5–20, series Title: Lecture Notes in Computer Science.

[17] P. G. Jensen, K. G. Larsen, A. Legay, and U. Nyman, "Integrating tools: Co-simulation in UPPAAL using FMI-FMU," in *2017 22nd International Conference on Engineering of Complex Computer Systems (ICECCS)*. IEEE, 2017, pp. 11–19. [Online]. Available: http://ieeexplore.ieee.org/document/8292158/

[18] M. Palmieri, C. Bernardeschi, and P. Masci, "A framework for FMI-based co-simulation of human–machine interfaces," *Software and Systems Modeling*, vol. 19, no. 3, pp. 601–623, 2020.

[19] S. Thrane, C. Thule, and C. Gomes, "An FMI-based initialization plugin for INTO-CPS maestro 2," in *Software Engineering and Formal Methods*. Springer, 2020, p. To appear.

[20] C. Gomes, B. Meyers, J. Denil, C. Thule, K. Lausdahl, H. Vangheluwe, and P. De Meulenaere, "Semantic Adaptation for FMI Co-simulation with Hierarchical Simulators," vol. 95, no. 3, pp. 1–29, 2019.