Monitoring Reconfigurable Simulation Scenarios in Digital Twins

Simon Thrane Hansen[1], Eduard Kamburjan[2], and Zahra Kazemi[1]

[1] Department of Electrical and Computer Engineering, Aarhus University, Denmark
[2] Department of Informatics, University of Oslo, Norway

Abstract. Co-simulation is necessary for many domains, as it allows for simulating systems comprised of heterogenous sub-systems developed in isolation by combining them into a scenario using a specific coordination strategy called an orchestration algorithm. Co-simulation use in digital twins has led to a growing interest in reconfigurable scenarios, where the scenario changes dynamically during the simulation. However, reconfigurable scenarios introduce new challenges, as the orchestration algorithm must be dynamically reconfigured and verified to ensure the simulation results' correctness despite the scenario changes. This paper presents a method to monitor reconfigurable scenarios developed in SMOL with a monitor implemented in UPPAAL to ensure the correctness of the orchestration algorithm at runtime. A case study from the power system domain demonstrates the approach.

Keywords: Runtime Verification, Dynamic Simulation, Digital Twins

1 Introduction

Co-simulation is a well-established technique used in a wide range of domains, including automotive, aerospace, and power systems, as it allows for simulating systems comprised of heterogenous sub-systems developed in isolation [15,24]. Each sub-system is typically represented by a simulation unit (SU), responsible for computing the behavior of a single sub-system as a discrete trace and typically implementing a standardized interface, e.g., the widely adopted Functional Mock-up Interface (FMI) [8]. The SUs are composed into a *scenario* by connecting their interfaces to denote data dependencies between the SUs.

The behavior of the system is obtained by simulating the SUs in parallel and combining their traces according to the data dependencies between the SUs using specific coordination strategies called *orchestration algorithms* [24,15,20]. The orchestration algorithm is performed by centralized *orchestrator* that interacts with the SUs to exchange data and progress them in time. Although the FMI standard does not formalize the orchestration algorithm [9], it must be specifically tailored to the scenario, as a wrong orchestration algorithm can lead to hard-to-debug issues in the form of numerical inaccuracies and performance issues [20,28,30]. The orchestration algorithm must, among other, respect the

interaction structure expected by the SU, handle cyclic dependencies (algebraic loops) using a suitable numerical method [18], and negotiate an appropriate time step size between the numerical solvers within the SUs.

Traditionally, the scenario is fixed at design time and simulated in an environment for co-simulation [35]. Nevertheless, there is a growing interest in reconfigurable scenarios to enable (1) exploration different configurations of the system, e.g., to find the optimal configurations [29], and in the context of digital twins, (2) to account for the changes due to maintenance or other external factors to the physical system it represents. This requires a more dynamic approach to change the scenario and manipulate it in a general purpose language.

Reconfigurable scenarios introduce new challenges: The orchestration algorithm must be dynamically reconfigured and verified as well. However, current verification approaches to orchestration algorithms is limited to static scenarios [20]. Consequently, we propose a method combining the benefits of general purpose programming languages to reconfigure the scenario at runtime with the benefits of online verification using a approach to ensure the correctness of the orchestration algorithm throughout the simulation, thereby enabling the verification of reconfigurable scenarios. Under correctness we understand, beyond correctness of the orchestration algorithm, also that the orchestration algorithm indeed follows the scenario.

Simulation require high performance, and runtime monitoring adds an inevitable overhead to the system. However, co-simulations are typically running the same scenario in an iterative manner – if the first iteration of the simulation loop is correct, then results from these checks are liekly to be reusable for the later iterations. Our system, thus, results in an overhead for the first iteration after reconfiguration (or explorative setup), but is optimized to reuse verdicts and drastically reduces overhead for later iterations.

Contribution and Structure. We present a system to load simulation scenarios at runtime and (a) check properties of the scenario, as part of the digital twin, and (b) check that the orchestration algorithm follows the simulation scenario. This ensures the correct *reconfigurable simulation structures in digital twins.*

We implement our approach as (a) a library for online monitoring according to the FMI standard, and (b) an extension of the FMI-intergration [21] into the SMOL [22] programming language. We demonstrate the applicability of our approach by applying it to a power system, where we show how the orchestration algorithm can be reconfigured on the fly to account for changes in the power grid.

Section 2 motivates our work before Section 3 provides the necessary preliminaries. The monitor is presented in Section 4 and the monitoring library is presented in Section 5. Section 6 evaluates the approach, Section 7 discusses related work, and Section 8 concludes.

2 Motivating Example

We, first, illustrate the need for monitoring of reconfigurable simulation scenarios with an example and provide an overview over our approach.

Consider a co-simulation scenario of two prey-predator systems in a digital twin, where it serves as simulation model that is used to detect unexpected behavior. It follows the Lotka-Volterra equations, where each of the equations is simulated by an own simulation unit. Here, x_i is the number of predators in habitat i, y_i is the number of prey, and $\alpha_i, \beta_i, \gamma_i, \delta_i > 0$ are the paramters.

$$x_1' = \alpha_1 x_1 - \beta_1 x_1 y_1 \qquad y_1' = \delta_1 x_1 y_1 - \gamma_1 y_1$$
$$x_2' = \alpha_2 x_2 - \beta_2 x_2 y_2 \qquad y_2' = \delta_2 x_2 y_2 - \gamma_2 y_2$$

The system contains two algebraic loops, which must be considered by the orchestrator. Given the FMUs `prey`i and `predator`i, the (verifiable) scenario is given in Lst. 1.1. For brevity, we only give an excerpt for the first habitat.

The SMOL [22] program in Fig. 1.2 implements the scenario. It creates two wrapper objects with additional data per habitat, as well as implements the simulation driver. Despite the similarities in structure, it does not directly correspond to the simulation model – there is no notion of *connection* in the system, and the actions of the scenario are realized using imperative statements. Furthermore, the code is interleaved with book keeping for the convergence condition and can be embeded in further, object-oriented structures. It is, thus not obvious whether the given code follows the verified simulation model.

Digital twins are used throughout the whole lifecycle of a system, and must be reconfigured if the system changes. For example, one may discover that the prey populations are actually connected, and a small of migrations is possible in one direction. This changes the equations for the two prey populations y_1, y_2 to the following, where $c \geq 0$ is the number of migrations [31].

$$y_1' = \delta x_1 y_1 - \gamma y_1 + \frac{c}{y_1} \qquad y_2' = \delta x_2 y_2 - \gamma y_2 - \frac{c}{y_1}$$

Now, the simulation model must be adapted. First, the FMUs be replaced. Second, the master algorithm must be changed, as the overall systems now contain one algrebraic loop. Again, the challenge of correctness arises: does the algorithm implemented in the digital twin follow the (verified) scenario? The reconfigured system is not known at design time – it is constructed by self adaptating to changes in the simulation structure, and by loading a new simulation scenario that is developed during the use of the twin.

```
scenario = {
 fmus =        { prey1 = { ... }, pred1 = { ... } ... }
 connections =
   [ prey1.x -> pred1.x, pred1.y -> prey1.y, ... ] }
cosim-step = { [ {
 loop: { until-converged: [pred1.y, prey1.x]
   iterate: [ {get: prey1.x} {get: pred1.y}
              {set: prey1.y} {set: pred1.x}] }
   {step: pred1} {step: prey1} ] ... }
```

Listing 1.1: Example scenario for one habitat of prey-predator dynamics.

Our solution is to use use the automata-based model of the simulation scenario at runtime to (a) verify newly loaded simulation scenarios, and (b) ensure that the scenario is followed by the simulation units. To this end, the scenario becomes an own object at runtime, to which simulation units can be assigned. During simulation one, can then check which steps is possible. Note that the scenario file (`PreyPredator.conf`) may not be available at runtime yet, so the verification of the code must happen at runtime. Indeed, the new simulation code may be loaded at runtime as well, for example as part of a DT-as-a-Service platform [1,32]. The code in Fig. 1.3 is an exempt from a SMOL program that adapts the above code to runtime monitoring.

3 Preliminaries

This section introduces the FMI standard and the model checker UPPAAL.

3.1 Functional Mock-up Interface

The FMI standard [8] describes how to implement composable and implementation-independent FMUs. The FMUs are composed into a scenario to explore their global behavior, which is the system's behavior. We define the concepts of FMUs and scenarios similarly to [19].

Definition 1 (Simulation Unit). *An FMU 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: (i)* S_c *is a set denoting the state space of the FMU. (ii)* 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 FMU. (iii)* $\mathcal{V}$ *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. (iv) 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. (v)* $\texttt{step}_c : S_c \times \mathbb{R}_{>0} \to S_c \times \mathbb{R}_{>0}$ *is a function; it instructs the FMU to compute its state after a given duration. If an FMU 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 FMU at time* $t + h$, *where* $h \leq H$.

Definition 2 (Scenario). *A* scenario $\mathcal{S}$ *is a tuple* $\mathcal{S} \triangleq \langle C, L, M, R, F \rangle$, *where (i)* C *is a finite set (of FMU identifiers). (ii)* L *is a function* $L : U \to Y$, *where*

```
1 FMO[in Double y, out Double x] prey1 = simulate("Prey.fmu");
2 ...
3 while (i++ <= iterations) do
4   while /* convergence condition */ do
5     prey1.y = pred1.y; pred1.x = prey1.x;
6     /* book keeping */ end
7   prey1.tick(0.1); pred1.tick(0.1); end
```

Listing 1.2: Example implementation for one habitat of prey-predator dynamics.

```
1  FMO[in Double y, out Double x] prey1 = simulate("Prey.fmu");
2  prey1.role = "prey1";
3  ...
4  Scenario[prey, predator] scenario = monitor("PreyPredator.conf");
5  scenario.assign(prey1);
6  ...
7  while (i++ <= iterations) do
8   while /* convergence condition */ do
9    if(pred1.canReadY()) buf = pred1.y; else /* error handling */ end
10   if(prey1.canWriteY()) prey1.y = buf; else /* error handling */ end
11   ... end
12  ...
```

Listing 1.3: Monitoring one habitat.

$U = \bigcup_{c \in C} U_c$ *and* $Y = \bigcup_{c \in C} Y_c$*, and where* $L(u) = y$ *means that the output* y *is coupled to the input* u*. (iii)* $M \subseteq C$ *denotes the FMUs that implement error estimation. (iv)* $R : U \to \mathbb{B}$ *is a predicate describing the FMUs' input approximation functions.* $R(u) = true$ *means that the function* `step` *assumes that the timestamp* t_c *of the FMU* 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. More details on reactivity can be found in [16] (v)* 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 FMU.*

Orchestration Algorithms. An orchestrator simulates a scenario by employing an orchestration algorithm describing how the FMUs are coordinated in the simulation. The orchestration algorithm is a sequence of operations on the FMUs dictating when and how the different subsystems affect each other (by getting and setting port) and when an FMU can change its state within a given time horizon (by calling `step`).

We define the orchestration algorithm as a trace of the orchestrator's actions on the FMUs. Let π be a co-simulation trace, then π is a sequence of actions where each action is of the form $\pi_i = \alpha_i$, where α_i is one of the actions defined in Definition 1 for an FMU `i`. We define the empty co-simulation trace as $\emptyset$. An example of a co-simulation trace with three FMUs can be seen in Eq. (1).

$$\pi = \mathtt{step}_A.\mathtt{get}_B.\mathtt{set}_C.\mathtt{step}_A.\mathtt{get}_B.\mathtt{set}_C.\mathtt{step}_A.\mathtt{get}_B.\mathtt{set}_C \tag{1}$$

The dot notation is used to denote the concatenation of two traces. The function $\dagger : \pi \times C \to \pi$ projects a co-simulation trace to a specific FMU and is inductively defined as for any trace π and `m`:

$$\begin{aligned} \emptyset \dagger \mathtt{m} &\triangleq \emptyset \\ \alpha_i.\pi \dagger \mathtt{m} &\triangleq \alpha_i \cup \pi \dagger \mathtt{m} && \text{if} \quad \mathtt{i} = \mathtt{m} \\ \alpha_i.\pi \dagger \mathtt{m} &\triangleq \pi \dagger \mathtt{m} && \text{if} \quad \mathtt{i} \neq \mathtt{m} \end{aligned}$$

Informally, the projection removes all actions not performed on the FMU `m`. The projection of the trace in Eq. (1) to the FMU A is:

$$\pi \dagger A = \texttt{step}_A.\texttt{step}_A.\texttt{step}_A$$

This projection function permits us to reason about the actions performed on a specific FMU without considering the other FMUs. The reasoning is done concerning the semantics of the actions defined in Section 3.1.

3.2 FMI-based Co-Simulation Semantics

The semantics of the FMU actions have previously been defined in [19,20] using the so-called run-time state of an FMU, an abstraction that serves as a concise and implementation-neutral representation of the FMU's state, achieved by disregarding the internal representation of the FMU. We adopt the same approach here and repeat the definition of the run-time state of an FMU from [19].

Definition 3 (Run-time State of an FMU). *The run-time state s^R of an FMU 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: (i) $S_{U_c}^R : U_c \to \mathbb{R}_{\geq 0}$ is a function mapping each input port to a timestamp. (ii) $S_{Y_c}^R : Y_c \to \mathbb{R}_{\geq 0}$ is a function mapping each output port to a timestamp. (iii) $S_{\mathcal{V}_c}^R : \texttt{VAR}_c \to \mathcal{V}$ is a function mapping each port to a value. The first component, $\mathbb{R}_{\geq 0}$, denotes the current time of the FMU.*

The composition of the states of all FMUs is called the co-simulation state.

Definition 4 (Co-simulation State). *The co-simulation state $s_{\mathcal{S}}^R$ of a scenario $\langle C, L, M, R, F\rangle$ is an element of the set $S_{\mathcal{S}}^R = time \times S_U^R \times S_Y^R \times S_V^R$ where: 1. $time : C \to \mathbb{R}_{\geq 0}$ is a function, where $time(c)$ denotes the current simulation time of FMU 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 simultaneous. 2. $S_U^R = \prod_{c \in C} S_{U_c}^R$ maps all inputs of the scenario to a timestamp. 3. $S_Y^R = \prod_{c \in C} S_{Y_c}^R$ maps all outputs of the scenario to a timestamp. 4. $S_{\mathcal{V}}^R = \prod_{c \in C} S_{\mathcal{V}_c}^R$ maps all ports of the scenario to a value.*

The co-simulation state is manipulated during the co-simulation by the orchestrator as actions are performed on the FMUs according to the orchestration algorithm. The orchestration algorithm aims to establish a consistent co-simulation state, which we define next. We use the notation $s \xrightarrow{\text{P}} s'$ is read as "s is changed to s' by executing P".

Definition 5 (Consistent Orchestration Algorithm). *A* orchestration algorithm P *is valid if it takes a consistent co-simulation state to another consistent co-simulation state. Formally:*

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

where `consistent` *is defined as:*

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

Informally, all coupled ports must have the same value, and all FMUs are synchronized concerning time.

The orchestration algorithm must not establish a consistent state, but it must also respect the semantics of the FMUs, which we define next.

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

$$s^R \xrightarrow{\texttt{get}(s^{(t)},y)} (v, s^{R\prime}) \implies \texttt{preGet}(y, s^R) \wedge \texttt{postGet}(y, s^R, s^{R\prime}, v)$$

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

The precondition (above) states that no value must have been obtained from the output y since the FMU was stepped, formally described as $s_Y^R(y) < t$. It also mandates 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.

$$\begin{aligned}\texttt{postGet}(y, \langle t, s_U^R, s_Y^R, s_{\mathcal{V}}^R\rangle, \left\langle t, s_U^R, s_Y^{R\prime}, s_{\mathcal{V}}^R\right\rangle, \langle t_v, x\rangle) \triangleq\ & s_Y^{R\prime}(y) = t \\ & \wedge \forall y_m \in (Y \setminus y) \cdot s_Y^{R\prime}(y_m) = s_Y^R(y_m) \\ & \wedge t_v = t \wedge s_{\mathcal{V}}^R(y) = x\end{aligned}$$

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 FMU 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:*

$$s^R \xrightarrow{\texttt{set}(s^{(t)},u,v)} s^{R\prime} \implies \texttt{preSet}(u, v, s^R) \wedge \texttt{postSet}(u, v, s^R, s^{R\prime})$$

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

The precondition says that the input must not have been assigned a new value since the FMU 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 the timestamp of the FMU. Delayed inputs ($\neg R(u)$) should be set with a value with the same timestamp as the FMU. The postcondition (below) ensures the value and time of the input u is updated to match the value assigned to the input.

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

Although the precondition `preSet` could be subsumed by the precondition `preStepT`, having it as an explicit precondition localizes the error.

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

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

$$\begin{aligned}\texttt{preStepT}(H, \langle t, s_U^R, s_Y^R, s_{\mathcal{V}}^R\rangle) \triangleq\ & \forall u \in U \cdot ((R(u) \wedge t + H = s_U^R(u)) \\ & \vee (\neg R(u) \wedge t = s_U^R(u)))\end{aligned}$$

The precondition specifies that all the inputs of the FMU have been updated based on their reactivity constraints, with reactive inputs at time $t + H$ and delayed inputs at time t. The postcondition guarantees that the time of the FMU has progressed by the step duration H and that new port values have been computed.

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

3.3 Model Checking & UPPAAL

Model-checking [12,4] is a technique for automated verification of complex reactive systems by expressing the specification of a system using logical formulas.

The model of this paper is developed using *UPPAAL* [7], a model checker based on the timed automata (TA) theory [2]. The evolution of a model is modeled by a succession of transitions between a finite set of *locations*. A transition can be labeled with one or more actions, a guard, and a synchronization. The *state* of a model is given its current location and the values of the variables. Models can be composed by synchronizing on the same action/synchronization channel, which is used to model communication between models. Two transitions labeled with the same synchronization channel can only be executed if both transitions' guards are satisfied. The concept of time is not utilized in this work, indicating that the work can be replicated in other model checkers.

4 Traces for Simulation Scenarios

Monitors are a well-known concept in the field of run-time verification [6], where they are used to detect deviations from a given specification. Following Falcole et al. [13], we define a monitor in Definition 9.

Definition 9 (Monitor). *The monitor M is the tuple $\langle D, A, Q, q_0, \Delta, \Gamma \rangle$ where: (i) D is the verdict domain. (ii) A is a set of actions/events. (iii) Q is a set of states. (iv) q_0 is the initial state. (v) $\Delta : Q \times A \to Q$ is the transition function. (vi) $\Gamma : Q \to D$ is the verdict function.*

We aim to develop a monitor capable of verifying the execution of a co-simulation dynamically throughout the entire simulation to detect deviations from the proposed simulation. The monitor employs a UPPAAL model capable of detecting whether a given co-simulation trace respects the FMI semantics. The UPPAAL model, described in the following section, can be automatically instantiated to a given co-simulation scenario using the Scenario-Verifier tool [3].

4.1 The UPPAAL model

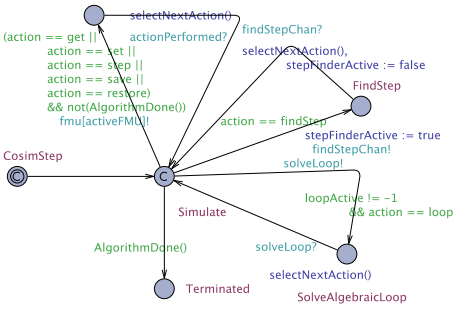
We have developed a UPPAAL model that can be instantiated to a given co-simulation scenario and used to verify whether the scenario respects the FMI semantics. The model is based on the model presented in [20] and is extended to support monitoring of dynamic co-simulations s The model formalizes a co-simulation as a set of SUs, each defined by a TA encoding Definition 1. An orchestrator, another TA, drives the simulation by interpreting the orchestration algorithm and exercising the SUs accordingly. A scenario is instantiated in UPPAAL by instantiating one SU template for each SU in the scenario and one of the other templates per scenario. The Scenario-Verifier automatically generates the couplings and other scenario-specific data as global variables in UPPAAL. The following subsections introduce SU and Orchestrator templates.

The Orchestrator template. The Orchestrator drives the co-simulation by sequentially interpreting the provided orchestration algorithm and exercising the SUs accordingly using channel synchronization (`fmu[activeFMU]!`) and the shared variables (`action` and `var`). A unique channel is used for each SU to ensure that only one SU receives and synchronizes on a given action request. Figure 1a shows how the Orchestrator exercises an SU using the channel `fmu[activeFMU]!` and waits for a confirmation that the action was successfully performed on the channel `actionPerformed?`.

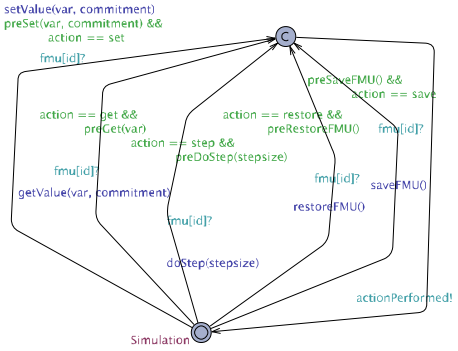
The Orchestrator waits after each action request until the SU has successfully performed the requested action. Upon receiving the confirmation, the Orchestrator selects the next action using the function `selectNextAction()`. This pattern repeats until the entire algorithm has been interpreted, i.e., `AlgorithmDone()`, whereafter the Orchestrator terminates the co-simulation by going to the state `Termination`. The edges to the states `FindStep` and `SolveAlgebraicLoop` are only relevant for scenarios with cyclic dependencies and active error-estimation, which we omit in this paper.

The SU template. The SU template shown in Fig. 1b represents the run-time state defined by Definition 3 and implements the SU interface described in Definition 1. The figure shows the different actions the Orchestrator can invoke on

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



(a) The Orchestrator template in UPPAAL. The Orchestrator invokes action using the channel `fmu[activeFMU]!`.



(b) The SU template in UPPAAL. The Orchestrator can invoke the actions `getValue`, `setValue` and `step` by synchronizing on the channel `fmu[id]?`.

the SU by synchronizing on the channel `fmu[id]?`. The incoming edge to the `Simulation` state confirms that the SU has successfully performed the requested action by synchronizing on the channel `actionPerformed!`, thus enabling the Orchestrator to issue new actions. The actions from Definitions 6 to 8 have been implemented in UPPAAL as `getValue`, `setValue` and `step` respectively. The preconditions (`preSet`, `preGet`, and `preStep`) of the actions are implemented as guards on the corresponding edges. An action is only performed if the guard evaluates to true, which means that all the violations of the proposed semantics result in a deadlock since the Orchestrator cannot synchronize with the given SU. This means that the Orchestrator can only reach the state `Terminated` if all the performed actions respect the FMI semantics. Consequently, detection of errors can be accomplished by verifying that the Orchestrator always reaches the state *Terminated* using the following CTL formula:

$$A\Diamond Orchestrator.Terminated \tag{2}$$

We have now described the UPPAAL model and the mechanism, for detecting violations of the FMI semantics. The following section describes how this can dynamically verify a co-simulation during execution.

4.2 Monitoring of co-simulations

We use the UPPAAL model to monitor the execution of a co-simulation. The set of states Q, the transition function Δ, and the initial state q_0 are all implicitly defined by the UPPAAL model. In contrast, the set of actions in the orchestration algorithm defines the set of actions A. The verdict function Γ is defined using the CTL formula in Eq. (2). The Scenario-Verifier can now monitor a co-simulation by instantiating the UPPAAL model to a given co-simulation scenario and checking whether the co-simulation trace respects the FMI semantics to produce a verdict $v \in D$. The verdict domain D is a tuple: $D : \langle \mathbb{B}, A_E \rangle$,

where $\mathbb{B}$ specifies whether the co-simulation trace is valid and A_E denotes a set of actions referred to as the set of enabled actions. The enabled actions consist of those actions that would not have resulted in a deadlock. The set A_E is empty if the verdict is true and non-empty otherwise.

The verdict function permits dynamic adaptation of the orchestration algorithm, as the Scenario-Verifier tool can be called during the execution of the co-simulation with a sequence of previously performed actions and the current action. The Scenario-Verifier tool will then return a verdict stating whether the intended action is valid. If the verdict is false, it returns a set of alternative actions which the Orchestrator knows are safe.

Caching

As model checking is computationally expensive and unsuitable for real-time verification, we have implemented a caching mechanism to reduce the number of calls to UPPAAL. The caching mechanism is based on the following three observations: 1. All verification of the same scenario starts from the same initial state I, where `consistent`(I) (see Definition 5). 2. The simulation is deterministic, i.e., the same sequence of actions will always result in the same state. 3. Every valid orchestration algorithm will, from a consistent initial state, establish a consistent state S, where the only difference between I and S in terms of their run-time state is their time stamps. More information can be found in [20,16].

The caching mechanism exploits these observations by storing the verdicts of previously performed actions in a cache and reusing them if the same sequence of actions is performed again. Furthermore, the tool trims the orchestration algorithm by removing all actions before the last consistent state S was established, which tremendously reduces the size of the orchestration algorithm.

5 Online Monitoring

Equiped with a precise notion of what it means to follow a verified simulation scenario, we now introduce simulation scenarios as runtime monitor into our language as *scenario monitors*. We then define the operations that enable to monitor loaded FMUs. While we present it in the context of SMOL, the principles are applicable to any language.

A *functional mock-up object (FMO)* is an object-oriented wrapper that encapsulates an FMU. An FMO itself is a layer around an SU, together with the role it plays in the simulator.

Definition 10 (FMOs). *Let $\mathcal{I}$ be a set of object identifiers and $\mathcal{M}$ be a set of scenario identifiers. A* functional mock-up object (FMO) *is a triple* `FMO` $= \langle i, SU, c, s^R, MI \rangle$*, where $i \in \mathcal{I}$ is a unique identifier for the FMO, SU is a simulation unit with state space S, $s^R \in S$ its runtime state, c is an SU identifier and $MI \subseteq M$ is a set of monitor identifiers.*

A functional mock-up scenario (FMS) *is a triple $\langle m, M, \pi \rangle$, where $m \in \mathcal{M}$ is a scenario identifier, M is a monitor, and π is a co-simulation trace.*

An FMO does not keep track of its trace – its trace is only relative to a scenario, and thus stored with the scenario.

As for the semantics, we formalize program configuration using a set of variables Var that are assigned some values Val. In particular, object and scenario identifiers and doubles $\mathbb{D}$ are values: $\mathcal{I}, \mathcal{M}, \mathbb{D} \subseteq \mathsf{Val}$. FMOs and scenarios are stored in an additional set σ. The runtime semantics are given as a transition system between such states.

Definition 11 (State). *A* configuration $\mathsf{conf} = \langle \rho, \sigma \rangle$ *is a pair of a map* $\rho : \mathsf{Var} \mapsto \mathsf{Val}$ *that assigns values to locations and a set* σ *of FMOs and FMS's.*

To single out elements in states, we write $\{\mathtt{F}_1, \mathtt{F}_2, \ldots \mathtt{F}_n\}$ as $\mathtt{F}_1 \cdot \{\mathtt{F}_2, \ldots \mathtt{F}_n\}$ We refrain from introducing full operational semantics (which are a straightforward extension of the one given by Kamburjan and Johnsen [21]), and use $[\![\cdot]\!]_\rho$ for evaluation of expressions.

$$\textbf{(io)}\ \frac{s = \texttt{obj=simulate("fmu", p}_1\texttt{=e}_1, \ldots, \texttt{p}_n\texttt{=e}_n\texttt{)} \quad \mathcal{I}(\texttt{fmu}, [\![\texttt{e}_1]\!]_\rho, \ldots, [\![\texttt{e}_n]\!]_\rho) = \langle i, SU, \mathsf{role}, s^R, \emptyset \rangle}{\langle \rho, \sigma \rangle \xrightarrow[\top]{s} \langle \rho[\texttt{obj} \mapsto i], \langle i, SU, \mathsf{role}, s^R, \emptyset \rangle \cdot \sigma \rangle}$$

$$\textbf{(is)}\ \frac{s = \texttt{obj=scenario("fms", p)} \quad \mathcal{I}(\texttt{fms}) = \langle m, M, \epsilon \rangle}{\langle \rho, \sigma \rangle \xrightarrow[\top]{s} \langle \rho[\texttt{obj} \mapsto m], \langle m, M, \epsilon \rangle \cdot \sigma \rangle}$$

$$\textbf{(ro)}\ \frac{[\![\texttt{f}]\!]_\rho = i}{\langle \rho, \langle i, SU, \mathsf{role}, s^R, MI \rangle \cdot \sigma \rangle \xrightarrow[\top]{\texttt{f.role=e}} \langle \rho, \langle i, SU, [\![\texttt{e}]\!]_\rho, s^R, MI \rangle \cdot \sigma \rangle}$$

$$\textbf{(rs)}\ \frac{[\![\texttt{e}]\!]_\rho = i \qquad [\![\texttt{s}]\!]_\rho = m}{\langle \rho, \langle i, SU, \mathsf{role}, s^R, MI \rangle \cdot \sigma \rangle \xrightarrow[\top]{\texttt{s.assign(e)}} \langle \rho, \langle i, SU, \mathsf{role}, s^R, MI \cup \{m\} \rangle \cdot \sigma \rangle}$$

$$\textbf{(step)}\ \frac{[\![\texttt{fmo}]\!]_\rho = i \qquad [\![\texttt{e}]\!]_\rho = t \qquad b = \mathsf{canAdvance}(\sigma, \mathsf{role}, \mathsf{step}, MI)}{\langle \rho, \langle i, SU, \mathsf{role}, s^R, MI \rangle \cdot \sigma \rangle \xrightarrow[b]{\texttt{fmo.step(e)}} \langle \rho, \langle i, SU, \mathsf{role}, \mathsf{step}(s^R, t), MI \rangle \cdot \mathsf{advance}(\sigma, \mathsf{role}, \mathsf{step}, MI) \rangle}$$

$$\textbf{(get)}\ \frac{[\![\texttt{fmo}]\!]_\rho = i \qquad b = \mathsf{canAdvance}(\sigma, \mathsf{role}, \mathsf{get}_\mathtt{v}, MI)}{\langle \rho, \langle i, SU, \mathsf{role}, s^R, MI \rangle \cdot \sigma \rangle \xrightarrow[b]{\texttt{x=fmo.v}} \langle \rho[\mathtt{x} \mapsto \mathsf{get}(s^R, \mathtt{v})], \langle i, SU, \mathsf{role}, s^R, MI \rangle \cdot \mathsf{advance}(\sigma, \mathsf{role}, \mathsf{get}_\mathtt{v}, MI) \rangle}$$

$$\textbf{(set)}\ \frac{[\![\texttt{fmo}]\!]_\rho = i \qquad b = \mathsf{canAdvance}(\sigma, \mathsf{role}, \mathsf{set}_\mathtt{v}, MI)}{\langle \rho, \langle i, SU, \mathsf{role}, s^R, MI \rangle \cdot \sigma \rangle \xrightarrow[b]{\texttt{fmo.v=x}} \langle \rho, \langle i, SU[\mathtt{v} \mapsto [\![\mathtt{x}]\!]_\rho], \mathsf{role}, s^R, MI \rangle \cdot \mathsf{advance}(\sigma, \mathsf{role}, \mathsf{set}_\mathtt{v}, MI) \rangle}$$

Fig. 2: Rules for operations on states.

The language allows the following operations on objects and scenarios. The rules are given in Fig. 2 and for examples of these operations we refer to Sec. 2. The use the auxiliary functions $\mathsf{canAdvance}(\sigma, \mathsf{role}, a, MI)$ to check whether the FMO playing role role can perform action a for all scenarios with identifiers in MI, and $\mathsf{advance}(\sigma, \emptyset, a, \mathsf{role})$ which updates these scenarios in σ accordingly. They are defined in the appendix. The transition relation has the form

$$\mathsf{conf} \xrightarrow[b]{s} \mathsf{conf}'$$

where s is the statement whose execution transitions conf into conf$'$. The boolean parameter b is either $\top$, if no error occurs, or $\bot$, if an error occurs.

Loading Using the statement `obj=simulate("fmu",p = e)` one can load an FMU at location `fmu` into the language and initialize its parameters `p` **(io)**. Let $\mathcal{I}(loc, v_1, \ldots, v_n)$ be the FMO generated by initializing the FMU at location loc, with a fresh identifier, a unique SU identifier, and an empty set of monitor identifiers.
Similarly, a scenario is loaded with the statement `scn=scenario("fms");` **(is)**. Let $\mathcal{I}(loc)$ be the FMS generated by initializing the monitor at location loc, with a fresh identifier and an empty trace.

Role Management To manipulate the role of an FMO, it has a special field `role` that can be assigned a role (cf. Fig. 1.3, **(ro)**). The effect of the statement is that the manipulated FMO has a new role identifier, but remains unchanged otherwise. To assign an FMO to a scenario, an FMO offers the method `fms.assign`. The scenario is then linked to the FMO and uses it according to the current value of its role **(rs)**.

Input/Output Each input and output `v` can similarly accessed using a field `fmo.v`. Semantically, this corresponds to the `set` **(set)** and `get` **(get)** functions on the contained SU. Accessing the fields checks that all scenarios assigned the FMO is question is assigned to allow to do so. If the check fails, the scenario issues a error message. This is the absence of this very error messages that we aim to monitor. To actively avoid them, there are methods `canGet_v()` and `canSet_v()` for each field `v`.

Time Advance Advancing the time, i.e., using the `step` function on the contained FMO **(step)**, is done with the method `fmo.step(e)`, where `e` must be an expression of `Double` type. Again, all scenarios the FMO is assigned to are consulted and there is also a method `canStep()` to check whether all scenarios this FMO participates it allow it to perform a time advance next, we give its semantics below.

We say that a run is *valid* if none of its transitions is annotated with $\bot$. It is easy to see that valid runs to not violate consistency of the monitored scenarios.

Discussion. The assignment of an FMO to an FMS can change over time, by either assigning a new role to an FMU, or by changing the assignment of FMU to scenario. With this capability it is possible to exchange one FMU for another within a simulation scenario on the fly, without restarting the others. Furthermore, one can transfer one FMU to another, meaning that results can be reused if the overall system, e.g., a Digital Twin, uses a multi-stage experiment.

An FMO can be assigned to several FMS objects, meaning that we can share simulation units between two simulations, as long as the operations on them are in the intersection of allowed behavior. Consider a scenario where an FMU for a computationally heavy simulation is only read in both scenarios – instead of running the simulation twice, it is run once and the monitor ensures that it is

used correctly according to both scenarios. Current co-simulation framework not only require that scenarios are static, but also require that no FMUs are shared.

Our monitors verify the order of operations on FMUs. The scenario also describes how data is transmitted between the simulation units. This is a standard data flow problem which is not specific to simulations, and we refrain from introducing it here. Similarly, reassignment upon change of the scenario requires a *meta-scenario* that describes correct reassignment. As no formalization of meta-scenarios exists, we leave this as an open challenge.

6 Evaluation

This section showcases the application of the proposed approach to a case study from the power systems domain. All code and models [4] in the case study are implemented as FMUs according to the FMI 2.0 standard [8] using UniFMU [25].

Experimental Setup.

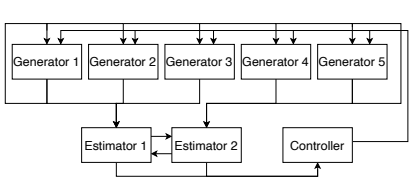
We consider the problem of dynamic state estimation (DSE) [23] on the standard IEEE-14 bus test system [10] to demonstrate the proposed approach for online verification of reconfigurable simulation scenarios on a realistic case study. DSE are used in the control of power systems to ensure stability and reliability [3]. The simulated system consists of a power system model, control center, and controllers of the generation system. The power system model comprises the power system's generation, transmission, and distribution segments wherefrom operational data are measured and transmitted to the control center. The estimated states (frequency and angle) are sent to the controller that adjusts the generation units' mechanical power to ensure the power system's stability. The equations of the system can be found in [23], but are omitted here for brevity.

The topology of the power grid changes dynamically as the load demand varies [27]. We consider two different topologies of the power system, each with a different number of synchronous generators and estimators as depicted in Fig. 3. The number of synchronous generators and estimators is related as the number of synchronous generators determines the computation load of the estimators. The first topology *scenario 1*, depicted in Fig. 3a, consists of 5 synchronous generators, two estimators, and one controller, while the second topology *scenario 2*, depicted in Fig. 3b, consists of 4 synchronous generators, one estimator, and one controller. The scenarios depicted in Fig. 3 are simplified versions of the simulated system, as many connections between the different components have been omitted for clarity. Concretely, every connection between the components in Fig. 3 represents between 1 and 10 connections in the simulated system.

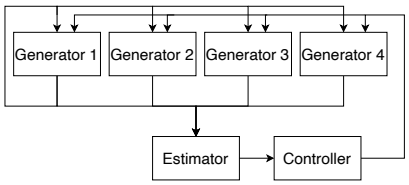
6.1 Changing the Power System Topology

Experimental Setup.

[4] Auxiliary material is available at `https://github.com/smolang/SemanticObjects/tree/fmonitor`



(a) Scenario 1: 5 synchronous generators, 2 estimators, and 1 controller.



(b) Scenario 2: 4 synchronous generators, 1 estimator, and 1 controller.

Fig. 3: The two different power system topologies considered in the case study.

The two scenarios in Figs. 3a and 3b overlap significantly as the controller and the synchronous generators (despite the change in the number between the two scenarios) are the same in both scenarios. In comparison, the estimators are different, as highlighted by the different names used for the estimators in the two scenarios. This overlap allows us to reuse many FMUs between the two scenarios. The only FMUs that need to be changed between the two scenarios are the estimators, as the estimators in Fig. 3a cannot estimate the states of the synchronous generators in Fig. 3b. Consequently, the estimators in Fig. 3a must be removed and replaced with the estimators in Fig. 3b. Another change that must be made between the two scenarios is the number of synchronous generators the controller controls, hence the number of connections changes between the two scenarios. Finally, there is a slight change in the behavior of the synchronous generators and the controller between the two scenarios, as the synchronous generators and the controller must adapt to the change in the topology. This change can be triggered using a parameter to the relevant FMUs.

The simulation was carried out in SMOL. The simulation was run 10 times for $3s$ with a step size of $0.001s$ for the first scenario, then with the same parameters for the second scenario. The average times to perform this simulation with and without monitoring are depicted in Fig. 4.

Monitoring causes a significant overhead in the first iteration (+1591%), and an acceptably small one in the following ones (+200%) if the monitor is running. Given that the simulation of n seconds requires $n * 1000$ iterations, this overhead occurs rarely and once per reconfiguration. Independent of monitoring, the reconfiguration requires additional time and cannot be performed on-the-fly: For one, the reconfiguration of the simulator must the performed, e.g., FMUs must be loaded. Loading the FMUs alone takes 87% of the initial overhead.

	avg. reconf. (ms)	avg. first iteration (ms)	avg. other iterations
Without monitoring	5864	24	25
With monitoring	6669	406	80
Change	+13%	+1591%	+220%
Monitoring first only	6669	406	25

Fig. 4: Monitoring overhead

Furthermore, loaded FMUs must be warmed-up, e.g., for calibration [14]. Thus, we deem the overhead acceptable for digital twins and exploration.

Lastly, as we pointed out the iterations are the same, a simple static analysis on the dynamically loaded orchestration algorithm (e.g., no branching in the main loop) can be used to determine whether the monitoring should be turned off after the first iteration and only turned on reconfiguration. As $3s$ simulated time require $\approx 75s$, an overhead of $1.8s$ corresponds to merely 2% overhead.

7 Related Work

Temperekidis et al. [33] synthesize LTL-monitors for FMI-based simulations, and use the monitors to verify the *values* of the simulation. In contrast, we monitor that the simulation indeed follows the scenario. Furthermore, they do not provide capabilities allowing to dynamically change the orchestration algorithm. Balakrishnan et al. [5] and Zapridou et al. [39] introduces runtime verification of simulations in the context of self-adaptive autonomous driving systems, again for the values of the simulations, not their structure and adherence.

Runtime monitoring is an important tool for self-adatptive systems, and we refer to the surveys [37,36] for a detailed treatment. In particular Weyns [36] discusses the application of general formal methods to digital twins. The work by Wright et al. [38] focuses on ensuring system level properties of a digital twin, and uses reachability analysis to ensure that changes to the physical system are safe. Similarly, our work uses reachability analysis to ensure that the digital twin is a faithful representation of the simulation scenario.

Most works on runtime monitoring and digital twins are focusing on using the twins as the monitor – such as the example in Sec. 2 is a monitor that detects drift between simulated and observed behavior. For example, Leucker et al. [26,34] discuss the use of simulation to monitor rescue missions or energy grids to recommend and plan actions. Hallé et al. [17] gives a general approach to design a specification language to compare simulation results and observed behaviors online. As for the use of a model checking appraoch for runtime verification, Cimatti et al. [11] extend nuXmv is a similar fashion as our works uses UPPAAL, but not for model checking.

8 Conclusion

This paper has presented a novel approach to online verification of reconfigurable simulation models. We in particular investigate reconfiguration of simulations in digital twins, either to adapt to changing situation unforseen at design time, or to enable exploration. Our approach is a step towards more flexible, yet safe uses of co-simulation, with the runtime overhead focused on the first iteration, whre other overhead, such as loading and calibration, already occurs. Instead of using a specification based on temporal properties, we repurpose simulation scenario as a specification language for runtime monitoring.

As for future work, we plan to use specification approaches closer to programming to describe simulation scenarios in programming languages, such as session types and typestate, to further integrate simulation units and programming.

References

1. Aheleroff, S., Xu, X., Zhong, R.Y., Lu, Y.: Digital twin as a service (dtaas) in industry 4.0: An architecture reference model. Adv. Eng. Informatics **47** (2021)
2. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science **126**(2) (1994)
3. Aweya, J., Al Sindi, N.: Role of time synchronization in power system automation and smart grids. In: 2013 IEEE ICIT (2013)
4. Baier, C., Katoen, J.P.: Principles of model checking. The MIT Press, Cambridge, Mass (2008)
5. Balakrishnan, A., Deshmukh, J., Hoxha, B., Yamaguchi, T., Fainekos, G.: PerceMon: Online Monitoring for Perception Systems. In: Runtime Verification. Springer, Cham, Switzerland (Oct 2021)
6. Bartocci, E., Falcone, Y., Francalanza, A., Reger, G.: Introduction to runtime verification. In: Lectures on Runtime Verification. Springer International Publishing (2018)
7. Behrmann, G., David, A., Larsen, K., Hakansson, J., Petterson, P., Yi, W., Hendriks, M.: UPPAAL 4.0. In: Third International Conference on Quantitative Evaluation of Systems (QEST 2006). Springer (2006)
8. Blockwitz, T., Otter, M., Åkesson, J., Arnold, M., Clauss, C., Elmqvist, H., Friedrich, M., Junghanns, A., Mauss, J., Neumerkel, D., Olsson, H., Viel, A.: Functional Mockup Interface 2.0: The Standard for Tool independent Exchange of Simulation Models. In: Proc. 9th International Modelica Conference. Linköping University Electronic Press (2012)
9. Cavalcanti, A., Woodcock, J., Amálio, N.: Behavioural models for FMI co-simulations. Theoretical Aspects of Computing – ICTAC 2016 **9965** (2016)
10. Christie, R.D.: Power Systems Test Case Archive - UWEE (Mar 2023), `http://labs.ece.uw.edu/pstca`, [Accessed 8. Mar. 2023]
11. Cimatti, A., Tian, C., Tonetta, S.: Nurv: A nuxmv extension for runtime verification. In: RV. Lecture Notes in Computer Science, vol. 11757, pp. 382–392. Springer (2019)
12. Clarke, Jr., E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge, MA, USA (1999)
13. Falcone, Y., Havelund, K., Reger, G.: A tutorial on runtime verification. Engineering Dependable Software Systems **34** (01 2013)
14. Feng, H., Gomes, C., Thule, C., Lausdahl, K., Sandberg, M., Larsen, P.G.: The Incubator Case Study for Digital Twin Engineering. arXiv (Feb 2021)
15. Gomes, C., Broman, D., Vangheluwe, H., Thule, C., Larsen, P.G.: Co-simulation: A survey. ACM Computing Surveys **51**(3) (2018)
16. Gomes, C., Lucio, L., Vangheluwe, H.: Semantics of co-simulation algorithms with simulator contracts. In: Proc. ACM/IEEE MODELS'19. IEEE, (2019)
17. Hallé, S., Soueidi, C., Falcon, Y.: Leveraging runtime verication for the monitoring of digital twins. In: Formal Methods and Digital Twins Workshop, co-located with FM'23 (2023)
18. Hansen, S.T., Gomes, C., Larsen, P.G., van de Pol, J.: Synthesizing co-simulation algorithms with step negotiation and algebraic loop handling. In: ANNSIM'21. IEEE (2021)
19. Hansen, S.T., Ölveczky, P.C.: Modeling, algorithm synthesis, and instrumentation for co-simulation in maude. In: WRLA 2022, Munich, Germany. Springer (2022)

20. Hansen, S.T., Thule, C., Gomes, C., van de Pol, J., Palmieri, M., Inci, E.O., Madsen, F., Alfonso, J., Castellanos, J.Á., Rodriguez, J.M.: Verification and synthesis of co-simulation algorithms subject to algebraic loops and adaptive steps. STTT **24**(6) (2022)
21. Kamburjan, E., Johnsen, E.B.: Knowledge structures over simulation units. In: ANNSIM. IEEE (2022)
22. Kamburjan, E., Klungre, V.N., Schlatte, R., Johnsen, E.B., Giese, M.: Programming and debugging with semantically lifted states. In: ESWC. LNCS, vol. 12731. Springer (2021)
23. Kazemi, Z., Safavi, A.A., Naseri, F., Urbas, L., Setoodeh, P.: A secure hybrid dynamic-state estimation approach for power systems under false data injection attacks. IEEE Transactions on Industrial Informatics **16**(12) (2020)
24. Kübler, R., Schiehlen, W.: Two methods of simulator coupling. Mathematical and Computer Modelling of Dynamical Systems **6**(2) (2000)
25. Legaard, C.M., Tola, D., Schranz, T., Macedo, H.D., Larsen, P.G.: A universal mechanism for implementing functional mock-up units. In: 11th International Conference on Simulation and Modeling Methodologies, Technologies and Applications. SIMULTECH 2021, Virtual Event (2021)
26. Leucker, M., Sachenbacher, M., Vosteen, L.B.: Digital twin for rescue missions – a case study. In: Formal Methods and Digital Twins Workshop, co-located with FM'23 (2023)
27. Marchiano, M., Rayworth, D.M.J., Alegria, E., Undrill, J.: Power generation load sharing using droop control in an island system. IEEE Transactions on Industry Applications **54**(2) (2018)
28. Oakes, B.J., Gomes, C., Holzinger, F.R., Benedikt, M., Denil, J., Vangheluwe, H.: Hint-based configuration of co-simulations with algebraic loops. In: Simulation and Modeling Methodologies, Technologies and Applications, vol. 1260. Springer International Publishing (2020)
29. Posse, E., Vangheluwe, H.: Kiltera: A simulation language for timed, dynamic structure systems. In: 40th Annual Simulation Symposium (ANSS'07). IEEE (2007)
30. Schweizer, B., Li, P., Lu, D.: Explicit and implicit cosimulation methods: Stability and convergence analysis for different solver coupling approaches. Journal of Computational and Nonlinear Dynamics **10**(5) (2015), publisher: ASME
31. Tahara, T., Gavina, M.K.A., Kawano, T., Tubay, J.M., Rabajante, J.F., Ito, H., Morita, S., Ichinose, G., Okabe, T., Togashi, T., Tainaka, K.i., Shimizu, A., Nagatani, T., Yoshimura, J.: Asymptotic stability of a modified lotka-volterra model with small immigrations. Scientific Reports **8**(1) (May 2018)
32. Talasila, P., Gomes, C., Mikkelsen, P.H., Arboleda, S.G., Kamburjan, E., Larsen, P.G.: *Digital Twin as a Service (DTaaS): A Platform for Digital Twin Developers and Users*. In: IEEE Digital Twins (2023), to appear
33. Temperekidis, A., Kekatos, N., Katsaros, P.: Runtime verification for FMI-based co-simulation. In: Runtime Verification. Springer International Publishing (2022)
34. Thoma, D., Sachenbacher, M., Leucker, M., Ali, A.T.: A digital twin for coupling mobility and energy optimization: The renubil living lab. In: Formal Methods and Digital Twins Workshop, co-located with FM'23 (2023)
35. Thule, C., Lausdahl, K., Gomes, C., Meisl, G., Larsen, P.G.: Maestro: The INTO-CPS co-simulation framework. Simul. Model. Pract. Theory **92**, 45–61 (2019)
36. Weyns, D.: Software engineering of self-adaptive systems. In: Handbook of Software Engineering. Springer International Publishing (2019)

37. Weyns, D., Iftikhar, M.U., de la Iglesia, D.G., Ahmad, T.: A survey of formal methods in self-adaptive systems. In: Proceedings of the Fifth International Conference on Computer Science and Software Engineering. ACM (jun 2012)
38. Wright, T., Gomes, C., Woodcock, J.: Formally verified self-adaptation of an incubator digital twin. In: LNCS. Springer Nature Switzerland (2022)
39. Zapridou, E., Bartocci, E., Katsaros, P.: Runtime Verification of Autonomous Driving Systems in CARLA. In: Runtime Verification. Springer, Cham, Switzerland (Oct 2020)

.1 Auxiliary Definitions

$$\mathsf{canAdvance}(\sigma, \mathsf{role}, a, MI) \iff$$
$$\forall m \in MI.\ \exists \Delta, \Gamma, q.\ \langle m, \langle \_, \_, \_, \_, \Delta, \Gamma \rangle, q, \_ \rangle \in \sigma \wedge \Gamma(\Delta(q, (a, \mathsf{role}))) = (\top, \_)$$
$$\mathsf{advanceS}(\langle m, \langle \_, \_, \_, \_, \Delta, \Gamma \rangle, q, \pi \rangle \cdot \sigma, m, \mathsf{role}, a) =$$
$$\langle m, \langle \_, \_, \_, \_, \Delta, \_ \rangle, \Delta(q, (a, \mathsf{role})), \pi \cdot (a, \mathsf{role}) \rangle \cdot \sigma$$
$$\mathsf{advance}(\sigma, \emptyset, a, \mathsf{role}) = \sigma$$
$$\mathsf{advance}(\sigma, m \cdot MI, a, \mathsf{role}) = \mathsf{advance}(\mathsf{advanceS}(\sigma, m, a, \mathsf{role}), MI, a, \mathsf{role})$$