Bridging the Semantic Gap Between Heterogeneous Modeling Formalisms and FMI

Stavros Tripakis
Aalto University and University of California, Berkeley

***Abstract*—FMI (Functional Mockup Interface) is a standard for exchanging and co-simulating model components (called FMUs) coming from potentially different modeling formalisms, languages, and tools. Previous work has proposed a formal model for the co-simulation part of the FMI standard, and also presented two co-simulation algorithms which can be proven to have desirable properties, such as determinacy, provided the FMUs satisfy a formal contract. In this paper we discuss the principles for encoding different modeling formalisms, including state machines (both untimed and timed), discrete-event systems, and synchronous dataflow, as FMUs. The challenge is to bridge the various semantic gaps (untimed vs. timed, signals vs. events, etc.) that arise because of the heterogeneity between these modeling formalisms and the FMI API.**

I. Introduction

FMI (Functional Mockup Interface) is an evolving standard for model exchange and co-simulation (see https://www.fmi-standard.org/). FMI for co-simulation allows to import and co-simulate within a single framework model components which have been designed using potentially distinct modeling formalisms, languages, and tools. This is a key functionality needed to design modern cyber-physical systems, which often involve several design teams coming from different disciplines and using different modeling environments.

The FMI standard defines an API (application programming interface) to which these model components, called FMUs (functional mockup units), must conform. Thus, each FMU can be seen as a *black-box* which implements the methods defined in the FMI API (some of these methods are optional, while others are mandatory, meaning that all FMUs must implement them).

FMUs by themselves are *passive* objects, in the sense that they do not execute. For that reason they are also called *slaves*. To execute (simulate) an FMU, we need a so-called *master algorithm* (MA). The MA coordinates the execution of a number of FMUs connected in a network. This network can be seen as the entire model, sub-models of which are FMUs. Running the MA on this global model means performing one simulation run.

Previous work [1] proposed a formal model of (a subset of) FMI for co-simulation, together with a formal *contract* that the FMUs must obey. That work also proposed two master algorithms and proved that these two algorithms have desirable properties, such as termination and determinacy. The latter means that the results of a simulation do not depend on arbitrary factors (e.g., FMU names), but only on the interconnections of the FMUs in a network.

To our knowledge, the question *how to create FMUs* has not been formally addressed. This question is addressed in this paper. In particular, we discuss the principles of encoding *heterogeneous* modeling formalisms (untimed and timed state machines, discrete event models, and synchronous dataflow) as FMUs. As a result, this paper together with previous paper [1] provides a complete methodology for modeling and simulation of heterogeneous systems: (1) model different subsystems of the system using various languages and tools; (2) use the results of this paper to generate an FMU from each subsystem model; (3) use the results of [1] to co-simulate the collection of all FMUs produced in step (2).

In order to encode as FMUs such a broad set of heterogeneous formalisms, we need to overcome a significant challenge, namely, *how to bridge the gap between the semantics of the original formalisms, and the FMI API*. The exact nature of this semantic gap depends on the original formalism. For instance, in the case of encoding a finite state machine (FSM) such as a Mealy or Moore machine [2] as an FMU, we need to bridge the gap between the *untimed* semantics of FSMs and the timed semantics of FMI. While in the case of encoding a discrete-event (DE) actor such as those used in Ptolemy [3], [4] as an FMU, we need to bridge the gap between the event-based semantics of DE and the persistent signal-based semantics of FMI. In the rest of the paper we discuss in more detail the semantic gap and explain how it can be bridged, for each source formalism in turn.

An extended version of this paper can be found in [5].

II. A Formal Model for FMI

We recall the formal model for FMI proposed in [1]. An FMU is a tuple $F = (S, U, Y, D, s_0, \mathtt{set}, \mathtt{get}, \mathtt{doStep})$, where:

- S is the set of (internal) states of F. Note that an element of S is a state, not a state variable.
- U is the set of input variables of F. Note that an element $u \in U$ is a variable, not a value. Each variable in U ranges over a set of values $\mathbb{V}$. F is called a *source* if U is the empty set.
- Y is the set of output variables of F. Each variable in Y ranges over the same set of values $\mathbb{V}$. F is called a *sink* if Y is the empty set.
- $D \subseteq U \times Y$ is a set of *input-output dependencies*. D specifies for each output variable which input variables it

This work was partially supported by the iCyPhy Research Center (Industrial Cyber-Physical Systems, supported by IBM and United Technologies), by the Academy of Finland, and by the NSF (awards #1329759 and #1139138). The author would like to thank David Broman from KTH and UC Berkeley, Lev Greenberg and Michael Masin from IBM, Edward A. Lee and Christopher Brooks from UC Berkeley, Michael Wetter from Lawrence Berkeley National Laboratories, and Fabio Cremona from Scuola Superiore Sant'Anna.

depends upon (if any). This information is used to ensure that a network of FMUs has no cyclic dependencies, and also to determine the order in which all network values are computed during a simulation step [1].

- $s_0 \in S$ is the initial state of F.[1]
- $\mathtt{set} : S \times U \times \mathbb{V} \to S$ is the function that sets the value of an input variable. Given state s, input variable $u \in U$, and value $v \in \mathbb{V}$, $\mathtt{set}(s, u, v)$ returns the new state obtained after setting u to v.
- $\mathtt{get} : S \times Y \to \mathbb{V}$ is the function that returns the value of an output variable. Given state s and output variable $y \in Y$, $\mathtt{get}(s, y)$ returns the value of y in s.
- $\mathtt{doStep} : S \times \mathbb{R}_{\geq 0} \to S \times \mathbb{R}_{\geq 0}$ is the function that implements one simulation step. Given state s, and time step $h \in \mathbb{R}_{\geq 0}$, $\mathtt{doStep}(s, h)$ returns a pair (s', h') such that:
  - either $h' = h$, which is interpreted as F having *accepted* h, and having advanced to new state s';
  - or $0 \leq h' < h$, which is interpreted as F having *rejected* h, but having made partial progress up to h', and having reached new state s'.

A. Semantics of a single FMU

For pedagogical purposes, we explain first the semantics of a single FMU, and then the semantics of a network of interconnected FMUs.

Consider an FMU $F = (S, U, Y, D, s_0, \mathtt{set}, \mathtt{get}, \mathtt{doStep})$. Since F generally has inputs, its behavior may depend on the values that these inputs take. In addition, the behavior of F depends on the times when functions such as doStep are invoked. Therefore, the behavior of F is a function of a *timed input sequence* (TIS). A TIS is an infinite sequence

$$\mathbf{v}_0 h_1 \mathbf{v}_1 h_2 \mathbf{v}_2 h_3 \cdots$$

of alternating input assignments, $\mathbf{v}_i$, and time delays, $h_i \in \mathbb{R}_{\geq 0}$. An input assignment is a function $\mathbf{v} : U \to \mathbb{V}$. That is, $\mathbf{v}$ assigns a value to every input variable in U.

A TIS like the above defines a *run* of F, which is an infinite sequence of quadruples $(t, s, \mathbf{v}, \mathbf{v}')$, where $t \in \mathbb{R}_{\geq 0}$ is a time instant, $s \in S$ is a state of F, $\mathbf{v}$ is an input assignment, and $\mathbf{v}' : Y \to \mathbb{V}$ is an output assignment:

$$(t_0, s_0, \mathbf{v}_0, \mathbf{v}'_0)(t_1, s_1, \mathbf{v}_1, \mathbf{v}'_1)(t_2, s_2, \mathbf{v}_2, \mathbf{v}'_2) \cdots$$

defined as follows:

- $t_0 = 0$ and s_0 is the initial state of F.
- For each $i \geq 1$, $t_i = t_{i-1} + h_i$, that is, t_i is the sum of all delays up to the i-th step, in other words, the time when the i-th step occurred.
- For each i, $\mathbf{v}'_i$ is obtained as follows. First, starting from the current state s_i, the set method is used repeatedly to set all input variables to the values specified by $\mathbf{v}$. This results in a new state s'_i. Next, starting from this new state s'_i, the get method is used to read the values of all output variables. This results in $\mathbf{v}'_i$.
  Note that this step also resulted in a new, intermediate state s'_i. This state is further used in computing the next state using doStep.
- For each i, we require that $\mathtt{doStep}(s'_i, h_{i+1}) = (s_{i+1}, h_{i+1})$, where s'_i is the intermediate state computed in the previous step. This means that we are assuming that every h_i is accepted by F,[2] and results in the "next" state s_{i+1}.

Note that time delays h_i can be zero. As a consequence, the run of F can be seen as a sequence of events (i.e., changes of state, from s_i to s_{i+1}) taking place in so-called *superdense time* [6]. An event takes place at superdense time instant (t, n), where $t \in \mathbb{R}_{\geq 0}$ and $n \in \mathbb{N}$. The first element, t, models the "real-time" instant when the event took place, and the second element, n, is the *index* modeling how many events prior to this one also took place at the same t. For example, consider a run of the form:

$$(0, s_0, \_, \_)(1, s_1, \_, \_)(1, s_2, \_, \_)(4, s_3, \_, \_) \cdots$$

In this run, the event $s_0 \to s_1$ takes place at superdense time $(1, 0)$. The event $s_1 \to s_2$ takes place at $(1, 1)$. The event $s_2 \to s_3$ takes place at $(4, 0)$. And so on.

B. Semantics of a network of FMUs

A network of FMUs is a set of FMUs plus a set of connections. A connection is a pair (y, u) where y is an output variable of one FMU and u is an input variable of another (or the same) FMU. We require that an input be connected to at most one output. On the other hand, an output may be connected to many inputs. We require that this set of connections, together with the input/output dependencies defined by individual FMUs, form an acyclic graph. Provided this holds, the network of FMUs defines a single FMU F, as follows [1].

- F has as output variables all output variables from all FMUs in the network.
- F has as input variables all input variables which are not connected to an output.
- The functions get and set for F are mapped to the corresponding get and set functions of individual FMUs in the network, to which the variable which is read or written belongs to.
- The function doStep of F corresponds to running one integration step of (one of) the MA proposed in [1]. In a nutshell, the MA propagates values from outputs to inputs throughout the network, and then performs a doStep to all FMUs. The key issue is how to choose the right time step to make sure that it is accepted by all FMUs. Different techniques, such as rollback, can be used to achieve this goal. See [1] for details.

Since a network of FMUs can be seen as a single FMU, based on the principles above, the semantics of a network of

[1] Our formalization differs in this point from the formalization in [1]. The latter considered a function $\mathtt{init} : \mathbb{R}_{\geq 0} \to S$ which, given as input a time $t \in \mathbb{R}_{\geq 0}$, returns the state, implicitly at that time t. This requires, for correct simulation, that all FMUs in a model are initialized to the same initial time t. We note that the initialization phase of the MAs was not discussed in [1], and the init function was not used in the algorithms presented there. Here, we take a simpler approach and consider that all FMUs are initialized to a given state at time 0.

[2] We are not concerned here with how the environment can "guess" the right values for h_i so that they are all accepted by F. In fact, the environment does not have to guess, as this is precisely the job of the MA. The MA finds the right time step h which is accepted by all FMUs in a model, possibly by trial-and-error, i.e., using rollback. See [1] for details.

FMUs can be defined to be the semantics of the single FMU that this network defines.

III. Encoding Untimed State Machines as FMUs

We begin by considering state machines of type Moore or Mealy, which are typically used to model digital circuits [2]. Usually this type of machine is finite, in the sense that it has a finite number of states, and finite domain of possible input and output values. Here we will consider a more general model, where the domains of input, output, and state variables can be infinite. Such machines are useful for capturing, for example, embedded controllers, such as those that can be programmed using synchronous languages like Lustre [7].

A. Mealy and Moore machines

A Mealy machine is a tuple $M = (I, O, S, s_0, \delta, \lambda)$, where I is a set of input values, O a set of output values, S a set of states, $s_0 \in S$ an initial state, $\delta : S \times I \to S$ the transition function, and $\lambda : S \times I \to O$ the output function. Given current state s_n and current input i_n, $\delta(s_n, i_n)$ determines the next state s_{n+1}, i.e., $s_{n+1} = \delta(s_n, i_n)$. Given current state s_n and current input i_n, $\lambda(s_n, i_n)$ determines the current output o_n, i.e., $o_n = \lambda(s_n, i_n)$.

A Moore machine is a special case of a Mealy machine where the current output does not depend on the current input, but only on the current state, i.e., where $\lambda(s, i) = \lambda(s, i')$ for all $i, i' \in I$. In that case we can simplify and write λ only as a function of S, $\lambda : S \to Y$, and then also write $\lambda(s)$ instead of $\lambda(s, i)$.

B. Semantic gap: from untimed to timed

How can we encode a given Mealy machine as an FMU? The main issue is that Mealy machines are *untimed*, in the sense that they have no a-priori notion of quantitative or real time, but only a notion of "logical" time (a totally ordered set of ticks). For instance, and in contrast to models such as timed automata [8], we cannot express things like "the second input is given 3 time units after the first input" or "between two successive transitions, 2 seconds elapse." We can only express order, namely, that the second input/transition comes after the first, the third after the second, etc. On the other hand, FMUs are timed in the sense that `doStep` takes as input a time delay $h \in \mathbb{R}_{\geq 0}$, and also the semantics of an FMU as defined in Section II-A is timed.

Given the above, in order to map a Mealy machine to an FMU, we have to somehow create a "timed wrapper" of the untimed state machine. There are (at least) two ways to do that:

Periodic wrapper : in this case, the user specifies a *period* $T \in \mathbb{R}_{>0}$ and the machine takes transitions precisely at multiples of T. In this approach, the advancement of time is controlled by the machine itself, and not by the environment of the machine.

Aperiodic wrapper : in this case, every `doStep` invocation is mapped to a transition of the state machine. In this approach, the advancement of time is controlled by the environment, since it is the one that decides when the transitions occur.

In what follows we focus on the periodic wrapper. See [5] for a more detailed discussion.

C. Periodic wrapper

The periodic wrapper approach for encoding a Mealy machine as an FMU can be seen as wrapping the machine with a periodic sampler at the inputs and a "hold" at the outputs. Inputs are sampled every T time units (T is a parameter provided by the user). Outputs remain constant until the next sampling occurs.[3]

Let us describe the periodic wrapper approach in more detail. Given a period $T \in \mathbb{R}_{>0}$, a Mealy machine $M = (I, O, S, s_0, \delta, \lambda)$ is encoded as the FMU $F = (I \times S \times [0, T], \{p\}, \{q\}, \{(p, q)\}, \hat{s}_0, \mathtt{set}, \mathtt{get}, \mathtt{doStep})$, where:

- F has a single input variable p, ranging over I, and a single output variable q, ranging over O.
- F has an internal state variable s, ranging over S, and an internal state variable t, ranging in the real interval $[0, T]$. Note that in addition to those state variables, F also has p and q as state variables.
- In $\hat{s}_0$, p and q are set to some initial arbitrary values in I and O, respectively, s is set to s_0 and t is set to T. Note that the initial input value does not matter, since `set` must be called before `get` and `doStep` are called.
- `set` sets p to a given $i \in I$.
- `get` computes and returns $\lambda(s, i)$, where i is the current value of p.
- `doStep` behaves as follows, for given input $h \in \mathbb{R}_{\geq 0}$:
  1) If $h < t$ then `doStep` accepts h, sets t to $t - h$, and returns h.
  2) If $h = t$ then `doStep` accepts h, resets t to T, sets s to $\delta(s, x)$, where x is the current value of p, and returns h.
  3) If $h > t$ then `doStep` rejects h, sets a temporary variable d to t, resets t to T, sets s to $\delta(s, x)$, where x is the current value of p, and returns d.

Case 1 corresponds to the case where the environment requests a time step h smaller than t, the time remaining until the end of the period. In this case the FMU accepts the step, and advances time by h. Case 2 corresponds to the case where $h = t$. In that case, the step is accepted, time advances, and since the end of a period is reached, a new period begins by performing a transition and resetting the timer t to T. Case 3 corresponds to the case where the environment requests a time step h greater than t. In this case h is rejected, but the FMU still makes partial progress, advancing time up to the end of the period, i.e., by t, and again taking a transition as in the second case.

The above encoding works, however, an alternative encoding may be better, although slightly more complex to describe. In this alternative encoding `doStep` behaves as follows, for given input $h \in \mathbb{R}_{\geq 0}$:

1) If $h < t$ then `doStep` accepts h, sets t to $t - h$, and returns h.
2) If $h = t$ then
   a) If $h > 0$ then `doStep` sets $t := 0$ and returns h.
   b) If $h = 0$ then `doStep` sets s to $\delta(s, x)$, where x is the current value of p, sets $t := T$, and returns 0.
3) If $h > t$ then

[3] An alternative is to consider outputs as *events* which occur only at multiples of T, and are "absent" otherwise. This approach is discussed in Sections IV and VI.

a) If $t > 0$ then `doStep` rejects h, sets a temporary variable d to t, sets $t := 0$ and returns d.
b) If $t = 0$ then `doStep` sets s to $\delta(s, x)$, where x is the current value of p, sets $t := T$, and returns 0.

The difference is that now a transition happens in two *superdense steps*. First t reaches 0. Then t is reset to T. We find this encoding preferable, since it allows the MA to distinguish between rapid changes of a continuous signal vs. a discrete change of, say, a piecewise constant signal.

IV. Encoding Discrete-Event Actors as FMUs

Discrete-event (DE) is a timed model where a set of processes, called *actors*, interact by exchanging timed events. DE is one of the models of computation supported in a number of languages and tools such as ns-3, VHDL, SimEvents, and Ptolemy [3]. The semantics of DE have been formalized in various papers, e.g., [4], [9], [10]. Here we show how to encode a typical DE actor such as *Periodic Clock* as an FMU. The Periodic Clock actor has a parameter T (the period) and produces a sequence of discrete events at multiples of T, i.e., at times $0, T, 2T, \cdots$.

A. Semantic gap: from events to persistent signals

DE is a timed model, so encoding DE actors as FMUs does not raise the untimed vs. timed issues encountered in Section III. On the other hand, we need to deal with another type of semantic gap, namely, the fact that DE relies on a primitive notion of discrete event, whereas FMUs communicate a-priori by *persistent* input and output signals. These signals are "persistent" in the sense that the value of an output, for instance, can be requested at any point in time by calling `get`.

To solve this problem we follow the approach of [9] and introduce a special value denoted `absent`, which models the absence of an event at a certain point in time. Output variables that carry events will have value `absent` most of the time, except at those times when an event occurs, in which case the value of the output variable is the value of the occurring event.

For example, consider the case of Periodic Clock, which will be encoded as an FMU F_C with no inputs, i.e., a source FMU. For source FMUs, a TIS is just a sequence of time delays, $h_1 h_2 h_3 \cdots$. Suppose a TIS with $h_1 := 1$ is fed into F_C when the period is $T = 2$. Then, `doStep` is called for the first time with $h = 1$, and time advances from 0 to 1. At that point `get` is called, and F_C should return `absent`, since no event is output at time 1.

The Periodic Clock FMU is defined formally below. Our approach however is general, and applies to all kinds of DE actors, including actors which receive input events arriving with arbitrary "frequency". An example is the *Constant Delay* actor, parameterized by Δ (the delay), which delays every event that it receives at its input by Δ time units. For instance, if it receives events at times $t_1, t_2, t_3, \cdots$, then it produces events at times $t_1 + \Delta, t_2 + \Delta, t_3 + \Delta, \cdots$, and so on. In [9] the formalization of this actor relies on the so-called *deadline* function which communicates to the simulation algorithm (called *director* in the Ptolemy world) how much time the actor is willing to let elapse. FMI has no such deadline function, but the FMU can still communicate the same information to the simulation algorithm, by rejecting a time step and returning the maximum amount of time it is willing to let elapse.

B. FMU for a periodic clock

The Periodic Clock actor has a parameter T (the period) and produces events at times $0, T, 2T, \cdots$. These events typically also have a value, which we will assume to be some other parameter v. As mentioned above, we will also assume that the set of values $\mathbb{V}$ contains the special value `absent`.

Let us try to model the Periodic Clock as an FMU F_C. A reasonable first attempt is to define F_C as the tuple $([0, T], \{\}, \{q\}, \{\}, s_0, \mathtt{set}, \mathtt{get}, \mathtt{doStep})$, where:

- There is a state variable $t \in [0, T]$ modeling a timer.
- The set of input variables is empty (and therefore `set` is a no-op).
- There is a single output variable q (with no dependencies since there are no inputs).
- s_0 sets the timer variable t to 0 (assuming we want the clock to "tick" also at time 0, otherwise we would initialize t to T).
- $\mathtt{get}(t, q) = \begin{cases} 1, & \text{if } t = 0 \\ \mathtt{absent}, & \text{otherwise.} \end{cases}$
- $\mathtt{doStep}(t, h) = \begin{cases} (t - h, h), & \text{if } t \geq h \\ (0, t), & \text{otherwise.} \end{cases}$

The idea is that F_C maintains a timer t and decrements the counter by the amount of the step size h. Then, F_C outputs the value 1 when t reaches 0, and `absent` before that. The problem with the above modeling is that the counter is never reset to T since, once it is at 0, only $h = 0$ is accepted as a step size and the clock is "stuck". Instead we use the following, which is in accordance with superdense time semantics:

- $\mathtt{doStep}(t, h) = \begin{cases} (T, 0), & \text{if } t = 0 \\ (t - h, h), & \text{if } t > h \\ (0, t), & \text{otherwise.} \end{cases}$

V. Encoding SDF Actors as FMUs

Synchronous Data Flow (SDF) [11] is a dataflow model where a set of actors execute asynchronously and communicate via FIFO queues of (a-priori) unbounded length. The main characteristic of SDF is that the number of tokens that each actor consumes from its input queues and produces to its output queues every time it fires is constant and known in advance. In that sense, SDF can be seen as a restricted subclass of Kahn Process Networks [12].

A. Semantic gap: from asynchronous queues to persistent signals

Encoding an SDF actor as an FMU is not straightforward. In addition to the fact that ("pure") SDF is an untimed model, whereas FMI is timed, we have to solve the problem of bridging the semantic gap between the asynchronous model of concurrency with FIFO queue based communication that SDF is based on, and the somewhat synchronous model that FMI uses, based on persistent signals as discussed above. This is the problem we focus on in this section. We first show how a *closed* SDF graph (i.e., one without open inputs) can be mapped to a network of FMUs in a modular way, i.e., one SDF actor at a time, independently from the rest of the SDF graph. We then discuss how SDF FMUs can interface with other FMUs, e.g., DE FMUs.

B. Encoding a closed SDF graph as a network of FMUs

Consider first the SDF graph shown in Figure 1. This graph has three SDF actors, denoted A, B, C. A is a source actor with a single output variable. A produces 3 tokens every time it fires. B has a single input and a single output variable. B needs at least 4 input tokens in order to fire, and when it does, it consumes 4 tokens from its input queue and produces 2 tokens to its output queue. C is a sink actor with a single input variable. C needs at least 5 tokens in order to fire, and consumes 5 tokens from its input queue every time it fires.[4] An SDF graph may also have initial tokens on the queues. In this graph, there are two queues, denoted α and β. We can identify the output queue of A with the input queue of B, denoted α, and the output queue of B with the input queue of C, denoted β. There are 3 initial tokens in β, and no initial tokens in α.

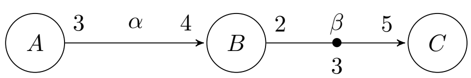

Fig. 1. An SDF graph.

The principles for mapping an SDF graph such as the one above to a network of FMUs are the following. We generate three FMUs, one for each SDF actor. Let us denote them F_A, F_B, F_C, for the actors A, B, C of Figure 1. F_A has a state variable holding an output FIFO queue, denoted Q^o_A. F_B has a state variable holding an input FIFO queue, denoted Q^i_B, and another state variable holding an output FIFO queue, denoted Q^o_B. F_C has a state variable holding an input FIFO queue, denoted Q^i_C. Note that Q^o_A and Q^i_B are distinct, not shared, variables. So are Q^o_B and Q^i_C. This is necessary, since the translation from SDF actors to FMUs is modular, e.g., we map actor B to F_B independently from the other actors A and C. Initially all queues are empty, except for the input queue of C, which has 3 initial tokens, corresponding to the 3 initial tokens of β.[5]

The FMI API is implemented as follows:

- F_A.`get`: returns the current value of the (entire) queue Q^o_A.
- F_A.`doStep`: writes 3 fresh tokens into Q^o_A. Note that the previous value of Q^o_A is over-written.
- F_B.`set`: receives a (possibly empty) ordered list of tokens, and writes this as the value of the input variable of F_B, denoted v^i_B. Note that v^i_B also holds a queue as its value, but is different from the input queue state variable Q^i_B. Also note that any previous value stored in v^i_B is over-written.
- F_B.`doStep`: first, it appends the list stored in v^i_B at the end of Q^i_B. Then it checks whether Q^i_B has enough tokens, in this case at least 4.
  - If so, 4 tokens are removed from Q^i_B, the firing computation of B is performed, and the 2 computed output tokens are (over-)written to Q^o_B.
  - If not, the empty list is (over-)written to Q^o_B.
- F_B.`get`: returns the current value of the (entire) queue Q^o_B.
- F_C.`set`: receives a (possibly empty) ordered list of tokens, and (over-)writes this to v^i_C.
- F_C.`doStep`: first, it appends the list stored in v^i_C at the end of Q^i_C. Then it checks whether Q^i_C has enough tokens, in this case at least 5.
  - If so, 5 tokens are removed from Q^i_C, and the firing computation of C is performed.
  - If not, F_C.`doStep` does nothing more and returns.

One might object that it is redundant to have two separate variables v^i and Q^i. Why not just have Q^i and let `set` append to it whatever it receives as input? This would work, however, it would not conform to the FMU contract described in [1]. In particular, `set` would violate Assumption (A2) of Section 4.2 of [1], since according to (A2) `set` should only over-write the value of an input variable, and cannot implement a more complex operation such as adding an element to a queue.

To illustrate the mapping from SDF to FMUs, let us simulate the beginning of an execution of the network of FMUs F_A, F_B, F_C obtained from the SDF graph of Figure 1:

- Initially, all queues are empty except Q^i_C which contains 3 tokens, say the list $[y_1, y_2, y_3]$.
- F_A.`get` and F_B.`get` both return [] (the empty list).
- Thus, F_B.`set` and F_C.`set` both result in [] being written to v^i_B and v^i_C, respectively.
- F_A.`doStep` produces 3 tokens on its output queue, say the list $[x_1, x_2, x_3]$.
- F_B.`doStep` appends v^i_B, i.e., [], at the end of Q^i_B. The latter remains empty. There are not enough tokens to fire B, so F_B.`doStep` returns.
- F_C.`doStep` returns since there are not enough tokens to fire C.
- This marks the end of the first iteration of the MA, and the second iteration begins.
- F_A.`get` returns $[x_1, x_2, x_3]$. F_B.`set` sets v^i_B to $[x_1, x_2, x_3]$.
- F_B.`get` returns []. F_C.`set` sets v^i_C to [].
- F_A.`doStep` produces 3 tokens on its output queue, say the list $[x_4, x_5, x_6]$.
- F_B.`doStep` appends v^i_B, i.e., $[x_1, x_2, x_3]$, at the end of Q^i_B, which becomes $[x_1, x_2, x_3]$. There are not enough tokens to fire B, so F_B.`doStep` returns.
- F_C.`doStep` returns since there are not enough tokens to fire C.
- This marks the end of the second iteration of the MA, and the third iteration begins. Etc.

C. Interfacing SDF FMUs with other FMUs

Note that although the mapping given above is described for closed SDF graphs such as the one of Figure 1, the translation is still modular, that is, it maps each SDF actor to a separate FMU. This opens the possibility for interfacing *open* SDF FMUs to other types of FMUs, like the ones discussed in previous sections. This interfacing must be performed with care, however, in order to resolve issues of type consistency. We illustrate some of these issues next.

[4] SDF actors may also have internal state, which they update every time they fire.

[5] This is slightly non-modular, as one could argue that the initial tokens are not really part of C, but rather part of the SDF graph. We do not consider it a problem, however, as the initialization could also be done during the instantiation of F_C.

Interfacing SDF FMUs with DE FMUs: Let us consider interfacing SDF FMUs with DE FMUs. By an *SDF FMU* we mean an FMU which is the result of a mapping of an SDF actor such as SDF actor B from Figure 1 and its corresponding FMU F_B. By a *DE FMU* we mean here an FMU producing or consuming discrete events.

Let us first consider a DE FMU F producing a sequence of discrete events at its output. Suppose we want to connect this output to the input of SDF FMU F_B. Our intention here might be that every discrete event produced by F is mapped to a (single) token given to F_B. The above connection does not immediately "type check", however, since F.get returns scalar values (of some type, or absent), whereas F_B.set expects a list. Therefore, we need an FMU to perform the conversion from scalars to lists. This is a simple FMU, which we will denote $F_{DE \to SDF}$, with a single input variable, a single output variable that directly depends on the input, no internal state variables, and a get method which transforms a scalar input value $v \neq \texttt{absent}$ to the list $[v]$ of length 1, and the value absent to the empty list [].

Now, suppose F receives discrete events at its input, and that we want to connect the output of F_B to the input of F. Here, the interpretation would be that every token generated by F_B is mapped to a discrete event. Since F_B generally produces more than one tokens simultaneously (in the case of F_B, 2) we can assume that multiple simultaneous events must be fed into F. We therefore need an FMU $F_{SDF \to DE}$ which takes as input a (possibly empty) list of tokens, and produces as output a sequence of simultaneous discrete events, one per each token in the list.

$F_{SDF \to DE}$ is also easy to define. It has a single input and a single output variable, and a state variable which keeps count of the number of events that the FMU still needs to output, to exhaust the number of tokens it received. Every time $F_{SDF \to DE}$ receives a list of, say k tokens, it sets the counter to k. It then forbids time from advancing (by rejecting time steps when its doStep is called) until k simultaneous discrete events have been produced at the output. If the received list of tokens is empty, then $F_{SDF \to DE}$ outputs absent. The details are omitted.

VI. ENCODING TIMED STATE MACHINES AS FMUS

In Section III we considered untimed state machines and in Section IV we considered timed discrete-event actors. In this section we consider *timed* state machines, which can be seen as a model combining state machines with timed discrete events. Timed state machines have a timed semantics, and therefore there is no need to bridge the untimed-timed semantic gap when encoding them as FMUs. On the other hand, there are many variants of timed state machines, with many different semantics (sometimes not formal). Some of these semantics raise semantic gaps that need to be bridged. For instance, in the case of state machines communicating with timed discrete-events, adding the absent value may be necessary.

We begin with timed automata [8], which is a formal model, and show how a deterministic timed automaton can be encoded as an FMU. We then show examples of other types of timed state machines, similar to those used in languages such as UML, SysML, and Rhapsody Statecharts, and show how they can be encoded as FMUs as well.

A. Timed Automata

We consider timed automata communicating with input and output events. Such a timed automaton is a tuple

$$(E_I, E_O, C, Q, q_0, \mathsf{Inv}, \rhd)$$

where E_I is a set of input events; E_O is a set of output events; C is a finite set of *clocks*; Q is a finite set of *control states*; $q_0 \in Q$ is the initial control state; Inv is a function assigning to each $q \in Q$ a *clock invariant*, explained below; and $\rhd$ is a finite set of *actions*, each being a tuple of the form

$$(q, q', e, g, C')$$

where $q, q' \in Q$ are the source and destination control states; $e \in E_I \cup E_O$ is either an input event or an output event; g is the clock *guard*, explained below; and C' is a subset of clocks to *reset* to 0, $C' \subseteq C$.

Invariants and guards are conjunctions of simple constraints on clocks, of the form $c \leq k$ and $c < k$, where $c \in C$ is a clock and $k \in \mathbb{Z}$ is an integer constant. For clock invariants, we assume only constraints of the form $c \leq n$ where n is a non-negative integer.

The semantics of a timed automaton (TA) is defined as a *timed transition system* (TTS). A TTS is a tuple $(S, s_0, \longrightarrow)$ where S is its state of TA states, s_0 is the initial TA state, and $\longrightarrow$ is its transition relation. A state $s \in S$ is a pair (q, v), where $q \in Q$ is a control state of the TA; and $v : C \to \mathbb{R}_{\geq 0}$ is a *clock valuation*, i.e., a function that assigns a non-negative real value to every clock. The initial state is $s_0 = (q_0, \vec{0})$ where $\vec{0}$ is the clock valuation assigning 0 to every clock in C. Note that, because of the special form of clock invariants, $\vec{0}$ is guaranteed to satisfy $\mathsf{Inv}(q_0)$. The transition relation has two types of transitions:

Discrete transitions of the form $(q, v) \xrightarrow{e} (q', v')$ where $e \in E_I \cup E_I$. Such a transition is possible iff there exists an action (q, q', e, g, C') such that v satisfies the guard g, and v' is obtained from v by resetting all clocks in C' to zero and leaving all others unchanged. That is, $\forall c \in C' : v'(c) = 0$ and $\forall c \in C - C' : v'(c) = v(c)$. We denote v' by $v[C' := 0]$. In addition, it must be the case that v' satisfies the invariant of the destination control state q', written $v' \models \mathsf{Inv}(q')$.

Timed transitions of the form $(q, v) \xrightarrow{\delta} (q, v + \delta)$ where $\delta \in \mathbb{R}_{\geq 0}$ and $v + \delta$ denotes the clock valuation v' defined by $\forall c \in C : v'(c) = v(c) + \delta$. Such a transition is possible iff the invariant at control state q is not violated by the elapse of time, i.e., $v + \delta \models \mathsf{Inv}(q)$. Note that, because of the special form of clock invariants, and the fact that the starting valuation v satisfies $\mathsf{Inv}(q)$ (this can be shown by induction), $v + \delta \models \mathsf{Inv}(q)$ implies that $\forall 0 \leq \delta' \leq \delta : v + \delta' \models \mathsf{Inv}(q)$.

A TA state (q, v) is called *reachable* if there exists a path in the TTS defined by the TA that ends on that state.

Example: An example of a timed automaton modeling a simple controller for a light is shown in Figure 2. The automaton has a single input event, "touch" (a label a? on an action denotes the fact that a is an input event), and three output events, "on", "off", and "bright" (a label b! denotes the fact that b is an output event). The automaton has 6 control states and one clock, c. The states labeled with the clock

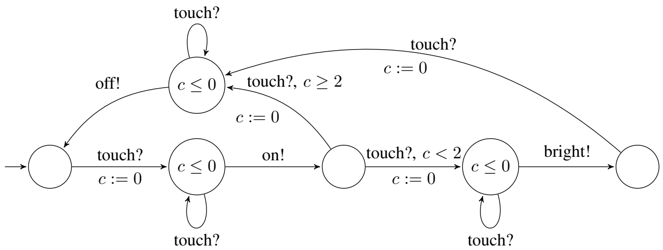

Fig. 2. A timed automaton modeling a light controller.

invariant $c \leq 0$ are transient states, meaning that no time can elapse on those states. This is ensured by the constraint $c \leq 0$ and the fact that c is reset to zero whenever entering such a control state. Clock resets are denoted on the actions by $c := 0$. Control states labeled with no clock invariant implicitly means that the invariant in that state is $true$, i.e., an arbitrary amount of time can elapse in that control state.

The logic of this controller is as follows. Assume that initially the light is off. Touching the lamp once triggers a sensor issuing the input event "touch". This is followed (in zero time) by the light being turned on, at a normal level of brightness. If a second touch follows quickly after the first one (within less than 2 time units, similarly to a double-click on a computer mouse) this is interpreted as the user wanting a brighter light. Otherwise, a touch at a state where the light is on is interpreted as the user wanting to turn the light off.

B. Determinism and other restrictions

Timed automata is a modeling formalism developed primarily with verification in mind. As such, the model is very general, and allows to describe non-deterministic automata, automata with *timelocks* (where time cannot elapse at all) or *zenoness* (where time cannot elapse beyond a certain upper bound), and other phenomena which may be considered problematic in the context of FMI. In particular, non-determinism is a problem, since FMUs are by definition deterministic (in the sense that the methods of the FMI API are mathematically modeled as total functions). Therefore, in order to be able to encode a timed automaton as an FMU, we will impose some restrictions on it.

First, we require that the clock invariants of the automaton are such that, for any reachable state (q, v), and any action (q, q', e, g, C'), whenever the guard g is satisfied by the current clock valuation v, then the invariant of the destination control state q' is also satisfied by the new valuation $v[C' := 0]$ obtained after the reset, i.e., $v[C' := 0] \models \mathsf{Inv}(q')$. We call this the *invariant sanity condition*. The condition that the next state of an FMU is always well defined. This condition forbids, for instance, an automaton such as the one shown in Figure 3 (left), where at the initial state, when clock $c = 2$, say, the action to the middle state is enabled, but the clock invariant of that state, $c \leq 1$, is violated. A simple fix is shown to the right of the figure.

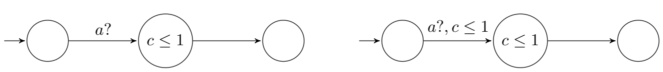

Fig. 3. A pathological timed automaton (left); fixed version (right).

Second, we require that the automaton is *receptive* meaning that it is able to accept any input event at every reachable state. Formally, for every reachable state (q, v) and every input event $e \in E_I$, the TTS of the automaton must have a discrete transition $(q, v) \xrightarrow{e} (q', v')$ to some state (q', v'). This condition removes ambiguity about what to do when an input event is received by an FMU, in cases where there is no corresponding outgoing action labeled with that input in the automaton. Note that, in order to ensure this condition, we had to add self-loops labeled with "touch?" at all transient control states of the light controller automaton of Figure 2.

Third, we require that the automaton is *deterministic*. Intuitively, we want two things. First, for any reachable state, and any input event, we want the automaton to have a uniquely defined successor state when (and if) it receives that event. Second, if the automaton decides to produce an output event, we want the output event to be unique, *but also the timed at which it is produced to be uniquely defined*. These are several conditions, and somewhat tricky to get right, therefore, we proceed step by step.

We say that a TA is *input-deterministic* if for every reachable state (q, v) and every input event $e \in E_I$, there is at most one state (q', v') such that $(q, v) \xrightarrow{e} (q', v')$. Notice that input-determinism together with receptiveness, imply that there is a unique successor state (q', v').

We say that a TA is *output-deterministic* if for every reachable state (q, v) and every output event $e \in E_O$, if there exists a state (q', v') such that $(q, v) \xrightarrow{e} (q', v')$, then the following conditions hold:

1) There is no $e' \in E_O$ such that $e' \neq e$ and $(q, v) \xrightarrow{e'} (q'', v'')$ for some (q'', v'').
2) There is no $\delta \in \mathbb{R}_{\geq 0}$ such that $\delta > 0$ and $(q, v) \xrightarrow{\delta} (q, v + \delta)$.

The first condition says that if the automaton decides to output e, then it doesn't also have a transition with a different output event e'. The second condition says that if the automaton decides to output e, then time cannot elapse. This forbids an ambiguity in the FMU of the form "should we output something now, or should we wait?"

An example of a TA which violates output-determinism is shown to the left of Figure 4. This automaton is problematic for two reasons. First, it doesn't specify precisely at what time in the interval $[0, 1]$ the output event a should be issued. Second, it doesn't even specify whether output a should be issued at all. Indeed, since there is no clock invariant at the initial control state, time can in principle elapse beyond $c > 1$ without the automaton issuing any output. A fixed version of the automaton is shown to the right of the figure. Here, a clock invariant is imposed so that time cannot elapse beyond $c = 1$. Moreover, the guard $c = 1$ at the output action ensures that the output event is issued precisely at time 1.

Fig. 4. An output-nondeterministic timed automaton (left); fixed version (right).

One might think that together input and output-determinism give us what we want, but there is a subtlety. Consider the example shown in Figure 5 (left). Suppose the automaton is at its initial control state with $c = 1$, and there is input event b present at that time. What should the automaton do? Should it output event a, or should it take the discrete transition labeled b?

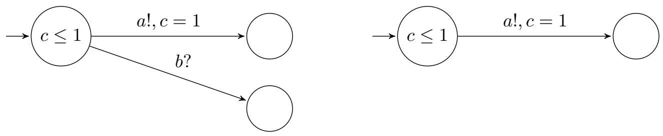

Fig. 5. A nondeterministic timed automaton (left); fixed version (right).

To avoid such ambiguities, we impose a further condition. We say that a TA is *deterministic* if it is input-deterministic, output-deterministic, and in addition, for every reachable state (q, v) which is an *output state*, that is, which has a transition $(q, v) \xrightarrow{e} (q', v')$ with $e \in E_O$, the following condition holds:

- For any $e \in E_I$ there is a transition $(q, v) \xrightarrow{e} (q, v)$.

The last condition, together with input-determinism, ensures that when an output event is ready to be issued, all input events are ignored (i.e., are consumed without a change of state).

Even with the above restrictions, users can still design timed automata which may appear pathological. An example is shown in Figure 6. This TA is *zeno* in the sense that it produces an infinite number of output events a in zero time. However, this automaton satisfies all our conditions above, so we do not forbid it (meaning we are able to encode it as an FMU). We prefer not to impose additional restrictions forbidding such automata, in order not to limit the users' modeling options.

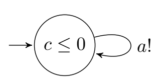

Fig. 6. A zeno timed automaton.

C. Encoding timed automata as FMUs

We assume given a TA $(E_I, E_O, C, Q, q_0, \mathsf{Inv}, \rhd)$ which satisfies the clock invariant sanity condition, receptiveness, and determinism. We encode this TA as an FMU $F = (S, U, Y, D, s_0, \mathtt{set}, \mathtt{get}, \mathtt{doStep})$, where:

- F has $n+1$ state variables where $n = |C|$ is the number of clocks in the TA. F has a state variable q ranging over Q, and a state variable x_i ranging over $\mathbb{R}_{\ge 0}$, for every clock $c_i \in C$, $i = 1, ..., n$. Note that with these state variables, a state s of F has the same form as a state of the original TA, i.e., s can be viewed as a pair (q, v) where v is a clock valuation.
- F has a single input variable u, ranging over $E_I \cup \{\mathtt{absent}\}$.
- F has a single output variable y, ranging over $E_O \cup \{\mathtt{absent}\}$.
- $D = \{(u, y)\}$, i.e., y depends on u.
- The initial state s_0 of F is such that q is set to q_0, every x_i is set to 0, and u and y are set to some arbitrary value.
- `set` sets the input variable u to a given value. This value is either an input event in E_I, or `absent`.
- `get` behaves as follows, depending on the current state (q, v) of the FMU.
    - If (q, v) is an output state of the TA, that is, there exists $e \in E_O$ and (q', v') such that $(q, v) \xrightarrow{e} (q', v')$, then `get` sets the output variable y to e. Note that determinism ensures that both e and (q', v') are unique.
    - Otherwise, `get` sets the output variable y to `absent`.

The intuition behind the definition of `get` is that if an outgoing transition is enabled at a given state, and this transition emits an output event, then `get` returns this event, otherwise it returns `absent`. Note that the output event can be seen as emitted before (or simultaneously with, but not after) the transition takes place. Indeed, taking the transition happens only after `doStep`, whereupon the automaton also changes state.

`doStep`(h) behaves as follows, again depending on the current state (q, v) of the FMU:

- If (q, v) is an output state of the TA, then let $e \in E_O$ and (q', v') be the uniquely defined output event and successor state such that $(q, v) \xrightarrow{e} (q', v')$.
    - If $h = 0$ then `doStep` accepts h, sets $q := q'$, $x_i := v'(c_i)$, for $i = 1, ..., n$, and returns 0. This zero-time step corresponds to taking the discrete transition.
    - If $h > 0$ then `doStep` rejects h, but again sets $q := q'$, $x_i := v'(c_i)$, for $i = 1, ..., n$, and returns 0. Note that the FMU behavior here is identical to the previous case, except that the interpretation accepts/rejects is different. Here the FMU is asked to perform a non-zero time step, but cannot, since a discrete transition must first take place, and this happens in zero time (instantaneously).

  Note that `doStep` ignores any input event which might be present at input variable u in this case. This does not violate the TA semantics, thanks to the determinism assumption which ensures that such inputs are in any case ignored by the TA (i.e., leave its state unchanged).
- Otherwise:
    - If $u = \mathtt{absent}$ and $v + h \models \mathsf{Inv}(q)$ then `doStep` accepts h and updates all x_i variables to $x_i := x_i + h$. This corresponds to time elapsing by h.
    - If $u = \mathtt{absent}$ and $v + h \not\models \mathsf{Inv}(q)$ then, by the fact that $v \models \mathsf{Inv}(q)$ and the form $c \le k$ of clock invariants, there exists a largest $h' \in \mathbb{R}_{\ge 0}$, such that $0 \le h' \le h$ and $v + h' \models \mathsf{Inv}(q)$. Then, `doStep` rejects h, updates all x_i variables to $x_i := x_i + h'$, and returns h'. This case is similar to the previous, except that time can only elapse by h' instead of h, so as not to violate the urgency constraints.
    - If $u = e$ for some $e \in E_I$, then by the assumptions of receptiveness and determinism, there is a unique successor state (q', v') such that $(q, v) \xrightarrow{e} (q', v')$. Then, `doStep` rejects h, sets $q := q'$, $x_i := v'(c_i)$, for $i = 1, ..., n$, and returns 0. This is the case of a discrete transition triggered by an input event, and occuring as usual in zero time.

D. Ptolemy and Rhapsody state machines

So far in this section we considered the formal model of timed automata. Other variants of timed state machines are also used in tools such as SysML/Rhapsody from IBM and Ptolemy from UC Berkeley (ptolemy.org), to name a few. It is beyond the scope of this paper to show complete and formal encodings of these types of state machines as FMUs, as this would also require formalizing their semantics. Still, it is worth discussing a few examples in order to present the basic principles of how such encodings could be developed.

First, we look at a timed state machine from SysML/Rhapsody, shown in Figure 7. For simplicity, in this example there are only input events, labeled `eventA1`, `eventA2` and `eventA3`, and no outputs. Ignoring for this discussion the mechanism of event-based interaction in SysML/Rhapsody (which is itself non-trivial), let us focus on the timed part of the machine of Figure 7, namely, the *timeout* statement `tm(10)`. This can be intuitively explained as follows. Upon entering `state_2`, set a clock to 0; when the clock reaches 10, the timeout transition to `state_0` is taken, unless in the meantime event `eventA3` occurred, in which case the machine has already moved to `state_1`.

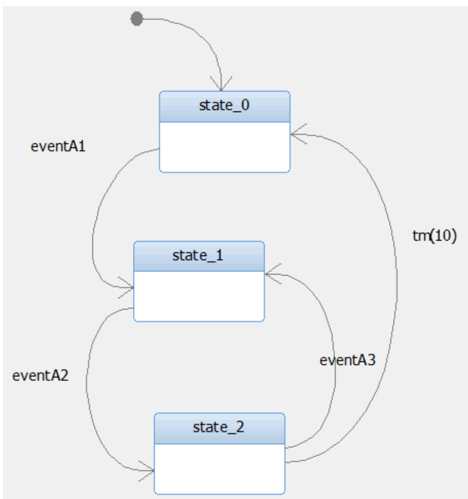

Fig. 7. A Rhapsody state machine with a timeout statement.

This logic looks simple enough to model as a timed automaton. Attempting to do that, we come up with the TA shown in Figure 8. In this automaton, we included a "dummy" output event `TO`, since our current formalization of TA does not allow "silent" actions (i.e., without input nor output events).

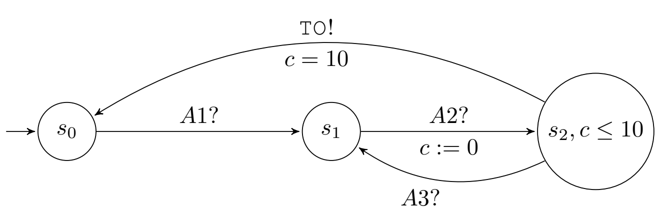

Fig. 8. A timed automaton attempting to model the state machine of Figure 7.

Unfortunately, the TA of Figure 8 suffers from several problems. First, it is not receptive, since for instance, there is not outgoing transition from s_0 labeled with input event $A2$. This problem can be fixed by adding appropriate self-loops as in Figure 9. A second problem is that the TA of Figure 8 does not satisfy the determinism condition. In particular, it is ambiguous what to do in the case where at control state s_2, $c = 10$ and input event $A3$ arrives. A possible fix is shown in Figure 9. This fix is simple enough, but it is unclear whether it captures the semantics of SysML/Rhapsody. Ultimately, defining the mapping from general SysML/Rhapsody state machines to FMUs amounts to defining a semantics for the SysML/Rhapsody language, which beyond the scope of this work.

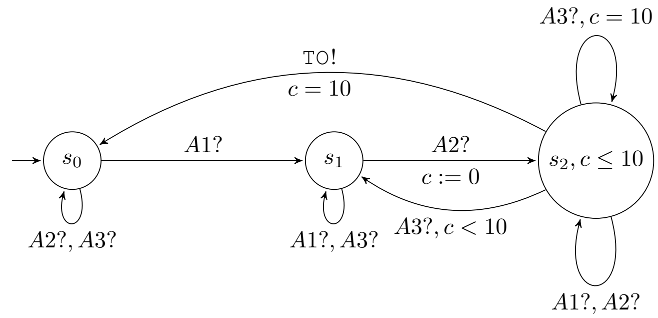

Fig. 9. Fixing the timed automaton of Figure 8 to ensure receptiveness and determinism.

We now turn our attention to state machines in Ptolemy. An example is shown in Figure 10. Ptolemy state machines also have a `timeout` statement as this example illustrates. Ptolemy state machines communicate via various mechanisms. One mechanism is input and output events, as in timed automata and SysML/Rhapsody state machines. The machine of Figure 10 has two input ports and two output ports, all of which carry events. Reacting to an input event a (similar to the label $a?$) is written in Ptolemy as a`_isPresent`. Emitting an output event b is achieved by output actions such as `out1=x` in the transition from `s1` to `s2`. Note that in this case the event emitted carries a value, in this case, the current value of `x`. The latter is a state variable of the machine, which can be set in set actions such as `x=x+1` in the transition from `s1` to `s2`.

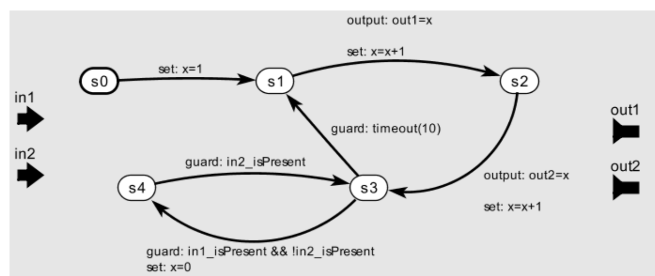

Fig. 10. A Ptolemy state machine with a timeout statement and state variable x.

The machine of Figure 10 has several similarities, but also several differences with the one in Figure 7. Regarding differences, first, the machine of Figure 10 has more than one input ports, and more than one output ports. It also has complex guards on input events, such as the guard of the transition from `s3` to `s4`, which requires an event to be present at `in1` and no event to be present at `in2`. Also, a variable such as `x` could sometimes be observable to the external world, and therefore can be considered an output. Because of this, the machine of Figure 10 can be seen as having not only output events, but also persistent output signals.

These additions can be easily accommodated when encoding the machine of Figure 10 as an FMU. First, the two input ports `in1` and `in2` can be mapped to two input variables, say, u_1 and u_2 in the FMU, and similarly for output ports. The special value `absent` allows to encode the guard `in1_isPresent && !in2_isPresent` without issues, as $u_1 \neq \texttt{absent} \wedge u_2 = \texttt{absent}$. Finally, persistent outputs are the default output mechanism in FMUs, so having an additional output variable for `x` is also straightforward.

VII. Related Work

The FMI standard [13] is currently used by many modeling and simulation tools, both within industry and academia. More than 50 tools have support for FMI[6], where approximately 20 of them support export of version 1.0 models for co-simulation. Although many tools implement ways to generate FMUs from continuous-time models (such as Modelica models), we are not aware of any work that formally describes how to encode FMUs in general, especially for non-continuous-time models like the ones we focus on here. Discussions about the limitations of the current FMI standard can be found in [1], [14]. [14] also defines a suite of test models that should be supported by a hybrid co-simulation environment, giving a mathematical model of an ideal behavior, plus a discussion of practical implementation considerations.

Ptolemy II [3] is an environment for composing and simulating heterogenous concurrent components. The components in Ptolemy II are implemented in Java and called *actors* and have similar structures as FMUs, but with a different interface. A formalization of Ptolemy's actor interface and encodings of various models of computations have been proposed in [9]. Although some of the principles of encoding actors as FMUs (e.g., the use of `absent` in modeling events as signals) are borrowed from [9], the encodings themselves are different, because the FMI API is very different from the Ptolemy API.

There exist works that describe how FMUs can be used in existing modeling environments and how master algorithms may be implemented. Bastian *et al.* [15] have proposed a fixed-step size MA that is designed to be platform independent. Schierz *et al.* [16] describe a strategy for adaptive communication size control. Feldman *et al.* [17] present a plugin for Rhapsody for generating FMUs from Statechart SysML blocks. They provide high level guidelines for how to generate Statechart FMUS, but do not provide a formalization. Pohlmann *et al.* [18] also describe how to encode statechart models, described in MechatronicUML. Their paper is also discussing implementation strategies, but the implementation is for FMI for model exchange and not co-simulation.

VIII. Conclusion

In this paper, we show how various models of computation, such as state machines, discrete-event, dataflow, and timed automata, can be encoded as functional mock-up units (FMUs), which are model components implementing the FMI standard. The main challenge is the gap between the semantics of the source formalism and the semantics of FMI. Faced with this problem, we show how to overcome the semantic gaps from untimed to timed models, from events to persistent signals, and from asynchronous queues to persistent signals. Combined with [1], this work offers a complete methodology for heterogeneous modeling and cosimulation, consisting of (1) modeling subsystems using different modeling tools, (2) generating FMUs corresponding to each submodel as shown in this paper, and (3) co-simulating the set of resulting FMUs using the results of [1].

We believe that the main challenge regarding FMI (as well as any other co-simulation framework) is to validate the results of co-simulation with respect to simulation in a single environment. Ideally, one would like the co-simulation results to be identical to those produced if one were to model and simulate the entire system in a single tool. Towards performing such a validation experimentally, we are currently working on an implementation of FMI co-simulation and FMU generation in Ptolemy II.

[6]http://www.fmi-standard.org/tools

References

[1] D. Broman, C. Brooks, L. Greenberg, E. A. Lee, S. Tripakis, M. Wetter, and M. Masin, "Determinate Composition of FMUs for Co-Simulation," in *Embedded Software (EMSOFT)*, 2013.

[2] Z. Kohavi, *Switching and finite automata theory*. McGraw-Hill, 1978.

[3] J. Eker, J. Janneck, E. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, "Taming heterogeneity – the Ptolemy approach," *Proc. of the IEEE*, vol. 91, no. 1, pp. 127–144, Jan. 2003.

[4] E. A. Lee, "Modeling concurrent real-time processes using discrete events," *Ann. Softw. Eng.*, vol. 7, no. 1-4, pp. 25–45, 1999.

[5] S. Tripakis and D. Broman, "Bridging the Semantic Gap Between Heterogeneous Modeling Formalisms and FMI," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-30, Apr 2014.

[6] E. A. Lee, "Constructive models of discrete and continuous physical phenomena," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-15, Feb 2014.

[7] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice, "Lustre: a declarative language for programming synchronous systems," in *POPL*, 1987.

[8] R. Alur and D. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, pp. 183–235, 1994.

[9] S. Tripakis, C. Stergiou, C. Shaver, and E. A. Lee, "A modular formal semantics for Ptolemy," *Mathematical Structures in Computer Science*, vol. 23, pp. 834–881, Aug. 2013.

[10] C. Stergiou, S. Tripakis, E. Matsikoudis, and E. A. Lee, "On the Verification of Timed Discrete-Event Models," in *Formal Modeling and Analysis of Timed Systems (FORMATS)*. Springer, 2013.

[11] E. Lee and D. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.

[12] G. Kahn, "The semantics of a simple language for parallel programming," in *IFIP Congress*, 1974.

[13] MODELISAR Consortium and Modelica Association, "Functional Mock-up Interface for Model Exchange and Co-Simulation," October 18, 2013, 2.0 Release Candidate 1, from https://www.fmi-standard.org.

[14] D. Broman, L. Greenberg, E. A. Lee, M. Masin, S. Tripakis, and M. Wetter, "Requirements for Hybrid Cosimulation Standards," in *Hybrid Systems: Computation and Control (HSCC)*, 2015.

[15] J. Bastian, C. Clauss, S. Wolf, and P. Schneider, "Master for Co-Simulation Using FMI," in *8th Modelica Conference*, 2011, pp. 115–120.

[16] T. Schierz, M. Arnold, and C. Clauss, "Co-simulation with communication step size control in an FMI compatible master algorithm," in *9th Modelica Conference*, 2012, pp. 205–214.

[17] Y. A. Feldman, L. Greenberg, and E. Palachi, "Simulating Rhapsody SysML Blocks in Hybrid Models with FMI," in *10th Modelica Conference*, 2014, pp. 43–52.

[18] U. Pohlmann, W. Schäfer, H. Reddehase, J. Röckemann, and R. Wagner, "Generating Functional Mockup Units from Software Specifications," in *9th Modelica Conference*, 2012, pp. 765–774.