Hybrid Co-simulation: It's About Time

Fabio Cremona
Marten Lohstroh
David Broman
Stavros Tripakis
Edward A. Lee

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2017-6
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-6.html

April 6, 2017

Copyright © 2017, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Hybrid Co-simulation: It's About Time

Fabio Cremona · Marten Lohstroh ·
David Broman · Edward A. Lee ·
Michael Masin · Stavros Tripakis

December 21, 2016

Abstract Model-based design methodologies are commonly used in industry for the development of complex cyber-physical systems (CPS). There are many different languages, tools, and formalisms for model-based design, each with its strengths and weaknesses. Instead of accepting some weaknesses of a particular tool, an alternative is to embrace heterogeneity, and to develop tool integration platforms and protocols to leverage the strengths from different environments. A fairly recent attempt in this direction is the Functional Mock-up Interface

This work is partially based on previous work published by the authors [7, 8, 15], but the contributions presented in this article stand on their own. This work was supported in part by the TerraSwarm Research Center, one of six centers administered by the STARnet phase of the Focus Center Research Program (FCRP) a Semiconductor Research Corporation program sponsored by MARCO and DARPA, by the National Science Foundation (NSF) awards #1446619 (Mathematical Theory of CPS), #1329759 (COSMOI) and #1139138 (ExCAPE), and by iCyPhy (the Industrial Cyber-Physical Systems Research Center) and the following companies: Denso, National Instruments, and Toyota. This work was also supported by the Swedish Research Council #623-2013-8591 and by the Academy of Finland.

Fabio Cremona
University of California, Berkeley, USA
E-mail: f.cremona@eecs.berkeley.edu

Marten Lohstroh
University of California, Berkeley, USA
E-mail: marten@eecs.berkeley.edu

David Broman
KTH Royal Institute of Technology, Sweden
E-mail: dbro@kth.se

Edward A. Lee
University of California, Berkeley, USA
E-mail: eal@eecs.berkeley.edu

Michael Masin
IBM Research – Haifa, Israel
E-mail: michaelm@il.ibm.com

Stavros Tripakis
University of California, Berkeley, USA & Aalto University, Finland
E-mail: stavros@eecs.berkeley.edu

(FMI) standard that includes support for co-simulation. Although this standard has reached acceptance in industry, it provides only limited support for simulating systems that mix continuous and discrete behavior, which are typical of CPS. This paper identifies the representation of time as a key problem, because the FMI representation does not support well the discrete events that typically occur at the cyber-physical boundary. We analyze alternatives for representing time in hybrid co-simulation and conclude that a superdense model of time using integers only solves many of these problems. We show how an execution engine can pick an adequate time resolution, and how disparities between time representations internal to co-simulated components and the resulting effects of time quantization can be managed. We propose a concrete extension to the FMI standard for supporting hybrid co-simulation that includes integer time, automatic choice of time resolution, and the use of absent signals. We explain how these extensions can be implemented modularly within the frameworks of existing simulation environments.

Keywords Co-simulation · Functional Mock-up Interface · Time

1 Introduction

Model-based design of **cyber-physical systems** (CPS) requires modeling techniques that embrace both the cyber and the physical parts of a system [24]. There is a long history of modeling languages and tools that integrate techniques that were originally developed independently, and on different sides of the border that separates the cyber and the physical. Modelica [20, 39], for example, integrates object-oriented design (a cyber modeling technique) with differential-algebraic equations (DAEs, a physical modeling technique). Languages and tools for hybrid systems design [11] integrate finite state automata (cyber) with ordinary differential equations (ODEs, physical). Discrete-event (DE) modeling tools [12,18,29,49] integrate a model of a time continuum (physical) with discrete, instantaneous events (cyber). Such simulation tools are capable, in principle, of simulating both cyber components (software and networks) and physical components (mechanical, electrical, fluid flows, etc.).

In spite of the power and utility of existing tools, we should not be sanguine about CPS modeling. All of the above integrations have pitfalls, limitations, and corner cases where a model that can be easily handled in one tool cannot be easily handled in another. Modelica-based tools, for example, have difficulty with some discrete phenomena, even purely physical ones, forcing model builders to sometimes model discrete behaviors as rapid continuous dynamics [41]. Conversely, tools that handle discrete events well, such as DE tools, may have difficulty with continuous dynamics [36], forcing model builders into brute-force methods such as sampled-data models with high sampling frequencies.

One possible solution is to embrace the heterogeneity of tools and to provide tool integration platforms and protocols that enable co-simulation using a multiplicity of tools [23]. There is a long history of tool integration platforms (sometimes called "simulation backplanes") for DE modeling and a well-established standard called the High-Level Architecture (HLA) for tool interoperability [27]. A more recent development is the **Functional Mock-up Interface** (FMI), a standard initiated by Daimler AG within the ITEA2 MODELISAR project [5, 40], now maintained by the Modelica Association. It has been designed to enable the exchange or

co-simulation of model components, Functional Mock-up Units (FMUs), designed with different modeling tools. The standard consists of a C application program interface (API) for simulation components and an XML schema for describing components. Largely unspecified is the algorithm that coordinates the execution of a collection of FMUs, the **master algorithm** (MA). The idea is that the standard should be flexible enough to accommodate the inevitable differences between execution engines in different tools. FMI provides two distinct mechanisms for interaction between an FMU and a host simulator: i) **model exchange** (FMI-ME), where the host simulator is responsible for all numerical integration methods, and ii) **co-simulation** (FMI-CS), where the FMU implements its own mechanisms for advancing the values of state variables. FMI for co-simulation is more focused on tool interoperability; 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.

The current standard for co-simulation (version 2.0 [38]), however, is unable to correctly simulate many mixed discrete and continuous behaviors, limiting its utility in current form for model-based design for CPS [8]. As a consequence, the community-driven standardization process is considering another mechanism called **hybrid co-simulation** that strives for the loose coupling of co-simulation, but with support for discrete and discontinuous signals and instantaneous events. The intent of this mechanism is to support hybrid systems [1, 11, 34, 42, 46], where continuous dynamics are combined with discrete mode changes and discrete events. Hybrid co-simulation promises better interoperability between models of the cyber and the physical sides of the CPS problem.

In this article, we focus on a particular issue with hybrid co-simulation that has proved central to the problem, namely the *modeling of time*. Time is a central concept in reasoning about the physical world, but is largely abstracted away when reasoning about the cyber world. As a result, the engineering methods that CPS builds on have misaligned abstractions between the physics domain, the mathematical domain used to model physics, the computational domain used to implement these mathematical abstractions for simulation, and the computational domain used on the cyber side of CPS. The most egregious misaligned abstractions concern time, where all four domains routinely use mutually incompatible models of time.

The most common resolution for this conundrum is to adopt the naive Newtonian ideal model of time, where time is a real number known everywhere and advancing uniformly, and the real number is approximated in software as a floating-point number. In this paper, we show that floating-point numbers are inadequate for hybrid co-simulation. We describe instead a representation of time that eliminates the problems associated with floating-point representations, allows for multiplicity of time resolutions, and allows for cyber abstractions where events can occur discretely in time and in sequences without time advancing. For the latter property, we adopt a form of **superdense time** [33, 34] (see Section 2.2). Our solution satisfies all of the requirements for hybrid co-simulation stated in [8].

Although the approach presented in this paper is general and could potentially be applicable to many different hybrid co-simulation environments, we have chosen to illustrate the concept concretely, by applying it to FMI and showing that only modest extensions to the FMI standard are needed to follow our recommendations. Within this framework, we show how to perform hybrid co-simulation with

heterogeneous time models and accommodate mixtures of components that may internally represent time differently. Specifically, our solution supports hybrid co-simulation of FMUs that use floating-point time together with integer-time FMUs, even if those integer-time FMUs internally use a different resolutions.
In summary, we make the following contributions:

- We analyze and compare alternatives for representing time, including floating-point numbers, rational numbers, and integers. We discuss superdense time and the concepts of time resolution. We also propose a model of time that supports a multiplicity of time resolutions, differing even within the same simulation, supports discrete events with an exact notion of simultaneity, invulnerable to quantization errors, and is efficiently converted to and from legacy floating-point representations of time to accommodate legacy simulators within a co-simulation environment. It also supports abstractions of time such as sequences of events where time does not elapse, enabling better integration of cyber models with physical ones (Section 2).
- We present a concrete proposal for a new FMI standard for hybrid co-simulation, FMI-HC. The three main parts of the proposal are: i) the use of integer time, ii) the capability of FMUs to negotiate the resolution of time, iii) the use of absent signals for handling discrete events (Section 3).
- We describe how a master algorithm can use the FMI-HC extensions and supports co-simulation of components that operate at different time resolutions. The algorithm finds a suitable global time resolution for the simulation based on the FMUs' preferences and is able to handle disparities between the time resolutions of co-simulated FMUs. Our modular implementation using wrappers demonstrates that it is easy to add support for hybrid co-simulation to existing master algorithms (Section 3).
- We give a detailed analysis and a solution to the time conversion and quantization problem, an unavoidable consequence when different components operate at different time resolutions (Section 4).

And finally, our implementation of the proposed FMU wrappers, a simple method to achieve compatibility with FMI-HC, is explained in detail in Appendix A.

1.1 A Motivating Example

Broman et al. in [8] give the model shown in Figure 1 as a "simplest possible" nontrivial illustration of the challenges of hybrid co-simulation. This model includes a simplest-possible physical side, an Integrator integrating a constant and the result being offset by a constant by an Adder. This model detects a zero crossing of the offset output of an integrator, providing a simplest-possible nontrivial interface from the physical side to the cyber side. This detector produces a discrete event that is the processed by the simplest-possible cyber component, labeled "Microstep Delay." The output of this component then resets the Integrator, creating a simplest-possible closed-loop cyber-physical control system. In this model, the cyber component is abstracted as instantaneous, and the ensuing potential causality loop is averted by using superdense time, introducing an infinitesimal delay that makes the model constructive (for subtleties surrounding such models, see [30]). In this model, the resetting of the continuous output of the integrator

is required to occur as a discrete, zero-duration discontinuity, but to ensure consistent semantics for the physical models, all continuous-time signals are required to be piecewise continuous, and continuous from both the left and the right at all points of discontinuity. These requirements imply that some form of superdense time is required.

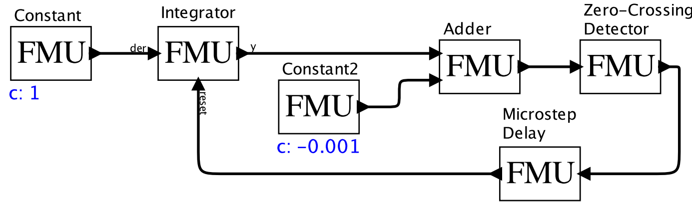

Fig. 1 A zero-delay feedback model.

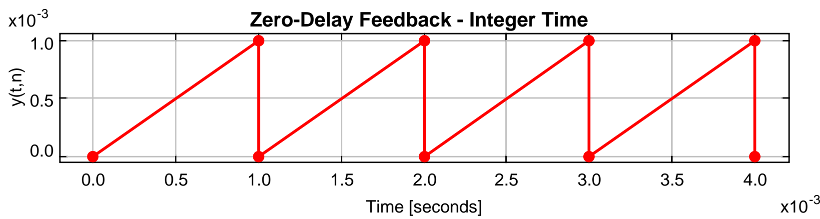

Fig. 2 Output of `Integrator`.

Figure 2 shows the output of the `Integrator` assuming the constant values 1 and -0.001 shown in Figure 1. An essential feature of this output is that discontinuities occur at precise times, that they take zero time to transition, and that between the discontinuities, signals are continuous.

Our goal in this paper is to provide a model of time for the interactions between these components that is semantically well-defined and exact. If the interactions between the components are well defined, then the components themselves can be made much more complex, with predictable results. The Integrator FMU could be replaced with a sophisticated ordinary differential equation (ODE) solver with a much more complicated internal model, the Adder could be replaced with some continuous-time simulation engine, the Zero-Crossing Detector could be replaced with more elaborate discrete-event processing, and the Microstep Delay could be replaced with some software engineering model of the control strategies.

1.2 Related Work

This paper follows the line of research on FMI initiated in [7], where the FMI standard was formalized, and two co-simulation algorithms were proposed and proven to be determinate. In that same paper, small extensions to the standard

were proposed with the goal of enhancing the standard's ability to handle mixed discrete and continuous behaviors. Some of these extensions are reminiscent of functions included in the actor interface in the modular formal semantics of the Ptolemy tool [48]. For instance, the "getMaxStepSize" function of [7] is similar to the D ("deadline") function of [48].

Followup work includes [8], which proposes a collection of test cases together with acceptance criteria that can be used to determine whether a hybrid co-simulation technique is acceptable. Tripakis in [47] investigates techniques to bridge the semantic gap between various formalisms (state machines, discrete-event models, dataflow, etc.) and FMI. Cremona et al. in [14] propose a new master algorithm that uses step size refinement to enable state event detection with FMI. An implementation of the FMI extensions on top of Ptolemy II is described in [15]. This implementation has been used in [6] to connect via co-simulation the model-checkers Uppaal [28] and SpaceEx [19]. Several authors have also described approaches to implement FMI master algorithms [2, 45], and ways to implement FMUs in the currently available FMI standard [17, 43], without considering the time aspects for hybrid co-simulation.

The topics addressed in this paper are relevant to modeling of hybrid and cyber-physical systems at large, but we choose to present our ideas on the basis of a concrete framework, namely the FMI standard. They could equally well be applied to other frameworks, such as HLA. Several papers exist in the literature that address the problem of formal modeling of cyber-physical systems, in particular, using hybrid automata with an emphasis on verification [1, 22, 26, 46]. The focus of this paper, however, is not formal verification, but rather (co-)simulation, with an emphasis on the practicalities, and in particular the representation of time.

The list of modeling languages and tools for CPS and hybrid systems design is long, and beyond the scope of this paper to cover exhaustively. A survey dating from 2006 can be found in [11], and a description of the mapping between formalisms, languages, and tools can be found in [9]. There have been numerous developments in the field, including many others, e.g., [3, 4, 10, 50, 51].

A side benefit of our proposal in this paper is that it potentially enables co-simulation between classical ODE simulators and a relatively newer way of modeling continuous dynamics called "quantized-state systems" (QSS) [25]. QSS simulators model continuous dynamics using discrete events, and sometimes the resulting simulations are more accurate (because of the use of symbolic computation) and more efficient than classical ODE solvers [31]. Since our proposed technique facilitates interoperability between continuous-time solvers and discrete-event systems, and QSS is based on discrete-event systems, it potentially enables interesting hybrid simulation techniques, where QSS can be used where it is most beneficial.

2 Representing Time

A major challenge in the design of cyber-physical systems is that time is almost completely absent from models used on the cyber side, while time is central on the physical side. In order for hybrid simulators to interact in predictable and controllable ways, we will need a semantic notion of time that can be used to model both continuous physical dynamics and discrete events. It is naive to assume that we can just use the Newtonian ideal, where time is absolute, a real number t,

visible everywhere, and advancing uniformly. We begin in this section by reviewing a set of requirements that any useful model of time must satisfy. We then elaborate with an analysis of practical, realizable alternatives that meet these requirements, at least partially.

2.1 Models of Time

Like the Newtonian ideal, any useful semantic notion of time has to provide a clear ordering of events. Specifically, each component in a system must be able to distinguish past, present, and future. The *state* of a component at a "present" is a summary of the past, and it contains everything the component needs to react to further stimulus in the future. A component changes state as time advances, and every observer of this component should see state changes in the same order.

We also require a semantic notion of time to respect an intuitive notion of causality. If one event A causes another B, then every observer should see A ordered before B.

In order to cleanly support discrete events, we also require a semantic notion of simultaneity. Under such a notion, two events are simultaneous if all observers see them occurring at the same time. We need to avoid models where one observer deems two events to be simultaneous and another does not.

We could easily now digress into philosophy or modern physics. For example, how could a notion of simultaneity be justifiable, given relativity and the uncertainty principles of quantum mechanics? We resist the temptation to digress, and appeal instead to practicality. We need models that are *useful* for co-simulation. The goal is be able to design and build better simulators, not to unlock the secrets of the universe. Even after the development of relativity and quantum mechanics, Newtonian ideal time is a practical choice for studying many macroscopic systems.

But ironically, Newtonian time proves not so practical for hybrid co-simulation. The most obvious reason is that digital computers do not work with real numbers. Computer programs typically approximate real numbers using floating-point numbers, which can create problems. While real numbers have infinite precision, their floating-point representation does not. This discrepancy leads to quantization errors. Quantization errors may accumulate. Although real numbers can be compared for equality (e.g. to define "simultaneity"), it rarely makes sense to do so for floating-point numbers. In fact, some software bug finders, such as Coverity, report equality tests of floating-point numbers as potential bugs.

Consider a model where two components produce periodic events with the same period starting at the same time. The modeling paradigm should assure that those events will appear simultaneously to any other component that observes them. Without such a notion of simultaneity, the order of these events will be arbitrary, and changing the order of discrete events can have a much bigger effect than perturbing their timing, and a much bigger effect than perturbing samples of a continuous signals. Periods that are simple multiples of one another should also yield simultaneous events. Quantization errors should not be permitted to weaken this property.

Broman et al. [8] list three requirements for a model of time:

1. The precision with which time is represented is 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 precision differs between observers, then they 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,

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

In contrast to the above quote from Broman et al. [8], and to avoid confusion with the term precision in measurement theory, henceforth we will in this article use the term *resolution* instead of precision to denote the grain at which we can tell apart two distinct time stamps.

Definition 1 (Time resolution) Time resolution is the smallest representable time difference between two time stamps.

For instance, if we state that "a model has a time resolution of one millisecond," or for short, "the time resolution is milliseconds," it means that the time points 0.001s, 0.002s, 0.003s, ... are representable, but the time points in between are not. No values can be defined at unrepresentable time points.

Note that properties 2 and 3 are not satisfied by floating-point numbers due to rounding errors [21]. For instance, consider the following C code that adds double precision floating-point numbers.

```
double r = 0.8;
double k = 0.7;
k = k + 0.1;
printf("%f,%f,%d\n",r,k,r==k);
```

The output of this program is `0.800000,0.800000,0`. Both `r` and `k` appear to have value 0.800000, but due to rounding errors, the test for equality `r==k` evaluates to false, which is represented as a 0-value integer in C. Hence, floating-point numbers should not be used as the primary representation for time if there is to be a clean notion of simultaneity. Unfortunately, in FMI 2.0 and many other simulation frameworks, it is exactly the representation that is used. This is problematic.

2.2 Superdense Time

A model of time that is particularly useful for hybrid co-simulation is superdense time [13,33,34,35]. Superdense time is supported by FMI-ME, but not by FMI-CS. Fundamentally, superdense time allows two distinct ordered events to occur in the same signal without time elapsing between them.

A superdense time value can be represented as a pair (t, n), called a **time stamp**, where t is the **model time** and n is a **microstep** (also called an **index**). The model time represents the time at which some event occurs, and the microstep represents the sequencing of events that occur at the same model time. Two time stamps (t, n_1) and (t, n_2) can be interpreted as being **simultaneous** (in a weak

sense) even if $n_1 \neq n_2$. A stronger notion of **simultaneity** would require the time stamps to be equal (both in model time and microstep).

Superdense time is ordered lexicographically (like a dictionary), which means that $(t_1, n_1) < (t_2, n_2)$ if either $t_1 < t_2$, or $t_1 = t_2$ and $n_1 < n_2$. Thus, an event is considered to occur before another if its model time is less or, if the model times are the same, if its microstep is lower.

An **event** is value with a time stamp. Time stamps are a particular realization of **tags** in the tagged-signal model of [32]. They provide a semantic ordering relationship between events that can be used in software simulations of physical phenomena and also in the programming logic on the cyber side of a cyber-physical system. But computers cannot perfectly represent real numbers, so a time stamp of form $(t, n) \in \mathbb{R} \times \mathbb{N}$ is not realizable in software. Many software systems approximate a time t using a double precision floating-point number. But as we noted above, this is not a good choice. We examine alternatives below.

The microstep can also be problematic for software, because in theory, it has no bound. But computers *can* represent unbounded integers (assuming that memory is unbounded), although the implementation cost of doing so may be high, and the benefit may not justify the cost. The microstep, therefore, should either be represented using a bounded integer (such as a 32-bit integer), or not represented at all. With some care in simulator design, it may be possible to never construct an explicit representation of the microstep, and instead rely only on well-defined ordering of time stamped values. Microsteps can implicitly begin at zero and increment until a signal stabilizes. This is the approach used in FMI-ME, where there is no explicit microstep, and yet, superdense time is supported. By contrast, in FMI-CS version 2.0, microsteps are explicitly disallowed [40].

2.3 Integer Time

Given that floating-point numbers are a problematic representation of time, what should we use? An obvious alternative is integers. We postulate that a hybrid co-simulation extension must use integer numbers in some way to represent the progress of time for coordinating FMUs. But how, exactly? And at what cost?

Integers are typically represented in a computer using a fixed number of bits. E.g., a C `int32_t` is a 32-bit, two's-complement integer. A `uint32_t` in C is a 32-bit unsigned integer. Note that the integer values can be *interpreted* as representing a time value with some arbitrary units. For example, we might interpret an integer value as having units of microseconds, in which case a value 100, for example, represents 0.0001 seconds.

Integers can be added and subtracted without quantization errors, a key property enabling a clean semantic notion of simultaneity. For example, suppose that one discrete-event signal has regularly spaced events with a period of $p_1 = 3$, and another has regularly spaced events with a period of $p_2 = 1$, both beginning at time 0. The times of the events in the first signal are $0, p_1, p_1 + p_1, p_1 + p_1 + p_1, \cdots$, and the times of the events in the second signal are $0, p_2, p_2 + p_2, p_2 + p_2 + p_2, \cdots$. Then no matter how these additions are performed, every third event in the second signal will be simultaneous with an event in the first signal.

Again, we have no such assurance with floating-point numbers. For example, suppose that we are using the IEEE 754 double precision floating-point standard,

and we let $p_1 = 0.000003$ (3 microseconds). If we add p_1 to itself 12 times, performing $(\cdots((p_1 + p_1) + p_1) + p_1 \cdots)$, then the result is 0.0000035999999999999999. On the other hand, if we let $q = ((p_1 + p_1) + p_1)$, then $(((q + q) + q) + q)$ yields 0.0000036. The results are not equal.

If signals are continuous, then such small differences in time have very little effect on system behavior. But if signals are discrete, then any difference in time can change the order in which events occur, and the potential effects on system behavior are not bounded.

One possible solution is to explicitly use an error tolerance when comparing two floating-point numbers. For example, suppose we assume an error tolerance of 100 nanoseconds. That is, we consider two times to be simultaneous if their difference is less than 100 nanoseconds. Then the above two times are simultaneous. But now consider three times, $t_1 = 0.0000036$, $t_2 = 0.00000367$, and $t_3 = 0.00000374$. Then t_1 is simultaneous with t_2, and t_2 is simultaneous with t_3, but t_1 is not simultaneous with t_3. Surely we would want simultaneity to be a transitive property!

An alternative to floating-point numbers is rational numbers. A time value could be given by two unsigned integers, a numerator and denominator. Addition of two such numbers will require first finding the least common multiple M of the denominators, then scaling all four numbers so that the two denominators equal M. Then the numerators can be added, and the denominator of the result will be M. However, this makes addition a relatively expensive operation, unless measures are taken to ensure that denominators are equal. Such measures, however, are equivalent to reaching agreement across a model on a time resolution, so we believe that a simpler solution uses an integer representation of time with an agreed resolution. It is also much more difficult to determine when overflow will occur with rational numbers. For example, if denominators are represented using 32-bit unsigned integers, and two times with denominators 100,000 and 100,001 are added, will overflow occur?

Suppose we adopt an integer representation of time. What units should we choose? We could start by considering existing integer representations of time. For example, VHDL, a widely used hardware simulation language, uses integer time with units of femtoseconds. Another example is the Network Time Protocol (NTP) [37], a widely used clock synchronization protocol that sets the current time of day on most computers today. NTP represents time using two 32 bit integers, one counting seconds, one counting fractions of a second (with units of 2^{-32} seconds). This can be treated as an ordinary 64-bit integer with units of 2^{-32} seconds (about 0.23 nanoseconds). IEEE 1588 [16], a more recent clock synchronization protocol, is designed to deliver higher precision clock synchronization on local area networks. A time value in IEEE 1588 is represented using two integers, a 32-bit integer that counts nanoseconds, and a 48-bit integer that counts seconds.

NTP and IEEE 1588 are designed to coordinate notions of time across a network. All participants in such a network agree to a time resolution (based on a resolution of 2^{-32} seconds for NTP, 1 nanosecond for 1588). The first requirement in Broman et al. [8] stipulates simply that the time resolution should be the same for all observers *in a model*. It need not be the same across models. In fact, simulation models tend to have very different time scales; high-speed circuits require femtoseconds while astronomy may only require years. Co-simulation involves the coupling of independent models that are coordinated in a black-box manner, each of which can operate at a different time resolution. From the perspective of the

master that coordinates the exchange of data between components, however, all components must be understood to progress in increments that are multiples of the time resolution used by the master.

In Ptolemy II [44], the time resolution is a single global property of a simulation model, shared by all components. The resolution is given as a floating-point number, but the time itself is given as an integer, representing a multiple of that resolution. All arithmetic is done on the integer representation, and the unit is only used when rendering the resulting times for human observation. E.g., if the resolution is given by the floating-point number `1E-10`, which in units of seconds denotes 0.1 nanoseconds, then the integer 10,000,000 will be presented to the user as 0.001 seconds.

Integers are, of course, vulnerable to overflow. Adding two integers can result in an integer that is no longer representable in the same bit format. Subtracting two unsigned integers can result in a negative number, which is not representable using an unsigned integer.

Whether and when an overflow occurs depends on the resolution, but also on the *origin* (what time is zero time). NTP and IEEE 1588 both set time relative to a fixed zero time, which in the case of NTP is 0h January 1, 1900, and in the case of 1588 is 0h January 1, 1970, TAI (International Atomic Time). Sometime in the year 2036, 2^{32} seconds will have elapsed since January 1, 1970, and all NTP clocks will overflow. IEEE 1588 uses 48 bits, so the first overflow will not occur for approximately 9.1 million years. If we define the time origin to be, say, the start time of a simulation, then the NTP representation will be able to simulate approximated 62 years before its representation of time overflows. A VHDL simulator using a 64 bit integer representation of time with units of femtoseconds can simulate approximately 2.56 hours of operation before overflow occurs, so clearly choosing the origin to be January 1, 1900 would not be reasonable. In Ptolemy II, overflow cannot occur, because the integer representation of time uses an unbounded data structure to represent an arbitrarily large integer.[1] And the resolution is a parameter of the model, so Ptolemy II simulations can handle high speed circuits as well as astronomical simulations.

In computers, addition and subtraction of integers is extremely efficient. In the IEEE 1588 representation, however, the two numbers cannot be conjoined into a single number, and arithmetic on the numbers must account for carried digits from the nanoseconds representation (32 bits) to the seconds representation (48 bits). Since computers do not have hardware support for such arithmetic, such a representation will be more computationally expensive to support. Adding two IEEE 1588 times takes quite a few steps in software. For the Ptolemy II unbounded integers, addition and subtraction are also potentially more expensive than addition on ordinary 32 or 64-bit integers, but the cost is not as high as for IEEE 1588 because overflow is more easily detected in the hardware.

In modern computers, addition and subtraction of 32 and 64-bit integers is at least as fast as addition and subtraction of floating-point numbers, and it requires significantly less energy. Multiplication, however, is a more complicated story. The problem with multiplication of integers lies in the units. Consider two integers with

[1] Strictly speaking, overflow *can* occur in the sense that the machine may run out of memory to represent the integer times. But this would occur at such absurdly large times, beyond the age of the universe with any imaginable resolution, that it is simply not worth worrying about.

units of microseconds. If we multiply the two times, the units of the result will be microseconds squared. First, this is not a time, and hence there is no reason to insist that this result be represented the same way times are represented. Second, whether the result is representable in a 32 or 64-bit integer will depend on the origin and resolution of the times.

Multiplication of two times, however, is a relatively rare operation. A more common operation is multiplication of a time by a unitless scalar. For the example above, instead of adding p_1 to itself 12 times, we might have multiplied $12 * p_1$. As long as there is no overflow, such multiplication will typically be performed without quantization error and reasonably efficiently in a computer. However, not all processors have hardware support for integer division. And multiplication by a non-integer, as in $0.1 * p_1$, will yield a floating-point representation of the result, not an integer representation. Hence, it will be vulnerable to quantization errors.

We claim that for the purposes of coordinating FMUs, addition, subtraction, and multiplication by integers are mostly sufficient, and hence an integer representation of time can be very efficient. Within an FMU, however, there may be more complex operations involving time, and the FMU may include legacy software or ODE solvers that use floating-point representations of time. Such FMUs will suffer a (hopefully small) cost of conversion of time values at the interface. Presumably, since such FMUs already tolerate quantization errors inherent in a floating-point representation, any errors that are introduced in the conversion process will also be tolerated. For example, such FMUs should never compare two times for equality, because if they are using a floating-point representation of time, such a comparison is meaningless. They should also not have behavior that depends strongly on the relative ordering of two time values.

We choose to represent time with a 64-bit unsigned integer with arbitrary resolution, where the resolution is a parameter of the model, and origin equal to the simulation start time. It is computationally efficient on modern machines. And for well-chosen resolutions, will tolerate very long simulations without overflow. It is also easily converted to and from floating-point representations (with losses, of course). Also, given the enormous range of time scales that might be encountered in different simulation models, choosing a fixed universal resolution that applies to all models probably does not make sense. We believe further that all the acceptance criteria of [8] can be met without an explicit representation of the microstep.

2.4 The Choice of Resolution

The only remaining issue is how to choose the resolution. There are two questions here. First, what data type should be used to represent the resolution? Second, should an FMU be able to constrain the selected resolution?

The latter question seems relatively easier to answer. In hybrid co-simulation, an FMU may encapsulate considerable expertise about a system that it models, and the FMU's model may only be valid over a range of time scales. It seems reasonable, therefore, that an FMU should be able to insist on a resolution. On the other hand, to be composable with other FMUs, the FMU should be capable of adapting to a finer resolution than the one it requests. If two FMUs provide different resolutions, or if their resolution differ from the default resolution of the simulation, then how should the differences be reconciled?

We see two possibilities:

(i) The selected resolution for the model is the finest of all specified resolutions.
(ii) The selected resolution for the model is the greatest common divisor (GCD) of all specified resolutions.

Whether the second option is even possible depends on the data type used for the resolution. Here, we see several possibilities:

(a) **double**. In Ptolemy II, all components share a single double precision floating-point number, the unit, which specifies the resolution of the model. All timestamps are interpreted as an integer multiple of this value.
(b) **rational**. Alternatively, resolution can be specified given as a pair of integers, a numerator and a denominator. In this case, it is always possible in theory to find a GCD, although there is risk of overflow if the numerator and denominator are represented with a bounded number of bits. In addition, conversion to and from a floating-point representation, which is often needed internally by an FMU, may be costly.
(c) **decimal**. Finally, the resolution can also be specified using integer exponent n that stipulates a resolution of 10^n seconds. For instance, IEEE 1588 resolution is achieved with $n = -9$, VHDL resolution is achieved with $n = -12$. Using a decimal resolution, the finest resolution is always the same as the GCD and is always precise.

Defining resolution as a power of ten has the very nice feature that any parameter that is specified using a decimal representation, such as 0.3332, is exactly representable, with no quantization error, as long as the resolution is sufficient, for example $n = -4$ for 0.3332. In contrast, when such a decimal number is converted to a binary floating-point representation, errors may be introduced. Since it is extremely common to give parameter values in decimal, this advantage cannot be ignored.

Also, since parameters are often related to one another, the ability to, for example, calculate the difference between two parameter values without quantization error can be important. For instance, if a component specifies using parameter values that it produces events at times t_1 and t_2, given in decimal, then the time interval $t_2 - t_1$ can be calculated without error.

If parameter values are not given in decimal, for example "1/3," then decimal resolution is not sufficient to avoid quantization errors. Such errors can be avoided by selecting rational resolution instead. A rational resolution has the advantage that if an FMU internally performs computation according to its specified resolution, then simultaneity of any events it generates compared to events generated by other similar FMUs is well defined. The simultaneity of such events will not be subject to quantization errors. However, this choice comes at a possibly considerable cost in converting time values to and from floating-point numbers. And, as we have noted, this choice has a more complicated overflow risk.

For these reasons, we prefer option (c), which also implies option (i). An FMU can stipulate a minimum required resolution and be guaranteed that resolution or a resolution that divides its resolution by some power of ten:

$$r = 10^n \tag{1}$$

However, not every FMU may be prepared to adapt to a finer resolution than the preferred resolution it declares. For instance, an FMU may generate events exclusively at time instants that are multiples of its declared time resolution. For such an FMU there is no reason to adopt a finer resolution than the one it specifies; it would merely complicate the design of the FMU. For that reason, we believe that this capability should not be mandatory, rather it should be optional. In the next section, we describe an architecture that accommodates such flexibility.

3 Hybrid Co-simulation with Integer Time

In order to examine the effects of using integer time in hybrid co-simulation we need a practical framework for our analysis. Instead of defining our own, we leverage the existing FMI-CS 2.0 standard, and extend it to use integer time. In addition, in order to support discrete events, we enrich the interface to encode the absence of an event and allow FMUs to react instantaneously, i.e., without moving forward in time. We call this framework FMI-HC (FMI for Hybrid Co-simulation).

3.1 Extensions to the FMI standard

As a consequence of using integer time, the FMUs and MA need to agree on a resolution before the simulation starts. Two new functions are introduced for this: `getPreferredResolution` and `setResolution`. In addition, we introduce a hybrid step function `doStepHybrid` that uses integer time instead of doubles, a function `getMaxStepSizeHybrid` that returns the maximal allowed communication step size, and the functions `getHybrid` and `setHybrid` that in addition to the exchange of regular signal values can also communicate "absent," to indicate that there is no value present at the corresponding time instant.

3.1.1 Advancing Time

In FMI, simulation is driven by a master that keeps time and instructs FMUs to advance their time in increments called "steps." Once all participating FMUs have advanced their time by some delta, an iteration has finished. In each iteration, FMUs exchange data, the master proposes a new step, and so forth. The specifics of this sequence are encoded in a master algorithm. Only the interface of the FMU is standardized. FMUs are prescribed to advance time through calls to the function `doStep`. In the FMI-CS 2.0 standard, this function has the following signature:

```
fmi2Status fmi2DoStep(
    fmi2Component c,
    fmi2Real      currentCommunicationPoint,
    fmi2Real      communicationStepSize,
    fmi2Boolean   noSetFMUStatePriorToCurrentPoint);
```

The first parameter points to a particular FMU. The second parameter states the current time, using a double-precision floating-point value (named `fmi2Real` in the standard). The third parameter states the communication step size, which is the time interval over which the master requests the FMU to advance. Finally,

the fourth parameter provides information as to whether any rollbacks can occur, which allows the FMU to abandon any kept state if this parameter is `true`.

Clearly, this function is not suitable for communicating an integer step size to the FMU. For backward compatibility, we preserve the original `fmi2doStep` function and add a new function that is used to advance hybrid co-simulation FMUs using integer time steps[2]:

```
fmi2Status doStepHybrid(
    fmi2Component    c,
    fmiXIntegerTime  currentCommunicationPoint,
    fmiXIntegerTime  communicationStepSize,
    fmi2Boolean      noSetFMUStatePriorToCurrentPoint),
    fmiXIntegerTime* performedStepSize);
```

Instead of using `fmi2Real` data type, we use the type definition `fmiXIntegerTime`, a 64-bit unsigned integer type. The additional parameter `performedStepSize` is used for communicating back to the master the size of the *performed* step, which could be smaller than the *requested* step, `communicationStepSize`. If the performed step size is equal to the requested step, then the FMU has accepted the requested step. If the performed step is smaller than the requested step, then the FMU has rejected the requested step, but nevertheless advanced to time `currentCommunicationPoint` + `performedStepSize`.

As observed by Broman et al. [7], adding a function `fmiGetMaxStepSize` makes it possible for an FMU to state a predictable step size. Such function with integer time can be defined to have the following signature:

```
fmi2Status getMaxStepSizeHybrid(
    fmi2Component    c,
    fmiXIntegerTime* maxStepSize);
```

This function returns an upper bound of the step size that the FMU will accept on the next invocation of `doStepHybrid`. The master algorithm should query this function before calling `doStepHybrid`.

3.1.2 Negotiating the Resolution of Time

In the previous section, we explained that FMUs may be designed to preferably (or exclusively) operate at some specific time scale. To accommodate this, an FMU must be able to inform the master of its preferred resolution, which can be done by extending FMI for hybrid co-simulation with:

```
fmi2Status getPreferredResolution(
    fmi2Component               c,
    fmiXTimeResolutionExponent* n);
```

[2] Our proposed extension does not target a specific version of FMI. For this reason (and for brevity), we have removed the prefix "`fmi2`" from all newly proposed functions. Newly introduced datatypes honor the naming convention but have the FMI version number replaced by a wildcard, "`X`."

The preferred resolution is returned as an integer using the second parameter. As described in the previous section, the `resolution` represents an integer n, that stipulates a resolution of 10^n seconds.

Although it is important that an FMU can express its preferred resolution, we will also show the need for the master to explicitly state that the FMU should use a specific resolution. We define the following function to enforce this behavior:

```
fmi2Status setResolution(
    fmi2Component              c,
    fmiXTimeResolutionExponent n);
```

3.1.3 Discrete Events

The aforementioned functions are introduced to offer support for integer time. In order to support discrete signals, an FMU must be able to output or take in discrete events, which are present only for a duration of zero time (one microstep in superdense time) and absent otherwise.

The FMI standard defines two kinds of functions for setting and getting input and output signal values: `fmi2SetXXX` and `fmi2GetXXX`. There are different functions for different variable types. The substring `XXX` is a placeholder for the type[3]. For instance, `fmiSetReal` is the function that is used to set input signal values of type `fmi2Real`, which is implemented using double-precision floating-point numbers.

In the FMI-CS 2.0 standard, values exchanged between FMUs are always present. This means that the current co-simulation standard does not yet have the support for discrete events. To make it possible to express discrete events, FMI needs to have functions for setting and getting values, where the values can be stated to be either present or absent. By extending the current standard get and set functions, we obtain the following signatures:

```
fmiStatus setHybrid (
    fmi2Component            c,
    const fmi2ValueReference vr[],
    size_t                   nvr,
    const fmi2XXX            value[],
    const fmi2SignalStatus   flag[]);

fmiStatus getHybrid(
    fmi2Component            c,
    const fmi2ValueReference vr[],
    size_t                   nvr,
    fmi2XXX                  value[],
    fmiXSignalStatus         flag[]);
```

The first argument points to the FMU to set values for or get values from. The second argument, `vr`, is an array of identifiers that refer to specific variables. Furthermore, `nvr` is the size of that array, and the array `value` (also of length `nvr`) specifies the values that should be set or gotten. The i_{th} element from `value` is

[3] For simplicity, we omit this implementation detail from the remainder of the discussion and refer to these functions without the “`XXX`” wildcard suffix.

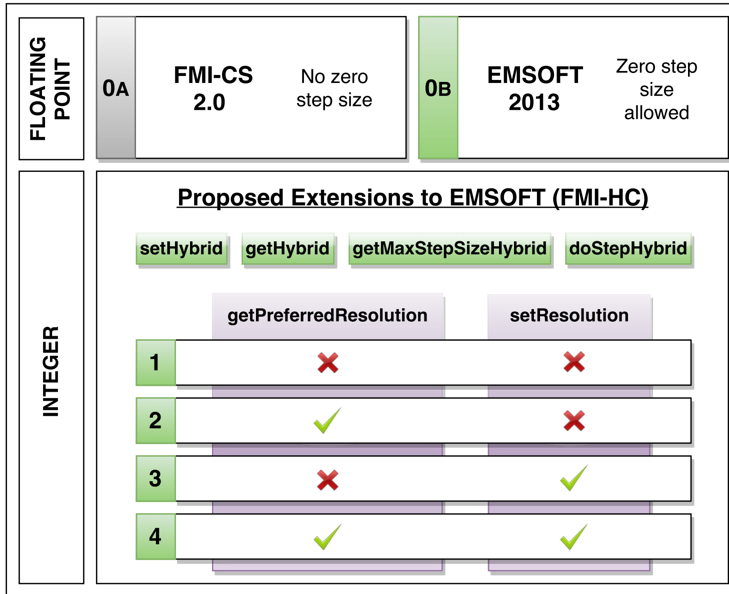

Fig. 3 A taxonomy for different categories of hybrid co-simulation FMUs.

assigned to or read from the i_{th} variable declared in `vr`. These function arguments are the same as in the original `fmi2Get` and `fmi2Set` functions in the FMI-CS 2.0 standard. The "hybrid" get and set functions introduce an additional argument, `fmiXSignalStatus`, which is defined as:

```
typedef enum {present, absent} fmiXSignalStatus;
```

If `flag[i] == present`, the signal is considered to be present and the value of the variable `vr[i]` is `value[i]`. In case `flag[i] == absent`, the signal is not present and the `value[i]` of variable `vr[i]` should be ignored. Note that there are many alternative ways of extending the standard with capabilities of expressing absent and present signals. For instance, instead of creating new `get` and `set` functions, separate functions can be used for indicating if a signal is present or absent. However, these implementation details are outside the scope of this paper and would be a decision for the FMI steering committee.

3.2 Categories of FMUs in FMI-HC

Hybrid co-simulation should be able to work with "legacy" FMUs that do not implement any of the FMI-HC extensions. Moreover, to make it easy to write FMI-HC FMUs, we do not wish to require that every FMU implement every extension. We can divide FMUs into different categories based on the extensions they implement. It is useful to organize FMUs in a taxonomy based on this criterion because different category FMUs require different handling by the master. We use the taxonomy presented in Figure 3 throughout the remainder of the paper to identify and refer to FMUs in terms of their category.

- **Category 0:** An FMU in this category internally uses floating-point numbers to represent time (see the top of Figure 3). This category can be further refined into two sub categories. Category 0_A denotes an FMU that is compatible with existing FMI 2.0 co-simulation FMUs. Such FMUs are *not suitable* for hybrid co-simulation because the standard disallows zero step-sizes and insists on always calling `doStep` (advance time) in between `set` (providing inputs) and `get` (retrieving outputs). The former rules out the use of superdense time while the latter prohibits the handling of direct feedthrough loops. Therefore, we do not further discuss category 0_A FMUs in this paper. In contrast, Category 0_B follows the assumptions in [7], which allows a zero step size and getting and setting values (multiple times) without having to advance time.
- **Category 1:** This is the first out of four possible categories of FMUs that use integers to represent time. Note that categories $1 - 4$ represent the exhaustive combinations of `getPreferredResolution` and `setResolution`. In category 1, neither of these functions is supported. The operation of an FMU in this category is time invariant; it does not use time to determine its outputs or state updates. Such a component can implement, for example, a time-invariant memoryless function such as addition.
- **Category 2:** In category 2, function `getPreferredResolution` is supported, but `setResolution` is not. This means that the FMU states which resolution it will use, but does not allow the master to change its resolution. That is, the resolution is actually *required*, not just preferred. A composition of multiple category 2 FMUs may result in a heterogeneous model with respect to the resolution of time. Category 2 FMUs are natural to use in cases where the FMU should output data at periodic time internals, e.g., periodic samplers or signal generators. Tools that have a fixed time resolution, such as Rhapsody from IBM or VHDL programs, would produce FMUs of this category.
- **Category 3:** FMUs in this category support `setResolution`, but do not support `getPreferredResolution`. This means that the FMU *is* using the integer notation of time (in contrast to category 1), but any resolution is acceptable. For instance, a zero-crossing detector would fall into to this category.
- **Category 4:** An FMU in this category supports both `getPreferredResolution` and `setResolution`. This means that the FMU may first communicate to the master the resolution that it prefers, and be followed by the master telling the FMU what resolution it should use. An ODE solver FMU would belong to this category.

3.3 Modular Support for FMI-HC

Technically, an FMU developer can choose whether or not to support functions like `getPreferredResolution` or `setResolution` and notify the master of the functions it supports through so-called capability flags in the FMU's accompanying XML-file, as prescribed by the FMI standard. With this approach, the master algorithm must accommodate the use of all the different categories of FMUs in Figure 3.

Figure 4 depicts an architectural view of how an FMI simulation tool can interact with different category FMUs modularly, without drastic changes to its simulation engine.

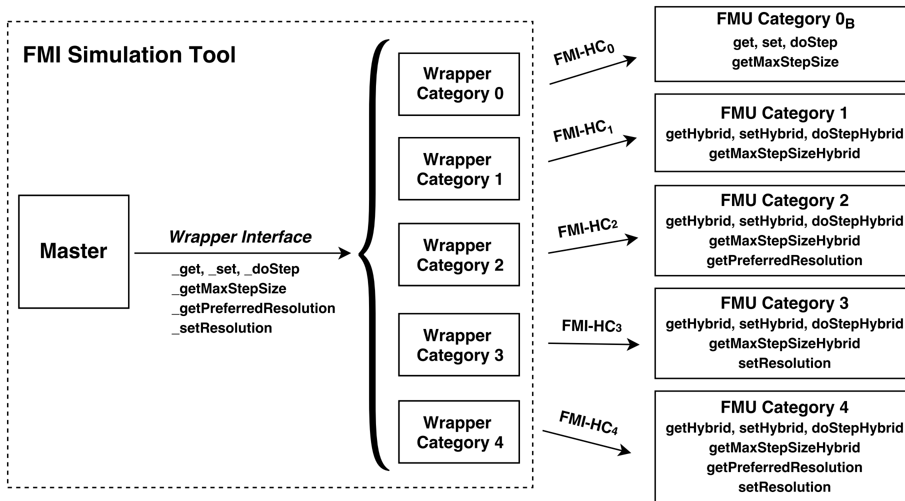

Fig. 4 A generic architecture for supporting hybrid co-simulation using wrappers. The left side of the figure shows an (arbitrary) internal interface between the wrappers and the simulation tool, the right part shows the new FMI-HC extensions used by the wrappers.

In the left part of the figure, the dashed line represent an FMI simulation tool that takes a set of connected FMUs as input and produces a simulation result as output. The basic idea is to separate the concerns of the master algorithm from the logic that handles the translation between different resolutions for different categories of FMUs. This logic is instead encoded into *wrappers*, that is, software components that translate function calls from the master algorithm to the FMUs. Both the implementation of the master algorithm component and the wrapper components are internal to a specific tool implementation. The design discussed in this article does not assume any specific implementation language. Hence, the wrapper interface functions (with prefix "_") are arbitrary; tool vendors can choose the specifics of their wrapper interface as they see fit and are not tied to a specific programming language for the implementation of their execution engine.

The right part of the figure depicts all different categories of FMUs along with the particular FMI-HC extensions they implement. The wrapper treats the FMU as a black box and performs the conversion between the model of time used by the master and the model of time used by the FMU. This can either be a conversion between integer time resolutions (for a category 2 FMU) or a conversion between integer and floating-point time (for a category 0_B FMU).

3.4 An Implementation of FMI-HC

The interface extensions described in this paper can be used by making relatively small adaptations to existing master algorithms. In a nutshell, it requires adopting our integer-based representation time, using specific wrappers for FMUs based on their category, and letting the master negotiate a time resolution as part of its initialization procedure.

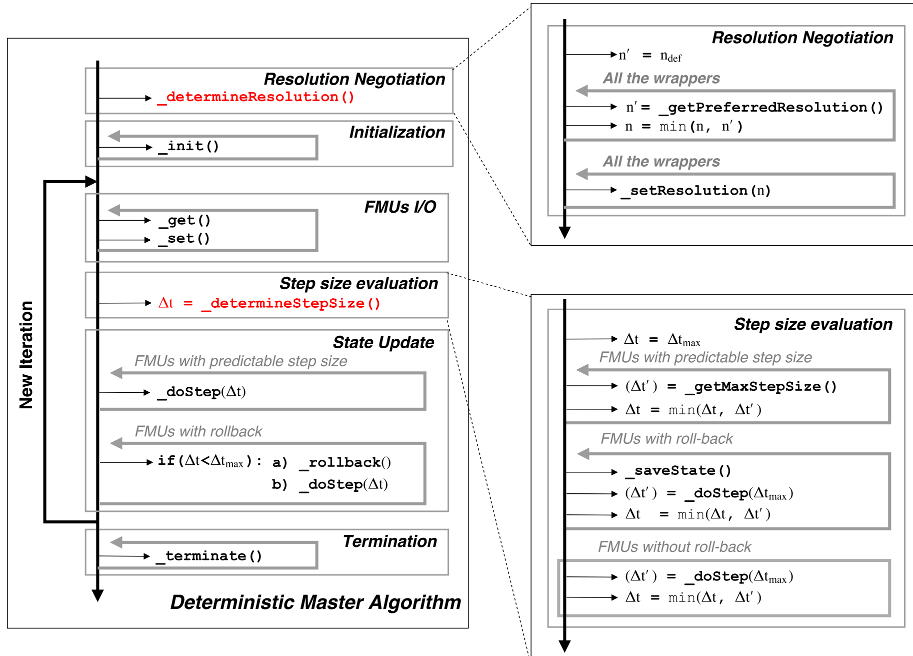

Fig. 5 Schematic description of a variant of the deterministic master algorithm by Broman et al. [7], extended to accommodate integer time. The figure and implementation are based on Cremona et al. [15].

In the following, we outline how to make these adjustments based on our own reference implementation called FIDE [15], an FMI Integrated Development Environment. FIDE implements a master algorithm based on the work of Broman et al. [7]. It is capable of deterministically simulating mixtures of continuous-time and discrete-event dynamics, has a superdense model of time [33,34], and features extended data types to support an explicit notion of *absent*. Superdense time is modeled by allowing the master to take zero-size steps, allowing the simulation to iterate over a number of lexicographically ordered indexes before it advances to the next Newtonian time instant. The absence of events in signals is enabled through the FMI-HC functions `getHybrid` and `setHybrid` described in Section 3.1.3.

Figure 5 provides a schematic description of the MA implemented in FIDE. Any tool that supports FMI will feature an execution engine much like the one in FIDE. The simulation tool reads a model description that describes how a set of FMUs is connected. It loads each FMU by reading the FMI XML-file and dynamically linking the required C libraries as it normally would. However, in order to accommodate FMI-HC, each FMU is now identified by category and a matching wrapper object is instantiated for every FMU. The wrapper is specifically designed to interface with FMUs of a particular category. Since all wrappers are using the same interface (all are using integer time), the logic of the execution engine is not complicated by the different ways that FMUs may interpret time: all the necessary conversions are performed by the wrappers. For instance, if a

category 0_B FMU is used, the wrapper is handling the correct conversion between integer time and floating-point time.

During initialization, the master calls the function `_determineResolution()` that determines the time resolution for the simulation. This function iterates over all the wrappers and queries them for the time resolution exponent using `_getPreferredResolution`. The time resolution exponent of the simulation is computed as the minimum among a default value and the resolution exponents obtained from all the FMUs that partake in the simulation. The chosen resolution exponent is then communicated to the wrappers using `_setResolution`. The wrappers will eventually use it to convert the integer time stamps used by the master to whichever model of time is used internally by the wrapped FMU, if necessary.

FIDE must keep track of the global time of the master and the current step size using integers. Hence, it cannot used the `fmi2Real` data type that is prescribed by FMI-CS 2.0 for this purpose. We use a new data type, `fmiXIntegerTime`, instead. Finally, all direct calls from the master to the FMU functions `fmi2DoStep`, `getMaxStepSize`, `get`, and `set` have to be removed and replaced by function calls to the corresponding intermediate functions provided by the wrappers.

4 Time Conversion and Quantization

Using integers for representing time does not completely remove time quantization errors: an FMU may still use a floating-point number internally for time keeping. In such case, conversion from the floating-point representation of the time kept inside of the FMU to the integer time used by the master comes with a loss of precision. Specifically, the effects of time quantization come into play when a category 0_B FMU rejects a proposed step size and makes partial progress over an interval of which the length (a floating-point number) cannot be losslessly converted into a corresponding fixed-resolution integer time for the master to interpret. Similar problems arise when there is a mismatch in time resolution between the master and a category 2 FMU. Quantization errors also result when a higher-resolution integer time is converted to a lower-resolution integer time. For instance, the master may instruct a category 2 FMU to take a step that is too small to represent with the resolution that the FMU uses internally.

The key insight here is that in co-simulation participants are treated as a black box; each component has its own isolated understanding of time that is based on the characteristics of its local clock. Each level of hierarchy in a co-simulation gives rise to a different **clock domain** in which the passing of time may register differently from another clock domain. The degree to which two components can be synchronized therefore depends on compatibility of their clock domains. The issue of translating time across different clock domains gets complicated by corner cases, which, if not handled appropriately, may lead to Zeno behavior or may cause discrete events emitted by one component to be missed by another. These kinds of issues play a role *only* in the interaction with category 0_B and category 2 FMUs. The former does not admit integer time and hence requires an conversion from and to integer time, and the latter cannot adapt its resolution to its environment, which requires a conversion between integer times. On the other hand, category 1 FMUs have no time resolution at all, and therefore their behavior must be time-invariant, while categories 3 and 4 can adapt their resolution and therefore synchronize

perfectly with their environment. Hence, category 0_B and category 2 FMUs are the main focus of the remainder of this section. A full C implementation of wrappers sufficient to co-simulate any combination of FMUs of any of the aforementioned categories is given in Appendix A.

4.1 Converting from Integer to Real-valued Time

In the following, we will show the relationship between a 0_B FMU's internal notion of time and its environment's integer representation, with the assumption that the FMU internally represents time as a *real* number with no quantization errors. Please recognize that this is impossible in a computer, and the FMU will internally encode these real numbers as double-precision floating-point numbers. The environment's time resolution is given as a power of ten, $r = 10^n$, measured in seconds, where n is an integer. An integer time index, i, once scaled by a resolution, denotes a real-valued time:

$$t = i \cdot r, \tag{2}$$

also measured in seconds.

We express the step size for the master and the FMU separately, each in terms of a relative increment with respect to their local time representation. For the master, we define the new time index i' after having taken a step Δi with respect to the previous time index i to be $i' = i + \Delta i$, where i' corresponds to the time $i' \cdot r$. Similarly, for the FMU, we define the time after a step to be $t' = t + \Delta t$, where t is the current time in the clock domain of the FMU and Δt is the time step. When the master and the FMU agree on the next time step, we have have $t' = i' \cdot r$, and therefore $t + \Delta t = (i + \Delta i) \cdot r$. We derive the size of the time step of the FMU as follows:

$$\Delta t = (i + \Delta i) \cdot r - t. \tag{3}$$

4.2 Converting from Real-valued to Integer Time

No matter how fine a time resolution we choose, an arbitrary real-valued time instant is unlikely to align perfectly with some integer-time instant. As a result, conversion from a real-valued time to integer time can introduce a time quantization error up to one unit of the integer time resolution. Notice that this quantization error is controlled by the user, modeler, or tool integrator through the simulation parameter r, the time resolution. The finer the resolution, the smaller the quantization error.

Ideally, according to (3), the FMU's step size in the clock domain of the master should be:

$$\Delta i = \frac{t + \Delta t}{r} - i. \tag{4}$$

But in general, this will not yield an integer Δi, and we require Δi to be an integer. We have three alternatives to obtain an integer step from a real-valued time step: (i) take the floor (the largest smaller integer), (ii) take the ceiling (the smallest

larger integer), or (iii) round (the nearest integer). Our implementation uses the ceiling operator, rounding to the next larger integer:

$$\Delta i = \left\lceil \frac{t + \Delta t}{r} \right\rceil - i. \tag{5}$$

This choice prevents quantization errors from blocking the progress of time. To see this, consider the model depicted in Figure 6. This model consists of two category 0_B FMUs composed in parallel. FMU A outputs a piecewise-constant signal with a discontinuity at $t = 0.4$, while FMU B outputs a discrete event at $t = 0.8$. Assume that these FMUs are coordinated by an integer-time master and interfaced through wrappers. Both FMUs use floating-point time internally, and the wrapper provides the glue code between the master and the FMU. Specifically, the wrapper's `_getMaxStepSize` and `_doStep` functions implement the conversions between integer time and floating-point time in accordance with Equations (3) and (5), and determine when and how the FMU is allowed to advance time.

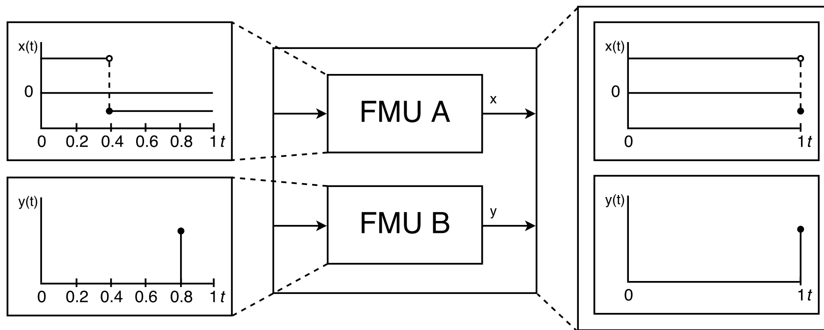

Fig. 6 The outputs of two FMUs composed in parallel. The signal x shows a discontinuity that registers at $t = 0.4$ in the clock domain of FMU A. The signal y shows a discrete event that registers at $t = 0.8$ in the clock domain of FMU B. The discontinuity and the discrete event, however, occur at the same time, $t = 1$, in the clock domain of the master.

For simplicity, we assume the master adopted a time resolution of 1 second ($n = 0$). At time $t = 0$ the master calls `_getMaxStepSize` of the FMU A wrapper (FMU B does not implement `_getMaxStepSize`, and hence gives no indication as to what step size it will accept). The FMU A wrapper then calls `getMaxStepSize` which returns $\Delta t = 0.4$. Because of the master's time resolution $r = 1$, Δt cannot be represented exactly in terms of multiples of r. Therefore, using the ceiling operator as its quantization method, the wrapper reports back 1, indicating to the master that the FMU will accept a step size of size 1. Notice that had we used the floor operator instead, the wrapper would have returned 0, and the simulation would have gotten stuck forever at $t = 0$ because the master will proceed with the *smallest* of the step sizes that the wrappers return through `_getMaxStepSize`. Similarly, if we had used rounding, and rounding of 0.4 returns 0, then again, the simulation would get stuck.

The master will next invoke `_doStep` with a proposed step size of 1. It can invoke this function in either order, first for FMU A, or first for FMU B. Assume FMU B goes first. It will reject the step and indicate that it has made progress up to

time 0.8. But that time is not representable in integer time either, so the wrapper rounds it up using the ceiling function. Since $\lceil 0.8 \rceil = 1$, the wrapper *accepts* the step, but makes an internal annotation that the FMU only progressed to time 0.8. The next invocations of `get` and `set` will provide inputs and retrieve outputs that for the master will appear to occur at time 1, but will look to FMU B as if they occur at time 0.8.

The procedure for FMU A is similar, but the wrapper has a bit more information to work with, since the FMU has previously indicated that it would accept a maximum step of 0.4. Hence, when the master proposes a step of size 1, the wrapper can propose a step of 0.4 to the FMU. The next invocations of `get` and `set` will provide inputs and retrieve outputs that, again, for the master appear to occur at time 1, but will look to FMU A as if they occur at time 0.4.

Assume further that the output of FMU A has a discontinuity at time 0.4. This means that the FMU requires that the next invocation of `doStep` has a step size of 0. It indicates this by returning 0 when `getMaxStepSize` is called. The wrapper passes this on to the master by returning 0 in `_getMaxStepSize`. The master now has no choice but to propose a zero step size. Upon invocation of this zero step, FMU A advances in superdense time because of the discontinuity, so its local time remains at 0.4. Since FMU B produces a discrete event at this time, its local time will remain at 0.8. The outputs of both FMUs will appear to the master to occur at time 1.

Suppose that after this neither FMU has any anticipated events and therefore will accept any step size. Suppose the master proposes a step of size 10 to the wrappers. The wrappers will need to compensate for the lag of their FMUs, and instead propose a step of 10.6 and 10.2, respectively, to FMU A and FMU B.

It may be possible to design other wrappers with a *different* API that use the floor or other rounding functions instead of the ceiling function, but the our solution appears to be simple, to work well, to preserve causality, and to ensure that time continues to advance. We observe that using the floor in `_getMaxStepSize` will always make the FMU *lag behind* with respect to the master. For 0_B FMUs, the quantization effects due to the use of integer time only play a part in the conversion of time steps in the FMU's clock domain to time steps in the master's clock domain, not vice versa. Conversion from master time to FMU time suffers only from ordinary rounding that is a consequence of the floating-point representation of time inside the FMU.

4.3 Converting Between Different-resolution Integer Times

Conversions between times expressed in different-resolution clock domains can be derived in a similar fashion as shown in Section 4.2. These conversions are necessary for the support of fixed-resolution integer-time FMUs; the master might choose to operate using a different time resolution than a category 2 FMU. In such scenario, each time the master proposes a time step Δi, expressed as a multiple of the master's resolution, 10^n, this step must be converted into a time step Δj that is interpreted as a multiple of the FMU's resolution, 10^k. Assuming that the FMU has accepted the time step, Δi in clock domain of the master, Δj in the clock domain of the FMU, then $j' = j + \Delta j$ is the future time index of the FMU and $i' = i + \Delta i$ is the target time index of the master. After the step is

completed, we assume master and FMU have reached the same point in time, hence: $i' \cdot 10^n = j' \cdot 10^k$. We obtain Δj, the step to be taken by the FMU, as follows:

$$\Delta j = (i + \Delta i) \cdot 10^{n-k} - j. \tag{6}$$

Observe that the term 10^{n-k} in Equation (6) could be a fractional number (specifically, it is fractional when $n - k < 0$). This is possible because n shall always be smaller than or equal to k (as per the resolution negotiation procedure in Figure 5, the resolution of the master must be *at least* as fine as the resolution of the FMU). Therefore, since Δj must be an integer, quantization may be necessary. Just as we did for the time conversion method for category 0_B FMUs, we need to pick a quantization method for category 2 FMUs.

It should be noted that we can compute the floor or ceiling of Δj using solely integer arithmetic; there is no need for floating-point arithmetic for either of them. The floor can be implemented using an integer division that truncates toward zero, which is standard in C99 and most other contemporary programming languages. The ceiling function can also be implemented using integer division: if the division truncates to zero then $\lceil \frac{x}{y} \rceil$ can be computed using the following expression: `(x+y-1) / y`.

Using the ceiling operator as our quantization method allows the FMU to move ahead of the master (and any other higher-resolution FMUs), while using the floor operator would let the FMU lag behind. Neither of the two solutions is more accurate than the other, but given that we quantize time for 0_B FMUs such that they lag behind with respect to the master's clock, it would make sense to adopt the same policy for category 2 FMUs and therefore select the floor operator; to round to the previous smaller integer. Hence, we obtain:

$$\Delta j = \lfloor (i + \Delta i) \cdot 10^{n-k} \rfloor - j. \tag{7}$$

To convert a step in the clock domain of the FMU to a step in the clock domain of the master, we essentially use Equation 7, except here, we can omit the rounding. Because the master operates at a time resolution greater than or equal to the time resolution of the FMU, it must be that $k - n \geq 0$, so no rounding is ever needed. Hence, we compute a step in the master's clock domain based on a step in the FMU's clock domain as follows:

$$\Delta i = (j + \Delta j) \cdot 10^{k-n} - i. \tag{8}$$

It is important to emphasize that time quantization plays a different role for category 2 FMUs than it does for category 0_B FMUs. The former experience quantization only in the conversion from master time to time FMU time, while the latter experience quantization only in the conversion from FMU time to master time; the directionality of time quantization is opposite in comparison between the two. This observation also explains why it is no issue to use ceiling quantization for category 2 FMUs, because the Zeno condition described in Section 4.1 is due to loss of precision in the conversion from FMU time to master time, which for category 2 FMUs is lossless. A detailed description of the application of Equation (7) and (8) in the wrapper for category 2 FMUs can be found in Appendix A.

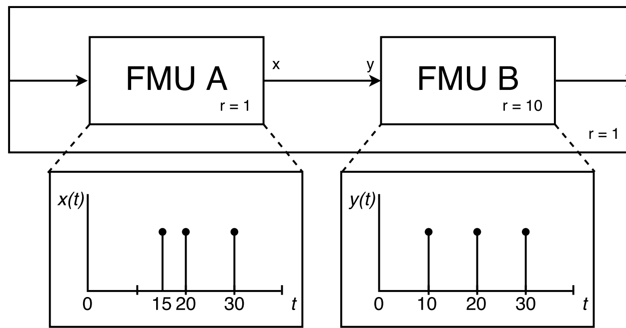

Fig. 7 Two category 2 FMUs in cascade composition. Each FMU can only update their input and output signal at times that are multiples of their resolution. The discrete event produced by FMU A at local time 15 is received at by FMU B at local time 10. No time quantization occurs with the discrete events produced at times 20 and 30.

Finally, let us examine the effects of time quantization using an example. Consider the model in Figure 7 that depicts two category 2 FMUs. The master, along with FMU A, uses a resolution of $1\,s$, while FMU B uses a resolution of $10\,s$. In other words, a step of size 1 in the clock domain of FMU B represents a step size of 10 in the clock domain of FMU A. Conversion the other way around, dividing by 10^1, may not yield a whole number and therefore incurs a quantization error.

Interestingly, the event emitted by FMU A at internal time index $j_A = 15$, which corresponds to time $t = 15$, will appear on the input of FMU B (due the use of the floor function) when it is at internal time index $j_B = 1$ which corresponds to time $t = 10$. Superficially, this may look like a causality violation, but it is not, because the two internal clock domains are completely isolated from each other. They are analogous to two people having a phone conversation, but where one is looking at a clock that is ahead compared to a clock the other is looking at. They cannot see each other's clocks. An outside observer (the master) has its own clock, which may differ from both the internal clocks (although in this particular example it is perfectly synchronized with FMU A because they use the same resolution). In all three clock domains, causality is preserved.

5 Conclusions

Although we all harbor a simple intuitive notion of time, how it is measured, how it progresses, and what it means for two events to be simultaneous, a deeper examination of the notion, both in models and in physics, reveals considerable subtleties. Cyber-physical systems pose particularly interesting challenges, because they marry a world, the cyber side, where time is largely irrelevant and is replaced by sequences and precedence relations, with a physical world, where even the classical Newtonian idealization of time stumbles on discrete, instantaneous behaviors and notions of causality and simultaneity. Since CPS entails both the smooth continuous dynamics of classical Newtonian physics, and the discrete, algorithmic dynamics of computation, it becomes impossible to ignore these subtleties.

We have shown that the approach taken in FMI (and many other modeling frameworks) that embraces a naive Newtonian physical model of time, and a cyber-

approximation of this model using floating-point numbers, is inadequate for CPS. It is suitable only for modeling continuous dynamics without discrete behaviors. Using this unfortunate choice for CPS results in models with unnecessarily inexplicable, nondeterministic, and complex behaviors. Moreover, we have shown that these problems are solvable in a very practical way, resulting in CPS models with clear semantics that are invulnerable to the pragmatics of limited-precision arithmetic in computers. To accomplish this, our solution requires an explicit choice of time resolution that quantizes time so that arithmetic on time values is performed on integers only, something that modern computers can do exactly, without quantization errors. Moreover, we have shown that such an integer model of time can be used in a practical co-simulation environment, and that this environment can even embrace components that internally use floating-point representations of Newtonian time, for example to model continuous dynamics without discrete behaviors.

We have gone to considerable effort in this paper to show that choosing a better model of time does not complicate a co-simulation framework such as FMI by much. A small number of very simple extensions to the existing standard are sufficient, and these extensions can be realized in a way that efficiently supports legacy simulation environments that use floating-point Newtonian time. But while supporting such legacy simulators, it also admits integration of a new class of simulators, including discrete-event simulators, software engineering models, hybrid systems modelers, and even the new QSS classes of simulators for continuous dynamics. Such a co-simulation framework has the potential for offering a clean and universal modeling framework for CPS. And although we have only worked out the details for FMI, we are convinced that the same principles can be applied to other co-simulation frameworks such as HLA and to simulators that directly embrace mixed discrete and continuous behaviors such as Simulink/Stateflow. We hope that our readers include the people who can make this happen.

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. Jens Bastian, Christoph Clauss, Susann Wolf, and Peter Schneider. Master for Co-Simulation Using FMI. In *8th Modelica Conference*, pages 115–120, 2011.
3. Albert Benveniste, Timothy Bourke, Benoit Caillaud, and Marc Pouzet. Non-standard semantics of hybrid systems modelers. *Journal of Computer and System Sciences*, 78:877–910, May 2012.
4. Simon Bliudze and Daniel Krob. Modelling of complex systems: Systems as dataflow machines. *Fundam. Inform.*, 91(2):251–274, 2009.
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, March 2011. Modelica Association.
6. S. Bogomolov, M. Greitschus, P. G. Jensen, K. G. Larsen, M. Mikucionis, T. Strump, and S. Tripakis. Co-Simulation of Hybrid Systems with SpaceEx and Uppaal. In *Proceedings of the 11th International Modelica Conference*. Linkoping University Electronic Press, 2015.
7. David Broman, Christopher Brooks, Lev Greenberg, Edward A. Lee, Michael Masin, Stavros Tripakis, and Michael Wetter. Determinate Composition of FMUs for Co-Simulation. In *Proceedings of the International Conference on Embedded Software (EMSOFT 2013)*. IEEE, 2013.

8. 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)*, pages 179–188. ACM, 2015.
9. David Broman, Edward A. Lee, Stavros Tripakis, and Martin Törngren. Viewpoints, formalisms, languages, and tools for cyber-physical systems. In *Proceedings of the 6th International Workshop on Multi-Paradigm Modeling*, pages 49–54. ACM, 2012.
10. David Broman and Jeremy G. Siek. Modelyze: a gradually typed host language for embedding equation-based modeling languages. Technical Report UCB/EECS-2012-173, EECS Department, University of California, Berkeley, June 2012.
11. Luca P. Carloni, Roberto Passerone, Alessandro Pinto, and Alberto Sangiovanni-Vincentelli. Languages and tools for hybrid systems design. *Foundations and Trends in Electronic Design Automation*, 1(1/2), 2006.
12. C. G. Cassandras. *Discrete Event Systems, Modeling and Performance Analysis*. Irwin, 1993.
13. Adam Cataldo, Edward A. Lee, Xiaojun Liu, Eleftherios Matsikoudis, and Haiyang Zheng. A constructive fixed-point theorem and the feedback semantics of timed systems. In *Workshop on Discrete Event Systems (WODES)*, Ann Arbor, Michigan, 2006.
14. Fabio Cremona, Marten Lohstroh, David Broman, Marco Di Natale, Edward A. Lee, and Stavros Tripakis. Step Revision in Hybrid Co-simulation with FMI. In *International Conference on Formal Methods and Models for System Design (MEMOCODE)*, 2016.
15. Fabio Cremona, Marten Lohstroh, Stavros Tripakis, Christopher Brooks, and Edward A. Lee. FIDE – An FMI Integrated Development Environment. In *Symposium on Applied Computing (SAC)*, 2016.
16. John C. Eidson. *Measurement, Control, and Communication Using IEEE 1588*. Springer, 2006.
17. Yishai A. Feldman, Lev Greenberg, and Eldad Palachi. Simulating Rhapsody SysML Blocks in Hybrid Models with FMI. In *10th Modelica Conference*, pages 43–52, 2014.
18. George S. Fishman. *Discrete-Event Simulation: Modeling, Programming, and Analysis*. Springer-Verlag, 2001.
19. Goran Frehse, Colas Le Guernic, Alexandre Donzé, Scott Cotton, Rajarshi Ray, Olivier Lebeltel, Rodolfo Ripado, Antoine Girard, Thao Dang, and Oded Maler. Spaceex: Scalable verification of hybrid systems. In *International Conference on Computer Aided Verification*, pages 379–395. Springer, 2011.
20. Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. Wiley, 2003.
21. David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, March 1991.
22. Thomas A. Henzinger. The theory of hybrid automata. In M.K. Inan and R.P. Kurshan, editors, *Verification of Digital and Hybrid Systems*, volume 170 of *NATO ASI Series F: Computer and Systems Sciences*, pages 265–292. Springer-Verlag, 2000.
23. Gabor Karsai, Andras Lang, and Sandeep Neema. Design patterns for open tool integration. *Software and Systems Modeling*, 4(2):157–170, 2005.
24. Gabor Karsai, Janos Sztipanovits, Akos Ledeczi, and Ted Bapty. Model-integrated development of embedded software. *Proceedings of the IEEE*, 91(1):145–164, 2003.
25. Ernesto Kofman and Sergio Junco. Quantized-state systems: A DEVS approach for continuous system simulation. *Transactions of The Society for Modeling and Simulation International*, 18(1):2–8, 2001.
26. P. Kopke, T. Henzinger, A. Puri, and P. Varaiya. What's decidable about hybrid automata? In *27th Annual ACM Symposioum on Theory of Computing (STOCS)*, pages 372–382, 1995.
27. Frederick Kuhl, Richard Weatherly, and Judith Dahmann. *Creating Computer Simulation Systems: an Introduction to the High Level Architecture*. Prentice Hall PTR, 1999.
28. Kim G Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1):134–152, 1997.
29. Edward A. Lee. Modeling concurrent real-time processes using discrete events. *Annals of Software Engineering*, 7:25–45, 1999.
30. Edward A. Lee. Fundamental limits of cyber-physical systems modeling. *ACM Transactions on Cyber-Physical Systems*, 1(1):26, 2016.
31. Edward A. Lee, Mehrdad Niknami, Thierry S. Nouidui, and Michael Wetter. Modeling and simulating cyber-physical systems using CyPhySim. In *International Conference on Embedded Software (EMSOFT)*, 2015.

32. Edward A. Lee and Alberto Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Circuits and Systems*, 17(12):1217–1229, 1998.
33. Edward A. Lee and Haiyang Zheng. Operational semantics of hybrid systems. In Manfred Morari and Lothar Thiele, editors, *Hybrid Systems: Computation and Control (HSCC)*, volume LNCS 3414, pages 25–53, Zurich, 2005. Springer-Verlag.
34. Oded Maler, Zohar Manna, and Amir Pnueli. From timed to hybrid systems. In *Real-Time: Theory and Practice, REX Workshop*, pages 447–484. Springer-Verlag, 1992.
35. Zohar Manna and Amir Pnueli. Verifying hybrid systems. In *Hybrid Systems*, volume LNCS 736, pages 4–35, 1993.
36. Gustavo Migoni, Mario Bortolotto, Ernesto Kofman, and François E. Cellier. Linearly implicit quantization-based integration methods for stiff ordinary differential equations. *Simulation Modelling Practice and Theory*, 35:118–136, 2013.
37. D. L Mills. A brief history of NTP time: confessions of an internet timekeeper. *ACM Computer Communications Review*, 33, 2003.
38. Modelica Association. Functional mock-up interface for model exchange and co-simulation. Report 2.0, 2014.
39. 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`.
40. Modelisar Consortium and the Modelica Association. Functional mock-up interface for model exchange and co-simulation. Report Version 2.0, July 25 2014. `https://www.fmi-standard.org/downloads`.
41. Martin Otter, Hilding Elmqvist, and José Daz López. Collision handling for the Modelica multibody library. In *Modelica Conference*, pages 45–53, 2005. Describes three approaches, impulsive, spring-damper ignoring contact area, and spring-damper including contact area.
42. Martin Otter, Martin Malmheden, Hilding Elmqvist, Sven Erik Mattsson, and Charlotta Johnsson. A new formalism for modeling of reactive and hybrid systems. In *Modelica Conference*. The Modelica Association, 2009.
43. Uwe Pohlmann, Wilhelm Schäfer, Hendrik Reddehase, Jens Röckemann, and Robert Wagner. Generating Functional Mockup Units from Software Specifications. In *9th Modelica Conference*, pages 765–774, 2012.
44. Claudius Ptolemaeus, editor. *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, Berkeley, CA, 2014.
45. Tom Schierz, Martin Arnold, and Christoph Clauss. Co-simulation with communication step size control in an FMI compatible master algorithm. In *9th Modelica Conference*, pages 205–214, 2012.
46. P. Tabuada. *Verification and Control of Hybrid Systems: A Symbolic Approach*. Springer, 2009.
47. S. Tripakis. Bridging the Semantic Gap Between Heterogeneous Modeling Formalisms and FMI. In *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation – SAMOS XV*, 2015.
48. S. Tripakis, C. Stergiou, C. Shaver, and E. A. Lee. A modular formal semantics for Ptolemy. *Mathematical Structures in Computer Science*, 23:834–881, August 2013.
49. Bernard P. Zeigler, Herbert Praehofer, and Tag Gon Kim. *Theory of Modeling and Simulation*. Academic Press, 2nd edition, 2000.
50. Yun Zhu, Edwin Westbrook, Jun Inoue, Alexandre Chapoutot, Cherif Salama, Marisa Peralta, Travis Martin, Walid Taha, Marcia O'Malley, Robert Cartwright, Aaron Ames, and Raktim Bhattacharya. Mathematical equations as executable models of mechanical systems. In *Proceedings of the 1st ACM/IEEE International Conference on Cyber-Physical Systems*, ICCPS '10, pages 1–11, New York, NY, USA, 2010. ACM.
51. Dirk Zimmer. *Equation-Based Modeling of Variable-Structure Systems*. PhD thesis, Swiss Federal Institute of Technology, Zurich, Switzerland, 2010.

A Appendix: Implementation Details

This appendix details a C implementation of the wrappers outlined in Section 3. For each category of FMU we discuss the logic required to let it successfully partake in hybrid co-simulation. Each wrapper fully implements the API shown at the left in Figure 4 in Section 3.3 and makes use of the FMI-HC extensions implemented by the FMU. It should be fairly straightforward to augment *any* master algorithm to support FMI-HC through the use of wrappers like the ones we describe in this section. Before discussing any category-specific implementation details, we first provide implementations of the functions that should be the same for wrappers of all categories.

A.1 A Template for Wrappers

Although each FMU category requires a different wrapper, there is an intersection between the functionality of all these wrappers. One could think of this intersection in terms of a "base class" in object-oriented terminology. Yet the C language has no object-oriented features, so we implement a makeshift wrapper base class using a `struct` that bundles pointers to the wrapper API functions, along with the state kept by the wrapper, and of course, a pointer to the FMU itself. The C code is given in Figure 8. This bundle serves as a template for wrappers category 0_B through 4.

Not all categories of FMUs implement the functions `getPreferredResolution` and `setResolution`. The wrapper, however, must implement both. We list an implementation of `_getPreferredResolution` in Figure 9 that is generic in the sense that it can be used for all FMU categories. In case the FMU *does* implement `getPreferredResolution` (categories 2 and 4), the wrapper invokes it and returns to the master the preferred time resolution specified by the FMU. In case `getPreferredResolution` is not implemented (categories 0_B, 1, and 3), the function simply returns a `fmi2Discard` status. We assume that the master interprets this response as if the component states no preference.

After negotiating a time resolution based on the preferences stated by the FMUs, according to the algorithm described in Figure 5 in Section 3.4, the master calls `_setResolution` on each wrapper to inform it of the time resolution that it has adopted. Note that instead of passing the actual resolution to the wrappers, the master passes the exponent n that determines the adopted time resolution as 10 to the power of n (see Equation (1) in Section 2.4).

In Figure 10 we list an implementation that can be used for all FMU categories. Importantly, the wrapper acts differently depending on the category of FMU it interacts with. A category 0_B wrapper computes and stores the time resolution in floating-point format, whereas a category 2 wrapper computes the ratio between its own time resolution and the resolution that the master has chosen. This ratio, as is further explained in Section 4, is used by the wrapper to compute the step size in the master's clock domain with respect to a step taken in the FMU's clock domain, and vice versa.

It should be noted that in Figure 10 we use two different functions to compute powers of ten. On line 10, we use function `fmi2Real realPow10(int n)`. It takes the time resolution exponent of the master (n) and returns a double-precision

```
1  typedef struct {
2      FMU*             fmu;          // pointer to the FMU
3      fmi2Real         r_master;     // floating-point representation
4                                     // of the master's time resolution
5                                     // i.e., 10^n
6      fmi2Integer      r_ratio;      // 10^(k - n)
7      fmiXIntegerTime  t_FMU;        // local time of the FMU
8  } wrapperState;
9
10 typedef struct {
11     wrapperState component;
12     fmi2Status  (*_doStep)(wrapperState*,
13                            fmiXIntegerTime,
14                            fmiXIntegerTime,
15                            fmi2Boolean,
16                            fmiXIntegerTime*);
17     fmi2Status  (*_setResolution)(wrapperState*,
18                                   fmiXTimeResolutionExponent);
19     fmi2Status  (*_getPreferredResolution)(wrapperState*,
20                                            fmiXTimeResolutionExponent*);
21     fmi2Status  (*_getMaxStepSize)(wrapperState*,
22                                    fmiXIntegerTime,
23                                    fmiXIntegerTime*);
24 } WRAPPER;
```

Fig. 8 We declare the wrapper as a `struct` containing pointers to each function in the wrapper interface and a pointer to the wrapper's state. The state is also kept in a `struct`, which keeps a pointer to the FMU and holds the following state: a floating-point representation of the master's time resolution (`r_master`, used only by the category 0_B wrapper), the ratio between the FMU's time resolution and the master's time resolution (`r_ratio`), and the current integer time of the FMU (`t_fmu`). The last two variables are only used by the category 2 wrapper. In our working implementation of `WRAPPER` struct inside the FIDE framework, we also included pointers to functions like `_rollback`, `_saveState`, `_init`, `_set` and `_get`. However, we did not include these functions here since they are not inherent to the problem of converting time.

```
1  fmi2Status _getPreferredResolution(wrapperState* wrp,
2                                     fmiXTimeResolutionExponent* n) {
3      FMU* fmu          = (*wrp).fmu;
4      fmi2Component c   = (*fmu).component;
5      fmi2Status status = fmi2Discard;
6
7      // FMU-HC2 and FMU-HC4
8      if ((*fmu).canGetPreferredResolution) {
9          status = (*fmu).getPreferredResolution(c, n);
10     }
11
12     return status;
13 }
```

Fig. 9 Our implementation of `_getPreferredResolution`. This function is the same for all categories. When the FMU does not implement the method `fmiGetPreferredResolution` (capability flag `canGetPreferredResolution == fmi2False`), the wrapper returns `fmi2Discard`. In this case, the master should ignore the value that pointer `n` points to.

floating-point number that represents the time resolution (10^n). Function `unsigned int intPow10(unsigned int n)` at line 21, on the other hand, takes an unsigned integer as input (the difference between the time resolution exponent of the FMU, k, and n) and returns the integer representing the ratio between the FMU's time resolution (10^k) and the master's resolution (10^n). Function `realPow10` returns a floating point number since n can be any integer value, positive or negative.

```
1  fmi2Status _setResolution(wrapperState* wrp,
2                            fmiXTimeResolutionExponent n) {
3      FMU* fmu              = (*wrp).fmu;
4      fmi2Component c       = (*fmu).component;
5      fmi2Status    status  = fmi2OK;
6      unsigned int  delta_n = 0;
7
8      // FMU-HC0B
9      if (!(*fmu).canHandleIntegerTime) {
10         (*wrp).r_master = realPow10(n);
11         return status;
12     }
13     // FMU-HC2
14     if ((*fmu).canGetPreferredResolution &&
15        !(*fmu).canSetResolution) {
16         fmiXTimeResolutionExponent k;
17         status = (*fmu).getPreferredResolution(c, &k);
18         // the master's resolution must always finer or equal!
19         assert(n <= k);
20         delta_n = k - n;
21         (*wrp).r_ratio = intPow10(delta_n);
22         return status;
23     }
24     // FMU-HC3 and FMU-HC4
25     if ((*fmu).canSetResolution) {
26         status = (*fmu).setResolution(c, n);
27         return status;
28     }
29     return status;
30 }
```

Fig. 10 Our implementation of `_setResolution`. This function is shared between all categories. For an FMU of category 0_B it computes and stores the master's resolution `r_master`, for an FMU of category 2 it computes and stores `r_ratio`, the ration between the FMU's time resolution 10^k and the master's resolution 10^n. For categories 3 and 4, the function simply invokes `setResolution()` on the FMU to pass on the time resolution exponent selected by the master.

It suffices for function `intPow10` to solely work with unsigned integers since the variable `delta_n = k - n` at line 21 is always positive because k is always smaller than or equal to n.

Aside from `_getPreferredResolution` and `_setResolution`, our template lacks implementations for `_get`, `_set`, `_getMaxStepSize`, and `_doStep`. The first two are trivial for category 1-4; they simply pass their arguments to `getHybrid` and `setHybrid`, respectively, and return. For the category 0_B wrapper the situation is a bit different, because this type of FMU does not support *absent*. Therefore, when the wrapper encounters *absent* in `_get` or `_set`, it substitutes the absent value by the last-known *present* value of the referenced variable. This mechanism implements a so-called zero-order hold and is consistent with the semantics of FMI 2.0, and therefore suitable for legacy FMUs. The functions `_getMaxStepSize`, and `_doStep` require logic that is specifically tailored to their category of FMU. Hence, we discuss these two functions separately for each of the different categories.

```
1  fmi2Status _getMaxStepSize(wrapperState* wrp,
2      fmiXIntegerTime currentCommunicationPoint,
3      fmiXIntegerTime* maxStepSize) {
4
5      FMU*           fmu        = (*wrp).fmu;
6      fmi2Component c          = (*fmu).component;
7      fmi2Status    status     = fmi2OK;
8      fmi2Real      resolution = (*wrp).r_master;
9      fmi2Real      h_FMU      = 0;
10     fmi2Real      t_FMU      = 0;
11
12     status = (*fmu).getMaxStepSize(c, &h_FMU);
13     (*fmu).getRealStatus(c, fmi2LastSuccessfulTime, &t_FMU);
14
15     *maxStepSize = ceil((h_FMU + t_FMU) / resolution)
16             - currentCommunicationPoint;
17     return status;
18 }
```

Fig. 11 An implementation of `_getMaxStepSize` function for category 0_B wrappers.

A.2 Wrapper Implementations

A Wrapper for Category 0_B *FMUs*

The wrapper for a category 0_B FMU is predominantly tasked with performing conversions between integer time and floating-point time, and vice versa. Refer to Sections 4.1 and 4.2, respectively, for a detailed discussion on these types of conversions.

Figure 11 shows an implementation of the function `_getMaxStepSize`. This function queries a category 0_B FMU for the maximum step size using `getMaxStepSize` (line 12), which returns a floating-point number. The conversion from FMU time (a floating-point number) to master time (an integer) is based on Equation (5), and implemented on lines 15-16 (we use here the ceiling function defined in the C standard library `math.h`). The correspondence between the variables in Equation (5) and the variables in the code is as follows: $t \leftrightarrow$ `t_FMU` , $\Delta t \leftrightarrow$ `h_FMU`, $r \leftrightarrow$ `resolution`, $i \leftrightarrow$ `currentCommunicationPoint` , and $\Delta i \leftrightarrow$ `maxStepSize`. Function `_doStep` is presented in Figure 12. Conversion from master time to local time is based on Equation (3) and is performed at line 17 and 18. If the FMU only made partial progress (the performed step size is not equal to the requested step size), converting the performed step size in the time resolution of the master, again involves time quantization. The conversion is implemented on lines 25-26 according to Equation (5).

A Wrapper for Category 1 FMUs

A category 1 FMU neither implements function `setResolution`, nor function `getPreferredResolution`. This means that the FMU is not making use of time. Hence, the functions `_setResolution` and `_getPreferredResolution` do not have to do anything. The `_doStep` function only needs to forward the call to `doStepHybrid`, but the actual communication time can be arbitrary because the FMU of category 1 does not consider time. Finally, `_getMaxStepSize` should return that it accepts any step size.

```
1  fmi2Status _doStep(wrapperState* wrp,
2      fmiXIntegerTime currentCommunicationPoint,
3      fmiXIntegerTime communicationStepSize,
4      fmi2Boolean noSetFMUStatePriorToCurrentPoint,
5      fmiXIntegerTime* performedStepSize) {
6
7      FMU*           fmu         = (*wrp).fmu;
8      fmi2Component  c           = (*fmu).component;
9      fmi2Status     status      = fmi2OK;
10     fmi2Real       resolution  = (*wrp).r_master;
11     fmi2Real       h_FMU       = 0;
12     fmi2Real       t_FMU       = 0;
13     fmi2Real       new_t_FMU   = 0;
14
15     (*fmu).getRealStatus(c, fmi2LastSuccessfulTime, &t_FMU);
16
17     h_FMU = (currentCommunicationPoint + communicationStepSize)
18         * resolution - t_FMU;
19
20     status = (*fmu).doStep(c, t_FMU, h_FMU,
21                 noSetFMUStatePriorToCurrentPoint);
22
23     (*fmu).getRealStatus(c, fmi2LastSuccessfulTime, &new_t_FMU);
24
25     *performedStepSize = ceil((new_t_FMU) / resolution)
26                             - currentCommunicationPoint;
27
28     // Overwrite status in case of partial progress
29     if (*performedStepSize == communicationStepSize)
30         status = fmi2OK;
31
32     return status;
33 }
```

Fig. 12 An implementation of `_doStep` function for category 0_B wrapper.

A Wrapper for Category 2 FMUs

A category 2 FMU implements `getPreferredResolution`, but not `setResolution`. The FMU therefore does not only prefer, but insists on using the resolution returned by `getPreferredResolution`. This means that the category 2 wrapper needs to convert between the resolution that the master algorithm decides to use and the resolution that the FMU insists on using, and vice versa. These types of conversions are discussed in depth in Section 4.3.

The first step in the implementation of function `_doStep` (Figure 13) is to compute the local step size (Δj) according to Equation (7). The next step in function `_doStep` is to call the FMU function `doStepHybrid` using the local step size `h_FMU`. If the FMU accepts the step, the wrapper returns that the performed step size `h_FMU_accepted` is equal to the requested step size `communicationStepSize`. However, if the local progress `h_FMU_accepted` is less than the local step size `communicationStepSize`, then the FMU made partial progress. In such a case, we compute the performed step size according to Equation (8).

In `_getMaxStepSize` (Figure 14), we need to compute the maximum step size of the FMU. The first step is to call `getMaxStepSizeHybrid`, which returns the local maximal step size (Δj). The returned value is an integer that encodes a multiple of the FMU's resolution. To convert the step into a multiple of the master's resolution (Δi), we use again Equation (8). Function `_getMaxStepSize` returns Δi. It should

```
1  fmi2Status _doStep(wrapperState* wrp,
2      fmiXIntegerTime currentCommunicationPoint,
3      fmiXIntegerTime communicationStepSize,
4      fmi2Boolean noSetFMUStatePriorToCurrentPoint,
5      fmiXIntegerTime* performedStepSize) {
6
7      FMU*             fmu             = (*wrp).fmu;
8      fmi2Component    c               = (*fmu).component;
9      fmi2Status       status          = fmi2OK;
10     fmi2Integer      r_ratio         = (*wrp).r_ratio;
11     fmiXIntegerTime  t_FMU           = (*wrp).t_FMU;
12     fmiXIntegerTime  h_FMU           = 0;
13     fmiXIntegerTime  h_FMU_accepted  = 0;
14
15     h_FMU = (currentCommunicationPoint
16             + communicationStepSize) / r_ratio - t_FMU;
17
18     status = (*fmu).doStepHybrid(c, t_FMU, h_FMU,
19                 noSetFMUStatePriorToCurrentPoint, &h_FMU_accepted);
20
21     if ((t_FMU + h_FMU_accepted) * r_ratio < currentCommunicationPoint)
22         *performedStepSize = 0;
23     else
24         *performedStepSize = (t_FMU + h_FMU_accepted) * r_ratio
25                 - currentCommunicationPoint;
26
27     (*wrp).t_FMU = t_FMU + h_FMU_accepted;
28     return status;
29 }
```

Fig. 13 An implementation of `_doStep` function for category 2 wrapper.

be noted that the ratio between the resolutions of the FMU and the master as used in Equations (8) and (7) is precomputed during initialization in `_setResolution` (see Figure 10) and stored in `wrp->r_ratio`.

There is one additional subtlety that this wrapper accounts for. When the global time advances with time steps that are smaller than the time resolution of the FMU, the FMU time does not advance (as can be deduced from Equation 7) causing the FMU to lag behind the master algorithm. When the FMU indeed lags behind, and the master instructs the wrapper to take a zero-size step, however, Equation 8 would yield a negative value for Δi (a negative time index), which is clearly wrong. Therefore, the wrapper returns 0 in this particular situation (Figure 13 line 22 and Figure 14 line 15).

Wrappers for Category 3 and Category 4 FMUs

Wrappers for category 3 and category 4 FMUs are straightforward to implement. Both category 3 and 4 FMUs implement `setResolution`, which means that the FMUs must accept the resolution that the master algorithm states. The only difference between these two categories is that category 4 FMUs also implement `getPreferredResolution`, whereas category 3 FMUs do not. All the wrapper functions are simply transferring the call to the corresponding FMI function. No translation of resolutions are necessary because the FMUs promise to handle the resolution that is set by the master.

```
1  fmi2Status _getMaxStepSize(wrapperState* wrp,
2      fmiXIntegerTime currentCommunicationPoint,
3      fmiXIntegerTime* maxStepSize) {
4
5      FMU*            fmu     = (*wrp).fmu;
6      fmi2Component   c       = (*fmu).component;
7      fmi2Status      status  = fmi2OK;
8      fmi2Integer     r_ratio = (*wrp).r_ratio;
9      fmiXIntegerTime t_FMU   = (*wrp).t_FMU;
10     fmiXIntegerTime h_FMU   = 0;
11
12     status = (*fmu).getMaxStepSizeHybrid(c, &h_FMU);
13
14     if ((h_FMU + t_FMU) * r_ratio < currentCommunicationPoint)
15         *maxStepSize = 0;
16     else
17         *maxStepSize = (h_FMU + t_FMU) * r_ratio -
18             currentCommunicationPoint;
19
20     return status;
21 }
```

Fig. 14 An implementation of `_getMaxStepSize` function for category 2 wrappers.