Semantics of Co-simulation Algorithms with Simulator Contracts

Cláudio Gomes
University of Antwerp, Flanders Make vzw,
Antwerp, Belgium
claudio.gomes@uantwerp.be

Levi Lúcio
fortiss,
Munich, Germany
lucio@fortiss.org

Hans Vangheluwe
University of Antwerp, Flanders Make vzw,
Antwerp, Belgium
hans.vangheluwe@uantwerp.be

***Abstract*—The rapid adoption of co-simulation techniques allows for holistic complex system development. However, ensuring trustworthy results when combining simulators requires a careful consideration of their implementation and capabilities. Especially in black box integration, these are frequently left implicit.**

In this paper, we explore a way to account for simulator capabilities, by formalizing the execution of a co-simulation that respects such contracts. This formalization is specific to two kinds of contracts, but could serve as a basis to a general approach to black box co-simulation. An example application of the semantics to generate master algorithms is presented.

***Index Terms*—co-simulation, prolog, contract-based design, constraint solving.**

I. Introduction

Traditional modeling and simulation techniques, where the meaning of a model is described by a solver algorithm, are no longer sufficient to foster integrated development processes of complex systems [1]. To understand why, note that these techniques require a model of the system, the construction of which becomes a challenge due to the following factors: (i) heterogeneous systems are best modeled with a mix of formalisms [2], often creating the need for a hybrid model conforming to an *ad-hoc* formalism that is specific to the models being combined [3]; (ii) some parts of the system may be difficult to model accurately, because the former might be an assemblage of components supplied by third party companies, with little interest in sharing detailed models of their Intellectual Property (see, e.g., [4]). In short, complex systems (item (i) above) impose complex development processes (item (ii)).

While the study of model integration techniques can tackle the first item, the second item restricts the scope to black box model integration. Co-simulation is a technique to combine multiple black-box simulators, each responsible for a model, in order to compute the behavior of the combined models over time [5]. The simulators, developed independently of each other, are coupled using a master algorithm that communicates with each simulator via its interface. This interface comprises functions for setting/getting inputs/outputs, and computing the associated model behavior over a given interval of time. An example of such interface is the Functional Mockup Interface (FMI) Standard [6], [7].

C. G. is a FWO Research Fellow, at the University of Antwerp, supported by the Research Foundation - Flanders (File Number 1S06316N). We thank the organizers of the CAMPaM workshop for providing a platform where the ideas described here flourished. Additionally, we thank the reviewers for the thorough feedback.

Co-simulation is therefore a promising solution to evaluate complex systems under complex development processes: • Systems which are best represented by diverse sub-models can be simulated using the most appropriate simulation tool for each sub-model; and • Systems assembled from externally supplied components can be simulated by integrating black-box simulators, provided by those component manufacturers. These advantages have led to a proliferation of tools that support the import/export of simulators implementing the FMI Standard, called Functional Mockup Units (FMUs), and an increasing number of applications [8], [9].

This rapid adoption is hindered by the little guarantees on the correctness of the results [10]. Traditionally, co-simulations were developed among pairs of simulators, by practitioners that have some control over how those simulators are implemented. Now, practicioners have less control over how FMUs are implemented. Indeed, a recent empirical survey has shown that practitioners still experience difficulties in the configuration of co-simulations [9], [11]. Moreover, recent work [12] shows that one of the reasons for these difficulties is the lack of information about the implementation of each FMU, which constraints the ways they can be interacted with (see Section II for other sources of errors and related work).

a) Contribution: In this paper, we formalize: (i) the constraints imposed by a restricted set of FMU contracts, and (ii) the meaning of a family of fixed step size co-simulation algorithms that respect such contracts. We focus on the FMI standard because it allows for simulators to be represented as black boxes. While our long term research goal is to consider arbitrary contracts, in this paper, we restrict our attention to input approximation and output calculation contracts. These contracts correspond to a partial view of how the FMUs implement their input approximation schemes and the algebraic dependencies used to calculate the outputs. Hence, the contracts do not expose intellectual property. As we argue next, respecting these contracts is a necessary (but not sufficient) condition to obtaining correct results.

b) Prior Work: The need for these contracts has been identified in prior work [13] and an incomplete solution is advanced in [12]. The solution proposed in [13] works under the assumptions that FMUs have the same contract for every input (because it assumes FMUs have a single vector input/output), and the solution described in [12] neglects how the outputs are computed [12, Assumption 2]. The current manuscript addresses these omissions and sketches a framework that can be extended to more complex contracts (Section V discusses some of these).

c) Structure: The next section introduces preliminary concepts and related work. Section III explores the research problem and Section IV describes our contribution. Finally, Section V discusses and concludes.

II. Sources of Error in Co-simulation

This section introduces background concepts in an informal manner, and discusses some of the sources of error.

A. Background

We adopt the definitions and nomenclature introduced in [10] and refer the reader to it for a rigorous exposition.

A co-simulation is the behaviour trace of a coupled system, produced by the coordination of FMUs. The behaviour trace is a function mapping time to values, representing the timestamped outputs of each FMU. An example behaviour trace is shown at the bottom of Figure 1.

An FMU is an executable software entity responsible for simulating a part of the system. To communicate with other FMUs, each FMU implements the Functional Mockup Interface (FMI) Standard [7]. This allows a master algorithm, described below, to communicate with it. The main functionality of an FMU is encoded in three main C functions: a function to set inputs, a function to perform a step with a given step size, and a function to get outputs.

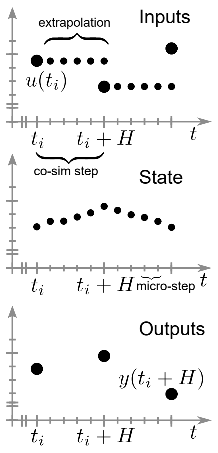

Figure 1: Simulator behavior.

A master is a software component that sets/gets inputs/outputs of each FMU and asks it to estimate the state of its allocated subsystem at a future time. For example, in Figure 1, the master sets an input to the FMU at time t_i, and asks it to compute the state at time t_i+H, where $H > 0$ is denoted the step size. The FMU in turn might perform multiple micro-steps and employ an input approximation scheme (this computation is hidden from the master). Then, once the FMU has reached time $t_i + H$, the master requests an output, illustrated at the bottom of the figure.

The master follows the co-simulation scenario to know the order in which to ask the FMU to simulate and where to copy their outputs. A co-simulation scenario, or just scenario, is a description of how the FMUs are interconnected and the configuration of the co-simulation, e.g. step size. Example master algorithms and scenarios are presented in Section III.

B. Related Work

In order to focus only on the errors caused by the co-simulation, we make the assumption that the models and corresponding FMUs are correct and consistent with each other[1]. Under this assumption, the error in the co-simulation can come from input approximations and failure to detect and handle discontinuities in the behavior computed by the FMUs.

1) Input Approximation: As discussed in the previous section, in between communication times, each FMU performs an approximation of its input (recall Figure 1). Any error in these approximations translates into errors in the internal state approximation, which in turn translates into errors in the outputs produced. Naturally, the larger the communication step size, the larger the input approximation error might be.

Traditional convergence analysis shows that, in the absence of algebraic loops, the growth rate of the error in the co-simulation is dominated by the growth rate of the error in the input approximation functions, which in turn is dependent on the size of the co-simulation step [5], [14].

These results apply only to continuous co-simulations, that is, co-simulation that comprise FMUs of continuous models. We next look at the errors introduced in hybrid co-simulations.

2) Event Detection: Hybrid co-simulations are those that comprise continuous, discrete, and hybrid FMUs. A hybrid FMU computes the solution to a hybrid model, which exhibits continuous behavior, interleaved with discrete changes [15].

When a discrete change happens in a hybrid FMU in between communication times, the other FMUs need to know about that change, as it can affect their inputs. If they only know about that change in the next communication time, the error might be so big that it renders the results useless (see e.g., [16]).

Much work has focused on producing masters that ensure the other FMUs know the time at which the discontinuity happened. For instance, the works in [17]–[21] focus on the correct synchronization of a discrete event simulator with a continuous simulator. This seminal work assumes a standard synchronization algorithm, where, in the presence of possible state events, the discrete simulator is always one step behind the continuous simulator, to avoid rollbacks.

Instead of enforcing a correct synchronization, some work has focused on finding the maximum allowed delay in the event detection. For instance, the work in [16] explores how the energy of a hybrid system can be increased when state events are not accurately reproduced by the co-simulation. It presents a way to find the largest co-simulation step that prevents this from happening.

III. Problem Statement

In this section, we show, through a simple but representative example, that controlling the error in the co-simulation involves a careful consideration of both master and FMU

[1]As one reviewer pointed out, it may happen that a model makes an assumption about another model, which may not be true during the co-simulation, leading to nonsensical results. We exclude these scenarios although they may happen in practice.

implementations. We first formalize the concept of FMU, and we then show that we need the notion of contract to reduce the error in a co-simulation.

Definition 1. An FMU with identifier c is a structure $\langle S_c, U_c, Y_c, \mathtt{set}_c, \mathtt{get}_c, \mathtt{doStep}_c \rangle$, where: • S_c represents the state space; • U_c and Y_c the set of input and output variables, respectively; • $\mathtt{set}_c : S_c \times U_c \times \mathcal{V} \to S_c$ and $\mathtt{get}_c : S_c \times Y_c \to \mathcal{V}$ are functions to set the inputs and get the outputs, respectively (we abstract the set of values that each input/output variable can take as $\mathcal{V}$); and • $\mathtt{doStep}_c : S_c \times \mathbb{R}_{\geq 0} \to S_c$ is a function that instructs the FMU to compute its state after a given time step.

Note the black box nature of this definition. The setting of an input $s_c^{(1)} = \mathtt{set}_c(s_c^{(0)}, u_u, v)$ changes the internal state of the FMU from $s_c^{(0)}$ to $s_c^{(1)}$, to reflect the fact that the input value v has been recorded for input variable u_u. We use the notation $s_c^{(0)}, s_c^{(1)}, \ldots$ to stress the transformations on the internal state of the FMU. The index is independent of the co-simulation time, so the state can undergo multiple transformations at the same co-simulation time. The stepping function $s_c^{(1)} = \mathtt{doStep}_c(s_c^{(0)}, H)$ computes a new state $s_c^{(1)}$, representing the internal state of the FMU after H units of time from the state $s_c^{(0)}$. That is, if an FMU is in state $s_c^{(0)}$ at time t, $\mathtt{doStep}_c(s_c^{(0)}, H)$ approximates the behavior of the corresponding model at time $t + H$. The result of this approximation is encoded in state $s_c^{(1)}$. If this model is a continuous one, the FMU will internally approximate the evolution in the interval $[t, t + H]$, using an estimation function to estimate the values of the inputs in that interval (recall Figure 1). In our notation, we choose to leave this function implicit in the $\mathtt{doStep}_c$, as reflected in the current version of the FMI Standard. There is no restriction as to when should the inputs be set, outputs computed, and stepping function invoked. Indeed, different interpretations of the FMI Standard (version 2.0) lead to different master algorithms, which in turn lead to different constraints on the implementation of FMUs.

As we show next, these constraints arise out of the way the FMUs are connected, and how they are implemented.

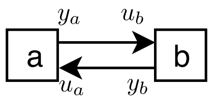

Figure 2: Running Example.

Definition 2. A scenario is a structure $\langle C, L \rangle$ where each identifier $c \in C$ is associated with an FMU, as defined in Definition 1, and $L(u) = y$ means that the output y is connected to input u. Let $U = \bigcup_{c \in C} U_c$ and $Y = \bigcup_{c \in C} Y_c$, then $L : U \to Y$. Figure 2 shows an example scenario.

Definition 3 (Co-simulation Step)**.** Given a scenario $\langle C, L \rangle$, a co-simulation step, or just step, is a finite ordered sequence of FMU function calls $(f_i)_{i \in \mathbb{N}} = f_0, f_1, \ldots$ with $f_i \in F = \bigcup_{c \in C} \{\mathtt{set}_c, \mathtt{get}_c, \mathtt{doStep}_c\}$, and i denoting the order of the function call.

Definition 4 (Initialization)**.** Given a scenario $\langle C, L \rangle$, we define the initialization procedure $(I_i)_{i \in \mathbb{N}}$ in the same way as a step, with $I_i \in F$.

Definition 5. Given a scenario $\langle C, L \rangle$, a step size H, a step $(f_i)_{i \in \mathbb{N}}$, and an initialization procedure $(I_i)_{i \in \mathbb{N}}$, a master algorithm is a structure defined as $\mathcal{A} = \langle C, L, H, (I_i)_{i \in \mathbb{N}}, (f_i)_{i \in \mathbb{N}} \rangle$.

The following are examples of master algorithms, for the scenario introduced in Figure 2. Algorithms 1 and 2 need no initialization. Their step is indicated in Lines 1–6 (inclusive). Algorithm 3 contains the initialization instructions in Lines 1–4, and the step in Lines 5–12.

All these algorithms respect the constraints imposed by the scenario, but they assume different implementations of the same FMUs. For instance, suppose that the output y_a depends instantaneously on the input u_a. Formally, $\exists v, v' \in \mathcal{V}$, such that $s_a^{(1)} = \mathtt{set}_a(s_a^{(0)}, u_a, v)$ and
$s_a^{(2)} = \mathtt{set}_a(s_a^{(0)}, u_a, v')$ and
$\mathtt{get}_a(s_a^{(1)}, y_a) \neq \mathtt{get}_a(s_a^{(2)}, y_a)$. In addition, suppose the output y_b does not depend instantaneously on u_b.

With these suppositions, Algorithm 1 is inadequate, because the value of y_a can only be computed after the value of u_a is known. The only value known of u_a is the one from the previous step.

Algorithm 1
```
1: $v \leftarrow \mathtt{get}_a(s_a^{(0)}, y_a)$
2: $s_b^{(1)} \leftarrow \mathtt{set}_b(s_b^{(0)}, u_b, v)$
3: $v \leftarrow \mathtt{get}_b(s_b^{(1)}, y_b)$
4: $s_a^{(1)} \leftarrow \mathtt{set}_a(s_a^{(0)}, u_a, v)$
5: $s_a^{(2)} \leftarrow \mathtt{doStep}_a(s_a^{(1)}, H)$
6: $s_b^{(2)} \leftarrow \mathtt{doStep}_b(s_b^{(1)}, H)$
7: $s_a^{(0)} \leftarrow s_a^{(2)}$
8: $s_b^{(0)} \leftarrow s_b^{(2)}$
9: Go to Line 1
```

Algorithm 2
```
1: $v \leftarrow \mathtt{get}_b(s_b^{(0)}, y_b)$
2: $s_a^{(1)} \leftarrow \mathtt{set}_a(s_a^{(0)}, u_a, v)$
3: $v \leftarrow \mathtt{get}_a(s_a^{(1)}, y_a)$
4: $s_b^{(1)} \leftarrow \mathtt{set}_b(s_b^{(0)}, u_b, v)$
5: $s_b^{(2)} \leftarrow \mathtt{doStep}_b(s_b^{(1)}, H)$
6: $s_a^{(2)} \leftarrow \mathtt{doStep}_a(s_a^{(1)}, H)$
7: $s_a^{(0)} \leftarrow s_a^{(2)}$
8: $s_b^{(0)} \leftarrow s_b^{(2)}$
9: Go to Line 1
```

In addition to the previous suppositions, assume that the input approximation scheme of FMU a is an interpolation. This requires that FMU b do a step before FMU a, so that when $\mathtt{doStep}_a$ is called, the most recent input known to FMU a has been computed from a FMU that is at time $t + H$. Now Algorithm 2 is incorrect, because, even though $\mathtt{doStep}_b$ is invoked before $\mathtt{doStep}_a$, there is no exchange of values after $\mathtt{doStep}_b$ is called, and before $\mathtt{doStep}_a$ is called. In contrast, Algorithm 3 satisfies all the previous suppositions.

Algorithm 3

```
1:  $v \leftarrow \texttt{get}_b(s_b^{(1)}, y_b)$
2:  $s_a^{(1)} \leftarrow \texttt{set}_a(s_a^{(0)}, u_a, v)$
3:  $v \leftarrow \texttt{get}_a(s_a^{(1)}, y_a)$
4:  $s_b^{(2)} \leftarrow \texttt{set}_b(s_b^{(1)}, u_b, v)$
5:  $s_b^{(1)} \leftarrow \texttt{doStep}_b(s_b^{(0)}, H)$
6:  $v \leftarrow \texttt{get}_b(s_b^{(1)}, y_b)$
7:  $s_a^{(1)} \leftarrow \texttt{set}_a(s_a^{(0)}, u_a, v)$
8:  $v \leftarrow \texttt{get}_a(s_a^{(1)}, y_a)$
9:  $s_b^{(2)} \leftarrow \texttt{set}_b(s_b^{(1)}, u_b, v)$
10: $s_a^{(2)} \leftarrow \texttt{doStep}_a(s_a^{(1)}, H)$
11: $s_a^{(0)} \leftarrow s_a^{(2)}$
12: $s_b^{(0)} \leftarrow s_b^{(2)}$
13: Go to Line 5
```

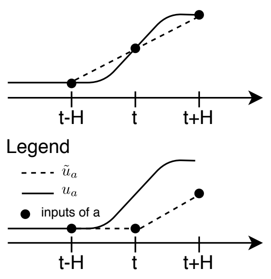

Figure 3: Comparison of Algorithms 2 and 3.

The main difference between Algorithm 2 and Algorithm 3 is a characteristic delay shift in the input signal of FMU a. An illustration is shown in Figure 3. In the top, FMU a guesses correctly at time t that the input provided is at time $t + H$. Therefore, the linear interpolation is correct. On the bottom figure however, the FMU thinks it is getting an input at time $t + H$, whereas it is just getting the input at time t. This causes a delay in the input approximation, which can increase the error, or lead to abrupt behavior changes, as is shown in [22].

For examples where these suppositions occur, refer to [23].

We now formalize the input/output instantaneous dependencies as *feed-through*, and the interpolation as *reactivity*. These form our notion of contracts. The reason we use the term reactivity is because it can be used for purposes other input approximation. For example, a software FMU may be reactive to reflect the fact that it contains a very short sampling interval (short relative to H).

Definition 6 (Feed-through)**.** The input $u_c \in U_c$ feeds through to output $y_c \in Y_c$, that is, $(u_c, y_c) \in D_c$, when there exists $v_1, v_2 \in \mathcal{V}$ and $s_c \in S_c$, such that

$$\texttt{get}_c(\texttt{set}_c(s_c, u_c, v_1), y_c) \neq \texttt{get}_c(\texttt{set}_c(s_c, u_c, v_2), y_c).$$

Definition 7 (Reactivity)**.** For a given FMU c with input $u_c \in U_c$, $R_c(u_u) = \mathit{true}$ if the function $\texttt{doStep}_c$ assumes that the input u_u comes from a FMU that has advanced forward relative to FMU c.

Our problem can now be formalized as:

Problem 1. *Given a scenario $\langle C, L \rangle$, and the set of contracts $\bigcup_{c \in C} \{(R_c, D_c)\}$, find a master algorithm*

$$\mathcal{A} = \langle C, L, H, (I_i)_{i \in \mathbb{N}}, (f_i)_{i \in \mathbb{N}} \rangle,$$

that satisfies those contracts.

IV. Semantics of Co-simulation Algorithms

The long term objective of this research is to provide a framework for generating master algorithms from a set of arbitrary unit contracts. As such, we choose to use Prolog. We assume the reader is familiar with the basic features of the language (see, e.g., [24] for a tutorial). In the following, capitalized identifiers denote variables, whereas un-capitalized identifiers denote facts.

A. Co-simulation Semantics

We first describe the representation of a scenario, using Figure 2 as running example. Then we describe the symbolic state of the co-simulation, and what the pre/post conditions of each operation on the state. Finally, we describe the meaning of a step.

Definition 8 (Scenario)**.** A scenario is given by a list of FMUs, and a list of connections. An fmu is defined by its identifier, a list of input ports, and a list of output ports (recall Definition 1). Each port has an identifier and a contract. The contract of an input port relates to its reactivity (recall Definition 7), and the contract of an output port is the list of input ports that it depends instantaneously on (recall Definition 6). For example, the following is the Prolog equivalent to Figure 2.

```
scenario_example_feedback([
  fmu(a,[port(u1,delayed)],[port(y1,[])]),
  fmu(b,[port(u2,reactive)],[port(y2,[u2])])],
 [connect(y2,u1),connect(y1,u2)]).
```

Definition 9 (Co-simulation State)**.** The state of a co-simulation is given by a list of the states of each fmu. The state of an fmu is the list of state of each of its ports, and the timestamp of its internal state. The state of each port comprises the timestamp of the port value, and whether it has been defined at that timestamp. The values for timestamp are `t` or `tH`, represent the current step, and the next, respectively. For example, the following shows three states: the state at the beginning of the co-simulation, where no value has been computed yet; the state at time `t`, where every value has been defined and has timestamp `t`; and the state after a step concludes, where every value has been defined and is at timestamp `tH`.

```
initial_state([
 fstate(a,t,
  [pstate(u1,undefined,t)],[pstate(y1,undefined,t)]),
 fstate(b,t,
  [pstate(u2,undefined,t)],[pstate(y2,undefined,t)])]).
state_at_t([
 fstate(a,t,
  [pstate(u1,defined,t)],[pstate(y1,defined,t)]),
 fstate(b,t,
  [pstate(u2,defined,t)],[pstate(y2,defined,t)])]).
state_at_tH([
 fstate(a,tH,
  [pstate(u1,defined,tH)],[pstate(y1,defined,tH)]),
 fstate(b,tH,
  [pstate(u2,defined,tH)],[pstate(y2,defined,tH)])]).
```

Definition 10 (Output Computation)**.** The `getOut(F, O)` represents the calculation of output `O` of fmu `F`. Given a co-simulation state `SB`, it checks whether all inputs that feed-through to `O` are defined and have the same timestamp `T`. If such is the case, then the `SA` is related to `SB` by replacing the state of `O` with `defined` and timestamp `T`:

```
executeOp(getOut(F,O),FMUs,_,SB,SA):-
member(fmu(F,_,Outports),FMUs),member(port(O,Deps),Outports),
member(fstate(F,T,InSB,OutSB),SB),member(pstate(O,_,T),OutSB),
namesInState(Deps,InSB,defined,T),setDefine(O,defined,SB,SA).
```

Example 1. The following shows an example use of the getOut(F, O) predicate.

```
FMUs=[fmu(a,[port(u1, delayed)],[port(y1,[u1])])],
SB=[fstate(a,t,
    [pstate(u1,defined,t)],[pstate(y1,undefined,t)])],
SA=[fstate(a,t,
    [pstate(u1,defined,t)],[pstate(y1,defined,t)])],
executeOp(getOut(a,y1),FMUs,_,SB,SA).
```

Definition 11 (Input Computation)**.** The setIn(F, I) represents the setting of input I of fmu F. Given a co-simulation state SB, it checks whether all outputs connected to I are defined and have the same timestamp T. If that is the case, then the state after the computation SA is related to SB by replacing the state of I with defined and timestamp T:

```
executeOp(setIn(F,I),FMUs,Conns,SB,SA):-
 member(fmu(F,Inports,_),FMUs),member(port(I,_),Inports),
 member(fstate(F,_,InSB,_),SB),member(pstate(I,_,_),InSB),
 member(connect(_,I),Conns),
 connectionsState(I,Conns,defined,T,SB),
 setDefine(I,defined,SB,SAUX),setTimestamp(I,T,SAUX,SA).
```

Definition 12 (Step Computation)**.** The doStep(F) represents the advancement in time of an fmu F whose timestamp is t. The success of this function depends on the state of the co-simulation, SB, and the contracts on its input ports. If F contains an input port that is delayed, then the state of that port must be defined at timestamp t. If, on the other hand, F contains an input port that is reactive, then the state of that port must be defined at timestamp t. If these conditions hold for every input port of the FMU, then the timestamp of F and its output ports becomes tH, and the outputs become undefined:

```
executeOp(doStep(F),FMUs,_,SB,SA):-
 member(fmu(F,Inps,Outps),FMUs),
 member(fstate(F,t,InSB,OutSB),SB),
 checkInputContract(Inps,InSB),
 portsInSameTimeStamp(Outps,OutSB,t),
 setTimestamp(F,tH,SB,SX),
 setPortsDefine(Outps,undefined,SX,SX2),
 setPortsTimestamp(Outps,tH,SX2,SA).
```

Example 2. The following shows an example application of the doStep(F) predicate.

```
FMUs=[fmu(a,[port(u1, reactive)],[port(y1,[u1])])],
SB=[fstate(a,t,[pstate(u1,defined,tH)],
    [pstate(y1,defined,t)])],
SA=[fstate(a,tH,[pstate(u1,defined,tH)],
    [pstate(y1,undefined,tH)])],
executeOp(doStep(a),FMUs,_,SB,SA).
```

Definition 13 (Schedule)**.** The execution of a sequence of operation, is defined inductively by:

```
executeOps([],_,_,State,State).
executeOps([Op|NOps],FMUs,Conns,SB,SA):-
 executeOp(Op,FMUs,Conns,SB,SAUX),
 executeOps(NOps,FMUs,Conns,SAUX,SA).
```

With these definitions, the objective of the initialization schedule is to define all values at time t. The objective of the step is to take a state that has all values defined at time t, and define them at time tH. This way, the state at the end of the initialization procedure becomes the state at the beginning of a step. Moreover, the state at the end of a step becomes the state at the beginning of the next co-simulation state.

Definition 14 (Master Algorithm)**.** A master algorithm is defined as follows.

```
isMasterAlgorithm(Init,Step,FMUs,Connections):-
 inStateT0(FMUs,ST0),inStateT(FMUs,ST),inStateTH(FMUs,STH),
 executeOps(Init,FMUs,Connections,ST0,ST),
 executeOps(Step,FMUs,Connections,ST,STH).
```

Predicates StateT0, StateT, and StateTH, define the state according to the FMUs declared.

B. Master Algorithm Generation

The specification we propose in Section IV-A cannot be used as is with Prolog's unification algorithm to generate master algorithms. However, with minor modifications, one obtains an exponential procedure to generate such algorithms.

Definition 15 (Optimized Master Generation)**.** The optimized master generation is defined as in Definition 14, except it uses the following schedule definition:

```
executeOpsM(_,[],_,_,State,State).
executeOpsM(POps,[Op|NOps],FMUs,Conns,SB,SA):-
 executeOp(Op,FMUs,Conns,SB,SAUX),SAUX\==SB,\+member(Op,POps),
 executeOpsM([Op|POps],NOps,FMUs,Conns,SAUX,SA).
```

Compared to Definition 14, the main difference is in enforcing that operations must change the state, and cannot be repeated.

If there are no algebraic loops in the scenario, then the application of Definition 15 to unbound Init and Step variables will always terminate. To see why, note that any operation that is successfully executed in will have to change the state, and there is only a finite number of operations to try. Every operation that changes the state, moves it to being closer to the target state, and there is no operation that can rollback the state.

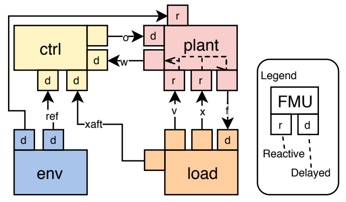

Figure 4: Case study scenario. The reactivity and feed-through of each port is indicated in the figure.

Example 3 (Based on [12])**.** Consider the scenario in Figure 4. The generated step schedule, which took about 9 minutes to produce in a 3.5GHz laptop, is:

1. doStep(load),
2. getOut(load, l_x),
3. getOut(load, l_v),
4. getOut(load, l_xaft),
5. setIn(plant, p_x),
6. setIn(plant, p_v),
7. doStep(env),
8. getOut(env, e_psu),
9. getOut(env, e_ref),
10. setIn(plant, p_psu),
11. doStep(plant),
12. getOut(plant, p_w),
13. getOut(plant, p_f),
14. setIn(ctrl, c_w),
15. setIn(load, l_f),
16. doStep(ctrl),
17. getOut(ctrl, c_o),
18. setIn(ctrl, c_ref),
19. setIn(ctrl, c_xaft),
20. setIn(plant, p_o).

V. Conclusion

Inspired by prior work in generation of synchronization protocols between robotic parts, we formalized co-simulation in Prolog under a restricted set of FMU contracts.

The choice of Prolog is motivated by the flexibility in expressing structural operation semantics. The theory presented here, however, can be adapted to other languages. It is our hope that such flexibility allows more contracts to be supported. For example, when incorporating step size adaptation contracts (such as the ability to reject H and suggest a smaller one), the symbols `t,tH` will not be enough. Moreover, this is a first step to incorporating more complex contracts, such as those relating to the models that lye behind the FMUs.

Existing works focusing on the correct synchronization of hybrid co-simulations, mentioned in Section II-B, complement our own. Moreover, the work in [25] formalizes the semantics of FMI co-simulation, and can potentially be extended with simulator contracts. However, its objective is to prove properties about the system being co-simulated, whereas our goal is to guarantee certain basic properties of the co-simulation. Ongoing work is revising the contracts and semantics in order to accommodate the rollback operation, which is also formalized in [25].

The code to reproduce the experiments in this paper is available for download[2].

References

[1] T. Blochwitz, M. Otter, M. Arnold, C. Bausch, C. Clauss, H. Elmqvist, A. Junghanns, J. Mauss, M. Monteiro, T. Neidhold, D. Neumerkel, H. Olsson, J.-V. Peetz, and S. Wolf, “The Functional Mockup Interface for Tool independent Exchange of Simulation Models,” in *8th International Modelica Conference*. Dresden, Germany: Linköping University Electronic Press; Linköpings universitet, Jun. 2011, pp. 105–114.

[2] H. Vangheluwe, J. De Lara, and P. J. Mosterman, “An introduction to multi-paradigm modelling and simulation,” in *AI, Simulation and Planning in High Autonomy Systems*. SCS, 2002, pp. 9–20.

[3] S. Mustafiz, C. Gomes, B. Barroca, and H. Vangheluwe, “Modular Design of Hybrid Languages by Explicit Modeling of Semantic Adaptation,” in *Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*. Pasadena, California: IEEE, Apr. 2016, pp. 29:1–29:8, series Title: DEVS ’16.

[4] N. Pedersen, K. Lausdahl, E. Vidal Sanchez, P. G. Larsen, and J. Madsen, “Distributed Co-Simulation of Embedded Control Software with Exhaust Gas Recirculation Water Handling System using INTO-CPS,” in *7th International Conference on Simulation and Modeling Methodologies, Technologies and Applications*. SCITEPRESS - Science and Technology Publications, 2017, pp. 73–82.

[5] R. Kübler and W. Schiehlen, “Two Methods of Simulator Coupling,” *Mathematical and Computer Modelling of Dynamical Systems*, vol. 6, no. 2, pp. 93–113, Jun. 2000.

[6] FMI, “Functional Mock-up Interface for Model Exchange and Co-Simulation,” FMI development group, Tech. Rep., 2014.

[7] T. Blockwitz, M. Otter, J. Akesson, 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 *9th International Modelica Conference*. Munich, Germany: Linköping University Electronic Press, Nov. 2012, pp. 173–184.

[8] C. Gomes, C. Thule, D. Broman, P. G. Larsen, and H. Vangheluwe, “Co-simulation: State of the art,” University of Antwerp, Tech. Rep., Feb. 2017. [Online]. Available: http://arxiv.org/abs/1702.00686

[9] G. Schweiger, C. Gomes, G. Engel, I. Hafner, J. Schoeggl, A. Posch, and T. Nouidui, “Functional Mock-up Interface: An empirical survey identifies research challenges and current barriers,” in *The American Modelica Conference*. Cambridge, MA, USA: Linköping University Electronic Press, Linköpings universitet, 2018, pp. 138–146.

[10] C. Gomes, C. Thule, D. Broman, P. G. Larsen, and H. Vangheluwe, “Co-simulation: A Survey,” *ACM Computing Surveys*, vol. 51, no. 3, p. Article 49, Apr. 2018.

[11] G. Schweiger, C. Gomes, G. Engel, I. Hafner, J.-P. Schoeggl, A. Posch, and T. Nouidui, “An empirical survey on co-simulation: Promising standards, challenges and research needs,” *Simulation Modelling Practice and Theory*, vol. 95, pp. 148–163, Sep. 2019.

[12] C. Gomes, B. J. Oakes, M. Moradi, A. T. Gamiz, J. C. Mendo, S. Dutre, J. Denil, and H. Vangheluwe, “HintCO - Hint-Based Configuration of Co-Simulations,” in *International Conference on Simulation and Modeling Methodologies, Technologies and Applications*, Prague, Czech Republic, 2019, pp. 57–68.

[13] C. Gomes, B. Meyers, J. Denil, C. Thule, K. Lausdahl, H. Vangheluwe, and P. De Meulenaere, “Semantic Adaptation for FMI Co-simulation with Hierarchical Simulators,” *SIMULATION*, vol. 95, no. 3, pp. 1–29, 2018.

[14] M. Arnold, C. Clauß, and T. Schierz, “Error Analysis and Error Estimates for Co-simulation in FMI for Model Exchange and Co-Simulation v2.0,” in *Progress in Differential-Algebraic Equations*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 107–125.

[15] P. J. Mosterman, “Hybrid dynamic systems: Modeling and execution,” in *Handbook of Dynamic System Modeling*. Chapter, 2007, vol. 15.

[16] C. Gomes, P. Karalis, E. M. Navarro-López, and H. Vangheluwe, “Approximated Stability Analysis of Bi-modal Hybrid Co-simulation Scenarios,” in *1st Workshop on Formal Co-Simulation of Cyber-Physical Systems*. Trento, Italy: Springer, Cham, 2017, pp. 345–360.

[17] L. G. Iugan, H. Boucheneb, and G. Nicolescu, “A generic conceptual framework based on formal representation for the design of continuous/discrete co-simulation tools,” *Design Automation for Embedded Systems*, vol. 19, no. 3, pp. 243–275, Sep. 2015.

[18] L. Gheorghe, F. Bouchhima, G. Nicolescu, and H. Boucheneb, “Semantics for Model-based Validation of Continuous/Discrete Systems,” in *Design, Automation and Test in Europe*. New York, NY, USA: ACM, 2008, pp. 498–503, series Title: DATE ’08.

[19] L. Gheorghe, G. Nicolescu, and H. Boucheneb, “Semantics for Rollback-Based Continuous/Discrete Simulation,” in *Behavioral Modeling and Simulation Workshop, 2008. BMAS 2008. IEEE International*, 2008, pp. 106–111.

[20] L. Gheorghe, F. Bouchhima, G. Nicolescu, and H. Boucheneb, “Formal Definitions of Simulation Interfaces in a Continuous/Discrete Co-Simulation Tool,” in *Rapid System Prototyping, 2006. Seventeenth IEEE International Workshop On*, 2006, pp. 186–192.

[21] ——, “A Formalization of Global Simulation Models for Continuous/Discrete Systems,” in *Summer Computer Simulation Conference*. San Diego, CA, USA: Society for Computer Simulation International San Diego, CA, USA, Jul. 2007, pp. 559–566, series Title: SCSC ’07.

[22] C. Thule, C. Gomes, J. Deantoni, P. G. Larsen, J. Brauer, and H. Vangheluwe, “Towards Verification of Hybrid Co-simulation Algorithms,” in *Workshop on Formal Co-Simulation of Cyber-Physical Systems*. Toulouse, France: Springer, Cham, 2018.

[23] C. Gomes, C. Thule, P. G. Larsen, J. Denil, and H. Vangheluwe, “Co-simulation of Continuous Systems: A Tutorial,” University of Antwerp, Tech. Rep. arXiv:1809.08463 [cs, math], Sep. 2018. [Online]. Available: http://arxiv.org/abs/1809.08463

[24] M. Bramer, *Logic Programming with Prolog*. Springer, 2005, vol. 9.

[25] F. Zeyda, J. Ouy, S. Foster, and A. Cavalcanti, “Formalising Cosimulation Models,” in *Software Engineering and Formal Methods*, A. Cerone and M. Roveri, Eds., vol. 10729. Cham: Springer International Publishing, 2018, pp. 453–468.

[2]http://msdl.cs.mcgill.ca/people/claudio/projs/PrologCosimSemantics.zip