Journal Title
XX(X):1–28
©The Author(s) 2017
Reprints and permission:
sagepub.co.uk/journalsPermissions.nav
DOI: 10.1177/ToBeAssigned
www.sagepub.com/
SAGE

Semantic Adaptation for FMI Co-simulation with Hierarchical Simulators

Cláudio Gomes[1], Bart Meyers[12], Joachim Denil[12], Casper Thule[3], Kenneth Lausdahl[3], Hans Vangheluwe[124] and Paul De Meulenaere[12]

Abstract

Model-based design can shorten the development time of complex systems by the use of simulation techniques. However, it can be hard to simulate the system as a whole if it is developed in a concurrent fashion, by multiple and specialized teams. Co-simulation, with the support of the Functional Mock-up Interface (FMI) Standard, is proposed as a way to promote tool interoperability while protecting intellectual property of subsystems. The standard allows a uniform communication between subsystem simulators, but does not state how the inputs and outputs should be interpreted, nor how the subsystems should interact correctly.

Semantic adaptations can be quickly made to correct the interactions with sub-system simulators that were produced with different assumptions, and avoid changing those sub-systems, their simulators, or the orchestration algorithm that computes the co-simulation. In this work, we explore how to describe common adaptations and what their meaning is in the context of FMI co-simulation.

The result is a sound language that enables the implementation of adaptations with minimal effort. A distinct feature is that it describes adaptations for groups of interconnected subsystem simulators in the same way as for a single simulator, and the implementation is itself a simulator, thanks to a sound definition of hierarchical co-simulation.

This work paves the way for research into the correct combination and interfacing of different adaptations.

Keywords

Co-simulation, Semantic adaptation, Functional Mockup Interface Standard, Hierarchical co-simulation

Introduction

The systems we engineer today are characterised by an increasing complexity. Model-based design can boost the development of such systems by enabling their analysis at higher levels of abstraction via simulation. However, it can be hard to simulate the system as a whole if it is developed in a distributed fashion, by multiple and specialized teams[1].

Two factors contribute to this difficulty: (i) specialized teams have their own tools; and (ii) some of the components of the system are provided by different suppliers[2], and have valuable Intellectual Property (IP).

Difficulty (i) is a natural consequence of using the most appropriate formalism for a specific domain[3]. The same can be expected from external suppliers. During the development process, if a team wishes to understand how a component being developed behaves when interacting with the rest of the system, it is useful that the tool in use can import and *simulate correctly* the models created by the other teams (with different tools). As we show shortly, it can be hard to correctly simulate imported models, as these belong to potentially different domains, each with its own set of specialized simulators[3].

As for difficulty (ii), if the team is using externally supplied components and wishes to simulate them, it may not be able to import the components' models because these contain protected IP. For the sufficiently complex components, a "lock-in" contract can be made to allow the team access to those models. However, the team will no longer be free to benchmark components from different competing suppliers.

Co-simulation, with the support of the Functional Mock-up Interface (FMI) Standard[4], is proposed as a way to promote tool interoperability while addressing the IP protection requirement. The models are exported as executable black boxes, that receive inputs and produce outputs, allowing for the

[1]University of Antwerp
[2]Flanders Make
[3]Aarhus University
[4]McGill University

Corresponding author:
Cláudio Gomes, University of Antwerp, Middelheimlaan 1
Middelheimcampus Building G, Room G.028, Antwerp, Belgium 2020
+32 488040130
Email: claudio.gomes@uantwerp.be

simulation of the component they stand for. In the FMI, each black box is called a Functional Mock-up Unit (FMU), the term adopted in this document.

The standard provides a common interface to allow a uniform communication with the black boxes, solving the combinatorial explosion of import/export formats. However, it does not ensure that the black boxes are interacted with in a semantically correct manner.

When a team is given an FMU that does not behave as it is expected to, we say that there is an interaction mismatch between the FMU and the rest of the system. Interaction mismatches can be roughly classified as:

Signal Data Mismatch happens when the signals provided by the FMU are not compatible with the ones that are expected (e.g., different frame of reference or different physical units).

Model of Computation Mismatch happens when the provided FMU assumes a different model of computation[5] than the one actually used to compute the overall behavior of the system (e.g., FMUs exported by a timed automata modelling and simulation tool[6,7] have to make assumptions about the other interacting FMUs).

Capability Mismatch happens when a given FMU lacks some capabilities [*] that affect the simulation performance (e.g., FMUs that lack higher order input extrapolation, an important capability that affects the accuracy and stability of the co-simulation[9,10]).

Rather than asking the original producer of the FMU to correct an interaction mismatch, it can be useful that the team is able to correct it immediately. Note that any mismatch happens between a given FMU and a usage intent, and therefore it is not necessarily the case that the best correction of a mismatch is done by the producer (if the FMU is to be reused).

In fact, as[9,11–13] show, some mismatches happen as a product of the (incorrect) handling of multiple interacting FMU's, and the correction has to be done for that specific interaction.

The above arguments motivate the need for semantic adaptations, and lead to the following research question:

RQ1 How can we *describe* the most common semantic adaptations on multiple types of black box FMUs in a *productive* manner, and realise them without violating *modularity* and *transparency*.

Informally, we call semantic adaptation of an FMU to the set of modifications made to the inputs/outputs and interaction with environment, of the FMU, with the purpose of correcting an interaction mismatch. This concept is formalized later in this work.

Productivity is related to the effort required to describe an adaptation. *Modularity* refers to the fact that any FMU should be adapted by changing how it is interacted with, and not how it is implemented. *Transparency* means that any tool that imports FMUs should not have to be changed in order to import, and interact with, an adapted FMU.

The descriptions should be made in an independently developed language because it is impractical that every tool capable of importing FMUs is able to implement the adaptations. Furthermore, one cannot expect that any user of an FMU has the ability to modify the importing tool to support these. Compared to implementing these adaptations manually, a language reduces the accidental complexity, prevents mistakes, and allows soundness analyses to be carried out.

In this paper, we build on prior work[14–16] to define a language that allows for the descriptions of the most common semantic adaptations that can be used in FMI co-simulation, surveyed in[17]. A distinct feature of the language proposed here is that it describes adaptations for groups of interconnected FMUs in the same way as for a single FMU, thanks to a sound definition of hierarchical co-simulation.

The definition of hierarchical co-simulation, and the semantics of the language, are presented in a bottom up approach, as illustrated in Figure 1. In the Background section, we introduce a co-simulation abstraction with simulation units and how these relate to FMUs. Section "Hierarchical Co-simulation for Semantic Adaptation" contains the formal foundations of a special kind of simulation unit that is the template to implement any semantic adaptation. In Section "Running Example", a running example is described, and in Section "A DSL for Semantic Adaptation" the language and its semantics are described. Section "Evaluation" judges how well we have addressed the research question. Section "Discussion and Future Work" discusses the flaws of our approach and research opportunities. Finally, Sections "Related Work" and "Conclusion" present the related work and conclude, respectively.

Background

In this section, we introduce the concepts, terminology, and assumptions used throughout this document. We cover co-simulation, the Functional Mockup Interface (FMI) standard, semantic adaptation, and domain specific languages.

Co-Simulation

We briefly summarize the main concepts related to co-simulation and we refer the reader to[17] for a more detailed introduction of each concept.

We call *dynamical system* to a model that has a notion of state and rules describing the evolution of

[*]See[8] for an overview of capabilities of FMUs.

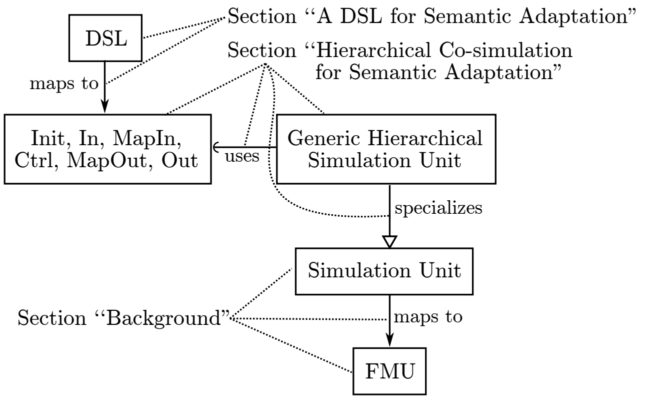

DSL
Section "A DSL for Semantic Adaptation"
Section "Hierarchical Co-simulation for Semantic Adaptation"
maps to
Init, In, MapIn, Ctrl, MapOut, Out
uses
Generic Hierarchical Simulation Unit
specializes
Simulation Unit
Section "Background"
maps to
FMU

Figure 1. Overview of DSL semantics and document structure.

that state across time, starting from an initial state. Inputs and outputs can be defined, to describe the environment.

A *simulator* is an algorithm that takes a dynamical system and input signals, as input, and computes an approximated behavior trace of the dynamical system.

A *simulation unit* (also known as a simulation application[18]) is the composition of a simulator together with a dynamical system, essentially representing a mockup of a real system. It accepts input trajectories and produces a behavior trace.

Simulation units (or just units) can be coupled through their inputs and outputs. A *coupling restriction* (or just couplings) is an output connected to an input. It means that the trajectory computed at that output — e.g., a function of the continuous time — must be equal to the one computed at the input, *at all times*.

The *orchestrator* (or master) is an algorithm that takes a set of simulation units and their coupling restrictions — that is, a *co-simulation scenario* — and computes the behavior trace of all units, trying to satisfy the coupling restrictions. In practice, these restrictions can only be satisfied at certain countable points in time, called *communication points*. These points are agreed upon by the simulation units and the orchestrator.

A basic orchestrator will, at each communication point, copy data points from outputs to inputs, and ask each unit to compute its own behavior trace until the next communication point. The collective behavior trace is called the *co-simulation*.

We capture the essence of a *simulation unit* with reference i, using the discrete time system notation, in one of the following *four* possible ways:

$$\begin{aligned}\left\langle \boldsymbol{x_i}(t+\tilde{H}_i), \tilde{H}_i \right\rangle &= F_i(t, H, \boldsymbol{x_i}(t), \underbrace{\boldsymbol{u_i}(t+H)}_{\text{reactive}} \text{ or } \underbrace{\boldsymbol{u_i}(t)}_{\text{delayed}}) \\ \boldsymbol{y_i}(t) &= \underbrace{G_i(t, \boldsymbol{x_i}(t), \boldsymbol{u_i}(t))}_{\text{Mealy}} \text{ or } \underbrace{G_i(t, \boldsymbol{x_i}(t))}_{\text{Moore}} \\ \boldsymbol{x_i}(0) &= \underbrace{Init_i(\boldsymbol{u_i}(0))}_{\text{reactive}} \text{ or } \underbrace{Init_i()}_{\text{delayed}}\end{aligned} \quad (1)$$

where t denotes the simulated time, $\boldsymbol{x_i}$ denotes the state vector, $\boldsymbol{u_i}$ the input vector, $Init_i$ computes the initial state, $H > 0$ denotes the requested communication step size, $0 < \tilde{H}_i \leq H$ denotes the communication step size taken by the unit, F_i is the state transition function, and G_i the output function. Bold symbols will always refer to vectors in this paper.

The definition in Equation (1) covers the different kinds of simulation units considered (based on the orchestrators surveyed in[17]): Reactive Mealy, Reactive Moore, Delayed Mealy, and Delayed Moore. The difference is in *where* and *when* the unit expects inputs to be provided. For example, a delayed Moore unit can compute its output without requiring an input, and can compute its future state ($\boldsymbol{x_i}(t+\tilde{H}_i)$) with just the current input $\boldsymbol{u_i}(t)$. A reactive Mealy unit, on the other hand: requires an initial input to compute the initial state; and needs to know the next input in order to compute the next state/output.

We use shortcuts such as $F_i(t, H, \boldsymbol{x_i}(t), \ldots)$, $G_i(t, \ldots)$, and $Init_i(\ldots)$, to denote the appropriate function depending on the kind of unit i. Furthermore, we make the following remarks about each simulation unit i:

- F_i and G_i are mathematical functions (also denoted pure).
- The internal definition of F_i and G_i is unknown, but the kind of unit is known.
- If $\left\langle \cdot, \tilde{H}_i \right\rangle = F_i(t, H, \boldsymbol{x_i}(t), \ldots)$ and $\tilde{H}_i < H$, then the unit rejects the step size H requested. Furthermore, for any $\tilde{H} \leq \tilde{H}_i$, we assume that $\left\langle \cdot, \tilde{H} \right\rangle = F_i(t, \tilde{H}, \boldsymbol{x_i}(t), \ldots)$.

Given a set of unique unit references $D = \{1, \ldots, n\}$, a *co-simulation scenario* is defined as the aggregation of each simulation unit definition, in Equation (1), plus a coupling function that defines the input of i as a function of the outputs of units $\{j : j \in D \setminus \{i\}\}$. Formally, combining the notation used in[9,19], a scenario is given by:

$$\begin{cases} \left\langle \boldsymbol{x_i}(t+\tilde{H}_i), \tilde{H}_i \right\rangle = F_i(t, H, \ldots) \\ \boldsymbol{y_i}(t) = G_i(t, \ldots) \\ \boldsymbol{u_i} = c_i(\boldsymbol{y_1}, \ldots, \boldsymbol{y_{i-1}}, \boldsymbol{y_{i+1}}, \ldots, \boldsymbol{y_n}) \\ \boldsymbol{x_i}(0) = Init_i(\ldots) \end{cases} \quad (2)$$
$$\text{for each} \quad i \in D$$

where c_i denotes the coupling function, and each F_i, G_i follows one of the definitions in Equation (1). Commonly, c_i is linear and maps at most one component of one of the inputs (the inputs/outputs are vector quantities), onto one component of the output. We assume that c_i is linear.

Let $i, j \in D$ be two different units, and $\bar{0}$ be the zero matrix of appropriate dimension. If $\frac{\partial c_i}{\partial \boldsymbol{y_i}} \neq \bar{0}$, then i *gets part of its input from* j. Informally, this means that at least one component of $\boldsymbol{u_i} = c_i(\ldots)$ is determined by at least one component of $\boldsymbol{y_i}$. We say that a unit $i \in D$ depends algebraically on unit $j \in D$, with $i \neq j$, if i gets part of its input from j and i is *not* a *delayed Moore*. So, e.g., if i gets part of its input from j, but it is a *delayed Moore*, then i does not depend algebraically on j.

Using the algebraic dependency relationship, one can build a directed graph — called the dataflow graph — with one node n_i per simulation unit $i \in D$, and an edge (n_j, n_i) between two nodes n_j, n_i whenever the unit i depends algebraically on unit j. This procedure is based on the Causal Block Diagram Simulation algorithm[20,21]. A topological order of the resulting graph gives an execution order that respects the units' algebraic dependencies.

Depending on the coupling function and on the kind of simulation units being coupled, *algebraic loops* may occur. An algebraic loop includes any input/output/state that depends on itself, at the same time point[19].

If an algebraic loop exists between the units, then it is not possible to compute a topological ordering of the dataflow graph. For now, we assume that such topological order can always be computed for a given co-simulation scenario. We denote that order via a mapping $\sigma : \mathbb{N} \to D$, that returns the unit reference $\sigma(j)$ that is the j-th in the topological order. So $\sigma(1)$ gives a unit that is first in the topological order, i.e., has no algebraic dependencies.

With a well defined topological order, the orchestrator only has to provide inputs to, execute, and get outputs from, the units in that order. Algorithm 1 formalizes what is known in the state of the art as the Gauss-Seidel orchestrator. It computes the behavior trace of a given co-simulation scenario as described in Equation (2). To be concise, we abbreviate the output and state transition function calls, which depend on the kind of unit (lines 11, 19, and 21). Furthermore, the orchestrator provides the inputs ($\boldsymbol{uc}_{\sigma(j)}$ or $\boldsymbol{up}_{\sigma(j)}$, in line 19) that each unit expects, working for both reactive and delayed units alike. This is the main reason we single out this orchestrator in this work.

Without loss of generality, we assume the most basic step size control policy in Algorithm 1: the communication step size is never increased after being rejected by some unit*. The orchestrator uses the most recent consistent state.

Algorithm 1: Gauss-seidel orchestrator for co-simulation scenarios can be topologically sorted.

```
   Data: The stop time $T$, a starting communication step
         size $\hat{H}$, and a set of unit references
         $D = \{1, \ldots, n\}$.
1  $t := 0$ ;                                  // Simulation time
2  $H := \hat{H}$ ;                            // Communication step size
   // Initialize variables
3  for $i = 1, \ldots, n$ do
4      $\boldsymbol{x_i} := \boldsymbol{0}$ ;                          // State vector
5      $\boldsymbol{uc_i} := \boldsymbol{y_i} := \boldsymbol{0}$ ;      // Current I/O variables
6      $\boldsymbol{up_i} := \boldsymbol{0}$ ;                         // Previous input variables
7  end
   // Compute initial states
8  for $j = 1, \ldots, n$ do
9      $\boldsymbol{uc}_{\sigma(j)} := c_{\sigma(j)}(\boldsymbol{y_1}, \ldots, \boldsymbol{y}_{\sigma(j)-1}, \boldsymbol{y}_{\sigma(j)+1}, \ldots, \boldsymbol{y_n})$ ;
10     $\boldsymbol{x}_{\sigma(j)} := Init_{\sigma(j)}(\boldsymbol{uc}_{\sigma(j)})$ or $Init_{\sigma(j)}()$
11     $\boldsymbol{y}_{\sigma(j)} := G_{\sigma(j)}(t, \boldsymbol{x}_{\sigma(j)}, \boldsymbol{uc}_{\sigma(j)})$ or $G_{\sigma(j)}(t, \boldsymbol{x}_{\sigma(j)})$;
12     $\boldsymbol{up}_{\sigma(j)} := \boldsymbol{uc}_{\sigma(j)}$ ;
13 end
14 while $t < T$ do
15     $accepted := false$ ;
16     while $not\ accepted$ do
17         for $j \in (1, \ldots, n)$ do
18             $\boldsymbol{uc}_{\sigma(j)} :=$
                   $c_{\sigma(j)}(\boldsymbol{y_1}, \ldots, \boldsymbol{y}_{\sigma(j)-1}, \boldsymbol{y}_{\sigma(j)+1}, \ldots, \boldsymbol{y_n})$ ;
19             $\left\langle \tilde{\boldsymbol{x}}_{\sigma(j)}, \tilde{H}_{\sigma(j)} \right\rangle :=$
                   $F_{\sigma(j)}(t, H, \boldsymbol{x}_{\sigma(j)}, \boldsymbol{uc}_{\sigma(j)}$ or $\boldsymbol{up}_{\sigma(j)})$ ;
20             $\boldsymbol{y}_{\sigma(j)} := G_{\sigma(j)}(t + \tilde{H}_{\sigma(j)}, \boldsymbol{x}_{\sigma(j)}, \boldsymbol{uc}_{\sigma(j)})$
21                 or $G_{\sigma(j)}(t + \tilde{H}_{\sigma(j)}, \boldsymbol{x}_{\sigma(j)})$;
22         end
23         $\tilde{H} := \min_{i \in D}(\tilde{H}_i)$ ;
24         if $\tilde{H} < H$ then
25             $H := \tilde{H}$ ;
26         else
27             $accepted := true$ ;
28         end
29     end
       // Commit state and update I/O
30     for $j = 1, \ldots, n$ do
31         $\boldsymbol{x_i} := \tilde{\boldsymbol{x}}_{\boldsymbol{i}}$ ;
32         $\boldsymbol{up_i} := \boldsymbol{uc_i}$ ;
33     end
34     $t := t + H$ ;                              // Advance time
35 end
```

Functional Mock-up Interface Standard (FMI)

The FMI standard[4] defines the interface and interaction pattern that allows simulation units to communicate. In the standard, a simulation unit is called a Functional Mockup Unit (FMU).

FMUs and Simulation units. This subsection establishes the equivalence between FMUs and simulation units (recall Figure 1), and the assumptions we make throughout this document.

Given a simulation unit i (described in Equation (1)) we define its equivalent FMU, and vice versa, as follows:

FMU State – The state of the FMU corresponds to the state of the unit x_i. The FMU does not make the state explicit, but instead implements functions that can be used to set and retrieve the state.

Inputs – FMUs have input ports, each accepting a scalar quantity. Each dimension in the input $\boldsymbol{u_i}$ corresponds to one input port of the FMU.

*Algorithm 1 can be greatly optimized (e.g., rolling back as soon as a reject occurs).

The FMU implements functions that allow the orchestrator to set those inputs (e.g., `fmi2SetReal` and `fmi2SetInteger`) and a single vector quantity $\boldsymbol{u_i}$ can be set via multiple calls to those functions.

Outputs – The outputs of the FMU are analogous to the inputs. To obtain an output $\boldsymbol{y_i}$, multiple calls are made to the dedicated functions (e.g., `fmi2GetReal` and `fmi2GetInteger`).

Initial State – The initial state computed by the $Init_i$ function corresponds to the computation performed by the FMU in the initialization mode. We assume that an initial state of a unit/FMU can always be found from the $Init(\ldots)$ function (and initial input, in case of a reactive unit). This is in accordance with the FMI Standard, but it *restricts our scope* to scenarios were the consistent initial state of one unit depends on factors (e.g., the initial state of another unit) other than its initial inputs.

Co-simulation Step – A state transition invocation $\left\langle \tilde{\boldsymbol{x}}_{\boldsymbol{i}}, \tilde{H}_i \right\rangle := F_i(t, H, \boldsymbol{x_i}, \boldsymbol{u_i})$ is mapped to (in order): an optional invocation to set the state of the FMU to $\boldsymbol{x_i}$; multiple invocations to set the input $\boldsymbol{u_i}$; an invocation to the `fmi2DoStep` function; a query to find out up to which time the FMU computed the step (to get $\tilde{H}_i$); and an (optional) invocation to get the new state of the FMU $\tilde{\boldsymbol{x}}_{\boldsymbol{i}}$. The manipulation of the state is optional for orchestration algorithms that do not perform rollback operations. However, in this document, we assume that *the FMUs support rollback*.

Output Function – If the unit is a Mealy unit, then the execution of the output function $\boldsymbol{y_i} := G_i(t, \boldsymbol{x_i}(t), \boldsymbol{u_i}(t))$ corresponds to setting the inputs to the FMU, and then getting the outputs. If the unit is a Moore unit, then the outputs can be enquired without first setting the inputs.

It is the role of the orchestrator to set the appropriate inputs depending on whether the FMU is reactive or delayed, or mealy and Moore.

We define the type of the FMU by applying the following rules, in order:

1. If the unit does not disclose any input-to-output feedthrough, it is assumed to be *Mealy*.
2. If at least one output variable depends instantaneously on an input variable, we assume that the unit is *Mealy*.
3. If the previous two do not apply, the unit is assumed to be *Moore*.
4. If the capability flag *canInterpolateInputs* is set, then the unit is *reactive*.
5. Otherwise, the unit is *delayed*.

To establish the equivalence of the couplings restrictions of units and those of FMUs, we note that the definition of algebraic dependency remains the same between FMUs. Thus, the dataflow graph can be built as described in the previous subsection.

Having established the equivalence between simulation units and FMUs, we will henceforth use the two terms interchangeably.

Semantic Adaptation

The interface of an FMU (or of a simulation unit) comprises not only the specification of the inputs and outputs, but also how it is to be interacted with[14]. It may be the case that in different co-simulation scenarios, the same FMU has to be interacted with differently (e.g., for accuracy/performance concerns). While modifying the orchestrator to support a new interaction pattern will solve the problem, it is not ideal since: (i) the interaction pattern may be specific to a single FMU (therefore not reusable), and (ii) modifications to the orchestrator may require extensive testing to ensure that it retains its correctness properties (e.g., see[22]).

Our work avoids changes to the underlying orchestration algorithm, and focuses those changes around the FMU itself in the form of semantic adaptations, using hierarchical co-simulation.

An adaptation targets an FMU, or group of FMUs, which we will call the *internal FMU(s)*, and the end result of an adaptation is a new FMU, which we call *external FMU*. The external FMU interacts with the internal FMU(s), without requiring them to be modified (*modularity*). The adjectives *internal* and *external* reflect the hierarchical nature of our approach and are illustrated in Figure 2.

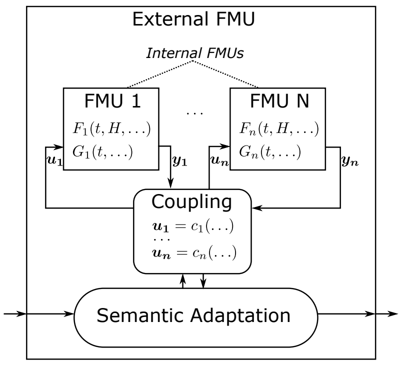


Figure 2. Internal FMUs, External FMU, and Semantic Adaptation.

We introduce below a non-exhaustive list of semantic adaptations that can be classified according to the interaction mismatch they intent to correct:

Signal Data Mismatch: Conversion of Units and Reference Frame translation.

Model of Computation Mismatch: Hold, Quantization, Data Triggered Execution, and Timed Transitions.

Capability Mismatch: Interpolation/Extrapolation of Inputs, Fixed Point Iteration, Multi-Rate

Adaptation, Time and Partial Derivative Adaptation, Accurate Threshold Crossing, and Re-Initialisation.

See[8,17] and references thereof, for variants of these adaptations.

Conversion of Units and Reference Frame Translation. The conversion of units and reference frame adaptations, take an internal FMU and create an external FMU whose inputs/outputs are algebraic transformations of the input/outputs of the internal FMU.

Interpolation/Extrapolation of Inputs. An FMU that stands for a continuous system, such as a DC motor, approximates its behavior trace by discretizing the time continuum into a finite set of points[23] and applying a numerical method at each of those points.

In co-simulation, when the orchestrator asks an FMU to compute the behavior trace over an interval of time, from t to $t+H$, the FMU discretizes the interval and computes the internal solution at each of these points, called micro-steps. The most common FMUs assume that, in between t and $t+H$ the inputs provided by the orchestrator are constant. Naturally, for large H, this assumption causes a significant error in the co-simulation[10,24–26].

Instead of reducing H, it is possible to adapt the FMU to better approximate its inputs. Essentially, the external FMU discretizes the interval $t \to t+H$ and runs the state transition function of the internal FMU multiple times, providing an approximated input at each of the time points. The internal FMU will still assume a constant input, but will do so in smaller intervals of time.

Fixed Point Iteration. If an algebraic loop exists, then the involved units will belong to the same cycle in the corresponding dependency graph.

As proposed in[8,16], given a co-simulation scenario (recall Equation (2)) that has one cycle involving at least two simulation units (non-trivial), it is possible to create an external FMU that replaces all the units in the cycle. All the couplings external to the cycle become couplings to the hierarchical simulation unit.

At each state transition of the external FMU, a fixed point iteration technique is applied to the inputs/outputs of the internal FMUs.

If a scenario has multiple non-trivial cycles, this adaptation can be applied to reduce the scenario to one where all the algebraic loops are solved[16]. Algorithm 1 can then be applied to compute the co-simulation.

Multi-Rate Adaptation. For FMUs simulating first order Ordinary Differential Equations (ODE), the larger the interval between the points, the less accurate the computed behavior trace will be[27].

The multi-rate adaptation is used to increase the accuracy while not sacrificing the performance in a co-simulation. Applied to co-simulation, the technique, well known in the circuit simulation domain[28], consists of a groups of interconnected internal FMUs that communicate more frequently[16,29]. This can serve two purposes: optimize the communication cost between the internal units[8], or optimize the accuracy of the co-simulation (especially when the internal units are physically tightly coupled[17]).

Similarly to the input extrapolation adaptation, the state transition function of the external unit instructs the internal units to perform multiple steps and exchange values at each of those steps. The higher the rate of the adaptation, the higher the number of internal steps performed.

This adaptation can be combined with the approximation of inputs adaptation, to provide for approximated inputs at each of the internal state transition invocations.

Time and Partial Derivative Adaptation. Time and partial derivative information about each simulation unit's outputs can be used to optimize the co-simulation process in many different ways (e.g., see[30]).

In the FMI standard, since the FMUs can optionally provide time and partial derivative information, it is often the case that some units do not support it. To mitigate this, a derivative adaptation can be used to produce an external FMU that provides (numerically estimated) partial and time derivatives.

Accurate Threshold Crossing. A co-simulation trace is more accurate if all units exchange values at the time when a certain signal crosses a given threshold. The problem of accurately finding that time is well known in the hybrid system simulation domain[31,32] and many techniques exist to address it[23,27]. In FMI co-simulation, the most basic technique to accurately locate a crossing consists of rejecting a step size and proposing a new one, that possibly coincides with the threshold crossing moment.

The accurate zero crossing adaptation ensures that the external FMU rejects the proposed step size when one of the inputs of the internal FMU crosses a significant threshold *too late*[15].

Re-Initialisation. An internal FMU that is expecting a smooth input signal may yield unexpected behavior trace when given a discontinuous signal (we consider a discontinuous signal to be a sufficiently rapid changing one in between co-simulation communication points)[10,33,34]. For example, an FMU that is using a multi-step numerical solver which assumes the input to be continuous (see, e.g.,[35] for a possible solution to this problem).

A re-initialization adaptation ensures that the external unit: (1) locates accurately the time of the discontinuity (e.g., in the same manner as the accurate crossing adaptation), and (2) the external unit is properly reset before handling the new value of the input. In the FMI standard, item (2) requires three steps: save the unit state; reset and initialize the unit; and restore the state.

Quantization. Quantization is an adaptation commonly used to convert a continuous signal into a discrete event one. The (continuous) set of possible input values is discretized into regions and, during the co-simulation, whenever the continuous signal enters a new region, an event is produced[36,37].

In co-simulation, this adaptation transforms an internal FMU that expects continuous inputs and produces continuous outputs, into an external FMU that deals with events (see, e.g.,[38–41]).

The realization of this adaptation is very similar to the zero crossing one, except that the thresholds to locate are induced by the input space discretization.

Hold. The hold family of adaptations can be seen as the dual of the multi-rate adaptations.

If an internal FMU should run slower than the rest of the simulation units, then it can be adapted with a hold adaptation. The external FMU will *trick* the orchestrator and obey to the proposed step sizes, but will avoid executing the internal FMU every time a step is requested. For example, if a zero order Hold adaptation is used, then the external unit will produce an output that is equal to the most recent output produced by the internal unit.

There are many variants of this adaptation, with varying degrees of accuracy, borrowed from well known approximation techniques[27].

The two adaptations below are novel in FMI based co-simulation domain, but well known in the discrete event domain.

Data Triggered Execution. The data triggered execution is an adaptation most useful when the modeller knows that a particular internal FMU will only produce relevant behavior when certain conditions are true over its inputs. The adaptation executes the internal FMU only when these conditions are met.

Timed Transitions. The time transition adaptation can be used when the internal FMU is known to have internal state changes, triggered after a known amount of time. The adaptation will query the internal FMU to know when exactly should the next state transition function call take place, and will call it only when that time is arrived. It can be combined with the data triggered execution to achieve a *lazy* execution of units.

Each of the semantic adaptations described above has many variants that make its ad-hoc implementations not only error prone, but also tedious. Additionally, one can extract the shared commonalities in the implementation of all semantic adaptations. The interplay between many small variants and shared commonalities is one of the motivating factors to use a Domain Specific Language for the description of the adaptations.

Domain-Specific Languages

Domain-specific languages (DSLs) offer a way to deal with the essential complexity of a given domain, while avoiding its accidental complexity[42].

We highlight two important advantages that come from the use of a DSL in the context of our contribution:

1. The most common tasks in the target domain are performed in a very simple, productive, and intuitive manner (for a trained domain expert) — the descriptions made in our DSL do not deal with the idiosyncrasies of an implementation of the FMI Standard, even though a FMI compliant external FMU can be generated.
2. By maximally constraining the user, a DSL ensures that he/she makes less mistakes and allows domain level validation — our DSL allows the user to specify extra information that can be used to detect mistakes (a simple validation being the compatibility of units in inputs/outputs).

Running Example

To showcase the language, the case study we present is adapted from a power window system, described in[43,44]. This system is the familiar automated car window, which responds to the driver/passenger pressing up/down buttons to raise/lower it. If an obstruction is detected, the window retracts for a few moments to avoid injury. This example was chosen for its heterogeneity and need for semantic adaptations.

The Example Scenario

Figure 3 shows the co-simulation scenario of the power window, consisting of five FMUs, with the illustrated input and output ports. The figure is a block representation of a co-simulation scenario as described in Equation (2). The FMUs were produced by the authors using independent tools.

The environment FMU, coded manually, is an abstraction of the behavior of the driver and passenger. Whenever the driver/passenger pushes a button up/down, the respective output will *pulse* to signal the event. When the button is released, the stop output pulses.

The controller FMU, produced from the Yakindu Statecharts tool, represents the software subsystem that ensures the safe operation of the window. It gets boolean pulse inputs and decides whether the motor should go up or down, through its boolean pulse outputs. If an object is detected (that is, obj_detected pulses) and the passenger (or driver) has pushed the up button, then the controller should instruct the DC motor to go down for one second. This is done by

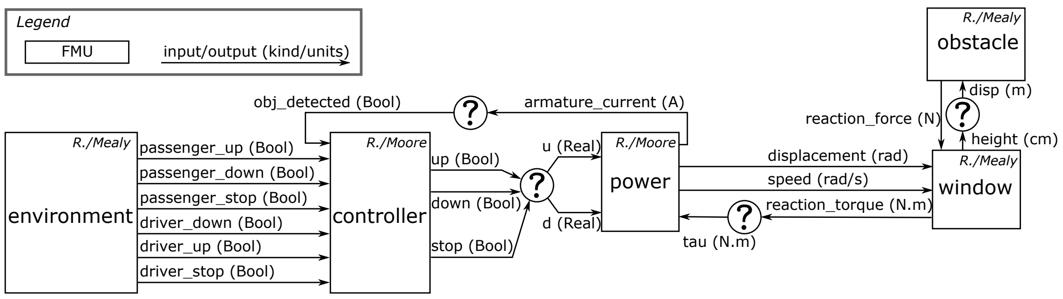


Figure 3. Power window co-simulation scenario.

pulsing the down output and, after 1 second, pulsing the stop output.

The power is an ODE based unit, exported with OpenModelica, representing the DC motor and the up/down switched circuit that drives the motor. Whenever the u input is bigger than 0.5, the DC motor moves the window up. Analogously, whenever the d input is bigger than 0.5, it moves the window down.

The window and obstacle are stateless units, coded manually, that map the inputs to the outputs using algebraic equations. The obstacle FMU outputs a force proportional to how compressed it is. Non-zero compression happens only when the input displacement exceeds a given threshold (e.g., $0.45m$).

An object is detected when the armature_current spikes, caused by a sudden increase in the reaction_torque input of the DC motor, cause in turn by an increase in the reaction_forced of the object being compressed.

As illustrated in the figure, *all units in this example are reactive*, so the controller, power, window, and obstacle form a single cycle. The power and controller are *Moore* and the remaining units are *Mealy*.

Figure 4 shows the behavior trace of the example produced via a monolithic model produced in OpenModelica[45]. In the figure, the driver continuously pushes the up button, asking the controller to move the window up, but the controller detects an object at about 2.5 seconds (due to the armature current spike), which causes it to override the requests of the driver and retract the window for 1 second.

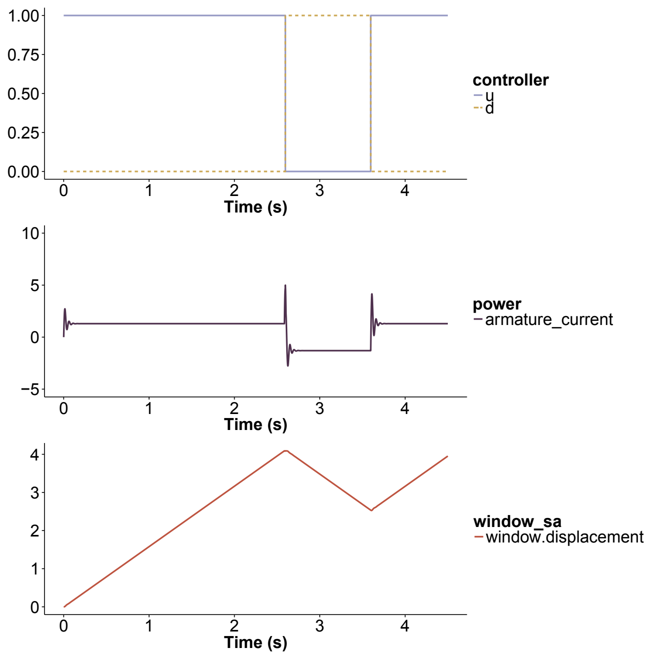


Figure 4. Power window monolithic simulation results.

Semantic Adaptations

The scenario presented in Figure 3 cannot be used *as is* to compute a co-simulation as the one shown in Figure 4 because the FMUs are incompatible.

The adaptations that need to be made were introduced in the “Background” section, and are detailed in the list below and illustrated in Figure 5. These will be referred to throughout this document.

lazy_sa – for controller:

- execute only if the inputs change (*data triggered execution*).
- execute only when its state transition needs to be called (*timed transition adaptation*) due to internal triggers. In FMI, this information can be obtained by asking controller to perform a very large step.
- zero order hold its outputs.

controller_sa – for lazy_sa:

- map the armature_current to a boolean signal object_detected that is $true$ whenever there is a threshold crossing. The condition that defines the crossing is $|\text{armature\_current}| > 5$ [*] and the lazy_sa unit should be invoked at the time of crossing.
- convert output, taking into account the stop signal.

window_sa – for window:

- negate the reaction_torque value;

[*]The value 5 is used here for the purposes of illustration. In practice, it is obtained by calibration with the DC Motor.

- convert the units of height from centimetres to metres.

power_sa – for power:

- ignore the algebraic loop between controller and power, and between the power and window, by delaying the outputs of the power by one co-simulation step. This effectively makes the external FMU a delayed unit.

loop_sa – for window_sa and obstacle:

- solve the algebraic loop between obstacle and window_sa by successive substitution providing an initial guess for height.

rate_sa – in order to prevent divergence in the fixed point iteration caused by the above adaptation, smaller communication step sizes should be taken between the obstacle and the window FMUs. To this end:

- use a multi-rate adaptation, where loop_sa is executed 10 times faster than the remaining scenario.
- interpolate the input signal motor_speed.

Hierarchical Co-simulation for Semantic Adaptation

The most straightforward way of dealing with semantic adaptations is by creating a master algorithm that implements them. There are multiple problems with this approach: 1) it forces the master algorithm to be specific to the scenario, which hinders the potential for reuse; and 2) it violates the *transparency* principle by not allowing the FMU (plus adaptations) to be easily imported onto other tools that perform co-simulation, such as Simulink®, INTO-CPS[46], or DACCOSIM[47].

To avoid these problems, we implement the semantic adaptations as FMUs, in a hierarchical way. In fact, our language defines semantic adaptations (plus internal FMUs) as FMUs themselves, allowing for adaptations to be described “on top of” other adaptations. This way, the orchestrator and semantic adaptations can be clearly separated, as well as the semantic adaptations between themselves.

As part of our contribution, we extend the definitions provided in Section “Background” to explain what hierarchical co-simulation is, and we give an overview on how the main semantic adaptations are implemented.

Hierarchical Co-simulation

Before giving the formal definition of hierarchical co-simulation, we start with an example of a “default” hierarchical co-simulation unit is and does.

A default hierarchical simulation unit is one that *wraps* a set of connected internal units, along with their inter-dependencies, and behaves in a manner that is indistinguishable from any other simulation unit. The internal FMUs have internal inputs/outputs (in between the units) and external inputs/outputs. This is called the default hierarchical unit because it does not adapt the behavior of the internal units. It merely wraps them.

To give details about how the default hierarchical unit is constructed, we extend the definition of co-simulation scenario to make the distinction between internal and external inputs. Let $\boldsymbol{u_{ext}}$ denote the input vector that is external to the co-simulation scenario. A co-simulation scenario with $D = \{1, \ldots, n\}$ units, and with external input $\boldsymbol{u_{ext}}$, is then described as:

$$\begin{cases} \left\langle \boldsymbol{x_i}(t+\tilde{H}_i), \tilde{H}_i \right\rangle = F_i(t, H, \ldots) \\ \boldsymbol{y_i}(t) = G_i(t, \ldots) \\ \boldsymbol{u_i}(t) = c_i(\boldsymbol{u_{ext}}(t), \boldsymbol{y_1}(t), \ldots, \boldsymbol{y_{i-1}}(t), \boldsymbol{y_{i+1}}(t), \ldots, \boldsymbol{y_n}(t)) \\ \boldsymbol{x_i}(0) = Init(\ldots) \end{cases}$$
$$\text{for each} \quad i \in D \tag{3}$$

Given then a co-simulation scenario as defined in Equation (3), and assuming that the topological order $\sigma : \mathbb{N} \to D$ is well defined, the default hierarchical *reactive Mealy* FMU is constructed by:

1. aggregating the state $\boldsymbol{x_i}$ and the previous input $\boldsymbol{up_i}$ of each FMU i, into a single entity $\boldsymbol{x}$ that becomes the state of the hierarchical unit;
2. implementing the state transition function as a single co-simulation step of Algorithm 1.

Formally, the unit is defined as:

$$\begin{aligned} \left\langle \boldsymbol{x}(t+\tilde{H}), \tilde{H} \right\rangle &= F(t, H, \boldsymbol{x}(t), \boldsymbol{u_{ext}}(t+H)) \\ \boldsymbol{y}(t) &= G(t, \boldsymbol{x}(t), \boldsymbol{u_{ext}}(t)) \\ \boldsymbol{x}(0) &= Init(\boldsymbol{u_{ext}}(0)) \end{aligned} \tag{4}$$

where: $\boldsymbol{x} = [\boldsymbol{up_1}, \ldots, \boldsymbol{up_n}, \boldsymbol{x_1}, \ldots, \boldsymbol{x_n}]^T$ is the total state vector and $[\cdot]^T$ is the matrix transpose operation; the initial state vector is calculated by the $Init$ function, defined in Algorithm 2, which finds the initial inputs and states to each of the internal units depending on their types; $\boldsymbol{u_{ext}}$ is the external input vector; function G is described in Algorithm 3, which computes the outputs of all internal units from the given inputs; and function F is detailed in Algorithm 4, which executes a single co-simulation step of Algorithm 1 and returns the minimum step size selected.

The construction of the default hierarchical reactive Moore, delayed Mealy, or delayed Moore, is done similarly and we omit it. The next subsection presents similar constructions for all kinds of units, incorporating adaptations.

The default hierarchical unit gives the basic transformation that underlies the semantic adaptation of one, or a connected group of, internal FMUs. In the subsection below, we describe the generic mechanism that enables the creation of hierarchical units with semantic adaptations.

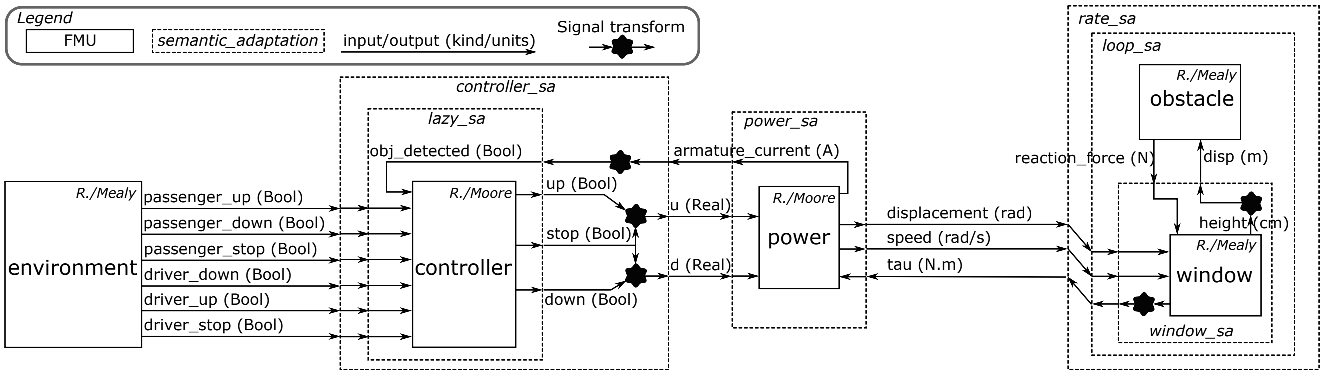


Figure 5. The modelled adaptations in the power window example.

Algorithm 2: *Init* function of the default hierarchical *reactive Mealy*, described in Equation (4).

```
Function $Init(\boldsymbol{u_{ext}})$
    for $i = 1, \ldots, n$ do
        $\boldsymbol{x_i} := \boldsymbol{up_i} := \boldsymbol{y_i} := \boldsymbol{0}$ ;
    end
    for $j \in (1, \ldots, n)$ do
        $\boldsymbol{up_{\sigma(j)}} :=$
          $c_{\sigma(j)}(\boldsymbol{u_{ext}}, \boldsymbol{y_1}, \ldots, \boldsymbol{y_{\sigma(j)-1}}, \boldsymbol{y_{\sigma(j)+1}}, \ldots, \boldsymbol{y_n})$;
        $\boldsymbol{x_{\sigma(j)}} := Init_{\sigma(j)}(\boldsymbol{up_{\sigma(j)}})$ or $Init_{\sigma(j)}()$ ;
        $\boldsymbol{y_{\sigma(j)}} := G_{\sigma(j)}(0, \boldsymbol{x_{\sigma(j)}}, \boldsymbol{up_{\sigma(j)}})$
                        or $G_{\sigma(j)}(0, \boldsymbol{x_{\sigma(j)}})$;
    end
    return $[\boldsymbol{up_1}, \ldots, \boldsymbol{up_n}, \boldsymbol{x_1}, \ldots, \boldsymbol{x_n}]^T$;
end
```

Algorithm 3: Output function of the default hierarchical *reactive Mealy*, described in Equation (4).

```
Function $G(t, [\boldsymbol{up_1}, \ldots, \boldsymbol{up_n}, \boldsymbol{x_1}, \ldots, \boldsymbol{x_n}]^T, \boldsymbol{u_{ext}})$
    for $i = 1, \ldots, n$ do
        $\boldsymbol{uc_i} := \boldsymbol{y_i} := \boldsymbol{0}$ ;
    end
    for $j \in (1, \ldots, n)$ do
        $\boldsymbol{uc_{\sigma(j)}} :=$
          $c_{\sigma(j)}(\boldsymbol{u_{ext}}, \boldsymbol{y_1}, \ldots, \boldsymbol{y_{\sigma(j)-1}}, \boldsymbol{y_{\sigma(j)+1}}, \ldots, \boldsymbol{y_n})$;
        $\boldsymbol{y_{\sigma(j)}} := G_{\sigma(j)}(t, \boldsymbol{x_{\sigma(j)}}, \boldsymbol{uc_{\sigma(j)}})$
                        or $G_{\sigma(j)}(t, \boldsymbol{x_{\sigma(j)}})$;
    end
    return $[\boldsymbol{y_1}, \ldots, \boldsymbol{y_n}]^T$;
end
```

Algorithm 4: State transition function of the default hierarchical *reactive Mealy*, described in Equation (4).

```
Function
  $F(t, H, [\boldsymbol{up_1}, \ldots, \boldsymbol{up_n}, \boldsymbol{x_1}, \ldots, \boldsymbol{x_n}]^T, \boldsymbol{u_{ext}})$
    for $i = 1, \ldots, n$ do
        $\boldsymbol{uc_i} := \boldsymbol{y_i} := \boldsymbol{0}$ ;
    end
    for $j \in (1, \ldots, n)$ do
        $\boldsymbol{uc_{\sigma(j)}} :=$
          $c_{\sigma(j)}(\boldsymbol{u_{ext}}, \boldsymbol{y_1}, \ldots, \boldsymbol{y_{\sigma(j)-1}}, \boldsymbol{y_{\sigma(j)+1}}, \ldots, \boldsymbol{y_n})$;
        $\left\langle \boldsymbol{\tilde{x}_{\sigma(j)}}, \tilde{H}_{\sigma(j)} \right\rangle :=$
          $F_{\sigma(j)}(t, H, \boldsymbol{x_{\sigma(j)}}, \boldsymbol{uc_{\sigma(j)}}$ or $\boldsymbol{up_{\sigma(j)}})$ ;
        $\boldsymbol{y_{\sigma(j)}} := G_{\sigma(j)}(t + \tilde{H}_{\sigma(j)}, \boldsymbol{\tilde{x}_{\sigma(j)}}, \boldsymbol{uc_{\sigma(j)}})$
                        or $G_{\sigma(j)}(t + \tilde{H}_{\sigma(j)}, \boldsymbol{\tilde{x}_{\sigma(j)}})$;
    end
    $\tilde{H} := \min_{i \in D}(\tilde{H}_i)$;
    return $\left\langle [\boldsymbol{uc_1}, \ldots, \boldsymbol{uc_n}, \boldsymbol{\tilde{x}_1}, \ldots, \boldsymbol{\tilde{x}_n}]^T, \tilde{H} \right\rangle$;
end
```

Generic Semantic Adaptation

Previous work[14,15] supports the hypothesis that any semantic adaptation can be described by the following elements, that mediate the interactions of the external FMU with the internal units:

- *external input rules*, describing how the inputs provided to the external FMU are stored in the state of the external FMU;
- *internal input rules*, detailing how the values stored internally are mapped into inputs of the internal FMUs;
- *control rules*, determining what happens when the state transition function of the external FMU is invoked;
- *internal output rules*, describing how the outputs of the internal FMUs are stored in the state of the external FMU;
- *external output rules*, detailing how the values stored in the state of the external FMU are mapped to output values of the external FMU;

This subsection formalizes how a generic external FMU incorporating the above rules is constructed.

To formalize the above rules, we define the state of the external FMU. The external FMU is constructed from a given co-simulation scenario, defined in Equation (3), with $D = \{1, \ldots, n\}$ units and external input vector $\boldsymbol{u_{ext}}$. Its state is then defined as

$$\boldsymbol{x} = [\boldsymbol{x_{in}}, \boldsymbol{x_{ctrl}}, \boldsymbol{x_{out}}, \boldsymbol{x_1}, \ldots, \boldsymbol{x_n}]^T$$

with $\boldsymbol{x_{in}}$, $\boldsymbol{x_{ctrl}}$, and $\boldsymbol{x_{out}}$, denoting the input, output and control storage vectors, respectively, and $\boldsymbol{x_1}, \ldots, \boldsymbol{x_n}$ being the internal units' states. The vectors $\boldsymbol{x_{in}}$, $\boldsymbol{x_{ctrl}}$, and $\boldsymbol{x_{out}}$ form the semantic adaptation storage and depend on the adaptations implemented in the external FMU.

Depending on the kind of external FMU being constructed, its initial state is computed by

$$\begin{aligned} Init(\boldsymbol{u_{ext}}) &= [\boldsymbol{x_{in}}, \boldsymbol{x_{ctrl}}, \boldsymbol{x_{out}}, \boldsymbol{x_1}, \ldots, \boldsymbol{x_n}]^T \\ \text{or } Init() &= [\boldsymbol{x_{in}}, \boldsymbol{x_{ctrl}}, \boldsymbol{x_{out}}, \boldsymbol{x_1}, \ldots, \boldsymbol{x_n}]^T \end{aligned}$$

where $Init()$, to be detailed shortly, makes use of the initialization functions $Init_i$ of the internal units to get their initial states.

We now introduce the formal representation of the semantic adaptation rules, introduced at the beginning of this subsection:

- The application of the *external input rules* to the provided input is

$$In([\boldsymbol{x_{in}}, \boldsymbol{x_{ctrl}}, \boldsymbol{x_{out}}]^T, \boldsymbol{u_{ext}}) = \boldsymbol{\tilde{x}_{in}}$$

- The application of the *internal input rules* to create the internal input vector is denoted as

$$MapIn([\boldsymbol{x_{in}}, \boldsymbol{x_{ctrl}}, \boldsymbol{x_{out}}]^T, h, dt) = [\boldsymbol{\tilde{u}_1}, \ldots, \boldsymbol{\tilde{u}_n}]^T$$

  This function is used whenever the input to any of the internal units needs to be computed. It is used in the $Ctrl$ rules (defined next) and in the output function of the external unit. In most adaptations, this function is invoked immediately before a call to the state transition function F_i of any internal unit. In line with the FMU interface, h is the communication step size that will be passed to the state transition F_i invocation, dt is the displacement of the time in unit i, relative to the external unit, and $\boldsymbol{\tilde{u}_i}$ denotes the vector that will be used as external input to unit i, or ignored if the unit does not depend on the external input. Multiple calls to this function can be made: potentially one per internal state transition call.
- The application of the *control rules*, to compute the new state $\tilde{x}_i$ of each internal unit i, the step size advanced $\tilde{H}$, and the new control/output storage state $\boldsymbol{\tilde{x}_{ctrl}}, \boldsymbol{\tilde{x}_{out}}$ of the semantic adaptation, is

$$\begin{aligned} &Ctrl(t, H, [\boldsymbol{x_{in}}, \boldsymbol{x_{ctrl}}, \boldsymbol{x_{out}}]^T, [\boldsymbol{x_1}, \ldots, \boldsymbol{x_n}]^T) = \\ &\quad \left\langle \boldsymbol{\tilde{x}_{ctrl}}, \boldsymbol{\tilde{x}_{out}}, [\boldsymbol{\tilde{x}_1}, \ldots, \boldsymbol{\tilde{x}_n}]^T, \tilde{H} \right\rangle \end{aligned}$$

  This function invokes the $MapIn$/$MapOut$ functions before/after a state transition of an internal unit is invoked.
- The application of the *internal output rules*

$$MapOut([\boldsymbol{x_{in}}, \boldsymbol{x_{ctrl}}, \boldsymbol{x_{out}}]^T, [\boldsymbol{y_1}, \ldots, \boldsymbol{y_n}]^T, h, dt) = \boldsymbol{\tilde{x}_{out}}$$

  Analogously to the $MapIn$, the invocation of this function is controlled by the $Ctrl$. Parameters h and dt denote the communication step size, and time displacement, passed as arguments to the most recently invoked state transition function F_i.
- The application of the *external output rules* to compute the external outputs, from the semantic adaptation state

$$Out([\boldsymbol{x_{in}}, \boldsymbol{x_{ctrl}}, \boldsymbol{x_{out}}]^T) = \boldsymbol{y}$$

Intuitively, the internal input/output functions serve to decouple the rate of execution of the internal units, from the rate of execution of the external FMU.

A semantic adaptation is a concrete definition of:

- Storage structure — $\boldsymbol{x_{in}}$, $\boldsymbol{x_{ctrl}}$, and $\boldsymbol{x_{out}}$;
- Initialization — $Init()$;
- External input rules — In;
- Internal input rules — $MapIn$;
- Control rules — $Ctrl$;
- Internal output rules — $MapOut$;
- External output rules — Out;

We now describe how these functions are used in the specification of an external FMU.

The generic external unit is defined exactly as a simulation unit (recall Equation (1)):

$$\begin{aligned} \left\langle \boldsymbol{x}(t+\tilde{H}), \tilde{H} \right\rangle &= F(t, H, \boldsymbol{x}(t), \boldsymbol{u_{ext}}(t+H) \text{ or } \boldsymbol{u_{ext}}(t)) \\ \boldsymbol{y}(t) &= G(t, \boldsymbol{x}(t), \boldsymbol{u_{ext}}(t)) \text{ or } G(t, \boldsymbol{x}(t)) \\ \boldsymbol{x}(0) &= Init(\boldsymbol{u_{ext}}) \text{ or } Init() \end{aligned} \tag{5}$$

where $\boldsymbol{x} = [\boldsymbol{x_{in}}, \boldsymbol{x_{ctrl}}, \boldsymbol{x_{out}}, \boldsymbol{x_1}, \ldots, \boldsymbol{x_n}]^T$ denotes the state of the external FMU. Both an external reactive or delayed unit has the same implementation of F, described in Algorithm 5 (but note that the definition of $Ctrl$ will likely differ). The definitions of G differ for a Mealy or Moore external unit, and are detailed in Algorithm 6.

In Algorithm 6, we stress the following:

- The definitions take into account that it may not be possible to sort the internal units topologically, so the semantic adaptations support dependency cycles.
- Multiple calls to G can be made without changing the state of the external unit.
- If a Moore external FMU has at least one internal unit which depends on external input, then this input must be stored in the input storage $\boldsymbol{x_{in}}$ of the semantic adaptation by the In function (Line 2 of Algorithm 5), and then retrieved by the $MapIn$ function (Line 8 of Algorithm 6).

To make these definitions easier to understand, we provide two examples: the default reactive Mealy hierarchical unit presented in the sub-previous section, and the algebraic loop semantic adaptation that involves the obstacle and window_sa units of the power window example (loop_sa).

Algorithm 5: State transition function of the generic external FMU, defined in Equation (5).

```
Function $F(t, H, [\boldsymbol{x_{in}}, \boldsymbol{x_{ctrl}}, \boldsymbol{x_{out}}, \boldsymbol{x_1}, \ldots, \boldsymbol{x_n}]^T, \boldsymbol{u_{ext}})$
    $\tilde{\boldsymbol{x}}_{\boldsymbol{in}} := In([\boldsymbol{x_{in}}, \boldsymbol{x_{ctrl}}, \boldsymbol{x_{out}}]^T, \boldsymbol{u_{ext}})$;
    $\left\langle \tilde{\boldsymbol{x}}_{\boldsymbol{ctrl}}, \tilde{\boldsymbol{x}}_{\boldsymbol{out}}, [\tilde{\boldsymbol{x}}_{\boldsymbol{1}}, \ldots, \tilde{\boldsymbol{x}}_{\boldsymbol{n}}]^T, \tilde{H} \right\rangle :=$
        $Ctrl(t, H, [\tilde{\boldsymbol{x}}_{\boldsymbol{in}}, \boldsymbol{x_{ctrl}}, \boldsymbol{x_{out}}]^T, [\boldsymbol{x_1}, \ldots, \boldsymbol{x_n}]^T)$;
    return $\left\langle [\tilde{\boldsymbol{x}}_{\boldsymbol{in}}, \tilde{\boldsymbol{x}}_{\boldsymbol{ctrl}}, \tilde{\boldsymbol{x}}_{\boldsymbol{out}}, \tilde{\boldsymbol{x}}_{\boldsymbol{1}}, \ldots, \tilde{\boldsymbol{x}}_{\boldsymbol{n}}]^T, \tilde{H} \right\rangle$;
end
```

The default reactive Mealy hierarchical unit can be informally described as follows:

- the state $\boldsymbol{x_{ctrl}}$ of the semantic adaptation includes the previous inputs of the internal units;
- the $Init$ function is analogue to the one described in Algorithm 2;
- the In, $MapIn$, $MapOut$, and Out, are roughly identity functions;
- and the $Ctrl$ function implements the body of F, in Algorithm 4;

Formally, functions $Init$ and $Ctrl$ are defined in Algorithm 7, and:

$$
\begin{aligned}
&In([\boldsymbol{x_{in}}, \boldsymbol{x_{ctrl}}, \boldsymbol{x_{out}}]^T, \boldsymbol{u_{ext}}) = \boldsymbol{u_{ext}} \\
&MapIn([\boldsymbol{x_{in}}, \boldsymbol{x_{ctrl}}, \boldsymbol{x_{out}}], h, dt) = \\
&\qquad [\tilde{\boldsymbol{u}}_{\boldsymbol{1}}, \ldots, \tilde{\boldsymbol{u}}_{\boldsymbol{n}}]^T, \text{ with } \tilde{\boldsymbol{u}}_{\boldsymbol{i}} = \boldsymbol{x_{in}} \\
&MapOut([\boldsymbol{x_{in}}, \boldsymbol{x_{ctrl}}, \boldsymbol{x_{out}}]^T, [\boldsymbol{y_1}, \ldots, \boldsymbol{y_n}]^T, h, dt) = \\
&\qquad [\boldsymbol{y_1}, \ldots, \boldsymbol{y_n}]^T \\
&Out([\boldsymbol{x_{in}}, \boldsymbol{x_{ctrl}}, \boldsymbol{x_{out}}]^T) = \boldsymbol{x_{out}}
\end{aligned}
\tag{6}
$$

The second example refers to the adaptation loop_sa, which essentially performs a fixed point iteration between the obstacle and window_sa units, computing improved values for their input/outputs via successive substitution.

The external FMU, called loop_sa in Figure 5 is a reactive Moore unit, and has an input $\boldsymbol{u_{ext}} \in \mathbb{R}^2$ with two dimensions — displacement and speed — and one output – tau. Whenever the state transition of the external FMU is called, a successive substitution is performed between the two internal units, using the most recently found value of disp as an initial guess. Formally, let the index 1 refer to the window_sa unit, and 2 to obstacle, so that, e.g., uc_2 refers to the input to the obstacle unit. For the sake of simplicity, we assume that the system starts with all inputs/outputs being zero. Then, the functions that characterize the adaptation are shown in Equation (7). Note that had we not assumed that the system starts with zero inputs/outputs, the $Init$ would have to compute a fixed point iteration to find a consistent initial state. This is possible with our formalization.

Algorithm 6: Output functions of the generic external FMU, per kind of unit, defined in Equation (5).

```
Function $G(t, [\boldsymbol{x_{in}}, \boldsymbol{x_{ctrl}}, \boldsymbol{x_{out}}, \boldsymbol{x_1}, \ldots, \boldsymbol{x_n}]^T, \boldsymbol{u_{ext}})$
    $\tilde{\boldsymbol{x}}_{\boldsymbol{in}} := In([\boldsymbol{x_{in}}, \boldsymbol{x_{ctrl}}, \boldsymbol{x_{out}}]^T, \boldsymbol{u_{ext}})$;
    if $\sigma$ is defined then
        for $i = 1, \ldots, n$ do
            $\boldsymbol{uc_i} := \boldsymbol{y_i} := \tilde{\boldsymbol{y}}_{\boldsymbol{i}} := \boldsymbol{0}$;
        end
        for $j \in (1, \ldots, n)$ do
            $[\tilde{\boldsymbol{u}}_{\boldsymbol{1}}, \ldots, \tilde{\boldsymbol{u}}_{\boldsymbol{n}}]^T :=$
                $MapIn([\tilde{\boldsymbol{x}}_{\boldsymbol{in}}, \boldsymbol{x_{ctrl}}, \boldsymbol{x_{out}}]^T, 0, 0)$;
            $\boldsymbol{uc_{\sigma(j)}} :=$
                $c_{\sigma(j)}(\tilde{\boldsymbol{u}}_{\boldsymbol{\sigma(j)}}, \boldsymbol{y_1}, \ldots, \boldsymbol{y_{\sigma(j)-1}}, \boldsymbol{y_{\sigma(j)+1}}, \ldots, \boldsymbol{y_n})$;
            $\boldsymbol{y_{\sigma(j)}} := G_{\sigma(j)}(t, \boldsymbol{x_{\sigma(j)}}, \boldsymbol{uc_{\sigma(j)}})$
                    or $G_{\sigma(j)}(t, \boldsymbol{x_{\sigma(j)}})$;
            $\tilde{\boldsymbol{x}}_{\boldsymbol{out}} :=$
                $MapOut([\tilde{\boldsymbol{x}}_{\boldsymbol{in}}, \boldsymbol{x_{ctrl}}, \boldsymbol{x_{out}}]^T, [\boldsymbol{y_1}, \ldots, \boldsymbol{y_n}]^T, 0, 0)$;
        end
    else
        $\tilde{\boldsymbol{x}}_{\boldsymbol{out}} := \boldsymbol{x_{out}}$;
    end
    $\boldsymbol{y} := Out([\tilde{\boldsymbol{x}}_{\boldsymbol{in}}, \boldsymbol{x_{ctrl}}, \tilde{\boldsymbol{x}}_{\boldsymbol{out}}]^T)$;
    return $\boldsymbol{y}$;
end
Function $G(t, [\boldsymbol{x_{in}}, \boldsymbol{x_{ctrl}}, \boldsymbol{x_{out}}, \boldsymbol{x_1}, \ldots, \boldsymbol{x_n}]^T)$
    if $\sigma$ is defined then
        for $i = 1, \ldots, n$ do
            $\boldsymbol{uc_i} := \boldsymbol{y_i} := \tilde{\boldsymbol{y}}_{\boldsymbol{i}} := \boldsymbol{0}$;
        end
        for $j \in (1, \ldots, n)$ do
            $[\tilde{\boldsymbol{u}}_{\boldsymbol{1}}, \ldots, \tilde{\boldsymbol{u}}_{\boldsymbol{n}}]^T :=$
                $MapIn([\boldsymbol{x_{in}}, \boldsymbol{x_{ctrl}}, \boldsymbol{x_{out}}]^T, 0, 0)$;
            $\boldsymbol{uc_{\sigma(j)}} :=$
                $c_{\sigma(j)}(\tilde{\boldsymbol{u}}_{\boldsymbol{\sigma(j)}}, \boldsymbol{y_1}, \ldots, \boldsymbol{y_{\sigma(j)-1}}, \boldsymbol{y_{\sigma(j)+1}}, \ldots, \boldsymbol{y_n})$;
            $\boldsymbol{y_{\sigma(j)}} := G_{\sigma(j)}(t, \boldsymbol{x_{\sigma(j)}}, \boldsymbol{uc_{\sigma(j)}})$
                    or $G_{\sigma(j)}(t, \boldsymbol{x_{\sigma(j)}})$;
            $\tilde{\boldsymbol{x}}_{\boldsymbol{out}} :=$
                $MapOut([\boldsymbol{x_{in}}, \boldsymbol{x_{ctrl}}, \boldsymbol{x_{out}}]^T, [\boldsymbol{y_1}, \ldots, \boldsymbol{y_n}]^T, 0, 0)$;
        end
    else
        $\tilde{\boldsymbol{x}}_{\boldsymbol{out}} := \boldsymbol{x_{out}}$;
    end
    $\boldsymbol{y} := Out([\boldsymbol{x_{in}}, \boldsymbol{x_{ctrl}}, \tilde{\boldsymbol{x}}_{\boldsymbol{out}}]^T)$;
    return $\boldsymbol{y}$;
end
```

The next section describes a DSL for the definition of such semantic adaptations. The examples provided in that section clarify the need for the semantic adaptation functions defined in the current section.

Algorithm 7: *Init* and *Ctrl* functions of the default reactive Mealy hierarchical unit.

```
Function $Init(\boldsymbol{u_{ext}})$
    for $i = 1, \ldots, n$ do
        $\boldsymbol{x_i} := \boldsymbol{up_i} := \boldsymbol{y_i} := \boldsymbol{0}$ ;
    end
    for $j \in (1, \ldots, n)$ do
        $\boldsymbol{up_{\sigma(j)}} := c_{\sigma(j)}(\boldsymbol{u_{ext}}, \boldsymbol{y_1}, \ldots, \boldsymbol{y_{\sigma(j)-1}}, \boldsymbol{y_{\sigma(j)+1}}, \ldots, \boldsymbol{y_n})$;
        $\boldsymbol{x_{\sigma(j)}} := Init_{\sigma(j)}(\boldsymbol{up_{\sigma(j)}})$ or $Init_{\sigma(j)}()$ ;
        $\boldsymbol{y_{\sigma(j)}} := G_{\sigma(j)}(0, \boldsymbol{x_{\sigma(j)}}, \boldsymbol{up_{\sigma(j)}})$
                    or $G_{\sigma(j)}(0, \boldsymbol{x_{\sigma(j)}})$;
    end
    $\boldsymbol{x_{in}} := \boldsymbol{x_{out}} := \boldsymbol{0}$ ;
    $\boldsymbol{x_{ctrl}} := [\boldsymbol{up_1}, \ldots, \boldsymbol{up_n}]^T$ ;
    return $[\boldsymbol{x_{in}}, \boldsymbol{x_{ctrl}}, \boldsymbol{x_{out}}, \boldsymbol{x_1}, \ldots, \boldsymbol{x_n}]^T$;
end
Function $Ctrl(t, H, \left\langle \boldsymbol{x_{in}}, [\boldsymbol{up_1}, \ldots, \boldsymbol{up_n}]^T, \boldsymbol{x_{out}} \right\rangle, [\boldsymbol{x_1}, \ldots, \boldsymbol{x_n}]^T)$
    $\boldsymbol{x_{ctrl}} := [\boldsymbol{up_1}, \ldots, \boldsymbol{up_n}]^T$;
    for $i = 1, \ldots, n$ do
        $\boldsymbol{uc_i} := \boldsymbol{y_i} := \boldsymbol{0}$;
    end
    for $j \in (1, \ldots, n)$ do
        $[\tilde{\boldsymbol{u}}_{\boldsymbol{1}}, \ldots, \tilde{\boldsymbol{u}}_{\boldsymbol{n}}]^T := MapIn([\boldsymbol{x_{in}}, \boldsymbol{x_{ctrl}}, \boldsymbol{x_{out}}]^T, 0, 0)$;
        $\boldsymbol{uc_{\sigma(j)}} := c_{\sigma(j)}(\tilde{\boldsymbol{u}}_{\boldsymbol{\sigma(j)}}, \boldsymbol{y_1}, \ldots, \boldsymbol{y_{\sigma(j)-1}}, \boldsymbol{y_{\sigma(j)+1}}, \ldots, \boldsymbol{y_n})$;
        $\left\langle \tilde{\boldsymbol{x}}_{\boldsymbol{\sigma(j)}}, \tilde{H}_{\sigma(j)} \right\rangle := F_{\sigma(j)}(t, H, \boldsymbol{x_{\sigma(j)}}, \boldsymbol{uc_{\sigma(j)}}$ or $\boldsymbol{up_{\sigma(j)}})$ ;
        $\boldsymbol{y_{\sigma(j)}} := G_{\sigma(j)}(t + \tilde{H}_{\sigma(j)}, \tilde{\boldsymbol{x}}_{\boldsymbol{\sigma(j)}}, \boldsymbol{uc_{\sigma(j)}})$
                    or $G_{\sigma(j)}(t + \tilde{H}_{\sigma(j)}, \tilde{\boldsymbol{x}}_{\boldsymbol{\sigma(j)}})$;
        $\tilde{\boldsymbol{x}}_{\boldsymbol{out}} := MapOut([\boldsymbol{x_{in}}, \boldsymbol{x_{ctrl}}, \boldsymbol{x_{out}}]^T, [\boldsymbol{y_1}, \ldots, \boldsymbol{y_n}]^T, 0, 0)$;
    end
    $\tilde{H} := \min_{i \in D}(\tilde{H}_i)$;
    return $\left\langle [\boldsymbol{uc_1}, \ldots, \boldsymbol{uc_n}]^T, \tilde{\boldsymbol{x}}_{\boldsymbol{out}}, [\tilde{\boldsymbol{x}}_{\boldsymbol{1}}, \ldots, \tilde{\boldsymbol{x}}_{\boldsymbol{n}}]^T, \tilde{H} \right\rangle$;
end
```

$$
\begin{aligned}
&Init(\boldsymbol{u_{ext}}) = [\boldsymbol{0}, \boldsymbol{0}, \boldsymbol{0}, Init_1(\boldsymbol{0}), Init_2(\boldsymbol{0})]^T \\
&In([\boldsymbol{x_{in}}, \boldsymbol{x_{ctrl}}, \boldsymbol{x_{out}}]^T, \boldsymbol{u_{ext}}) = \boldsymbol{u_{ext}} \\
&MapIn([\boldsymbol{x_{in}}, \boldsymbol{x_{ctrl}}, \boldsymbol{x_{out}}]^T, h, dt) = [\boldsymbol{x_{in}}, \boldsymbol{0}]^T \\
&MapOut([\boldsymbol{x_{in}}, \boldsymbol{x_{ctrl}}, \boldsymbol{x_{out}}]^T, [\boldsymbol{y_1}, \boldsymbol{y_2}]^T, h, dt) = \boldsymbol{y_1} \\
&Out([\boldsymbol{x_{in}}, \boldsymbol{x_{ctrl}}, \boldsymbol{x_{out}}]^T) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \boldsymbol{x_{out}}
\end{aligned}
\tag{7}
$$

Ctrl is defined in Algorithm 8

Algorithm 8: *Ctrl* function of external FMU loop_sa, illustrated in Figure 5.

```
Function $Ctrl(t, H, [\boldsymbol{x_{in}}, \boldsymbol{yp_1}, \boldsymbol{x_{out}}]^T, [\boldsymbol{x_1}, \boldsymbol{x_2}]^T)$
    $\boldsymbol{u_1} := \boldsymbol{0}$;
    $\boldsymbol{y_1} := \boldsymbol{yp_1}$;
    $\boldsymbol{u_2} := \boldsymbol{yp_2} := \boldsymbol{y_2} := 0$;
    $[\tilde{\boldsymbol{u}}_{\boldsymbol{1}}, \tilde{\boldsymbol{u}}_{\boldsymbol{2}}]^T := MapIn([\boldsymbol{x_{in}}, \boldsymbol{yp_1}, \boldsymbol{x_{out}}]^T, 0, 0)$;
    for $i \in (1, \ldots, MAX\_ITERATIONS)$ do
        $\boldsymbol{u_2} := c_2(\tilde{\boldsymbol{u}}_{\boldsymbol{2}}, \boldsymbol{y_1})$;
        $\left\langle \tilde{\boldsymbol{x}}_{\boldsymbol{2}}, \tilde{H}_2 \right\rangle := F_2(t, H, \boldsymbol{x_2}, \boldsymbol{u_2})$ ;
        $y_2 := G_2(t + \tilde{H}_2, \tilde{\boldsymbol{x}}_{\boldsymbol{2}}, \boldsymbol{u_2})$;
        $\boldsymbol{u_1} := c_1(\tilde{\boldsymbol{u}}_{\boldsymbol{1}}, \boldsymbol{y_2})$;
        $\left\langle \tilde{\boldsymbol{x}}_{\boldsymbol{1}}, \tilde{H}_1 \right\rangle := F_1(t, H, \boldsymbol{x_1}, \boldsymbol{u_1})$ ;
        $\boldsymbol{y_1} := G_1(t + \tilde{H}_1, \tilde{\boldsymbol{x}}_{\boldsymbol{1}}, \boldsymbol{u_1})$;
        if $\|\boldsymbol{y_1} - \boldsymbol{yp_1}\| \approx 0$ and $\|\boldsymbol{y_2} - \boldsymbol{yp_2}\| \approx 0$ then
            $\tilde{\boldsymbol{x}}_{\boldsymbol{out}} := MapOut([\boldsymbol{x_{in}}, \boldsymbol{x_{ctrl}}, \boldsymbol{x_{out}}]^T, [\boldsymbol{y_1}, \boldsymbol{y_2}]^T, 0, 0)$;
            break;
        else
            $\boldsymbol{yp_1} := \boldsymbol{y_1}$;
            $\boldsymbol{yp_2} := \boldsymbol{y_2}$;
    end
    return $\left\langle \boldsymbol{y_1}, \tilde{\boldsymbol{x}}_{\boldsymbol{out}}, [\tilde{\boldsymbol{x}}_{\boldsymbol{1}}, \tilde{\boldsymbol{x}}_{\boldsymbol{2}}]^T, H \right\rangle$;
end
```

A DSL for Semantic Adaptation

We introduce a DSL for the specification of the set of rules introduced in the previous section (which form a semantic adaptation). Since research in semantic adaptation is ongoing, the language should be expressive enough to cover future semantic adaptations. Additionally, the implementation should not violate the modularity and transparency principles.

To these ends, the DSL — named baseSA— mixes imperative concepts with convenient functions that perform common operations on simulation units. A description made in this DSL can be used to generate hierarchical units.

The language and the examples used in this paper are available for download*.

The baseSA allows the description of the internal FMUs and their couplings (that is, the internal scenario as described in Equation (3)), and *how* semantic adaptation rules (*Init*, *In*, *MapIn*, *Ctrl*, *MapOut*, and *Out*), are implemented.

The remainder of this section is organised as follows. First, the baseSA DSL is introduced by describing the semantic adaptations of the running example and what their intended meaning is. Then, a

*`https://msdl.uantwerpen.be/git/claudio/HybridCosimulation`

more detailed description of the language (syntax and semantics) is provided.

The baseSA DSL

```
1  semantic adaptation reactive mealy WindowSA windowSA
2  at "./path/to/WindowSA.fmu"
3
4    for inner fmu Window window
5      at "./path/to/Window.fmu"
6      with input ports displacement (rad), speed (rad/s), reaction_force (N)
7      with output ports height (cm), reaction_torque (N.m)
8
9  output ports disp (m)  <- window.height, tau
10
11 out rules {
12   true -> {} --> {
13     windowSA.tau := -window.reaction_force;
14   };
15 }
```

Listing 1: The simple data adaptation window_sa in baseSA.

Listing 1 shows the baseSA definition of the semantic adaptation that generates the window_sa in Figure 5. The first few lines (Line 1 and 2 in the example) of any description, declare the name of the semantic adaptation and where the resulting external FMU will be generated.

Following that, the internal scenario is declared. The example listing (Lines 4 – 7) declares a single internal FMU and its ports.

baseSA descriptions work by *exclusion*: the user only specifies what needs to be changed, and the rest is assumed from the information provided. Hence, Listing 1 only needs to declare the output ports of the external FMU (disp and tau), in Line 13, and how they get their values: disp gets its value implicitly from the height port, and tau gets its value explicitly (via the specification of output rules).

Lines 11–15 declare the output rules. These specify how the tau output port of the external FMU gets its value, and this is done by assigning it the value of the reaction_torque output port, of the window FMU. The examples declares a single output rule, but in general multiple output rules can be declared. In general, each output rule has three parts: a condition, a *MapOutRule* part (syntactically preceded by “->”), and a *OutRule* part (syntactically preceded by “–>”). The condition decides whether the rule should be applied, and the other two parts contribute to the definition of the corresponding functions $MapOut$ and Out, respectively.

Following the exclusion principle, Listing 1 omits several bits of information about the external FMU, that are required for a full definition of a semantic adaptation: input ports; $Init$ function; In function; $MapIn$ function; and $Ctrl$ function;

In general, this information is assumed by applying multiple conventions, detailed in Section “Semantics”. The intended behavior is to follow the default hierarchical unit definition wherever the information is omitted (recall Equation (6) and Algorithm 7). For the example in Listing 1, the following is applicable:

- The external FMU (windowSA) has an input port for every input port of any internal FMU that has no incoming connection. This means that windowSA has three input ports, each bound to the corresponding input port of the internal FMU window.
- Each of the input ports of the internal FMU that have no incoming connections, gets its value from the corresponding external input port declared by the previous convention. The implementation of bindings is made via a storage variable. In Listing 1, this means that an extra input rule is created to encode the transfer of values. The input storage variables are also created.
- Any output variable bindings are realized in a manner similar to the previous convention: add an output rule and declare the necessary output variables to perform the transfer of values.
- Any expression referring ot the output of any internal FMU, in the *Out* part of an output rule, is assumed to refer to the storage variable with the most recent value of that output (output variables are created for the outputs of each internal FMU). In Listing 1, this means that window.reaction_force, in Line 13, gets replaced by a reference to an output variable.
- After applying the previous two conventions, the implicit bindings are removed.

```
1  semantic adaptation reactive mealy WindowSA windowSA
2  at "./path/to/WindowSA.fmu"
3
4    for inner fmu Window window
5      at "./path/to/Window.fmu"
6      with input ports displacement (rad), speed (rad/s), reaction_force (N)
7      with output ports height (m), reaction_torque (N.m)
8
9  input ports   reaction_force,
10         displacement,
11         speed
12
13 output ports  disp,
14         tau
15
16 param   INIT_WINDOWSA_REACTION_FORCE := 0.0,
17     INIT_WINDOWSA_DISPLACEMENT := 0.0,
18     INIT_WINDOWSA_SPEED := 0.0,
19     INIT_WINDOW_REACTION_TORQUE := 0.0,
20     INIT_WINDOW_REACTION_HEIGHT := 0.0;
21
22 control rules {
23   var H_window := do_step(window, t, H);
24   return H_window;
25 }
26
27 in var  stored_windowsa_reaction_force := INIT_WINDOWSA_REACTION_FORCE,
28     stored_windowsa_displacement := INIT_WINDOWSA_DISPLACEMENT,
29     stored_windowsa_speed := INIT_WINDOWSA_SPEED;
30
31 in rules {
32   true -> {
33     stored_windowsa_reaction_force := windowSA.reaction_force;
34     stored_windowsa_displacement := windowSA.displacement;
35     stored_windowsa_speed := windowSA.speed;
36   } --> {
37     window.reaction_force := stored_windowsa_reaction_force;
38     window.displacement := stored_windowsa_displacement;
39      window.speed := stored_windowsa_speed;
40    };
41  }
42
43  out var stored_window_reaction_torque := INIT_WINDOW_REACTION_TORQUE,
44      stored_window_height := INIT_WINDOW_REACTION_HEIGHT;
45
46  out rules {
47    true -> {
48      stored_window_reaction_torque := window.reaction_torque;
49      stored_window_height := window.height;
50    } --> {
51      windowSA.disp := stored_window_height / 100;
52    };
53    true -> { } --> {
54      windowSA.tau := -stored_window_reaction_torque;
55    };
56  }
```

Listing 2: The adaptation window_sa in explicit form.

Listing 2 shows the same adaptation as Listing 1, after applying the conventions introduced above:

- All input ports and output ports of the external FMU are declared, with no implicit bindings defined.
- Input storage variables, and their initial values, are declared (stored_windowsa_reaction_force, and stored_windowsa_displacement, stored_windowsa_speed). These are part of the x_{in} state vector of the semantic adaptation.
- Output storage variables, and their initial values, are declared (stored_window_reaction_torque and stored_window_height), comprising part of the x_{out} state vector.
- A parameter per storage variable is added to allow the configuration of the initial value of that variable (technical detail: the parameters are mapped to FMI parameters).
- Input rules, as the one in Lines 31–41, are in general comprised of two parts: the *InRule* part, which in the example assigns values to the input storage variables; and the *MapInRule* part, which assigns the stored values to the input ports of the internal FMUs in the example. These make up the respective functions In and $MapIn$.
- The control rules make use of the special function H_window := do_step(window, t, H), which automatically: uses the $MapIn$ function to compute the inputs to the internal FMU window, computes any extra internal input (this applies to internal interconnected units), invokes the state transition function of window with t and H, and invokes the $MapOut$ function to compute its outputs. do_step also takes into account the type (Mealy/Moore and reactive/delayed) of the internal unit invoked. The returned value is the step size taken by the unit.
- Output rules defined the functions $MapOut$ (which stores the outputs of window in the output storage variables), and Out (which sets the outputs of the external FMU from the output storage variables). Notice that the conversion of units between the height and disp ports is also done.

In any baseSA description, there is no need to define explicitly the initial state (computed by the $Init$ function). It is inferred from the input, control and output storage variables, plus the information about the internal units (extracted from their *xml* description file).

```
1   semantic adaptation reactive moore LoopSA loop_sa
2   at "./path/to/LoopSA.fmu"
3
4     for inner fmu WindowSA window_sa
5       at "./path/to/WindowSA.fmu"
6       with input ports displacement (rad), speed (rad/s), reaction_force (N)
7       with output ports disp (m), tau (N.m)
8
9     for inner fmu Obstacle obstacle
10      at "./path/to/Obstacle.fmu"
11      with input ports disp (m)
12      with output ports reaction_force (m)
13
14    with window_sa.disp -> obstacle.disp
15    with obstacle.reaction_force -> window_sa.reaction_force
16
17  output ports tau <- window_sa.tau
18
19  param   MAXITER := 10,
20      REL_TOL := 1e-05,
21      ABS_TOL := 1e-05;
22
23  control var prev_disp := 0.0;
24  control rules {
25    var repeat := false;
26    for (var iter in 0 .. MAXITER) {
27      save_state(obstacle);
28      save_state(window_sa);
29      obstacle.disp := prev_disp;
30      do_step(obstacle,t,H);
31      do_step(window_sa,t,H);
32
33      repeat := is_close(prev_disp, window_sa.disp, REL_TOL, ABS_TOL);
34      prev_disp := window_sa.disp;
35      if (repeat) {
36        break;
37      } else {
38        rollback(obstacle);
39        rollback(window_sa);
40      }
41    }
42    return H;
43  }
```

Listing 3: Adaptation that generates loop_sa.

Listing 3 describes the adaptation defining the external FMU loop_sa in Figure 5. The adaptation is targeted at two internal FMUs (window_sa and obstacle) that are interconnected as specified in Lines 14–15. In general, the internal connectivity information is needed so that the generated code knows how to set the inputs to the internal FMUs. The listing does not declare input ports, therefore, according to the general conventions, the external FMU has all the input ports that that have no incoming connections (displacement and speed). A single output port is declared (tau), which gets its value from the tau output of window_sa.

Notice that the external FMU is declared as reactive Moore, and that the internal FMUs cannot be topologically sorted. Whenever this is the case, when the external output function is called, the values of the output ports returned (in the example, the value of tau) are the ones computed in the most recent state transition function.

The control block of Listing 3 implements Algorithm 8 with the following differences.

- As part of the semantics of the `do_step` function: the *MapInRule* and *MapOutRule* instructions (which are implicit in Listing 3 by convention) are executed automatically to set the inputs of the internal FMUs; and the inputs of each FMU, if unspecified by an assignment, are set according to the internal connectivity information declared in Lines 14–15.
- The convergence test (Line 33) is made only in the `disp` port (to simplify).
- The state manipulation of the internal FMUs is facilitated by the use of the `save_state` and `rollback` functions.

```
1  semantic adaptation reactive moore RateSA rate_sa
2  at "./path/to/RateSA.fmu"
3  
4    for inner fmu LoopSA loop_sa
5      at "./path/to/LoopSA.fmu"
6      with input ports displacement (rad), speed (rad/s)
7      with output ports tau (N.m)
8  
9  input ports speed
10 output ports tau <- loop_sa.tau
11 
12 param   RATE := 10;
13 
14 control var previous_speed := 0;
15 control rules {
16   var micro_step := H/RATE;
17   var inner_time := t;
18 
19   for (var iter in 0 .. RATE) {
20     do_step(loop_sa,inner_time,micro_step);
21     inner_time := inner_time + micro_step;
22   }
23 
24   previous_speed := current_speed;
25   return H;
26 }
27 
28 in var current_speed := 0;
29 in rules {
30   true -> {
31     current_speed := speed;
32   } --> {
33     loop_sa.speed := previous_speed + (current_speed - previous_speed)*(dt + h);
34   };
35 }
```

Listing 4: Adaptation that generates rate_sa.

The rate_sa adaptation is implemented in Listing 4. It is worth noticing the $MapIn$ portion of the input rules, in Line 33, which calculates the interpolation of the speed value. This function is called whenever inputs to the internal FMUs need to be provided, with $h = micro\_step$ being the communication step size asked to the internal FMU ($micro\_step$ refers to the argument used in the state transition invocation, in Line 20), and $dt = inner\_time - t$ (where $inner\_time$ is the argument used for the state transition call).

```
semantic adaptation reactive moore LazySA lazy_sa
at "./path/to/LazySA.fmu"

  for inner fmu Controller controller
  at "./path/to/Controller.fmu"
  with input ports obj_detected, passenger_up, passenger_down, passenger_stop,
       driver_up, driver_down, driver_stop
  with output ports up, down, stop

input ports obj_detected -> controller.obj_detected,
      passenger_up -> controller.passenger_up,
      passenger_down -> controller.passenger_down,
      passenger_stop -> controller.passenger_stop,
      driver_up -> controller.driver_up,
      driver_down -> controller.driver_down,
      driver_stop -> controller.driver_stop

output ports up, down, stop

param   INIT_OBJ_DETECTED := false,
    INIT_PASSENGER_UP := false,
    INIT_PASSENGER_DOWN := false,
    INIT_PASSENGER_STOP := false,
    INIT_DRIVER_UP := false,
    INIT_DRIVER_DOWN := false,
    INIT_DRIVER_STOP := false;

control var tn := -1.0,
      tl := -1.0,
      prev_obj_detected := INIT_OBJ_DETECTED,
      prev_passenger_up := INIT_PASSENGER_UP,
      prev_passenger_down := INIT_PASSENGER_DOWN,
      prev_passenger_stop := INIT_PASSENGER_STOP,
      prev_driver_up := INIT_DRIVER_UP,
      prev_driver_down := INIT_DRIVER_DOWN,
      prev_driver_stop := INIT_DRIVER_STOP;

control rules {
  if (tl < 0.0){
    tl := t;
  }

  var step_size := min(H, tn - t);
  if (lazy_sa.obj_detected != prev_obj_detected or
    lazy_sa.passenger_up != prev_passenger_up or
    lazy_sa.passenger_down != prev_passenger_down or
    lazy_sa.passenger_stop != prev_passenger_stop or
    lazy_sa.driver_up != prev_driver_up or
    lazy_sa.driver_down != prev_driver_down or
    lazy_sa.driver_stop != prev_driver_stop or
    (t+H) >= tn
  ){
    var step_to_be_done := (t+H-tl);
    var step_done := do_step(controller, t, step_to_be_done);
    tn := tl + step_done + get_next_time_step(controller);
    step_size := tl + step_done - t;
    tl := tl + step_done;
  }

  prev_obj_detected := lazy_sa.obj_detected;
  prev_passenger_up := lazy_sa.passenger_up;
  prev_passenger_down := lazy_sa.passenger_down;
  prev_passenger_stop := lazy_sa.passenger_stop;
  prev_driver_up := lazy_sa.driver_up;
  prev_driver_down := lazy_sa.driver_down;
  prev_driver_stop := lazy_sa.driver_stop;

  return step_size;
}
```

Listing 5: Adaptation that generates lazy_sa.

Listing 5 implements adaptation lazy_sa. This adaptation assumes the default mappings for the inputs, but is declares them because they are referred to in the $Ctrl$ block.

In general, every reference to an input port of the external FMU, made outside of the In block, is replaced with a reference to the variable that stores the most recently given value of that. For example, the expression `lazy_sa.obj_detected`, is replaced by the variable that stores that input.

The adaptation in Listing 5 performs two tasks: it keeps track of the previous value of each signal, and invokes the internal unit state transition function (i.e., the `do_step`) whenever there is a change; and it keeps track of the next time to execute the internal unit (assuming that no inputs change) and invokes it when such time arrives, to cater for internal timed transitions. At the same time, the output signals are always available (held constant) because of the storage output variables.

```
semantic adaptation reactive moore ControllerSA controller_sa
at "./path/to/ControllerSA.fmu"

  for inner fmu LazySA lazy
  at "./path/to/LazySA.fmu"
  with input ports obj_detected, passenger_up, passenger_down, passenger_stop,
       driver_up, driver_down, driver_stop
  with output ports up, down, stop

input ports armature_current -> lazy.obj_detected,
      passenger_up -> lazy.passenger_up,
      passenger_down -> lazy.passenger_down,
      passenger_stop -> lazy.passenger_stop,
      driver_up -> lazy.driver_up,
      driver_down -> lazy.driver_down,
      driver_stop -> lazy.driver_stop

output ports  u,
        d

param RTOL := 0.0001,
    ATOL := 1e-8,
    T := 5.0,
    INIT_V := 0.0;

control var c := false,
      p_v := INIT_V;
control rules {
  var step_size := H;
  var aux_obj_detected := false;
  var crossedTooFar := false;
  if ((not is_close(p_v, T, RTOL, ATOL) and p_v < T)
        and (not is_close(f_v, T, RTOL, ATOL) and f_v > T)) {
    crossedTooFar := true;
    var negative_value := p_v - T;
    var positive_value := f_v - T;
    step_size := (H * (- negative_value)) / (positive_value - negative_value);
  } else {
    if ((not is_close(p_v, T, RTOL, ATOL) and p_v < T)
          and is_close(f_v, T, RTOL, ATOL )) {
      c := true;
    }
  }

  if (not crossedTooFar){
    step_size := do_step(lazy, t, H);
  }

  if (is_close(step_size, H, RTOL, ATOL)) {
    p_v := f_v;
  }
  return step_size;
}

in var  f_v := INIT_V;
in rules {
  true -> {
    f_v := controller_sa.armature_current;
  } --> {
    lazy.obj_detected := c;
  };
}

out rules {
  lazy.up -> { } --> {controller_sa.u := 1.0; };
  not lazy.up -> { } --> {controller_sa.u := 0.0; };

  lazy.down -> { } --> {controller_sa.d := 1.0; };
  not lazy.down -> { } --> {controller_sa.d := 0.0; };

  lazy.stop -> { } --> {controller_sa.u := 0.0 ; controller_sa.d := 0.0; };
}
```

Listing 6: Adaptation that generates controller_sa.

The adaptation controller_sa is shown in Listing 6. The control rules apply *regula falsi* to locate the crossing of the armature signal into the threshold T.

This example shows how the conditions in the output rules can be used to select which rules are applied. Informally, in general, at the end of each external state transition, when $MapOut$ is invoked, all the conditions in the rules are evaluated. The ones that evaluate to true, are recorded as part of the x_{out} state. Afterwards, whenever Out is called, only the rules that evaluated to true contribute to the output of Out.

The power_sa adaptation was omitted due to its simplicity. It declares the external FMU as a delayed Moore and lists the output port bindings.

The above adaptations generate the FMUs for the co-simulation scenario illustrated in Figure 5. The orchestrator in Algorithm 1 then computes the results shown in Figure 6. Comparing these results with the ones in Figure 4, one sees that they are similar, except for the fact that the armature current has a higher peak in the co-simulation. This is because the threshold crossing adaptation was disabled, since the power FMU does not support rollback.

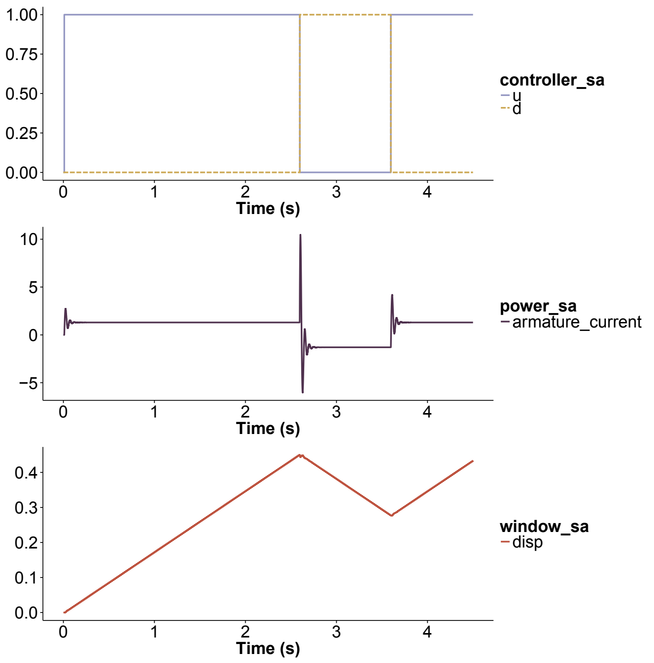


Figure 6. Power window co-simulation results.

In the following subsections, we describe the language (syntax and semantics) in more detail. The syntax is described using extended Backus–Naur form (EBNF)[48], and the semantics are presented informally by describing a transformation of baseSA descriptions, to the $Init$, In, $MapIn$, $Ctrl$, $MapOut$, and Out functions, introduced in Line 13.

Syntax

The partial syntax of baseSA is detailed in Listing 7. We omit the definition of the most common symbols:

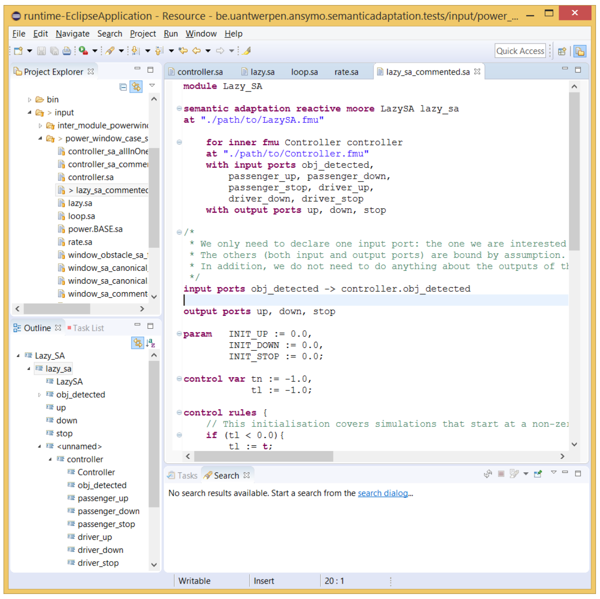
- ID is an identifier;
- URL is a URL;
- PhysicalUnit denotes any physical unit;
- Expression is an expression that defines a value, e.g., comparison, addition, constant, variable reference, etc.
- Statement is a programming language statement. It includes *if*-statement, static for loop, local variable declarations, assignments, references to variables/parameters, *built-in* function calls, etc.

```
SemanticAdaptation = 'semantic', 'adaptation', KindInput, KindOutput, UnitName,
        UnitInstance,
             'at', URL,
             InnerUnits,
             'input', 'ports', Port, {',', Port},
             'output', 'ports', Port, {',', Port},
             {ParamDeclarations}, [ControlRuleBlock], [InRulesBlock], [OutRulesBlock];
KindInput = 'reactive' | 'delayed';
KindOutput = 'moore' | 'mealy'
UnitName = ID;
UnitInstance = ID;
InnerUnits = {'for', InnerUnit} {'with', Connection};
InnerUnit = 'inner', 'fmu', UnitName, UnitInstance,
      'at', URL,
      'with', 'input', 'ports', Port, {',', Port},
      'with', 'output', 'ports', Port, {',', Port},
Port = ID, ['(', PhysicalUnit, ')'], [PortBinding];
PortBinding = ('->', ID) | ('<-', ID);
Connection = ID, '->', ID;
ParamDeclarations = 'param', SingleDeclaration, {',', SingleDeclaration};
SingleDeclaration = ID ':=' Expression;
ControlRuleBlock = {'control', VarDeclarations}, ControlRule;
VarDeclarations = 'var', SingleDeclaration, {',', SingleDeclaration};
ControlRule = 'control', 'rules', '{', {Statement}, '}';
InRulesBlock = {'in', VarDeclarations}, 'in' 'rules', '{', {DataRule}, '}';
DataRule = RuleCondition, "->", InRule, "-->", MapInRule, ';';
RuleCondition = BooleanExpression;
InRule = '{', {Statement}, '}';
MapInRule = '{', {Statement}, '}';
OutRulesBlock = RuleCondition, "->", MapOutRule, "-->", OutRule, ';';
```

Listing 7: The (partial) EBNF grammar of baseSA.

Table 1 summarizes the special functions and variables.

The full grammar definition, and an editor of baseSA descriptions, developed with Xtext[49], is available for download [*]. Figure 7 shows the editor interface.



```
module Lazy_SA

semantic adaptation reactive moore LazySA lazy_sa
at "./path/to/LazySA.fmu"

    for inner fmu Controller controller
    at "./path/to/Controller.fmu"
    with input ports obj_detected,
        passenger_up, passenger_down,
        passenger_stop, driver_up,
        driver_down, driver_stop
    with output ports up, down, stop

/*
 * We only need to declare one input port: the one we are interested
 * The others (both input and output ports) are bound by assumption.
 * In addition, we do not need to do anything about the outputs of tl
 */
input ports obj_detected -> controller.obj_detected

output ports up, down, stop

param   INIT_UP := 0.0,
        INIT_DOWN := 0.0,
        INIT_STOP := 0.0;

control var tn := -1.0,
            tl := -1.0;

control rules {
    // This initialisation covers simulations that start at a non-zeı
    if (tl < 0.0){
        tl := t;
```

No search results available. Start a search from the search dialog...

Figure 7. The baseSA editor.

Semantics

In this subsection, we define the semantics by describing informally how each syntactic construction in baseSA is mapped to the definition of $Init$, In, $MapIn$, $Ctrl$, $MapOut$, and Out functions, introduced in Section “Hierarchical Co-simulation for Semantic Adaptation” (recall Figure 1). This is done in two stages: first we detail how any baseSA description is reduced to its explicit form; and then we describe how each baseSA description in explicit form can be mapped to the semantic functions.

Reduction to Explicit Form. Let sa be the name of a given baseSA description. For the sake of brevity, we make the assumption that every port has a unique name (this is not assumed by the code generator).

In order to reduce the given baseSA description to its explicit form, the following rules are applied in order, with the description resulting from the application of one rule being used in the next rule.

AddInPorts – For each input port ip of any internal FMU f that has no incoming connections, create an external input port declaration ip -> f.ip, if there is none already declared with the same name.

AddInParams – For each external input port declaration ip, create a parameter declaration INIT_SA_IP := v (if it does not exist already), where v is the default value of the parameter.

AddInVars – For each declared external input port ip, declare an input variable stored_sa_ip := INIT_SA_IP (if it does not exist) with initial value equal to the corresponding declared parameter in the previous rule.

AddInRule – Prepend a new rule to the input rules block, with a true condition, and: in the *InRule* part, for each declared external input port ip,

[*]https://msdl.uantwerpen.be/git/claudio/HybridCosimulation

Table 1. List of *built-in* symbols and their meaning.

Symbol	Availability	Description
$t \in \mathbb{R}$	ControlRule ($Ctrl$)	Argument provided in the state transition function of the external FMU.
$H \in \mathbb{R}$	ControlRule ($Ctrl$)	Co-simulation step size passed as argument to the state transition function of the external FMU.
$do\_step(fmu, t_i, h) \in \mathbb{R}$	ControlRule ($Ctrl$)	Asks an internal FMU to perform a co-simulation step and returns the size of the computed interval.
$h \in \mathbb{R}$	MapInRule, MapOutRule ($MapIn, MapOut$)	Co-simulation step size passed as argument to the state transition function of the internal FMU.
$dt \in \mathbb{R}$	MapInRule, MapOutRule ($MapIn, MapOut$)	Let t_i denote the time given as argument to the state transition function of an internal FMU. Then $dt = t_i - t$.
$save\_state(fmu)$	ControlRule ($Ctrl$)	Stores the state of an internal FMU.
$rollback(fmu)$	ControlRule ($Ctrl$)	Rolls back an internal FMU to the last saved state.
$is\_close(x, y, rtol, atol) \in Bool$	Everywhere	Approximate equality.
$get\_next\_time\_step(fmu) \in \mathbb{R}$	Everywhere	Returns the maximum time step an internal FMU is willing to accept.
$\sin, \cos, \min, \ldots$	Everywhere	Implements the corresponding mathematical function.

add an assignment `stored_sa_ip := sa.ip`; in the *MapInRule* part, for each input binding declared `ip -> f.ip`, create an assignment `f.ip := ip`. If units need to be converted, the right hand side of the assignment is replaced accordingly.

RemoveInBindings – For each input binding declared `ip -> f.ip`, replace it by just `ip`. Any physical unit declaration is also removed.

AddOutPorts – If no output ports are declared, create an output port declaration `op <- f.op` per output port `op` of each internal unit `f`.

AddOutParams – Analogous to **AddInParams**: for each output port declaration `op` of each internal FMU `f`, create a parameter declaration `INIT_F_OP := v` (if such parameter does not exist), with `v` being the default value.

AddOutVars – Analogous to **AddInVars**: for each output port declaration `op` of each internal FMU `f`, create an output variable declaration `stored_f_op := INIT_F_OP`, if it does not exist already.

AddOutRule – Prepend a new output rule to the output rules block, with a `true` condition, and: in the *MapOutRule* part, add an assignment `stored_f_op := f.op`, per output port `op` of each internal unit `f`; in the *OutRule* part, for each output binding `op <- f.op` declared, add an assignment `sa.op := f.op`. If units need to be converted, the assignment is replaced accordingly.

RemoveOutBindings – Analogous to **RemoveInBindings**: for each declared output binding `op <- f.op`, remove the binding (and any unit declaration), leaving just the output declaration `op`.

CreateCtrlRules – If there is no control rules block declared, create one, and: compute the topological order σ of the internal scenario (if it cannot be computed, abort with an error); for each internal unit declaration `f`, in topological order, append `var` H_f `:= do_step(f, t, H)`; append (at the end of the block) either `return H_f` if there is only one internal FMU, or `return min(`H_1`, ...,` H_n`)`, where H_i refers to each of the local variables declared in the previous assignments.

ImplementInternalBinding – For each connection in the internal scenario `f1.op -> f2.ip`, locate the `do_step(f2, ...)` instruction in the control rules block. *Before* this instruction, if there is no assignment of the form `f2.ip := ...`, insert `f2.ip := f1.op` immediately before the instruction `do_step(f2, ...)`.

ReplacePortsRefsByVars – For every input rule, go through the *MapInRule* part and replace every reference to an external input port `ip`, by a reference to the `stored_sa_ip` input variable. In the control rules block, replace every reference to an output port `op` of an internal unit `f` by a reference to the corresponding storage variable `stored_f_op`. For each output rule, in the *OutRule* part, replace any reference to an output port `op` of an internal unit `f` by a reference to the corresponding storage variable `stored_f_op`.

Listing 2 is the result of applying the above rules to Listing 1.

Mapping to Generic Semantic Adaptation. Given a baseSA description in explicit form, we now explain how it is mapped to the formal definition of a generic external unit. In the generic external unit definition (recall Equation (5)), the elements that need to be defined are:

- The space of $\boldsymbol{x_{in}}$, $\boldsymbol{x_{ctrl}}$, and $\boldsymbol{x_{out}}$;
- $Init(\boldsymbol{u_{ext}})$ or $Init()$, depending on the kind of external unit;
- $In([\boldsymbol{x_{in}}, \boldsymbol{x_{ctrl}}, \boldsymbol{x_{out}}]^T, \boldsymbol{u_{ext}})$;
- $MapIn([\boldsymbol{x_{in}}, \boldsymbol{x_{ctrl}}, \boldsymbol{x_{out}}]^T, h, dt)$;
- $Ctrl(t, H, [\boldsymbol{x_{in}}, \boldsymbol{x_{ctrl}}, \boldsymbol{x_{out}}]^T, [\boldsymbol{x_1}, \ldots, \boldsymbol{x_n}]^T)$;
- $MapOut([\boldsymbol{x_{in}}, \boldsymbol{x_{ctrl}}, \boldsymbol{x_{out}}]^T, [\boldsymbol{y_1}, \ldots, \boldsymbol{y_n}]^T, h, dt)$;
- $Out([\boldsymbol{x_{in}}, \boldsymbol{x_{ctrl}}, \boldsymbol{x_{out}}]^T)$;

Each of the above elements are now defined.

Part of $\boldsymbol{x_{in}}$ is determined by the input variables declared: $\boldsymbol{x_{in}}$ has one dimension per declared input variable. The type of the dimension (real, boolean, etc...) corresponds to the type of the declared variable. In addition, $\boldsymbol{x_{in}}$ has one *boolean* dimension per input rule. For example, is there are three numeric variables declared, and one input rule, then $\boldsymbol{x_{in}} \in \mathbb{R}^3 \times Bool$.

Analogously to $\boldsymbol{x_{in}}$, $\boldsymbol{x_{out}}$ has one dimension per declared output variable, and an additional *boolean* dimension per declared output rule.

The control storage vector $\boldsymbol{x_{ctrl}}$ has one dimension per declared control variable. Additionally, if the semantic adaptation is a reactive one and the initial baseSA description (not the explicit one) does not include any control rules, the $\boldsymbol{x_{ctrl}}$ has one dimension per internal delayed unit.

The external input function

$$In([\boldsymbol{x_{in}}, \boldsymbol{x_{ctrl}}, \boldsymbol{x_{out}}]^T, \boldsymbol{u_{ext}}) = \boldsymbol{\tilde{x}_{in}}$$

is defined to perform the the following steps in order:

1. Evaluate all conditions of the input rules in the order that they are declared, and for each condition, mark the corresponding location of $\boldsymbol{\tilde{x}_{in}}$ with the outcome (true or false).
2. For the input rules whose conditions evaluated to true in the previous step, execute the *InRule* part, in the order that the rules are declared (this computes the remainder of $\boldsymbol{\tilde{x}_{in}}$).

Function

$$MapIn([\boldsymbol{x_{in}}, \boldsymbol{x_{ctrl}}, \boldsymbol{x_{out}}]^T, h, dt) = [\boldsymbol{\tilde{u}_1}, \ldots, \boldsymbol{\tilde{u}_n}]^T$$

executes the *MapInRule* part of the input rules whose condition evaluated to true (this information is stored in $\boldsymbol{x_{in}}$) in order of their declaration. The executed input port assignments form $[\boldsymbol{\tilde{u}_1}, \ldots, \boldsymbol{\tilde{u}_n}]^T$.

Function

$$MapOut([\boldsymbol{x_{in}}, \boldsymbol{x_{ctrl}}, \boldsymbol{x_{out}}]^T, [\boldsymbol{y_1}, \ldots, \boldsymbol{y_n}]^T, h, dt) = \boldsymbol{\tilde{x}_{out}}$$

is analogous to In. It evaluates all the conditions of the output rules in the order that they are declared, and for each of those conditions, marks the appropriate location of $\boldsymbol{\tilde{x}_{out}}$ with the outcome of the condition evaluation. Then it computes the remaining portion of $\boldsymbol{\tilde{x}_{out}}$ by executing the *MapOutRule* part of each of the output rules whose conditions evaluated to true.

Function

$$Out([\boldsymbol{x_{in}}, \boldsymbol{x_{ctrl}}, \boldsymbol{x_{out}}]^T) = \boldsymbol{y}$$

is analogous to $MapIn$. It executes the *OutRule* part of the output rules whose condition evaluated to true (in the order in which they are declared) to compute the output vector $\boldsymbol{y}$.

The role of the initialization function (derived automatically from the baseSA description) is to find a consistent initial state, defining the initial values of the storage vectors $\boldsymbol{x_{in}}$, $\boldsymbol{x_{out}}$, and $\boldsymbol{x_{ctrl}}$. If the semantic adaptation is declared as reactive, then $Init$ requires the initial input, according to Equation (1).

First, the parts of $\boldsymbol{x_{in}}$, $\boldsymbol{x_{ctrl}}$, and $\boldsymbol{x_{out}}$ that correspond to the declared input/control/output variables are initialized according to the initial value that is declared for them.

If it exists, the part of $\boldsymbol{x_{ctrl}}$ that corresponds to the previous inputs to the internal units is initialized by compting the initial input to all the internal units in the topological order (such order exists by assumption). This is similar to Algorithm 2, except that the functions In, $MapIn$, and $MapOut$, are invoked to adapt any external input to the internal units, and initialize the condition flags.

Function

$$Ctrl(t, H, [\boldsymbol{x_{in}}, \boldsymbol{x_{ctrl}}, \boldsymbol{x_{out}}]^T, [\boldsymbol{x_1}, \ldots, \boldsymbol{x_n}]^T) = \left\langle \boldsymbol{\tilde{x}_{ctrl}}, \boldsymbol{\tilde{x}_{out}}, [\boldsymbol{\tilde{x}_1}, \ldots, \boldsymbol{\tilde{x}_n}]^T, \tilde{H} \right\rangle$$

runs the instructions declared in the control rules block, in the order that they are declared. The assignments performed to control variables make up part of the output vector $\boldsymbol{\tilde{x}_{ctrl}}$. The executed assignments to the input ports of each internal FMU i, up to the instruction $do\_step(i, t_i, h_i)$, make up part of the unit input vector $\boldsymbol{u_i}$.

Any variable reference in the control rules block refers to the most recently given value of that variable.

Each instruction $do\_step(i, t_i, h_i)$ maps to the following steps, performed in $Ctrl$:

- Invoke $MapIn$ function to compute the external input of unit i:

$$[\ldots, \boldsymbol{\tilde{u}_i}, \ldots]^T := MapIn([\boldsymbol{x_{in}}, \boldsymbol{\tilde{x}_{ctrl}}, \boldsymbol{x_{out}}]^T, h_i, t_i - t) \quad (8)$$

Note that $\tilde{\boldsymbol{x}}_{\boldsymbol{ctrl}}$ represents the control state vector that was affected by the assignments made since the beginning of the execution of the $Ctrl$ function. $\boldsymbol{x_{in}}$ and $\boldsymbol{x_{out}}$ represent the (unchanged) vector provided as input to $Ctrl$.

- Merge the input vector $\boldsymbol{u_i}$ computed by previous assignments with $\tilde{\boldsymbol{u}}_{\boldsymbol{i}}$ to form the unit input $\boldsymbol{uc_i}$;
- Invoke the state transition function of the unit:

$$\left\langle \tilde{\boldsymbol{x}}_{\boldsymbol{i}}, \tilde{H}_i \right\rangle := F_i(t, H, \boldsymbol{x_i}, \boldsymbol{uc_i} \text{ or } \boldsymbol{up_i}) \tag{9}$$

- Get the output of the unit:

$$\boldsymbol{y_i} := G_i(t + \tilde{H}_i, \tilde{\boldsymbol{x}}_{\boldsymbol{i}}, \boldsymbol{uc_i}) \text{ or } G_{\sigma(j)}(t + \tilde{H}_i, \tilde{\boldsymbol{x}}_{\boldsymbol{i}}) \tag{10}$$

- Invoke the $MapOut$ function to compute an updated output storage vector:

$$\tilde{\boldsymbol{x}}_{\boldsymbol{out}} := MapOut([\boldsymbol{x_{in}}, \tilde{\boldsymbol{x}}_{\boldsymbol{ctrl}}, \boldsymbol{x_{out}}]^T, [\boldsymbol{y_1}, \ldots, \boldsymbol{y_n}]^T, h_i, t_i - t) \tag{11}$$

Finally, upon returning, if the external FMU is a reactive unit, and the initial baseSA description does not declare a control rules block, $Ctrl$ stores the most recent inputs provided to each delayed internal units in the $\tilde{\boldsymbol{x}}_{\boldsymbol{ctrl}}$ vector, to be used as delayed inputs in a subsequent external state transition call. This instruction is similar to Line 29 of Algorithm 7.

Evaluation

In this section, we judge how well our approach answers the research question posed in this work.

The requirements set by the research question are:

Productivity – Does the language have impact in the productivity of its users?

Expressivity – Is the language expressive enough to cover *current* and *future* needs?

Modularity – Does the internal FMUs need to be changed?

Transparency – Does the external FMU behave exactly as an FMU?

Productivity

In general, DSLs have the potential to boost its users' productivity[42,50]. For baseSA, we describe an early experiment to assess the productivity.

Goals. Productivity is measured by comparing the time it takes for a *trained user* to: (1) create an external FMU using our DSL; and (2) code the same external FMU.

As a surrogate measure, we compare the approximate number of lines of code (LOC) required for a semantic adaptation coded by hand, with the LOC of the corresponding semantic adaptation expressed in baseSA.

Experimental Setup. As part of the development of the code generator, all semantic adaptations identified in Figure 5, except the rate_sa, were coded by hand and the effort taken was recorded.

Results. Table 2 shows the adaptation, the approximated lines of code (LOC), and the effort in coding the semantic adaptations in C.

Table 2. Effort in hand-coding hierarchical semantic adaptations.

Semantic Adaptation	LOC	Effort (man-hour)
lazy_sa	700	9
controller_sa	750	24
power_sa	680	16
window_sa	690	8
loop_sa	690	16
Total	3510	73

As Table 2 shows, even though the semantic adaptations differ in complexity, they have a similar number of LOC. This is evidence that there is a large portion of code dedicated to common FMI-related management tasks. With baseSA, the user does not have to code:

- Memory management – The inputs, outputs, and local variables, of the external FMU are stored in dynamically allocated memory.
- Variable de-referencing – To set/get values to/from an internal FMU, a list of value references (integers which identify a variable) has to be provided. Any mistake here may cause the internal FMU to give wrong results, but not necessarily crash, which makes it hard to debug.
- State management – The external FMU has to support rollback, and for that, the state variables must be properly serialized and de-serialized. In the case study, each semantic adaptation requires approximately 140 LOC to implement the set/get state.
- Consistent inputs management – The external FMU which is reactive and has internal delayed units, has to keep track of the previous inputs to these.

Threats to Validity. LOC is only a surrogate measure for the productivity of a DSL, albeit a common one[51], and depends on the programmer. However, the tasks described in the above list are handled automatically by the code generator of baseSA.

The values provided in Table 2 lack external validation. We intend to perform a second round of experiments, where we will ask a participant to code a semantic adaptation, then train him/her, and measure the effort it takes to code the same adaptation, in baseSA.

Expressivity

The baseSA DSL is imperative in the sense that it describes *how* the semantic adaptations are performed. However, it forces a structure in the definition of the semantic adaptations, aided by the distinction between data (input/output rules) and control adaptations. We argue that this structure does not restrict the expressiveness of the semantic adaptations.

To provide evidence for this, we describe how the adaptations used in the case study are representative of the semantic adaptations and coupling algorithms surveyed in[17].

Extrapolation/interpolation schemes These techniques, used in[10,27,33,34,52–54], are similar to the rate_sa.

Jacobi-based orchestration This orchestration algorithm, used in[11,47,55–60], is similar to the Gauss-seidel coupling except that it assumes that all units are delayed. A way to implement it as a semantic adaptation is to define a control rule that sets explicitly the inputs to the internal FMUs, and then invokes the do_step function on them.

Algebraic constraint couplings This coupling technique, reported in[24,25,30,61], can be implemented by a fixed point iteration (recall adaptation loop_sa) and extra algebraic computations on the units inputs and outputs.

Semi-implicit coupling These techniques, presented in[25,62–65], are similar to the ones above, except they perform two iterations only.

Error control Richard extrapolation[9,47,66] can be implemented by creating a semantic adaptation which runs a whole scenario at twice the rate of the original one; Multi-order input extrapolation[59,67] amounts to implementing two approximation schemes (see item above) and run in parallel; Embedded method[68] requires that a semantic adaptation is implemented to perform a discretized numerical integration of some of the signals in the internal scenario; Energy based[69] techniques can be implementing by coding semantic adaptations which monitor for energy dissipativity in some of the signals in the internal FMUs.

We do restrict the expressiveness of the language, with the intent of guaranteing that it terminates:

- No function definitions are allowed;
- No recursive definitions of semantic adaptations are allowed;
- For loops must have a static range.

These restrictions make expressing some of the above techniques more cumbersome, but not impossible.

Modularity

The simulation unit specification, introduced in Equation (1) was shown to be a valid abstraction of an implementation of an FMU in Section "Background". Furthermore, it is clear that changing the implementation of any of the functions $Init, G, F$ implies a change in the FMU implementation. In Section "Semantics", these functions are invoked as part of the implementation of each semantic adaptation, but never changed, thus showing that the corresponding FMU implementations are not affected by the implementation of the language.

Transparency

Section "Hierarchical Co-simulation for Semantic Adaptation" describes how a generic semantic adaptation forms a simulation unit that obeys the definition in Equation (1) (see Equation (5)). Furthermore, Section "Semantics" describes how a baseSA is implemented by "filling in" the semantic adaptation functions, that are used in Section "Hierarchical Co-simulation for Semantic Adaptation". The semantics does not require the hierarchical unit definition, in Equation (5), to be changed. Therefore, our approach does not violate transparency.

Discussion and Future Work

This section discusses some of the characteristics and limitations of our contribution, and research opportunities for the future.

Automatic Semantic Adaptation Identification. Throughout this work, we assumed that the user knows that an adaptation is required in order to make the co-simulation possible. An interesting research direction is to explore what means can be employed in trying to identify the need for specific semantic adaptations.

Runtime Performance. Despite not being our primary goal, the performance of the generated FMU should be similar to a custom coded one. To this end, the code generator under development performs most tasks at compile time. However, we have not carried out any experiments to measure the performance of the generated code.

A research direction is to explore how to merge multiple adaptations, to avoid generating the intermediate hierarchical FMUs. For example, in Figure 5, adaptations loop_sa and rate_sa could be merged into one single adaptation, provided that the user has no intention of using loop_sa for other purposes. However, while it is clear what the result should be for this example, in general this is non-trivial task: *When can two arbitrary adaptations be merged?*

Solving this problem brings a performance benefit, but also provides new insights into the nature of adaptations. In addition, one can ask: *If two semantic adaptations can be combined, are they commutative?* This question is important because it allows us to optimize: if there is a semantic adaptations which will cause rollbacks, we want it to be the first to execute, to avoid wasting computation. An example of this is the controller_sa, which performs the crossing location before the *lazy_sa* gets the opportunity to run.

Trying to answer the above questions will inevitably lead to another question: *what is the right level of abstraction to analyse the combination of semantic adaptations?* This question is related to the next discussion topic.

Usability and Productivity. As part of trying to find out what the right level of abstraction to analyze semantic adaptations is, we are developing a new language that allows for a more declarative description of semantic adaptations. This language, as opposed to baseSA, allows for a much more concise description of the most common semantic adaptations by just enumerating *what* semantic adaptations should be used to form the external FMU.

The descriptions made in this language compile to baseSA, whose role is to provide a solid foundation.

The main benefits of using this language are:

1. The user does not to know how semantic adaptations are implemented.
2. It is minimal, meaning that it enables the user to specify common semantic adaptations (e.g., multi-rate, successive substitution) as concisely as describing them in natural language;
3. It further restricts the user into using well known semantic adaptations, which prevents mistakes.
4. It may provide insight into the research questions identified in the previous subsection.

```
1 import PowerWindowModel
2
3 semantic adaptation reactive moore RateLoopSA rate_loop
4 at "./path/to/RateLoopSA.fmu"
5 for fmu WindowSA windowSA, Obstacle obstacle
6 successive substitution starts at height with absolute tolerance = 1e-8 and
      relative tolerance = 0.0001
7 multiply rate 10 times with first order interpolation
```

Listing 8: Example description in higher level semantic adaptation DSL.

Listing 8 shows an example of what such DSL looks like. The syntax reuses part of the syntax of baseSA. The description of the FMUs can be done in a separate module, which is then imported (Line 1). For this example, the FMUs are described as in Lines 4–15 of Listing 3. After the preliminaries, the description of each semantic adaptation occupies one line (Line 6 for loop_sa, and Line 7 for rate_sa). In this language, adaptations are applied in order, meaning that the outer most adaptation is the multi-rate one.

Each adaptation has some degree of configuration. For example, the multi-rate is configurable with an input approximation adaptation. This highlights another interesting research direction, related to the combination of semantic adaptations: *how and when can semantic adaptations interface with each other?* In this example, it is clear that any input approximation adaptations can complement a multi-rate adaptation, but what are the essential characteristics of input approximation and multi-rate adaptations, that make them so compatible? The same question applies to output approximation adaptations (the family of Hold adaptations) and the lazy related ones. A possible direction to explore is to look at the object oriented world, and study how can semantic adaptations define interfaces and specialization, so that their interaction is well defined.

Discrete Event FMU Implementation. The current version of the FMI standard (version 2.0) lacks essential features to enable accurate hybrid co-simulation (see, e.g.,[70–74]).

Until new extensions are made, there are many different ways in which a cyber system (e.g., a state chart) can be simulated in an FMU[7,73–77]. At least one of the implementations the authors used before (the Stategraph[7]), already includes semantic adaptations, to facilitate its integration with the FMI.

Our work shows that, when implementing an FMU that simulates a cyber system, it is best to leave as many semantic adaptations as possible out. The more adaptations an FMU already contains, the harder it is to adapt it to other contexts.

Related Work

Outside the context of FMI, the problem of composing and adapting operational semantics of multiple languages is discussed in[14,78–84] and references therein.

Within the context of FMI, we can divide the related works in two categories: (A) those whose prime purpose is to describe co-simulation scenarios; and (B) those that target the description of orchestration algorithms. Both these categories do not target primarily the description of semantic adaptations, but can potentially be extended to include simple descriptions. Due to our pure hierarchical co-simulation approach, our contribution complements any of these works.

Under Category (A), we highlight[16],[46],[47], and[85]. These works introduce a language for the description of a co-simulation scenario, with the purpose of running a co-simulation. The work in[46,47] assumes that a generic orchestration algorithm is used, whereas[16,85] aim at generating an orchestrator that is specific to the scenario described. Our DSL allows for the description of a co-simulation scenario, and a specific master algorithm can be generated from that description.

DACCOSIM[47] follows a related approach with respect to hierarchical co-simulation, allowing the scenario to grouped by computational nodes. In contrast to our work, this hierarchy is computational and not functional. Moreover, it is not transparent, as the distinction is made between local (internal to computational nodes) and global orchestrators. Nevertheless, each FMU is wrapped with code that performs error control, highlighting the need for semantic adaptation.

In Category (B), we highlight[85], [22], and[86].

The work in[85] allows the description of master algorithms using the Business Process Modelling Notation. We argue that the visual notation for the description of an orchestration algorithm works well for simple cases, with two units. However, when multiple semantic adaptations become necessary, or the number of simulation units increases, the visual notation rapidly becomes cluttered. The work does not describe any intention of using the notation to describe semantic adaptations, but the notation has an extension mechanism that can in principle be used to describe simple semantic adaptations.

The most related to our own is[86]. It introduces an object oriented framework for co-simulation that allows for both the development of FMUs, as well as for orchestration algorithms, in C++. Class specialization is used extensively to maximize reuse, sharing some of the benefits with our contribution. The main difference to our work is the level of abstraction and the intention to use semantic adaptations. While their work is capable of expressing semantic adaptations, our work is targeted towards that purpose. One can position their work as helping develop FMUs for simulators that need to support the FMI Standard, and our work can be used to adapt already existing FMUs. Furthermore, the description of a complex adaptations such as rate_sa is more compact in our DSL.

Conclusion

This paper addressed the problem of describing the most common semantic adaptations on multiple types of black box simulation units in a productive manner while avoiding the modification of the units (modularity) and tools for co-simulation (transparency).

To make this possible, we propose a DSL, available for download[*], that is both expressive (due to its imperative nature) but also productive (due to its conventions and high level constructs). Each description refers to a group of interconnected FMUs and dictates how those FMUs interact with the environment.

The essential mechanism that enables the semantic adaptations is the concept of hierarchical co-simulation, formalized in this work. The meaning of each adaptation is given by mapping it onto hierarchical co-simulation units, which in turn is mapped to units and FMUs, as illustrated in Figure 1.

The main distinguishing factor from the related work, is our focus in semantic adaptations for FMI based co-simulation, which imposes the modularity and transparency requirements.

This work opens up new opportunities for research into semantic adaptations, for example, how to find higher levels of abstraction to describe semantic adaptations, and explore how different semantic adaptations can interface and complement each other. We intend to explore these in the future.

Funding

This research was partially supported by Flanders Make vzw, the strategic research centre for the manufacturing industry, a PhD fellowship grant from the Agency for Innovation by Science and Technology in Flanders (IWT, dossier number 151067), and by the INTO-CPS project, funded by the European Commission's Horizon 2020 programme (grant number 664047).

Acknowledgements

We are grateful to Prof. Peter Gorm Larsen (Aarhus University) for the in-depth discussions regarding the potential applications of our work. Furthermore, we are in debt to the reviewers for the in-depth improvements made on this manuscript.

Author Biographies

Cláudio Gomes is a PhD student in the University of Antwerp and member of the Modelling, Simulation and Design Lab (MSDL), interested in co-simulation of hybrid systems.

Bart Meyers is a research engineer at Flanders Make vzw, and is associated with the University of Antwerp, where he obtained his PhD. His main research topic is modelling of software-intensive systems.

Joachim Denil is an Assistant Professor in the University of Antwerp and member of the Cosys-lab. His main research interest is multi-paradigm modelling of software-intensive systems.

Casper Thule is a PhD student at Aarhus University and member of the Software Engineering Research Group. His area of research is hybrid co-simulation and cyber-physical systems.

Kenneth Lausdahl is a PostDoc at Aarhus University and member of the Software Engineering group, interested in tool automation and co-simulation of hybrid systems.

[*]`https://msdl.uantwerpen.be/git/claudio/HybridCosimulation`

Hans Vangheluwe is a Professor at the University of Antwerp (Belgium), an Adjunct Professor at McGill University (Canada) and an Adjunct Professor at the National University of Defense Technology (NUDT) in Changsha, China. He heads the MSDL research lab. His research interest is the multi-paradigm modelling of complex, software-intensive, cyber-physical systems.

Paul De Meulenaere is a Professor at the University of Antwerp and member of the research group CoSys-Lab. His main research field is model-based development methods for embedded real-time systems.

References

1. Nielsen CB, Larsen PG, Fitzgerald J et al. Systems of Systems Engineering: Basic Concepts, Model-Based Techniques, and Research Directions. *ACM Computing Surveys* 2015; 48(2): 18:1—-18:41. DOI:10.1145/2794381.
2. Van der Auweraer H, Anthonis J, De Bruyne S et al. Virtual engineering at work: the challenges for designing mechatronic products. *Engineering with Computers* 2013; 29(3): 389–408. DOI:10.1007/s00366-012-0286-6.
3. Vangheluwe H, De Lara J and Mosterman PJ. An introduction to multi-paradigm modelling and simulation. In *AI, Simulation and Planning in High Autonomy Systems*. SCS, pp. 9–20.
4. Blockwitz T, Otter M, Akesson J et al. 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, pp. 173–184. DOI:10.3384/ecp12076173.
5. Lee EA and Sangiovanni-Vincentelli A. A framework for comparing models of computation. *Transactions on Computer-Aided Design of Integrated Circuits and Systems* 1998; 17(12): 1217–1229. DOI:10.1109/43.736561.
6. Tripakis S. Bridging the semantic gap between heterogeneous modeling formalisms and FMI. In *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. Samos, Greece: IEEE. ISBN 978-1-4673-7311-1, pp. 60–69. DOI:10.1109/SAMOS.2015.7363660. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7363660`.
7. Otter M, Malmheden M, Elmqvist H et al. A New Formalism for Modeling of Reactive and Hybrid Systems. In *7th International Modelica Conference*. Como, Italy: Linköping University Electronic Press; Linköpings universitet, pp. 364–377. DOI:10.3384/ecp09430108.
8. Gomes C. Foundations for Continuous Time Hierarchical Co-simulation. In *ACM Student Research Competition (ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems)*. Saint Malo, France: ACM New York, NY, USA, p. to appear.
9. Arnold M, Clauß C and Schierz T. Error Analysis and Error Estimates for Co-simulation in FMI for Model Exchange and Co-Simulation v2.0. In Schöps S, Bartel A, Günther M et al. (eds.) *Progress in Differential-Algebraic Equations*. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN 978-3-662-44926-4, pp. 107–125. DOI:10.1007/978-3-662-44926-4_6.
10. Busch M. Continuous approximation techniques for co-simulation methods: Analysis of numerical stability and local error. *ZAMM - Journal of Applied Mathematics and Mechanics* 2016; 96(9): 1061–1081. DOI:10.1002/zamm.201500196.
11. Bastian J, Clauß C, Wolf S et al. Master for Co-Simulation Using FMI. In *8th International Modelica Conference*. Dresden, Germany: Linköping University Electronic Press, Linköpings universitet, pp. 115–120. DOI:10.3384/ecp11063115.
12. Gomes C, Karalis P, Navarro-López EM et al. 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. ISBN 978-3-319-74781-1, pp. 345–360. DOI:10.1007/978-3-319-74781-1_24. URL `http://link.springer.com/10.1007/978-3-319-74781-1{_}24`.
13. Gomes C, Legat B, Jungers RM et al. Stable Adaptive Co-simulation : A Switched Systems Approach. In *IUTAM Symposium on Co-Simulation and Solver Coupling*. 1, Darmstadt, Germany, p. to appear.
14. Meyers B, Denil J, Boulanger F et al. A DSL for Explicit Semantic Adaptation. In Moreira A, Schätz B, Gray J et al. (eds.) *7th International Workshop on Multi-Paradigm Modeling*. Number 1112 in CEUR Workshop Proceedings, Miami, United States: Springer, Berlin, Heidelberg, pp. 47–56.
15. Denil J, Meyers B, De Meulenaere P et al. Explicit Semantic Adaptation of Hybrid Formalisms for FMI Co-Simulation. In Fernando Barros, Wang MH, Prähofer H et al. (eds.) *Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*. Alexandria, Virginia: Society for Computer Simulation International San Diego, CA, USA, pp. 99–106.
16. Van Acker B, Denil J, Meulenaere PD et al. Generation of an Optimised Master Algorithm for FMI Co-simulation. In Barros F, Wang MH, Prähofer H et al. (eds.) *Symposium on Theory of Modeling & Simulation-DEVS Integrative*. Alexandria, Virginia, USA: Society for Computer Simulation International San Diego, CA, USA, pp. 946–953.
17. Gomes C, Thule C, Broman D et al. Co-simulation: State of the art. Technical report, 2017. URL `http://arxiv.org/abs/1702.00686`. `1702.00686`.
18. 1730-2010 - IEEE Recommended Practice for Distributed Simulation Engineering and Execution Process (DSEEP). *IEEE Std 1730-2010 (Revision of IEEE Std 15163-2003)* 2011; : 1–79DOI:10.1109/IEEESTD.2011.5706287.

19. Kübler R and Schiehlen W. Two Methods of Simulator Coupling. *Mathematical and Computer Modelling of Dynamical Systems* 2000; 6(2): 93–113. DOI:10.1076/1387-3954(200006)6:2;1-M;FT093.
20. Posse E, de Lara J and Vangheluwe H. Processing causal block diagrams with graphgrammars in atom3. In *Workshop on Applied Graph Transformation (AGT)*. Grenoble, France: Springer, Berlin, Heidelberg, pp. 23–34.
21. Gomes C, Denil J and Vangheluwe H. Causal-Block Diagrams. Technical report, University of Antwerp, 2016. URL `http://msdl.cs.mcgill.ca/people/claudio/pub/Gomes2016a.pdf`.
22. Gheorghe L, Bouchhima F, Nicolescu G et al. A Formalization of Global Simulation Models for Continuous/Discrete Systems. In *Summer Computer Simulation Conference*. SCSC '07, San Diego, CA, USA: Society for Computer Simulation International San Diego, CA, USA. ISBN 1-56555-316-0, pp. 559–566.
23. Cellier FE and Kofman E. *Continuous System Simulation*. Springer Science & Business Media, 2006. ISBN 9780387261027.
24. Arnold M. Stability of Sequential Modular Time Integration Methods for Coupled Multibody System Models. *Journal of Computational and Nonlinear Dynamics* 2010; 5(3): 9. DOI:10.1115/1.4001389.
25. Schweizer B, Lu D and Li P. Co-simulation method for solver coupling with algebraic constraints incorporating relaxation techniques. *Multibody System Dynamics* 2016; 36(1): 1–36. DOI:10.1007/s11044-015-9464-9.
26. Andersson C. *Methods and Tools for Co-Simulation of Dynamic Systems with the Functional Mock-up Interface*. PhD Thesis, Lund University, 2016.
27. Burden RL and Faires JD. *Numerical Analysis*. 9 ed. Cengage Learning, 2010. ISBN 0538733519.
28. Lelarasmee E, Ruehli AE and Sangiovanni-Vincentelli AL. The Waveform Relaxation Method for Time-Domain Analysis of Large Scale Integrated Circuits. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 1. ISBN 0278-00701, pp. 131–145. DOI:10.1109/TCAD.1982.1270004.
29. González F, Naya MÁ, Luaces A et al. On the effect of multirate co-simulation techniques in the efficiency and accuracy of multibody system dynamics. *Multibody System Dynamics* 2011; 25(4): 461–483. DOI:10.1007/s11044-010-9234-7.
30. Sicklinger S, Belsky V, Engelmann B et al. Interface Jacobian-based Co-Simulation. *International Journal for Numerical Methods in Engineering* 2014; 98(6): 418–444. DOI:10.1002/nme.4637.
31. Zhang F, Yeddanapudi M and Mosterman PJ. Zero-Crossing Location and Detection Algorithms For Hybrid System Simulation. In *IFAC Proceedings Volumes*, volume 41. Seoul, Korea: Elsevier Ltd, pp. 7967–7972. DOI:10.3182/20080706-5-KR-1001.01346. URL `http://linkinghub.elsevier.com/retrieve/pii/S1474667016402296`.
32. Mosterman PJ. An Overview of Hybrid Simulation Phenomena and Their Support by Simulation Packages. In Vaandrager FW and van Schuppen JH (eds.) *Hybrid Systems: Computation and Control SE - 17, Lecture Notes in Computer Science*, volume 1569. Berg en Dal, The Netherlands: Springer Berlin Heidelberg. ISBN 978-3-540-65734-7, pp. 165–177. DOI:10.1007/3-540-48983-5_17.
33. Dronka S and Rauh J. Co-simulation-interface for user-force-elements. In *SIMPACK user meeting*. Baden-Baden, Germany.
34. Busch M. *Zur effizienten Kopplung von Simulationsprogrammen*. PhD Thesis, Kassel university, 2012.
35. Andersson C, Führer C and Åkesson J. Efficient Predictor for Co-Simulation with Multistep Sub-System Solvers. Technical Report 1, 2016. URL `http://lup.lub.lu.se/record/dbaf9c49-b118-4ff9-af2e-e1e3102e5c22`.
36. Kofman E and Junco S. Quantized-state systems: a DEVS Approach for continuous system simulation. *Transactions of The Society for Modeling and Simulation International* 2001; 18(3): 123–132.
37. Bolduc JS and Vangheluwe H. Expressing ODE models as DEVS: Quantization approaches. In Barros F and Giambiasi N (eds.) *AI, Simulation and Planning in High Autonomy Systems*. Lisbon, Portugal: IEEE, pp. 163–169.
38. Awais MU, Palensky P, Elsheikh A et al. The high level architecture RTI as a master to the functional mock-up interface components. In *International Conference on Computing, Networking and Communications*. San Diego, USA: IEEE. ISBN 978-1-4673-5288-8, pp. 315–320. DOI:10.1109/ICCNC.2013.6504102.
39. Bolduc JS and Vangheluwe H. Mapping ODES to DEVS: Adaptive quantization. In *Summer Computer Simulation Conference*. Montreal, Quebec, Canada: Society for Computer Simulation International. ISBN 0094-7474, pp. 401–407.
40. Camus B, Galtier V, Caujolle M et al. Hybrid Co-simulation of FMUs using DEV&DESS in MECSYCO. In *Symposium on Theory of Modeling & Simulation - DEVS Integrative M&S Symposium (TMS/DEVS 16)*. Pasadena, CA, United States: Society for Computer Simulation International San Diego, CA, USA, p. No. 8.
41. Quesnel G, Duboz R, Versmisse D et al. DEVS coupling of spatial and ordinary differential equations: VLE framework. In *Open International Conference on Modeling and Simulation*, volume 5. Citeseer, pp. 281–294.
42. Kelly S and Tolvanen JP. *Domain-specific modeling: enabling full code generation*. John Wiley & Sons, 2008. ISBN 0470249250.
43. Prabhu SM and Mosterman PJ. Modeling, Simulating, and Validating a Power Window System Using a Model-Based Design Approach. URL `https://fr.mathworks.com/company/newsletters/articles/modeling-simulating-and-validating-a-power-window-system-using-a-model-based-`

design-approach.html.

44. Denil J. *Design, Verification and Deployment of Software Intensive Systems - A multiparadigm approach*. PhD Thesis, University of Antwerp, 2013.
45. Fritzson P, Aronsson P, Pop A et al. OpenModelica - A free open-source environment for system modeling, simulation, and teaching. In *Conference on Computer Aided Control System Design, International Conference on Control Applications, International Symposium on Intelligent Control*. Munich, Germany: IEEE, pp. 1588–1595. DOI:10.1109/CACSD-CCA-ISIC.2006.4776878. URL http://ieeexplore.ieee.org/document/4776878/.
46. Larsen PG, Fitzgerald J, Woodcock J et al. Integrated tool chain for model-based design of Cyber-Physical Systems: The INTO-CPS project. In *2nd International Workshop on Modelling, Analysis, and Control of Complex CPS (CPS Data)*. Vienna, Austria: IEEE. ISBN 978-1-5090-1154-4, pp. 1–6. DOI:10.1109/CPSData.2016.7496424.
47. Galtier V, Vialle S, Dad C et al. FMI-Based Distributed Multi-Simulation with DACCOSIM. In *Spring Simulation Multi-Conference*. Alexandria, Virginia, USA: Society for Computer Simulation International San Diego, CA, USA. ISBN 978-1-5108-0105-9, pp. 804–811.
48. Wirth N. Extended backus-naur form (EBNF), 1996.
49. Xtext - Language Engineering for Everyone. URL https://eclipse.org/Xtext/index.html.
50. Kieburtz RB, McKinney L, Bell JM et al. A software engineering experiment in software component generation. In *18th international conference on Software engineering*. Berlin, Germany: IEEE Computer Society, pp. 542–552.
51. Boehm BW, Abts C, Brown AW et al. *Software cost estimation with Cocomo II*. Prentice Hall, 2000. ISBN 0130266922.
52. Ben Khaled A, Duval L, Gaïd MEMB et al. Context-based polynomial extrapolation and slackened synchronization for fast multi-core simulation using FMI. In *10th International Modelica Conference*. Lund, Sweden: Linköping University Electronic Press, pp. 225–234.
53. Stettinger G, Horn M, Benedikt M et al. Model-based coupling approach for non-iterative real-time co-simulation. In *European Control Conference (ECC)*. Strasbourg, France: IEEE, pp. 2084–2089. DOI:10.1109/ECC.2014.6862242.
54. Brembeck J, Pfeiffer A, Fleps-Dezasse M et al. Nonlinear State Estimation with an Extended FMI 2.0 Co-Simulation Interface. In *10th International Modelica Conference*. Lund, Sweden: Linköping University Electronic Press; Linköpings universitet, pp. 53–62. DOI:10.3384/ecp1409653.
55. Friedrich M. *Parallel Co-Simulation for Mechatronic Systems*. PhD Thesis, Fakultät für Maschinenwesen, 2011.
56. Krammer M, Fritz J and Karner M. Model-Based Configuration of Automotive Co-Simulation Scenarios. In *48th Annual Simulation Symposium*. Alexandria, Virginia: Society for Computer Simulation International San Diego, CA, USA. ISBN 978-1-5108-0099-1, pp. 155–162.
57. Enge-Rosenblatt O, Clauß C, Schneider A et al. Functional Digital Mock-up and the Functional Mock-up Interface–Two Complementary Approaches for a Comprehensive Investigation of Heterogeneous Systems. In *8th International Modelica Conference*. Dresden, Germany: Linköping University Electronic Press; Linköpings universitet, pp. 748–755.
58. Gu B and Asada HH. Co-simulation of algebraically coupled dynamic subsystems. In *American Control Conference*, volume 3. Arlington, VA, USA: IEEE. ISBN 0743-1619 VO - 3, pp. 2273–2278. DOI:10.1109/ACC.2001.946089.
59. Busch M and Schweizer B. An explicit approach for controlling the macro-step size of co-simulation methods. In *7th European Nonlinear Dynamics*. Rome, Italy: European Mechanics Society. ISBN 978-88-906234-2-4, pp. 24–29.
60. Wetter M. Co-simulation of building energy and control systems with the Building Controls Virtual Test Bed. *Journal of Building Performance Simulation* 2010; 4(3): 185–203. DOI:10.1080/19401493.2010.518631.
61. Gu B and Asada HH. Co-Simulation of Algebraically Coupled Dynamic Subsystems Without Disclosure of Proprietary Subsystem Models. *Journal of Dynamic Systems, Measurement, and Control* 2004; 126(1): 1. DOI:10.1115/1.1648307.
62. Schweizer B and Lu D. Semi-implicit co-simulation approach for solver coupling. *Archive of Applied Mechanics* 2014; 84(12): 1739–1769. DOI:10.1007/s00419-014-0883-5.
63. Schweizer B, Li P, Lu D et al. Stabilized implicit co-simulation methods: solver coupling based on constitutive laws. *Archive of Applied Mechanics* 2015; 85(11): 1559–1594. DOI:10.1007/s00419-015-0999-2.
64. Schweizer and Lu D. Predictor/corrector co-simulation approaches for solver coupling with algebraic constraints. *ZAMM - Journal of Applied Mathematics and Mechanics / Zeitschrift für Angewandte Mathematik und Mechanik* 2015; 95(9): 911–938. DOI:10.1002/zamm.201300191.
65. Schweizer B and Lu D. Stabilized index-2 co-simulation approach for solver coupling with algebraic constraints. *Multibody System Dynamics* 2015; 34(2): 129–161. DOI:10.1007/s11044-014-9422-y.
66. Arnold M, Hante S and Köbis MA. Error analysis for co-simulation with force-displacement coupling. *PAMM* 2014; 14(1): 43–44. DOI:10.1002/pamm.201410014.
67. Busch M and Schweizer B. Coupled simulation of multibody and finite element systems: an efficient and robust semi-implicit coupling approach. *Archive of Applied Mechanics* 2012; 82(6): 723–741. DOI:10.1007/s00419-011-0586-0.

68. Hoepfer M. *Towards a Comprehensive Framework for Co- Simulation of Dynamic Models With an Emphasis on Time Stepping*. PhD Thesis, Georgia Institute of Technology, 2011.
69. Sadjina S, Kyllingstad LT, Skjong S et al. Energy conservation and power bonds in co-simulations: non-iterative adaptive step size control and error estimation. *Engineering with Computers* 2017; 33(3): 607–620. DOI:10.1007/s00366-016-0492-8.
70. Broman D, Brooks C, Greenberg L et al. Determinate composition of FMUs for co-simulation. In *Eleventh ACM International Conference on Embedded Software*. Montreal, Quebec, Canada: IEEE Press Piscataway, NJ, USA. ISBN 978-1-4799-1443-2, p. Article No. 2.
71. Broman D, Greenberg L, Lee EA et al. Requirements for Hybrid Cosimulation Standards. In *18th International Conference on Hybrid Systems: Computation and Control*. HSCC '15, Seattle, Washington: ACM New York, NY, USA. ISBN 978-1-4503-3433-4, pp. 179–188. DOI:10.1145/2728606.2728629.
72. Centomo S, Deantoni J and de Simone R. Using SystemC Cyber Models in an FMI Co-Simulation Environment: Results and Proposed FMI Enhancements. In *Euromicro Conference on Digital System Design (DSD)*. Limassol, Cyprus: IEEE. ISBN 978-1-5090-2817-7, pp. 318–325. DOI:10.1109/DSD.2016.86. URL `http://ieeexplore.ieee.org/document/7723569/`.
73. Cremona F, Lohstroh M, Broman D et al. Step Revision in Hybrid Co-simulation with FMI. In *14th ACM-IEEE International Conference on formal Methods and Models for System Design*. Kanpur, India: IEEE.
74. Feldman YA, Greenberg L and Palachi E. Simulating Rhapsody SysML Blocks in Hybrid Models with FMI. In *10th International Modelica Conference*. Lund, Sweden: Linköping University Electronic Press, pp. 43–52. DOI: 10.3384/ecp1409643.
75. Tripakis S, Broman D and Sciences C. Bridging the Semantic Gap Between Heterogeneous Modeling Formalisms and FMI. Technical report, 2014.
76. Pohlmann U, Schäfer W, Reddehase H et al. Generating Functional Mockup Units from Software Specifications. In *9th International MODELICA Conference*. 078, Munich, Germany: Linköping University Electronic Press; Linköpings universitet, pp. 765–774. DOI:10.3384/ecp12076765. URL `http://www.ep.liu.se/ecp/article.asp?issue=076{%}26article=78`.
77. Cremona F, Lohstroh M, Tripakis S et al. FIDE: an FMI integrated development environment. In *31st Annual ACM Symposium on Applied Computing*. SAC '16, Pisa, Italy: ACM New York, NY, USA. ISBN 9781450337397, pp. 1759–1766. DOI:10.1145/2851613.2851677.
78. Lacoste-Julien S, Vangheluwe H, de Lara J et al. Meta-modelling hybrid formalisms. In *IEEE International Symposium on Computer Aided Control Systems Design*. New Orleans, LA, USA: IEEE. ISBN VO -, pp. 65–70. DOI:10.1109/CACSD.2004.1393852.
79. Davis II J, Goel M, Hylands C et al. Overview of the Ptolemy project. Technical report, 1999. URL `http://ptolemy.eecs.berkeley.edu/`.
80. Vara Larsen ME, De Antoni J, Combemale B et al. A Behavioral Coordination Operator Language (BCOoL). In *18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. Ottawa, ON, Canada: IEEE. ISBN 978-1-4673-6908-4, pp. 186–195. DOI:10.1109/MODELS.2015.7338249.
81. Deantoni J. Modeling the Behavioral Semantics of Heterogeneous Languages and their Coordination. In *Architecture-Centric Virtual Integration (ACVI)*. Venice, Italy: IEEE. ISBN 978-1-5090-2488-9, pp. 12–18. DOI: 10.1109/ACVI.2016.9. URL `http://ieeexplore.ieee.org/document/7510564/`.
82. Mustafiz S, Gomes C, Barroca B et al. 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*. DEVS '16, Pasadena, California: IEEE, pp. 29:1—-29:8. DOI:10.23919/TMS.2016.7918835.
83. Boulanger F, Hardebolle C, Jacquet C et al. Semantic Adaptation for Models of Computation. In *11th International Conference on Application of Concurrency to System Design (ACSD)*. Newcastle Upon Tyne, UK: IEEE. ISBN 1550-4808 VO -, pp. 153–162. DOI:10.1109/ACSD.2011.17.
84. Boulanger F and Hardebolle C. Simulation of Multi-Formalism Models with ModHel'X. In *1st International Conference on Software Testing, Verification, and Validation*. Lillehammer, Norway: IEEE Computer Society. ISBN VO -, pp. 318–327. DOI:10.1109/ICST.2008.15.
85. Campagna D, Kavka C, Turco A et al. Solving time-dependent coupled systems through FMI co-simulation and BPMN process orchestration. In *IEEE International Symposium on Systems Engineering (ISSE)*. Edinburgh, Scotland: IEEE. ISBN 978-1-5090-0793-6, pp. 1–8. DOI:10.1109/SysEng.2016.7753140. URL `http://ieeexplore.ieee.org/document/7753140/`.
86. Aslan M, Durak U and Taylan K. MOKA: An Object-Oriented Framework for FMI Co-Simulation. In *Conference on Summer Computer Simulation*. Chicago, Illinois: Society for Computer Simulation International San Diego, CA, USA, pp. 1–8.