International Journal on Software Tools for Technology Transfer (2022) 24:999–1024
https://doi.org/10.1007/s10009-022-00686-8

GENERAL

Special Issue: FMICS 2021

Verification and synthesis of co-simulation algorithms subject to algebraic loops and adaptive steps

Simon Thrane Hansen[1] · Casper Thule[1] · Cláudio Gomes[1] · Jaco van de Pol[2] · Maurizio Palmieri[3] · Emin Oguz Inci[5] · Frederik Madsen[1] · Jesús Alfonso[4] · José Ángel Castellanos[4] · José Manuel Rodriguez[4]

Accepted: 19 October 2022 / Published online: 12 November 2022

Abstract
Simulation-based analyses are becoming increasingly vital for the development of cyber-physical systems. Co-simulation is one such technique, enabling the coupling of specialized simulation tools through an orchestration algorithm. The orchestrator describes how to coordinate the simulation of multiple simulation tools. The simulation result depends on the orchestration algorithm that must stabilize algebraic loops, choose the simulation resolution, and adhere to each simulation tool's implementation. This paper describes how to verify that an orchestration algorithm respects all contracts related to the simulation tool's implementation and how to synthesize such tailored orchestration algorithms. The approaches work for complex and adaptive co-simulation scenarios and have been applied to several real-world case studies.

Keywords Co-simulation · Model-checking · Cyber-physical systems · Formal methods

1 Introduction

Cyber-physical systems (CPSs) are omnipresent and part of the critical infrastructure. A CPS is a hybrid system that embodies physical processes controlled by digital devices. CPSs are becoming increasingly complex and critical [1], which leads to the desire for techniques to assist in their development. Traditional modeling and simulation techniques, where one algorithm describes an entire system, are no longer sufficient to cope with the integrated development processes of such systems [2], which consist of heterogeneous subsystems typically developed using different tools and formalisms. *Co-simulation* is a technique enabling the simulation of a complex CPS consisting of multiple black-box simulation units (SUs), where each SU represents a subsystem and can compute the behavior of that system [3,4]. Co-simulation allows iterative integration of constituents to explore the global system behavior as a discrete trace without violating the constituents' intellectual property (IP) during the entire development cycle.

An example of an SU is a Functional Mock-up Unit (FMU) defined by the Functional Mock-up Interface Standard [2] (FMI), which inspires the notion of an SU in this paper. FMI is a widely adopted standard used commercially and supported by many tools [5].

An SU interacts with its environment through input and output ports. A set of SUs can be composed into a *scenario* by *coupling* the input ports to output ports. The syntax in Fig. 1 is used to graphically present co-simulation scenarios.

A *coupling* connects one input port of an SU to an output port of another SU. The *coupling restriction* states that the value of an input and an output of a coupling must be the same at all times. However, in reality, the coupling restrictions can only be satisfied at specific points in time called *communication points*. Therefore, each SU makes its own assumptions about the evolution of its input values between

✉ Simon Thrane Hansen
sth@ece.au.dk

1 DIGIT, Department of Electrical and Computer Engineering, Aarhus University, Åbogade 34, 8000 Aarhus, Denmark

2 DIGIT, Department of Computer Science, Aarhus University, Åbogade 34, 8000 Aarhus, Denmark

3 DII, Department of Information Engineering, University of Pisa, Pisa, Italy

4 Department of Information Engineering, Instituto Tecnologico de Aragon, Zaragoza, Spain

5 Department of Mechanical Engineering, KU Leuven, Celestijnenlaan 300, 3001 Leuven, Belgium

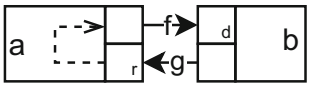


Fig. 1 A co-simulation scenario with two SUs a and b. The dashed arrows denote feed-through connections, the ports are represented as small squares, the instrumentation of an input port is denoted by the letters r (reactive) or d (delayed). The solid arrows f and g represent couplings

the communication points, which can introduce errors in the co-simulation [6].

The orchestrator is an algorithm interacting with the SUs through their interfaces. The orchestrator computes the behavior of a scenario as a discrete trace while it tries to satisfy the coupling restrictions by exchanging values between the coupled ports. The orchestrator aims to find the communication points that minimize the co-simulation error while ensuring that the SUs move in lockstep. The optimal communication points depend on the SUs [7–11]. An orchestrator is in practice often referred to as an *orchestration engine*.

1.1 Problem definition

The overarching challenge of co-simulation is ensuring accurate simulation results. The challenge is a consequence of the nature of co-simulation being a black-box simulation technique containing many potential error sources. The simulation result depends not only on the correctness of the individual SUs but also on their orchestration [7,9,10]. For example, if a mistake is made while coding the orchestration algorithm, it is challenging and costly to identify the source of the bug from the inaccurate simulation result. The error could have been introduced by one of the SUs, the orchestration algorithm, or the chosen step duration (simulation resolution). This is a significant challenge for co-simulation practitioners typically domain experts such as electrical, mechanical, and software engineers are not trained in spotting such algorithmic errors. Furthermore, the FMI Standard does not restrict or provide well-defined semantics for the orchestration algorithm used to simulate a given scenario [12].

The scenario in Fig. 1 can, for example, without the contributions described in our work be simulated using Algorithms 1 and 2 in Fig. 2, that both conform to the FMI Standard. Even though the algorithms consist of the same actions, they can lead to two utterly different simulation results, as shown in Fig. 3. The only difference between the two algorithms is their communication points. Algorithm 1 exchanges values between the ports u_g and y_g and u_f and y_f after SU B has been stepped, but before stepping SU A. Algorithm 2 exchanges values between the ports u_g and y_g and u_f and y_f after stepping both SUs.

Obtaining an accurate co-simulation result requires an algorithm tailored explicitly to the scenario [7–10]. Nevertheless, it is challenging to generate such tailored algorithms based on the black-boxes/interfaces described by the FMI standard.

The problem was addressed for a specific class of co-simulation scenarios by adding *contracts* to the inputs of an SU [7,8]. The developer of an SU can specify non-confidential information about an SU's input approximation functions using contracts. The orchestrator uses the contracts to determine the communication points that minimize the co-simulation error [7,8,13] when generating a tailored algorithm. Tailored algorithms can, in some cases, substantially reduce the co-simulation error, as shown in Fig. 3. The plots in Fig. 3 show two different simulations of the same scenario with two different algorithms. The simulation results are compared with an analytical solution.

Previous techniques for synthesizing tailored algorithms [7,8] are limited to a particular class of co-simulation scenarios, which are not subject to either algebraic loops or adaptive steps: the so-called *complex scenarios*. Complex scenarios are complicated to simulate because the orchestrator needs to *adapt* to the observed behavior to obtain an accurate simulation result.

Complex scenarios are simulated using an iterative algorithm [13]. The iterative algorithm solves algebraic loops (cyclic dependencies between the SUs) and ensures that all SUs agree on a step duration (step negotiation). Step negotiation permits the SUs to implement error estimation and refuse specific future state evaluations to minimize the simulation error while ensuring that the SUs move in lockstep. None of the existing techniques for synthesizing orchestration algorithms supports adaptive scenarios where the most accurate simulation results are achieved by changing the contracts of the SUs during the simulation.

The two key problems we address in this paper (in addition to the formalization of co-simulation and co-simulation algorithms) are:

- Given an orchestration algorithm P and a scenario $\mathcal{S}$: Determine if P is the correct orchestration algorithm to simulate $\mathcal{S}$ without simulating the system.
- Given a scenario $\mathcal{S}$: Synthesize a tailored orchestration algorithm P for $\mathcal{S}$, where $\mathcal{S}$ can be an arbitrary adaptive scenario.

1.2 Contribution

The paper is an extended version of two previous papers [13, 14]. The paper in [14] describes how to verify orchestration algorithms, while an approach for synthesizing implementation-aware orchestration algorithms is presented

Algorithm 1

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: $(s_A^{(H)}, H) \leftarrow \texttt{step}_A(s_A^{(0)}, H)$

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: $s_B^{(H)} \leftarrow \texttt{set}_B(s_B^{(H)}, u_f, f_v)$

Fig. 2 Two algorithms conforming to the FMI Standard (version 2.0) of the scenario in Fig. 1

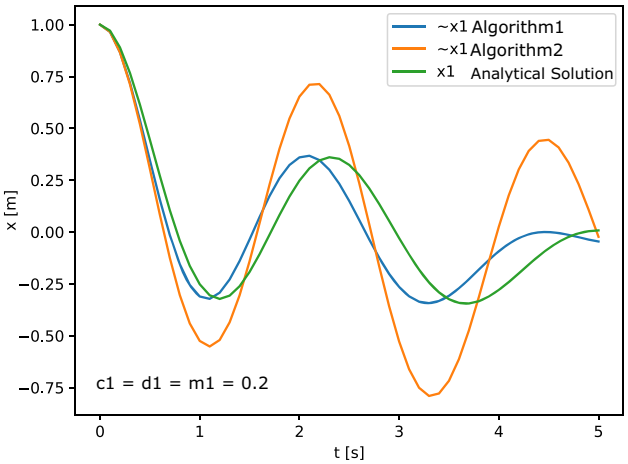


Fig. 3 Simulation results from Algorithms 1 and 2 compared with an analytical solution. The step size used in the co-simulation was 0.1 s

in [13]. This manuscript extends its predecessors by covering adaptive scenarios and presenting verification and synthesis of orchestration algorithms in a detailed, self-contained, and integrated setting.

All of the techniques from the paper have been implemented in a tool called the *Scenario-Verifier*. The Scenario-Verifier enables co-simulation practitioners to verify that their algorithm respects the contracts of the SUs and synthesize tailored orchestration algorithms for a given scenario subject to step negotiation, algebraic loops and dynamic contracts. Our techniques cannot account for the numerical aspect of the co-simulations, which is an inherent problem for all general analytical techniques of co-simulation.

Another novel contribution is the integration of the Scenario-Verifier with the orchestration engine Maestro 2 [15], which we believe to be one of the first connections between a formal model and an orchestration engine.

The integration enables co-simulation practitioners to simulate their co-simulation scenarios using a correct co-simulation algorithm seemlessly.

The benefits of the integration with Maestro 2 have been explored using two real-world case studies in Sect. 7. The case studies show how formal methods can add value to the co-simulation toolchain by letting the co-simulation practitioners focus on the individual subsystems instead of their orchestration.

1.3 Structure

Section 2 gives some necessary background on model checking and co-simulation. Section 3 to 5 gives an overview of different types of co-simulation scenarios simple, complex and adaptive, respectively. Each section concentrates on a specific class of co-simulation scenarios and describes the approach for verifying the orchestration algorithm of such scenarios in UPPAAL. The sections extend each other, so the length of the section is inversely proportional to the section number. Section 6 describes the synthesis of an orchestration algorithm for all of the described scenario classes. Section 7 introduces two case studies where the approaches have been utilized. Section 8 discusses related work, Sect. 9 gives some concluding remarks.

2 Preliminaries

This section shortly describes the preliminaries of the work. We refer interested readers to the cited material for more information.

2.1 The FMI standard

The FMI standard [2] describes how to implement composable SUs. The SUs can be created using different tools and techniques and be composed into a scenario to explore their global behavior as the result of a co-simulation.

This work focuses on co-simulations of scenarios composed of SUs described by Definition 1, which is based on [16,17].

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 is a set, denoting the state space of the SU.
- U_c and Y are sets, of input and output ports, respectively. The union $\texttt{VAR}_c = U_c \cup Y_c$ of the inputs and outputs is called the ports of the SU.
- $\mathcal{V}$ is a set, intuitively denoting the values that a variable can hold. $\mathcal{V}_\mathcal{T} = \mathbb{R}_{\geq 0} \times \mathcal{V}$ is the set of timestamped values exchanged between input and output ports.
- The functions $\texttt{set}_c : S_c \times U_c \times \mathcal{V}_\mathcal{T} \to S$ and $\texttt{get}_c : S_c \times Y_c \to \mathcal{V}_\mathcal{T}$ sets an input and gets an output, respectively.
- $\texttt{step}_c : S_c \times \mathbb{R}_{>0} \to S_c \times \mathbb{R}_{>0}$ is a function; it instructs the SU to compute its state after a given duration. If an SU is in state $s^{(t)}$ at time t then, $(s_c^{(t+h)}, h) = \texttt{step}(s_c^{(t)}, H)$ denotes the state $s_c^{(t+h)}$ of the SU at time $t + h$, where $h \leq H$.

The state of SU c at time t is denoted $s_c^{(t)}$. The function $\texttt{step}_c$ returns a step duration ($h \in \mathbb{R}_{>0}$) 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.

The FMI standard describes some state-changing functions that are not included in Definition 1. An example is the *Terminate*-function that tells an SU the simulation has ended. We do not treat them in this paper; nevertheless, they are still being enforced by our formal model. A collection of coupled SUs forms a scenario:

Definition 2 (*Scenario*) A *scenario* $\mathcal{S}$ is a tuple

$$\mathcal{S} \triangleq \langle C, L, M, R, F \rangle ,$$

where

- C is a finite set (of SU identifiers).
- L is a function $L : U \to Y$, where $U = \bigcup_{c \in C} U_c$ and $Y = \bigcup_{c \in C} Y_c$, and where $L(u) = y$ means that the output y is coupled to the input u.
- $M \subseteq C$ denotes the SUs that implement error estimation.
- $R : U \to \mathbb{B}$ is a predicate, which describes the SUs' input approximation functions. $R(u) = \texttt{true}$ means that the function $\texttt{step}$ assumes that the timestamp t_c of the SU c of the u is smaller than the timestamp t_v of the timestamped value $\langle t_v, v \rangle$ set on the input u. We call an input port u *reactive* if $R(u)$, and *delayed* otherwise.
- F is a family of functions $\{F_c : Y_c \to \mathcal{P}(U_c)\}_{c \in C}$. The statement $u_c \in F_c(y_c)$ says that the input u_c *feeds through* to the output y_c of the same SU. It means that there exists $v_1, v_2 \in \mathcal{V}_\mathcal{T}$ and $s_c \in S_c$, such that $\texttt{get}_c(\texttt{set}_c(s_c, u_c, v_1), y_c) \neq \texttt{get}_c(\texttt{set}_c(s_c, u_c, v_2), y_c)$.

Definition 2 extends the FMI 2.0 standard for co-simulation [2] with feed-through and reactivity constraints to cover a broader class of co-simulation scenarios.

2.2 Notation and abbreviations

Most of the notation used in this paper is based on the notation from [3,4]. We use the convention that capital letters denote a set and lowercase letters denote a member of a set. For example, U_c denotes the input ports of the SU c where $u \in U_c$. A table of the most common abbreviations is found in Sect. A on page 40

2.3 Model checking

Model checking [18,19] is a technique for automated verification of complex reactive systems such as hardware components, embedded controllers, and network protocols. The technique works by expressing the specification of a system using logical formulas.

The model of this paper is developed using the tool *UPPAAL*. UPPAAL is a model checker developed by Uppsala University and Aalborg University [20]. It is based on the theory of timed automata [21]. Interested readers are referred to [20,22] for more details on UPPAAL. Every UPPAAL model is characterized by:

- a finite set of *locations* with one *initial* location;
- a finite set of *transitions* connecting locations;
- a finite set of *variables*;
- a finite set of *actions* performed on the variables when executing a transition;
- a finite set of predicates, called *guards* allowing the execution of a transition;
- a finite set of *synchronization channels*.

A sequence of transition executions represents the evolution of the model. Each transition can be labeled with one or more actions, a guard, and a synchronization. The *state* of a model is given by the current location and the current values of the variables.

It is possible to combine more models to build a complex one. The composition between two models is modeled by the existence of two transitions, one for each of the two models, labeled with the same synchronization channel. Two transitions labeled with the same synchronization channel can be executed only if the guards on each of them, if any, are

satisfied. The following additional features of UPPAAL are used:

- *Committed locations* that can be used to force no delay in a location, i.e., the next transition must change the current location;
- *Location invariants*, i.e., conditions that must hold while the model is in a specific location.

Graphically, an input action is denoted by a question mark (*?*), and an output action is by an exclamation mark (*!*), an example is presented in Fig. 5 on page 15.

Uppaal is chosen due to its graphical user interface, simplicity in the model creation, and powerful simulator/debugger. The notion of time is not utilized in this work, meaning that the work can be replicated in other model checkers.

3 Simple co-simulation scenarios

This section introduces simple co-simulation scenarios and orchestration algorithms. A co-simulation scenario is simple if it is not complex (see Sect. 4) or adaptive (see Sect. 5). This means that a simple scenario is not subject to either step negotiation, algebraic loops, or changing coupling and reactivity constraints. The scenario in Fig. 1 is simple.

Section 3.1 introduces co-simulation algorithms and defines the SU actions in the abstract state space. Section 3.2 follows with a formal definition of a correct orchestration algorithm. Section 3.3 presents the UPPAAL model to verify co-simulation algorithms in a fully automated fashion.

3.1 Orchestration algorithms for simple scenarios

A co-simulation scenario is orchestrated by an orchestration engine, executing an orchestration algorithm, which describes when future states are calculated and how values are exchanged between the SUs. An orchestration algorithm consists of an initialization procedure and a co-simulation step procedure. The initialization and co-simulation step procedures are both a sequence of the SU operations (`set`, `get`, and `step`).

The initialization procedure sets up the system such that it is ready for simulation. The initialization procedure initializes the scenario by setting all parameters of the SUs and calculating an initial value for all ports and SUs [23].

The co-simulation step simulates the scenario; it does so by moving all SUs from an initial state with the timestamp t to a future state with the timestamp $t + H$. All coupling restrictions must be satisfied at the initial and final state.

An orchestrator runs a co-simulation by iteratively applying the co-simulation step until the simulation reaches the end time where the orchestrator terminates the simulation. Therefore, a co-simulation algorithm is entirely determined by its initialization and co-simulation step procedure.

This work concentrates on the co-simulation step, which we refer to as *the algorithm* in the paper. The initialization can be derived using the presented method.

In order to formally define a co-simulation algorithm, we introduce the abstract co-simulation state that the algorithm operates on. An example of an abstract co-simulation state is shown in Example 1.

Definition 3 *(Abstract SU State)* The observable abstract state s^R of an SU c in a scenario $\mathcal{S}$ is an element of the set $S_c^R = \mathbb{R}_{\geq 0} \times S_{U_c}^R \times S_{Y_c}^R \times S_{\mathcal{V}_c}^R$, where:

- $S_{U_c}^R : U_c \to \mathbb{R}_{\geq 0}$ is a function mapping each input port to a timestamp.
- $S_{Y_c}^R : Y_c \to \mathbb{R}_{\geq 0}$ is a function mapping each output port to a timestamp.
- $S_{\mathcal{V}_c}^R : \mathtt{VAR}_c \to \mathcal{V}$ is a function mapping each port to a value.

The first component of the abstract state denotes the current time of the SU.

We use the abstract state s_c^R of an SU c instead of the internal state s_c since the latter is non-observable. The orchestrator can only observe the SU as a black-box via its interface, which means that the numerical behavior of the SU cannot be predicted by the orchestrator. The state of a co-simulation scenario is the combination of the states of its subcomponents:

Definition 4 *(Abstract Co-simulation State)* The abstract co-simulation state $s_{\mathcal{S}}^R$ of a scenario $\mathcal{S} = \langle C, L, M, R, F \rangle$ is an element of the set $S_{\mathcal{S}}^R = \mathit{time} \times S_U^R \times S_Y^R \times S_V^R$ where:

- $\mathit{time} : C \to \mathbb{R}_{\geq 0}$ is a function, where $\mathit{time}(c)$ denotes the current simulation time of SU c. We denote by a time value $t \in \mathbb{R}_{\geq 0}$ the function $\lambda c.t$, which we use if all SUs are at the same time.
- $S_U^R = \prod_{c \in C} S_{U_c}^R$ maps all inputs of the scenario to a timestamp.
- $S_Y^R = \prod_{c \in C} S_{Y_c}^R$ maps all outputs of the scenario to a timestamp.
- $S_{\mathcal{V}}^R = \prod_{c \in C} S_{\mathcal{V}_c}^R$ maps all ports of the scenario to a value.

Example 1 The initial co-simulation state of the scenario in Fig. 1 is: $s_B^R \times s_A^R$, where $s_A^R = \langle 0, \{u_g \to 0\}, \{y_f \to 0\}, \_ \rangle$, and $s_B^R = \langle 0, \{u_f \to 0\}\{y_g \to 0\}, \_ \rangle$. The valuation of the ports is not described (indicated by _) since we cannot say anything concrete about these values.

An algorithm P operates on the abstract co-simulation state s_S^R. A co-simulation step P is a sequence of operations that takes a co-simulation from one consistent state to another consistent state. We write $s \xrightarrow{P} s'$ if executing the co-simulation step P from the initial state s results in the final state s'.

Definition 5 *(Co-simulation Step)* A *co-simulation step* P is a sequence of SU actions. A *co-simulation step* P is consistent if it takes a consistent co-simulation state to another consistent co-simulation state. The state of the co-simulation is consistent if all input ports have a source, and all coupled ports have the same value. Formally:

$$\begin{aligned}&\left\langle t, s_U^R, s_Y^R, s_{\mathcal{V}}^R\right\rangle \xrightarrow{P} \left\langle t', s_U^{R'}, s_Y^{R'}, s_{\mathcal{V}}^{R'}\right\rangle \\ &\quad \Longrightarrow \left(\texttt{consistent}(\left\langle t, s_U^R, s_Y^R, s_{\mathcal{V}}^R\right\rangle\right) \\ &\quad \Longrightarrow \left(\texttt{consistent}\left(\left\langle t', s_U^{R'}, s_Y^{R'}, s_{\mathcal{V}}^{R'}\right\rangle\right) \wedge t' > t)\right)\end{aligned}$$

where `consistent` is defined as:

$$\begin{aligned}&\texttt{consistent}(\left\langle t, s_U^R, s_Y^R, s_{\mathcal{V}}^R\right\rangle) \\ &\quad \triangleq (\forall u \in U\, \exists y \in Y \cdot L(u) = y) \\ &\quad \wedge (\forall u, y \cdot L(u) = y \implies s_{\mathcal{V}}^R(u) = s_{\mathcal{V}}^R(y))\end{aligned}$$

The plots in Fig. 3 show why we need a tailored algorithm to simulate a scenario. Although the algorithms Algorithms 1 and 2 lead to different simulation results they both satisfy Definition 5. To differentiate between them, we need to consider the semantics of the SU actions.

An implementation-aware algorithm is one of the preconditions for obtaining accurate simulation results, which we consider to be the ultimate objective of the orchestration algorithm. An algorithm is tailored to the scenario if it respects Definition 2 and the SU semantics described below.

Definition 6 *(Get Action)* Obtaining a value from an output port y of an SU at time t using the action, $\texttt{get}(s^{(t)}, y)$ changes the state of the SU according to:

$$\begin{aligned}&s^R \xrightarrow{\texttt{get}(s^{(t)}, y)} (v, s^{R'}) \implies \texttt{preGet}(y, s^R) \\ &\quad \wedge \texttt{postGet}(y, s^R, s^{R'}, v)\end{aligned}$$

Where:

$$\begin{aligned}&\texttt{preGet}\left(y, \left\langle t, s_U^R, s_Y^R, s_{\mathcal{V}}^R\right\rangle\right) \\ &\quad \triangleq s_Y^R(y) < t \wedge \forall u \in F(y) \cdot s_U^R(u) = t\end{aligned}$$

The precondition (above) states that no value must have been obtained from the output y since the SU was stepped, formally described as $s_Y^R(y) < t$. Furthermore, it requires that all the inputs that feed through to y have been updated, so they are at time t. The postcondition (below) ensures that the output is advanced to time t and that we have obtained the value of the output.

$$\begin{aligned}&\texttt{postGet}(y, \left\langle t, s_U^R, s_Y^R, s_{\mathcal{V}}^R\right\rangle, \left\langle t, s_U^R, s_Y^{R'}, s_{\mathcal{V}}^R\right\rangle, \langle t_v, x\rangle) \\ &\quad \triangleq s_Y^{R'}(y) = t \\ &\quad \wedge \forall y_m \in (Y \setminus y) \cdot s_Y^{R'}(y_m) = s_Y^R(y_m) \\ &\quad \wedge t_v = t \wedge s_{\mathcal{V}}^R(y) = x\end{aligned}$$

The $\texttt{get}(s^{(t)}, y)$ action also gives us a value $\langle t_v, x\rangle$ that we can set on an input port using the set action described next in Definition 7.

Definition 7 *(Set Action)* Setting a value $\langle t_v, x\rangle$ on the input port u of an SU using $\texttt{set}(s^{(t)}, u, \langle t_v, x\rangle)$ updates the time and value of the input port u such that it matches $\langle t_v, x\rangle$, formally:

$$\begin{aligned}&s^R \xrightarrow{\texttt{set}(s^{(t)}, u, v)} s^{R'} \implies \texttt{preSet}(u, v, s^R) \\ &\quad \wedge \texttt{postSet}(u, v, s^R, s^{R'})\end{aligned}$$

where:

$$\begin{aligned}&\texttt{preSet}\left(u, \langle t_v, x\rangle, \left\langle t, s_U^R, s_Y^R, s_{\mathcal{V}}^R\right\rangle\right) \triangleq s_U^R(u) < t_v \\ &\quad \wedge ((R(u) \wedge t_v > t) \vee (\neg R(u) \wedge t_v = t))\end{aligned}$$

The precondition says that the input must not have been assigned a new value since the SU was stepped, formally $s_U^R(u) < t_v$. Furthermore, it ensures that reactive inputs $(R(u))$ are set with a value with a timestamp larger than timestamp of the SU. Delayed inputs $(\neg R(u))$ should be set with a value with the same timestamp as the SU. The postcondition (below) ensures the value and time of the input u is updated so it matches the value assigned on the input.

$$\begin{aligned}&\texttt{postSet}(u, \langle t_v, x\rangle, \left\langle t, s_U^R, s_Y^R, s_{\mathcal{V}}^R\right\rangle, \left\langle t, s_U^{R'} s_Y^R, s_{\mathcal{V}}^{R'}\right\rangle) \triangleq t_v \\ &\quad = s_U^{R'}(u) \wedge (\forall u_m \in (U \setminus u) \cdot s_U^{R'}(u) \\ &\quad = s_U^R(u)) \wedge s_{\mathcal{V}}^{R'}(u) = x \wedge t_v = t\end{aligned}$$

The precondition `preSet` is not needed and could be caught by the precondition `preStep`. However, experience shows that incorrect algorithms are easier to correct if we detect the violation of an input contract once we try to set the value.

Definition 8 *(Step Computation)* Stepping an SU using $\texttt{step}(s^{(t)}, H)$ advances the state of the SU by $H \in \mathbb{R}_{>0}$,

formally:

$$s^R \xrightarrow{\texttt{step}(s^{(t)},H)} s^{R'} \implies \texttt{preStep}(H, s^R) \\ \wedge \texttt{postStep}(H, s^R, s^{R'})$$

where:

$$\texttt{preStep}\left(H, \left\langle t, s_U^R, s_Y^R, s_{\mathcal{V}}^R \right\rangle\right) \\ \triangleq \forall u \in U \cdot ((R(u) \wedge t + H = s_U^R(u)) \\ \vee (\neg R(u) \wedge t = s_U^R(u)))$$

The precondition (above) states that all the SU's inputs have been updated according to their reactivity constraints. All reactive inputs are at time $t + H$ and all delayed inputs are at time t. The postcondition ensures that the time of the SU advances by the step duration H and that new values have been calculated on the ports.

$$\texttt{postStep}\left(H, \left\langle t, s_U^R, s_Y^R, s_{\mathcal{V}}^R \right\rangle, \left\langle t', s_U^R, s_Y^R, s_V^{R'} \right\rangle\right) \\ \triangleq t + H = t'$$

Note that we, for now, assume that an SU will accept all step durations. Step rejection is described in Sect. 4 on page 16.

3.2 Correct orchestration algorithms

An algorithm P must satisfy Definition 5 while respecting the semantics. We use Dijkstra's weakest precondition calculus [24] to check if all actions of P are enabled. The weakest precondition calculus takes a predicate defining the final state and a program that must terminate in a state satisfying the predicate. The weakest precondition calculus uses backward reasoning and the semantics of the operations to calculate the weakest predicate/precondition that defines the initial state, such that all executions of the program from a state defined by the precondition results in a state defined by the postcondition.

We use the weakest precondition calculus to define that an algorithm P respects the semantics of Definitions 6 to 8 and Definition 5:

Definition 9 *(Consistent Co-simulation Step)* A co-simulation step P is consistent/correct if it takes an initial consistent co-simulation state at time t to a future consistent co-simulation state at time $t + H$ while respecting the SU semantic.

$$\texttt{consistent}\left\langle t, s_U^R, s_Y^R, s_{\mathcal{V}}^R \right\rangle) \implies wp(P, \texttt{consistent} \\ \left\langle t + H, s_U^{R'}, s_Y^{R'}, s_{\mathcal{V}}^{R'} \right\rangle) \wedge H > 0$$

An algorithm P is tailored to the scenario if it respects the semantics of Definitions 6 to 8 and Definition 5.

Example 2 For example, Algorithm 1 is a tailored algorithm. On the other hand, Algorithm 2 is not a tailored algorithm since it violates Definition 9. The $\texttt{step}_A$ operation on line 2 is not enabled since the precondition $\texttt{preStep}_A$ is not respected by the state $s_A^R = \langle 0, \{u_g \to 0\}, \{y_f \to 0\}, \_ \rangle$ that does not contain $\{u_g \to H\}$. Intuitively, we step SU A without having provided it with a value on the reactive input u_g; this is an apparent violation of the precondition $\texttt{preStep}_A$.

Definition 9 says that a correct algorithm transforms the scenario from one consistent state to another consistent state, where the only difference between the initial and final state is the timestamp of the state.

This enables inductive reasoning to show that if an algorithm P satisfies Definition 9, then we can assume that all future executions of P from a consistent will reach a consistent final state. We use UPPAAL to establish the inductive step. We start from a consistent state and use UPPAAL to interpret the algorithm P to see if the execution of P reaches a final state. The base case is to show that the algorithm can establish a consistent. The proof can be found in [4]. Inductive reasoning allows us to verify infinite co-simulations in UPPAAL.

3.3 Verification of orchestration algorithms in UPPAAL

We use UPPAAL to prove that a given orchestration algorithm P for the scenario $\mathcal{S}$ respects Definition 9. Before defining the UPPAAL model, we briefly describe the overall idea of using UPPAAL to verify orchestration algorithms. The UPPAAL model is a part of a tool-chain called the *Scenario-Verifier* that allows co-simulation practitioners to verify an orchestration algorithm of a scenario described in the domain-specific language (DSL) defined by the grammar in Sect. B.

The UPPAAL model is automatically instantiated from the provided scenario and algorithm. All scenarios described in the grammar (including those in the paper) can be automatically instantiated and verified in UPPAAL. We use model checking because it is a fully automatic verification method, which is an absolute must in the context of co-simulation where practitioners come from a wide range of backgrounds. Furthermore, we believe model checking is ideal for verifying orchestration algorithms since it allows us to specify non-deterministic SU behavior, and, under the assumptions that will be illustrated throughout the paper, the verification problem is always finite.

3.3.1 The UPPAAL model

The UPPAAL model[1] is a formalization of a co-simulation. The UPPAAL model formalizes the co-simulation as a set of SUs described as timed automata (TA) orchestrated by an orchestrator described as another TA. The orchestrator exchanges data between the SUs and asks them to compute future states by interpreting the provided orchestration algorithm. The UPPAAL model is designed such that all the violations of the semantics of the SUs lead to a *deadlock*. This allows us to automatically verify orchestration algorithms in UPPAAL.

The model consists of four templates:

- The SU template that formalizes the interface of an SU described in Definition 1 on page 6.
- The Orchestrator template interprets the provided algorithm and exercise the SUs accordingly.
- The Step Negotiation template is used to verify step negotiation procedures.
- The Fixed Point template is used to verify fixed-point iteration procedures.

The templates of the SU and the Orchestrator are described in this section. In contrast, the other templates are described in Sect. 4.

The Scenario-Verifier instantiates the UPPAAL model based on the provided scenario and algorithm by instantiating one SU template for each SU in the scenario, and one of the other templates per scenario. The data of the scenario (couplings, feed-through, etc.) and the orchestration algorithm are globally defined and automatically generated by the tool.

3.3.2 The orchestrator template

The orchestrator sequentially interprets the orchestration algorithm and exercises the SUs accordingly. In UPPAAL, an orchestration algorithm is interpreted as a sequence of SU-actions executed in the provided order. The orchestrator delegates actions to the SUs using channel synchronization (`su[activeSU]!`) and shared variables (`action` and `var`). The orchestrator has one channel per SU to ensure that only one SU receives and synchronizes on a given action request.

The orchestrator is created according to the FMI standard [2], such that it initializes, simulates, and terminates the scenario. Fig. 4 shows the UPPAAL template of the orchestrator; more specifically, it shows how it interprets and runs the co-simulation step. It sends action requests to SUs using the channel `su[activeSU]!` and waits for a confirmation that the action was successfully performed on the broadcast channel `actionPerformed?`. The Orchestrator then selects the next action in the algorithm using the function `selectNextAction()`. This pattern is repeated until the entire algorithm has been interpreted, i.e. `!AlgorithmDone()`. The edges to the states `FindStep` and `SolveAlgebraicLoop` are only relevant for complex scenarios, which means that we will not treat them in this section. The same goes for the state `SelectMaxStepSize`. In case the Orchestrator has performed the entire co-simulation step (`AlgorithmDone()`), reached the end time of the simulation (`time >= end`), and established a consistent co-simulation state (`ConsistentState`) the Orchestrator goes to the `Termination` state. The transition from the state `Simulate` to the state `SelectMaxStepSize` makes a non-deterministic choice that enables verification of adaptive scenarios, which we discuss in Sect. 5.

A correct orchestration algorithm ensures that the orchestrator reaches the state `Termination` while an incorrect hits a deadlock.

3.3.3 The SU template

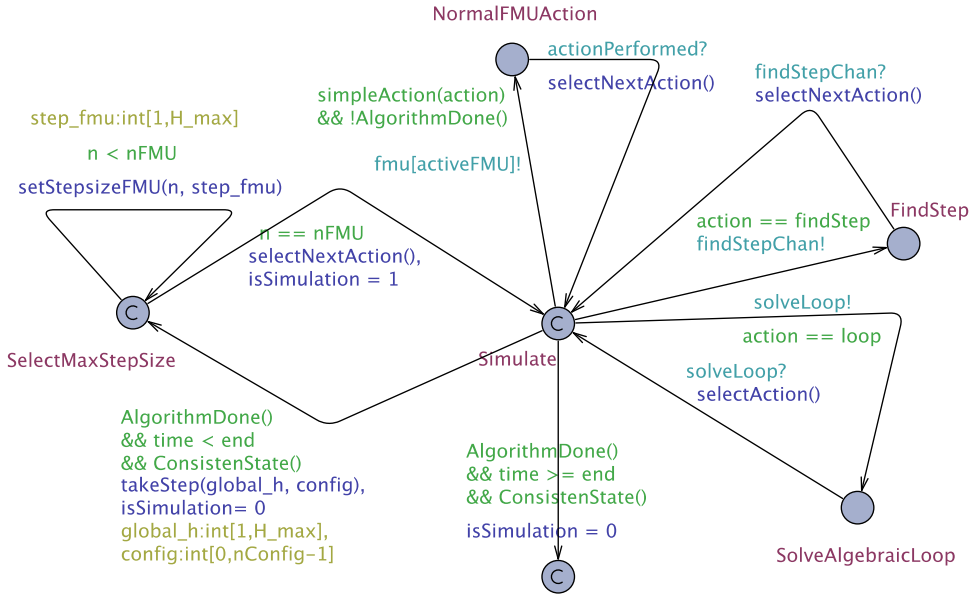
The SU template abstracts an SU by defining its interface and life-cycle. The state of the SU in UPPAAL is defined in Definition 3. The SU template is depicted in Fig. 5. The figure shows the interface of the SU when it is in `simulation` mode. The outgoing edges of the `Simulation` state show different actions that the Orchestrator can invoke on the SU by synchronizing on the channel `su[id]`. The incoming edge to the `Simulation` state is used to synchronize on the channel `actionPerformed` to confirm that the SU successfully performed the requested action, enabling the Orchestrator to issue a new action request. The preconditions of the actions described in Definitions 6 to 8 on pages 10–11 are encoded in UPPAAL as invariant functions of specific states as shown in Fig. 5 by the functions `preSet`, `preGet` and `preStep`. The actions described by `setValue`, `getValue` and `step` in the figure changes the state of the SU according to Definitions 6 to 8 on pages 10–11.

The invariant function ensures that a simulation in UPPAAL respects the defined semantics. A violation of an invariant function results in a deadlock since the SU cannot synchronize with the orchestrator, enabling us to detect errors in the orchestration algorithm in UPPAAL by checking for deadlocks.

3.4 Checking an algorithm in UPPAAL

A correct algorithm correctly instantiates, simulates, and terminates all SUs meaning that the Orchestrator reaches the

[1] The model is available online: https://github.com/INTO-CPS-Association/Scenario-Verifier.



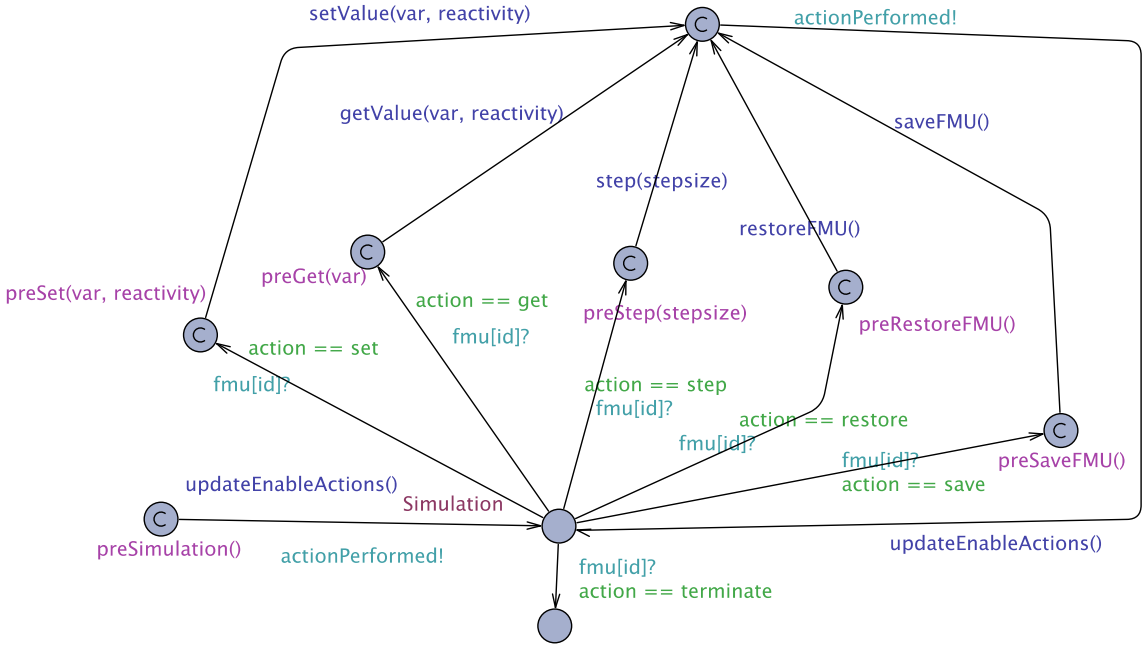


Fig. 4 A part of the Orchestrator template in UPPAAL

Fig. 5 A part of the SU template in UPPAAL

state *Terminated*. If the Orchestrator reaches the state *Terminated* it means that the algorithm respects the semantics (preconditions) since no deadlocks are detected. Furthermore, it shows that a consistent co-simulation state is established by the Orchestrator using the algorithm. All transitions are guarded except one identity transition in the trap state *Terminated* in the Orchestrator-template. This ensures that all semantical violations result in a deadlock; furthermore, no deadlock can occur if the Orchestrator-template reaches the state *Terminated*.

The correctness of an algorithm is checked using the CTL formula:

$$A\Diamond \text{Orchestrator.Terminated} \tag{1}$$

The formula ensures that the orchestrator always eventually ($A\Diamond$) reaches the state Terminated, implying that the co-simulation algorithm terminates in a final consistent state. Since the final state is consistent, and the simulation starts in a consistent state, UPPAAL can check if the algorithm

respects Definition 9. The initial state and the final state of the simulation differ only by their timestamp, which allows us to use inductive reasoning to establish the correctness of the algorithm for all future co-simulation steps starting from a consistent state as stated in the previous section. We do not check explicitly for deadlocks since the formula Eq. (1) implies that the model is deadlock free.

Looking back at Algorithms 1 and 2 on page 4, we see that Algorithm 2 violates `preStep` when it tries to step the SU `A` in line 2, since no value is provided for the reactive input u_g. Thus, the SU cannot synchronize with the orchestrator, resulting in a deadlock that falsifies the formula in Eq. (1). Algorithm 1 satisfies the CTL formula and therefore also Definition 9.

4 Complex co-simulation scenarios

This section describes complex scenarios which are an extension of the simple scenarios. Complex scenarios are subject to algebraic loops or step negotiation (see Definition 12). Examples of complex scenarios are presented in Fig. 6a and b.

Before describing the orchestration algorithms of complex scenarios (Sect. 4.3), we provide some background on them. Finally, Sect. 4.4 presents a technique to verify orchestration algorithms of complex scenarios in UPPAAL. The technique has been implemented in the *Scenario-Verifier*.

4.1 Algebraic loops

An algebraic loop states that the value of a port depends on itself. Real-world examples of systems with algebraic loops include the correct initialization of a suspension system of a car or a system consisting of multiple connected spring dampers [23]. The scenario in Fig. 6a shows the scenario of a suspension system with an algebraic loop.

An algebraic loop in a co-simulation scenario is a consequence of the contracts, feed-through, and couplings between the SUs. There exist two kinds of algebraic loops, as identified in [3, Fig. 5 and 6]:

- Feed-through loops—happens due to feed-through constraints that indicates that an output port is explicitly dependent on an input port. A feed-through loop exists if we have a sequence of `get` and `set` actions P_{FA} between multiple SUs such that there exists an output port $y \in Y$ which is sensitive to the number of executions of P_{FA}:

$$\begin{aligned} &\left\langle t, s_U^R, s_Y^R, s_{\mathcal{V}}^R \right\rangle \xrightarrow{P_{FA}} \left\langle t, s_U^{R'}, s_Y^{R'}, s_{\mathcal{V}}^{R'} \right\rangle \\ &\quad \xrightarrow{P_{FA}^{+}} \left\langle t, s_U^{R'}, s_Y^{R'}, s_{\mathcal{V}}^{R''} \right\rangle \\ &\quad \implies \exists y \in Y \cdot s_{\mathcal{V}}^{R'}(y) \neq s_{\mathcal{V}}^{R''}(y) \end{aligned}$$

- Reactivity loops—happens due to reactivity constraints that indicates that an output port is explicitly dependent on a state computation of another SU. A reactivity loop exists if we have a sequence of `get`, `set` and `step` actions P_{RA} between multiple SUs such that there exists an output port $y \in Y$ which is sensitive to the number of executions of P_{RA}:

$$\begin{aligned} &\left\langle t, s_U^R, s_Y^R, s_{\mathcal{V}}^R \right\rangle \xrightarrow{P_{RA}} \left\langle t, s_U^{R'}, s_Y^{R'}, s_{\mathcal{V}}^{R'} \right\rangle \\ \wedge &\left\langle t, s_U^R, s_Y^R, s_{\mathcal{V}}^R \right\rangle \xrightarrow{P_{RA}} \left\langle t, s_U^{R'}, s_Y^{R'}, s_{\mathcal{V}}^{R'} \right\rangle \end{aligned}$$

They differ in nature, but the way to simulate them and their impact on the scenario are very similar. The scenario in Fig. 6a contains a feed-through loop, while the scenario in Fig. 8b on page 26 contains a reactivity loop. An algebraic loop can be a combination of the two described kinds.

The orchestrator solves an algebraic loop using a technique called *fixed point iteration* (see Algorithm 5 on page 25 for an example). The fixed point iteration is an iterative approach aiming to find a fixed point for all the ports in the loop using an iterative search. The set $algebraic_S$ denotes all ports of a scenario that are a part of an algebraic loop.

4.2 Step negotiation

Step negotiation is a consequence of SUs rejecting to perform a step of the desired duration. Step negotiation is the process for detecting step rejections and ensuring that all SUs move in lockstep by negotiating an appropriate step duration. A step rejection is defined in Definition 10 and can be triggered by:

1. Events occurring inside the SU leading to discontinuous changes to its outputs (e.g., a collision);
2. Conservative numerical error control schemes that adjust the step size based on how quickly the state of the SU changes over time. When the error is too large, these schemes ask the orchestrator for a smaller step duration and to repeat the step procedure [25, Sect. 3.9].

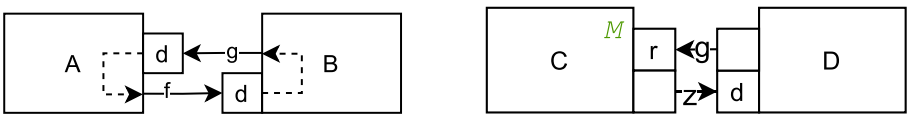


Fig. 6 Complex co-simulation scenarios

(a) Scenario $S2$ with an algebraic loop. **(b)** Scenario $S3$ needs step negotiation.

Definition 10 *(Step rejection)* An SU m rejects a step if the following can happen:

$$\texttt{step}_m(s_m, H) = \langle \_, h \rangle \text{, where } h < H$$

A step rejection means that we should adjust the step duration for all the SUs in the scenario to ensure they move in lockstep. We can adjust the step duration during the co-simulation as we discover step rejections. However, this does not help us if we have stepped some SUs with the original step duration of, for example, 5 and now discover a step rejection meaning that one SU is at time 3. The only way to handle this is to restore the SUs at time 5 to their original state and ask them to step with the new and smaller duration 3.

The set M (defined in Definition 2 on page 6) contains all the SUs that may reject a step. Step rejections need special attention since they can result in a co-simulation step where the SUs do not move in lockstep, resulting in a violation of Definition 9 on page 12. Step negotiation is needed if $M \neq \emptyset$ - indicating that at least one SU could reject a step. A step duration is acceptable if all SUs accept it:

Definition 11 *(Acceptable step)* A step duration $h \in \mathbb{R}_{>0}$ is accepted if:

$$\forall c \in C \cdot \texttt{step}_c(s_c^{(t)}, h) = \langle \_, h \rangle$$

Step negotiation can, in some cases, be avoided by choosing an appropriate *fixed* step duration that can be used throughout the simulation. However, an appropriate step duration can be hard to determine for a complex CPS described as the composition of black-box SUs. Furthermore, a fixed duration can be inefficient for stiff systems where the optimal step duration varies over time due to rapid and unpredictable state changes [25].

4.3 Orchestration algorithms for complex scenarios

This section describes how complex scenarios are simulated.

Definition 12 *(Complex Scenario)* A scenario $\mathcal{S}$ is complex if $M \neq \emptyset \ \vee \ algebraic_{\mathcal{S}} \neq \emptyset$.

Complex scenarios are challenging to simulate because the orchestration algorithm needs to adapt to the behavior of the SUs to satisfy the constraints associated with each SU, account for possible step rejections, and solve algebraic loops.

Since the internal state of an SU is hidden from the orchestrator, it cannot accurately predict the behavior of the SUs. The orchestrator can only experiment and see how the SUs react to different step sizes and input values. The orchestrator performs such experiments using a *valuation* to see if any of the SUs rejects a given step size and check if all algebraic loops have been solved. A valuation (Definition 13) describes the step duration of the simulation and the values to solve the algebraic loops. If the experiment fails, the orchestrator backtracks the simulation to perform a new experiment using a different valuation from the same initial state. The experiment is repeated until the SUs are in lockstep, and all algebraic loops are solved. Therefore, we call algorithms for complex scenarios *iterative algorithms*.

Definition 13 *(Valuation)* The valuation is a tuple $\langle H, \texttt{guess} \rangle$, denoting the step duration $H \in \mathbb{R}_{>0}$. The function $\texttt{guess} : U_{algebraic} \to \mathcal{V}_{\mathcal{T}}$, where $U_{algebraic} = algebraic_S \cap U$ links all inputs of the algebraic loops with a timestamped value that tries to solve the algebraic loop. A general algorithm for finding a correct valuation is shown in Algorithm 3.

Algorithm 3 Finding a Correct Valuation.

```
1: s_0^R ← s^R                                   ▷ Save the SUs.
2: Converged ← false                             ▷ The experiment has not converged.
3: v_i ← v_0                                     ▷ Use the initial value in the first iteration.
4: while !Converged do                           ▷ Start the experient
5:     ⟨v_{i+1}, s^{R'}⟩ ← RunAlgorithm(P_S, v_i, s^R)
6:     Converged ← CheckConv(v_i, v_{i+1})
7:     if Converged then
8:         return ⟨v_{i+1}, s^{R'}⟩               ▷ The experiment converged, so we return.
9:     else
10:        s^R ← s_0^R                           ▷ Restore the SUs and try again
11:        v_i ← v_{i+1}                         ▷ Try with the updated valuation.
12:    end if
13: end while
```

Algorithm 3 starts by saving the SUs such that they can be restored later and computes an initial valuation $v_0 = \langle H_0, \texttt{guess}_0 \rangle$ using the strategy described below.

$$H_0 = \text{The chosen step duration } H \text{ for the simulation} \tag{2}$$

$$\texttt{guess}_0 = \{(u \to val) \mid u \in U_{algebraic} \wedge val \in \mathcal{V}_{\mathcal{T}} \wedge L(u) = y \wedge val = \left\langle t, s_{\mathcal{V}}^R(y) \right\rangle\} \tag{3}$$

The initial step duration is defined as a parameter of the co-simulation. Equation (3) explains that each input in the set $U_{algebraic}$ is associated with the current value of its coupled output.

The orchestrator then runs the experiment using the algorithm P_S with the current valuation v_i to see how the SUs react The subscript notation $*_i$ describes the values of the valuation at the i^{th} search attempt/experiment. Note that $P_S \subseteq P$, where P is the orchestration algorithm of the scenario. An execution of P_S transforms the current valuation $v_i = \langle H_i, \texttt{guess}_i \rangle$ to a new valuation $v_{i+1} = \langle H_{i+1}, \texttt{guess}_{i+1} \rangle$ where H_{i+1} is the smallest step duration accepted by an SU during the execution of P_S as seen in Eq. (4). The function $\texttt{guess}_{i+1}$ has the same domain as $\texttt{guess}_i$, but the range is updated such that the inputs are mapped to the new value of the coupled output as seen in Eq. (5). This is the standard approach for solving algebraic loops using the most recent value as a guess [3]. An example of the process for updating the step duration is shown in Example 3.

$$\begin{aligned}(\langle H_i, \texttt{guess}_i \rangle, \left\langle t, s_U^R, s_Y^R, s_\mathcal{V}^R \right\rangle) \xrightarrow{P_I} (\langle H_{i+1}, \texttt{guess}_{i+1} \rangle, \\ \times \left\langle time', s_U^{R'}, s_Y^{R'}, s_\mathcal{V}^{R'} \right\rangle) \\ \implies H_{i+1} = h \text{ where: } h \in \mathbb{R}_{>0} \wedge \forall c \in C \cdot h \leq time'(c)\end{aligned} \tag{4}$$

$$\begin{aligned}\wedge\ \texttt{guess}_{i+1} = \{(u \to val) \mid u \in U_{algebraic} \wedge val \in \mathcal{V}_\mathcal{T} \\ \wedge L(u) = y \wedge val = \left\langle s_Y^{R'}(y), s_\mathcal{V}^{R'}(y) \right\rangle\}\end{aligned} \tag{5}$$

Then the experiment is finished, the orchestrator checks if the SUs are in lockstep and if all algebraic loops are solved by comparing the valuations v_i and v_{i+1} to see if the simulation has converged according to Definition 14.

Definition 14 *(Correct Valuation)* Two valuations $v_j = \langle H_1, \texttt{guess}_1 \rangle$ and $v_{j+1} = \langle H_2, guess_2 \rangle$ of the same scenario $\mathcal{S}$ are correct if:

$$\begin{aligned}v_j \approx v_{j+1} \triangleq H_1 \\ = H_2 \wedge (\forall i \in dom(guess).\, \texttt{guess}_1[i] \approx \texttt{guess}_2[i]),\end{aligned}$$

where $\texttt{guess}_1[i] = \langle t_1, v_1 \rangle$ and $\texttt{guess}_2[i] = \langle t_2, v_2 \rangle$. When $\langle t_1, v_1 \rangle \approx \langle t_2, v_2 \rangle$, if:

$$\mid v_1 - v_2 \mid \leq \epsilon \wedge t_1 = t_2$$

Epsilon ϵ is a constant defined by the co-simulation practitioner.

Definition 14 says that the simulation has converged, e.g., the experiment succeeds if all SUs accepted the step and a fixed-point was established on all algebraic loops.

If the experiment fails, the orchestrator backtracks the simulation to the initial co-simulation state and tries again using the new and updated valuation. We assume that all SUs of a complex scenario allow their state to be restored, which is an optional feature in the FMI standard.

Assumption 1 All SUs C of a complex scenario S allow their state to be restored.

Example 3 *(Finding for an Acceptable Step Duration)* Consider Algorithm 4 that is the procedure to negotiate a step between the SUs C and D. Assuming that the scenario starts from a consistent state, where $s_C^R = \langle t, \{u_g \to s\}, \{y_z \to s\}, \_ \rangle$, and $s_D^R = \langle t, \{u_z \to s\}\{y_f \to s\}, \_ \rangle$. The initial valuation is $\langle H, \emptyset \rangle$.

The states of the SUs after one iteration of the algorithm are $s_C^R = \langle H_C, \_, \_, \_ \rangle$ and $s_D^R = \langle H_D, \_, \_, \_ \rangle$, where the SUs either have taken the same step ($H_C = H_D$) or to different steps ($H_C \neq H_D$). If the SUs are synchronized, an acceptable step has been found. Otherwise, the initial state is restored and a new search attempt is initiated with an updated valuation $v_1 = \langle min(H_C, H_D), \emptyset \rangle$.

Algorithm 4 Step negotiation Procedure of the scenario in Fig. 6b.

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

The valuation is only relevant for complex scenarios since all valuations for simple scenarios are correct because simple scenarios are not subject to either algebraic loops or step rejections.

On the other hand, the valuation is crucial for simulating complex scenarios since the orchestrator may have to deal with incorrect valuations. It can be the case that no correct valuation exists, these scenarios are typical a consequence of an error in one of the SUs. A scenario can only be correctly simulated if a correct valuation exists since an execution that uses an incorrect valuation does not respect the semantics.

A complex scenario is correctly simulated if the orchestrator can find a correct valuation that can take the scenario from a consistent initial state to a consistent final state as stated by Definition 15.

Definition 15 An algorithm P is correct for the complex scenario $\mathcal{S}$ if:

$$\begin{aligned}
&\texttt{consistent}\left(\left\langle t, s_U^R, s_Y^R, s_V^R\right\rangle\right) \\
&\quad \implies wp\left(P, \texttt{consistent}\left(\left\langle t + H, s_U^{R'}, s_Y^{R'}, s_V^{R'}\right\rangle\right)\right) \\
&\quad \wedge H > 0 \wedge v = \langle H, \texttt{guess}\rangle \\
&\quad \wedge \forall u \in dom(\texttt{guess}) \cdot \texttt{guess}(u) \approx s_V^{R'}(u)
\end{aligned}$$

We have now presented the general approach for finding a correct valuation to satisfy the constraints of a complex scenario. Next, we introduce the approach for verifying such scenarios in UPPAAL.

4.4 Verifying complex simulation scenarios in UPPAAL

This section presents the approach for verifying complex simulation scenarios in UPPAAL by extending/updating the approach presented in Sect. 3.3.

The UPPAAL model still verifies the algorithm by instantiating the scenario and verifying that its algorithm respects all the constraints of the scenario and reaches a consistent final state.

An orchestration engine must find a correct valuation in *each* co-simulation step since the valuation depends on the current state of the system. However, this is not the case for the UPPAAL model that takes place in the state space of the abstraction defined in Definition 4 on page 9, where the only difference between two consistent states is their timestamp as shown in Definition 9.

The abstraction allows us to conclude that if we can use the search procedure P_S to find a correct valuation from one consistent state, we can use P_S to find a correct valuation from any consistent state. The limitations of this choice are discussed in Sect. 4.6.

The following paragraphs describe the extensions of the UPPAAL model described in Sect. 3.3 on page 12 to cover complex scenarios.

4.4.1 Avoiding false positives

The UPPAAL model presented in Sect. 3.3 on page 12 deadlocks on *all* violations of a precondition and interprets the orchestration algorithm as incorrect. This approach is too strict for complex scenarios and will make the UPPAAL model discover *false positives* since all search attempts using an incorrect valuation violates at least one precondition since the algorithm of complex scenarios only respect the semantics using a correct valuation. Therefore, it is wrong to immediately declare an algorithm of a complex scenario incorrect, if we have not found a correct valuation.

An example of a false positive can be seen in Algorithm 4. The false positive arises when SU C cannot step as far as SU D, resulting in a simulation where the SUs are not synchronized. This would have been a mistake in the co-simulation. However, the backtracking and the next iteration of the algorithm will ensure they move in lockstep by negotiating a proper step.

The UPPAAL model ignores such false positives by temporarily *disabling* the preconditions/invariant functions `preSet`, `preGet` and `preStep` by setting a global flag while searching for a correct valuation.

The global flag is set by the function `disableChecks`, which is called on the transitions to the states `FindStep` and `SolveAlgebraicLoop` in Fig. 4. The transition to `FindStep` activates the FindStep template on the channel `findStepChan`. Once the model has found a correct valuation, the checks are re-enabled. An extra iteration is performed to verify that the algorithm is correct, e.g., it respects the semantics using a given valuation to reach a consistent final state.

4.4.2 Verifying a step negotiation procedure

UPPAAL verifies a step negotiation to ensure that the algorithm will be able to find an acceptable step duration (see Definition 11 on page 18) using the supplied algorithm. Furthermore, it ensures that all preconditions are satisfied using the found step duration. The FindStep template depicted in Fig. 7 is responsible for verifying the step negotiation procedure.

The FindStep template executes the step negotiation procedure by exercising the SUs through the channel `fmu[activeSU]!`. When all actions in the iterative algorithm have been executed (`AlgorithmDone`), UPPAAL checks whether a correct valuation has been identified using the function `loopConverged` following Definition 14. If a correct valuation has been found, the model reactivates the preconditions using the function `UpdateIsExtra`. Then it backtracks the involved SUs to ensure that all search attempts start from the same state before initiating a new search attempt with an updated valuation. The algorithm describes how the simulation should be backtracked and when the SUs states should be saved and restored. Note that preconditions are only checked using a convergent/correct valuation. This ensures that the model does not discover false positives since it only considers the semantics of a correct valuation.

To enable step rejections in UPPAAL, we assume that an SU has a maximal step h_{Max}, which is the largest step that the SU can take. An SU will reject all steps larger than h_{Max} and accepts all positive step durations smaller or equal to its maximal step:

Definition 16 *[Maximal Step] A step $h_{Max} \in \mathbb{R}_{>0}$ is maximal for an SU m if:*

$$\forall h \in \mathbb{R}_{>0} \cdot$$
$$(h < h_{Max} \implies \texttt{step}_m(s_m, h) = \langle \_, h \rangle) \wedge$$
$$(h > h_{Max} \implies \texttt{step}_m(s_m, h) = \texttt{step}_m(s_m, h_{Max}))$$

The UPPAAL model selects a maximal step in the set $\{1, 2\}$ for each SU in M as a non-deterministic choice (see `setStepSize` in Fig. 4). The values of the set are not related to the real maximal step of the SU. The important information is that the set contains two ordered elements, and the SUs that randomly select 1 will reject the step. The two elements enable the UPPAAL model to exhibit non-deterministic behavior (step rejections) while keeping the model's state space as small as possible.

Assumption 2 An SU c where $c \notin M$ has a maximal step larger than any SU d where $d \in M$.

The maximal step for all SUs not in M is 2 in the UPPAAL model, which means they will never reject a step.

Definition 16 and the strategy for updating the step duration in Eq. (4) allows us to deduce that the step negotiation always terminates within two iterations. The reason is that the step duration of the first iteration H_1 is always an accepted step duration since all SUs return a step duration h smaller or equal to their maximal step. This means that $\forall c \in C \cdot H_1 \leq h_{Max}$.

4.4.3 Verifying a fixed-point procedure

Scenarios with algebraic loops are simulated using a fixed-point iteration procedure (see Algorithm 5 for an example) that solves the algebraic loops by finding fixed points on the involved ports. The fixed-point procedure is verified in UPPAAL by the AlgebraicLoop template, which is almost identical to the StepFinder template.

Algebraic loops are a consequence of the numerical aspect of the coupled SUs, which we have abstracted from in our formalization.

The abstraction means that we using a correct search algorithm can find the fixed points of an algebraic loop within two iterations.

Example 4 Consider Algorithm 5; the abstract state of the SUs A and B before the FP procedure (line 1) are $s_A^R = \langle s, \{u_f \to s\}, \{y_x \to s\}, \_ \rangle$, and $s_B^R = \langle s, \{u_x \to s\}\{y_f \to s\}, \_ \rangle$, respectively. The valuation is initially given by $v0 = \langle H, \{f_{val} \to s\}\{x_{val} \to s\} \rangle$, which will violate the preconditions of the set actions in lines 3 and 4 on the reactive inputs u_x and u_f and therefore also the step actions of the two SUs. However, these violations will not result in an error in UPPAAL, since we have disabled the checks. The valuation after running one iteration of the search (line 13) $v_1 = \langle H, \{f_{val} \to s + H\}\{x_{val} \to s + H\} \rangle$. Since $v_0 \not\approx v_1$, we backtrack the SUs to the initial state and runs the search again. The valuation after running the second iteration of the search (line 13) $v_2 = \langle H, \{f_{val} \to s + H\}\{x_{val} \to s + H\} \rangle$, where $v_1 \approx v_2$, e.g., the valuation is correct. A correct valuation respects the reactive inputs u_x and u_f.

Algorithm 5 Step procedure for the scenario in Fig. 8b.

1: $(s_{A_v}^{(s)}, s_{B_v}^{(s)}) \leftarrow (s_A^{(s)}, s_B^{(s)})$ ▷ Save A and B
2: **while** $!conv$ **do** ▷ FP procedure
3: $s_A^{(s)} \leftarrow \texttt{set}_A(s_A^{(s)}, u_f, f_{val})$ ▷ Informed Guess
4: $s_B^{(s)} \leftarrow \texttt{set}_B(s_B^{(s)}, u_x, x_{val})$ ▷ Informed Guess
5: $(s_A^{(s+h)}, h), \leftarrow \texttt{step}_A(s_A^{(s)}, h)$
6: $(s_B^{(s+h)}, h), \leftarrow \texttt{step}_B(s_B^{(s)}, h)$
7: $x_{vala} \leftarrow \texttt{get}_A(s_A^{(s+h)}, y_x)$
8: $f_{vala} \leftarrow \texttt{get}_B(s_B^{(s+h)}, y_f)$
9: $conv \leftarrow CheckConv((f_{vala}, f_{val}), (x_{vala}, x_{val}))$
10: **if** $!conv$ **then**
11: $(s_A^{(s)}, s_B^{(s)}) \leftarrow (s_{A_v}^{(s)}, s_{B_v}^{(s)})$ ▷ Restore A and B
12: **end if**
13: $(f_{val}, x_{val}) \leftarrow (f_{vala}, x_{vala})$ ▷ Update the informed guesses x and f
14: **end while**
15: $(s_C^{(s+h)}, h), \leftarrow \texttt{step}_C(s_C^{(s)}, h)$
16: $j_{val} \leftarrow \texttt{get}_C(s_C^{(s+h)}, y_j)$
17: $s_B^{(s+h)} \leftarrow \texttt{set}_B(s_B^{(s+h)}, u_j, j_{val})$

4.5 Nested complex scenarios

A complex scenario can be subject to both algebraic loops and step rejections; such a scenario is called a *nested complex scenario*, Fig. 8a shows an example.

A nested complex scenario is simulated using a nested search. The outer search establishes the step duration using step negotiation, while the inner search solves the algebraic loops using fixed-point iteration. The transition to `SolveAlgebraicLoop` in Fig. 7 on page 23 activates the SolveAlgebraicLoop template, which synchronizes on the channel `solveAlgebraicLoopChan`. This allows us to verify nested complex scenarios in UPPAAL.

The algorithm to simulate the scenario in Fig. 8a is shown in Sect. C. Algorithm 7 is too complex to be analyzed with a simple visual inspection, showing the necessity of the UPPAAL model created in this paper. The tool can analyze the algorithm in a few seconds.

4.6 Limitations of the UPPAAL model

The UPPAAL model can only be used to analyze the correctness of the algorithms concerning the semantics, not the numerical aspect of the simulation. This is because the UPPAAL model does not know the numerical properties of the simulation, which is the case for all general analyses of

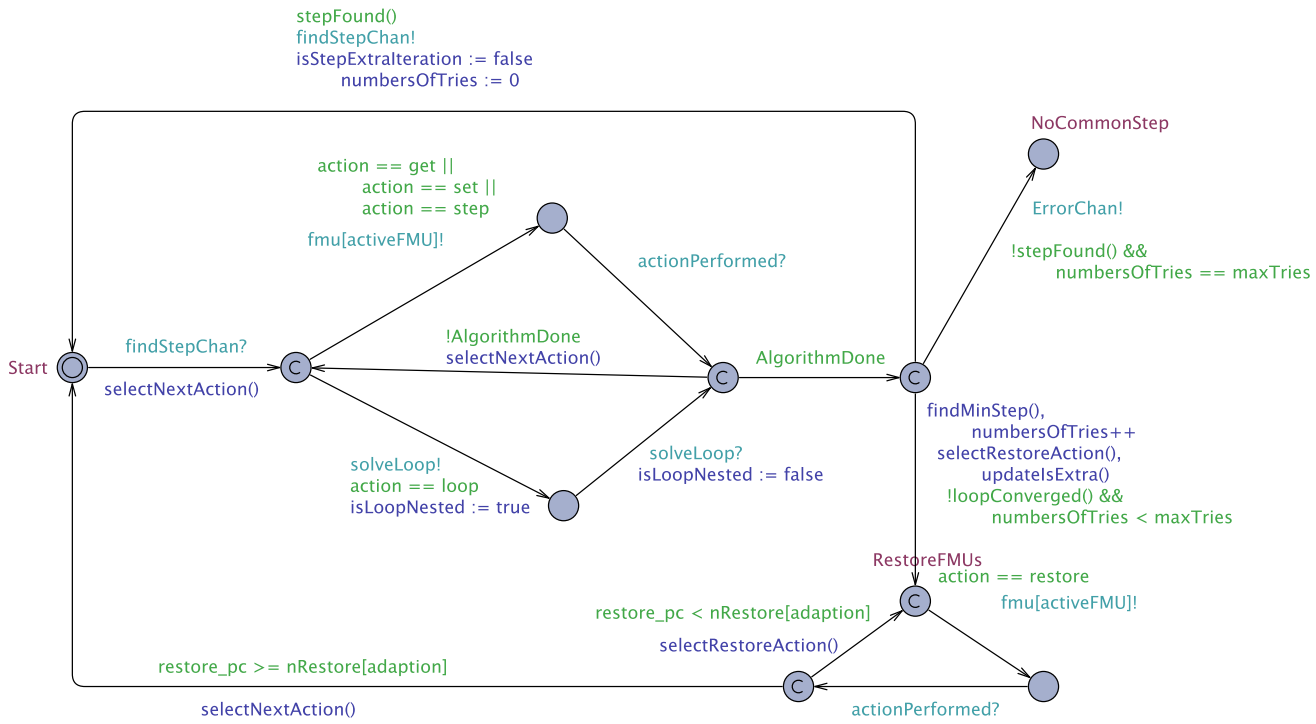
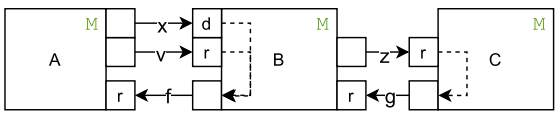
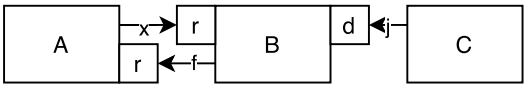


Fig. 7 FindStep template



(a) A nested complex scenario.



(b) Co-simulation scenario with a reactivity loop.

Fig. 8 Two complex co-simulation scenarios

co-simulations and orchestration engines. Consequently, the UPPAAL model cannot infer if a convergent valuation exists. We, therefore, decided to take the safe choice and delegate the problem of finding a convergent valuation for algebraic loops to the orchestration engine that performs the simulation using the verified algorithm. This means that the orchestration engine should take the appropriate measures to detect and handle such cases; a typical approach uses a finite number of search attempts to see if a convergent valuation can be established and aborts if it is not the case.

Nevertheless, co-simulation practitioners benefit from the UPPAAL analysis because it helps discovering unstable solutions caused by wrong co-simulation algorithms.

Assumptions 2 and 1 restrict the behavior of the SUs, restricting the use of the tool to a particular class of scenarios. We believe that the Assumption 1 is necessary and the only way to treat complex co-simulation scenarios generally. In practice, all complex scenarios violating these assumptions would be hard to simulate. It would require many simulations and extensive bookkeeping to find and keep track of the correct valuations to simulate the system properly.

5 Adaptive co-simulation scenarios

This section introduces *adaptive scenarios* with adaptive reactivity and couplings, where the reactivity and couplings can change *during* the simulation. Adaptive scenarios were introduced by Inci et al. in [26] to minimize the co-simulation error by dynamically choosing the co-simulation step. This means that the orchestrator simulates the scenario by executing different co-simulation step procedures. For example, a trace of a simulation of an adaptive scenario could be:

$$s^R \xrightarrow{P_1} s_1^R \xrightarrow{P_2} s_2^R \xrightarrow{P_3} s_3^R$$

where P_1, P_2, and P_3 shows three different co-simulation steps. The orchestrator dynamically selects the algorithm such that the chosen algorithm results in the smallest pos-

sible simulation error. The co-simulation shown by the trace above is correct if the initial state s^R is consistent and the algorithms P_1, P_2, and P_3 are correct such that the states s_1^R, s_2^R, and s_3^R are consistent states.

Adaptive scenarios are not simulated differently from simple and complex scenarios. The difference is that the orchestrator dynamically chooses the appropriate co-simulation step procedure during the simulation.

The dynamical co-simulation step selection is outside the scope of this work since the step selection depends on the numerical aspects of the simulation. Our work is about ensuring that all possible choices (co-simulation procedures) the orchestration engine could make are correct. Inci et al. [26] select the appropriate co-simulation step by comparing the analytical solution against the expected simulation results produced by different algorithms. Then, they select the algorithm with the smallest discrepancy to the analytical solution.

Nevertheless, the verification technique described in Sects. 3 and 4 still applies to adaptive scenarios. The following paragraphs describe the technique.

To adequately describe adaptive scenarios, we change the definition of a scenario from Definition 2 to Definition 17.

Definition 17 *(Adaptive Scenario)* An adaptive scenario $\mathcal{S}_\mathcal{A}$ is a tuple

$$\langle C, \mathcal{A}, M, F \rangle ,$$

where:

- $\mathcal{A}$ is a function $\mathcal{A} : \mathbb{I} \rightarrow L \times R$ mapping an adaptation identifier to its connections and contracts. L and R are as in Definition 2.
- M, C, and F are defined similarly as in Definition 2.

The most significant difference between Definition 2 and Definition 17 is an extra level of indirection such that an adaptation specifies the changeable aspects of the scenario (couplings and reactivity constraints). We use the syntax in Fig. 9a to present an adaptive scenario where all inputs are *reactive* in adaptation 1 and *delayed* in adaptation 2. The scenario in Fig. 9a represents a system consisting of two linear mass-spring-damper subsystems and originates from [26] where more details of the system can be found. Figures 9b and 9c show the two adaptations of the scenario.

All previously described scenarios in the paper are adaptive scenarios with only one adaptation. This means that the extension of this section is backwards compatible. An adaptive scenario has N adaptations where $\mid \mathbb{I} \mid = N$, which can be seen as N distinct scenarios. An adaptive scenario can contain both simple and complex adaptations.

To properly simulate an adaptive scenario with N adaptations, we need N correct algorithms. Different algorithms are placed in a map $P_\mathbb{I} : \mathbb{I} \rightarrow Algorithm$ where each identifier $a \in \mathbb{I}$ of an adaptation selects the algorithm P_a to simulate the corresponding adaptation. We can now formulate the criteria for the correct simulation of an adaptive scenario.

Definition 18 *(Correct Algorithm)* An adaptive scenario $\mathcal{S} = \langle C, \mathcal{A}, M, F \rangle$ is simulated correctly by the corresponding algorithm in $P_\mathbb{I}$ concerning Definition 15.

5.1 Verification of adaptive scenarios in UPPAAL

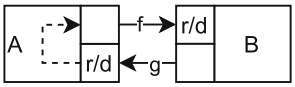
The verification approach for simple and complex scenarios also applies to adaptive scenarios. The technique is to verify each adaptation individually using the previously described techniques, such that we can verify the scenario and its algorithms against Definition 18.

The UPPAAL model verifies each adaptation individually. It uses a non-deterministic choice enabling quantification over different adaptations. The non-deterministic choice, `config:int[nConfig-1]`, is shown in Fig. 4 on page 14, where the model selects an adaptation (`config`) and instantiates the adaptation using the function `takeStep`. The function `takeStep` updates the scenario to the chosen adaptation by changing the adaptive parts (couplings, algorithm, and reactivity). The adaptive parts are placed in a dictionary that links each adaptation with its corresponding parts. This structure makes it easy to change between different adaptations. The chosen adaptation is stored in a global variable in UPPAAL, which allows all TA to adapt to the current adaptation. An example of this is the reactivity constraints of an SU, which are stored in a dictionary, where the current adaptation selects the reactivity constraints to consider. We only change the adaptations between co-simulation steps. The CTL formula in Eq. (1) ensures that all adaptations are verified.

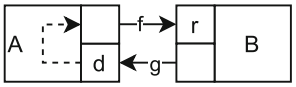
5.2 General remarks on the UPPAAL model

The state-space of the UPPAAL model is always *finite*. The number of transitions in the UPPAAL model is linear to the number of actions in the orchestration algorithm. The following equation computes the number of actions in the co-simulations step: $\mid U \mid + \mid Y \mid + \mid C \mid$ for simple scenarios. This number can be multiplied by three to get the number of actions/transitions UPPAAL needs to consider for a complex scenario.

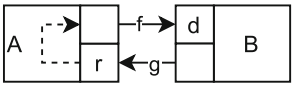
The verification in UPPAAL has certain limitations, primarily consequences of the nature of co-simulation. They exist *for all* formalizations of co-simulation, which consider the SUs as black boxes. The black boxes are necessary because they protect the model's IP, an absolute must from an industrial viewpoint. However, from a verification perspective, it means that the verification cannot account for the numerical aspect of the system because the numerical aspect of the individual SU, in general, is unknown.



(a) An adaptive scenario. The adaptive contracts are specified in the input of the SUs. The two adaptations are split by the "/".



(b) Adaptation 1 of the scenario Fig. 9a.



(c) Adaptation 2 of the scenario Fig. 9a.

Fig. 9 An adaptive co-simulation scenario with two adaptations

5.2.1 Debugging algorithm errors

When an algorithm is deemed incorrect by UPPAAL, the user can understand the error by analyzing the provided counter-example. However, counter-examples can be hard to understand/debug for an average user unfamiliar with the model checking technique. The Scenario-Verifier mitigates this problem by interpreting and visualizing the counter-example as an animation, using a graphical representation familiar to most co-simulation practitioners. The animation visualizes the trace of the simulation leading to the error. The trace shows the co-simulation state during the simulation and some metadata describing the simulation time and orchestration algorithm that is being verified. The trace also describes a set of possible actions—the SU actions that are currently enabled. A deadlock/counter-example is reached if the orchestrator tries to perform a non-enabled action. Figure 10 shows a counter-example.

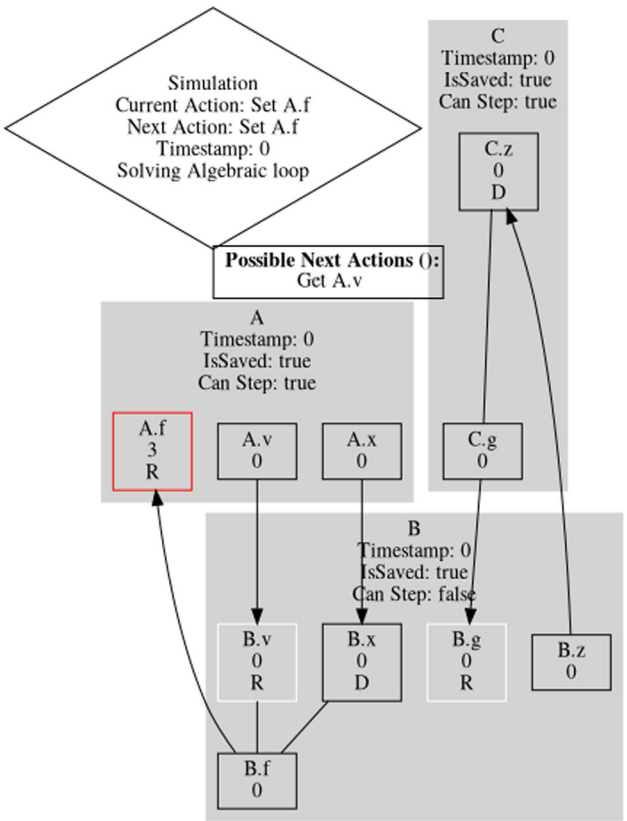


Fig. 10 An algorithmic error highlighted by the animation. The error is found in an algorithm simulating the scenario in Fig. 8a

6 Synthesizing Orchestration Algorithms

One challenge is to verify that an algorithm P of a given scenario $\mathcal{S}$ is the optimal algorithm to simulate $\mathcal{S}$, another challenge is to find P based on $\mathcal{S}$.

This section addresses the synthesizing challenge for simple, complex, and adaptive scenarios. All algorithms in the paper have been synthesized using the described approach by the Scenario-Verifier.

We start by summarizing the previous work in this domain. Gomes et al. [17] present a graph-based approach where orchestration algorithms can be extracted as a topological order.

Definition 19 *(Step Operation Graph)* Given a co-simulation scenario $\langle C, L, M, F, R\rangle$, we define the step operation graph where each node represents an operation $\texttt{set}_c(\_, u_c, \_)$, $\texttt{step}_c(\_, H)$, or $\texttt{get}_c(\_, y_c)$, of some SU $c \in C$, $y_c \in Y_c$, and $u_c \in U_c$. The edges are formed by the following rules:

R1 For each $c \in C$ and $u_c \in U_c$, if $L(u_c) = y_d$, add an edge $\texttt{get}_d(\_, y_d) \rightarrow \texttt{set}_c(\_, u_c, \_)$;

R2 For each $c \in C$ and $y_c \in Y_c$, add an edge $\texttt{step}_c(\_, H) \rightarrow \texttt{get}_c(\_, y_c)$;

R3 For each $c \in C$ and $u_c \in U_c$, if $R_c(u_c) = \texttt{true}$, add an edge $\texttt{set}_c(\_, u_c, \_) \rightarrow \texttt{step}_c(\_, H)$;

R4 For each $c \in C$ and $u_c \in U_c$, if $R_c(u_c) = \texttt{false}$, add an edge $\texttt{step}_c(\_, H) \rightarrow \texttt{set}_c(\_, u_c, \_)$;

R5 For each $c \in C$ and $(u_c, y_c) \in F_c$, add an edge $\texttt{set}_c(\_, u_c, \_) \rightarrow \texttt{get}_c(\_, y_c)$.

The rule R4 does not represent a data dependency; it is used to ensure that the synthesized algorithm respects the *delayed* inputs. The approach works for simple scenarios, but not for complex scenarios since it does not address the challenge of finding a correct valuation. We show how to

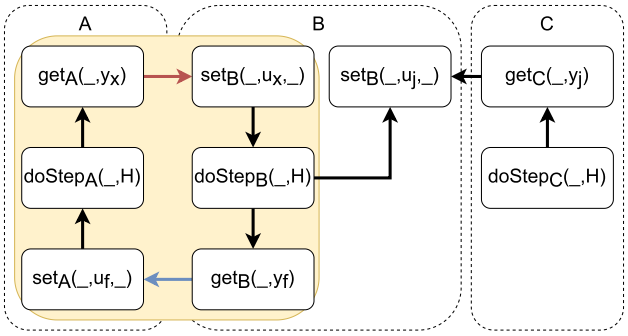


Fig. 11 Step operation graph of the scenario in Fig. 8b

extend the approach to complex and adaptive scenarios in the next section.

6.1 Synthesizing algorithms for complex scenarios

This section addresses the synthesis of algorithms for complex scenarios. Orchestration algorithms for complex scenarios should find the correct valuation using an iterative algorithm as discussed in Sect. 4 to satisfy the constraints of the scenario.

There exist two different kinds of complex scenarios: algebraic loops and step negotiation. We start by describing how we can use the same approach to synthesize algorithms for both before addressing their differences.

We extend the graph-based approach to synthesize algorithms for complex scenarios by ensuring that the iterative algorithms can be identified as non-trivial strongly connected components (SCC) in the graph. We use Tarjan's algorithm [27] to find both SCCs and their topological order in linear time.

Once a non-trivial SCC is discovered, the next challenge is transforming the non-trivial SCC into an algorithm. The general technique is to make the non-trivial SCC into a directed acyclic graph (DAG) by removing specific edges. We start with addressing algebraic loop solvers before addressing step negotiation; finally, we address adaptive scenarios.

6.1.1 Algebraic loops

A non-trivial SCC in the graph of Definition 19 highlights an algebraic loop. An example of an algebraic loop in the graph is shown in Fig. 11, the non-trivial SCC is highlighted in yellow. The non-trivial SCC highlights the set of operations that must be iteratively executed to solve the algebraic loop. Even though there exists two different kinds of algebraic loops we describe them as one, since the same approach can be used to synthesize algorithms for both.

To generate the operations that comprise each iteration of the algebraic loop solver, we make the graph of the non-trivial SCC into a DAG using reductions. A reduction is a technique for reducing the number of edges in the non-trivial SCC $\langle V, E\rangle$, where V is the vertices of the SCC and E is the edges.

We can remove edges from the SCC graph by substituting these data dependencies with informed guesses. This corresponds to the technique described in Sect. 4 with the valuation that contains an informed guess for all inputs in an algebraic loop.

Definition 20 *(Non-trivial SCC Reduction)* Given a non-trivial SCC $SCC = \langle V, E\rangle$ in the graph of Definition 19, we define the reduction of SCC as the set of edges E_R such that S can be reduced to a DAG by removing E_R.

The Scenario-Verifier supports two different reduction schemes: maximal and minimal. A reduction scheme r is maximal if it removes all edges created by couplings that are a part of an algebraic loop:

$$\langle V, E\rangle \xrightarrow{r} \langle V, E \setminus E_R\rangle \implies$$
$$E_R \subseteq E \wedge (dom(E_R) \cup ran(E_R)) \subseteq algebraic_{\mathcal{S}}$$
$$\wedge\, (dom(E) \cup ran(E)) \cap algebraic_{\mathcal{S}} = \emptyset$$

A reduction scheme r is minimal if it removes the fewest number of edges to make the graph into a DAG. A minimal reduction would, for example, only remove the blue edge from the non-trivial SCC in Fig. 11. The maximal reduction would, in contract, remove both the red edge and blue edge from the non-trivial SCC in Fig. 11.

Once we have made the graph into a DAG, we transform it into an algorithm by topologically sorting the graph and transforming each vertex into the action it represents. The synthesizing algorithm is shown in Algorithm 6.

6.1.2 Step negotiation

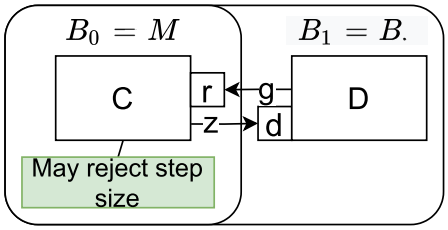
Step negotiation performs an iterative search for a step duration that all SUs accept. The technique is described in detail in Sect. 4.2.

A step rejection can affect more than the rejecting SU. All SUs stepped before the step rejection need to be backtracked. We keep track of the SUs that should be backtracked in case of a step rejection in the set B. The set B is calculated using the following recurrence relation:

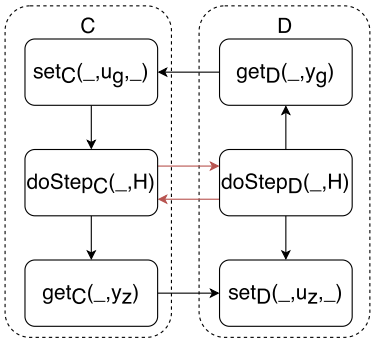
$$B_0 = M \tag{6}$$
$$B_{j+1} = B_j \cup \{c \mid d \in B_j \wedge c \notin B_j \wedge R(u_d) \wedge y_c \in Y_c \wedge L(u_d) = y_c\} \tag{7}$$

The set B is in each iteration updated such that all SUs $c \notin B$ having a reactive coupling to an input of SU d where

Fig. 12 A scenario needing step negotiation



(a) Co-simulation scenario.



(b) The extended step operation graph. The red edges are added by Def 24.

$d \in B$ is added to the set. Figure 12a graphically presents the process for calculating B. The update process continues until a fixed point is reached.

The graph in Definition 19 is extended with the rules in Definition 21 to incorporate that all SUs in the set B should be a part of the step negotiation.

Definition 21 *(Step Negotiation extension of Definition 19)* The step operation graph is extended with the following edges:

R6 For each $\mathtt{c} \in C$ and $\mathtt{d} \in C$ where $\mathtt{c} \neq \mathtt{d}$ and $\mathtt{c} \in B$ and $\mathtt{d} \in B$ add an edge from $\mathtt{step}_c(\_, H) \rightarrow \mathtt{step}_d(\_, H)$;
R7 For each $\mathtt{c} \in C$ and $\mathtt{d} \in C$ where $\mathtt{c} \in B$ and $\mathtt{d} \notin B$ add an edge from $\mathtt{step}_c(\_, H) \rightarrow \mathtt{step}_d(\_, H)$.

Rule R6 introduces a non-trivial SCC in the graph between all the SUs in B, making it possible to identify the need for step negotiation in the graph. Step negotiation should be performed if the graph contains a non-trivial SCC satisfying Definition 22. The rule R7 ensures that the step negotiation procedure is executed first in the co-simulation step. Executing it first reduces the number of SUs that need to be restored if an SU rejects a step.

The graph of scenario Fig. 6b on page 16 evolves to the cyclic graph presented in Fig. 12b by applying the rules from Definition 21.

Definition 22 *(Step-finding SCC)* A step-finding SCC is an SCC in the extended step operation graph with an edge from one $\mathtt{step}_c(\_, \_)$ node to another $\mathtt{step}_d(\_, \_)$ node.

The step negotiation procedure is derived from the non-trivial SCC satisfying Definition 22 by removing the edges introduced by Definition 21.

This yields a new graph equivalent to the original graph but with the edges removed. The new graph can either be a DAG or a cyclic graph. If the graph is a DAG, the graph is topologically sorted, and the algorithm is derived from the topological sort. Then, each vertex is transformed into the action it represents. If the graph is cyclic, we have identified a nested complex scenario. An algorithm for a nested complex scenario is derived by synthesizing the step negotiation procedure and then creating the fixed-point iteration procedure of the algebraic loop inside the step negotiation as seen in Algorithm 6 that shows a generic algorithm for synthesizing orchestration algorithms. The *save* and *restore* operations are omitted in the algorithm for clarity.

Algorithm 6 Synthesizing an Orchestration Algorithm for scenario $\mathcal{S}$.

```
1:  graph ← createGraph(S)                          ▷ Create the graph from the scenario.
2:  sccs ← tarjan(graph)                            ▷ Obtain a topological ordered set of SCCs.
3:  A ← []                                          ▷ Create an empty list for the algorithm.
4:  for all scc ∈ sccs do
5:    switch scc do
6:      case StepFindingSCC(scc)                    ▷ The SCC is a step-finding SCC.
7:        scc_R ← removeEdges(scc)                  ▷ Remove edges added by Def 21.
8:        sccs_R ← tarjan(scc_R)                    ▷ Sort the reduced scc.
9:        if nonTrivial(sccs_R) then
10:         scc ← sccs_R                            ▷ The SCC shows an algebraic loop.
11:         goto 18
12:       else                                      ▷ The SCC is not a nested complex scenario.
13:         for all act ∈ sccs_R do                 ▷ Add the actions to the algorithm.
14:           A ← Append(A, actionOf(act))
15:         end for
16:       end if
17:     case AlgebraicLoop(scc)                     ▷ The SCC is an algebraic loop.
18:       scc_R ← reduce(scc)                       ▷ Reduce the SCC by reduction.
19:       sccs_R ← tarjan(scc_R)                    ▷ Sort the reduced scc.
20:       for all act ∈ sccs_R do                   ▷ Add the actions to the algorithm.
21:         A ← Append(A, actionOf(act))
22:       end for
23:     case _                                      ▷ The SCC is trivial.
24:       A ← Append(A, actionOf(scc))              ▷ Add the action to the list.
25: end for
```

The synthesizer uses a topological order of the SCCs to create the algorithm. The topological order is not necessarily unique, which means that Algorithm 6 can synthesize multiple algorithms for the same scenario. All the synthesized algorithms are correct and will lead to the same simulation. Nevertheless, different algorithms can still lead to different performance and memory usage. This aspect can make one algorithm more suitable than another.

6.2 Synthesizing algorithms for adaptive scenarios

An adaptive scenario can, as described in Sect. 5 on page 26, be regarded as multiple distinct scenarios, where each adaptation is a *distinct* scenario that should be simulated using *its orchestration algorithm*. Therefore, we can use the above techniques to synthesize algorithms for adaptive scenarios by treating each adaptation separately.

6.3 Verification of synthesized algorithms

The Scenario-Verifier uses the above techniques to synthesize implementation-aware orchestration algorithms for simple, complex, and adaptive scenarios. The Scenario-Verifier can both synthesize and verify orchestration algorithms enabling us to use the UPPAAL model described in Sects. 5 to 3 to verify the synthesized algorithms automatically. This connection between the synthesizer and verifier enables extensive testing of both techniques. A scenario-generator was created to generate hundreds of test scenarios, which we could use to test the techniques. All generated algorithms are correct by construction.

The Scenario-Verifier has been integrated into the orchestration engine Maestro 2 [15] to enable co-simulation practitioners to simulate their scenarios using an implementation-aware orchestration algorithm. We choose Maestro 2 as the orchestration engine due to its performance and extensibility, allowing users to provide their own orchestration algorithms.

7 Case study

This section describes two case studies where the quality of the simulation result relies on the orchestration algorithm. The case studies have been simulated in Maestro V2 using a *synthesized* and *verified* orchestration algorithm from the Scenario-Verifier. The domain experts produced the simulations by describing the reactivity constraints/contracts of the co-simulation scenario and used the Scenario-Verifier to synthesize and verify the orchestration algorithm. They verified the simulation results against monolithic baselines. This section aims not to dive deep into different systems (readers interested in the systems are referred to the cited materials) but to show the applicability of the techniques.

7.1 Skyhook active suspension system

The first case study is a distributed model of a skyhook active suspension system. The suspension system consists of an automotive damper system that in real-time communicates with a quarter car model running on a server placed in a different location. The simulation is a part of a real-time environment where the damper model is substituted for a test bench, making it necessary to include special modules to compensate for communication delays [28]. Figure 13 shows the configuration of the system and the communication delay.

The scenario is represented in Fig. 14b as the five SUs from Fig. 13: quarter car, damper, delay, sender, and receiver. The *quarter car* and *damper* represents continuous components system, while the *sender*, *delay* and *receiver* are discrete components. The *sender* tries to forecast the state of the system using a model of the quarter car. The *receiver* builds up an interpolation table with possible values based on the predictions sent by the sender. In consequence, this leads to a scenario that is very susceptible to the orchestration algorithm since the *receiver* needs to be executed after the *sender* in order to compile the interpolation table. Similarly, the *damper* needs the interpolation table to predict a value and should therefore be executed after the *receiver*.

The simulation results are shown in Fig. 14a. The results were produced by Maestro 2 using a *synthesized* and *verified* orchestration algorithm created from the system configuration shown in Fig. 14b. The results show that the system can be simulated accurately if the orchestration algorithm respects the constraints of the scenario, which allows the *sender* to forecast the state of the system and the *receiver* to build up an interpolation table. The code to perform the simulation is available at https://github.com/SimplisticCode/CosimulationCaseStudies/tree/master/journal_paper_experiments/delay_compensator_case_study.

7.2 Simplified full vehicle model: co-simulation of longitudinal and vertical dynamics models

The second example, a simplified full vehicle model, is decomposed into two subsystems: (i) longitudinal vehicle dynamics and (ii) vertical vehicle dynamics.

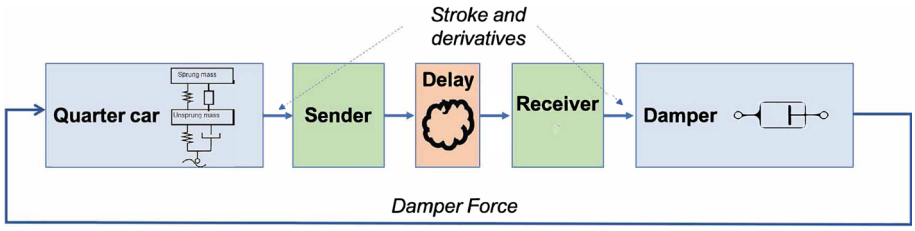
The longitudinal vehicle dynamics model is a simplified drivetrain model, see Fig. 15 with the parameters defined in Tables 1 and 2 on page 42.

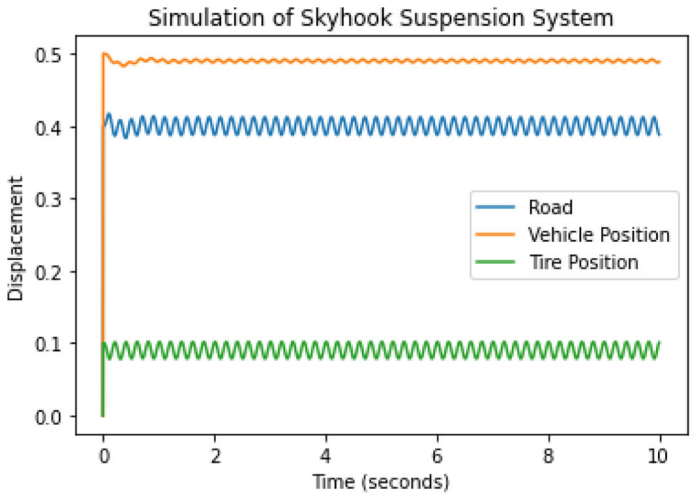
The torque on the wheels is translated into longitudinal forces by a linear tyre model formula:

$$F_w = T_w / r_w, \tag{8}$$

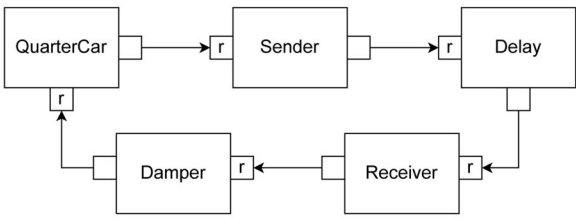
where F_w denotes the tyre forces at two wheels driving the vehicle ahead and T_w denotes the sum of the torque in left and right axles.

The vertical vehicle dynamics model is a linear quarter car model combined with the road profile, see Fig. 16. The parameters of the vertical vehicle dynamics model are defined in Table 3 on page 43. The road profile is a periodic bumpy road with a smooth surface. The front and rear suspensions are excited by the vertical displacement, z_g, and velocity, $\dot{z}_g$.





(a) Simulation results.



(b) The Skyhook scenario.

Fig. 13 Skyhook active suspension system

Fig. 14 Simulation of the Skyhook active suspension system

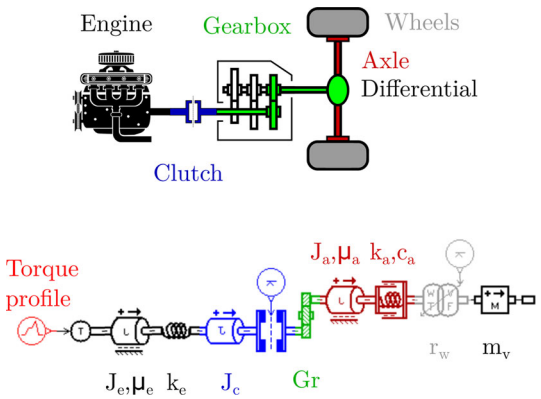


Fig. 15 Longitudinal vehicle dynamics model

The suspension is also sprung by the collision of the wheels to the speed bump, see Fig. 17a. The vertical component of the inelastic collision force, F_N, applied directly to the unsprung mass

$$F_N = 0.7 * (2 * m_w) * v * cos(\theta) \tag{9}$$

where 0.7 is the elasticity coefficient of the collision, and v is the longitudinal speed of the vehicle. The horizontal reaction force which is propagated back to the drivetrain model is the superposition of the horizontal component of the inelastic collision force is applied back to the drivetrain model and the gravitational force due to the inclination of the speed bump:

$$F_R = 0.7 * (2 * m_w) * v * sin(\theta) + \left(\frac{m_v + 2 * m_w}{2}\right) * g * sin(\theta) \tag{10}$$

where $g = 9.81 m/s^2$ denotes the gravity.

The scenario is similar to the scenario in Fig. 1 on page 2. The vertical dynamics depend on the longitudinal dynamics, which means that it needs to be coupled with the longitudinal dynamics using a *delayed* connection, allowing the vertical dynamic to react to the longitudinal dynamic. The reaction force caused by the bumps instantly changes the vertical velocity of the vehicle, which we model using a *reactive* connection. This means that scenario

Given the mentioned conditions and system parameters, we used the Scenario-Verifier to *synthesize* and *verify* an orchestration algorithm which we have used to simulate the system using Maestro 2 with a fixed step duration of 0.001s, and each FMU is solved by the variable step 4th order Runge–Kutta solver.

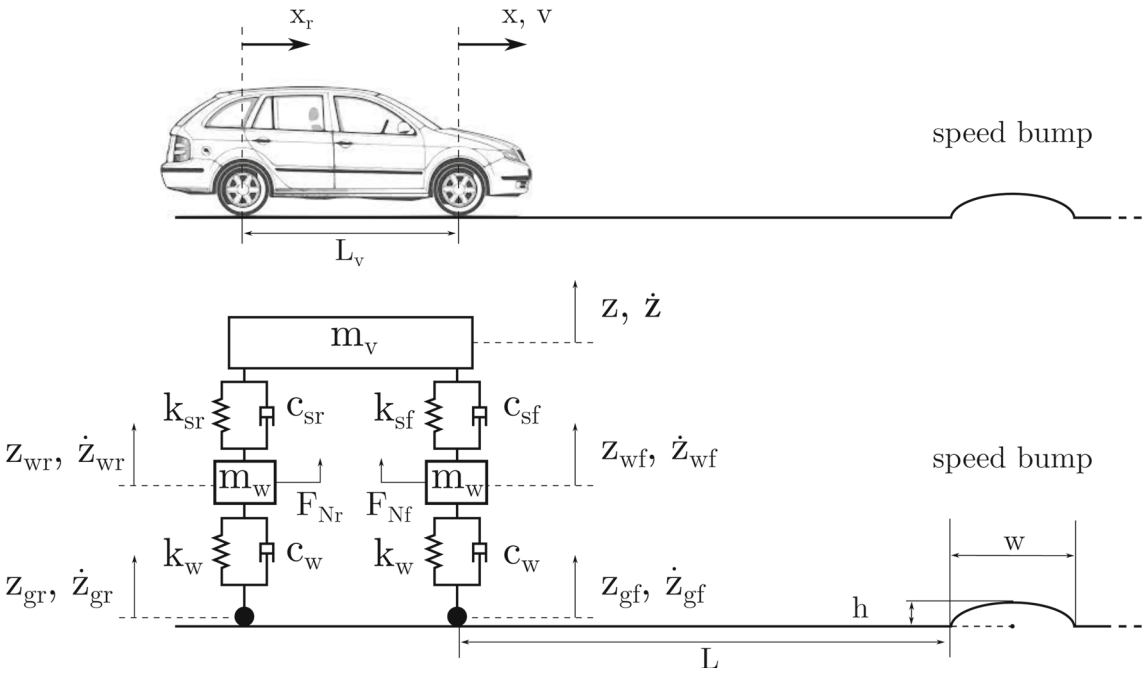
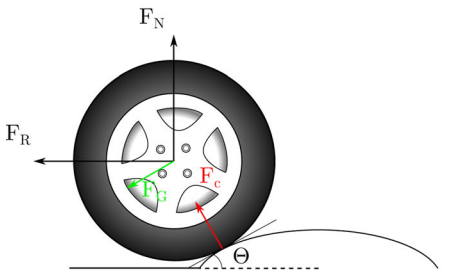
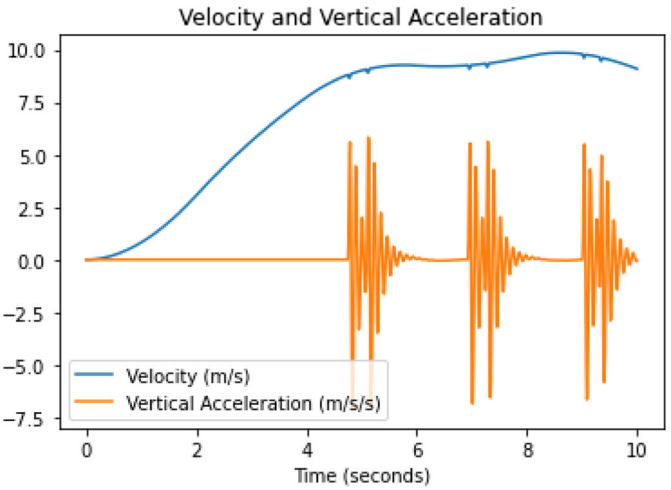


Fig. 16 Vertical vehicle dynamics model



(a) The interaction between the wheels and the speed bump.



(b) Simulation results of the simplified vehicle model.

Fig. 17 Simulation of the simplified vehicle model

The simulation results are shown in Fig. 17b where we can see how the speed bumps affect the vertical acceleration and velocity of the car. The synthesized orchestration algorithm ensures an instant change in the vertical acceleration once the vehicle collides with the speed bump. The vertical acceleration is then propagated back to the longitudinal dynamics, thereby changing the vehicle's velocity. Therefore, the scenario is a clear example of a scenario where the reactivity constraints are necessary to ensure an accurate simulation.

8 Related work

The semantics of co-simulation was studied in [13,14,16, 17,29]. The paper [17] formally describes FMI-based co-

simulation scenarios and places several correctness criteria on the co-simulation algorithms to generate and verify them. In addition, this paper extends their work by treating co-simulation scenarios subject to algebraic loops and adaptive steps. Thule et al. [30] studied how a co-simulation scenario's characteristics can be used to choose the correct simulation strategy for a given co-simulation algorithm. Orchestration algorithms of complex scenarios are described and synthesized in [13]. However, the paper lacks verification of these complex orchestration algorithms.

Broman et al. [16] place constraints on the co-simulation scenario to avoid algebraic loops and achieve deterministic co-simulation results. Furthermore, they propose a generic orchestration algorithm for handling step negotiation. However, their generic algorithm does not solve either algebraic loops or the contracts of the scenario. This manuscript deals with all these aspects.

Formal methods have previously been successfully used in the area of co-simulation [12,31–33]. Amálio et al. [31] 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. [12] claim to provide the first behavioral semantics of FMI. Their paper shows how to prove essential properties of master algorithms, like termination and determinism. Furthermore, they show that the example provided in the FMI standard is not a valid algorithm. The paper [32] 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 [33] use UPPAAL to analyze controller-based systems with FMUs and a master algorithm modeled in UPPAAL as a TA. However, they use UPPAAL quite differently; they focus on the controller and not the co-simulation algorithm. Palmieri et al. in [34] have used UPPAAL to provide sound guarantees on the interleaving between a graphical user interface and a generic FMI master algorithm.

The paper extends its predecessors [13,14] by treating adaptive co-simulation scenarios and providing synthesis and verification of orchestration algorithms in an integrated setting. Furthermore, the paper describes two case studies where the techniques have been used on real co-simulation scenarios.

9 Concluding remarks

We showed how to synthesize and verify orchestration algorithms for co-simulation scenarios with algebraic loops, step rejections, adaptive couplings, and reactivity constraints.

The approaches have been implemented in a tool called the *Scenario-Verifier*, which has been integrated with the orchestration engine Maestro 2 to establish, to the best of our knowledge, one of the first connections between a formal tool and an orchestration engine running real-world scenarios.

The applicability of the formal approaches in a practical context was explored using two case studies, which were simulated in Maestro 2 using a verified and synthesized orchestration algorithm from the Scenario-Verifier.

A future work agenda includes formalizing the FMI 3.0 standard and integrating the Scenario-Verifier with other orchestration engines. We will also examine whether it is possible to synthesize an optimal orchestration algorithm for a given co-simulation scenario.

Acknowledgements 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. We are would also like to thank the anonymous reviewers of the paper, who have provided valuable feedback on the paper.

Appendix A: Table of conventions

This appendix contains a table describing the notation used throughout the paper. Capitalized letters refer to sets, while lower case letters refer to a variable belonging to the set represented by the capitalized letter.

Convention	Description
U	All Inputs of the Scenario
U_c	Inputs of the SU c
Y	All Outputs of the Scenario
U_c	Outputs of the SU c
S	States
$s_c^{(t)}$	State of SU c at time t
$\mathcal{V}_{\mathcal{T}}$	Time stamped values of the type $\mathcal{V} \times \mathbb{R}_{\geq 0}$
s_c^R	The abstract state of SU c
t	Time t ($t \in \mathbb{R}_{\geq 0}$)
H	Step duration H ($H \in \mathbb{R}_{>0}$)
L	Couplings between SUs
F	Feed-through constraints
R	Reactivity constraints
C	A set of SU identifiers
$\mathcal{A}$	Adaptations
M	A set of SUs that may reject a step duration
B	A set of SUs that must be backtracked

Appendix B: BNF grammar

The section presents the domain-specific language where user can describe co-simulation algorithms and scenarios for both simple, complex, and adaptive co-simulation scenario.

Examples of algorithms and scenarios described using the DSL are available online https://github.com/INTO-CPS-Association/Scenario-Verifier/tree/master/src/test/resources .

Listing 1 BNF Grammar to orchestration algorithms.

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

⟨*SU-action*⟩ ::= ⟨*get :* <*SUID*⟩.⟨*OId*⟩> | ⟨*set:* <*SUID*⟩.⟨*IId*⟩> | ⟨*step:* <*SUID*⟩> | ⟨*restore-state:* <*SUID*⟩> | ⟨*save-state:* <*SUID*⟩>

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

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

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

⟨*algebraic*⟩ ::= 'until-converged:' '['⟨<*SUID*⟩.⟨*Oid*⟩>*']' 'iterate:' '['⟨*SU-action*⟩*']' 'if-retry-needed:' '['⟨*restore-state:* <*SUID*⟩>*]'

Listing 2 BNF Grammar for the specification of Adaptive scenarios.

⟨*Scenario*⟩ ::= 'fmus=' '{' ⟨*SU*⟩+ '}' 'configuration= {' ⟨*configuration*⟩ '}' 'connections=' '[' ⟨*Connection*⟩* ']'

⟨*SU*⟩ ::= ⟨*SUID*⟩ '= {' ⟨'*can-reject-step* =' *Bool*⟩?',' 'inputs = {' ⟨*Input*⟩* '}' ',' 'outputs = {' ⟨*Output*⟩* '}' '}'

⟨*configuration*⟩ ::= 'configurable-inputs = ['⟨*SUID*⟩.⟨*IID*⟩* ']', 'configurations = {' ⟨*config*⟩* '}'

⟨*config*⟩ ::= 'inputs = {' ⟨*Input*⟩* '}' 'cosim-step=' ⟨*StepId*⟩ 'connections= [' ⟨*Connection*⟩* ']'

⟨*Input*⟩ ::= ⟨*IId*⟩ '= {reactivity: <Contract>'}'

⟨*Output*⟩ ::= ⟨*OId*⟩ '= {dependencies-init=['⟨*IId*⟩*'], dependencies=['⟨*IId*⟩*']}'

⟨*Connection*⟩ ::= ⟨*SUID*⟩.⟨*OId*⟩ '->' ⟨*SUID*⟩.⟨*IID*⟩

⟨*Contract*⟩ ::= delayed | reactive

Appendix C: Algorithm of nested complex scenario

The co-simulation step of the scenario in Fig. 8a on page 26.

Algorithm 7 Step procedure containing fixed-point iteration inside step negotiation procedure for simulating the scenario in Fig. 8a.

SaveSUs in $B \cup K$
$h \leftarrow H_{max}$
while !*Step_found* **do** ▷ Step negotiation
  while !conv **do** ▷ Fixed-point Iteration
    $s_A^{(s)} \leftarrow \mathtt{set}_A(s_A^{(s)}, u_f, f_{val})$
    $s_B^{(s)} \leftarrow \mathtt{set}_B(s_B^{(s)}, [u_v, u_g], [v_{val}, g_{val}])$ ▷ Setting v and g
    $(s_C^{(s+h_C)}, h_C), \leftarrow \mathtt{step}_C(s_C^{(s)}, h)$
    $(s_B^{(s+h_B)}, h_B), \leftarrow \mathtt{step}_B(s_B^{(s)}, h)$
    $(s_A^{(s+h_A)}, h_A), \leftarrow \mathtt{step}_A(s_A^{(s)}, h)$
    $(v_{vala}, x_{val}) \leftarrow \mathtt{get}_A(s_A^{(s+h_A)}, [y_v, y_X])$ ▷ Getting v and x
    $z_{val} \leftarrow \mathtt{get}_B(s_B^{(s+h_B)}, y_z)$
    $s_C^{(s+h_C)} \leftarrow \mathtt{set}_C(s_C^{(s+h_C)}, u_z, z_{val})$
    $g_{vala} \leftarrow \mathtt{get}_C(s_C^{(s+h_C)}, y_g)$
    $s_B^{(s+h_B)} \leftarrow \mathtt{set}_B(s_B^{(s+h_B)}, u_x, x_{val})$
    $f_{vala} \leftarrow \mathtt{get}_B(s_B^{(s+h_B)}, y_F)$
    $conv \leftarrow CheckConv((g_{vala}, g_{val}), (v_{vala}, v_{val}), (f_{vala}, f_{val}))$
    if !*conv* **then**
      RestoreSUs in K
    end if
    $(g_{val}, v_{val}, f_{val}) \leftarrow (g_{vala}, v_{vala}, f_{vala})$
  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**
    RestoreSUs in B
  end if
end while

Appendix D: Parameters of the full vehicle model

Table 1 Parameters of the longitudinal vehicle model

Constant	Description	Value
J_e	Inertia of Engine shaft	0.155kgm^2
J_c	Inertia of the clutch	0.02kgm^2
J_a	Inertia of the driving front axle	0.132kgm^2
k_e	Torsional stiffness of the engine shaft	10^6Nm/degree
k_a	Torsional stiffness of the driving front axle	10^6Nm/degree
μ_e	Damping of the engine shaft	0.2Nm/(rev/min)
μ_a	Damping of the axle	0.25Nm/(rev/min)
c_a	Damper rating at the end of the axle	0.02Nm/(rev/min)
r_w	Radius of a 205/55R16 tyre [35]	0.32m
m_v	Mass of the vehicle	1350kg

Table 2 Torque profile

Torque [Nm]	Time [s]
0	0.0
400	2.0
400	4.0
200	6.0
300	8.0
100	10.0

Table 3 Parameters of the vertical vehicle model

Constant	Description	Value
k_{sf}	Front suspension stiffness	$2 \times (15x10^3)$
c_{sf}	Front suspension damping	$2 \times (1700)$Ns/m
k_{sr}	Rear suspension stiffness	$2 \times (15x10^3)$
c_{sr}	Rear suspension damping	$2 \times (1500)$Ns/m
m_w	Unsprung mass	2×70kg
k_w	Tyre stiffness	$2 \times (2 \times 10^5)$N/m
c_w	Tyre damping	$c_w = 2 \times (100)$Ns/m
L_v	Distance between the front and rear wheels	$3m$
L	Length of periodic road of the axle	19.5m
w	Width of the speed bump	0.5m

References

1. Lee, E.A.: UNKNOWN (ed.) Cyber physical systems: Design challenges. (ed.UNKNOWN) International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC). IEEE, Los Alamitos, CA, USA (2008)
2. Blockwitz, T., et al.: Functional Mockup Interface 2.0: The Standard for Tool independent Exchange of Simulation Models. In: Otter, M., Zimmer, D. (eds.) Proceedings of 9th International Modelica Conference, pp. 173–184. Linköping University Electronic Press, Linköping (2012)
3. Kübler, R., Schiehlen, W.: Two methods of simulator coupling. Math. Comput. Model. Dyn. Syst. **6**(2), (2000)
4. Gomes, C., Broman, D., Vangheluwe, H., Thule, C. & Larsen, P. G. Co-simulation: a survey. *ACM Computing Surveys* **51** (3): (2018)
5. FMI. Functional mock-up interface tools (2014). https://fmi-standard.org/tools/
6. 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: Schops, S., Bartel, A., Gunther, M., ter Maten, E.J.W., Muller, P.C. (eds.) Progress in Differential-Algebraic Equations. Springer, Berlin, Heidelberg (2014)
7. Gomes, C., et al.: HintCO—hint-based configuration of co-simulations. In: Obaidat, M.S., Ören, T.I., Szczerbicka, H. (eds.) Proc. Simultech'19. SciTePress, Setubal, Portugal (2019)
8. Oakes, B.J., et al.: Hint-based configuration of co-simulations with algebraic loops. In: Obaidat, M., Obaidat, M., Obaidat, M., Ören, T., Szczerbicka, H. (eds.) Proc. Simultech'19, Vol. 1260 of Advances in intelligent systems and computing. Springer, Setubal, Portugal (2020)
9. Gomes, C., Thule, C., Lausdahl, K., Larsen, P.G., Vangheluwe, H., Mazzara, M., Ober, I., Salaün, G. (eds).: Stabilization technique in INTO-CPS. Mazzara, M., Ober, I., Salaün, G. (eds.), Proc. 2nd Workshop on Formal Co-Simulation of Cyber-Physical Systems, Vol. 11176 of LNCS, Springer, Cham (2018)
10. 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)
11. Gomes, C., et al.: Semantic adaptation for FMI co-simulation with hierarchical simulators. J. Simul. **95**(3), 241–269 (2019)
12. Cavalcanti, A., Woodcock, J., Amálio, N. Sampaio, A., Wang, F. (eds.), Behavioural models for FMI co-simulations. (eds Sampaio, A. & Wang, F.) Proc. ICTAC'16, Vol. 9965 of LNCS Springer, Cham (2016)
13. Hansen, S.T., Gomes, C., Larsen, P.G., van de Pol, J., Martin, C.R., Blas, M.J., Inostrosa-Psijas, A. (eds.), Synthesizing co-simulation algorithms with step negotiation and algebraic loop handling. In: Martin, C.R., Blas, M.J., Inostrosa-Psijas, A., (eds.), Proc. Annual Modeling and Simulation Conference (ANNSIM'21), IEEE, Virginia, USA, (2021)
14. Hansen, S.T., et al.: Verification of co-simulation algorithms subject to algebraic loops and adaptive steps. In: Lluch Lafuente, A., Mavridou, A. (eds.) Proc. FMICS'21, Vol. 12863 of LNCS. Springer, Cham (2021)
15. Thule, C., Lausdahl, K., Gomes, C., Meisl, G., Larsen, P.G. Maestro: The INTO-CPS co-simulation framework. Simulatio Modelling Practice and Theory **92** (2019). https://www.sciencedirect.com/science/article/pii/S1569190X1830193X
16. Broman, D. et al.: Determinate composition of FMUs for co-simulation. In: Ernst, R., Sokolsky, O. (eds.), Proc. EMSOFT'13, IEEE, (2013)
17. Gomes, C., Thule, C., Lúcio, L., Vangheluwe, H., Larsen, P.G., Camara, J., Steffen, M. (eds): Generation of co-simulation algorithms subject to simulator contracts. In: Camara, J., Steffen, M. (ed.), Proc. SEFM'19 Collocated Workshops, Vol. 12226 of LNCS, Springer, Cham (2020)
18. Clarke, E.M., Jr., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge, MA, USA (1999)
19. Baier, C., Katoen, J.-P.: Principles of Model Checking. The MIT Press, Cambridge, Mass (2008)
20. Behrmann, G. et al.: UNKNOWN (ed.) UPPAAL 4.0. (ed.UNKNOWN) Third International Conference on Quantitative Evaluation of Systems (QEST 2006), Springer, (2006)
21. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. **126**(2), (1994)
22. Bérard, B., et al.: UPPAAL—Timed systems. In: Bérard, B., et al. (eds.) Systems and Software Verification: Model-Checking Techniques and Tools. Springer, Berlin, Heidelberg (2001)
23. Hansen, S.T., Thule, C., Gomes, C. Cleophas, L., Massink, M. (eds.), An FMI-Based Initialization Plugin for INTO-CPS Maestro 2. In: Cleophas, L., Massink, M. (eds.), Proc. SEFM'20 Collocated Workshops, Vol. 12524, Springer, Cham (2020)
24. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Communications of the ACM **18**(8), (1975)
25. Cellier, F.E., Kofman, E.: Continuous System Simulation. Springer, New York (2010)
26. Inci, E.O. et al.: The effect and selection of solution sequence in co-simulation. In: Martin, C.R., Blas, M.J., Inostrosa-Psijas, A. (eds.), Proc. Annual Modeling and Simulation Conference (ANNSIM'21), IEEE, Virginia, USA (2021)

27. Tarjan, R.E.: Depth-first search and linear graph algorithms. SIAM J. Comput. **1**(2), 146–160 (1972)
28. Alfonso, J. et al.: Distributed simulation and testing for the design of a smart suspension. SAE International Journal of Connected and Automated Vehicles **3**(2), (2020)
29. Gomes, C., Lucio, L., Vangheluwe, H., Burgueño, L. et al.: (eds) Semantics of co-simulation algorithms with simulator contracts. In: Burgueño, L. et al. (eds.), Proc. ACM/IEEE MODELS'19, IEEE (2019)
30. Thule, C., et al.: Towards the verification of hybrid co-simulation algorithms. In: Mazzara, M., Ober, I., Salaün, G. (eds.) Proc. STAF'18 Collocated Workshops, Vol. 11176 of LNCS. Springer, Cham (2018)
31. Amálio, N., Payne, R.J., Cavalcanti, A., Woodcock, J. Ogata, K., Lawford, M., Liu, S.: Checking SysML models for co-simulation. In: Ogata, K., Lawford, M., Liu, S. (eds.), Proc. ICFEM'16, Vol. 10009 of LNCS Springer, Cham (2016)
32. Zeyda, F., Ouy, J., Foster, S., Cavalcanti, A. Cerone, A., Roveri, M.: Formalising cosimulation models. In: Cerone, A., Roveri, M. (eds.), Proc. SEFM'17 Collocated Workshops, Vol. 10729 of LNCS Springer, Cham (2017)
33. Jensen, P.G., Larsen, K.G., Legay, A., Nyman, U. UNKNOWN (ed.): Integrating tools: Co-simulation in UPPAAL using FMI-FMU. (ed.UNKNOWN) Proc. ICECCS'17, IEEE, Fukuoka (2017)
34. Palmieri, M., Bernardeschi, C., Masci, P.: A framework for FMI-based co-simulation of human-machine interfaces. Softw. Syst. Model. **19**(3), (2020)
35. Tire size calculator (2021). https://tiresize.com/calculator/