Monitoring Reconfigurable Simulation Scenarios in Co-simulated Digital Twins

Simon Thrane Hansen[1], Eduard Kamburjan[2], and Zahra Kazemi[3]

[1] University of Luxembourg, Luxembourg, simon.hansen@uni.lu
[2] University of Oslo, Norway, eduard@ifi.uio.no
[3] Aarhus University, Denmark, zka@ece.au.dk

Abstract

Co-simulation, essential for various domains, involves simulating systems with heterogeneous subsystems by combining them into scenarios using orchestration algorithms. Due to system evolutions, digital twins may require dynamic reconfiguration of the scenario and orchestration algorithm, which may introduce a significant simulation error that may go unnoticed if the scenario and orchestration algorithm are not correctly aligned. This paper introduces a monitor-based approach to automatically detect and help prevent any discrepancies between the scenario and the orchestration algorithm. The approach's overhead may be significant for short simulations, but amortizes over longer simulations, as illustrated by a case study.

1 Introduction

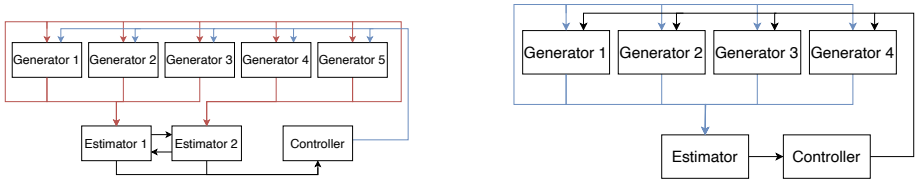
Co-simulation, vital for simulating complex systems, involves integrating heterogeneous simulation models into a single simulation that captures all aspects of the system [9,18]. Each model represents a *dynamic system* that evolves according to a set of *evolution rules* and *stimuli* from other models in the simulation. Interoperability between heterogeneous models, developed by different companies using different tools, can be achieved by exporting the model as a *Functional Mock-up Unit* (FMU) from one of the more than 170 tools supporting the *Functional Mock-up Interface* (FMI) [4,8]. A collection of FMUs can be composed into a *scenario*, representing their collective behavior by coupling their input and output ports.

Simulations of digital twins (DTs) require that the corresponding scenario mirrors the structure of the physical counterpart by adjusting both the involved FMUs and their connections dynamically [16]. This is a challenging task as the behavior of a scenario is obtained by advancing the states of the FMUs and exchanging stimuli between them according to a scenario-tailored *orchestration algorithm* (OA) [18,9,13,21]. Consequently, methods have emerged to verify that an OA is suitable for a specific scenario, ensuring that data are exchanged according to the evolution rules of the FMUs [13,21,22]. The existing methods apply only to static scenarios, where the scenario remains unchanged throughout the simulation [25]. Simulating a reconfigurable scenario requires dynamic loading and unloading FMUs and adjusting the OA accordingly to ensure accurate simulation results. To mitigate the risk of introducing errors during reconfiguration, we

propose a runtime verification method implemented in the Semantic Micro Object Language (SMOL) [15] and the UPPAAL toolbox [3] to ensure that the OA and scenario are aligned.

To illustrate the challenge and our solution in the remainder of this paper, consider a DT simulation of a power grid with a controller [17], based on the IEEE-14 system [5]. The system consists of a set of power generators and one or more estimators used by the controller to estimate the generators' hidden states. The controller adjusts the generators' operational parameters to ensure reliable operation. The number of power generators, and subsequently estimators, changes dynamically as the load demand varies [20]. We consider two different topologies, each with a different number of generators and estimators as depicted in Figure 1. The approach doesn't limit the number of topologies, but we focus on two for clarity.

Our solution introduces a dynamic and flexible scenario monitor, that gets queried before any FMU action to ensure that the action is allowed according to the current scenario. This monitor supports dynamic reconfiguration where the FMUs and connections can change in response to structural changes in the DT, as demonstrated when shifting from Scenario 1 to Scenario 2 (see Figure 1). Here, FMUs for the controller and generators 1-4 are reassigned, and a new estimator is introduced. The monitor checks that the OA follows the new scenario, which will include, e.g., a startup phase. Additionally, FMUs can be assigned to multiple scenarios, facilitating efficient FMU sharing Such reconfigurable scenarios are critical for DTs that must react to signals from *outside* the simulation by adopting both their structure and OA.



(a) Scenario 1: five synchronous generators, two estimators, and one controller.

(b) Scenario 2: four synchronous generators, one estimator, and one controller.

Fig. 1: Two different power system topologies. Each square represents an FMU, and each arrow represents between 1 and 10 connections between the FMUs.

Contribution. We present an approach to load co-simulation scenarios at runtime and (a) check properties of the scenario, as part of the DT, and (b) check that the OA and the scenario are aligned. This ensures the correctness of *reconfigurable co-simulation in the context of DTs*. We implement our approach as (a) a JVM library for online monitoring according to the FMI standard, and (b) an extension of the FMI-integration [14] into SMOL [15]. In the following, we introduce the notion of correctness (Section 2), the runtime checker (Section 3), and its integration for reconfiguration (Section 4). We

demonstrate how our method ensures that the OA correctly adapts to the changes in the above power grid case study (Section 5).

2 State-of-the-Art and Preliminaries

2.1 State-of-the-Art in Runtime Verification of Simulation Units

Co-simulation is commonly realized by frameworks that do not allow to reconfigure after the co-simulation is started [9]. Runtime monitoring has been investigated by Temperekidis et al. [23], who synthesize LTL-monitors for FMI-based simulations, verifying the values within simulations. Our work, in contrast, ensures that simulations adhere to the specified scenario and dynamically adapt the OA, a capability not provided by them. Similar to Balakrishnan et al.[1] and Zapridou et al.[29], who apply runtime verification in autonomous driving simulations, our approach emphasizes the structure and adherence of simulations, but uses co-simulation scenarios as specifications and adopts a language-based approach. In a broader sense, runtime monitoring is an important tool for self-adaptive systems, and we refer to the surveys [27,26] for a detailed treatment. Specifically to DTs, Weyns [26] discusses the application of general formal methods to DTs. The work by Wright et al. [28] uses reachability analysis to ensure system-level properties of a DT. Similarly, our work uses reachability analysis to ensure that the DT faithfully represents the simulation scenario. Most works on runtime monitoring and DTs focus on using the twins as the monitor – such as the example in Section 1 is a monitor that detects drift between simulated and observed behavior. For instance, Leucker et al. [19,24] discuss using simulation to monitor rescue missions or energy grids to recommend and plan actions. Hallé et al. [11] give a general approach for designing a specification language to compare simulation results and observed behaviors online. As for the use of a model checking approach for runtime verification, Cimatti et al. [6] extend nuXmv in a similar fashion as our work uses UPPAAL, but not for model checking. Current approaches to runtime monitoring of co-simulation either assume a fixed structure and cannot react to the changes triggered by a physical twin or focus on comparing data streams.

2.2 Preliminaries: Functional Mock-up Interface

To establish a notion of correctness for simulation scenarios, we require a precise, formal notion of consistent OAs and consistent simulation traces. Having established these properties, we can then check the consistency of the OA (whenever we reconfigure the scenario) and dynamically monitor the generated orchestration trace afterward. This section introduces the notion of valid OAs and traces and briefly introduces the necessary background on co-simulation and the FMI standard. We refer to [9] for a comprehensive introduction to FMI-based co-simulation. An FMU is a black-box representation of a model encapsulating the model's binary and a description file outlining the model's interface [4]. Following the FMI standard version 2.0 [4,12], we define the interface of an FMU as follows:

Definition 1 (Functional Mock-up 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 denotes the state space of the FMU. (ii) U_c and Y_c are the input and output ports of the FMU. Each port holds a value from a set of values $\mathscr{V}$ and a timestamp from a set of timestamps $\mathbb{R}_{\geq 0}$. The set of timestamped values is denoted $\mathscr{V}_{\mathscr{T}} = \mathbb{R}_{\geq 0} \times \mathscr{V}$. (iii) The functions $\texttt{set}_c : S_c \times U_c \times \mathscr{V}_{\mathscr{T}} \to S$ and $\texttt{get}_c : S_c \times Y_c \to \mathscr{V}_{\mathscr{T}}$ set an input and get an output, respectively. (iv) The function $\texttt{step}_c : S_c \times \mathbb{R}_{>0} \to S_c \times \mathbb{R}_{>0}$ advances the state of the FMU by a given duration. If an FMU is in state $s_c^{(t)}$ 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 $h \leq H$. The FMU advances by h because the FMU, due to implementation details, cannot advance by H.*

A set of FMUs is combined into a *scenario* by connecting their ports to specify dependencies between them.

Definition 2 (Scenario). *A* scenario $\mathscr{S}$ *is a tuple $\mathscr{S} \triangleq \langle C, L, M, R, F \rangle$, where (i) C is a finite set (of FMU identifiers). (ii) L is a function $L : U \to Y$, where $U = \bigcup_{c \in C} U_c$ 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 whether an FMU uses interpolation or extrapolation on a given input to approximate the evolution of the connected output when stepping the FMU. An input is called* reactive *if $R(u)$, which means the FMU uses interpolation, while an input is* delayed *and the FMU uses extrapolation if $\neg R(u)$ [10]. (v) F is a family of functions $\{F_c : Y_c \to \mathscr{P}(U_c)\}_{c \in C}$. 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.*

In addition to the FMUs and how they are coupled, a scenario introduces the functions M, R, and F referred to as the scenarios' contracts. Gomes et al. [9,10] introduced the contracts to capture non-confidential information about how an FMU is most optimally simulated in a scenario. For instance, an FMU B with a reactive input u connected to the output y of FMU A means that FMU A must be stepped before the value of the output y is obtained and set on the input u to ensure that FMU B can interpolate on the new input value of u when advancing its state.

An OA is a sequence of FMU operations outlining how the joint state of the scenario evolves by specifying when and how the state of the FMUs is advanced using the `step` operation and how data is exchanged between the FMUs using the `get` and `set` operations.

Definition 3 (Orchestration Algorithm). *The orchestration operations for a scenario $\mathscr{S} = \langle C, L, M, R, F \rangle$ is the set $\Big\{ \texttt{set}_c(\_, v, \_), \texttt{get}_c(\_, w), \texttt{step}_c(\_, \_) \mid v \in U_c, w \in Y_c, c \in C \Big\}$. An* OA *for a scenario $\mathscr{S}$ is a regular expression over orchestration operations for $\mathscr{S}$, which contains only sequence and iteration.*

A scenario should be simulated using a *suitable* OA, which is an OA A for a scenario S where no other algorithm produces more accurate simulation results than A, formally: $\forall OA \in OAs \cdot err(simulate(OA, S)) \geq err(simulate(A, S))$, where OAs is the set of all OAs, the result of a simulation of a scenario S using an OA OA is denoted

simulate(OA,S), and *err*(R) is a function that measures the error of a simulation result R. Unfortunately, the nature of FMUs inhibits the implementation of the function *err*(), making it impossible to reason directly about the accuracy of results in general. Nevertheless, Gomes et al. [10,21] showed that an OA respecting the contracts of a scenario is *suitable*, meaning it can be statically judged if an OA is suitable.

2.3 Preliminaries: FMI-based Co-Simulation Semantics

We incorporate the contracts defined in Definition 2 into the semantics of the actions described in Definition 1 to be able to conclude whether an OA is suitable for a given scenario solely based on the OA. Specifically, we adopt the notation and semantics defined in [12,13]. The semantics are defined in terms of a *runtime state* of an FMU. A runtime state abstracts the internal representation of an FMU while capturing the information necessary to reason about the actions performed on the FMU.

Definition 4 (Runtime State of an FMU). *The runtime state s^R of an FMU c in a scenario $\mathscr{S}$ is an element of the set $S_c^R = \mathbb{R}_{\geq 0} \times S_{U_c}^R \times S_{Y_c}^R \times S_{\mathscr{V}_c}^R$, where: (i) $\mathbb{R}_{\geq 0}$ is the timebase of the simulation; it denotes the current time of the FMU. (ii) $S_{U_c}^R : U_c \to \mathbb{R}_{\geq 0}$ maps each input port to a timestamp. (iii) $S_{Y_c}^R : Y_c \to \mathbb{R}_{\geq 0}$ is a function mapping each output port to a timestamp. (iv) $S_{\mathscr{V}_c}^R : (U_c \cup Y_c) \to \mathscr{V}$ maps each port to a value.*

The semantics are defined in terms of preconditions and postconditions for each action. We omit the semantics of the `get` and `set` functions for brevity, but they are defined similarly in [12,13]. The notation $s \xrightarrow{\mathrm{P}} s'$ is read as "s is changed to s' by executing the action or algorithm P".

Definition 5 (Step Computation). *Stepping an FMU using* $\texttt{step}(s^{(t)},H)$ *advances the state of the FMU by $H \in \mathbb{R}_{>0}$, formally:*

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

The precondition specifies that all the FMU inputs 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:

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

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

Every FMU action changes the *co-simulation state* - the runtime state of all the scenarios FMUs.

Definition 6 (Co-simulation State). *The co-simulation state $s_{\mathscr{S}}^R$ of a scenario $\langle C,L,M,R,F\rangle$ is an element of the set $S_{\mathscr{S}}^R = time \times S_U^R \times S_Y^R \times S_V^R$ 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 FMUs are simultaneous. 2. $S_U^R = merge(\{S_{U_c}^R \cdot c \in C\})$ maps all inputs of the scenario to a timestamp. 3. $S_Y^R = merge(\{S_{Y_c}^R \cdot c \in C\})$ maps all outputs of the scenario to a timestamp. 4. $S_{\mathscr{Y}}^R = merge(\{S_{\mathscr{Y}_c}^R \cdot c \in C\})$ maps all ports of the scenario to a value.

The co-simulation is performed using an iteratively applied OA (called a co-simulation step) that advances the co-simulation state from a time t to the time t' while ensuring that the values of the coupled ports remain equal. Furthermore, the algorithm must respect the contracts of the scenario, i.e., it must advance the FMUs according to their approximation functions and other contracts. This is captured by the following definition of a *consistent co-simulation state* that uses Dijkstra's weakest precondition calculus to account for the semantics of the FMU actions when advancing the co-simulation state from one consistent state to another.

Definition 7 (Consistent Orchestration Algorithm). *An OA A for a scenario $\mathscr{S}$ defined in Definition 2 is consistent if it takes an initial consistent co-simulation state at time t to a future consistent co-simulation state at time t' while respecting the semantics of the FMU actions.*

$$\texttt{consistent}(\langle t, s_U^R, s_Y^R, s_{\mathscr{Y}}^R \rangle) \implies wp(A, \texttt{consistent}(\left\langle t', s_U^{R\prime}, s_Y^{R\prime}, s_{\mathscr{Y}}^{R\prime} \right\rangle)) \wedge t' > t,$$

where `consistent` *states that all coupled ports are equal, and all FMUs have the same time [12].*

2.4 Preliminaries: Consistent Traces with Reconfiguration

While a consistent OA is suitable for a static scenario [10,12], it does not work for reconfigurable scenarios such as the power system in Figure 1 as a suitable OA for the scenario in Figure 1a is not suitable for the scenario in Figure 1b. Consequently, the OA must be correctly updated at every reconfiguration (a change in the scenario) to ensure that the used OA is always suitable OA for the current scenario. As the reconfigurations depend on changes to the simulation's physical counterpart, it cannot be checked statically whether the OA and scenario are properly aligned. Neither does it make sense to check this after the execution, as computational resources would have been wasted on am inconsistent simulation. To mitigate this problem, we use a runtime monitor to check that a DT simulation uses a suitable OA at all times. We say that such a simulation is *reconfiguration consistent*.

The monitor operates on the *co-simulation trace* π, a sequence of tuples capturing both the applied operations A and the current scenario $\mathscr{S}$ to capture the evolution of the co-simulation state indirectly. Reconfigurations can, due to the discrete nature of co-simulation, only occur between co-simulation steps, meaning we can group the co-simulation trace into a trace map using a series of projections, linking each time step to an OA and a scenario ($\pi_M : \mathbb{R}_{>0} \to \langle OAs, 2^{\mathscr{S}} \rangle$. We now define the notion of reconfiguration consistency.

Definition 8 (Reconfiguration Consistency). *A co-simulation trace π with the trace map π_M is reconfiguration consistent if the used OA is consistent with the corresponding scenario at all times, formally:*

$$consistentTrace(\pi) \triangleq \forall t \in dom(\pi_M) \cdot consistent(pr_1(\pi_M(t)), pr_2(\pi_M(t))),$$

where `consistent` *is defined in Definition 7 and the functions $pr_1()$ and $pr_2()$ projects, respectively, the first and second element of a tuple. A co-simulation trace π where $consistentTrace(\pi)$ holds, is reconfiguration consistent. Otherwise, we say that the co-simulation trace π is reconfiguration inconsistent.*

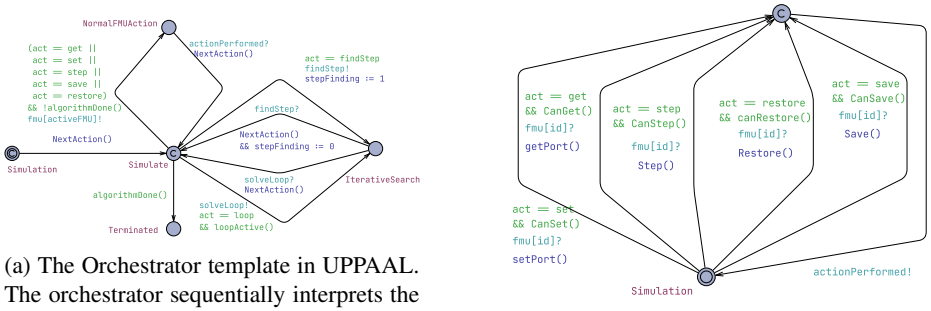
3 Runtime Verification of Co-simulations

This section defines a monitor for verifying whether a given co-simulation connected to a DT is reconfiguration consistent. Monitors enable runtime verification of a system by observing the system during execution and detecting violations of a given specification [2]. Falcone et al. [7] describe a monitor as a 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. The monitor is implemented in the UPPAAL model checker [3] and the Scenario-Verifier tool [13] to generate a tool-chain that can instantiate a UPPAAL monitor for a scenario described in a DSL.

3.1 The UPPAAL model

The UPPAAL model formalizes a co-simulation as a set of FMUs whose joint behavior is computed by an orchestrator executing an OA. Each FMU is represented by a UPPAAL template describing the runtime state of the FMU and the interface described in Definition 1. The orchestrator is represented by another UPPAAL template outlining how the orchestrator interprets the OA and exercises the FMUs accordingly. A scenario is instantiated in UPPAAL by instantiating one FMU template for each FMU in the scenario and one Orchestrator template per scenario. The two templates are outlined below.

The Orchestrator template. Figure 2a shows the (visualization of the) orchestrator template in UPPAAL. The orchestrator sequentially interprets the OA using the action `NextAction()` and exercises the FMUs accordingly. It executes the OA by invoking the actions defined in Definition 1 on the FMUs using the channel `fmu[activeFMU]!`, where `activeFMU` is the identifier of the FMU currently being exercised. The Orchestrator waits for the FMU to confirm that the action was successfully performed by synchronizing on the channel `actionPerformed?` before selecting and invoking the next action. When all actions in the OA have been performed (`algorithmDone`), the Orchestrator terminates the co-simulation by going to the state `Terminated`. The transitions to the state `IterativeSearch` are only relevant for particular scenarios with cyclic dependencies and active error estimation; we omit them due to space limitations.



(a) The Orchestrator template in UPPAAL. The orchestrator sequentially interprets the OA and exercises the FMUs accordingly.

(b) The FMU template in UPPAAL.

Fig. 2: The UPPAAL templates of the Orchestrator interpreting the OA and the FMU checking the OA.

The FMU template. In UPPAAL, each FMU is represented by a template that captures the runtime state of the FMU and implements the interface described in Definition 1. Each transition represents an action the Orchestrator can invoke on the FMU by synchronizing on the channel `fmu[id]` and setting the variables `act` and `id` to the action and identifier of the FMU, respectively. Each transition has a guard (`CanSet`, `CanGet`, and `CanStep`) to ensure that the model catches violations of Definition 7. Consequently, an action is only executed if the guard evaluates to true, which means that any violations of Definition 7 will result in a deadlock because the Orchestrator cannot synchronize with the given FMU. Once the FMU has performed the requested action, it synchronizes with the Orchestrator on the `actionPerformed!` channel to confirm that the action was performed so that the Orchestrator can invoke the next action.

3.2 (Online) Monitoring a Co-simulation

The UPPAAL model implements a monitor as the set of states Q, the transition function Δ, the set of actions A, and the initial state q_0 are defined by the UPPAAL model. The verdict function Γ is defined by the CTL formula in the UPPAAL query language $A\Diamond Orchestrator.Terminated$. It ensures that the simulation respects Definition 7 as all violations lead to a deadlock. UPPAAL evaluates this formula to determine the algorithm's suitability and suggests FMU actions to resolve potential deadlocks. The monitor is implemented as a Java library interfacing with UPPAAL to ensure that a simulation is reconfiguration consistent.

The implementation avoids unnecessary calls to UPPAAL by using a cache that exploits that a co-simulation step is typically applied more than once. The caching mechanism utilizes the following three observations: (i) All verification of the same scenario starts from the same consistent initial state I. (ii) The simulation is deterministic, i.e., performing the same sequence of actions from the same initial state always results in the same final state. (iii) A valid OA A starting from a consistent state S always leads to a consistent state S' where the only difference between S and S' is the simulation time. Moreover, applying A to S' results in a new consistent state S''. Consequently, all actions

performed before the last consistent state S was established can be removed [13,10]. The first two observations facilitate caching through verdict reuse, while the third reduces UPPAAL's state space by removing already verified actions from the OA.

4 Online Monitoring

Equipped with a precise notion of consistency, as well as means to verify a simulation and compare traces to scenarios, we now demonstrate how these techniques are applied for reconfiguration. To this end, we extend the SMOL programming language to handle *simulation scenarios* as runtime monitors by extending it with *scenario monitors*. While we present it in the context of SMOL, the principles apply to any language, but using it as a basis enables us to investigate DTs and simulation scenarios from the perspective of the software implementing them [14]. This section recaps the integration of FMUs into SMOL before describing how SMOL has been extended with a notion of scenarios to enable monitoring of reconfigurable scenarios.

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

Definition 9 (FMOs). *Let $\mathscr{I}$ be a set of object identifiers and $\mathscr{M}$ be a set of scenario identifiers. A* functional mock-up object (FMO) *is a tuple $\texttt{FMO} = \langle i, FMU, c, s^R, MI \rangle$, where $i \in \mathscr{I}$ is a unique identifier for the FMO, FMU is an FMU with state space S, $s^R \in S$ its runtime state, c is an FMU 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 $m \in \mathscr{M}$ 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 within the scenario. Note that in our system, the FMS is more than just the co-simulation scenario; it also includes the OA (through the monitor). We define the program's configuration using a set of variables Var assigned to values from the set Val. The values in Val include object identifiers $\mathscr{I}$ and scenario identifiers $\mathscr{M}$, as well as doubles $\mathbb{D}$. FMOs and scenarios are stored in the additional set σ. The runtime semantics are expressed as a transition system between these states.

Definition 10 (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 $\{\texttt{F}_1, \texttt{F}_2, \ldots \texttt{F}_n\}$ as $\texttt{F}_1 \cdot \{\texttt{F}_2, \ldots \texttt{F}_n\}$ We refrain from introducing full operational semantics (which are a straightforward extension of the one given by Kamburjan and Johnsen [14]), and use $[\![\cdot]\!]_\rho$ for evaluation of expressions.

The language permits the operations outlined below. The formal rules are given in Fig. 3. To facilitate these operations, we employ two auxiliary functions: $\mathsf{advance}(\sigma, \emptyset, a, \mathsf{role})$, which updates these scenarios in σ accordingly, and the function $\mathsf{canAdvance}(\sigma, \mathsf{role}, a, MI)$, which checks whether the FMO playing role role can perform action a for all scenarios with identifiers in MI. Fig. 4 provides their formal definitions. The transition relation has the form $\mathsf{conf} \xrightarrow[b]{s} \mathsf{conf}'$ where s is the statement

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

$$\textbf{(is)}\ \frac{\mathsf{Init}\left(\texttt{fms}\right) = \langle m, M, \varepsilon\rangle}{\langle \rho, \sigma\rangle \xrightarrow[\top]{s} \langle \rho[\texttt{obj} \mapsto m], \langle m, M, \varepsilon\rangle \cdot \sigma\rangle}$$

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

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

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

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

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

$$\textbf{(stop)}\ \frac{[\![\texttt{s}]\!]_\rho = m}{\begin{array}{l}\langle \rho, \mathsf{registered}(\sigma, m) \cdot \mathsf{unregistered}(\sigma, m)\rangle \xrightarrow[\top]{\texttt{s.stop()}} \\ \langle \rho, \mathsf{remove}(\mathsf{registered}(\sigma, m), m) \cdot \mathsf{unregistered}(\sigma, m)\rangle\end{array}}$$

Fig. 3: Operational Semantics Rules

$$
\begin{aligned}
&\mathsf{canAdvance}(\sigma, \mathsf{role}, a, MI) \\
\iff &\forall m \in MI.\ \exists \Delta, \Gamma, q.\ \langle m, \langle \_,\_,\_,\_,\Delta,\Gamma\rangle, q, \_\rangle \in \sigma \wedge \Gamma(\Delta(q,(a,\mathsf{role}))) = (\top, \_) \\
&\mathsf{advanceS}(\langle m, \langle \_,\_,\_,\_,\Delta,\Gamma\rangle, q, \pi\rangle \cdot \sigma, m, \mathsf{role}, a) \\
= &\langle m, \langle \_,\_,\_,\_,\Delta,\_\rangle, \Delta(q,(a,\mathsf{role})), \pi \cdot (a,\mathsf{role})\rangle \cdot \sigma \\
&\mathsf{advance}(\sigma, \emptyset, a, \mathsf{role}) = \sigma \\
&\mathsf{advance}(\sigma, m \cdot MI, a, \mathsf{role}) = \mathsf{advance}(\mathsf{advanceS}(\sigma, m, a, \mathsf{role}), MI, a, \mathsf{role}) \\
&\mathsf{remove}(\sigma, m) = \{\left\langle i, FMU, \mathsf{role}, s^R, MI\right\rangle \mid \left\langle i, FMU, \mathsf{role}, s^R, MI \cup \{m\}\right\rangle \in \sigma\} \\
&\mathsf{registered}(\sigma, m) = \{\left\langle i, FMU, \mathsf{role}, s^R, MI \cup \{m\}\right\rangle \mid \left\langle i, FMU, \mathsf{role}, s^R, MI \cup \{m\}\right\rangle \in \sigma\} \\
&\mathsf{unregistered}(\sigma, m) = \sigma \setminus \mathsf{registered}(\sigma, m)
\end{aligned}
$$

Fig. 4: Auxiliary definitions.

whose execution transitions conf into conf′ and b is a boolean parameter set to $\top$ for no errors or $\bot$ for errors.

Intuitively, our DT is a program that, at some point, needs to load a new scenario, reassign the correct FMUs, and run the new OA. The monitor implementing the scenario. Thus, the language or library must support the following operations: it must be able to (a) load FMUs and scenarios, (b) assign an FMU to a role in a scenario, (c) read/write ports of an FMU, such that this operation fails if it does not follow the scenario, (d) advance time, and (e) check whether a loaded scenario is consistent after loading. In more detail, this operation is provided as follows in SMOL.

Loading The statement `obj=simulate("fmu",p = e)` loads an FMU at location `fmu` into the language and initialize its parameters `p`. Let $\mathsf{Init}(loc, v_1, \ldots, v_n)$ be the FMO generated by initializing the FMU at location *loc*, with a fresh identifier, a unique FMU identifier, and an empty set of monitor identifiers. A scenario is loaded with the statement `scn=scenario("fms"); (is)` and deactivated with `scn.stop();`. Let $\mathsf{Init}(loc)$ 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. 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(fmo)`. The scenario is then linked to the FMO and uses it according to the current value of its role.

Input/Output Each input and output port `v` is accessed using a field `fmo.v`, corresponding to the `set` (**set**) and `get` (**get**) functions on the associated FMU. Accessing the fields checks that all scenarios assigned to the FMO in question are allowed to do so. If the check fails, the scenario issues an error message. Our goal is to prevent these error messages actively, and for this purpose, we provide 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,is done with the method `fmo.step(e)`, where `e` must be an expression of type `Double`. Again, *all* scenarios the FMO is assigned to are consulted, and method `canStep()` is used check whether all scenarios this FMO participates allow it to perform a time advance next.

Consistent Algorithm The statement `fms.check()` invokes a call to UPPAAL to directly check the consistency of the OA used in the scenario.

We are interested in the following properties: (1) Each reachable state should only contain *FMS*, where the co-simulation trace π follows the OA. If this is the case, it is *reconfiguration consistent*. (2) We only use suitable algorithms in the FMS's. While (2) is directly ensured by `fms.check()`, property (1) follows from the following.

Theorem 1. *A FMS $\langle m, M, \pi\rangle$ is* correct*, if its trace π is accepted by M. A run without transitions annotated with $\bot$ ensures that all reached configurations contain only correct FMS's.*

The proof is straightforward, given the observation that if a transition is annotated with $\bot$, then it is not accepted by M, exactly the case that we aim to avoid. We require that in the simulation algorithm, every read, write, and time advance is guarded by the corresponding call to `canGet_v`, `canSet_v` and `canStep`, respectively. A DT that can reconfigure its simulation scenarios, thus, has to perform the following steps for reconfiguration: First, the new scenario is loaded, and the old one is deactivated. The new scenario `fms` is checked using `fms.check()`, if the check fails, an error is raised to the user. Second, all FMOs are assigned to the new scenario using `fms.assign`.

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 which scenarios an FMU is assigned to. Thus, it is possible to change the scenario on the fly by changing the assignment of FMUs to scenarios. Furthermore, one can transfer one FMU to another, meaning that results can be reused if the overall system, e.g., a DT, uses a multi-stage experiment where multiple scenarios are run sequentially.

An FMO can be assigned to multiple FMS objects, allowing scenarios to share an FMU if their operations intersect in allowed behavior. For example, a computationally heavy simulation FMU may be read in two scenarios, running only once, with the monitor ensuring correct usage according to both. Our approach offers greater flexibility than current co-simulation frameworks, which mandate static scenarios and prohibit FMU sharing. The scenario also describes how data is transmitted between the FMUs, but as this standard data flow problem is not specific to co-simulations, and we refrain from introducing it here.

5 Evaluation

This section showcases an application of the proposed approach to the power system described in Section 1. The FMUs are implemented using the FMI 2.0 standard [4] and available at https://anonymous.4open.science/r/SemanticObjects-34A8/.

Experimental Setup. The experimental setup uses a system that first simulates the left topology of Figure 1 and then the right topology, using the reconfiguration pattern described above. Each topology was run for a simulated time of 3s with a step size of 0.001s.

This setup was used in three different ways: (a) Monitoring disabled (**NOMON**), (b) Monitoring enabled for the entire simulation (**FULLMON**), and (c) Monitoring enabled only for the first iteration (**FIRSTMON**).

Results and Discussion. The results, averaged over 10 runs, are shown in Table 1. Monitoring causes a minor overhead for the reconfiguration itself (+11%), a small overhead in the first iteration (+105%), and a bigger one in the following ones (+271%) if the monitor is running.

Table 1: Monitoring overhead. All times are in ms.

	avg. reconf.	avg. 1st iter.	avg. other iters.
NOMON	4998	37	21
FULLMON	5535	76	78
Change	+11%	+105%	+271%
FIRSTMON	5535	76	21
Change	+11%	+105%	+0%

Reconfiguration is significantly slower than an iteration, as time is needed to load and initialize FMUs, which accounts for 89% The first iteration is generally slower, as the FMUs are initialized and the ports are accessed for the first time, causing additional computations. Here, the overhead caused by the monitoring is significant. Again, every first iteration corresponds to a reconfiguration, which is rare.

In later iterations, runtime monitoring causes a more significant overhead. However, as we pointed out, the iterations in the simulation scenario are the same – if the OA contains no branching inside its main loop, a consistency violation will be detected in the first iteration. Thus, a simple static analysis of the dynamically loaded OA (e.g., no branching in the main loop) can determine whether the monitoring should be turned off after the first iteration and only turned on reconfiguration. We evaluated this approach (**FIRSTMON**), where the total overhead in the experiment is merely 0.02%, making it suitable for digital twins and configuration exploration. In conclusion, most overhead is caused during the reconfiguration itself, which we expect to occur rarely. By using the redundancy between the iterations and using the runtime monitoring not at every iteration, but only occasionally, brings total runtime overhead down to an acceptable level. Considering the minor overhead which occurs mainly during reconfiguration and first iterations, we find it acceptable for our target applications: digital twins and configuration exploration.

6 Conclusion

The paper presents a monitoring approach to ensure that the orchestration algorithm and the simulation scenario are aligned throughout a co-simulation where the scenario may change in unforeseen ways to adapt to the physical counterpart of a digital

twin. This is achieved by using the scenario as a runtime monitoring specification rather than relying on domain-specific properties specified by an expert. The approach introduces a non-negligible overhead in short simulation. Nevertheless, the caching mechanisms mean that the overhead only affects a few simulation iterations. To adopt our approach, one can use the provided library, available on Maven. for easy integration into other JVM-based projects. In future work, beyond expanding experimental evaluation, we plan to employ language-based approaches, such as session types and typestate, to integrate simulation models and programming more closely.

References

1. A. Balakrishnan, J. Deshmukh, B. Hoxha, T. Yamaguchi, and G. Fainekos. PerceMon: Online Monitoring for Perception Systems. In *Runtime Verification*. Springer, Switzerland, Oct. 2021.
2. E. Bartocci, Y. Falcone, A. Francalanza, and G. Reger. Introduction to runtime verification. In *Runtime Verification*. Springer, 2018.
3. G. Behrmann, A. David, K. Larsen, J. Hakansson, P. Petterson, W. Yi, and M. Hendriks. UPPAAL 4.0. In *Proc. QEST 2006*. Springer, 2006.
4. T. Blockwitz, M. Otter, J. Åkesson, M. Arnold, C. Clauss, H. Elmqvist, M. Friedrich, A. Junghanns, J. Mauss, D. Neumerkel, H. Olsson, and A. Viel. Functional Mockup Interface 2.0: The Standard for Tool independent Exchange of Simulation Models. In *Proc. 9th International Modelica Conference*. Linköping University Electronic Press, 2012.
5. R. D. Christie. Power Systems Test Case Archive - UWEE, Mar. 2023. [Accessed 8. Mar. 2023].
6. A. Cimatti, C. Tian, and S. Tonetta. NuRV: A nuXmv extension for runtime verification. In *RV*, volume 11757 of *LNCS*, pages 382–392. Springer, 2019.
7. Y. Falcone, K. Havelund, and G. Reger. A tutorial on runtime verification. *Engineering Dependable Software Systems*, 34, 01 2013.
8. FMI. Functional mock-up interface tools, 2014.
9. C. Gomes, D. Broman, H. Vangheluwe, C. Thule, and P. G. Larsen. Co-simulation: A survey. *ACM Computing Surveys*, 51(3), 2018.
10. C. Gomes, L. Lucio, and H. Vangheluwe. Semantics of co-simulation algorithms with simulator contracts. In *Proc. ACM/IEEE MODELS'19*, , 2019. IEEE.
11. S. Hallé, C. Soueidi, and Y. Falcone. Leveraging runtime verification for the monitoring of digital twins. In *FMDT@FM*, volume 3507. CEUR-WS.org, 2023.
12. S. T. Hansen and P. C. Ölveczky. Modeling, algorithm synthesis, and instrumentation for co-simulation in maude. In *WRLA 2022, Munich, Germany*. Springer, 2022.
13. S. T. Hansen, C. Thule, C. Gomes, J. van de Pol, M. Palmieri, E. O. Inci, F. Madsen, J. Alfonso, J. Á. Castellanos, and J. M. Rodriguez. Verification and synthesis of co-simulation algorithms subject to algebraic loops and adaptive steps. *STTT*, 24(6), 2022.
14. E. Kamburjan and E. B. Johnsen. Knowledge structures over simulation units. In *ANNSIM*. IEEE, 2022.
15. E. Kamburjan, V. N. Klungre, R. Schlatte, E. B. Johnsen, and M. Giese. Programming and debugging with semantically lifted states. In *ESWC*, volume 12731 of *LNCS*. Springer, 2021.
16. E. Kamburjan, V. N. Klungre, R. Schlatte, S. L. T. Tarifa, D. Cameron, and E. B. Johnsen. Digital twin reconfiguration using asset models. In *ISoLA (4)*, volume 13704 of *LNCS*. Springer, 2022.
17. Z. Kazemi, A. A. Safavi, F. Naseri, L. Urbas, and P. Setoodeh. A secure hybrid dynamic-state estimation approach for power systems under false data injection attacks. *IEEE Transactions on Industrial Informatics*, 16(12), 2020.
18. R. Kübler and W. Schiehlen. Two methods of simulator coupling. *Mathematical and Computer Modelling of Dynamical Systems*, 6(2), 2000.

19. M. Leucker, M. Sachenbacher, and L. B. Vosteen. Digital twin for rescue missions - a case study. In *FMDT@FM*, volume 3507. CEUR-WS.org, 2023.
20. M. Marchiano, D. M. J. Rayworth, E. Alegria, and J. Undrill. Power generation load sharing using droop control in an island system. *IEEE Transactions on Industry Applications*, 54, 2018.
21. B. J. Oakes, C. Gomes, F. R. Holzinger, M. Benedikt, J. Denil, and H. Vangheluwe. Hint-based configuration of co-simulations with algebraic loops. In *Simulation and Modeling Methodologies, Technologies and Applications*, volume 1260. Springer, 2020.
22. B. Schweizer, P. Li, and D. Lu. Explicit and implicit cosimulation methods: Stability and convergence analysis for different solver coupling approaches. *Journal of Computational and Nonlinear Dynamics*, 10(5), 2015. Publisher: ASME.
23. A. Temperekidis, N. Kekatos, and P. Katsaros. Runtime verification for FMI-based co-simulation. In *Runtime Verification*. Springer, 2022.
24. D. Thoma, M. Sachenbacher, M. Leucker, and A. T. Ali. A digital twin for coupling mobility and energy optimization: The ReNuBiL living lab. In *FMDT@FM*, volume 3507. CEUR-WS.org, 2023.
25. C. Thule, K. Lausdahl, C. Gomes, G. Meisl, and P. G. Larsen. Maestro: The INTO-CPS co-simulation framework. *Simul. Model. Pract. Theory*, 92:45–61, 2019.
26. D. Weyns. Software engineering of self-adaptive systems. In *Handbook of Software Engineering*. Springer, 2019.
27. D. Weyns, M. U. Iftikhar, D. G. de la Iglesia, and T. Ahmad. A survey of formal methods in self-adaptive systems. In *CSSE*. ACM, jun 2012.
28. T. Wright, C. Gomes, and J. Woodcock. Formally verified self-adaptation of an incubator digital twin. In *ISoLA*. Springer, 2022.
29. E. Zapridou, E. Bartocci, and P. Katsaros. Runtime Verification of Autonomous Driving Systems in CARLA. In *Runtime Verification*. Springer, Cham, Switzerland, Oct. 2020.