A Formal Framework for Co-simulation

Simon Thrane Hansen[1] and Peter Csaba Ölveczky[2]

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

Abstract. Simulation-based analysis of cyber-physical systems are vital in the era of Industry 4.0. Co-simulation enables composing specialized simulation tools via a co-simulation algorithm. In this paper we provide a formal model in Maude of co-simulation for complex scenarios involving algebraic loops and step negotiation. We show not only how Maude can formally analyze co-simulations, but also how Maude can be used to synthesize co-simulation algorithms, port instrumentations, and parameter values such that the resulting co-simulation satisfies desired properties.

1 Introduction

Modern cyber-physical systems (CPSs), such as, e.g., nuclear power plants, cars, and airplanes, consist of multiple heterogeneous subsystems that are developed by different companies using different tools and formalisms [23]. These companies will usually not share their models for commercial reasons. There is nevertheless a need to determine how these subsystems interact and to explore and analyze different design choices as early as possible [20]. One way of addressing this need is to use, for each subsystem, an interface that provides an abstraction of that subsystem. *Simulation units* (SUs) provide such abstractions and are widely used in industry. A class of SUs are described by the Functional Mock-up Interface Standard [3] (FMI), which is used commercially and is supported by many tools [7]. An SU implements a well-defined interface and represents a subsystem by computing its behavioral trace using a dedicated solver.

Co-simulation [19,10] addresses the need to simulate a CPS given as the composition of such black-box SUs. Co-simulation transforms a continuous system to a discrete simulation with discrete interactions between the different SUs. Furthermore, a *digital twin* can be a co-simulation connected to a physical systems.

The objective of a co-simulation is to capture as accurately as possible the behavior of the modeled system. This is challenging, due to discretization, cyclic dependencies between the SUs, and the fact that very few assumptions be made about the SUs: an SU may, e.g., be unable to predict its future state at the next desired point in time. A *co-simulation algorithm* is responsible for orchestrating the interaction of the SUs: it determines how and when the different SUs interact.

Since the co-simulation algorithm should make the virtual system correspond to its physical counterpart, the virtual system can be analyzed, and different design choices can be explored, to accurately predict the behavior of the system to be built. However, the FMI standard is only informally described, and has been shown to be inconsistent [5]. For both of the above reasons, there is a need for formal methods to provide a formal semantics for co-simulation and to provide early model-based formal analysis of the co-simulations.

However, providing a formal semantics to co-simulation is challenging, due to, e.g., the complex behavior of the SUs, and the need to resolve cyclic dependencies between the SUs by fixed-point computations and to perform step negotiation to ensure that all SUs move in lockstep. Rewriting logic [21], with its modeling language and high-performance analysis tool Maude [6], should be a suitable formal method for co-simulation: Its expressiveness allows us to conveniently specify both complex dynamic behaviors and sophisticated functions (e.g., for detecting and resolving cyclic dependencies), and Maude provides automatic formal analysis capabilities for correctness analysis and design space exploration. Maude also supports connections to *external objects*, which means that Maude should be able to orchestrate the composition of real external components.

In this paper we present a formal framework for representing co-simulation in Maude. We give a formal model for co-simulation beyond the FMI 2.0 standard, also covering feed-through constraints, input instrumentations, and step rejection. We then use Maude to synthesize and execute suitable (scenario-specific) co-simulation algorithms, which enables the formal analysis of the resulting co-simulation. We also show how Maude can be used to synthesize instrumentations, parameter values, and co-simulation algorithms for such complex scenarios so that the resulting system satisfies desired properties. As discussed in Section 6, to the best of our knowledge this paper presents the first formal framework that covers design space exploration of complex co-simulation scenarios with algebraic loops and step rejection, and that also synthesizes correct-by-construction co-simulation algorithms and parameters for such scenarios.

From a Maude perspective, we found that using rewrite conditions in rules allowed us to easily and elegantly solve quite challenging problems in co-simulation.

The rest of the paper is structured as follows. Section 2 provides necessary background to Maude and co-simulation. Section 3 presents a Maude model of co-simulation scenarios and SU behaviors. Section 4 shows how correct-by-construction co-simulation algorithms can be synthesized and executed in Maude. Section 5 describes how to synthesize instrumentation and parameter values such that resulting co-simulation satisfies desired properties. Section 6 discusses related work and Section 7 gives some concluding remarks.

2 Preliminaries

This section gives some background on Maude and co-simulation.

2.1 Rewriting Logic and Maude

Maude [6] is a rewriting-logic-based executable formal specification language and high-performance analysis tool for object-based distributed systems.
A Maude module specifies a *rewrite theory* $(\Sigma, E \cup A, R)$, where:

– Σ is an algebraic *signature*; i.e., a set of *sorts*, *subsorts*, and *function symbols*.
– $(\Sigma, E \cup A)$ is a *membership equational logic* theory, with E a set of possibly conditional equations and membership axioms, and A a set of equational axioms such as associativity, commutativity, and identity, so that equational deduction is performed *modulo* the axioms A. The theory $(\Sigma, E \cup A)$ specifies the system's states as members of an algebraic data type.
– R is a collection of *labeled conditional rewrite rules* $[l] : t \longrightarrow t'$ **if** *cond*, specifying the system's local transitions.

A function f is declared `op` f `:` $s_1 \ldots s_n$ `->` s. Equations and rewrite rules are introduced with, respectively, keywords `eq`, or `ceq` for conditional equations, and `rl` and `crl`. A conditional rewrite rule has the form `crl` $[l]$ `:` t `=>` t' `if` c_1 `/\` $\ldots$ `/\` c_n, where the conditions $c_1, \ldots, c_n$ are evaluated from left to right. A condition c_i can be a Boolean term, an equation, a membership, or a *matching equation* $u(x_1, \ldots, x_n)$ `:=` u' with variables $x_1, \ldots, x_n$ not appearing in t and not instantiated in $c_1, \ldots, c_{i-1}$; these variables become instantiated by *matching* $u(x_1, \ldots, x_n)$ to the normal form of the (appropriate instance of) u'. c_i can also be a *rewrite condition* u_i `=>` u'_i, which holds if u'_i can be reached in zero or more rewrite steps from u_i. Mathematical variables are declared with the keywords `var` and `vars`, or can have the form *var*`:`*sort* and be introduced on the fly.

A *class* declaration `class` C `|` att_1 `:` s_1`, ... ,` att_n `:` s_n declares a class C of objects with attributes att_1 to att_n of sorts s_1 to s_n. An *object instance* of class C is represented as a term `<` O `:` C `|` att_1 `:` $val_1, \ldots, att_n$ `:` val_n `>`, where O, of sort `Oid`, is the object's *identifier*, and where val_1 to val_n are the current values of the attributes att_1 to att_n. A system state is modeled as a term of the sort `Configuration`, and has the structure of a *multiset* made up of objects and messages (and *connections* in our case).

The dynamic behavior of a system is axiomatized by specifying each of its transition patterns by a rewrite rule. For example, the rule (with label `l`)

```
rl [l] :  < O : C | a1 : f(x, y), a2 : O', a3 : z >
     =>   < O : C | a1 : x + z,   a2 : O', a3 : z > .
```

defines a family of transitions in which the attribute `a1` of object `O` is updated to `x + z`. Attributes whose values do not change and do not affect the next state, such as `a2` and the right-hand side occurrence of `a3`, need not be mentioned.

Formal Analysis in Maude. Maude provides a number of analysis methods, including rewriting for simulation purposes, reachability analysis, and linear temporal logic (LTL) model checking. The rewrite command `frew` *init* simulates one behavior from the initial state/term *init* by applying rewrite rules. Given an initial state *init*, a state pattern *pattern* and an (optional) condition *cond*, Maude's `search` command searches the reachable state space from *init* for all (or optionally a given number of) states that match *pattern* such that *cond* holds:

```
search init  =>!  pattern such that cond  .
```

The arrow `=>!` means that Maude only searches for *final* states (i.e., states that cannot be further rewritten) that match *pattern* and satisfies *cond*. If the arrow is `=>*` then Maude searches for all reachable states satisfying the search condition.

2.2 Co-simulation

Complex CPSs are typically composed of multiple communicating subsystems. For example, an autonomous car consists of multiple subsystems, including suspension, braking and collision avoidance systems. *Co-simulation* is a technique enabling the simulation of a complex CPS consisting of multiple simulation units (SUs), where each SU represents a subsystem and can compute the behavior of that system. An SU interacts with its environment through its ports [11,19].

A set of SUs can be composed into a *scenario* by *coupling* the input ports to output ports. A *coupling* connects one input port of an SU to an output port of another SU. The *coupling restriction* states that the value of an input and an output of a coupling must be the same at all times in the continuous system. However, in is its discrete counterpart, the coupling restrictions can only be satisfied at specific points in time called *communication points*. Therefore, each SU makes its own assumptions about the evolution of its input values between the communication points, which can introduce errors in the co-simulation [2].

The orchestrator computes the behavior of a scenario as a discrete trace while it tries to satisfy the coupling restrictions by exchanging values between the coupled ports. The orchestrator aims to find the communication points that minimize the co-simulation error while ensuring that the SUs move in lockstep by adapting to the behavior of the scenario. The optimal communication points depend on the SUs [13,22,14,24,12] since an SU, implementing error estimation, may conclude that the desired simulation resolution is too low and, therefore, *rejects* to compute a specific future state, something the orchestrator solves using *step negotiation* [16]. Furthermore, a scenario can contain *algebraic loops* (cyclic dependencies) between the SUs, which are resolved using fixed-point computations [19,22,16]. An algebraic loop is a consequence of the decomposition of a system into subsystems with couplings between SUs representing differential equations; examples include the suspension system of a car [18]. Scenarios with algebraic loops and step rejections are called *complex scenarios*. The following definition of an SU is based on [4,15,16]:

Definition 1 (Simulation Unit). *A Simulation Unit is a tuple*

$$\mathcal{SU} \triangleq \langle S, U, Y, \mathcal{V}, \mathtt{set}, \mathtt{get}, \mathtt{step} \rangle\,,$$

where:

- *S is a set, denoting the state space of the SU.*
- *U and Y are sets, of input and output ports, respectively. The union $VAR = U \cup Y$ of the inputs and outputs is called the ports of the SU.*
- *$\mathcal{V}$ is a set, intuitively denoting the values that a variable can hold. $\mathcal{V}_{\mathcal{T}} = \mathbb{R}_{\geq 0} \times \mathcal{V}$ is the set of timestamped values exchanged between input and output ports.*

- *The functions* $\texttt{set} : S \times U \times \mathcal{V}_{\mathcal{T}} \rightarrow S$ *and* $\texttt{get} : S \times Y \rightarrow \mathcal{V}_{\mathcal{T}}$ *sets an input and gets an output, respectively.*
- $\texttt{step} : S \times \mathbb{R}_{>0} \rightarrow S_c \times \mathbb{R}_{>0}$ *is a function; it instructs the SU to compute its state after a given duration. If an SU is in state* $s^{(t)}$ *at time* t *then,* $(s^{(t+h)}, h) = \texttt{step}(s^{(t)}, H)$ *denotes the state* $s_c^{(t+h)}$ *of the SU at time* $t + h$*, where* $h \leq H$*.*

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

$$\mathcal{S} \triangleq \langle C, \{\mathcal{SU}_c\}_{c\in C}, L, M, R, F\rangle$$

- C *is a finite set (of SU identifiers).*
- $\{\mathcal{SU}_c\}_{c\in C}$ *is a set of SUs, where each* $\mathcal{SU}_c = \langle S_c, U_c, Y_c, \mathcal{V}, \texttt{set}_c, \texttt{get}_c, \texttt{step}_c\rangle$*.*
- L *is a function* $L : U \rightarrow Y$*, 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 connected to the input* u*.*
- $M \subseteq C$ *denotes the SUs that implement error estimation.*
- $R : U \rightarrow \mathbb{B}$ *is a predicate, which provides information about the SUs' input approximation functions.* $R(u) = true$ *means that the function* $\texttt{step}$ *assumes that the timestamp* t_{SU} *of the SU of the* u *is smaller than the timestamp* t_v *of the timestamped value* $\langle t_v, v\rangle$ *set on the input* u*.*
- F *is a family of functions* $\{F_c : Y_c \rightarrow \mathcal{P}(U_c)\}_{c\in C}$*.* $u_c \in F_c(y_c)$ *means that the input* u_c feeds through *to the output* y_c *of the same SU. It means that there exists* $v_1, v_2 \in \mathcal{V}_{\mathcal{T}}$ *and* $s_c \in S_c$*, such that* $\texttt{get}_c(\texttt{set}_c(s_c, u_c, v_1), y_c) \neq \texttt{get}_c(\texttt{set}_c(s_c, u_c, v_2), y_c)$*.*

The function R represents the *instrumentation* of the scenario. We call an input port u *reactive* if $R(u)$, and *delayed* otherwise. Changing the instrumentation of a scenario changes the algorithm used to simulate the scenario due to the semantics of the actions. We assume that the instrumentation of a scenario is constant through the simulation, which is the case for most commercially used SUs [12].

Definition 2 extends the FMI 2.0 standard [3] with feed-through and port instrumentation to cover a broader class of co-simulation scenarios. Figure 1 shows a way to graphically present co-simulation scenarios.

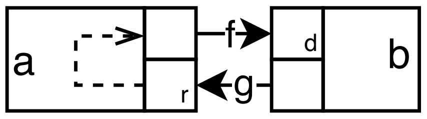

Fig. 1: A co-simulation scenario with two SUs a and b. The dashed arrows denote feed-through connections, the ports are represented as small squares, the instrumentation of an input port is denoted by the letters r (reactive) or d (delayed). The solid arrows f and g represent couplings.

2.3 Co-simulation Algorithms

An *orchestrator* simulates a scenario by executing a *co-simulation algorithm*. A co-simulation algorithm consists of an initialization procedure and a co-simulation step [3]. This work focuses on the co-simulation step, which we refer to as "the algorithm" in the paper.

The state of a co-simulation scenario is defined as the combination of the states of its subcomponents:

Definition 3 (Abstract SU State). *The observable abstract state s^R of an SU $\mathcal{SU}_c$ in a scenario $\mathcal{S}$ is an element of the set $S_c^R = \mathbb{R}_{\geq 0} \times S_{U_c}^R \times S_{Y_c}^R \times S_{V_c}^R$, where:*
- *$S_{U_c}^R : U_c \to \mathbb{R}_{\geq 0}$ is a function mapping each input port to a timestamp.*
- *$S_{Y_c}^R : Y_c \to \mathbb{R}_{\geq 0}$ is a function mapping each output port to a timestamp.*
- *$S_{V_c}^R : VAR_c \to \mathcal{V}$ is a function mapping each port to a value.*

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

We use the abstract state s_c^R of an SU c instead of the internal state s_c since the orchestrator cannot observe the latter.

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

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

A co-simulation step P is a sequence of operations that takes a co-simulation from one consistent state to another consistent state. We write $s \xrightarrow{\mathrm{P}} s'$ if executing the co-simulation step P from the state s results in the state s'.

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

$$\left\langle t, s_U^R, s_Y^R, s_V^R \right\rangle \xrightarrow{P} \left\langle t', s_U^{R'}, s_Y^{R'}, s_V^{R'} \right\rangle$$
$$\implies (\mathtt{consistent}(\left\langle t, s_U^R, s_Y^R, s_V^R \right\rangle) \implies (\mathtt{consistent}(\left\langle t', s_U^{R'}, s_Y^{R'}, s_V^{R'} \right\rangle) \wedge t' > t))$$

where consistent is defined as:

$$\begin{aligned}\mathtt{consistent}(\left\langle t, s_U^R, s_Y^R, s_V^R \right\rangle) \triangleq\ & (\forall u_c \in U \exists y_d \in Y \cdot L(u_c) = y_d) \\ & \wedge (\forall u_c, y_d \cdot L(u_c) = y_d \implies s_V^R(u_c) = s_V^R(y_d))\end{aligned}$$

Informally, the co-simulation step advances the scenario from an initial state at time t to a final state at time $t + H$, where $H > 0$, and ensures that the coupling restrictions are satisfied at both the initial and the final state.

Figure 2 shows three different co-simulation steps of the scenario in Fig. 1 that are allowed by the FMI standard 2.0 [3].

Algorithm 1

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

Algorithm 2

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

Algorithm 3

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

Fig. 2: Three co-simulation algorithms of the scenario in Fig. 1 conforming to the FMI Standard (version 2.0).

Although the three algorithms satisfy Definition 5 and consist of the same actions, they are not equivalent, and simulating with one algorithm instead of one of the others could change the co-simulation result as shown in [15,17]. To differentiate between them, we need to consider the semantics of the different actions described in Definition 1, which we describe in Definitions 6 to 8.

We base our semantics on [11,17] to which we refer for more information.

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

$$s^R \xrightarrow{\mathtt{get}(s^{(t)},y)} (v, s^{R'}) \implies \mathtt{preGet}(y, s^R) \land \mathtt{postGet}(y, s^R, s^{R'}, v)$$

Where:

$$\mathtt{preGet}(y, \langle t, s_U^R, s_Y^R, s_V^R \rangle) \triangleq s_Y^R(y) < t \land \forall u \in F(y) \cdot s_U^R(u) = t$$

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

$$\mathtt{postGet}(y, \langle t, s_U^R, s_Y^R, s_V^R \rangle, \left\langle t, s_U^R, s_Y^{R'}, s_V^R \right\rangle, v) \triangleq s_Y^{R'}(y) = t$$
$$\land \forall y_m \in (Y \setminus y) \cdot s_Y^{R'}(y_m) = s_Y^R(y_m)$$

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

$$s^R \xrightarrow{\mathtt{set}(s^{(t)},u,\langle t_v,x \rangle)} s^{R'} \implies \mathtt{preSet}(u, s^R) \land \mathtt{postSet}(u, v, s^R, s^{R'})$$

Where:

$$\texttt{preSet}(u, \langle t_v, x\rangle, \langle t, s_U^R, s_Y^R, s_V^R\rangle) \triangleq s_U^R(u) < t_v \\ \wedge ((R(u_c) \wedge s_U^R(u) = t) \vee (\neg R(u_c) \wedge s_U^R(u) < t))$$

The precondition says that the input must not have been assigned a new value since the SU was stepped, formally $s_U^R(u) < t_v$. Furthermore, it requires that the value $\langle t_v, x\rangle$ respects the instrumentation of the input. The postcondition (below) ensures the value and time of the input u is updated so it matches the value assigned on the input.

$$\texttt{postSet}(u, \langle t_v, x\rangle, \langle t, s_U^R, s_Y^R, s_V^R\rangle, \left\langle t, s_U^{R'} s_Y^R, s_V^{R'}\right\rangle) \triangleq t_v = s_U^{R'}(u) \\ \wedge (\forall u_m \in (U \setminus u) \cdot s_U^{R'}(u) = s_U^R(u)) \wedge s_V^{R'}(u) = x$$

Definition 8 (Step Computation). *To step an SU using* $\texttt{step}(s^{(t)}, H)$ *advances the state of the SU by H, formally:*

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

Where:

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

The precondition (above) states that all the SU's inputs have been updated according to their instrumentation. The postcondition (below) ensures that the time of the SU advances by H.

$$\texttt{postStep}(H, \langle t, s_U^R, s_Y^R, s_V^R\rangle, \left\langle t', s_{U_c}^R, s_Y^R, s_{V_c}^{R}{}'\right\rangle) \triangleq t + h' = t' \wedge h' \leq H$$

An algorithm P must satisfy Definition 5 while respecting the defined semantics. This means that Algorithm 3 is correct, while Algorithms 1 and 2 are incorrect since they do not respect the semantics.

2.4 Problem Statement

The two key problems in co-simulation that we addresses in this paper (in addition to the formalization of a co-simulation) are:

1. Given a scenario $\mathcal{S}$ synthesize a co-simulation algorithm P for $\mathcal{S}$. That is, find the sequence of SU actions P so that running P on $\mathcal{S}$ satisfies Definition 5 while respecting the semantics. This includes solving possible algebraic loops and performing step negotiation to ensure that all SUs move in lockstep.
2. Given a *parametric* or *partially instrumented* scenario $\mathcal{S}$, where some SU parameters are unknown and where the instrumentation is incomplete, i.e., not all input ports have been declared ether *reactive* or *delayed*, find concrete values for the parameters, and concrete instrumentation of the input ports, such that the resulting *instrumented* scenario represents a system with desirable properties.

3 Modeling Co-simulation Scenarios in Maude

This section describes how we model individual SUs and their composition in a co-simulation scenario in Maude. Due to space limitations, we only provide fragments of our Maude model. The entire model, including the synthesis and execution of co-simulation algorithms (Section 4) and the synthesis of instrumentations and parameters (Section 5) is available at `https://github.com/SimplisticCode/Co-simulation_WRLA` and consists of around 1400 LOC.

We formalize co-simulation scenarios in an object-oriented style. The state is a term {*SUs connections orchObjects*} of sort `GlobalState`, where *SUs* is set of objects modeling simulation units, *connections* denote the port couplings, and *orchObjects* are two additional objects used during synthesis and execution of co-simulation algorithms (see Section 4).

A simulation unit is modeled as an object instance of the following class:

```
class SU | time : Nat,                 inputs : Configuration,
           outputs : Configuration,    canReject : Bool,
           fmistate : fmiState,        parameters : LocalState,
           localState : LocalState .
```

The attribute `time` denotes the time of the SU; `inputs` and `outputs` denote the objects modeling the SU's input and output ports; `canReject` is `true` if the SU implements error estimation (i.e., is an element of the set M); `fmistate` denotes the *simulation mode* (see [3]) of the SU; `localState` denotes the SU's internal state; and `parameters` denotes the values of the SU's parameters.

Input and output ports are modeled as instances of the following classes:

```
class Port | value : FMIValue, time : Nat, status : PortStatus, type : FMIType .
class Input | contract : Contract .
class Output | dependsOn : OidSet .
subclasses Input Output < Port .
```

`value` and `time` denote, respectively, the value of the port and the time of its last set/get operation; `status` is `true` if the port was updated at the current time; `contract` denotes the input port's instrumentation (`delayed` or `reactive`); and `dependsOn` denotes the set of inputs that feed through to the output port.

Example 1. We illustrate our framework using a system where a *controller* controls the water level of a *water tank* with constant inflow of water, by opening and closing a valve at the bottom of the tank. The system can be modeled using one SU for the tank and one SU for the controller, and has the architecture shown in Fig. 1. The water tank (in its initial state) is modeled as an object

```
< "tank" : SU | parameters : ("flow" |-> <5>),  localState : ("waterlevel" |-> <0>),
                inputs : (< "valveState" : Input | value : <0>, time : 0, contract : delayed >),
                outputs : (< "waterlevel" : Output | value : <0>, time : 0,
                                                     status : Undef, dependsOn : empty >)
                time : 0,  canReject : false >
```

The `tank` has one delayed input port and one output port, and the local state indicates that the tank is empty. The parameter `flow` denotes the amount of water that flows into the tank per time unit.

To formalize the behaviors of an SU we formalize the operations `set`, `get`, and `step` in Definition 1. For example, the `get` operation that updates the `time` and `status` of a set of output ports is formalized as follows:[3]

```
op getAction : Object OidSet -> Object .
eq getAction(< SU1 : SU | >, empty) = < SU1 : SU | > .
eq getAction(< SU1 : SU | time : T,
                          outputs : (< O : Output | > OS) >, (O , P)) =
   getAction(< SU1 : SU | outputs :
                    (< O : Output | time : T, status : Def > OS) >, P) .
```

The application-specific behavior of an SU is given by defining its `step` function:

Example 2. The following definition of the `step` function in our running example defines how the water level of the tank changes as a function of the step duration `STEP`, the parameter `flow`, and the state (`value`) of the input `valve`:

```
eq step(< "tank" : SU | time : T, parameters : ("flow" |-> <FLOW>),
                        inputs : < "valve" : Input | value : <STATE> >,
                        outputs : < "waterlevel" : Output | time : T >,
                        localState : ("waterlevel" |-> <LEVEL>) >,
      STEP) =
  if STATE == 1 then    --- valve is open
    < "tank" : SU | time : (T+STEP), localState : ("waterlevel" |-> <0>),
          outputs : < "waterlevel" : Output | value : <0>, time : (T+STEP), status : Undef > >
  else                  --- valve is closed
    < "tank" : SU | time : (T+STEP),  localState : ("waterlevel" |-> <LEVEL+(STEP*FLOW)>),
                    outputs : < "waterlevel" : Output | value : < LEVEL + (STEP * FLOW) >,
                                                    time : (T + STEP), status : Undef > >
  fi .
```

A connection/coupling connecting the output port o of SU su_1 to the input port i of SU su_2 is represented by the term su_1 `!` o `==>` su_2 `!` i.

We define scenarios by defining constants `simulationUnits` and `externalConnection` that denote, resp., the simulation unit objects and their connections.

Example 3. The SUs and their couplings in our example are defined as follows:

```
eq simulationUnits =
   < "tank" : SU | parameters : ("flow" |-> <100>),  localState : ("waterlevel" |-> <0>),
                   time : 0,  fmistate : Instantiated, canReject : false,
                   inputs : (< "valveState" : Input | value : <0>, type : integer, time : 0,
                                                      contract : delayed, status : Undef >),
                   outputs : (< "waterlevel" : Output | value : <0>, type : integer, time : 0,
                                                        status : Undef, dependsOn : empty >)>
   < "ctrl" : SU | parameters : (("high" |-> <5>) , ("low" |-> <0>)), canReject : false,
                   localState : ("valve" |-> <false>), fmistate : Instantiated, time : 0,
                   inputs : (< "waterlevel" : Input | value : <0>, type : integer, time : 0,
                                                      contract : reactive, status : Undef >),
                   outputs : (< "valveState" : Output | value : <0>, type : integer, time : 0,
                                                        status : Undef, dependsOn : empty>)> .

eq externalConnection = ("tank" ! "waterlevel" ==> "ctrl" ! "waterlevel")
                        ("ctrl" ! "valveState" ==> "tank" ! "valveState") .
```

[3] We do not show variable declarations, but follow the convention that variables are written with capital letters.

The constant `setup` defines the initial state, and adds appropriate initialized orchestration objects to the scenario:

```
op setup : -> GlobalState .
ceq setup = {INIT}
  if SCENARIO := externalConnection simulationUnits
  /\ validScenario(SCENARIO)
  /\ LOOPS := tarjan(SCENARIO)
  /\ NeSUIDs := getSUIDsOfScenario(SCENARIO)
  /\ INIT := calculateSNSet(SCENARIO OData(1,LOOPS, NeSUIDs)) .
```

The function `validScenario` checks whether all inputs are coupled and that no input has two sources. The function `tarjan` returns (a possibly empty) set of algebraic loops in the scenario by searching for n on-trivial strongly connected components in the graph constructed using the rules in [15]. The function `getSUIDsOfScenario` returns the set of all SU identifiers. Finally, `calculateSNSet` checks if step negotiation should be applied in the simulation of the scenario, and generates a global initial state with orchestration objects that store information about the discovered algebraic loops and whether step negotiation is needed.

4 Synthesizing and Executing Co-simulation Algorithms

This section describes how co-simulation algorithms for a given scenario can be synthesized and then executed in Maude.

4.1 Orchestration Data

The *orchestration* executes a given co-simulation algorithm on a scenario, and requires keeping track of the co-simulation algorithm and the execution state.

The following class `SimData` stores such data about the simulation:

```
class SimData | SNSet : OidSet,            defaultStepSize : NzNat,
          actualStepSize : NzNat,          unsolvedSCC : AlgebraicLoopSet,
          solvedSCC : AlgebraicLoopSet,    guessOn : PortSet,
          values : PortValueMap,           simulationTime : Nat,
          suids : NeOidSet .
```

The attribute `SNSet` denotes the set M of SUs that implement error estimation and hence may reject to step the desired step size (see Definition 2); `defaultStepSize` is the default step duration of the simulation, and the attribute `actualStepSize` is the negotiated step duration. The attributes `actualStepSize` and `defaultStepSize` are equal if $M = \emptyset$. The attributes `unsolvedSCC` and `solvedSCC` respectively denote the solved and unsolved algebraic loops. The attribute `guessOn` denotes the set of ports which are used to solve algebraic loops using the technique described in [16]; `values` is a map linking an input port to a value. The orchestration uses `values` to track which values it has obtained but not set on an input port. The attribute `simulationTime` describes the current time of the simulation, and `suids` denotes the identifiers of the SUs.

The following class `AlgoData` stores the co-simulation algorithm:

```
class AlgoData | CosimStep : ActionList,   Initialization : ActionList,
                 Termination : ActionList, endTime : NzNat .
```

The attributes `Initialization` and `Termination` denote the initialization procedure and termination procedure, respectively. The attribute `CosimStep` denotes the co-simulation step procedure that the orchestration applies until it reaches the end time of the simulation (given by `endTime`). All elements of the algorithm are of the sort `ActionList`, which is a list of SU operations (where we do not show actions for handling complex scenarios):

```
ops Set Get Step Save : -> ActionType [ctor] .
op portEvent:_SU:_PId:_ : ActionType SUID OidSet -> Action [ctor] .
subsort Action < ActionList .
op emptyList : -> ActionList [ctor] .
op _;_ : ActionList ActionList -> ActionList [ctor assoc id: emptyList] .
```

4.2 Synthesis of Co-simulation Algorithms

We synthesize co-simulation algorithms for a scenario $\mathcal{S}$ by first performing and recording all possible SU actions, and then searching for consistent reachable final states. Any sequence of SU actions leading to such a state gives us a co-simulation algorithm.

A number of rewrite rules model the different SU actions. For example, the following rewrite rule describes a `get` operation:

```
crl [get-syn] :
    < SU1 : SU | fmistate : Simulation, inputs : IS,
                 outputs : (< O : Output | time : T,  status : Undef,
                                          value : V, dependsOn : FT > OS) >
    (SU1 ! O ==> SU2 ! I)
    < OCH : SimData | values : PV >
    < ALG : AlgoData | CosimStep : ALGO >
    =>
    getAction(< SU1 : SU | >, SU1 ! O)
    (SU1 ! O ==> SU2 ! I)
    < OCH : SimData | values : insert((SU2 ! I), < T ; V >, PV) >
    < ALG : AlgoData | CosimStep : (ALGO ; EVENT) >
  if feedthroughSatisfied(FT, IS, T)
    /\ EVENT := portEvent: Get SU: SU1 PId: O .
```

A value `V` is obtained from the output `O` of `SU1` if the state satisfies all feedthrough constraints `FT` of the output `O` (checked by `feedthroughSatisfied`). The rule updates the output `O` using the operation `getAction`, inserts the output's value and time `< T ; V >` into `values`, and adds the performed action `portEvent: Get SU: SU1 PId: O` to its list `CosimStep` of performed actions.

All such "synthesis" rules in our model follow the same pattern: they rewrite the scenario while remembering how they did it.

We synthesize a co-simulation algorithm by starting with a consistent initial state and exploring how a consistent final state can be established. An algorithm for a given scenario is therefore synthesized using the following rewrite rule:

```
crl [getAlgorithm]: {INIT} => {getOrchestrator(FINALSTATE)}
if isInitialState(INIT)
   /\ LOOPS := tarjan(INIT)
   /\ SUIDsNE := getSUIDsOfScenario(INIT)
   /\ SIMDATA := initialOrchestrationData(1,LOOPS,SUIDsNE)
   /\ CONF := calculateSNSet(INIT) SIMDATA initialAlgorithmData(1)
   /\ {CONF} => {FINALSTATE}
   /\ allSUsinUnloaded(SUIDsNE, FINALSTATE) .
```

This rule checks whether the scenario `INIT` is a suitable initial state using the predicate `isInitialState`. Then we construct an initial simulation configuration `CONF` as in Section 3. The key condition that does most of the work is the rewrite condition `{CONF} => {FINALSTATE}`, which searches for states reachable from `CONF` until it finds a state `FINALSTATE` that satisfies the property `allSUsinUnloaded`, which ensures that all SUs have been properly simulated and unloaded. The function `getOrchestrator` extracts the synthesized co-simulation algorithm from this final state.

The following Maude command then synthesizes all valid co-simulation algorithms for a given *scenario*:

```
Maude> search scenario => FINALSTATE:GlobalState .
```

Many SU actions can happen independently at the same time, which means that multiple valid algorithms often can be synthesized for a scenario. For example, there are six different co-simulation algorithms for our water tank scenario.

4.3 Executing Co-Simulation Algorithms

This section describes how co-simulation algorithms can be executed. The state of such an execution is a term `run:` *algorithm* `on:` *scenario* `with:` *simData*:

```
op run:_on:_with:_ : Object Configuration Object -> SimState [ctor].
```

A co-simulation algorithm is executed by sequentially performing its actions, starting with performing all actions in the `Initialization`, then performing all actions of the `CosimStep`, and finally executing all actions in `Termination`.

The following rule shows the execution of the first action (`Get`) in `CosimStep`:

```
crl [get-exec] :
    run: < ALG : AlgoData | CosimStep : (action: Get SU: SU1 PId: O) ; ALGO >
    on: CONF
      < SU1 : SU | inputs : IS,
                   outputs : (< O : Output | time : T, value : V, dependsOn : FT > OS) >
      ( SU1 ! O ==> SU2 ! INPUT)
    with: < OCH : SimData | values : PV >
    =>
    run: < ALG : AlgoData | CosimStep : ALGO >
    on: CONF getAction(< SU1 : SU | >, SU1 ! O) ( SU1 ! O ==> SU2 ! INPUT)
    with: < OCH : SimData | values : insert((SU2 ! INPUT), < T ; V >, PV) >
 if feedthroughSatisfied(FT, IS, T) .
```

We can combine algorithm synthesis and execution into the following rewrite rule, so that rewriting the term `runAnyAlgorithm` *scenario* synthesizes *and* executes a co-simulation algorithm the for co-simulation scenario *scenario*:

```
crl [runAlg] : runAnyAlgorithm INIT => run: ORC on: INIT with: SIMDATA
  if LOOPS := tarjan(INIT)
  /\ SUIDsNE := getSUIDsOfScenario(INIT)
  /\ SIMDATA := initialOrchestrationData(1,LOOPS,SUIDsNE)
  /\ ALGO := initialAlgorithmData(1)
  /\ CONF := calculateSNSet(INIT ALGO) SIMDATA
  /\ {CONF} => {FINALSTATE}
  /\ ORC := getOrchestrator(FINALSTATE)
  /\ allSUsinUnloaded(SUIDsNE, FINALSTATE) .
```

This rule is similar to the rule `getAlgorithm`, and also extracts the resulting algorithm `ORC` and simulation data `SIMDATA`.

Example 4. The water tank scenario described in Example 3 (`waterTankScenario` below) can be simulated by rewriting:

```
Maude> frew (runAnyAlgorithm waterTankScenario) .
```

The command returns the final simulation state

```
run: < "Algorithm" : AlgorithmData | CosimStep : emptyList, Initialization : emptyList,
                                     Termination : emptyList, endTime : 1 >
on: ("tank" ! "waterlevel" ==> "ctrl" ! "waterlevel")
    ("ctrl" ! "valveState" ==> "tank" ! "valveState")
    < "ctrl" : SU | canReject : false, fmistate : Unloaded, time : 1,
                    inputs : < "waterlevel" : Input | contract : reactive, status : Def,
                                                      time : 1, type : integer, value : < 100 > >,
                    outputs : < "valveState" : Output | dependsOn : empty, status : Def,
                                                        time : 1, type : integer, value : < 1 > >,
                    localState : "valve" |-> < true >,
                    parameters :"high" |-> < 5 >, "low" |-> < 0 >  >
    < "tank" : SU | canReject : false, fmistate : Unloaded, time : 1,
                    inputs : < "valveState" : Input | contract : delayed, status : Def,
                                                      time : 1, type : integer, value : < 1 > >,
                    outputs : < "waterlevel" : Output | dependsOn : empty, status : Def,
                                                        time : 1, type : integer, value : < 100 > >,
                    localState : "waterlevel" |-> < 100 >,
                    parameters : "flow" |-> < 100 > >
with: < "Orchestrator" : SimulationData | SNSet : empty,       actualStepSize : 1,
                                          defaultStepSize : 1, guessOn : empty,
                                          simulationTime : 1,  solvedSCC : empty,
                                          unsolvedSCC : empty, values : empty,
                                          suids :("ctrl", "tank") >
```

4.4 Checking Confluence of Synthesized Co-simulation Algorithms

Executing all valid co-simulation algorithms for a given scenario should give the same result. The following Maude search command checks whether all generated co-simulation algorithms for a scenario *scenario* result in the same final state:

```
Maude> search (runAnyAlgorithm scenario) =>! S:SimState .
```

This search command synthesizes and then executes all co-simulation algorithms for the scenario *scenario*. For our water tank scenario, the search produces a single result, which means that all synthesized algorithms give the same result.

5 Synthesizing Instrumentations and SU Parameters

Our framework makes possible different kinds of design space exploration to allow the practitioner to see how different design choices affect the behavior of the system. This section shows how our framework can be used to synthesize parameter values and instrumentations of the inputs that lead to desired simulations.

5.1 Instrumentation of a Scenario

Finding a good instrumentation of the input ports (i.e. deciding whether an input port should be `reactive` or `delayed`) is important not only to achieve accurate co-simulation results [13,22,17], but also because some instrumentations of a scenario may lead to algebraic loops while others do not.

We use reachability analysis to explore the consequences of different instrumentations of a scenario to find the instrumentation that yields the desired simulation results. To explore different instrumentations of a scenario, we create a *partially instrumented* scenario, where the some of the input ports have the contract `noContract` instead of `reactive` or `delayed`.

Example 5. The following water tank scenario is partially instrumented:

```
eq waterTankNotInstrumented =
< "tank" : SU | parameters : ("flow" |-> < 100 >),  localState : ("waterlevel" |-> < 0 >),
              time : 0,  fmistate : Instantiated, canReject : false,
              inputs : (< "valveState" : Input | value : < 0 >, type : integer, time : 0,
                                                contract : noContract , status : Undef >),
              outputs : (< "waterlevel" : Output | value : < 0 >, type : integer, time : 0,
                                                  status : Undef, dependsOn : empty >) >

< "ctrl" : SU | parameters : (("high" |-> < 5 >) , ("low" |-> < 0 >)), canReject : false,
              localState : ("valve" |-> < false >), fmistate : Instantiated, time : 0,
              inputs : (< "waterlevel" : Input | value : < 0 >, type : integer, time : 0,
                                                contract : noContract , status : Undef >),
              outputs : (< "valveState" : Output | value : < 0 >, type : integer, time : 0,
                                                  status : Undef, dependsOn : empty >) > .
```

A partially instrumented scenario has the form `findInstr`(*scenario*) and becomes an ordinary scenario when all ports have been instrumented (rule `remove-findInstr`). The rules `instr-delayed` and `instr-reactive` set uninstrumented input ports to be either `delayed` or `reactive`:

```
rl [instr-delayed]:
 findInstr(< SU1 : SU | inputs : < I : Input | contract : noContract > IS > C)
=> findInstr(< SU1 : SU | inputs :  < I : Input | contract : delayed > IS > C) .

rl [instr-reactive]:
 findInstr(< SU1 : SU | inputs : < I : Input | contract : noContract > IS > C)
=> findInstr(< SU1 : SU | inputs : < I : Input | contract : reactive > IS > C) .

crl [remove-findInstr]: findInstr(CONF) => CONF if instrumented(CONF) .
```

The different instrumentations of a partially instrumented scenario are found and explored using the following rule:

```
crl [findInstrumentation]: findContracts(INIT) => CONF
  if findInstr(INIT) => CONF
  /\ empty == tarjan(CONF)            *** no algebraic loops
  /\ runAnyAlgorithm CONF => run: ORC on: FINAL with: SIMDATA
  /\ simulationFinished(ORC)
  /\ desiredProperty(FINAL) .
```

This rule generates an instrumented scenario `CONF` from the partially instrumented scenario `INIT`. `CONF` is then simulated (in the rewrite condition), leading to a final state `FINAL`. The instrumentation can be restricted by giving properties that the instrumented scenario `CONF` and/or the simulation result `FINAL` must satisfy. For example, the condition `empty == tarjan(CONF)` says that the instrumentation should not lead to algebraic loops, and the last conjunct in the condition says that the simulation result `FINAL` must satisfy *desiredProperty*.

Example 6. We define *desiredProperty* to be that the water level of the tank is in a desired range. The following Maude command then finds all instrumentations which lead to simulations which end in a desired water level:

```
Maude> search findContracts(waterTankNotInstrumented) =>! C:Configuration .
```

This command returns the three instrumentations (with parts replaced by '`...`')

```
Solution 1
C:Configuration --> ...
< "ctrl" : SU | inputs : < "waterlevel" : Input | contract : delayed > ... >
< "tank" : SU | inputs : < "valveState" : Input | contract : reactive > ... >

Solution 2
C:Configuration --> ...
< "ctrl" : SU | inputs : < "waterlevel" : Input | contract : reactive > ... >
< "tank" : SU | inputs : < "valveState" : Input | contract : delayed > ... >

Solution 3
C:Configuration --> ...
< "ctrl" : SU | inputs : < "waterlevel" : Input | contract : delayed > ... >
< "tank" : SU | inputs : < "valveState" : Input | contract : delayed > ... >
```

5.2 Synthesizing SU Parameters

An SU may have different parameters. In our framework, the user can specify a finite set of possible values for a parameter using a `choose` operator, and we can then synthesize those parameter values that result in desired simulations.

Example 7. We want to synthesize the value of the parameter `flow` of the water tank such that the water level is above 10 in the final simulation state. The following predicate defines the desired water level:

```
op above10 : Configuration -> Bool .
eq above10(CONF < "tank" : SU | localState : "waterlevel" |-> < V > >) = V > 10 .
```

To synthesize a `flow` value from the set $\{1, 2, 30\}$ we initialize `flow` accordingly:

```
< "tank" : SU | parameters : "flow" |-> choose(< 1 >, < 2 >, < 30 >), ... >
```

We use the following rule to synthesize parameter values that result in a simulations that satisfy `above10`:

```
crl [getParamValues] : selectParams(UNITIALIZEDCONF) => CONF
  if UNITIALIZEDCONF => CONF
  /\ runAnyAlgorithm CONF =>
       run: < ALG : AlgData | Initialization : emptyList,
                               CosimStep : emptyList , Termination : emptyList >
       on: FINALSTATE with: SIMULATIONDATA
  /\ above10(FINALSTATE) .
```

The following Maude command gives all initialized scenarios which lead to desired simulations:

```
Maude> search selectParams(parametricWaterTank) =>! C:Configuration .

Solution 1
C:Configuration --> ... < "tank" : SU | parameters : "flow" |-> <30>, ... >

No more solutions.
```

We can also *simultaneously* synthesize both desired instrumentations *and* parameter values by having `noContract` ports and `choose`(...) values.

6 Related Work

A number of papers, e.g. [15,11,4,16], synthesize co-simulation algorithms for *fixed* scenarios. In contrast to our paper, this body of work does not provide *formal models* of co-simulation, and therefore no formal analysis. We exploit Maude's formal analysis features to synthesize suitable instrumentations and SU parameters, which is not addressed by the mentioned related work.

Design space exploration of SU parameters is described in [8,9]. This work uses genetic algorithms to find optimal parameters values. However, it does not consider how different instrumentations can affect the simulation result.

Formal methods have been used for co-simulation, e.g., [25,1,5,26,17]. Thule et al. [25] formalize a given scenario and two given co-simulation algorithms for that scenario in PROMELA and use the SPIN model checker to compare the two simulation algorithms, e.g., in terms of reachability. In contrast, we provide a general formal framework for co-simulation, synthesize co-simulation algorithms for a given scenario, synthesize instrumentations and parameter values, and capture a broader class of co-simulation scenarios (e.g., including scenarios with algebraic loops and step rejection) than those in the case study in [25].

Cavalcanti et al. [5] provide the first behavioral semantics of FMI. They show how to prove essential properties of co-simulation algorithms using CSP, and also show that the co-simulation algorithm provided in the FMI standard is not consistent. We cover an extension of FMI scenarios, and also include feed-through, step rejection, and input port instrumentation. Furthermore, as already mentioned, we also synthesize co-simulation algorithms and parameters.

Amálio et al. [1] show how formal tools can detect algebraic loops in a scenario. We not only detect such loops, but also solve them to synthesize co-simulation algorithms. Zeyda et al. [26] formalize a co-simulation scenario in Isabelle/UTP, and prove different properties–including behavioral properties–about the scenario. In contrast, we use automatic model checking methods to both synthesize and analyze co-simulation algorithms, and also cover complex scenarios (algebraic loops, step rejection, etc.) not covered in [26].

7 Concluding Remarks

This work presented a formal model of co-simulation in Maude for complex scenarios with algebraic loops and step negotiation. Furthermore, we showed how Maude could be used to synthesize and execute co-simulation algorithms enabling synthesis of port instrumentations and parameter values, such that the resulting scenario possesses desirable properties.

Future work include more advanced case studies, integration with *external* components, and symbolic methods to search for parameter values.

Acknowledgements: We will like to thank Claudio Gomes, Jaco van de Pol, Stefan Hallerstede, and Jose Meseguer for their support and feedback on the work.

References

1. Amálio, N., Payne, R.J., Cavalcanti, A., Woodcock, J.: Checking SysML Models for Co-simulation. In: Ogata, K., Lawford, M., Liu, S. (eds.) Formal Methods and Software Engineering - 18th International Conference on Formal Engineering Methods, ICFEM 2016, Proceedings. vol. 10009, pp. 450–465 (2016)
2. Arnold, M., Clauß, C., Schierz, T.: Error analysis and error estimates for co-simulation in FMI for model exchange and co-simulation v2.0. In: Progress in Differential-Algebraic Equations. pp. 107–125. Springer Berlin Heidelberg (2014), event-place: Berlin, Heidelberg
3. 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: Proceedings of the 9th International MODELICA Conference, Munich, Germany. vol. 76, pp. 173–184. Linköping University Electronic Press (2012)
4. Broman, D., Brooks, C.X., Greenberg, L., Lee, E.A., Masin, M., Tripakis, S., Wetter, M.: Determinate composition of FMUs for co-simulation. In: Ernst, R., Sokolsky, O. (eds.) Proceedings of the International Conference on Embedded Software, EMSOFT 2013, Montreal, QC, Canada. pp. 2:1–2:12. IEEE (2013)
5. Cavalcanti, A., Woodcock, J., Amálio, N.: Behavioural models for FMI co-simulations. In: Sampaio, A., Wang, F. (eds.) Theoretical Aspects of Computing - ICTAC 2016, Proceedings. Lecture Notes in Computer Science, vol. 9965, pp. 255–273 (2016)

6. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L. (eds.): All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic, Lecture Notes in Computer Science, vol. 4350. Springer (2007)
7. FMI: Functional mock-up interface tools (2014), https://fmi-standard.org/tools/
8. Gamble, C.: Design Space Exploration in the INTO-CPS Platform: Integrated Tool chain for model-based design of Cyber Physical Systems. Tech. rep. (Oct 2016)
9. Gamble, C., Pierce, K.: Design space exploration for embedded systems using co-simulation. In: Fitzgerald, J., Larsen, P.G., Verhoef, M. (eds.) Collaborative Design for Embedded Systems: Co-modelling and Co-simulation. Springer (2014)
10. Gomes, C., Broman, D., Vangheluwe, H., Thule, C., Larsen, P.G.: Co-simulation: A survey. ACM Computing Surveys **51**(3) (2018)
11. Gomes, C., Lucio, L., Vangheluwe, H.: Semantics of Co-Simulation Algorithms with Simulator Contracts. In: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C). pp. 784–789. IEEE (2019)
12. Gomes, C., Meyers, B., Denil, J., Thule, C., Lausdahl, K., Vangheluwe, H., De Meulenaere, P.: Semantic Adaptation for FMI Co-simulation with Hierarchical Simulators. Simul. **95**(3), 1–29 (2019)
13. Gomes, C., Oakes, B., Moradi, M., Gámiz, A., Mendo, J., Dutré, S., Denil, J., Vangheluwe, H.: HintCO – Hint-based Configuration of Co-simulations. In: Proceedings of the 9th International Conference on Simulation and Modeling Methodologies, Technologies and Applications. pp. 57–68. SCITEPRESS - Science and Technology Publications (2019)
14. Gomes, C., Thule, C., Lausdahl, K., Larsen, P.G., Vangheluwe, H.: Stabilization technique in INTO-CPS. In: 2nd Workshop on Formal Co-Simulation of Cyber-Physical Systems. vol. 11176. Springer, Cham (2018)
15. Gomes, C., Thule, C., Lúcio, L., Vangheluwe, H., Larsen, P.G.: Generation of co-simulation algorithms subject to simulator contracts. In: Camara, J., Steffen, M. (eds.) Software Engineering and Formal Methods. pp. 34–49. Lecture Notes in Computer Science, Springer International Publishing (2020)
16. Hansen, S.T., Gomes, C., Larsen, P.G., van de Pol, J.: Synthesizing Co-Simulation Algorithms with Step Negotiation and Algebraic Loop Handling. In: Martin, C.R., Blas, M.J., Inostrosa-Psijas, A. (eds.) Annual Modeling and Simulation Conference, ANNSIM 2021. pp. 1–12. IEEE (2021)
17. Hansen, S.T., Gomes, C., Palmieri, M., Thule, C., van de Pol, J., Woodcock, J.: Verification of Co-simulation Algorithms Subject to Algebraic Loops and Adaptive Steps. In: Lluch Lafuente, A., Mavridou, A. (eds.) Formal Methods for Industrial Critical Systems. pp. 3–20. Springer International Publishing, Online (2021)
18. Hansen, S.T., Thule, C., Gomes, C.: An FMI-Based Initialization Plugin for INTO-CPS Maestro 2. In: Cleophas, L., Massink, M. (eds.) Software Engineering and Formal Methods. SEFM 2020 Collocated Workshops - ASYDE, CIFMA, and CoSim-CPS, Revised Selected Papers. vol. 12524, pp. 295–310. Springer (2020)
19. Kübler, R., Schiehlen, W.: Two methods of simulator coupling. Mathematical and Computer Modelling of Dynamical Systems **6**(2), 93–113 (2000)
20. Lee, E.A.: Cyber physical systems: Design challenges. In: 2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC). pp. 363–369 (2008)
21. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science **96**(1), 73–155 (1992)

22. 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, pp. 1–28. Springer International Publishing (2020)
23. Paris, T., Wiart, J., Netter, D., Chevrier, V.: Teaching co-simulation basics through practice. In: Durak, U. (ed.) Proceedings of the 2019 Summer Simulation Conference, SummerSim 2019. pp. 31:1–31:12. ACM (2019)
24. 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), 051007 (2015)
25. Thule, C., Gomes, C., DeAntoni, J., Larsen, P.G., Brauer, J., Vangheluwe, H.: Towards the verification of hybrid co-simulation algorithms. In: Software Technologies: Applications and Foundations - STAF 2018 Collocated Workshops. vol. 11176, pp. 5–20. Springer (2018)
26. Zeyda, F., Ouy, J., Foster, S., Cavalcanti, A.: Formalising Cosimulation Models. In: Cerone, A., Roveri, M. (eds.) Software Engineering and Formal Methods - SEFM 2017 Collocated Workshops: DataMod, FAACS, MSE, CoSim-CPS, and FOCLASA, Revised Selected Papers. Lecture Notes in Computer Science, vol. 10729, pp. 453–468. Springer (2017)