Verification of Co-simulation Algorithms Subject to Algebraic Loops and Adaptive Steps

Simon Thrane Hansen[1(✉)], Cláudio Gomes[1], Maurizio Palmieri[2], Casper Thule[1], Jaco van de Pol[3], and Jim Woodcock[1,4]

[1] DIGIT, Department of Electrical and Computer Engineering, Aarhus University, Aarhus, Denmark
sth@ece.au.dk

[2] DII, Department of Information Engineering, University of Pisa, Pisa, Italy

[3] DIGIT, Department of Computer Science, Aarhus University, Aarhus, Denmark

[4] Department of Computer Science, University of York, York, UK

Abstract. Simulation-based analyses of cyber-physical systems are increasingly vital. Co-simulation is one such technique that enables the coupling of specialized simulation tools through an orchestration algorithm. The orchestrator dictates how each simulation tool should simulate its corresponding subsystem. Obtaining correct simulation results requires an implementation-aware orchestration algorithm tailored to the specific scenario, without the orchestrator knowing each simulation tool's implementation. Such an algorithm should stabilize algebraic loops, perform time step negotiation, and adhere to each simulation tool's implementation. This paper describes an approach and implementation to prove that a given orchestration algorithm respects all contracts related to the simulation units' implementation. The approach has been applied to an industrial case study and other complex scenarios. The tool and results are available online.

Keywords: Co-simulation · Model-checking · Cyber-physical systems

1 Introduction

Cyber-physical systems (CPS) are omnipresent and embody physical processes being controlled by cyber elements. A CPS is typically developed in a distributed fashion using different tools and techniques. Such systems are becoming increasingly complex [18], which leads to the desire for techniques to assist in the development of these. One such technique is co-simulation: the study of how

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).

A. Lluch Lafuente and A. Mavridou (Eds.): FMICS 2021, LNCS 12863, pp. 3–20, 2021.
https://doi.org/10.1007/978-3-030-85248-1_1

to coordinate multiple black-box simulation units (SUs), each responsible for computing the behavior of a sub-system, in order to compute their combined behavior, and therefore produce the global behavior of a system, as a discrete trace (see, e.g., [8,17]).

Co-simulation allows iterative integration of constituents to explore the global system behavior without violating the constituents' intellectual property. The SUs are coupled by an orchestration algorithm that interacts with each SU through an interface. An example of such an SU is a Functional Mock-up Unit (FMU) defined by the Functional Mock-up Interface Standard [4] (FMI), which inspires the notion of an SU in this paper. FMI is a widely adopted standard used commercially and supported by many tools [7].

The overarching challenge of co-simulation is ensuring correct simulation results. Previous studies [11,12,19,21] have shown that obtaining a correct co-simulation result requires an algorithm specifically tailored to the scenario that respects the SUs' input approximation functions. Not considering such details can lead to hard to debug errors in the co-simulation results as highlighted in [11,19], where it is shown how contracts on the co-simulation algorithm could be constructed based on the SUs. Obeying such contracts leads to a substantial reduction of co-simulation errors (see also Sect. 3 for more related work). An even more challenging class of scenarios to simulate are complex scenarios subject to either algebraic loops or adaptive steps. Complex scenarios are simulated using a specific iterative algorithm [14]. The iterative algorithm solves the algebraic loop (cyclic dependencies between the SUs) and ensures that all SUs agree on a step; the latter is referred to as step negotiation. Step negotiation permits the SUs to implement error estimation and refuse certain future state evaluations to minimize the simulation error while ensuring that the SUs move in lockstep. We propose an approach that has been implemented as a tool. The tool lets users verify that their algorithm respects the contracts of the SUs.

Contribution: This paper describes an approach for verifying that a co-simulation algorithm satisfies the contracts of the scenario. The approach covers complex scenarios subject to algebraic loops and adaptive steps. It has been implemented in UPPAAL [3] and has been applied to several case studies, including an industrial case study from Boeing [9] and complex scenarios subject to algebraic loops and step negotiation.

Structure: The paper starts with introducing co-simulation and the verification challenge of co-simulation algorithms in Sect. 2. Section 3 describes other approaches for obtaining reliable and deterministic co-simulation results. Section 4 follows with a presentation of the verification technique. Section 5 discusses a case study and Sect. 6 concludes.

2 Background

Co-simulation is a technique enabling global simulation of a system consisting of multiple black-box SUs. An SU has its own solver that calculates the behavior

trace of the dynamical system it represents. A dynamical system is a function from time and space into some often multi-dimensional and continuous space. Examples include population growth, water flow, and pendulums. The system interacts with the environment through inputs and outputs [9,17].

2.1 Simulation Units

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 - known as a coupling restriction. However, in practice, the coupling restrictions can only be satisfied at certain points in time, referred to as communication points. Furthermore, each SU makes assumptions about the evolution of the input values between the communication points, which can cause accumulable errors in the co-simulation [2].

A scenario is simulated using an orchestrator - an algorithm - that computes the behavior trace of all SUs trying to satisfy their coupling restrictions by exchanging values. The orchestrator's goal is to find the communication points that minimize the error introduced in the co-simulation and to ensure that the SUs move in lockstep. Studies [10–12,19,21] have shown that optimal communication points depend on the implementation of the SUs.

Definition 1 (Simulation Unit). *An SU with identifier c is represented by the tuple*

$$\langle S_c, U_c, Y_c, \texttt{set}_c, \texttt{get}_c, \texttt{step}_c \rangle ,$$

where:

- S_c *represents the state space.*
- U_c *and* Y_c *the set of input and output variables, respectively.*
- $\texttt{set}_c : S_c \times U_c \times \mathcal{V}_\mathcal{E} \to S_c$ *and* $\texttt{get}_c : S_c \times Y_c \to \mathcal{V}_\mathcal{E}$ *are functions to set the inputs and get the outputs, respectively (we abstract the set of values exchanged between input/output variables as* $\mathcal{V}_\mathcal{E}$*. The type of this set is the tuple* $\langle t, \mathcal{V} \rangle$*, where* $\mathcal{V}$ *denotes the value obtained at a given output port and* $t : \mathbb{R}_{\geq 0}$ *denotes the timestamp of* c *when the value was obtained by an action respecting the contracts).*
- $\texttt{step}_c : S_c \times \mathbb{R}_{>0} \to S_c \times \mathbb{R}_{>0}$ *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*,* $(s_c^{(t+h)}, h) = \texttt{step}_c(s_c^{(t)}, H)$ *approximates the state* $s_c^{(t+h)}$ *of the corresponding model at time* $t + h$*, where* $h \leq H$*.*

Definition 1 is inspired by [5,13] and represents a symbolic version of an SU. The state of SU A at time t is denoted $s_A^{(t)}$. We assume the last value set on an input/output port can be inspected, for example, the value of input u_x could be $u_x = \langle t, v_x \rangle$, where t is the timestamp when the value v_x set on u_x was obtained. The function $\texttt{step}_c$ returns a step size because some SUs implement error estimation and may conclude that taking a step size of H will result in an intolerable error meaning the SU takes a smaller step than planned.

Definition 2 (Scenario). *A scenario is a structure* $\langle C, L, M, F, R, D\rangle$ *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* $L : U \to Y$*.* $M \subseteq C$ *denotes the SUs that implement error estimation. The set of reactive components,* $R = \bigcup_{c\in C} R_c$*, where* $R_c(u_c) = true$ *means the function* $\texttt{step}_c$ *assumes that the input* u_c *comes from an SU that has advanced forward relative to SU* c*. The set of delayed components,* $D = \bigcup_{c\in C} \neg R_c$*, where* $R_c(u_c) = false$ *means the function* $\texttt{step}_c$ *assumes that the input* u_c *comes from an SU that is at the same time as SU* c*. Finally, the set of feed-through components,* $F = \bigcup_{c\in C} F_c$*, where the input* $u_c \in U_c$ *feeds through to output* $y_c \in Y_c$*, that is,* $(u_c, y_c) \in F_c$*, when there exists* $v_1, v_2 \in \mathcal{V}_\mathcal{E}$ *and* $s_c \in S_c$*, such that* $\texttt{get}_c(\texttt{set}_c(s_c, u_c, v_1), y_c) \neq \texttt{get}_c(\texttt{set}_c(s_c, u_c, v_2), y_c)$.

The syntax in Fig. 1 is used to graphically present co-simulation scenarios. The couplings of SUs and feedthrough can introduce algebraic loops like the one seen in the scenario in Fig. 2a: The port variables in that scenario form a cyclic dependency, requiring that all their values are being set at the same time. The set of port variables involved in algebraic loops are the port variables of the non-trivial SCCs in the step operation graph, constructed based on Definition 15 in [13]. The set $algebraic_S$ denotes such variables in scenario S:

$$algebraic_S \triangleq \{s \mid \text{for each } s \in SCCs \wedge s \in U \cup Y\},$$
$$\text{where } SCCs: \text{ is the flatten set of all nontrivial SCCs in S.}$$

The input variables involved in an algebraic loop in the scenario S are:

$$U_{algebraic_S} \triangleq algebraic_S \cap U \tag{1}$$

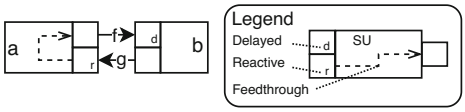


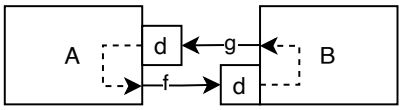
Fig. 1. A simple co-simulation scenario ($S1$).

Using the given definition of a scenario, we call a co-simulation scenario either simple or complex.

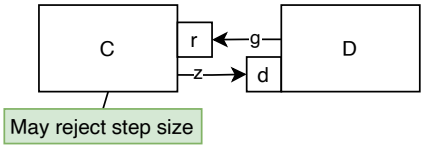
Definition 3. *A scenario* S *is simple if* $M = \emptyset \wedge algebraic_S = \emptyset$.

A scenario is complex if it is not simple. Using Definition 3 we conclude that the scenario ($S1$) in Fig. 1 is simple because $algebraic_{S1} = \emptyset$ and none of the SUs implement error estimation ($M_{S1} = \emptyset$). The scenario ($S2$) in Fig. 2a is complex since $algebraic_{S2} = \{u_f, u_g, y_f, y_g\}$ meaning that all variables are a part of a cyclic dependency that should be solved using a fixed point. The scenario ($S3$) in Fig. 2b is also complex because SU C implements error estimation ($M_{S2} = \{C\}$) and therefore can perform step rejection. Step rejection requires special attention since the orchestrator should backtrack the simulation and restart the simulation with a smaller step in case of a step rejection.

The reason for distinguishing between simple and complex scenarios is that the simulation strategy depends on the scenario type. This is treated in more



(a) Scenario $S2$ with an algebraic loop.



(b) Scenario $S3$ needs step negotiation.

Fig. 2. Complex co-simulation scenarios.

detail later in the paper. Next, we give a brief presentation of the contracts from [9] before describing how they can be used to verify a co-simulation algorithm.

The contracts are described as preconditions of the SU-actions `get`, `set` and `step`. It should be pointed out that each input and output has a timestamp referring to when it was last successfully activated by an action.

Definition 4 (Get Action). *The precondition of* $\texttt{get}_c(s_c^{(t)}, y_c)$ *is:*

$\forall(u_\_, y_c) \in F_c \implies u_\_ = \langle t_h, \_\rangle \wedge t_h = t.$

Informally saying that all inputs that feeds through to y_c *should be set. This criterion is denoted as the predicate:* $\texttt{preGet}_c : S_c \times Y_c \to \mathbb{B}$.

Definition 5 (Set Action). *The precondition of* $\texttt{set}_c(s_c^{(t)}, u_c, v)$ *depends on the contract of the input:*

- *If* $u_c \in R$ *then the operation is valid if* $v = \langle t_h, \_\rangle$, *where* $t_h = t + H$
- *If* $u_c \in D$ *then the operation is valid if* $v = \langle t_h, \_\rangle$, *where* $t_h = t$.

This informally says that the value set on u_c *should be obtained at a meaningful point in time dictated by the input contract. We denote this as the predicate:* $\texttt{preSet}_c : S_c \times U_c \times \mathcal{V}_\mathcal{E} \to \mathbb{B}$.

The criterion on a set-action will first have an effect when the SU is stepped. However, we found that it was easier to catch and correct an incorrect algorithm by moving this criterion to the set-action.

Definition 6 (Step Computation). *The precondition of* $\texttt{step}_c(s_c^{(t)}, H)$ *is satisfied if all the following conditions are fulfilled:*

- $\forall u_c \in U_c . u_c \in D \implies u_c = \langle t_h, \_\rangle \wedge t_h = t.$
- $\forall u_c \in U_c . u_c \in R \implies u_c = \langle t_h, \_\rangle \wedge t_h = t + H.$

Informally, this is that all inputs should set with a new value since the last time the SU was stepped. This is denoted as the predicate: $\texttt{preStep}_c : S_c \times \mathbb{R}_{>0} \to \mathbb{B}$.

Definition 7 (Preconditions of a Scenario). *The set of all preconditions Pre of a scenario is the union of the preconditions for each SU* $c \in C$*:*

$$Pre = \bigcup_{c \in C} \left\{ \bigcup_{i \in U_C} \texttt{preSet}_c(\_, u_i, \_), \bigcup_{i \in Y_C} \texttt{preGet}_c(\_, y_i), \texttt{preStep}_c \right\} \tag{2}$$

The FMI-standard [4] also describes some contracts on the state-changing function. The implementation in UPPAAL enforces them, but they are not treated in this paper.

2.2 Co-simulation Algorithms

A scenario is simulated by a co-simulation algorithm that consists of state-changing functions, an initialization procedure, and a co-simulation step. This work concentrates on the co-simulation step, which we refer to as the algorithm throughout the paper. The other aspects of a co-simulation algorithm can trivially be derived from the method.

The purpose of the co-simulation step is to move all SUs C, and all inputs and outputs $(U \cup Y)$, from the initial times t to some future time $t+H$, where $H > 0$. We use function $\texttt{ftime} : S_C \to \mathbb{R}_{\geq 0}$ to obtain the current timestamp of an SU. This makes it possible to define the Hoare-triple of the co-simulation step P:

$$Hoare(P) \triangleq \{\forall v \in U \cup Y.\, v = \langle t, \_\rangle \wedge \forall c \in C.\, \texttt{ftime}(s_c^{(t)}) = t\} \quad P \quad \{\forall v \in U \cup Y.\, v = \langle t+H, \_\rangle \wedge \forall c \in C.\, \texttt{ftime}(s_c^{(t+H)}) = t+H\}$$

A co-simulation step P is a sequence of instructions using the SU's functions $\texttt{set}_c, \texttt{get}_c$, and $\texttt{step}_c$. Each index i of the sequence $P[i]$ represents an action in the algorithm, for example, if P is the co-simulation step in Algorithm 1 $P[0] = \texttt{step}_A(s_A^{(0)}, \_)$. Figure 3 shows three different co-simulation steps of the scenario in Fig. 1.

Algorithm 1

1: $(s_A^{(H)}, H) \leftarrow \texttt{step}_A(s_A^{(0)}, H)$
2: $(s_B^{(H)}, H) \leftarrow \texttt{step}_B(s_B^{(0)}, H)$
3: $f_v \leftarrow \texttt{get}_A(s_A^{(H)}, y_f)$
4: $g_v \leftarrow \texttt{get}_B(s_B^{(H)}, y_g)$
5: $s_B^{(H)} \leftarrow \texttt{set}_B(s_B^{(s)}, u_f, f_v)$
6: $s_A^{(H)} \leftarrow \texttt{set}_A(s_A^{(H)}, u_g, g_v)$

Algorithm 2

1: $(s_B^{(H)}, H) \leftarrow \texttt{step}_B(s_B^{(0)}, H)$
2: $(s_A^{(H)}, H) \leftarrow \texttt{step}_A(s_A^{(0)}, H)$
3: $g_v \leftarrow \texttt{get}_B(s_B^{(H)}, y_g)$
4: $s_A^{(H)} \leftarrow \texttt{set}_A(s_A^{(H)}, u_g, g_v)$
5: $f_v \leftarrow \texttt{get}_A(s_A^{(H)}, y_f)$
6: $s_B^{(H)} \leftarrow \texttt{set}_B(s_B^{(H)}, u_f, f_v)$

Algorithm 3

1: $(s_B^{(H)}, H) \leftarrow \texttt{step}_B(s_B^{(0)}, H)$
2: $g_v \leftarrow \texttt{get}_B(s_B^{(H)}, y_g)$
3: $s_A^{(0)} \leftarrow \texttt{set}_A(s_A^{(0)}, u_g, g_v)$
4: $f_v \leftarrow \texttt{get}_A(s_A^{(0)}, y_f)$
5: $s_B^{(H)} \leftarrow \texttt{set}_B(s_B^{(H)}, u_f, f_v)$
6: $(s_A^{(H)}, H) \leftarrow \texttt{step}_A(s_A^{(0)}, H)$

Fig. 3. Three algorithms conforming to the FMI standard (version 2.0) of the scenario in Fig. 1.

Although the three algorithms in Fig. 3 consist of the same actions, they are not equivalent, and simulating with one algorithm instead of one of the others could drastically change the co-simulation result as shown in [13]. They showed that by obeying the contracts, the scenario will be simulated correctly. We assume that the contracts in the scenario are constant through the simulation, which is the case for most commercially used SUs. At the end of Sect. 2.3, we show which of these algorithms is correct.

A co-simulation step P is executed using a configuration c. The configuration $c \triangleq \langle H, guess \rangle$ consists of the parameters of the co-simulation step P. $H \in \mathbb{R}_{>0}$ defines the step size, and $guess : U_{algebraic} \rightarrow \mathcal{V}_{\mathcal{E}}$ is a total function linking all inputs in $U_{algebraic}$ to a guess that tries to satisfy the algebraic loops. Using the example from Algorithm 1, the action at index 0 in P applied with the configuration $\langle 1, \_ \rangle$ is: $P[0](c) = \texttt{step}_A(s_A^{(0)}, 1)$. The configuration defines the step size (1) of the step-action.

The set *Configurations* denotes all the possible configurations of the co-simulation step for a given scenario. The execution of a co-simulation step P is the execution of each action in P. We define such execution of P using configuration c as:

$$P(c) \triangleq \text{for each } i \in dom(P).\, P[i](c) \tag{3}$$

An execution of $P(c_j)$ yields another configuration $c_{j+1} \in$ *Configurations* where $P(c_j) = c_{j+1}$. The configuration $c_{j+1} : \langle H_1, guess_1 \rangle$ is obtained from the algorithm P and configuration $c_j : \langle H, guess \rangle$ by updating H_1 to the smallest step accepted by an SU during the execution of $P(c_j)$. And the function $guess_1$ has the same domain as $guess$, but the range is updated to the new value of the output coupled to the associated input in the domain of $guess$ after executing $P(c_j)$.

$$guess_{j+1}(u) = value(u, P(c_j)) \text{ and } H_1 = minStep(P(c_j)) \tag{4}$$

The execution of a configuration $c : \langle H, guess \rangle$ has converged if all SUs accept the step H, and all algebraic loops are stabilized (all values in the range of $guess$ are fixed-points). The domain of $guess$ is all the inputs in $U_{algebraic}$ for the scenario; this means that all configurations of the same scenario have the same domain.

Definition 8. *Two configurations of the same scenario S* $c_j : \langle H_1, guess_1 \rangle \in$ *Configurations and* $c_{j+1} : \langle H_2, guess_2 \rangle \in$ *Configurations are convergent if:*

$$c_j \approx c_{j+1} \triangleq H_1 = H_2 \wedge (\forall i \in dom(guess).\, guess_1[i] \approx guess_2[i])$$

Two values $v1_E : (v_1, t_1)$ *and* $v2_E : (v_2, t_2)$ *of type* $\mathcal{V}_{\mathcal{E}}$ *converge if:*

$$v1_E \approx v2_E \triangleq \mid v_1 - v_2 \mid \leq \epsilon \wedge t_1 = t_2 \tag{5}$$

Formally an execution of a co-simulation step P of a configuration c_j is stable or has converged if $P(c_j) = c_{j+1} \implies c_j \approx c_{j+1}$.

A complex scenario is a scenario where not all configurations are stable. Such a scenario can only be correctly simulated by an algorithm P if a convergent configuration exists:

$$\exists c \in \textit{Configurations}, \exists j \in \mathbb{N}.\, P(c_j) = c_{j+1} \implies c_j \approx c_{j+1} \tag{6}$$

Some measures should be taken to handle cases where no convergent configuration exists.

2.3 Correct Co-simulation Algorithms

To optimally simulate a co-simulation scenario using an algorithm P requires more than a convergent configuration c. The algorithm P should also successfully satisfy all the preconditions/contracts. To describe this, we introduce the sequence $\mathcal{C}$, which is a permutation of the set Pre (cf. Definition 7). The sequence $\mathcal{C}$ is constructed by a function $\mathcal{C} = contracts(P, Pre)$ that for each action in P finds the corresponding precondition $pre \in Pre$ and adds it to $\mathcal{C}$ such that for an arbitrary index i in P, $\mathcal{C}[i]$ is the precondition of the action $P[i]$.

If an action at index i in P satisfies its precondition $\mathcal{C}[i]$ using the configuration c, it is denoted as:

$$P[i](c) \models \mathcal{C}[i] \tag{7}$$

A co-simulation step P using a configuration c satisfies $\mathcal{C}$ if all actions satisfy its precondition.

$$P(c) \models \mathcal{C} \triangleq \forall i \in dom(P).\, P[i](c) \models \mathcal{C}[i] \tag{8}$$

$P(c) \models \mathcal{C}$ means the algorithm respects the scenario's contracts. Based on previous studies it is well-known that a non-convergent configuration c does not respect all the contracts:

$$c_j \not\approx c_{j+1} \implies P(c_j) \not\models \mathcal{C} \tag{9}$$

Therefore we only check the contracts of the scenario if the current configuration is convergent. We can now describe what it means for an algorithm to be correct for a given scenario in the following Hoare triple.

Definition 9. *An algorithm P and configuration $c : \langle H, \_ \rangle$ is correct if:*

$$P(c) \models \mathcal{C} \quad \wedge \quad \{\forall v \in U \cup Y.\, v = \langle t, \_ \rangle \wedge \forall c \in C.\, \texttt{ftime}(s_c^{(t)}) = t\} \quad P(c) \quad \{\forall v \in U \cup Y.\, v = \langle t + H, \_ \rangle \wedge \forall c \in C.\, \texttt{ftime}(s_c^{(t+H)}) = t + H\}$$

Meaning all preconditions are satisfied and all SUs and inputs have moved from time t to time $t + H$ through the execution of $P(c)$.

Using Definition 9 we conclude that Algorithm 3 is correct while the others are incorrect since they break one or more of the defined preconditions. Algorithm 1 and 2 violate the precondition of $\texttt{step}_b$ on line 2 by stepping it without having provided SU b with a value on the reactive input f. These definitions form the basis for describing the approach and implementation used to verify co-simulation algorithms in this work.

3 Related Work

The study of semantics and verification of co-simulation algorithms is presented in [5,9,13]. The paper [13] describes 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 subject to algebraic loops and adaptive steps. Thule et al. [22] studied how a co-simulation scenario's characteristics can be used to choose the correct simulation strategy for a given co-simulation algorithm. In [14], algorithms of complex scenarios are described, but this paper lacks the feature to verify the correctness of algorithms for complex scenarios.

Broman et al. describe in [5] an approach to achieve deterministic co-simulation results by placing constraints on the co-simulation scenario to avoid algebraic loops. They also propose a generic master algorithm for handling step negotiation. However, such generic algorithms do not consider other constraints on the SUs like reactive inputs or algebraic loops. This paper deals with all these constraints.

Formal methods have previously been successfully used in the area of co-simulation [1,6,23]. Amálio et al. [1] study how connections between simulation units can be formalized. They investigate how different formal tools can detect algebraic loops to obtain a deterministic co-simulation result. Cavalcanti et al. [6] claim to provide the first behavioral semantics of FMI. The paper shows how to prove essential properties of master algorithms, like termination and determinism. It also shows that the example provided in the FMI standard is not a valid algorithm. The paper [23] by Zeyda et al. formalizes models and proofs about co-simulation in Isabelle/UTP, illustrated by an industrial case study from the railway sector. However, their approach does not cover complex scenarios, unlike ours.

Nyman et al. in [16] examine how UPPAAL can analyze controller-based systems with FMUs and a master algorithm modeled in UPPAAL. UPPAAL was used to verify the properties of the controller used in the co-simulation. Palmieri et al. in [20] have used UPPAAL to provide sound guarantees on the interleaving between a graphical user interface and a generic FMI master algorithm. Our approach is more generic and relies on a parameterized template approach that can be applied to arbitrary co-simulation scenarios subject to both step negotiation and algebraic loops.

4 Verifying Complex Co-simulation Algorithms

This section describes our approach for verifying that a co-simulation algorithm respects the contracts of the scenario. The paper focuses on complex scenarios since the approach trivially covers simple scenarios. The section starts with an introduction of the UPPAAL-implementation. This is followed by some concrete examples of the approach using the scenarios in Figs. 2b and 8b.

4.1 Verifying an Algorithm Using UPPAAL

The approach is implemented in UPPAAL. A co-simulation is formalized as a collection of timed automata (TA) that formally describe a co-simulation as an orchestrator and some SUs. Two extra UPPAAL-templates are introduced to verify algorithms of complex scenarios by performing the search for a correct configuration. The structure of the UPPAAL model is fixed, however a translator (available online[1]) generates a unique model for each scenario and algorithm. The scenario and algorithm are expressed in a high-level domain-specific language similar to the algorithms in the paper. The tool can translate and verify all algorithms described by the grammar in Fig. 4. The tool checks both initialization procedures and co-simulation steps, but we only show the grammar of the co-simulation step due to space limitations.

⟨*cosim-step*⟩ ::= '`[`'⟨*cosim-action*⟩*'`]`'

⟨*SU-action*⟩ ::= ⟨*get : SU.Port*⟩ | ⟨*set: SU.Port*⟩ | ⟨*step: SU*⟩ | ⟨*restore-state: SU*⟩ | ⟨*save-state: SU*⟩

⟨*SU-step-action*⟩ ::= ⟨*SU-action*⟩ | '`{`' ⟨*algebraic*⟩ '`}`'

⟨*cosim-action*⟩ ::= ⟨*SU-action*⟩ | '`{`' ⟨*step-loop*⟩ '`}`' | '`{`' ⟨*algebraic*⟩ '`}`'

⟨*step-loop*⟩ ::= '`until-step-accept:`' '`[`'SU*'`]`'
'`iterate:`' '`[`'⟨*SU-step-action*⟩*'`]`'
'`if-retry-needed:`' '`[`'⟨*restore-state: SU*⟩*]'

⟨*algebraic*⟩ ::= '`until-converged:`' '`[`'SU.Port*'`]`'
'`iterate:`' '`[`'⟨*SU-action*⟩*'`]`'
'`if-retry-needed:`' '`[`'⟨*restore-state: SU*⟩*]'

Fig. 4. BNF Grammar for the specification of a co-simulation step

Notice, the tool does not allow nested fixed-point iteration procedures or step-negotiation inside a nested statement (step-loop or algebraic-loop). This is because the authors do not know any scenarios that such an algorithm should simulate. The following section explains the ideas behind the different UPPAAL-templates before describing how the verification of the algorithm is performed in UPPAAL.

The Interpreter orchestrates the behavior of the SUs by interpreting the supplied algorithm. It is responsible for executing the orchestration algorithm from *Instantiation* to *Termination*. A correct algorithm ensures that the Interpreter reaches the *Termination* state while an incorrect hits a deadlock. The Interpreter works deterministically by systematically picking an action of the algorithm and delegate it to one of the other automata using channel synchronization and shared variables.

[1] https://github.com/INTO-CPS-Association/Scenario-Verifier.

The preconditions described in Definitions 4 to 6 are encoded in UPPAAL as invariant functions of the states in the SU-template as shown in Fig. 5 by the function `preSet`, `preGet` and `preDostep`. The SU-template describes the interface (see Definition 1) and life-cycle of an SU. The values exchanged between the SUs are of the type $\mathcal{V}_{\mathcal{E}}$ (a status and a timestamp). Thus, the design contains enough information to check the contracts. However, nothing can be said about the numerical aspect of the co-simulation.

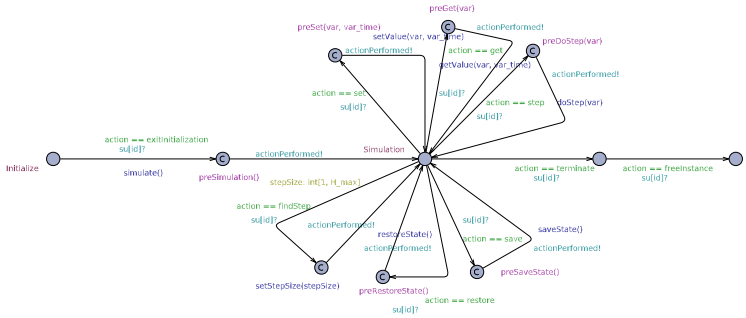


Fig. 5. Model of an SU in UPPAAL.

A violation of a guard or an invariant function results in a deadlock. A deadlock indicates that the algorithm either breaks a precondition or is not complete. A correct algorithm does not deadlock and reaches the state "Terminated", which means that all preconditions/contracts were satisfied and that the algorithm is complete. A complete algorithm correctly instantiates and informs all SUs that the simulation has ended. The automata in UPPAAL are designed, so all transitions are guarded except one identity transition in the trap state "Terminated" in the Interpreter-template. This ensures that all violations of the contracts result in a deadlock; furthermore, no deadlock can occur if the state "Terminated" is reached.

We define the two CTL-formulas: $A\Box$ not deadlock and $A\Diamond MasterA.Terminated$ for UPPAAL to verify. The first formula ensures that UPPAAL finds no deadlock. The second formula ensures that the Interpreter always reaches the state "Terminated", implying that the co-simulation is entirely executed and is not trapped into an infinite loop, which is not considered by the previous formula. Since the end time ($t + H$, where $H > 0$) of the simulation is greater than the start time (t), it can be concluded that at least one co-simulation step was successfully executed/checked. Moreover, since the contracts are constant over time, it can be concluded using induction that all future/other steps also satisfy these contracts. This is the essential argument for describing how the tool verifies that an algorithm is correct concerning Definition 9.

The method is clarified through a couple of examples. First, looking back at the incorrect Algorithms 1 and 2, we can see that they violate both of

the CTL-formulas since the broken precondition on line 2 is caught by the guard `preDostep` and results in a deadlock. Nevertheless, on the other hand, Algorithms 3 satisfies both CTL-formulas, and therefore also satisfies Definition 9.

4.2 Verifying Complex Simulation Scenarios in UPPAAL

Complex scenarios are, as previously described, simulated by an algorithm P that iteratively searches for a correct configuration, not violating any of the actions' preconditions. However, as described by Eq. (9) all unsuccessful search attempts violate some of the preconditions. Therefore, the implementation in UPPAAL tolerates such violations until a correct configuration is identified.

The approach is to temporarily turn off the preconditions for complex scenarios until a correct/convergent configuration is found (see Definition 8). Then, in between unsuccessful attempts, the co-simulation is backtracked, and a new search attempt is initiated where the configuration is updated as described in Sect. 2.2.

If the Interpreter finds a correct configuration, it backtracks the simulation, turns on the preconditions, and runs an extra iteration of the routine while checking the preconditions to ensure that the algorithm and configuration respect the contracts. To avoid state-space explosion and ensure termination, the number of search attempts is bounded. Thus, if the algorithm does not manage to find a correct configuration within the bound, the algorithm is considered incorrect. Since the simulation depends on the nature of the scenario, the next section is split to describe two different kinds of complex scenarios - step negotiation and algebraic loops.

Verifying a Step Negotiation Procedure. Step negotiation is a mechanism for the SUs to agree on the step size. The procedure iteratively searches for a step using a sequence of SU-actions in each iteration. The step size is shrunken between unsuccessful iterations. The step negotiation for the scenario in Fig. 2b is presented in Algorithm 4. UPPAAL verifies that a common step can be found using a given algorithm. Furthermore, it ensures that all preconditions of the actions are satisfied using the found step.

UPPAAL first tries to establish a proper step without considering the contracts. UPPAAL confirms this by transforming the supplied algorithm to a similar one that initially turns off the preconditions while it searches for a step. The transformation is shown from the original Algorithm 4 to the modified Algorithm 5 that UPPAAL checks. The preconditions are disabled to circumvent the model from deadlocking on unsuccessful search attempts. For example, if SU C is not capable of performing a step of the same size as SU D, this would break the preconditions of $\texttt{step}_D$ in line 6 in Algorithm 4. However, violations are tolerated until the SUs agree on a step due to the backtracking. If a correct step is identified, the preconditions are turned back on (see line 16) and an extra iteration of the procedure is performed to verify that all contracts are respected. If a correct step is not establish within N-tries, the algorithm is declared incorrect, and the verification aborts (line 20). UPPAAL non-deterministically picks a maximal positive

step for each SU in M to introduce the non-determinism of a step negotiation. We assume an SU accepts any step smaller or equal to its maximal step.

Algorithm 4 Step negotiation procedure of scenario in Fig. 2b.

```
1:  SaveSUs                                          ▷ Save the SUs
2:  while !Step_found do                             ▷ Step negotiation
3:      (s_D^(s+h_D), h_D) ← step_D(s_D^(s), h)
4:      g_v ← get_D(s_D^(s+h_D), y_g)
5:      s_C^(s) ← set_C(s_C^(s), u_G, G_v)
6:      (s_C^(s+h_C), h_C) ← step_C(s_C^(s), h_D)
7:      h ← min(h_C, h_D)                            ▷ Minimum step
8:      Step_found ← h == h_C ∧ h == h_D
9:      if !Step_found then                          ▷ Restore SUs
10:         RestoreSUs
11:     end if
12: end while
```

Algorithm 5 Modified Step negotiation procedure of scenario in Fig. 2b.

```
1:  SaveSUs                                          ▷ Save the SUs
2:  TurnPreconditionsOff()
3:  (I, isExtra) ← (0, false)
4:  while !Step_found do
5:      Line 3-8 from Algorithm 4
6:      if !Step_found then
7:          I ← I + 1
8:      else                                         ▷ Correct step found
9:          if !isExtra then                         ▷ Check contracts
10:             TurnPreconditionsOn()
11:             isExtra ← true
12:         else
13:             return(true)                         ▷ Correct algorithm
14:         end if
15:     end if
16:     if I == N then                               ▷ Max attempt
17:         return(false)                            ▷ Invalid algorithm
18:     end if
19:     RestoreSUs
20: end while
```

Verifying a Fixed-Point Procedure. The fixed-point procedure solves algebraic loops by finding fixed-points on the involved ports. Similar to step negotiation, a fixed-point procedure is an iterative search for a correct configuration. A fixed-point iteration procedure is shown in Algorithm 6.

The UPPAAL-model ensures the correctness of scenarios with algebraic loops that a fixed-point can be obtained using the algorithm without violating a single contract. However, the numerical aspect of finding a fixed-point is not included in our model since we restrict to a symbolic abstraction. However, the tool is still capable of checking that the contracts of the SUs are respected.

4.3 Debugging Algorithm Errors

When an algorithm is deemed incorrect by UPPAAL, the user needs to determine why. A model-checker (like UPPAAL) provides counter-examples. However, these are often hard to understand for a normal user. Therefore, a tool has been created to visualize the counter-example as an animation. The animation shows the violation found by UPPAAL. The animation shows which actions have been applied and which actions are enabled at any given time of the simulation. An example of the animation is shown in Fig. 8a.

5 Validation

The tool has been used to verify various algorithms and generate counter-examples for incorrect ones. The examples are publicly available[2]. The tested

[2] https://github.com/SimplisticCode/Co-simulation-Verifier.

scenarios include an industrial scenario from Boeing [11] of 4 SUs and 8 connections[3]. We have also tested it with what we believe to be the most common mistakes when implementing an algorithm to be confident that the tool catches these mistakes. The tests of the tool include abstract scenarios with multiple algebraic loops and more than 50 SUs and over 100 connections.

In this section, we show two examples: the first one, already introduced in Fig. 1, is designed to highlight the numerical errors that can happen when contracts are not satisfied, and the second one showcases an abstract scenario with all types of contracts described in the current manuscript.

5.1 Motivation Example

Our motivation example was introduced in Fig. 1, and its equations are displayed in Fig. 6. To highlight the numerical error caused by a mismatch in contracts, we conducted a series of experiments where the contracts were intentionally violated, i.e.; we applied the wrong algorithm to the contracts used. We studied this effect for multiple parameters' choices and selected two results representing the most common errors, illustrated in Fig. 7. Each plot shows the analytical solution, the correct co-simulation, and the incorrect co-simulation. Naturally, even the correct co-simulation will introduce errors. However, most of the results show an error caused solely by the contract mismatch, as shown on the left-hand side of Fig. 7. However, depending on the parameters and initial conditions of the experiment, some cases show that the mismatch is less important. This makes it very difficult to assess in practice whether some errors are caused by implementation mistakes or caused by the co-simulation discretization, and hence motivates our work. Some work has been carried out in [15] to characterize for which parameter values and initial conditions, the contracts are essential, but this is outside the scope of this paper.

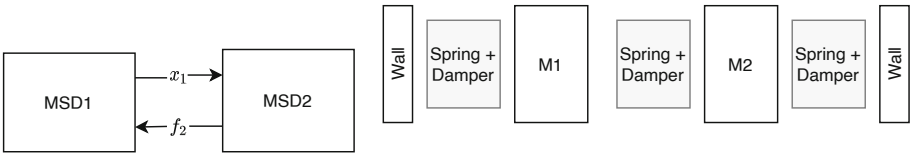


Fig. 6. Structure and equations of motivational example. Legend: x_i, v_i represent the position and velocity, respectively; f_i represents force, c_i, d_i represent stiffness and damping parameters, m_i represents mass.

[3] https://github.com/SimplisticCode/Co-simulation-Verifier/blob/master/Scenario/examples/industrial_casestudy.conf.

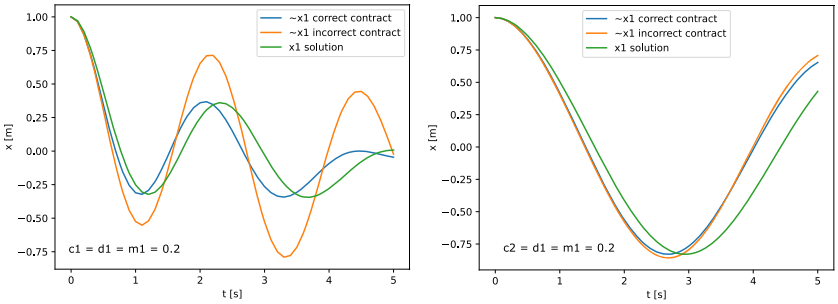
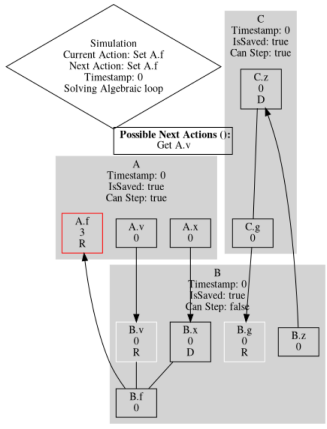


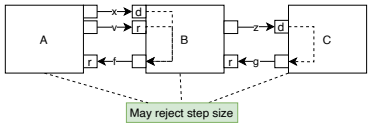
Fig. 7. Example results. Except when indicated in each plot, all parameters have the following default values: $m_i = c_i = 1$ and $d_i = 0$. The step used in the co-simulation is 0.1.

5.2 Complex Scenario

The scenario introduced here contains an algebraic loop between some SUs that may reject a step. The scenario is shown in Fig. 8b, and the algorithm of a valid co-simulation step is shown in Algorithm 6. The scenario is simulated using a



(a) Wrong algorithm of Fig. 8b highlighted by the animation.



(b) Case study scenario - Loop within Loop.

Algorithm 6 Co-simulation Step of Fixed-point Iteration inside Step finding procedure.

Save SUs ▷ Save all 3 SUs
$h \leftarrow H_{max}$
while !step_found **do** ▷ Step negotiation
  while !converged **do** ▷ FP procedure
    $s_A^{(s)} \leftarrow \texttt{set}_A(s_A^{(s)}, u_f, f_v)$
    $s_B^{(s)} \leftarrow \texttt{set}_B(s_B^{(s)}, [u_v, u_g], [v_v, g_v])$
    $(s_C^{(s+h_C)}, h_C), \leftarrow \texttt{step}_C(s_C^{(s)}, h)$
    $(s_B^{(s+h_B)}, h_B), \leftarrow \texttt{step}_B(s_B^{(s)}, h)$
    $(s_A^{(s+h_A)}, h_A), \leftarrow \texttt{step}_A(s_A^{(s)}, h)$
    $[v_a, x_v] \leftarrow \texttt{get}_A(s_A^{(s+h_A)}, [y_v, y_x])$
    $z_v \leftarrow \texttt{get}_B(s_B^{(s+h_B)}, y_z)$
    $s_C^{(s+h_C)} \leftarrow \texttt{set}_C(s_C^{(s+h_C)}, u_z, z_v)$
    $g_a \leftarrow \texttt{get}_C(s_C^{(s+h_C)}, y_g)$
    $s_B^{(s+h_B)} \leftarrow \texttt{set}_B(s_B^{(s+h_B)}, u_x, x_v)$
    $f_a \leftarrow \texttt{get}_B(s_B^{(s+h_B)}, y_F)$
    $conv \leftarrow CheckConv((g_a, v_a, f_a), (g_v, v_v, f_v))$
    if !$conv$ **then**
      Restore SUs ▷ Restore all 3 SUs
    end if
    $(g_v, v_v, f_v) \leftarrow (g_a, v_a, f_a)$
  end while
  $h \leftarrow min(h_A, h_B, h_C)$
  $Step\_found \leftarrow h == h_A \wedge h == h_B \wedge h == h_C$
  if !$Step\_found$ **then**
    Restore SUs ▷ Restore all 3 SUs
  end if
end while

Fig. 8. Advanced case study scenario (2b) and Algorithm (6). A counter-example is shown in 8a.

master algorithm consisting of a fixed-point iteration for solving the algebraic loop inside the step negotiation procedure like Algorithm 6.

Algorithm 6 is too complex to be analyzed with a simple visual inspection, showing the necessity of the UPPAAL tool created in this paper. The tool can analyze the algorithm in few seconds and has been used several times on intermediate versions of the algorithm to help the authors obtaining the correct version.

6 Concluding Remarks

This work proposed a model-checking approach to verify that an algorithm for an FMI-based co-simulation respects all the implementation contracts of the SUs. The contracts arose from previous work, which demonstrated that imposing them leads to better co-simulation results in the sense that the error introduced is caused only by the numerical discretization (as opposed to hard-to-debug contract mismatches, as Fig. 7 showed). In addition, the new approach can handle complex co-simulation scenarios containing both algebraic loops and step negotiation.

A tool generates a UPPAAL-model from a co-simulation scenario and an algorithm. The tool enables co-simulation practitioners to verify that their co-simulation algorithm is tailored to the scenario. Incorrect algorithms are presented using an animation of the simulation trace to clarify the problems. The approach inspires the work for synthesizing correct orchestration algorithms [14], and they together form the Scenario-Verifier.

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

References

1. Amálio, N., Payne, R., Cavalcanti, A., Woodcock, J.: Checking SysML models for co-simulation. In: Ogata, K., Lawford, M., Liu, S. (eds.) ICFEM 2016. LNCS, vol. 10009, pp. 450–465. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47846-3_28
2. Arnold, M., Clauß, C., Schierz, T.: Error analysis and error estimates for co-simulation in FMI for model exchange and co-simulation v2.0. In: Schöps, S., Bartel, A., Günther, M., ter Maten, E.J.W., Müller, P.C. (eds.) Progress in Differential-Algebraic Equations. DEF, pp. 107–125. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44926-4_6
3. Behrmann, G., et al.: UPPAAL 4.0. In: Third International Conference on Quantitative Evaluation of Systems (QEST 2006), pp. 125–126 (2006)
4. Blockwitz, T., et al.: 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, pp. 173–184. Linköping University Electronic Press (2012). https://doi.org/10.3384/ecp12076173

5. Broman, D., et al.: Determinate composition of FMUs for co-simulation. In: Eleventh ACM International Conference on Embedded Software. IEEE Press, Piscataway (2013). Article no. 2
6. Cavalcanti, A., Woodcock, J., Amálio, N.: Behavioural models for FMI co-simulations. In: Sampaio, A., Wang, F. (eds.) ICTAC 2016. LNCS, vol. 9965, pp. 255–273. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46750-4_15
7. FMI: Functional mock-up interface tools (2014). https://fmi-standard.org/tools/
8. Gomes, C., Broman, D., Vangheluwe, H., Thule, C., Larsen, P.G.: Co-simulation: a survey. ACM Comput. Surv. **51**(3), 49–49 (2018)
9. Gomes, C., Lucio, L., Vangheluwe, H.: 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), pp. 784–789. IEEE (2019)
10. Gomes, C., et al.: Semantic adaptation for FMI co-simulation with hierarchical simulators. SIMULATION **95**(3), 241–269 (2019)
11. Gomes, C., et al.: HintCO - hint-based configuration of co-simulations. In: Proceedings of the 9th International Conference on Simulation and Modeling Methodologies, Technologies and Applications, pp. 57–68. Scitepress - Science and Technology Publications (2019)
12. Gomes, C., Thule, C., Lausdahl, K., Larsen, P.G., Vangheluwe, H.: Demo: stabilization technique in INTO-CPS. In: Mazzara, M., Ober, I., Salaün, G. (eds.) STAF 2018. LNCS, vol. 11176, pp. 45–51. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-04771-9_4
13. Gomes, C., Thule, C., Lúcio, L., Vangheluwe, H., Larsen, P.G.: Generation of co-simulation algorithms subject to simulator contracts. In: Camara, J., Steffen, M. (eds.) SEFM 2019. LNCS, vol. 12226, pp. 34–49. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-57506-9_4
14. Hansen, S.T., Gomes, C., van de Pol, J., Larsen, P.G.: Synthesizing co-simulation algorithms with step negotiation and algebraic loop handling (2021, to appear)
15. Inci, E.O., et al.: The effect and selection of solution sequence in co-simulation. In: The Annual Modeling and Simulation Conference, Virginia, USA (2021, to appear)
16. Jensen, P.G., Larsen, K.G., Legay, A., Nyman, U.: Integrating tools: co-simulation in UPPAAL using FMI-FMU. In: 2017 22nd International Conference on Engineering of Complex Computer Systems (ICECCS), pp. 11–19. IEEE (2017)
17. Kübler, R., Schiehlen, W.: Two methods of simulator coupling. Math. Comput. Model. Dyn. Syst. **6**(2), 93–113 (2000)
18. Lee, E.A.: Cyber physical systems: design challenges. In: 2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC), pp. 363–369 (2008)
19. Oakes, B.J., Gomes, C., Holzinger, F.R., Benedikt, M., Denil, J., Vangheluwe, H.: Hint-based configuration of co-simulations with algebraic loops. In: Obaidat, M.S., Ören, T., Szczerbicka, H. (eds.) SIMULTECH 2019. AISC, vol. 1260, pp. 1–28. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-55867-3_1
20. Palmieri, M., Bernardeschi, C., Masci, P.: A framework for FMI-based co-simulation of human-machine interfaces. Softw. Syst. Model. **19**(3), 601–623 (2020). https://doi.org/10.1007/s10270-019-00754-9
21. Schweizer, B., Li, P., Lu, D.: Explicit and implicit cosimulation methods: stability and convergence analysis for different solver coupling approaches. J. Comput. Nonlinear Dyn. **10**(5), 051007 (2015)

22. Thule, C., Gomes, C., Deantoni, J., Larsen, P.G., Brauer, J., Vangheluwe, H.: Towards the verification of hybrid co-simulation algorithms. In: Mazzara, M., Ober, I., Salaün, G. (eds.) STAF 2018. LNCS, vol. 11176, pp. 5–20. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-04771-9_1
23. Zeyda, F., Ouy, J., Foster, S., Cavalcanti, A.: Formalising cosimulation models. In: Cerone, A., Roveri, M. (eds.) SEFM 2017. LNCS, vol. 10729, pp. 453–468. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-74781-1_31