The FMI 3.0 Standard Interface for Clocked and Scheduled Simulations

Cláudio Gomes[1] Masoud Najafi[2] Torsten Sommer[3] Matthias Blesken[4] Irina Zacharias[4]
Oliver Kotte[5] Pierre R. Mai[6] Klaus Schuch[7] Karl Wernersson[8] Christian Bertsch[5]
Torsten Blochwitz[9] Andreas Junghanns[10]

[1]Department of Electrical and Computer Engineering, Aarhus University, Denmark, claudio.gomes@ece.au.dk
[2]Altair, France, masoud@altair.com
[3]Dassault Systemes GmbH, Germany, Torsten.SOMMER@3ds.com
[4]dSPACE GmbH, Germany, {MBlesken,izacharias}@dspace.de
[5]Corporate Research, Robert Bosch GmbH, Renningen, Germany,
{Oliver.Kotte,Christian.Bertsch}@de.bosch.com
[6]PMSFIT, Germany, pmai@pmsfit.de
[7]AVL, Austria, klaus.schuch@avl.com
[8]Dassault Systemes AB, Sweden, karl.wernersson@3ds.com
[9]ESI ITI, Germany, Torsten.Blochwitz@esi-group.com
[10]Synopsys, Germany, Andreas.Junghanns@synopsys.com

Abstract

This paper gives an overview of the FMI 3.0 support for two kinds of clock-based simulations: Synchronous Clocked simulation, and Scheduled Execution. The former aims at scenarios where the information about multiple simultaneous events, as well as their causality and exact time of occurrence, can be unambiguously conveyed. The later facilitates real-time simulation comprising multiple black-box models, by allowing a finer grained control (compared to version 2.0 of the FMI Standard) over the computation time of sub-models. A formalization is presented, along with some example application scenarios, that is meant as an introduction to the clocks and their conceptualization in the FMI Standard.

Keywords: functional mockup interface, synchronous clocks, reactive systems, real-time simulation, scheduling, real-time operating system.

1 Introduction

As more and more Modeling and Simulation (M&S) tools are used in system engineering processes, it becomes clear that standards are needed to improve the interoperability of such tools. The Functional Mockup Interface (FMI) Standard (2.0 2014) aims at enabling the exchange and cooperative simulation of black-box models. Version 2.0 of the standard strikes a balance between supporting the most common features across the plethora of M&S tools, and enabling advanced simulation scenarios. Its wide adoption has, however, placed pressure in supporting even more advanced use cases. Two important use cases are: simulation scenarios where timed and state events play a frequent role in synchronizing a subset of the participating models (e.g., controller code with tasks running at different rates); and scenarios where the goal is to control the computation time of the different models, so that a real-time co-simulation can be achieved.

Contribution. This paper gives an overview of the FMI 3.0 support for two kinds of clock based simulations: Synchronous Clocked simulation (SC), and Scheduled Execution (SE). The former aims at scenarios where the information about multiple simultaneous events, as well as their causality and exact time of occurrence, can be unambiguously conveyed. The later facilitates real-time simulation comprising multiple black-box models, by allowing a finer grained control (compared to version 2.0) over when and which model partitions can be executed.

Structure. The next section will introduce the common concepts, and the interface elements that are common to SC and SE. Then in Section 3, SC is detailed, along with a motivating example. Section 4 then focuses on SE, following the same structure as Section 3. In Section 5, we discuss some of the relevant related works, and in Section 6 we summarize and conclude.

2 Common Interface and Concepts

Co-simulation is a technique to combine multiple black-box simulation units to compute the combined models' behaviour (e.g. see Kübler and Schiehlen (2000) and Gomes et al. (2018), for an introduction). The simulation units, often developed and exported independently from each other in different M&S tools, are coupled using an orchestration algorithm, often developed independently as well, that communicates with each simulation unit via its interface. This interface, an example of which is the FMI Standard interface for Co-Simulation, comprises functions for setting/getting inputs/outputs and computing the asso-

ciated model behaviour over a given time interval.

The FMI 3.0 defines three interface types: the Co-Simulation (CS), the Model Exchange (ME), and the Scheduled Execution (SE). In the FMI nomenclature, a simulation unit is called the Functional Mockup Unit (FMU), and it may implement one or more of the three interfaces. An FMU is a zip file containing: binaries and/or source code implementing the API functions; miscellaneous resources; and an XML file, describing the variables, model structure, and other data.

For each interface type, the FMU may implement optional features, such as declaring synchronous clocks (in case of ME or CS), or scheduled execution clocks (in case of SE). Figure 1 summarizes the different interface types and the main concepts relevant to this paper. All three interfaces (CS, ME, and SE) share common functionality, such as the declaration and usage of variables and clocks.

The differences between the three interface types can be seen on the left hand side of Figure 1. The Importer refers to the software that imports the FMU. We distinguish three importers, each corresponding to one of the interface types, and each with different responsibilities. The ME importer often needs to provide a differential equation (ODE) solver, and must handle events. In contrast, the CS importer does not need to provide an ODE solver, because such a solver can be implemented inside the CS FMU. Finally, the SE importer needs a task scheduler that will determine exactly when each task implemented in the FMU will be executed.

The ME and CS both contain mechanisms to communicate events to the importer, and, as we shall see later, both enable different possible clock interpretations: e.g. Synchronous Clocked (SC) simulation. Another clock interpretation is given to the clocks that are used in the context of the SE implementation.

Broadly, a simulation involving multiple connected FMUs, goes through the following modes[1]:

Initialize – The FMUs are instantiated and their initial state/inputs/outputs/parameters are calculated or set by the importer.

Step – The simulation is progressing in simulated time, and FMUs that represent ODEs are being numerically integrated.

Event – The simulated time is stopped and events (e.g. clock ticks, parameter changes) are being processed.

Terminate – The simulation has finished and all resources are freed.

The Step and Event modes come after the Initialize mode, and are interleaved.

In the following sub-sections, we introduce FMI3.0 clocks (how they are declared, connected, and interacted with), as well as common constraints imposed by the standard. These are common to the SC and SE clock interpretations.

[1]This is a simplification of the states or modes defined in the state diagrams of the FMI 3.0 standard.

2.1 Clock Taxonomy

Clocks represent an abstraction of activities whose occurrence is tied to specific points in time. They appear in many modeling formalisms for systems that interact with the real world (Benveniste et al. 2003; Modelica Association 2021), where it is important to represent computations that happen at different rates, or as a result of conditions observed in the environment. Conceptually, each clock represents a sequence of instants in time where the clock is active, called ticks. From the entities that can interact with a clock, we highlight the FMU and the importer (recall Figure 1). The FMU is the entity that declares the clock, while the importer is the code activating the clock in the FMU.

Clocks are declared in the XML file, and can be seen as a special kind of variable. The XML description for each clock contains, among others, an identifier called the value reference, a causality attribute (whether the clock is an input or output, as we will discuss later), and an interval attribute (declaring the type of clock, discussed later). Dynamically, during the simulation, each clock can be either active or inactive (denoted as the clock's state), and its state can be either set or get by the importer, depending on the clock type and its causality (see below).

There are two main types of clocks: time-based and triggered. Time-based clocks are associated with an interval, dictating, at any moment in simulated time, the interval (in simulated time units) between the last tick and the next tick. Such intervals can be queried or set by the importer, depending on the clock's interval attribute (see below). In contrast, triggered clocks have no a priori known interval. The FMU or importer has to set/get the (activation) state of the clock. The different clock types are listed in Table 1 according to who calculates the intervals and ticks the clock.

Before discussing the causality of clocks, it is important to distinguish between the entity that dictates the clock interval vs. the entity that actually activates the clock. This distinction is important in the context of the FMI Standard because the simulated time is a real-valued quantity, represented by a finite-resolution variable. For example, the FMU may declare the interval of a periodic clock in the XML, but it is the importer that will decide exactly at which simulated time the clock ticks. Due to numerical inaccuracies, it may happen that the interval (in simulated time) between clock ticks does not match exactly the interval declared by the FMU.

Time-based clocks are always input clocks, since it is always the importer that is responsible for activating the clock (even though the clock interval information may come from other entities, as shown in Table 1). Triggered clocks, on the other hand, can be input or output clocks. Triggered input clocks, just like time-based input clocks, can only be set by the importer, whereas triggered output clocks are set internally by the FMU, and can only be queried by the importer. The causality therefore plays a

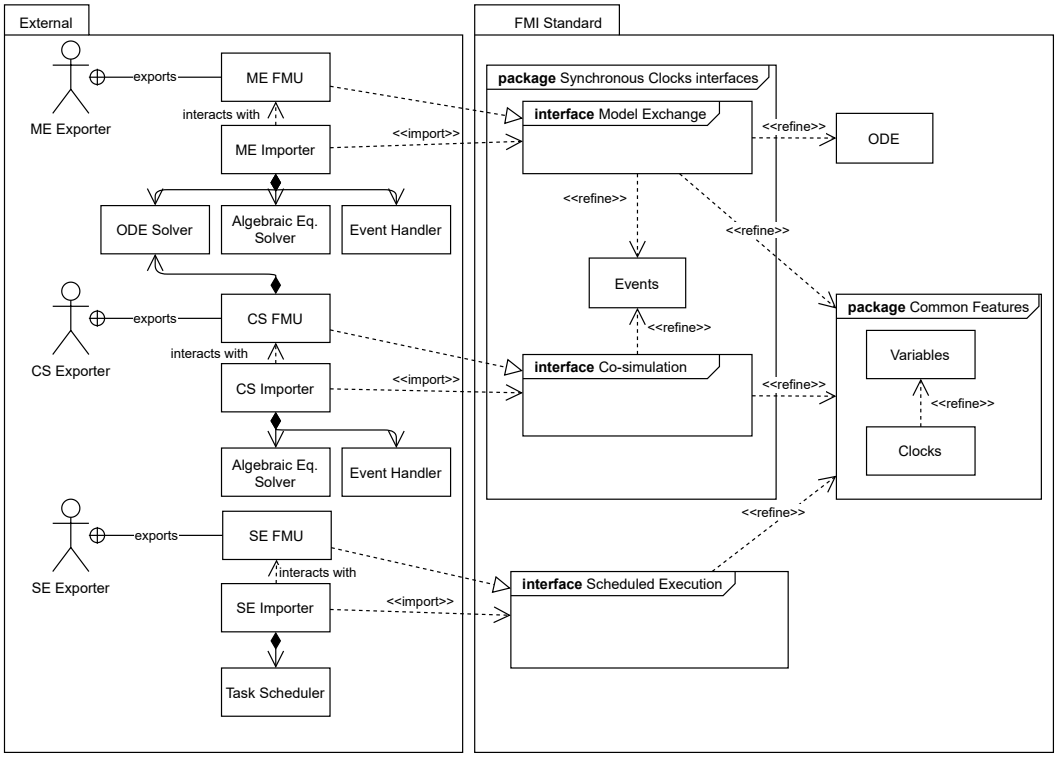

Figure 1. Overview of relevant concepts. Note that there might be domain specific importers which do not need an ODE solver because the supported FMUs do not contain continuous variables. This figure attempts to illustrate the most common differences between the interface types.

role in determining how clocks can be connected.

2.2 Clock Variables, Connections, and Dependencies

Just like any other variable, an output clock can be connected to an input clock. It is also possible to connect two input clocks or even have one input clock connected to two different output clocks. A connection from clock w^c to clock v^c means that whenever clock w^c ticks, then clock v^c should also tick. For triggered clocks, that is relatively easy to enforce: whenever one clock activates, the other should be activated. For time-based clocks, the importer must take into account the interval attributes of the clocks and decide whether such connection makes sense or not. For example, if one clock has a constant interval, and another clock has a fixed interval, then the importer may simply set the correct period for the second clock.

FMUs can declare the internal dependencies between their output and their input variables in the XML section denoted as Model Structure. An output variable y depends on an input variable u when the computation of y's requires the value of u. For example, in Figure 2, y_m is computed from, among other dependencies, u_m.

Each output clock y^c can also depend on one or more input clocks or variables. The meaning is that the state of such input clocks or the value of the input variables is taken into account when deciding whether y^c will tick. For example, in Figure 2, y^c_m *may* tick when u^c_m ticks, or because of the value of u_m. Note that it is not necessarily the case that y^c will tick whenever an input clock, that y^c depends on, ticks.

When a clock w^c ticks (we use w^c when the causality of the clock is irrelevant), there is a set of variables whose values are computed. We denote that set by "w^c's variables", or "clocked variables" when the specific clock is unimportant. FMI imposes few constraints on the clocked variables. However, the FMU can declare in its XML, for each variable, which clocks w^c it depends on (usually one). For example, in Figure 2, y_m is computed when u^c_m ticks. Lacking such declaration, the importer needs to assume the worst case: all output variables are computed when w^c ticks. The value of w^c's variables should only be accessed when (one of the) w^c is active, i.e., is ticking. Accessing the w^c's variables when w^c is not ticking results in undefined behaviour.

3 Synchronous Clocks

In this section, we describe the Synchronous Clock (SC) interpretation of the clocks interface, introduced in the previous section. This interpretation is inspired by the clock implementation in the Modelica specification (Modelica Association 2021) and existing synchronous clock theories (e.g. Benveniste et al. (2003)), but had to be adapted to reflect the constraints of black-box co-simulation. As such, we offer no guarantees of semantic equivalence.

We start with detailing the main simulation modes for both ME and CS FMUs, as if no clocks were declared. In

Table 1. Overview of clock types and their attributes.

Clock Type	Period	Interval	Interpretation
time-based	periodic	constant	FMU declares period in XML.
		fixed	Importer sets the interval during Initialize.
		calculated	FMU calculates period in Initialize mode.
		tunable	FMU calculates period in Event mode (CS) or after executing model partition (SE).
	aperiodic	changing	FMU calculates interval after each clock tick.
		countdown	FMU calculates interval after an event.
triggered	–	triggered	There's no known interval. The clock ticks unpredictably, either due to FMU current state/inputs, or due to events.

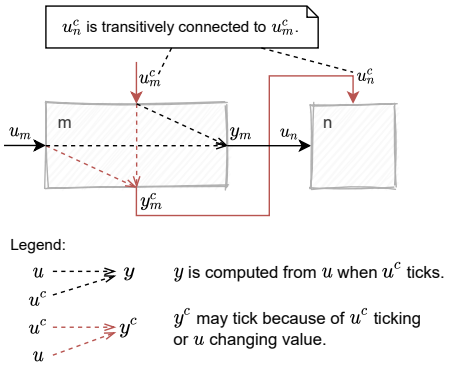

Figure 2. Example clock connections and dependencies. The symbols *m* and *n* refer to FMUs.

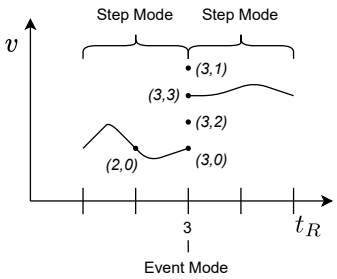

Figure 3. Example variable trajectory under super-dense time.

order to focus on the essential mechanisms, we abstract away from the ME and CS interfaces, and present them in a unified manner using set theoretic constructs, while referring the reader to the standard for more details.

3.1 Background on FMI CS and ME Simulations

Following the super-dense time formulation (e.g. Lee and Zheng (2005)), the simulation time is a tuple $t = (t_R, t_I)$ where $t_R \in \mathbb{R}_{\geq 0}$, $t_I \in \mathbb{N}_{\geq 0}$. In Step mode, the real part of time t_R is increasing and $t_I = 0$, and during Event mode, the integer part of time t_I is increasing while t_R is held constant. Figure 3 illustrates a possible trajectory for the values of a variable v under super-dense time. As can be seen, the Step mode produces a continuous evolution for the value of v, while the Event mode introduces discontinuities in the calculation of v. Under Event mode, a variable may acquire multiple values, each computed by one iteration of the Event mode, discerned by the t_I part of the timestamp.

In Step mode, the FMU and importer cooperate in approximating the solution of a system of differential equations, described by the FMU. In the case of ME, the FMU provides the derivatives and the importer provides the inputs and solver, whereas in CS, the importer provides the inputs, and the FMU provides the derivatives and solver (recall Figure 1).

The importer may then switch the FMU to Event mode if one or more of the following situations occur[2]:

Time events – the simulated time $t = (t_R, 0)$ achieved a particular value t_R that was known at the end of the most recent Event mode;

State events – Some value of a variable crossed a threshold that is known to the FMU;

Input events – Some value of an input variable changed in a discrete way, introducing a discontinuity.

In version 3 of the FMI Standard, both ME and CS interfaces describe the mechanism by which the FMU communicates the occurrence of events to the importer, so we will not discuss these mechanisms here. It suffices to assert that the importer is able to determine that the FMU should switch to Event mode at the appropriate simulated time.

During Event mode, the FMU and importer cooperate in solving a set of algebraic equations that are associated with the event that triggered the Event mode, known to the FMU. To solve the equations, the importer will typically construct a dependency graph between the output and input variables (using the Model Structure declared by the FMU). Note that the FMU may be part of a larger sim-

[2]There are other kinds of events, but for simplicity we highlight the main ones.

ulation model, where external variables form its inputs, and can also depend on its outputs. Therefore the dependency graph may involve not just the FMU variables, but other relevant external variables. As a result, there might exist cyclic dependencies between variables of the FMU (these manifest in the form of non-trivial strongly connected components in the dependency graph (Tarjan 1971)). It is up to the importer to solve the algebraic loop, by setting and querying the variables of the FMU. The FMU plays its role by recomputing any output variable that might change as a result of new values for the input variables set by the importer. The important outcome is that all variables of the FMU have acquired a value.

The FMU may remain in Event mode, and perform a new iteration, if more events occur. These new events may be caused by the importer or by a new value for some variable. The FMI defines the mechanism by which the FMU or importer agree that a new event iteration is needed. Each new event iteration corresponds to one increment in the integer part of the simulated time. If no more event iterations are needed, Event mode is finished.

Note that, in Event mode, as part of the procedure to solve non-linear equations, there may be hundreds of iterations to converge and obtain a solution. These intermediate values are not shown in Figure 3 and do not cause the integer part of super-dense time to increment because they happen within one super-dense time instant. Therefore, in Figure 3 there are three event handling iterations. When switching back to Step mode, the FMU also informs the importer of the next time-based event (if such event is defined).

3.2 Discerning Events

The basic event signaling mechanism offered by FMI 2.0 is adequate for most applications that do not rely on too many events. However, they are insufficiently expressive for simulations with many events. We illustrate this with a simple example, devised to motivate the need for clocks and shown in Figure 4. The example shows a closed loop control system, where the `CtrlFMU` is specified as an FMU, and the remaining sub-models are specified in some other language. We sketch the `CtrlFMU` equations, but note that the importer has no access to these (it can only query the FMU for the values of the output variables). The `CtrlFMU`, every $1/r$ seconds (with r denote both a clock r and its frequency), gets a sample from the `Plant` (produced by the `Sensor`), and calculates its next state, based on the previous state `pre(u_r)`, the sampled value `x_r`, and some configuration parameter `a` that is calculated by the `Supervisor`. The latter, depending on the `Plant` dynamics, the sampling rate of which we ignore, may decide to reconfigure the `Controller`.

Using only the basic event mechanism of FMI, it is cumbersome to simulate Figure 4, for the following reasons:

- If it is the FMU that decides when to sample, there is no way for the importer to know the sample rate r. The importer only receives information about the next time event, after each Event mode of the `CtrlFMU`.
- There is no way for the FMU to know exactly which equations to enable when entering Event mode. When the `Supervisor` computes a new value for `a_s`, the `CtrlFMU` must be in Event mode, because of the input event. Then `CtrlFMU` must rely on approximate floating point comparisons to know that only the `Config` equation is to be enabled. Conversely, when a new sample `x_r` is available, the `CtrlFMU` must know that the `Config` equation must remain disabled.

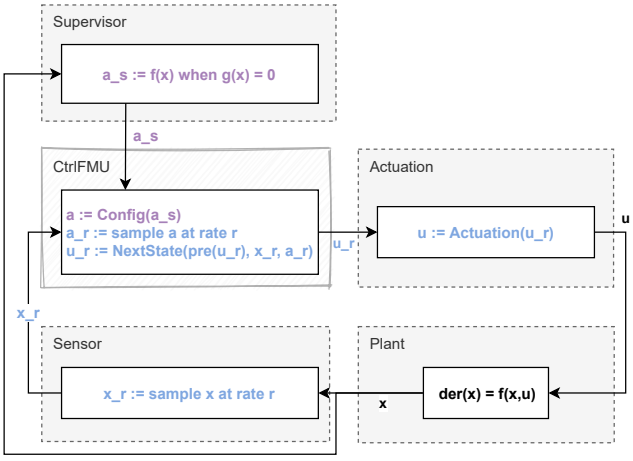

Figure 4. Motivating example with supervisor controller.

Figure 5 shows how clocks address the limitations highlighted by the example in Figure 4. By introducing a triggered input clock `s` and a time-based input clock `r`, it is made clear who is responsible for the unambiguous activation of the clocks: the `Supervisor` controls `s`, and the importer controls `r`. Furthermore, no approximate floating point comparisons are needed to know which equations have to be active when entering Event mode.

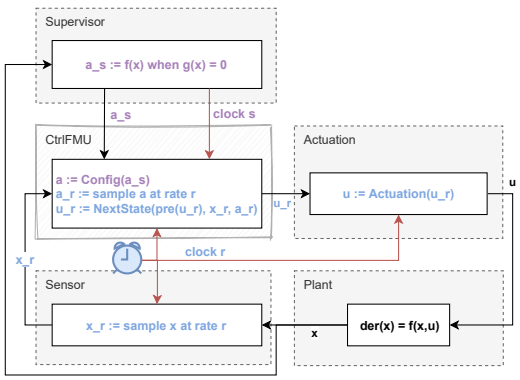

Figure 5. Clocked version of Figure 4.

3.3 Synchronous Clocks Semantics

We now detail the main functions that interact with clocks. In order to do so, we must define a compact FMU abstraction. Without loss of generality, we can focus on formalizing what happens in a simulation where the FMU is in Step mode, switches to Event mode, and then resumes Step mode.

Definition 1 (SC FMU Instance)**.** An SC FMU instance with identifier m is represented by the tuple

$$\langle S_m, U_m, Y_m, U_m^c, Y_m^c, \\ \texttt{set}_m, \texttt{get}_m, \texttt{set}_m^c, \texttt{get}_m^c, \texttt{stepT}_m, \texttt{stepE}_m, \texttt{nextT}_m \rangle$$

where:

- S_m represents the abstract set of possible FMU states. A given state $s_m \in S_m$ of m represents the complete internal state of m. This includes: active clocks, active equations, current mode (Step or Event mode) current valuations for input and output variables, etc.
- U_m and Y_m represent the set of input and output variables, respectively.
- U_m^c and Y_m^c represent the set of input and output clocks, respectively.
- $\texttt{set}_m : S_m \times U_m \times \mathscr{V} \to S_m$ and $\texttt{get}_m : S_m \times Y_m \to S_m \times \mathscr{V}$ are functions to set the inputs and get the outputs, respectively (we abstract the set of values that each input/output variable can take as $\mathscr{V}$). Both $\texttt{set}_m$ and $\texttt{get}_m$ return a new state because both can trigger the computation of equations.
- $\texttt{set}_m^c : S_m \times U_m^c \times \mathbb{B} \to S_m$ and $\texttt{get}_m^c : S_m \times Y_m^c \to S_m \times \mathbb{B}$ are the functions that (de-)activate the input clocks and query the output clocks (returning the activation status), respectively, and $\mathbb{B}$ denotes the boolean set.
- $\texttt{stepT}_m : S_m \times \mathbb{R}_{\geq 0} \to S_m \times \mathbb{R}_{\geq 0} \times \mathbb{B}$ is a function representing the Step mode computation. If m is in state s_m at simulated time (t_R, t_I), $(s_m{}', h, b) = \texttt{stepT}_m(s_m, H)$ approximates the state $s_m{}'$ of m at time $(t_R + h, 0)$, with $h \leq H$. When $b = \mathit{true}$, we know that the importer and m have agreed to interrupt the Step mode prematurely, and m is ready to go into Event mode.
- $\texttt{stepE}_m : S_m \to S_m \times \mathbb{B}$ represents one super-dense time iteration of the Event mode. If m is in state s_m at time (t_R, t_I), then $(s_m{}', b) = \texttt{stepE}_m(s_m)$ represents the computation of m's internal super-dense step transition, where $s_m{}'$ represents the state at $(t_R, t_I + 1)$ and b informs the importer whether one more Event iteration is needed.
- $\texttt{nextT}_m : S_m \times U_m^c \to \mathbb{R}_{\geq 0} \cup \{\mathit{NaN}\}$ is the function that allows the importer to query the time of the next clock tick. This function is only applicable to tunable, changing, and countdown clocks, and the returned value is calculated according to the clock type as discussed in Table 1. The value *NaN* can be returned for countdown clocks, and it means that the clock currently has no schedule.

The major differences between above formalization and the FMI interface are as follows.

- There is no explicit representation of state. Most FMI functions take an FMU instance as an argument, and the manipulations to the instance are performed implicitly. We choose to make state explicit so as to explicitly convey which functions change the state of the instance.
- The FMI describes the callback function by which, in CS, the FMU and importer may decide when to prematurely terminate the invocation to $\texttt{stepT}_m$. For ME, the importer is responsible for implementing $\texttt{stepT}_m$ (recall Figure 1).

We now discuss informally the semantics of each function implemented in an FMU instance m, with a focus on the clock functions. Since clock operations happen only in Event mode, we will focus on that mode. Moreover, we present the semantics in the order that a generic importer would interact with the FMI instance m.

Entering Event Mode. Clocks can only tick in Event mode. During Step mode, the FMI provides mechanisms for the FMU and importer to agree that there's a clock that needs to tick, and will therefore switch the FMU to Event mode at the appropriate time. Such mechanisms are represented by $(s_m{}', h, b) = \texttt{stepT}_m(s_m, H)$, when $b = \mathit{true}$. In this case, $s_m{}'$ represents the state of the FMU ready to begin the Event mode, at super-dense time $(t_R + h, 0)$, where t_R is the real part of the time of s_m. As discussed in Section 3.1, the causes of $b = \mathit{true}$ can be many (e.g., a clock will tick).

Ticking Clocks. In Event mode, m in state $s_m \in S_m$ may activate any triggered output clock $y_m^c \in Y_m^c$, a fact that can be communicated to the importer via the function call $\texttt{get}_m^c(s_m, y_m^c)$. Conversely, any input clock $u_m^c \in U_m^c$ that needs to be ticked (according to the interval information), is activated by the importer, through the function call $\texttt{set}_m^c(s_m, u_m^c, \mathit{true})$.

Enabling/Disabling Clock Equations. Let $s_m \in S_m$ denote the state of m right after a clock w_m^c (input or output) has been activated, and let (t_R, t_I) represent the current super-dense time. When w_m^c is activated, there is a set of equations, associated to w_m^c, that becomes active (are enabled) for the current super-dense time instant (t_R, t_I). The set of output variables whose value is computed by w_m^c's equations is denoted as "w_m^c's variables". While w_m^c is active, invocations to $\texttt{get}_m$ on w_m^c's variables will trigger their computation according to w_m^c's equations. However, the values that w_m^c's variables acquire while w_m^c is active are only made permanent when $\texttt{stepE}_m$ is invoked, at which point w_m^c becomes inactive and the super-dense time instant becomes $(t_R, t_I + 1)$. If $\texttt{set}_m^c$ is invoked to deactivate an active clock w_m^c at time (t_R, t_I) (before $\texttt{stepE}_m$ is invoked), then m should ensure that w_m^c's variables return to the values they had immediately before w_m^c became active (this is not a strict requirement, since those variables should not be consulted once w_m^c became inactive).

Propagating Clock Activations. In Event mode, if a clock w^c_m is (in-)active at super-dense time (t_R, t_I), then the importer must ensure that all other clocks that are connected to w^c_m must also be (in-)active for time (t_R, t_I). Triggered output clocks may become active during a super-dense time instant (e.g., because a variable acquired a particular value), or after a call to $\texttt{stepE}_m$. Therefore, the importer must query triggered output clocks to monitor activations. The FMI eases this task by allowing the variables that may influence a triggered output clock to be declared in the XML (recall Figure 2). It is outside the scope of FMI to ensure that a simulation scenario is well defined (e.g., it does not get stuck in an infinite number of (de-)activations).

Scheduling Time-Based Clocks. In Event mode, after a call to $\texttt{stepE}_m$, at super-dense time (t_R, t_I), m must be able to inform the importer of the time of the next tick of each clock $u^c_m \in U^c_m$ that is tunable, changing, or countdown. This is done through function $\texttt{nextT}_m$, when u^c_m satisfies one of the following conditions: 1. u^c_m is a countdown clock; or 2. u^c_m is not a countdown clock, and u^c_m was active in the super-dense time that was just concluded, at time $(t_R, t_I - 1)$. The Importer should use this information to schedule the next Event mode. If $\texttt{nextT}_m$ returns 0, then the importer must do a new Event iteration.

Generic Clocked Simulation Algorithm. The following summarizes the Event Mode algorithm that coordinates the simulation with multiple FMU instances, with connected inputs/outputs and clocks. Let M denote the set of FMU instances participating in the simulation. We assume that one FMU instance $m \in M$ or the importer has requested to enter Event mode. Therefore we assume that every other instance $m' \in M \wedge m' \neq m$ has been stepped up to the same super-dense time $(t_R, 0)$. In the following, we use _ to denote an non-important argument of a function.

1. Every $m \in M$ enters Event mode (super-dense time instant is $t_I = 0$);
2. Activate any time-based clocks scheduled to tick at $(t_R, 0)$, by invoking $\texttt{set}^c_{m'}(\_, w^c_{m'})$ for any input or output clock $w^c_{m'} \in U^c_{m'} \cup Y^c_{m'}$ and any instance $m' \in M$;
3. Construct and solve system of equations for t_I:
   (a) For all $y^c_m \in Y^c_m$ of any instance $m \in M$, forward activation state of triggered clocks:
      i. Invoke $\texttt{get}^c_m(\_, y^c_m)$, and $\texttt{set}^c_{m'}(\_, u^c_{m'})$ or $\texttt{get}^c_{m'}(\_, y^c_{m'})$, for any other clock $u^c_{m'} \in U^c_{m'}$ or $y^c_{m'} \in Y^c_{m'}$ and instance $m' \in M$ that is transitively connected to y^c_m or has become active as a result of the clock activations;
   (b) Invoke $\texttt{get}_{m'}(\_, y_{m'})$ and $\texttt{set}_{m'}(\_, u_{m'}, \_)$ in the appropriate order, for any instance $m' \in M$.
4. Invoke $\texttt{stepE}_m(\_)$ for $m \in M$ (signals end of Event iteration t_I).
5. Schedule clocks by invoking $\texttt{nextT}_m$ on every relevant clock, for $m \in M$.
6. If any $m \in M$ wishes to repeat the event iteration, or if a clock returned a zero interval, go to Step 3 (start iteration $t_I + 1$).

The goal of Step 3 is to solve the system of equations that became active due to the clock activations. There are no guarantees that such a system has a solution, or that the clock activations will stabilize. It is up to the Importer to determine this, so we leave it intentionally unspecified.

4 Scheduled Execution

SE and SC have the following in common: they use the same clock types, as introduced in Section 2; connected clocks will tick at the same simulated times (although the corresponding equations will be executed at different wall-clock times, see below); after each clock tick, there may be more clock ticks, either at the same simulated time, or at some time in the future.

However, there are key differences, the meaning of which will become clearer later.

- Each SE clock w, when activated at simulated time $t_R \in \mathbb{R}_{\geq 0}$, represents a task that needs to be executed at simulated time t_R. In contrast, in SC, w merely enables a set of equations that are subsequently solved.
- In SE, there is a clear distinction between the wall-clock time, and the simulated time. For example, two clocks may tick at the same simulated time $t_R \in \mathbb{R}_{\geq 0}$ (because they are connected, or because they have the same period), but their corresponding tasks will execute at different wall-clock times. The two tasks themselves, however, will be computed with simulated time t_R.
- In SE, the execution of a task can be pre-empted in order to execute a higher priority task. This has the important consequence that the FMU must inform the importer of when a task should not be pre-empted.

The main goal of SE is to facilitate real-time simulation.

4.1 Motivating Example

Figure 6 shows an abstract example, where an FMU declares three input clocks and one output clock. Each input clock, when ticked, instructs the importer, who acts as a task scheduler (recall Figure 1), to execute the corresponding model partition (defined next) as soon as possible.

A model partition, or just partition, represents code that should be executed as soon as (in real time) an input clock ticks. Partitions contain arbitrary code that reads the inputs of the FMU, writes to the FMU's local variables (which can be shared among tasks) and outputs, and can trigger other clocks or update their interval. The inputs to each partition are set by the importer immediately before executing that partition, as part of the task corresponding to that partition. In Figure 6, u^c_m's partition reads and writes the shared variable x_m, and either updates the interval of v^c_m or ticks y^c_m.

We stress the distinction between model partition and a task: the former represents code that is executed within

the context of the later. So a task T contains code that sets the inputs of the FMU, invokes the model partition P, and reads the outputs. Such a task will simply be denoted as "P's task". For example, in Figure 6, when execution Partition 1's task, the importer sets the values for input u_m before executing Partition 1.

In SE, there is a need for a function that the importer invokes, to tell the FMU to execute a partition. Note the difference between the partition activation function (defined later) and the clock set/get functions: the later inform the importer that a task should be scheduled, while the former executes as part of the previously scheduled task.

In Figure 6, input clock u_m^c ticks every 10ms and w_m^c ticks every 50ms, so every so often, the two clocks will tick simultaneously. When that happens, the scheduler needs to know whose task has the highest priority. As a result, the FMU needs to declare a priority level for each input clock. In Figure 6, u_m^c's task (the one executing Partition 1) should be executed before w_m^c's (Partition 3).

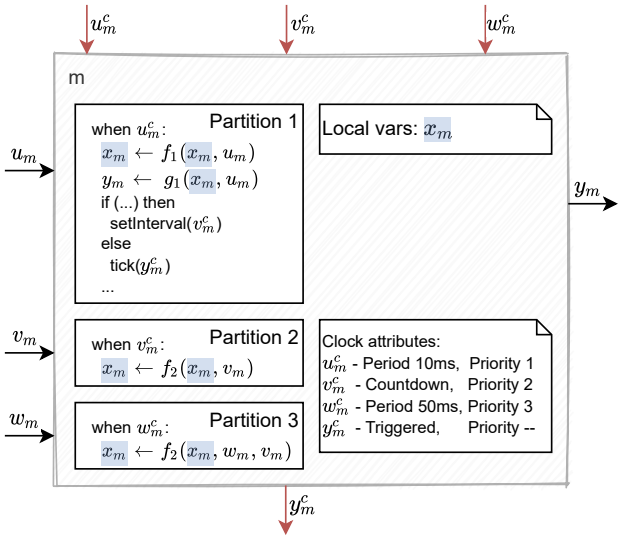

Figure 6. Motivating example, where an FMU declares three input clocks and one output clock.

Output clocks, in SE, are never directly associated to a partition of the FMU where they are declared. Instead, these can be connected to input clocks (including the ones of the owning FMU).

Because tasks can be pre-empted, certain operations, such as updating a shared variable, must be atomic (see example below). As such, the FMU must inform the importer of when it should not be interrupted, to prevent mixed resource access that would create inconsistent values.

Since partitions can trigger and update the interval of other clocks, there must be a mechanism for the FMU, in the middle of the calculation of a partition, to inform the importer that a clock has ticked or has a new interval, so that the importer can schedule the corresponding tasks.

Figure 7 illustrates a possible execution trace of the tasks corresponding to the partitions declared in Figure 6. At the initial wall-clock time, both task 1 and 3 are scheduled to execute. Since Task 1 has higher priority, it runs first, and Task 3 is delayed. While executing Task 1, the FMU informs the importer that v_m^c's task (Task 2) should be scheduled to run at wall-clock time t_2. At wall-clock time t_2, Task 1 is still executing, so Task 2 is delayed until wall clock time t_3. At t_3, Task 2 starts executing, but note that the activation time of Task 2 is still its scheduled time t_2. This is where the wall-clock time t_3 differs from the simulated time t_2. At t_4, Task 2 is pre-empted, because of Task 1. Finally, after being delayed substantially, Task 3 gets to execute, with its simulated time t_0.

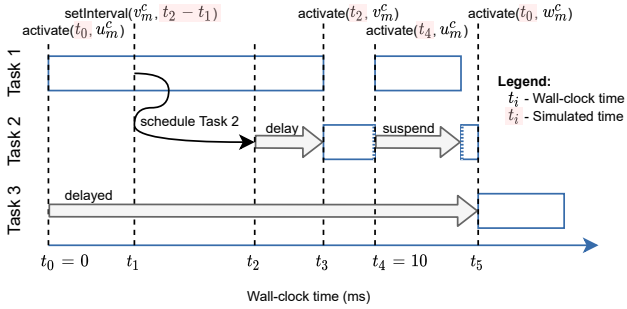

Figure 7. Example execution trace of Figure 6.

4.2 Scheduled Execution Semantics

The following formalization is a simplification meant to highlight the main functions defined in the FMI Standard. The main concepts being formalized are tasks, clocks, and activation of model partitions.

Definition 2 (SE FMU Instance)**.** An SE FMU instance with identifier m is represented by the tuple

$$\langle S_m, U_m, Y_m, U_m^c, Y_m^c, \\ \texttt{set}_m, \texttt{get}_m, \texttt{get}_m^c, \texttt{activate}_m, \texttt{nextT}_m \rangle$$

where:

- S_m, U_m, Y_m, U_m^c, and Y_m^c, are defined as in Definition 1.
- $\texttt{set}_m : S_m \times U_m \times \mathcal{V} \to S_m$ and $\texttt{get}_m : S_m \times Y_m \to \mathcal{V}$ are functions to set the inputs and get the outputs, respectively. In contrast with the SC FMU in Definition 1, $\texttt{get}_m$ does not alter m's state because any non-trivial computation of outputs should be done in the partitions associated with the input clocks, executed through the invocation of the $\texttt{activate}_m$ function.
- $\texttt{get}_m^c : S_m \times Y_m^c \to S_m \times \mathbb{B}$ queries the output clocks. Note that, in contrast to SC, $\texttt{get}_m^c(\_, y_m^c)$ changes the state of m, because it automatically de-activates y_m^c (the justification is provided below).
- $\texttt{activate}_m : S_m \times U_m^c \times \mathbb{R}_{\geq 0} \to S_m$ is a function representing the computation of a partition. If m is in state s_m at wall-clock time t_i, $s_m' = \texttt{activate}_m(s_m, u_m^c, t_i)$ represents three successive

steps: the activation of clock u_m^c, the computation of the partition associated to clock u_m^c, and de-activation of clock u_m^c. In state $s_m{}'$, clock u_m^c will be inactive.
- $\texttt{nextT}_m : S_m \times U_m^c \to \mathbb{R}_{\geq 0} \cup \{NaN\}$ is the function that allows the importer to query the time of the next clock tick. It is defined as in Definition 1.

In addition, we will use the notation $\texttt{task}(u_m^c)$, for $u_m^c \in U_m^c$, to denote the task that will execute u_m^c's partition.

Scheduling Tasks. In SE, the importer operates as a scheduler of tasks that will activate the model partitions. As summarized in Table 1, clocks can be ticked by the FMU or importer, but we will focus on input clocks, since these are the ones that can be associated to a partition (when an output clock ticks, the importer is responsible for ticking all connected input clocks and therefore scheduling the corresponding tasks). Right after invoking $(s_m{}', true) = \texttt{get}_m^c(\_, y_m^c)$ on an output clock y_m^c that is active, the clock y_m^c should be inactive in state $s_m{}'$.

An input clock u_m^c may tick, and its period may be updated, in the middle of an $\texttt{activate}_m$ computation. Because $\texttt{task}(u_m^c)$ may be of high priority, the importer must not wait until the end of $\texttt{activate}_m$ to schedule $\texttt{task}(u_m^c)$. As such, the importer implements a function $\texttt{update}_m : S_m \to S_m$ defined by the FMI, that the FMU can invoke (in the FMI Standard, this is implemented as a callback mechanism). The importer, inside $\texttt{update}_m$, may consult the status of clocks and their intervals (through $\texttt{get}_m^c$ and $\texttt{nextT}_m$ functions), and schedule the corresponding tasks accordingly.

The time at which the importer schedules $\texttt{task}(u_m^c)$ is computed according to: u_m^c's declared interval; function $\texttt{nextT}_m$; or through the $\texttt{get}_{m'}^c(\_, y_{m'}^c)$ function of some other clock $y_{m'}^c$ and FMU instance m'. In the last case, $\texttt{task}(u_m^c)$ is scheduled to execute as soon as possible, according to the priorities known to the importer.

Executing Tasks. A task $\texttt{task}(u_m^c)$ that is scheduled to time t_i, due to the priorities chosen and consequent delays incurred, may only execute at a later wall-clock time $t_j > t_i$. When $\texttt{task}(u_m^c)$ is executed, it should set the relevant inputs through function $\texttt{set}_m$ (the importer knows the relevant inputs through the XML of m), activate the partition trough function $\texttt{activate}_m(\_, u_m^c, t_i)$, and possibly read the calculated outputs, through $\texttt{get}_m$.

Safeguarding Pre-emption. Unless otherwise stated by the FMU or importer, a task can be pre-empted at any moment. In order to allow the FMU to inform its environment that the currently executing task should not be pre-empted, the FMI defines two functions: `lockP` and `unlockP` that the FMU and importer can invoke, and are implemented by the importer. `lockP` informs the environment that a task cannot be pre-empted until `unlockP` is invoked.

Generic Scheduled Execution Algorithm. Let M denote a set of FMU instances, assumed to be initialized.

1. Schedule $\texttt{task}(u_m^c)$, for all $u_m^c \in U_m^c$ and all $m \in M$, if interval of u_m^c is constant fixed, or calculated;
2. When $\texttt{update}_m(\_)$ is invoked, do:
   (a) Lock pre-emption with `lockP`;
   (b) If $(\_, true) = \texttt{get}_m^c(\_, y_m^c)$, schedule $\texttt{task}(u_{m'}^c)$ for any clock $u_{m'}^c$ that is transitively connected to y_m^c.
   (c) Unlock pre-emption with `unlockP`;
3. Each task $\texttt{task}(u_m^c)$ is implemented as:
   (a) Set the inputs of m using $\texttt{set}_m$ (locking pre-emption with `lockP` and `unlockP` if needed);
   (b) Invoke $\texttt{activate}_m(\_, u_m^c, t_i)$, where t_i is the simulated time that $\texttt{task}(u_m^c)$ was scheduled to execute.
   (c) Get the outputs of m using $\texttt{get}_m$ (locking pre-emption with `lockP` and `unlockP` if needed);

5 Related Work

Synchronous clocks are one of the solutions proposed to tackle the more general challenge of co-simulating hybrid systems. Other proposals have been made in the state of the art, but none of them tackle the problem of discerning different simultaneous events in the context of co-simulation. For instance, Cremona et al. (2016) proposes a master algorithm for hybrid co-simulation. The proposal includes support for absent signals, mandatory implementation of rollback, zero duration step size, co-simulation FMUs supporting feed-through, and predictable step sizes. However, it excludes algebraic loops, due to the introduced non-determinism. Our proposed interfaces enables algebraic loop resolution, even when clocks are involved, but does not provide guarantees of convergence.

An extensive study of hybrid system simulation challenges was carried out in Mosterman and Biswas (2000), and includes, for example, the possibility of an event iteration driving the system into chattering. And Tripakis and Broman (2014), Broman et al. (2015) and Liboni et al. (2018) focus such discussion in the context of the FMI Standard, providing solutions to some of these challenges. These works complement ours by helping importers assess whether a given simulation scenario is well behaved. We refer the reader to Gomes et al. (2018) for more references in co-simulation of hybrid systems.

The goal of this paper is to describe the main mechanisms standardized in the FMI Standard that enable synchronous clocked simulation and scheduled execution. We can therefore highlight related work that share the same goals.

Regarding SC simulation, we highlight the work in Otter, Thiele and Elmqvist (2012) and Elmqvist, Otter and Mattsson (2012), that introduce the synchronous clocks constructs used in the Modelica language, specified in Modelica Association (2021). Such work, and references thereof on synchronous languages (Benveniste et al. 2003; Colaço and Pouzet 2003), were used as basis for the definition of the SC approach described here. The main difference is that an SC clock does not enforce a partition on

the equations that can be written by it. These differences make it more difficult to ensure well-formedness of co-simulation scenarios, but provide more flexibility, reflecting the heterogeneous use cases of FMI.

In the domain of scheduled execution, we highlight the OSEK/VDX (ISO 17356-3:2005 2005) and AUTOSAR Standards, which enable different suppliers to develop and test software independently, and subsequently integrated the different applications. Such work complements the SE Interface by standardizing the importer environment, where FMU SE instances can execute.

6 Conclusion

This paper summarizes the results of the FMI project developing interfaces to interact with clocks. This is a challenging task, because the kinds of simulation scenarios covered can combine traditional events with clock ticks, and may possibly be ill-defined while still conforming to the FMI Standard. This is intentional, as the FMI aims at flexibility, placing the burden of ensuring well-formedness on the importer.

We have presented two interpretations of clocks. The main differences between them lie in the degree of control that the importer has over the duration of computations, and on the behavior of the independent variable with respect to the wall clock time. The formalization provided is meant as an introduction to the clocks and their conceptualization in the FMI Standard. The FMI Standard document is continuously being improved, and therefore remains the source of truth.

Acknowledgements

Cláudio Gomes is grateful to the Poul Due Jensen Foundation, which has supported the establishment of a new Centre for Digital Twin Technology at Aarhus University.

We are thankful to Simon Thrane Hansen and Peter Gorm Larsen, for their helpful comments on this paper.

Finally, we are thankful to the Modelica association for the continued support to the FMI Project.

References

2.0, FMI (2014). *Functional Mock-up Interface for Model Exchange and Co-Simulation*. https://fmi-standard.org/downloads/.

Benveniste, A. et al. (2003-01). "The Synchronous Languages 12 Years Later". en. In: *Proceedings of the IEEE* 91.1, pp. 64–83. ISSN: 0018-9219. DOI: 10.1109/JPROC.2002.805826.

Broman, David et al. (2015). "Requirements for Hybrid Cosimulation Standards". In: *18th International Conference on Hybrid Systems: Computation and Control*. Seattle, Washington: ACM New York, NY, USA, pp. 179–188. ISBN: 978-1-4503-3433-4. DOI: 10.1145/2728606.2728629.

Colaço, Jean-Louis and Marc Pouzet (2003). "Clocks as First Class Abstract Types". In: *Embedded Software*. Ed. by Gerhard Goos et al. Vol. 2855. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 134–155. ISBN: 978-3-540-20223-3 978-3-540-45212-6. DOI: 10.1007/978-3-540-45212-6_10.

Cremona, Fabio et al. (2016-11). "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.

Elmqvist, Hilding, Martin Otter and Sven Erik Mattsson (2012). "Fundamentals of Synchronous Control in Modelica". In: *9th International Modelica Conference*.

Gomes, Cláudio et al. (2018). "Co-Simulation: A Survey". In: *ACM Computing Surveys* 51.3, 49:1–49:33. DOI: 10.1145/3179993.

ISO 17356-3:2005 (2005). *Road Vehicles — Open Interface for Embedded Automotive Applications — Part 3: OSEK/VDX Operating System (OS)*. Tech. rep. ISO 17356-3:2005, p. 61.

Kübler, R. and W. Schiehlen (2000). "Two Methods of Simulator Coupling". In: *Mathematical and Computer Modelling of Dynamical Systems* 6.2, pp. 93–113. ISSN: 1387-3954. DOI: 10.1076/1387-3954(200006)6:2;1-M;FT093.

Lee, Edward A. and Haiyang Zheng (2005). "Operational Semantics of Hybrid Systems". English. In: *Hybrid Systems: Computation and Control*. Vol. 3414. Springer Berlin Heidelberg, pp. 25–53. ISBN: 978-3-540-25108-8. DOI: 10.1007/978-3-540-31954-2_2.

Liboni, Giovanni et al. (2018-01). "Beyond Time-Triggered Co-Simulation of Cyber-Physical Systems for Performance and Accuracy Improvements". In: *10th Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*. Manchester, United Kingdom.

Modelica Association (2021). *Modelica - A Unified Object-Oriented Language for Physical Systems Modeling*. Language Specification Version 3.5. Modelica Association.

Mosterman, Pieter J and Gautam Biswas (2000-08). "A Comprehensive Methodology for Building Hybrid Models of Physical Systems". In: *Artificial Intelligence* 121.1–2, pp. 171–209. ISSN: 0004-3702. DOI: http://dx.doi.org/10.1016/S0004-3702(00)00032-1.

Otter, Martin, Bernhard Thiele and Hilding Elmqvist (2012). "A Library for Synchronous Control Systems in Modelica". In: *9th International Modelica Conference*.

Tarjan, Robert (1971-10). "Depth-First Search and Linear Graph Algorithms". In: *12th Annual Symposium on Switching and Automata Theory (Swat 1971)*. Vol. 1. East Lansing, MI, USA. ISBN: SMJCAT00000100000200014600000 1. DOI: 10.1109/SWAT.1971.10.

Tripakis, Stavros and David Broman (2014). *Bridging the Semantic Gap Between Heterogeneous Modeling Formalisms and FMI*. Tech. rep.