David Broman, Lev Greenberg, Edward A. Lee, Michael Masin, Stavros Tripakis, and Michael Wetter. Requirements for Hybrid Cosimulation Standards. In Proceedings of 18th ACM International Conference on Hybrid Systems: Computation and Control (HSCC 2015), Pages 179-188, CPSWeek, Seattle, WA, USA, 2015.

Requirements for Hybrid Cosimulation Standards*

David Broman
KTH Royal Institute of Technology & UC Berkeley

Lev Greenberg
IBM Research – Haifa, Israel

Edward A. Lee†
UC Berkeley

Michael Masin
IBM Research – Haifa, Israel

Stavros Tripakis
UC Berkeley & Aalto University

Michael Wetter
Lawrence Berkeley National Laboratory

ABSTRACT

This paper defines a suite of requirements for future hybrid cosimulation standards, and specifically provides guidance for development of a hybrid cosimulation version of the Functional Mockup Interface (FMI). A cosimulation standard defines interfaces that enable diverse simulation tools to interoperate. Specifically, one tool defines a component that forms part of a simulation model in another tool. We focus on components with inputs and outputs that are functions of time, and specifically on mixtures of discrete events and continuous time signals. This hybrid mixture is not well supported by existing cosimulation standards, and specifically not by FMI 2.0, for reasons that are explained in this paper. The paper defines a suite of test components, giving a mathematical model of an ideal behavior, plus a discussion of practical implementation considerations. The discussion includes acceptance criteria by which we can determine whether a standard supports definition of each component. In addition, we define a set of test compositions that define requirements for coordination between components, including consistent handling of timed events.

1. INTRODUCTION

FMI (Functional Mock-up Interface) is an evolving standard for composing model components designed using distinct modeling and simulation tools [17]. The standard consists of a C API for simulation components and an XML schema for describing components. An **FMU** (Functional Mock-up Unit) is a component, typically exported from a modeling and simulation tool, that can be instantiated and used as part of a simulation in another modeling tool. To date, the emphasis of the standard has been on components that model the dynamics of physical systems, such as mechanical and electrical components. This emphasis reflects the origins of FMI as a way to achieve interoperability of simulators for models of automotive suppliers and OEMs [5].

FMI provides two distinct mechanisms for interaction between an FMU and a host simulator: i) **model exchange**, where the host simulator is responsible for all numerical integration methods, and ii) **cosimulation**, where the FMU implements its own mechanisms for advancing the values of state variables. In FMI for cosimulation, which is the focus of this paper, the host simulator provides input values to the FMU, requests that the FMU advance its state variables and output values in time, and then queries for the updated output values. In the FMI 2.0 standard [17], both of these mechanisms, but most particularly cosimulation, are optimized for simulating continuous dynamics.

However, the current standard for cosimulation is not designed for **reactive systems** [9], only for loose coupling between continuous integration algorithms, where time can advance somewhat independently within the FMU and the host simulator. For example, the current version of the standard provides no mechanism for a cosimulation FMU to output an instantaneous reaction to a changed input value. As a consequence, the community-driven standardization process is considering a third mechanism called **hybrid cosimulation** that strives for the loose coupling of cosimulation, but with support for discrete and discontinuous signals and instantaneous events. The intent of this mechanism is to support hybrid systems [1, 7, 15, 19, 23], where continuous dynamics are combined with discrete mode changes and discrete events. Designing a rigorous, deterministic, and sound hybrid cosimulation API is, however, a non-trivial task. In particular, it is vital to clearly and formally define what the requirements for such an API should be.

In this paper, we provide a series of formally defined test components that must be supported by any comprehensive mechanism that supports hybrid systems. These components are chosen to comprehensively cover the space of functionality needed to effectively model mixed continuous and discrete systems, and to be as simple as possible. More specifically, we make the following contributions:

- We provide principles and introduce formal notations for giving a precise mathematical formalization of the test

*Acknowledgments: This work was supported in part by the iCyPhy Research Center (Industrial Cyber-Physical Systems, supported by IBM and United Technologies), the Assistant Secretary for Energy Efficiency and Renewable Energy, Office of Building Technologies of the U.S. Department of Energy, under Contract No. DE-AC02-05CH11231, the Academy of Finland, the National Science Foundation (#1446619 (Mathematical Theory of CPS), #1329759 (COSMOI), and #0931843 (ActionWebs)), and the Swedish Research Council #623-2013-8591.

†Corresponding author, eal@eecs.berkeley.edu

HSCC '15 April 14-16, 2015, Seattle, WA, USA
Copyright 2015 ACM 978-1-4503-3433-4/15/04
http://dx.doi.org/10.1145/2728606.2728629 ...$15.00

components (Section 2).
- We define and formalize a nearly minimal set of test components that are critical for covering a hybrid co-simulation standard (Section 3).
- We provide a set of test cases that combines some of the test components. These test cases demonstrate necessary constraints for a co-simulation master algorithm that is coordinating the execution of components (Section 4).

All of the mechanisms presented in this paper are realized in Ptolemy II. What we are working on is a C version for FMI, possibly with different choices about representation of time. Ptolemy is a cosimulation engine, of a sort, since it supports mixing distinct directors. There are many ways to meet the requirements given here, and an analysis of the alternatives would take this paper in a very different direction.

2. PRINCIPLES

2.1 Definitions

We adopt the following definitions: A **host simulator** is a tool that imports a modeling component (an FMU) that is either written by hand or exported from another tool (or possibly even the same tool). The **master algorithm** is the execution procedure and policy by which the host simulator invokes the interface procedures of a component (an FMU). A **communication point** is the simulation time at which the master algorithm invokes an interface procedure of an FMU. A **deterministic FMU** is one where the output values and states are uniquely defined given initial conditions, input values, and communication points. A **deterministic composition of deterministic FMUs** is one where for a valid sequence of communication points, given initial conditions and inputs from outside the composition, the values of outputs of the deterministic FMUs are uniquely defined. Note that practical implementations may approximate those values using imprecise numerical methods, but the composition is still deterministic if the ideal correct values are uniquely defined. The definition of determinism is a bit subtle. See [12] for a rigorous definition.

2.2 Assumptions

We assume the following principles:

- We prefer a weaker contract over a stronger contract. That is, we prefer fewer constraints on the design of FMUs and master algorithms. This will maximize interoperability of FMUs and master algorithms.
- We assume superdense time and piecewise continuous signals (see below and [13]). This is necessary for rigorous modeling of discontinuities and discrete signals and is already included in FMI 2.0 for model exchange. .
- The specification should enable, but not require, efficient execution.

2.3 Notation

We use a particular mathematical notation to define the test cases in this paper. This notation is not intended to be used explicitly in the FMI specification for hybrid cosimulation. It is a mathematical idealization of what would be realized in an FMU and the host simulator.

The set $T = \mathbb{R}_+ \times \mathbb{N}$ represents time, where $\mathbb{R}_+$ is the set of non-negative real numbers, and $\mathbb{N} = \{0, 1, 2, \cdots\}$ is the set of natural numbers. A **superdense time** $\tau \in T$ is two-tuple, $\tau = (t, n)$, where the real number t represents a time in the usual Newtonian sense, and n is the **microstep**, which indexes sequences of values at Newtonian time t. Every **communication point** is a member of the set T.

T is a totally ordered set, where for any $\tau_1, \tau_2 \in T$ where $\tau_1 = (t_1, n_1)$ and $\tau_2 = (t_2, n_2)$, then $\tau_1 > \tau_2$ if either $t_1 > t_2$, or $t_1 = t_2$ and $n_1 > n_2$. Otherwise, $\tau_1 \leq \tau_2$.

A **signal** x is a function of the form $x\colon T \to \mathbb{R} \cup \{\varepsilon\}$, where ε represents the absence of a value. In the ideal, the signal is total, defined at all T, but in a simulation, signal values will be computed only at a finite subset of values of T. Note that the FMI specification will need to deal with data types other than reals as well, but we assume here that those data types simply match what is provided by FMI 2.0. There is no need for a hybrid cosimulation standard to deviate from the existing standard in this regard. Also note that nothing in this paper requires that FMI include any explicit representation of ε. It is a semantical concept.

A **continuous-time** (**CT**) signal is one that has a non-absent value for all $\tau \in T$. A **discrete-event** (**DE**) signal is one that has a non-absent value at only some $\tau \in T$. Specifically, following [13], a DE signal x has a non-absent value $x(\tau)$ only for $\tau \in D \subset T$, where D is a **discrete set**.[1]

A signal is **discontinuous** at any time $t \in \mathbb{R}$ if there exist $n, m \in \mathbb{N}$ such that $x(t, n) \neq x(t, m)$.

The **initial-value signal** x_i for a signal x is a function of the form $x_i\colon \mathbb{R}_+ \to \mathbb{R} \cup \{\varepsilon\}$ given by $x_i(t) = x(t, 0)$ for all $t \in \mathbb{R}_+$. At any $t \in \mathbb{R}_+$, the **final microstep** m_t of a signal x is a number $m_t \in \mathbb{N}$ such that for all $m > m_t$, $x(t, m) = x(t, m_t)$. The **final value** at time t is $x(t, m_t)$. If for any $t \in \mathbb{R}_+$, x has no final microstep, then x is said to be a **chattering Zeno** signal. It has a Zeno condition at time t, where it has an infinite sequence of changing values. Put another way, for a chattering Zeno signal, there exists a time where the signal does not settle to a final value.

The **final-value signal** x_f for a non-chattering Zeno signal x is a function of the form $x_f\colon \mathbb{R}_+ \to \mathbb{R} \cup \{\varepsilon\}$ given by $x_f(t) = x(t, m_t)$ for all $t \in \mathbb{R}_+$, where m_t is the final microstep at time t.

A **continuous signal** is a CT signal where for all $t \in \mathbb{R}_+$, $m_t = 0$ and x_i is continuous at t (in the usual sense for functions of reals). A **piecewise-continuous CT signal** is one where

1. $m_t = 0$ for all $t \in \mathbb{R}_+$, except $t \in D$, where $D \subset \mathbb{R}_+$ is a discrete set.
2. x_i is left-continuous for all t (in the usual sense for functions of reals).
3. x_f is right-continuous for all t.

We can extend the notion of a piecewise-continuous signal to include DE signals and signals that are neither CT nor DE (they are absent over some intervals and present over others). A **piecewise-continuous signal** is one where

1. $m_t = 0$ for all $t \in \mathbb{R}_+$, except $t \in D$, where $D \subset \mathbb{R}_+$ is a discrete set.
2. If $x_i(t) \neq \varepsilon$, then x_i is left-continuous at t (in the usual sense for functions of reals).
3. If $x_i(t) = \varepsilon$, then there exists a $\delta > 0$ such that for all $0 \leq \epsilon < \delta$, $x_i(t - \epsilon) = \varepsilon$.
4. If $x_f(t) \neq \varepsilon$, then x_f is right-continuous at t (in the usual sense for functions of reals).

[1] A **discrete set** is an ordered set that is order isomorphic with a subset of the natural numbers [13].

5. If $x_f(t) = \varepsilon$, then there exists a $\delta > 0$ such that for all $0 \leq \epsilon < \delta$, $x_f(t + \epsilon) = \varepsilon$.

This simply extends the usual notion of left and right continuity to absent values.

A **well formed simulation** models a system where all signals are piecewise continuous. Every piecewise-continuous signal has a well-defined (possibly empty) sequence of times $d_0, d_1, d_2, \cdots$, ordered in time, at which it is discontinuous. An interval between these times, $t \in (d_i, d_{i+1})$, is called a **continuous interval** of the signal.

Because of the desirable properties of piecewise-continuous signals, the components in this paper all have the property that if all their inputs are piecewise continuous, then all their outputs are piecewise continuous. If they have no inputs, then the outputs are by construction piecewise continuous.

Note that for a DE signal x to be piecewise continuous, we need $x_i(t) = x_f(t) = \varepsilon$ for all $t \in \mathbb{R}_+$. This is an important property that enables numerical integrators to interoperate cleanly with discrete events. Numerical integrators will only deal with initial and final-value functions. For a DE signal, these signals are absent everywhere, and therefore such signals have no effect on numerical integration. Hence, such signals can be used to represent cyber events in cyber-physical systems. For such signals to affect the physical dynamics, they can be first converted to CT signals, for example using the Zero-Order Hold component described below.

2.4 Time

The specifications in this paper use real numbers to represent time. Computer programs typically approximate real numbers using floating-point numbers, which can create problems. Specifically, real numbers can theoretically be compared for equality, but it rarely makes sense to do so for floating point numbers. Hence, approximation and a specified error tolerance will be required in practice.

Several of the components described below specify output events that occur at specific points in time. When composing components, it will be beneficial if the view of time across components is consistent. For example, if two components produce periodic events with the same period, then it should be assured that those events will appear simultaneously at any other component that observes them. Periods that are simple multiples of one another should also yield simultaneous events. Quantization errors should not be permitted to weaken this property. In short, a useful hybrid cosimulation standard should provide a model of time with a sound notion of simultaneity, something that can be provided with the following properties:

1. The precision with which time is represented should be finite and should be the same for all observers in a model. Infinite precision (as provided by real numbers) is not practically realizable in computers, and if precisions differ to different observers, then the different observers will not agree on which events are simultaneous.
2. The precision with which time is represented should be independent of the absolute magnitude of the time. In other words, the time origin (the choice for the meaning of time zero) should not affect the precision.
3. Addition of time should be associative. That is, for any three time intervals t_1, t_2, and t_3,

$$(t_1 + t_2) + t_3 = t_1 + (t_2 + t_3).$$

The last two of these three properties are not satisfied by floating-point numbers, so floating-point numbers should not be used as the primary representation for time. The first property implies that the precision of the representation of time should be a global property of a composition of components, not a property of individual components. For a practical implementation of a model of time that satisfies all three properties, see [21].

3. TEST COMPONENTS

In this section, we define a suite of test components[2] to be a (nearly) minimal set that represents hybrid (mixed discrete and continuous) behaviors, plus a small set of components required to create useful test cases for composition of components, described in Section 4 below. The test components vary wildly in sophistication, with some being quite trivial and some quite subtle. In each case, we give a Platonic ideal, a mathematical description of idealized behavior, which will be approximated by any real implementation (an FMU).

3.1 Constant Signal Generator

Output signal y. **Real parameter** c.

$$\text{For all } \tau \in T, \qquad y(\tau) = c \tag{1}$$

Discussion. As with many test cases here, this component provides an output value at all values τ of superdense time. Since there are an uncountably infinite number of such values, no computer execution of this component can actually provide output values at all such times. An FMU and host simulator for this test case is deemed correct if at every point τ in superdense time where the host simulator chooses to observe the output of this FMU (a communication point), then the value of that output will be c. This test case imposes no constraints on when such observations are made.

3.2 Gain

Input signal x. **Output** signal y. Real **parameter** a.

$$\text{For all } \tau \in T, \qquad y(\tau) = \begin{cases} ax(\tau) & \text{if } x(\tau) \neq \varepsilon \\ \varepsilon & \text{otherwise} \end{cases} \tag{2}$$

Discussion. How this component handles absent inputs is an essential part of the definition. As defined, given a DE input, the output will be DE.

With the definition in (2), it may be reasonable for an FMU to require that at every communication point τ for this FMU, $x(\tau) \neq \varepsilon$. That is, the master algorithm should not invoke interface procedures of this FMU except at times when the input is present. This would make handling absent inputs extremely efficient. The FMU would need to do nothing at all to handle them. The output will be absent if the input is absent because the FMU will not be invoked to make the output non-absent. To support this, the hybrid cosimulation standard would need to provide a way for an FMU to declare that it requires all input to be present when invoked. Or conversely, an FMU may indicate, as part of its interface definition, that it can react even if some (or all) inputs are absent. We do not include in this paper any such requirements, however, because we prefer weaker contracts over stronger ones, and we are focused on a minimal

[2]Note that this set is only instead to represent the capabilities that are needed for hybrid cosimulation, not to be part of a comprehensive tool test suite.

set of requirements, and not on making cosimulation more efficient.

Note that the current 2.0 FMI standard for cosimulation cannot realize this component, even for CT inputs only. We believe that there is no implementation of this component consistent with this constraint that supports the calling sequences defined in page 104 of the standard document [17]. Specifically, the standard states, "there is the additional restriction ... that it is not allowed to call fmi2GetXXX functions after fmi2SetXXX functions without an fmi2DoStep call in between." In addition, when fmi2DoStep is called, the standard requires that the step size be greater than zero (p.100: "... communicationStepSize ... must be > 0.0"). Hence, time must advance between setting the inputs and reading the resulting outputs.

3.3 Adder

Input signals x_1 and x_2. **Output** signal y.

For all $\tau \in T$,

$$y(\tau) = \begin{cases} x_1(\tau) + x_2(\tau) & \text{if } x_1(\tau) \neq \varepsilon \text{ and } x_2(\tau) \neq \varepsilon \\ x_1(\tau) & \text{if } x_1(\tau) \neq \varepsilon \text{ and } x_2(\tau) = \varepsilon \\ x_2(\tau) & \text{if } x_1(\tau) = \varepsilon \text{ and } x_2(\tau) \neq \varepsilon \\ \varepsilon & \text{otherwise} \end{cases} \tag{3}$$

Discussion. This component illustrates that an FMU may be presented at a communication point with some inputs that are absent and some that are not, and that its behavior may depend on which inputs are present. Of course, a simpler Adder component would require all inputs to be present simultaneously, or would use previous input values if an input is not present. We can certainly design such Adder components, and indeed a library for simulation might prefer those semantics. But our goal here is to test capabilities that may be required in hybrid cosimulation, and reacting differently to different patterns of presence of inputs most certainly will be required.

This component imposes no constraints on the communication points, but points of discontinuity of the inputs must be presented as communication points in order for the output to reflect the ensuing discontinuity.

3.4 Periodic Piecewise Constant Signal Generator

CT **output** signal y. Real **parameters** a, b, p.

Informally, this component outputs the constant a from time zero to p, b from time p to $2p$, a from $2p$ to $3p$, etc., alternating between a and b, as illustrated in Figure 1.

We require that the output be piecewise continuous. Specifically, for all $\tau = (t, n) \in T$,

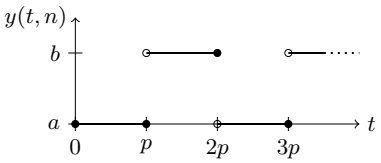


Figure 1: Example output from the Periodic Piecewise Constant component. The unfilled dots show values that occur only at microsteps $n \geq 1$, whereas the filled dots and lines show values at $n = 0$.

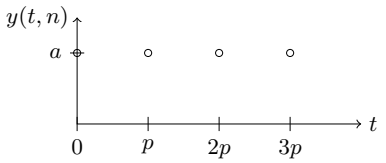


Figure 2: Example output from the Periodic Discrete Signal Generator. The unfilled dots are the only non-absent values, and they occur only at microstep $n = 1$.

$$y(t,n) = \begin{cases} a & \text{if } kp < t < (k+1)p \text{ and } k \in \mathbb{N} \text{ is even;} \\ b & \text{if } kp < t < (k+1)p \text{ and } k \in \mathbb{N} \text{ is odd;} \\ b & \text{if } t \text{ is an odd multiple of } p \text{ and } n \geq 1; \\ b & \text{if } t \text{ is an even multiple of } p,\ t > 0,\ \ n = 0; \\ a & \text{otherwise.} \end{cases}$$

Discussion. A correct implementation of this component and host simulator will produce at least the output values shown in Figure 1 as filled and unfilled dots. Hence, in a correct implementation, two communication points can be used at each multiple of p, one at microstep zero and one at microstep one. A host simulator may choose to invoke the FMU implementation at additional communication points, but this is not required. Typically a host simulator will use a step-size adjustment algorithm to choose communication points.

3.5 Periodic Discrete Signal Generator

DE **output** signal y. Real **parameters** a, p.

This component outputs the constant a at integer multiples of p (see Figure 2), and otherwise its output is absent. To be piecewise continuous, the output signal should be absent for all $(t, n) \in T$ where $n = 0$ or $n > 1$. Specifically, for all $\tau = (t, n) \in T$,

$$y(t,n) = \begin{cases} a & \text{if } t = kp \text{ and } n = 1, \text{ where } k \in \mathbb{N}; \\ \varepsilon & \text{otherwise.} \end{cases}$$

Discussion. This component provides a canonical source for a DE signal. It can be used to build regression tests to verify, for example, that the Gain component above behaves correctly with DE inputs. It can also be used to provide discrete inputs to any of the test cases below that require discrete inputs.

This component requires that there be a communication point at all $t = (kp, 1)$, $k \in \mathbb{N}$. Communication points at other times are not required, but if the host simulator provides them, then this component will produce no output (its output will be absent).

3.6 Modal Model with Discrete Control

DE **input** signal x. CT **output** signal y. Real **parameters** a, b.

This component initially outputs the constant a. When the first input event arrives, it switches to producing output b. When the second input event arrives, it switches back to a. Etc. Formally,

$$y(t,n) = \begin{cases} a & \text{if } s(t,n) = 0 \\ b & \text{otherwise.} \end{cases} \tag{4}$$

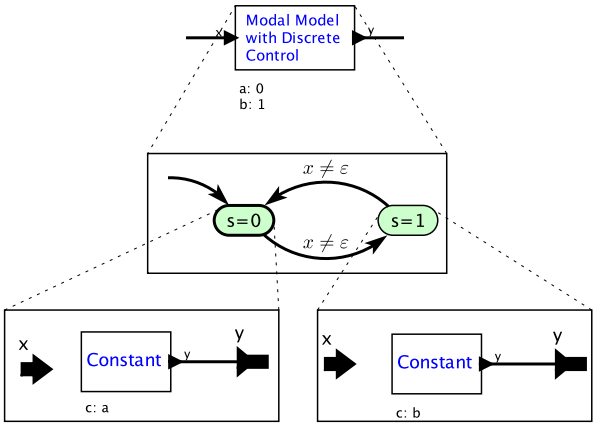


Figure 3: Modal Model with Discrete Control.

where s is a CT signal such that $s(t, n)$ is the **state** of the component at time (t, n), defined as follows. Let $d_0, d_1, d_2, \cdots$ denote the superdense times τ, in temporal order, at which $x(\tau) \neq \varepsilon$. We are assured that such a sequence, which may be empty, finite, or infinite, exists, because x is a DE signal. At any superdense time $(t, n) \in T$, there may exist a maximum $i \in \mathbb{N}$ such that $(t, n) > d_i$. If no such i exists, then either there are no events at all in x or $(t, n) \leq d_0$. Hence, at time (t, n), the state s is given by

$$s(t,n) = \begin{cases} 0 & \text{if no such } i \text{ exists} \\ 1 & \text{if } s(d_i) = 0 \\ 0 & \text{if } s(d_i) = 1 \end{cases} \tag{5}$$

In this definition, at time (t, n), d_i, if it exists, is the time of the most recent but strictly earlier input event. Hence, at time (t, n), $s(d_i)$ is the *previous* state of the component. The *current* state is always the opposite of the previous state.

In words, the state s is initially 0. Each time an input event arrives, the state toggles from 0 to 1 or from 1 to 0. The toggle occurs *strictly after* the input event arrives. So when an input event arrives at time (t, n), the output depends on the current state $s(t, n)$ as given by (4), and then, strictly later, the state is updated as in (5).

An illustration of such a component is given in Figure 3 using a hierarchical modal model in the style of [13]. The logic of the component is given as a **finite state machine** (FSM) (middle level) with two states, each with a mode refinement that outputs a constant.

Discussion. The state of this component is changed after each non-absent input. Since the input is required to be a DE signal, the state changes occur only at a discrete subset of T. Hence, the state updates are enumerable in temporal order, and hence computable.

The state of this component is changed *after* producing an output value. This ensures that the output is piecewise continuous, assuming the input is piecewise continuous.

Notice that if we combine this component with a Periodic Discrete Signal Generator to drive its input, then we can implement the Periodic Piecewise Constant Signal Generator. Nevertheless, we keep the Periodic Piecewise Constant Signal Generator in the suite of test cases to make it easier for readers to understand the progression of capabilities.

The only constraint that this component imposes on communication points is that there be a communication point at each superdense time $d_i = (t, n)$ where the input is not absent and one superdense time later $(t, n + 1)$. At (t, n), the output will be determined by the current state, whereas at time $(t, n+1)$, the output will be determined by the next state. If $a \neq b$, then this creates a discontinuity, but the output remains piecewise continuous.

Once we can support this component and the others included here, we can implement a broad class of **hybrid systems** [15], including **timed automata** [2] and many others.

3.7 Integrator

CT **input** signal x. CT **output** signal y.

$$\text{For all } (t, n) \in T, \qquad y(t, n) = \int_0^t x(\alpha, 0) d\alpha. \tag{6}$$

The output is the integral of the input. We propose three variants, which are meant to capture key properties of the most commonly used numerical integration algorithms.

All three variants require a communication point (t, m_t) at every t where x is discontinuous, where m_t is the final microstep at t (note that this requires that the input not have a chattering Zeno condition for time to advance). The variants are:

1. Variant 1 imposes no additional constraints on the communication points.
2. Variant 2 requires communication points at (t, m_t) for $t \in D$, where $D \subset \mathbb{R}_+$ is an arbitrary discrete set, chosen by the component.
3. Variant 3 requires a communication point at $(t, 0)$ for any t where there is a communication point.

In all cases, any additional communication points are optional, up the host simulator.

The first two variants do not require that the input $x(t, n)$ be set at a communication point (t, n) before the output $y(t, n)$ is retrieved (it must be eventually provided, but it can be provided after the output is retrieved). By definition (6), the output $y(t, n)$ does not depend on the input $x(t, k)$ for any $k \in \mathbb{N}$ at any time t. Hence, there is no **direct dependence** between the input x and the output y (i.e., no instantaneous dependence, unlike most of the components above). A component without such a direct dependence is called a **non-strict** component, meaning that the input need not be known at a particular time for the output at that time to be produced. The third variant is **strict**, reflecting that some integration algorithms *are* strict, despite the definition (6) (namely, those using implicit integration methods).

Discussion. Note that by definition (6), the output of this component does not depend on input values at non-zero microsteps, but a realization will need to see the final values at discontinuities of the input to maintain accuracy. For this reason, all variants require that inputs be provided at *final* microsteps. The final value of the input x at time t provides the boundary value for an initial value problem to be solved by the numerical integration algorithm. The algorithm needs this value, even if the mathematical ideal (6) does not. The only variant that requires the initial value $x(t, 0)$ to be also provided is variant 3, which reflects the requirements of an implicit integration algorithm. Such an algorithm requires input values at the end of an integration interval. At the start of the interval, the final value (confusingly) provides the initial value for an initial-value problem.

At the end of the interval, the initial value (confusingly) provides the final value of the signal in that interval.

Suppose that the input is discontinuous at times t_1 and t_2, where $t_1 < t_2$, and is continuous in between. In each such an interval, an FMU realizing any variant of this component needs to solve an initial-value problem, where the initial value of the integrator state y is equal to the final value of the signal y at time t_1, and the initial value of x (the derivative of y) is the final value of x at time t_1. The integration algorithm calculates the integral up to time t_2 using standard numerical integration techniques, and the value of y that it determines at t_2 will define the initial value of the signal y at time t_2.

During an interval (t_1, t_2) between discontinuities, the FMU may or may not be provided with additional input values x by the host simulator. I.e., there may or may not be communication points in between. In any case, it cannot be provided with *all* such input values, because in general there are an uncountably infinite number of them. Numerical analysts have developed various techniques for approximating such integrals given partial information about the input x. For example, some more sophisticated solvers assume that the input signal x is not only continuous, but also *smooth*, meaning that all its derivatives exist at all points in the interval. In this case, given only the final value of x at time t_1 and all its derivatives, then the value of x at any point in the interval is fully defined and can, in principle, be calculated. In other words, given the value of the input x at time t_1 and its derivatives, **extrapolation** to any point in the interval (t_1, t_2) is possible.

Even with the smooth assumption, providing *all* the derivatives at time t_1 is not practical. Only a finite number will be provided by a master algorithm (providing none or one derivative are the most common). To be interoperable with a wide range of simulation hosts, the FMU will have to adapt its integration algorithm to the information provided. For example, if no derivatives are provided, the FMU might extrapolate using **zero-order hold**, which assumes that the input is constant over the interval with the value given by the final value of x at time t_1. Or it might use the recent history of the input to approximate the derivatives.

No matter what integration algorithm is chosen, and what information is provided by the host simulator, at least for some inputs, there will be errors compared to the ideal value given by (6). The three variants above are intended to reflect key properties of the most commonly used algorithms.

3.8 Integrator with Reset

CT **input** signal x. DE **input** signal r. CT **output** signal y

The output is the integral of the input, but an additional discrete input can reset the state of the integrator.

Let $D \subset T$ denote the set of time stamps $(t, n) \in D$ where $r(t, n) \neq \varepsilon$. I.e., for all $(t, n) \in T$ where $(t, n) \notin D$, $r(t, n) = \varepsilon$. Since this set is order isomorphic with a subset of the natural numbers, we can list its elements in order, d_0, d_1, d_2, $\cdots$, where $d_n < d_{n+1}$. For all $(t, n) \in T$, the output is defined to be

$$y(t, n) = \begin{cases} \int_0^t x(\alpha, 0) d\alpha & \text{if } (t, n) < d_0 \text{ or } D = \emptyset \\ r(d) + \int_u^t x(\alpha, 0) d\alpha & \text{otherwise} \end{cases}$$

where $d = (u, m)$ is the largest element in D s.t. $(u, m) \leq (t, n)$. We assume the same three variants as the Integrator component, with the additional constraint that for all variants, there must be a communication point at every $(t, n) \in D$. In addition, to ensure a piecewise-continuous output, for every $(t, n) \in D$, we require a communication point at $(t, 0)$.

Discussion. Absent any reset events r, this is identical to the Integrator above. At the time of reset events, the state of the integrator (its output y) gets reset to the value of the reset event. Like the Integrator, the output y has no immediate dependence on the input x, and it does not depend on the value of the input at non-zero microsteps (in the ideal). But the output y does immediately depend on the input r.

Assuming r is piecewise continuous, it is absent at all $(t, 0)$. Reset events can only occur at microsteps greater than zero. This ensures that the output y is piecewise continuous. Suppose a reset event occurs at superdense time $(t_1, 1)$. The output $y(t_1, 0)$ is unaffected by it, and hence is part of the preceding continuous segment. At the next microstep, $y(t_1, 1)$ takes on the value given by $r(t_1, 1)$. If there are no further reset events at time t_1, then this will be the final value of the output, which provides the initial integrator state for the next integration interval.

3.9 Zero-Crossing Detector

The output is a discrete event when the input hits or crosses zero. CT **input** signal x. DE **output** signal y.

For all $(t, n) \in T$, the output is

$$y(t, n) = \begin{cases} 0 & \text{if } x(t, 0) = 0, n = 1, \text{ and there exists a} \\ & \delta > 0 \text{ s.t. for all } 0 < \epsilon < \delta, x(t - \epsilon, 0) \neq 0 \\ 0 & \text{if } n > 0 \text{ and } x(t, n-1) \text{ and } x(t, n) \\ & \text{have opposite signs,} \\ 0 & \text{if } n > 0 \text{ and } x(t, n-1) \neq 0 \text{ and } x(t, n) = 0, \\ \varepsilon & \text{otherwise} \end{cases} \tag{7}$$

To remove any ambiguity, "opposite signs" means that one value is negative, and the other value is positive.

Discussion. This is the most subtle of the components considered in this paper, but it represents widely used functionality in numerical simulation. Every widely used simulator provides mechanisms for monitoring signals for zero crossings, which may represent, for example, collisions of physical objects or other discrete physical phenomena (see [11] for a few examples). Note that "crossings" is a bit of a misnomer here, because our zero-crossing detector will output an event if the input merely "hits" but does not "cross" zero. This simplifies the test case considerably, because to detect a legitimate "crossing" would require non-causal behavior. The component would need to see the future of the input in order to output an event. Non-causal I/O relationships are problematic. Semantics of discrete-event, continuous-time, and hybrid systems models all rely on causal I/O relationships [24, 10, 13, 14, 16].

Existing simulators implement variations of this functionality. This particular definition is carefully constructed to ensure that if the input is piecewise continuous, then the output is a piecewise-continuous DE signal. The first case above produces an output at $n = 1$ to ensure that the output at $n = 0$ is absent, which ensures piecewise continuity since the output has been absent prior to this time. In addition, if an input hits zero and then stays there, this component produces an output event only when it first hits zero. Simulink's "Hit Crossing" block, in contrast, would continue to produce an event indicator as long as the input signal re-

mains at zero. Our test case is simpler because the Simulink behavior would yield an output signal that is neither a CT signal nor a DE signal. While such signals are useful and legitimate, the additional complexity of handling them does not add anything fundamental to our test suite.

Here, we assume that the input x is piecewise continuous. Otherwise, it is not clear that there is a reasonable definition of a zero crossing detector. Piecewise continuity ensures that if there is a zero crossing within a continuous interval, then the first condition in (7) will be satisfied within this interval at the point of the zero crossing. Furthermore, the second and third conditions in (7) ensure that if a discontinuous signal crosses or hits zero during a discontinuity, the event is detected.

If the input is not piecewise continuous, then this definition of a zero-crossing detector is not assured of detecting zero crossings. Consider a non-piecewise-continuous input given by

$$x(t,n) = \begin{cases} -1 & \text{if } t \leq 1 \\ 1 & \text{otherwise} \end{cases}$$

Our zero-crossing detector produces a constant absent output, failing to detect this zero crossing. But when does this zero crossing occur? There is no earliest time t at which $x(t,n) = 1$, so there is no clean definition of time of the zero crossing. We could, of course, invoke contortions using limits from the right, but the definition will be non-causal. An event produced at time $t = 1$ would depend on *future* inputs. No such difficulties arise if the input is piecewise continuous, so we simply define a test case that yields expected behavior when the input is piecewise continuous. We do not care what the behavior is when the input is not piecewise continuous, because the behavior yielded by our component is as defensible as any other.

The output of this component is a piecewise-continuous DE signal. Note that it is always absent at microstep zero, and that as long as the input has no chattering Zeno condition, the final value of the output is always absent.

The I/O dependencies of this component are interesting. When the input is continuous, there is arguably a microstep delay, because the input is zero at microstep zero, but the output event is not produced until microstep one. However, at discontinuities, the second and third conditions of equation (7) have no such microstep delay. So in general there is no microstep delay from input to output.

The first condition in (7) compares a real-valued input x to a fixed constant zero. Such a comparison is not exactly realizable in a computer when the input is a computed value of a continuous signal defined on a time continuum; hence an approximation is needed. A typical approximation allows $x(t,0)$ to be non-zero as long as it is small and has a sign that is opposite to that of $x(t-\epsilon,0)$ over an interval $\epsilon \in (0,\delta)$. That is, a zero-crossing is allowed to overshoot by a small amount. Undershoot is typically not allowed, because then there is no assurance that either a crossing or a "hit" has occurred.

With such an approximation, the time at which the crossing is detected will be slightly late. But any digital representation of time must be quantized in any case (see Section 2.4 below), so some approximation is always necessary. A useful regression test will specify a precision requirement, so a useful implementation of this ideal component should provide a mechanism for controlling the precision (a parameter, for example).

This component requires a communication point at or near the point of a zero crossing. This introduces an **unpredictable breakpoint**, because the time of this zero crossing cannot be known in advance, in general. A typical realization of such a Zero-Crossing Detector cooperates with the master algorithm to regulate the step size taken by a simulator so that the detection delay is less than some specified time resolution. As a consequence, a hybrid cosimulation FMI standard should include a mechanism for an FMU to cooperate with the host simulator, regulating time steps taken by the simulator. One mechanism would be for the FMU to be able to reject step sizes proposed by the host simulator until the step size is small enough to detect the zero crossing within the specified resolution. Again, see [6] for a discussion of such mechanisms.

3.10 Zero-Order Hold

DE **input** signal x. CT **output** signal y.

The output is defined to be

$$y(\tau) = \begin{cases} x(d) & \text{if } d \text{ exists,} \\ \varepsilon & \text{otherwise} \end{cases}$$

where d is the largest element in T s.t. $d \leq \tau$ and $x(d) \neq \varepsilon$.

Discussion. Recall that a DE signal is non-absent at a discrete subset of times. This property ensures that if there is any input event at all, then all outputs at the time of that event and beyond are not absent. Specifically, if there is any time $\tau_0 \in T$ where $x(\tau_0) \neq \varepsilon$, then for all $\tau \in T$ where $\tau \geq \tau_0$, there exists a largest $d \leq \tau$ such that $x(d) \neq \varepsilon$.

If the input is piecewise continuous, then the output will also be piecewise continuous. This fact is not entirely obvious from the definition. Note that for x to be a piecewise-continuous DE signal, it must be true that both its initial and final value functions are absent. Hence, if an event arrives on x at time $t_0 \in \mathbb{R}_+$, then the event must occur at a microstep strictly greater than 0. That is, $x(t_0,0) = \varepsilon$. Hence, $y(t_0,0)$ will have the value specified by *previous* event (or be absent if there is no previous event). Suppose that $x(t_0,1) \neq \varepsilon$. Then $y(t_0,1) = x(t_0,1)$. If final value of x at t_0 is $x(t_0,2) = \varepsilon$, then the final value of y at t_0 will be $y(t_0,1) = x(t_0,1)$, and this will be the value of y until (and including) $(t_1,0)$, where t_1 is the time of the arrival of the next event.

The output of this component has a discontinuity at each time t where $x(t,n) \neq \varepsilon$. In order for this discontinuity to manifest correctly as a sequence of two distinct values at the same time t with distinct microsteps, there will need to be two distinct communication points at the time t of the input event. Suppose that an input event occurs at (t,n) for some $n > 0$. That is, $x(t,n) \neq \varepsilon$. It is sufficient for the host simulator to provide a communication point at both $(t,0)$ and (t,n). But to provide a communication point at $(t,0)$, the host simulator needs to "anticipate" the event at (t,n). It need not provide a communication point at $(t,0)$ unless there will be an event at (t,n), but (t,n) is in the future. This seems to imply non-causal behavior.

Fortunately, we can meet this requirement without any non-causal mechanisms. The simplest mechanism is already realized in FMI 2.0, where whenever a non-zero step size is taken, the next communication point automatically occurs at microstep zero. There is no mechanism in FMI 2.0 to skip microstep zero. The Ptolemy II Continuous direc-

tor [21], which implements hybrid system simulation, also always provides a communication point $(t, 0)$ for every communication point (t, n). So a Zero-Order Hold FMU will never be forced to reject a time step if Ptolemy II is the host simulator.

But even if the future hybrid cosimulation standard does provide a mechanism to advance from communication point (t_0, n_0) to (t_1, n_1), where $t_1 > t_0$ and $n_1 > 0$, then an FMU implementing this Zero-Order Hold component can reject the proposed step. It thereby ensures that for every communication point (t, n), there will also be a communication point $(t, 0)$. This mechanism does not require anticipating future events.

3.11 Sampler

Input signal x. DE input signal s. DE **output** signal y.

For all $\tau \in T$, the output is

$$y(\tau) = \begin{cases} x(\tau) & \text{if } s(\tau) \neq \varepsilon \\ \varepsilon & \text{otherwise} \end{cases}$$

Discussion. The output is a discrete sampling of a continuous input. As long as the input s is a piecewise continuous DE signal, the output will be a piecewise continuous DE signal. Since the output is a DE signal, it will be absent at all $(t, 0) \in T$. If the input s is free of chattering Zeno conditions, then the output will also be free of chattering Zeno conditions. The communication points should include at least every $\tau \in T$ where $s(\tau) \neq \varepsilon$.

3.12 Discrete Time Delay

DE **input** signal x. DE **output** signal y. Real **parameter** d, where $d > 0$.

For all $(t, n) \in T$, the output is

$$y(t, n) = \begin{cases} x(t - d, n) & \text{if } t \geq d \\ \varepsilon & \text{otherwise} \end{cases}$$

The output is a time-delayed input.

Discussion. The communication points should include every (t, n) where $x(t, n) \neq \varepsilon$, and also every $(t + d, n)$. Notice that we do not attempt to define this for CT inputs because such time delays are extremely difficult to realize in simulation. In theory, they have an uncountably infinite state space. And they wreak havoc with step-size control mechanisms in variable-step solvers.

3.13 Discrete Microstep Delay

DE **input** signal x. DE **output** signal y.

For all $(t, n) \in T$, the output is

$$y(t, n) = \begin{cases} x(t, n - 1) & \text{if } n \geq 1 \\ \varepsilon & \text{if } n = 0 \end{cases}$$

The output is a microstep-delayed input.

Discussion. The communication points should include at least every (t, n) where $x(t, n) \neq \varepsilon$ and also $(t, n + 1)$. This component, therefore, requires a mechanism for ensuring zero step advancement of superdense time (which the microstep only advances).

Notice that we do not attempt to define this for CT inputs. Indeed, if presented with a CT input, this component will produce a rather odd output signal, one whose initial value is always absent, and subsequent values are present. If the input is a piecewise continuous DE signal, then the input is always absent at microstep zero, and the output will also be piecewise continuous. If the input is free of chattering Zeno conditions, then the output will be free of chattering Zeno conditions. In this case, the final value of the input is also ε, so the final value of the output will be ε.

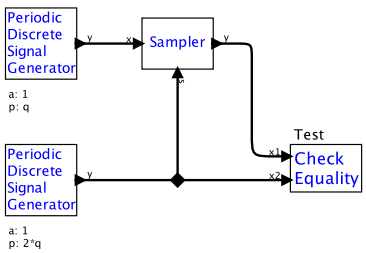
4. COMPOSITION TEST CASES

A hybrid cosimulation FMI standard that enables definition of the above components provides a rich framework for composition of discrete and continuous simulation tools. Any such standard should be able to unambiguously define FMUs that realize such components and should ensure that host simulators are capable of executing these FMUs. Such capabilities can be verified using unit tests that check each of the above components individually by providing a range of inputs and verifying that the outputs match the ideal (up to some precision, where appropriate). But such unit tests are not quite sufficient. We also need to ensure that interactions between multiple components behave correctly.

In this section, we discuss some test cases that combine a few of the above components, and give acceptance criteria that define correct behavior. These test cases are, in effect, constraints on master algorithms. Host simulators that conform with the standard must implement master algorithms that satisfy these acceptance criteria.

4.1 Simultaneous Events

This test case checks that multiple components with discrete timed behavior coordinate their representations of time. Consider the following composition:



This has three components:

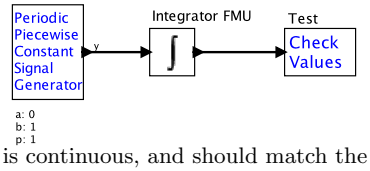
1. Periodic Discrete Signal Generator. Period p=q, a=1.
2. Periodic Discrete Signal Generator. Period p=2q, a=1.
3. Sampler with DE input x

The criterion is that the output of the Sampler should equal the output of second Periodic Discrete Signal Generator at all superdense times. Here, q is any representable time such that $2q$ is also a representable time.

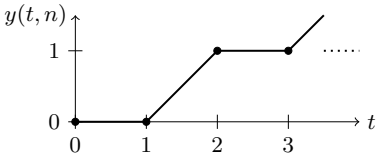
Discussion. FMUs may internally use representations of time that are different from that of the host simulator. This test criterion is intended to ensure that no matter how the FMU and host simulator internally represent time, the Sampler and Periodic Discrete Signal Generator semantics are respected. This test case also checks for a well-defined notion of simultaneity. In particular, the periods chosen are not exactly representable with double precision floating point numbers, and the test is intended to ensure that despite any roundoff errors, every second output of the first Periodic Discrete Signal Generator is exactly synchronous with every output of the second one. The Test component must see the events at the same communication point in superdense time.

4.2 Integrating Discontinuous Signals

The following figure shows a test case that integrates a discontinuous input.



The output is continuous, and should match the following:

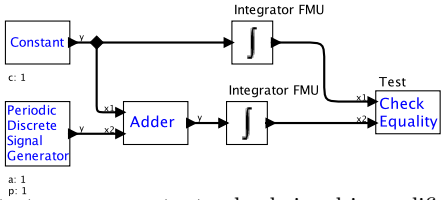


Every acceptable test result will produce at least one output sample at the times of the discontinuities of the output of the Periodic Piecewise Constant Signal Generator. Samples in between these points in time are optional, and may depend on the step-size control algorithm used by the host simulator.

Discussion. The key feature being tested here is that a host simulator does not get confused by a signal that has two distinct values at the same (real) time, at distinct microsteps, and that a sequence of values at a (real) time does not affect the output of an Integrator, except that the final value at a discontinuity becomes the initial value for the next integration interval.

4.3 Integrating Glitches

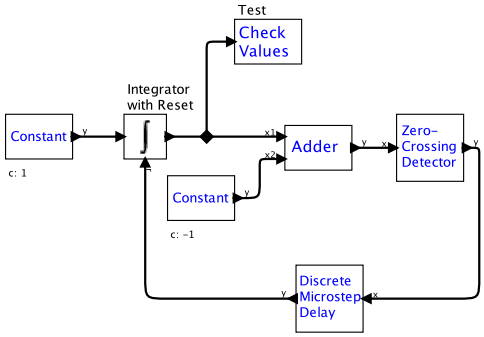
The following Figure shows a test case that verifies that the Integrator output is unaffected by input values whose duration is zero.



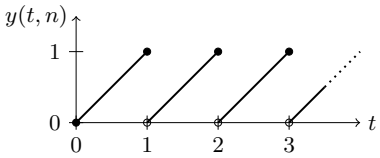
In this test case, a constant-valued signal is modified using an Adder so that its value at integer-valued times sequences from 1 to 2 and back to 1, without time elapsing. These **glitches** have zero width, and hence should not affect the output of the Integrator.

4.4 Zero-Delay Feedback

The following Figure shows a test model using a Zero-Crossing Detector in a feedback loop.



The Integrator with Reset is integrating a constant 1, and hence will produce a line with unit slope. When that line crosses 1, the Zero-Crossing Detector is triggered. The event it produces, which has value 0, is fed back through a Discrete Microstep Delay to the reset input of the Integrator with Reset. The expected output is as follows:



Because of the approximate nature of the Zero-Crossing Detector (see Section 3.9), the times at which the reset occurs and the value at which it is triggered are approximate, so a regression test needs to specify a tolerance.

In the plot above, the filled and unfilled dots are required samples, occurring at microsteps 0 and 2 respectively. Samples in between are optional and may depend on the step-size control algorithm of the host simulator. Specifically, at time $t = 1$, the output of the Integrator With Reset should be

$$y(1, 0) = a \qquad y(1, 1) = a \qquad y(1, 2) = 0$$

where $a \approx 1$. Notice that the reset actually occurs in microstep 2, because the event at the output of the Zero-Crossing Detector occurs at microstep 1, and it is then delayed by one additional microstep. In this particular instance, the Discrete Microstep Delay in the feedback loop might not seem to be required because the input to the Zero-Crossing Detector is continuous, and by the definition of the Zero-Crossing Detector, it introduces a microstep delay when the zero crossing occurs in a continuous region of the input. Nevertheless, our test case includes a Discrete Microstep Delay for two reasons. First, it provides a test where microsteps explicitly go beyond 1. Second, the Zero-Crossing Detector introduces a microstep delay only for some inputs. So the presence of a microstep delay in the loop is not a static property, which complicates scheduling of the components. Specifically, the Discrete Microstep Delay is non-strict, meaning that its input at superdense time τ need not be known to retrieve its output at τ. A scheduler can take this into account to break the apparent dependency loop created by the feedback. The Zero-Crossing Detector, however, is only non-strict at microstep zero (because its output is always absent at microstep zero). Hence, without the Discrete Microstep Delay, we would have a causality loop at all microsteps but zero. A master algorithm would have to ensure that at the input to the Zero-Crossing Detector, $m_t = 0$ for all t. In general, this is difficult to ensure.

Discussion. A subtle point raised by this composition is that the master algorithm needs to "know" at any time t when all signals have reached their final microstep. Specifically, it is not sufficient to stop incrementing microsteps at time t when all signals become absent at time t. First, CT signals never become absent at time t, so the mere presence of a CT signal will foil this strategy. Second, the Discrete Microstep Delay may have an absent input, and yet, in the next microstep, produce a non-absent output.

Our assumption is that each FMU constrains step sizes so that prior to reaching the final microstep of its outputs, it prevents the master algorithm from advancing time. It only permits advances in microstep. When no component does this, the master algorithm can assume that all signals have

reached their final microstep.

For simplicity, we have constrained Discrete Microstep Delay so that it can only produce a non-absent output at the very next microstep. I.e., it implements a unit microstep delay, not an m-step microstep delay. It is certainly possible to generalize this mechanism to allow an m-step microstep delay, but we have not encountered any need for such generality, so we have not included this in the test cases for a hybrid cosimulation standard.

5. RELATED WORK

Together with the FMI 2.0 standard, the FMI Development Group has developed an FMI Test Package consisting of Modelica models [18]. In this paper, we take a different approach: instead of providing a Modelica model as a reference, we give a mathematical ideal. The ideal correct behavior of a model is unambiguous and language and tool independent. These tests reflect discrete-event behaviors that are found in other types of simulators, such as network simulators, hardware description languages, DEVS-based system simulators [24], and synchronous-reactive languages [4].

A general discussion about the FMI standard's limitations on cosimulation can be found in [6]. Several authors discuss ways of implementing master algorithms [3, 6, 22] and how to generate and implement FMUs for the current FMI cosimulation standard [8, 20]. However, in this paper we formalize requirements and test cases that are needed for the *next standard* that should include a sound interface for hybrid co-simulation.

6. CONCLUSION

Hybrid cosimulation is a far from trivial goal. A standard that enables composition of simulation tools has two conflicting objectives. It needs to be sufficiently rigorous to define the meaning of a composition of components. And it needs to be flexible enough to embrace industry-standard and established simulators. The former demands a rigorous semantics, but the later creates pressure for less well-defined semantics.

This paper does not define such a standard, but instead enumerates a set of capabilities that such a standard must support to be useful. In particular, the paper enumerates a suite of components and test cases that cover a wide range of hybrid system behaviors.

7. REFERENCES

[1] R. Alur, C. Courcoubetis, N. Halbwachs, T. Henzinger, P. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138:3–34, 1995.

[2] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

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

[4] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, 1991.

[5] T. Blochwitz, M. Otter, M. Arnold, C. Bausch, C. Clauß, H. Elmqvist, A. Junghanns, J. Mauss, M. Monteiro, T. Neidhold, D. Neumerkel, H. Olsson, J.-V. Peetz, and S. Wolf. The functional mockup interface for tool independent exchange of simulation models. In *Proc. of the 8-th International Modelica Conference*, Dresden, Germany, Mar. 2011. Modelica Association.

[6] D. Broman, C. Brooks, L. Greenberg, E. A. Lee, M. Masin, S. Tripakis, and M. Wetter. Determinate Composition of FMUs for Co-Simulation. In *Proceedings of the International Conference on Embedded Software (EMSOFT 2013)*. IEEE, 2013.

[7] L. P. Carloni, R. Passerone, A. Pinto, and A. Sangiovanni-Vincentelli. Languages and tools for hybrid systems design. *Foundations and Trends in Electronic Design Automation*, 1(1/2), 2006.

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

[9] D. Harel and A. Pnueli. On the development of reactive systems. In K. R. Apt, editor, *Logic and Models for Verification and Specification of Concurrent Systems*, volume F13 of *NATO ASI Series*, pages 477–498. Springer-Verlag, 1985.

[10] E. A. Lee. Modeling concurrent real-time processes using discrete events. *Annals of Software Engineering*, 7:25–45, 1999.

[11] E. A. Lee. Constructive models of discrete and continuous physical phenomena. *IEEE Access*, 2(1):1–25, 2014.

[12] E. A. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Circuits and Systems*, 17(12):1217–1229, 1998.

[13] E. A. Lee and H. Zheng. Operational semantics of hybrid systems. In M. Morari and L. Thiele, editors, *Hybrid Systems: Computation and Control (HSCC)*, volume LNCS 3414, pages 25–53, Zurich, 2005. Springer-Verlag.

[14] X. Liu, E. Matsikoudis, and E. A. Lee. Modeling timed concurrent systems. In *CONCUR 2006 - Concurrency Theory*, volume LNCS 4137, pages 1–15. Springer, 2006.

[15] O. Maler, Z. Manna, and A. Pnueli. From timed to hybrid systems. In *Real-Time: Theory and Practice, REX Workshop*, pages 447–484. Springer-Verlag, 1992.

[16] E. Matsikoudis and E. A. Lee. On fixed points of strictly causal functions. In *International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS)*, volume LNCS 8053, pages 183–197. Springer-Verlag, 2013.

[17] Modelica Association. Functional mock-up interface for model exchange and co-simulation. Report 2.0, 2014.

[18] Modelica Association. *Modelica - A Unified Object-Oriented Language for Physical Systems Modeling - Language Specification Version 3.3 Revision 1*, 2014. Available from: `http://www.modelica.org`.

[19] M. Otter, M. Malmheden, H. Elmqvist, S. E. Mattsson, and C. Johnsson. A new formalism for modeling of reactive and hybrid systems. In *Modelica Conference*. The Modelica Association, 2009.

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

[21] C. Ptolemaeus, editor. *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, Berkeley, CA, 2014.

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

[23] P. Tabuada. *Verification and Control of Hybrid Systems: A Symbolic Approach*. Springer, 2009.

[24] B. P. Zeigler, H. Praehofer, and T. G. Kim. *Theory of Modeling and Simulation*. Academic Press, 2nd edition, 2000.