Step Revision in Hybrid Co-simulation with FMI

Fabio Cremona*‡¶, Marten Lohstroh*, David Broman†, Marco Di Natale‡,
Edward A. Lee*, and Stavros Tripakis*§
{f.cremona,marten,eal,stavros}@eecs.berkeley.edu, dbro@kth.se, marco@sssup.it
*University of California Berkeley, USA †KTH Royal Institute of Technology, Sweden
‡Scuola Superiore Sant'Anna, Italy §Aalto University, Finland ¶ALES — United Technologies Research Center, Italy

***Abstract*—This paper presents a master algorithm for co-simulation of hybrid systems using the Functional Mock-up Interface (FMI) standard. Our algorithm introduces step revision to achieve an accurate and precise handling of mixtures of continuous-time and discrete-event signals, particularly in the situation where components are unable to accurately extrapolate their input. Step revision provides an efficient means to respect the error bounds of numerical approximation algorithms that operate inside co-simulated FMUs. We first explain the most fundamental issues associated with hybrid co-simulation and analyze them in the framework of FMI. We demonstrate the necessity for step revision to address some of these issues and formally describe a master algorithm that supports it. Finally, we present experimental results obtained through our reference implementation that is part of our publicly available open-source toolchain called FIDE.**

I. Introduction

Cyber-physical systems (CPS) comprise many components, including physical parts (e.g., mechanical systems, electrical systems, radio systems) and cyber parts (software and communication networks). Each of these components is best described using specialized modeling formalisms, e.g., state machines for discrete components such as software, differential equations for continuous components (physical plants), and discrete-event systems for networks. Modeling and simulation are non-trivial problems that have received a lot of attention and have by now resulted in languages and tools highly specialized in each of the above domains. For instance, UML and SysML are standard modeling languages for software systems, including notations to specify hierarchical state machines; Simulink is a de facto standard for signal processing and control applications; and Modelica is an equation-based language that is particularly strong for continuous systems.

Co-simulation technology seeks to leverage existing modeling and simulation tools and the artifacts created with them, in order to provide users with the capability to simulate an entire system. Instead of using a single simulation language and tool for the entire system, the idea is to model separate parts or aspects of the system separately, using specialized tools, and then connect and co-simulate these models by means of an integration and coordination environment.

The **Functional Mock-up Interface** (FMI) is an international standard, one version of which supports co-simulation [1]. FMI defines a standard API (application programming interface) that enables simulators to work together. In co-simulation, a component called an FMU (**Functional Mock-up Unit**) is a "black box" with input ports, output ports, and a set of state variables. It implements functions in the API, including `init` (initialize the state variables), `set` (set the value of an input port), `get` (retrieve the value of an output port), and `doStep`(h) (advance the state and local time of the FMU by h time units). Internally, an FMU can implement a simulator that is quite distinct from the simulators implemented by other FMUs.

The FMI standard only specifies the API that FMUs must implement, and does not specify a **master algorithm** (MA), which is the algorithm that orchestrates the co-simulation by calling the API functions on interconnected FMUs. Devising an MA with good properties is not trivial, and has been the focus of earlier work [2]. That work proposes an MA that ensures determinism and consensus on the size of each time step. An implementation of the algorithm is described in [3].

A key feature of the MA of [2] is **step determination**, which entails finding the largest step that is acceptable by *all* FMUs before taking a step. This procedure ensures that maximal progress is made and all FMUs move in lockstep. After a simulation step has been completed, however, there might still be reason to conclude that the step taken was too large. For instance, consider the scenario where one of the FMUs implements a **zero-crossing detector** (ZCD). A ZCD is a component that conceptually takes as input a continuous-time, real-valued signal x and outputs a discrete event every time x crosses 0. Now, suppose $x(t) = -1$ and the master advances time to $t + h$. The ZCD does not have enough information to determine whether a zero crossing has occurred in the interval between t and $t + h$. To know whether a zero crossing has occurred, it will also need to know the value of the input x at time $t + h$. This value is not provided in the FMI API until *after* the step has been accepted and committed by all FMUs. Since the ZCD does not know whether a zero crossing has occurred, the only thing it can do to find out is to accept the step size h.

After the step has been committed, suppose that the input to the ZCD is $x(t + h) = 1$, which means that x crossed 0 at least once between t and $t + h$. Should the ZCD generate an output event at the current time $t + h$? This depends on the error parameters with which the ZCD is configured. It may be the case that generating an output event at time $t + h$ is unacceptable, if the uncertainty interval $[t, t+h]$ is considered too large. In that case, the only solution is to backtrack the entire simulation model, i.e., all FMUs, to time t and try again

with a smaller step. We call this procedure **step revision**. We identify it as an essential feature for FMI co-simulation because it provides an efficient means to respect the error bounds of numerical approximation algorithms that operate inside co-simulated FMUs. Hence, the main contribution of this paper is our enhancement of the MA from [2] by adding step revision.

The remainder of this paper is organized as follows. In Section II we discuss continuous-time and discrete-event simulation and how state of the art tools like Simulink and Ptolemy II combine them. In Section III we discuss step revision, and explain how exactly it is different from step determination. Section IV summarizes the extensions to the current FMI co-simulation standard that we rely on in order to offer support for **hybrid co-simulation** (i.e., co-simulation that integrates continuous-time and discrete-event semantics). In Section V we present our MA with step revision capability. We show an example that leverages this new capability in Section VI. Finally, in Section VII we deliver our conclusions and discuss possible avenues to explore as future work.

II. Simulation of Hybrid Systems

A. Principles of Numerical Integration

Numerical Integration is the area of numerical analysis that studies algorithms to calculate the value of a definite integral:

$$x(t) = x_0 + \int_{t_0}^{t} \dot{x}(\tau) d\tau$$

The derivative $\dot{x}$ is the integrated function, x_0 the initial state (initial conditions), and t_0 is the time instant from which the integral is solved. In the case of an **ordinary differential equation** (ODE), $\dot{x}(t)$ at any time t is a function of the current state $x(t)$, the current input $u(t)$, and t,

$$\dot{x}(t) = f(x(t), u(t), t).$$

All of these quantities may have vector values. If $f(x(t), u(t), t)$ is analytic, then the value of $x(t+h)$ at some future time $t+h$ is given by the Taylor Series:

$$x(t+h) = x(t) + h\dot{x}(t) + \frac{h^2}{2!}\ddot{x}(t) + \dots$$

which typically converges fast to $x(t+h)$ as the terms in the series become small after a few terms. The Taylor Series is at the base of many integration algorithms. These algorithms, also known as **ODE solvers**, rely on the truncation of the Taylor Series to estimate the new value $x(t+h)$ in finite time. To increase performance without sacrificing precision, **variable step size** techniques are used. These methods allow the user to specify an acceptable error rate and adjust the step size h so that each step introduces error at no more than that rate. The higher-order terms may be used to estimate the error. When the error due to the truncation is not acceptable, a smaller step size $h' < h$ is adopted. This approach allows larger steps to be taken when the error is small relative to current step size, which can yield a significant reduction in computational effort. For an overview of these techniques, see [4].

A commonly used category of ODE solvers is the **Runge-Kutta** (R-K) methods. An *explicit* R-K solver does not only use a variable step size, it also requires the evaluation of $f(x(t), u(t), t)$ at multiple time instants between t and $t+h$ to improve accuracy. A third-order Runga-Kutta, for example, uses the inputs at t, $t + 0.5h$ and $t + 0.75h$. The **local truncation error** (incurred by one integration step) is of the order $\mathcal{O}(h^{k+1})$ where k indicates the order of the R-K method. To know the inputs at each stage, the solver typically needs to evaluate the outputs of other components in the model at the corresponding time instants. In practice, in a computer-based model, such evaluation involves the execution of software functions that return, each for a particular component in the model, the value of the output signals of that component based on its internal state and provided inputs.

In modern modeling and simulation languages like Simulink [1] (The MathWorks) and Ptolemy II [5] (UC Berkeley), the output evaluation function is free of side effects, meaning that its invocation does not alter the internal state of the component. These languages indeed have separate functions for the evaluation of the output variables and the commitment to a new state. In FMI, on the other hand, both the evaluation of an FMU's outputs and the computation of its new state are performed inside the same API function called `doStep`. The input values at time t are provided to the FMU before `doStep` is invoked, and during the integration step from t to $t+h$, an R-K solver can only estimate later input values using extrapolation. Similarly, during `doStep`, a ZCD does not have access to the input value at time $t+h$, and hence can only guess whether a zero crossing occurs during the interval by extrapolating its input.

B. Models of Computation

The chief challenge in co-simulation is the orchestration of interactions between multiple concurrently operating components, each of which is driven by its own simulator. The specific coordination rules that a simulation follows, which make components synchronize a certain way or exchange inputs and outputs between components in a certain order, determine its results. In other words, these rules assign a **semantics** to the concurrent execution of the components that partake in a simulation — they constitute what is commonly referred to as a **Model of Computation** (MoC). In Ptolemy II, concurrent components are referred to as **actors** and a simulation is coordinated by a **director**, which implements a particular MoC [5]. In Simulink, the components are called **blocks** and the simulation engine behaves much like the Ptolemy II Continuous director. In FMI, the components are called FMUs and the simulation is orchestrated by a master algorithm (MA).

Hybrid systems — particularly common in the modeling and simulation of CPS — require the integration of two

[1] www.mathworks.com/simulink

different MoCs as they exhibit both discrete behaviors (e.g., a controller based on an **Finite State Machine** (FSM)) and continuous semantics (e.g., a model of a physical plant described by an ODE). The discrete transitions of an FSM may yield discrete events or cause discontinuities. A **discrete event** is present only at a precise time instant and is absent otherwise. A **discontinuity** is an instantaneous change in a continuous signal.

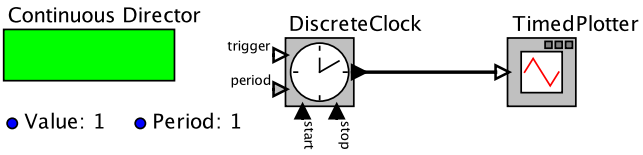
C. Continuous-Time Simulation in Ptolemy II, Simulink, FMI

Tools that simulate continuous-time systems typically provide different ODE solvers because one solver may outperform another depending on the particular simulation dynamics at hand. Ptolemy II, for example, provides a third-order and a fourth-order R-K explicit solvers with variable step size. Simulink has a greater variety of ODE solvers, and also features the third- and fourth-order R-K.

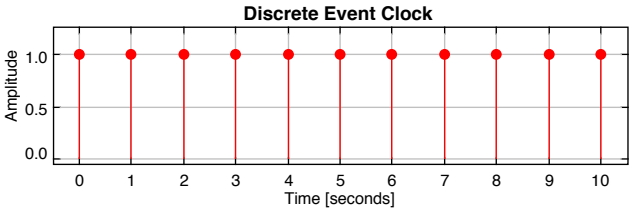
A simulation run in Ptolemy II is split between an *initialization*, *execution*, and *wrapup* phase. In the execution phase, the director is able to call two functions on each actor: `fire` and `postfire`. In `fire` the output of a component is computed. It is required that the implementation of `fire` is free of side effects. The state of a component may only be updated with the invocation of `postfire`.

The Continuous director estimates the simulation step size according to its solver (e.g., to keep the local truncation error below a predefined tolerance) as well as possible discrete events reported by actors through the callback `fireAt`. The process of refining the step size can require multiple firings of the model with different values of the step size. Only when a sufficient accuracy is reached, the director invokes `postfire` that finally commits the state. This two-phase execution scheme guarantees that no revision of the step size is needed.

The Simulink S-function (i.e., a simulation block written in MATLAB, C, C++, or Fortran) is executed in a split-phase fashion, similar to how actors are executed in Ptolemy II. Simulink further divides a simulation step into a *major step* and a *minor step*. In a major step, the simulation engine executes the functions `mdlOutputs` (output update) and `mdlUpdate` (state update) exactly once per simulation step. A periodic Simulink block for example, is executed at the major-step. As opposed to the simulation engine, the integration process executes in minor steps. In each minor step, the solver first computes the outputs with `mdlOutputs` and then updates the state calling `mdlDerivatives` if the step size is accepted. The evaluation of `mdlOutputs` and `mdlDerivatives` in minor steps can occur multiple times depending on the solver. At every execution of the minor step (called *integration stage*), the state is updated and the simulation time moved forward. The size of the minor steps depend on the solver, like in Ptolemy II. At each minor step, the Simulink engine performs zero-crossing detection through the functions `mdlOutput` and `mdlZeroCrossing`. For the detection of a zero crossing, Simulink first computes the outputs by invoking *mdlOutputs* and then checks whether



(a) Discrete event generator in Ptolemy II.



(b) Output signal of the discrete clock in Figure 1a.

Fig. 1: Example of timed events.

a zero crossing has occurred by invoking *mdlZeroCrossing*. Once detected, the solver uses a bisection search to determine the point in time when the zero crossing occurred.

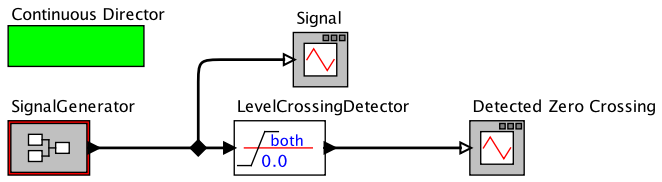
The FMI standard does not provide a two-phase API similar to ones of Ptolemy II or Simulink. Hence, `fmi2DoStep` is not free of side effects. The inputs to the FMU are supplied before each integration step. In this way, a third-order R-K algorithm embedded into an FMU for example, can only use **extrapolation** to estimate the inputs at t, $t+0.5h$ and $t+0.75h$. Extrapolation (except in the special case of some quantized state systems [6]) yields an *approximation* to the values of inputs past t, and the extrapolation error will factor into the integration error. Another disadvantage of R-K algorithms using extrapolation is that they are extremely inefficient. The IEX4 algorithm described in [4] for example, is a 10-stage algorithm while a classical R-K of the same order can be constructed with only two stages [4].

An alternative to input extrapolation would be to accept time advancement to $t + h$ in order to evaluate the inputs at that time, and then perform the integration using a classical R-K algorithm. Note that given t *and* h, input values at intermediate time instants needed for the R-K algorithm can be obtained now through **interpolation**. If the error of the integration performed at $t+h$ happens to be too large, then the committed state of *all* FMUs in the co-simulation must revert back to t. In other words, this approach would require step revision.

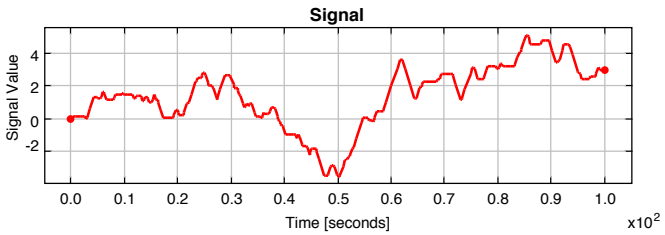
D. Combining Continuous Time with Discrete Events

When talking about discrete events, it is useful to distinguish between two different classes, namely, **state events** and **time events** [4]. The main difference between the two is that time events can be predicted, meaning that the time at which they occur is known in advance. State events are not known in advance and require evaluation of the state in order to be detected.

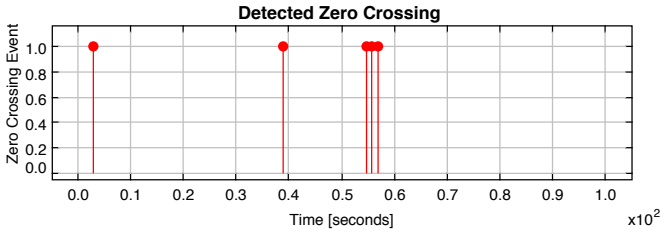
Consider for example the Ptolemy model in Figure 1a of a periodic discrete event signal generator and its output in



(a) Ptolemy II model with a ZCD.



(b) Output of the random signal generator in Figure 2a.



(c) Zero crossings detected in the signal in Figure 2b.

Fig. 2: Example of state events.

Figure 1b. For this model, it is known *a priori* at what time the discrete clock will generate discrete events. From the beginning of the simulation at time $t = 0$, an event is generated every second. These events thus are time events. Behind the scenes, the discrete clock provides a hint to the Continuous director as to when the next event will occur by requesting the director to fire it at a particular time. If the same model is implemented in FMI, the execution is slightly different. At any time t, the master may propose to the discrete clock FMU a step size h. Suppose an FMU wants to generate an event at $t+h'$ and $h' < h$, then the FMU will only advance its time to $t+h'$. The master will notice partial progress on behalf of the FMU and rolls back any other FMUs that ended up accepting step size h. After having moved all FMUs to $t+h'$ (assuming there are no earlier events to process), the simulation state is updated, and the event is generated.

An example of a state event is the determination of the level crossing of a continuous-time signal, implemented by the model in Figure 2. The input of the level crossing detector is continuous, its output discrete. Hence, this is a hybrid system. In this case the level is 0, so we refer to it as ZCD. A zero-crossing function $g(x(t), x(t+h))$, often called a **guard function**, determines when the level crossing occurs [7] by evaluating to *true* when $sign(x(t)) \neq sign(x(t+h))$. When the function $g(x(t), x(t+h))$ evaluates to true, a discrete event is generated at $t+h$. An example developed in Ptolemy II is shown in Figure 2a, the output signals are depicted in Figures 2b and 2c.

The finite precision of arithmetic in digital computers imposes an upper bound on the practically feasible accuracy of a ZCD. In a continuous-time simulation the situation is less than ideal. Since the simulation progresses stepwise, the mere detection of a zero crossing does not actually reveal where in time the zero crossing exactly took place. All that the detector indicates is that a crossing happened some time between t and $t+h$. Suppose that the zero crossing actually occurs at $t+h'$ where $h' < h$. Then depending on the difference between h' and h, the crossing might not be detected with the required precision (i.e., it is detected too late). In such case, the step size has to be revised in order to detect the zero crossing with the required precision. Importantly, the occurrence of a zero crossing and the error of its detection can only be evaluated *after* the input to the ZCD at $t+h$ is known. This means that the components upstream of the ZCD need to be evaluated at $t+h$ first, and might then have to revert to $t+h'$ after having already accepted the step to $t+h$.

Techniques to revise the step size in order to more accurately identify the time at which a zero crossing occurred are usually based on **bisection search** algorithms described by Cellier [4] and Zhang [7]. At each iteration of a bisection search, the output update functions of all components must be evaluated. Therefore, much like the ODE solvers we discuss in Section II, step size revision may require components to recompute their output multiple times per simulation step. In Ptolemy II, a component can be evaluated using the `fire` function without affecting its state; multiple calls to fire can follow each other up before, finally, `postfire` is called to commit the component to a new state. Hence, a model like the one in Figure 2 does not pose a problem in Ptolemy II. However, if the evaluation of signals is *not* free of side effects (like in the case of FMI), step size revision involves restoring each component to its last know valid state and then re-evaluating all outputs at the time of the event.

In previous work [3], we presented FIDE, an FMI Integrated Development Environment. FIDE uses Ptolemy II and its graphical user interface to allow a user to import FMUs, arrange them in a model, and generate a simulator in C code based on a template that implements an MA. The resulting code can then be compiled and executed outside Ptolemy II. FIDE implements the algorithm specified by Broman et al. in [2].

Figure 3 shows a model, also a hybrid system, from [2]. In this figure, the integrator integrates a constant signal with value 1, which feeds into an adder. The resulting ramp signal feeds into a ZCD which generates a discrete event as soon as its input exceeds the value $u1 = 0$. The event resets the integrator. The resulting signal at the output of the integrator is the sawtooth wave in Figure 3b.

Importantly, and not shown in the figure, the ZCD in this model is supplied with a derivative of its input signal, based on which it predicts the approximate time at which the next zero crossing will occur. The ZCD performs input extrapolation to

avoid stepping too far past the zero crossing. In this particular scenario, there is no need for multiple iterations of evaluating the output functions and adjusting the step size in order to converge to a step size that allows the crossing to be detected within given error bounds. Given the nature of the signal, that it is piecewise affine, its first-order derivative suffices to predict when it crosses zero. For this to work, however, the following requirements must be satisfied:

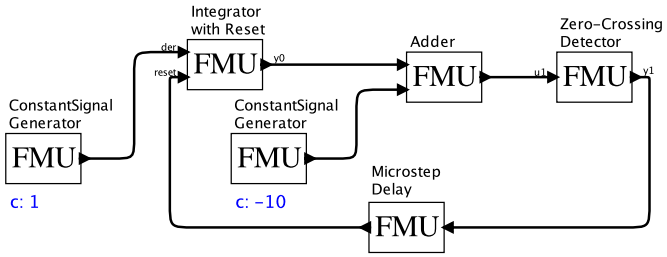
i The MA must be able to propagate derivatives provided by the FMUs;
ii Each FMU in the path from a signal source to the component that wants to extrapolate its input signal must be able to propagate the derivatives.

Only then will the extrapolation strategy work. In [2] it is assumed that components that require knowledge about future inputs in order to compute their current output (e.g., implicit solvers) would use extrapolation to approximate their future inputs. The authors, however do not provide alternatives for the scenario where extrapolation cannot be performed.

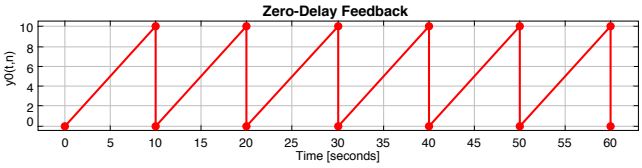
In a model-based design chain [8], and in the spirit of FMI, FMUs may be designed by different teams, unaware of the particular context in which a component will be used. In our example, the propagation of the derivatives from the integrator up to the ZCD requires a particular design of `Adder` so that it accepts an input derivative and also performs the "add" operation on the value of that derivative before propagating it. The integrator must also propagate the derivative. Support for derivatives is *optional* in FMI; no FMU can be designed to rely on other components implementing it without breaking the modularity that FMI is precisely intended to provide.

Now suppose that the ZCD in our example cannot use derivatives to predict its step size, perhaps because its upstream adder does not have the optional capability to propagate a derivative. Instead, we let the ZCD report a fixed maximum step size. It will then take steps of *at most* that size, yet the master may decide to take smaller steps, which will happen in case other FMUs report a smaller maximum step size. If the maximum step size of the ZCD is chosen very small the simulation will become inefficient, but if it is chosen too large the detector may overshoot zero crossings due to undersampling. In order to strike a sensible balance between accuracy and performance, one has to take into consideration the dynamics of the system, which are often not so predictable (if they were, the system would not be a very interesting one to model).

Clearly, using a fixed maximum step size for an FMU like our ZCD is problematic, and reliance on the availability of derivatives stands in the way of composability with general-use components. As an alternative, we propose an MA that supports step size revision triggered by state events. The ability to revise the step size allows a ZCD to solely rely on the simple guard function $g(x(t), x(t+h))$ which does not require derivatives. Such a ZCD is more generally composable as it can work on inputs from components that, like probably most FMUs, do not supply derivatives.



(a) Model based on the example from [3].



(b) Output of the integrator, using an extrapolating ZCD.

Fig. 3: Hybrid System with zero-delay feedback.

E. Related Work

The topic of hybrid co-simulation has seen several contributions and on-going research efforts in the recent years. Apart from the work already mentioned above, we would like to highlight the following projects.

DACCOSIM [9] is a co-simulator developed by the RISEGrid institute that can co-simulate FMUs in parallel on different nodes in a cluster. Although DACCOSIM claims to handle mixtures of continuous and discrete signals, its implementation offers merely an approximation of event handling. Following the definition in [10], discrete events are instantaneous. Instead, in DACCOSIM, events are present over the duration of a time *interval*, not at a precise time *instant*.

Another approach to co-simulation of hybrid systems with FMI is presented by Denil et al. [11]. The authors describe a semantic adaptation mechanism that uses state event detection to extract events from continuous signals in causal block diagrams and interfaces them with Statecharts though FMI. The pre- and post-conditions of events are specified in a "semantic adaptation model" and a wrapper that implements a corresponding state event detector FMU is then generated. The resulting style of event handling is similar to DACCOSIM.

Camus et al. [12] present a Discrete EVent System Specification (DEVS) [13] wrapper for hybrid co-simulation with FMI. Their approach is, instead of augmenting FMI with the capability to handle discrete events, to leverage an execution engine based on the DEV&DESS hybrid system formalism (which interfaces DEVS with the Differential Equation System Specification) and co-simulate FMUs with other components. The authors report only having successfully co-simulated a single FMU, which explains why they were able to do so mostly within the operational constraints of FMI 2.0. However, Camus et al., too, rely on the FMU's capability to perform rollback in order to conduct bi-sectional searches for the localization of state events.

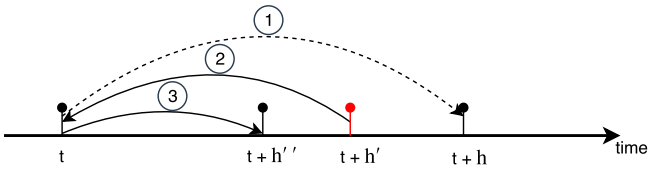


Fig. 4: Rollback during step determination.

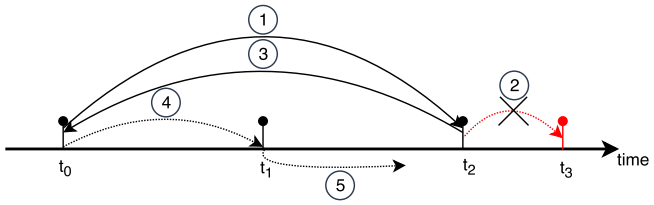


Fig. 5: The procedure of step revision.

III. Step Determination & Step Refinement

The ZCD in our example can only detect the sign switch and evaluate the error only *after* having committed to a step size and having received a new input. When it overshoots and exceeds the error margin, the correct course of action would be to **undo** the previous step and try again with a smaller step size. The MA in [2] has no provisions for such action. It does, however, employ a similar but different mechanism in its step determination routine. Specifically, rollback is used in the presence of one or more FMUs that do not support `fmi2GetMaxStepSize` and need to be stepped in order to find out whether they accept some proposed step size.

Step determination can be summarized as follows. The algorithm supposes that some (or none), but not all FMUs can report their maximum step size. First it interrogates those that can report their maximum step size and finds the minimum of the reported values, call it h. Then it speculatively lets those that cannot report a maximum step take a step of h (see ① in Figure 4) to find out what their maximum step size is. Specifically, an FMU can *accept* or *reject* a proposed time step h by returning different codes when `doStep` is called. The next simulation step is determined by taking the minimum of all obtained maximum step sizes. All FMUs that are speculatively executed during step determination must be rolled back to their previous state (transition ②) before all FMUs can be moved forward by the determined step (transition ③).

Step revision, on the other hand, involves moving the entire simulation back in time before moving it forward again. This procedure is necessary in FMI in order to support co-simulated FMUs that implement numerical approximation algorithms that suffer an error that is dependent on the simulation step size which they can only estimate after the step has already been taken. Both the R-K solver and the ZCD we discussed belong to this class of components. In order to keep the error of such FMU within limits, it must in principle be possible to revise each last-taken step in case an FMU exceeds those limits. An FMU can indicate its need for step revision simply by returning an appropriate error flag in `doStep` or in `getMaxStepSize`.

The procedure of step revision is illustrated by the sequence in Figure 5 and can be broken down as follows:

① save the state of all FMUs at the current time, say, t_0, determine h, and attempt to advance time to $t_2 = t_0 + h$ if all FMUs can move forward, by calling `doStep`(h) on all FMUs;

② if any FMU ended up in an invalid state (`getMaxStepSize` or `doStep` returned an error flag [1]), then proceed to ③, otherwise, repeat ①, now with t_2 being the current time (in this case, no rollback is performed).

③ restore all the FMUs' state to time t_0 and try with a smaller step size s.t. $t_1 = t_0 + h' < t_2$;

④ advance the simulation time to t_1 by calling `doStep`(h') on all FMUs;

⑤ if all FMUs arrive in a valid state, then go to ① and repeat the above sequence, now with t_1 being the current time, otherwise, repeat ③ with yet a smaller h'.

Step revision is complementary to step determination. In fact, our MA presented in Section V features both. After a step has been determined that is acceptable by all FMUs, some may still end up in an invalid state, demanding the step be revised. Both are techniques that involve speculative execution and rollback, but with step revision the speculation reaches further, namely, until *after* the speculative step has concluded. Also notice that step determination is guaranteed to succeed in one iteration, whereas step revision may require several iterations. Step revision, however, is not recursive; it will only go back and forth between $t+h$ and t, each time with a smaller h until the simulation can successfully move past $t + h$.

A. Support for Rollback in FMI

FMI 2.0 specifies the following functions to save and restore the internal state of an FMU [14], respectively:

- `fmi2GetFMUstate` saves the internal state of the FMU. It takes as a input a pointer to a FMU state, and copies the current FMU state into it.
- `fmi2SetFMUstate` restores the internal state of the FMU. It takes as a input a pointer to a previous copy of the FMU state, and substitutes the current state of the FMU with the copy.

Their implementation, however, is *optional*.

The rollback mechanism used in [2], [3], and leveraged in this paper to support step revision, relies on the capability of an FMU to save its state and restore it in whenever necessary. It is therefore our suggestion to make the implementation of these methods mandatory, as they are straightforward to implement and enable support for an important class of components that use state events to control the simulation step size and, if

needed, request step size revision. In our opinion it is more reasonable to rely on rollback abilities than it is to rely on extrapolation techniques that use derivatives and hence require each FMU to be able to work with derivatives, even if they do not use them. As such, the MA we present in Section V requires all participating FMUs to implement the optional functions `fmi2GetFMUstate` and `fmi2SetFMUstate`[2].

IV. Extensions to the FMI Standard

The FMI standard — currently at version 2.0 [1] — describes two distinct techniques for interactions between an FMU and a host simulator: i) model exchange (**FMI-ME**), where the host simulator carries out any numerical integration required by FMUs, and ii) co-simulation (**FMI-CS**), where the FMU implements its own mechanisms for advancing the values of its state variables. In model exchange, the simulator has access to the differential equations that describe the dynamics of the FMU, while in co-simulation, FMUs are treated as a "black box," meaning that the master only has access to the FMU's inputs and outputs. FMI-ME and FMI-CS, although their interfaces have a lot in common, are developed independently as they serve different purposes, and their evolution is driven by different stakeholders. FMI-CS 2.0 is optimized for simulating continuous dynamics and lacks provisions for discrete-event semantics or handling discontinuities. Hence, aside from the mandatory implementation of functions `fmi2GetFMUstate` and `fmi2SetFMUstate` argued for in Section III, our MA relies on the extensions to FMI-CS that were proposed in [2] and [3], which add support for hybrid co-simulation. The first two extensions we summarize here are necessary in order to simulate some very common classes of models, such as hybrid systems and feedback control systems, the third is a performance optimization that allows to reduce execution time. All three extensions are leveraged by the MA we present in Section V and are key to our capability to run the example given in Section VI. Hence, we succinctly explain them here so as to provide a self-contained description of our algorithm.

A. Hybrid Co-simulation

To support discrete event signals, i) an FMU should be able to indicate its output or detect its input to be "absent," and ii) it must support zero-duration steps [10]. The second feature is also necessary in order to support discontinuities. The following two aspects of FMI-CS 2.0 are incompatible with these requirements:

1) The step size proposed to an FMU with `fmi2DoStep` must be strictly greater than zero;
2) The type system of FMI does not support *absent*. Signals are defined in a time continuum, they are never absent.

The standard can be easily extended to support absent signals by extending the set of values associated with the original FMI data types to include an extra value to encode absent, e.g, for the real numbers: $\mathbb{R} \cup \{\epsilon\}$. Of course, the machine representation of FMI data types are ordinary 32-bit integers and 64-bit doubles that do not have extra bits to encode "special values." The solution, as we proposed in [3], is to augment the FMI functions for getting and setting values to indicate whether a signal is present or absent.

Zero-duration events or discontinuities can be achieved in FMI by extending the standard with the superdense model of time [15], [16]. In **superdense time**, time is represented as a tuple $\tau = (t, n)$ with $t \in \mathbb{R}_+$ and $n \in \mathbb{N}_0$. $t \in \mathbb{R}_+$ represents time in the Newtonian sense, while $n \in \mathbb{N}$ represents the **microstep**, i.e., the index of the current iteration at the same Newtonian time t. As proposed in [2], superdense time can be supported by simply enabling the FMU to accept a step size $h \geq 0$ and interpreting inputs that arrive at a superdense time index greater than zero as discrete changes. A discontinuity is characterized by a present value at $(t, 0)$ and a present value at $(t, 1)$, while a discrete event is absent at (t, n), present at $(t, n + 1)$, and absent at $(t, n + 2)$.

B. Feedback Loops

Algebraic loops can occur when an FMU with direct feedthrough is placed in a feedback loop. An FMU is called **strict** if it has one or more outputs that depend on one or more of its inputs at the *same* time instant. When a zero-delay path is created between such output and one of the inputs that it depends on, **direct feedthrough** will occur, and the assumed causal relationship between the FMU's input and output is broken. Not all loops are algebraic loops though. Notice that the possibility of direct feedthrough depends on the particular dependencies that exist between the input and output variables within the FMU. A loop that contains strict components may be falsely identified as an algebraic loop in case the strict components in fact do not allow direct feedthrough between the particular ports that connect them in a loop. We call this an **artificial algebraic loop**.

Causality can easily be restored by inserting a **non-strict** component in an algebraic loop. A non-strict component also has an input/output dependency, but with a delay. Introducing a delay, however, may not always be a viable option, and solving an algebraic loop is difficult and not always possible. The problem of solving algebraic loops is a deep topic and is outside of the scope of this paper, but tractable algebraic loop solvers, like the one provided by Simulink, based on the Newton-Raphson technique, generally require smooth functions and are incompatible with discrete changes. Hence, particularly in the case of hybrid co-simulation, algebraic loops should be rejected.

FMI-ME 2.0 permits the use of strict FMUs, and includes a mechanism for FMUs to specify the inputs on which each of their outputs depend, so that algebraic loops can be detected and handled accordingly. In FMI-CS 2.0, on the other hand, requires FMUs to be non-strict. For this reason, FMI-CS 2.0 FMUs can be connected in a loop with the guarantee that no algebraic loops will occur. However, the price for

[2]A slightly less restrictive solution would be to have FMUs indicate their possible calling for step revision with a capability flag, and only require those upstream of such FMU to support rollback. For the sake of brevity, our MA does not feature that optimization.

this restriction is high, as it precludes the implementation of components that react instantaneously to changes in their input signal, like a Mealy machine does, which, unlike a Moore machine, produces outputs based on its current state *and* current inputs, not just on its current state. We would not, for instance, want to implement an adder as a Moore machine. There is no semantic significance to the arbitrary delay that (depending on the step size) the output of such adder would incur. Besides, what should the initial output of such adder be?

A Mealy machine requires separate functions for computing outputs and transitioning between states. FMI, on the other hand, prescribes that both output computation and state updates be performed in `fmi2DoStep`. Alternatively, in [2], the authors suggest to perform the computation of the outputs in `fmi2GetXXX`[3] and update the state in `fmi2DoStep`, allowing FMUs to be modeled as Mealy machines. Our implementation in FIDE follows this suggestion. Consequently, the FMUs in our examples all declare internal variable dependencies conform FMI-ME, so that the master can detect artificial algebraic loops and will only reject real ones. For FMUs that do not declare dependencies it is conservatively assumed that all outputs depend on all inputs. Finally, it should be noted that we allow the master to perform `fmi2GetXXX` on strict components more than once before calling `fmi2DoStep`, which is currently prohibited by the FMI-CS standard but necessary in order to handle feedback.

C. Predictable Step Sizes

The third and last extension proposed in [2] concerns the ability of an FMU to communicate to the master the size of the maximum step it is able to take. This extension requires a function called `fmi2GetMaxStepSize`, which returns the maximum step size that is acceptable by the FMU. The reason for having this function is to increase the efficiency of the algorithm. Saving and restoring the state of the FMUs is indeed an operation that takes time and requires memory. Moreover, it prevents the master from taking small steps if none of the participating FMUs have an interest in taking such small steps.

V. A Master Algorithm for Hybrid Co-simulation with Step Revision

A. A Formalization of FMI

In Figure 6 we provide a formalization of FMI based on the model given in [2] along with some minor extensions. Algorithm 1 makes use of this formal model. Each FMU $c \in C$, the set of all the FMUs in the model, is a black box with input port variables U_c, output port variables Y_c and internal state S_c. Different from [2], each variable type defined in FMI has been extended to include the absent value. $\mathbb{V}_\varepsilon$ is a union type that ranges over all possible data types in FMI, augmented with a value for absent. An example of an instance of $\mathbb{V}_\varepsilon$ is the type `fmi2Real` $\cup \{\varepsilon\}$. With respect to [2] we also introduce $\mathbb{M}$, which denotes the set of all possible values for the status flag returned by a function call to `fmi2DoStep` or `fmi2GetMaxStepSize`. We are only interested in three particular values of $\mathbb{M}$: `OK`, `Discard` and `Error`. `fmi2DoStep` and `fmi2GetMaxStepSize` return `OK` if the FMU completes an entire step or, in case of `fmi2GetMaxStepSize`, the FMU can take a step. `Discard` is returned if the FMU cannot perform an entire step but completes a fraction of it. Finally, `Error` is returned when an FMU is unable to continue the simulation based on its current state and the current value of its inputs.

Set of FMU instances in a model	C
FMU instance identifier	$c \in C$
Set of state valuations for instance c	S_c
Set of input port variables for instance c	U_c
Set of output port variables for instance c	Y_c
Set of values that a variable may take on	$\mathbb{V}_\varepsilon$[4]
Return status flag for `doStep` and `getMaxStepSize`	$\mathbb{M}$ = {`OK`, `Discard`, `Error`}
Set of all input variables in a model	$U = \bigcup_{c \in C} U_c$
Set of all output variables in a model	$Y = \bigcup_{c \in C} Y_c$
I/O dependency for instance c	$D_c \subseteq U_c \times Y_c$
Set of all I/O dependencies	$D = \bigcup_{c \in C} D_c$
Port variable mapping	$P : U \to Y$

$$
\begin{aligned}
\texttt{init}_c &: \mathbb{R}_{\geq 0} \to S_c \\
\texttt{set}_c &: S_c \times U_c \times \mathbb{V}_\varepsilon \to S_c \\
\texttt{get}_c &: S_c \times Y_c \to \mathbb{V}_\varepsilon \\
\texttt{doStep}_c &: S_c \times \mathbb{R}_{\geq 0} \to S_c \times \mathbb{R}_{\geq 0} \times \mathbb{M} \\
\texttt{getMaxStepSize}_c &: S_c \to \mathbb{R}_{\geq 0} \times \mathbb{M}
\end{aligned}
$$

Fig. 6: A formal model of FMI.

The function $\texttt{init}_c$ performs a sequence of operations to initialize the FMU c. Typically, this includes calls to `fmi2SetupExperiment`, `fmi2GetXXXHybrid`, and `fmi2SetXXXHybrid` [3] to determine initial conditions and retrieve parameters. $\texttt{init}_c(t)$ returns s, the initial state of the FMU. $\texttt{get}_c$ and $\texttt{set}_c$ represent the get and set operations `fmi2GetXXXHybrid` and `fmi2SetXXXHybrid` [3], respectively. $\texttt{get}_c(s, y)$ returns the value $v \in \mathbb{V}_\varepsilon$ of output port y of FMU instance c. $\texttt{set}_c(s, u, v)$ assigns the value $v \in \mathbb{V}_\varepsilon$ to the input port variable u of FMU instance c and returns the updated state $s \in S_c$ of the FMU c. A call to $\texttt{set}_c(s, u, v)$ updates the inputs of the FMU, while $\texttt{get}_c(s, y)$ affects the value of an output variable based on the FMU's current state and inputs. Multiple calls to $\texttt{get}_c(s, y)$ based on the same state and the same inputs of the FMU will return the same output values; $\texttt{get}_c(s, y)$ does not alter the state s of the FMU c. $\texttt{doStep}_c(s, h)$ abstracts the functionality of `fmi2DoStep`, which performs a state update on the FMU. A call to $\texttt{doStep}_c(s, h)$ returns the tuple (s', h', m) with the updated state s', the performed

[3]FMI provides different API functions for each variable data type. XXX is replaced by the data type of the variable.

[4]$\mathbb{V}_\varepsilon$ represents the union of the value set of a particular FMU variable and the absent value ε, $\mathbb{V}_\varepsilon = \mathbb{V} \cup \{\varepsilon\}$.

step size h', and the status flag m. Although the real FMI function `fmi2DoStep` does not take the FMU's state as an input argument and only returns the status flag m, our abstraction is sound because the updated state can be retrieved from the FMU with the function `fmi2GetFMUstate`. Likewise, the performed step size h' can be obtained by recording the "current FMU time" before and after invocation of `doStep` using `fmi2GetRealStatus`. Notice that by having `doStep` take the current state as an input argument and letting it return the new state obtained by completing a step, we can use it to model rollback without introducing extra primitives for retrieving and restoring the state. The function $\texttt{getMaxStepSize}_c(s)$ abstracts `fmi2GetMaxStepSize`. This function returns a pair (h, m), where h is the maximum step size that the FMU can perform, and m is the return status flag. This function has no side effects; its invocation shall not alter the FMU's state.

We model each FMU as a Mealy machine with some internal I/O dependencies D_c. If a model, consisting of interconnected FMUs, contains an algebraic loop, then we discard that model. Hence, the graph $\mathbb{X}$ corresponding to a valid model, where vertices are port variables $U \cup Y$ and edges are $E = D \cup \{(y, u) \mid u \in U \wedge P(u) = y\}$, must be acyclic. Graph $\mathbb{X}$ can be totally ordered using Kahn's algorithm [17], for example. We call the totally ordered acyclic graph $\bar{\mathbb{X}}$.

B. The Algorithm

Algorithm 1, coordinates the simulation of a model with a set of FMUs from the initial time $t = t_{start}$ to t_{end}. At each simulation step, the algorithm tries to advance from a time instant t_2 to $t_3 = t_2 + h$, where h is the computed step size. The algorithm has the capability to rollback each FMU state to its previous known state in case one or more FMUs return an error status from the invocation of `doStep` or `getMaxStepSize`.

The vectors r and r' store the state of all FMUs during the execution of the algorithm, and s is a vector that denotes the current state of all FMUs. The current state of the FMU c, for instance, is $s[c]$. In order to perform rollback during step size determination, r stores the state of all FMUs *after the last I/O port update*, prior to a speculative step. Step refinement requires a snapshot of the collective state *after the last successful execution step*, which is stored in r'.

A simulation consists of four phases:

a) Initialization (lines 3 to 5): The master iterates over all the FMUs $c \in C$ to setup the simulation and set the initial value of variables and parameters.

b) I/O ports update (lines 7 to 11): The master iterates over the totally ordered list of ports. For each input port u, function P returns the output port y connected to it. The value of the output is retrieved and used to update the corresponding input port. After all I/O ports have been updated, the algorithm saves the state of each FMU in the state vector r and initializes the $errorState$ flag to $false$ (lines 12 to 13). The $errorState$ flag will indicate whether the master needs to perform rollback.

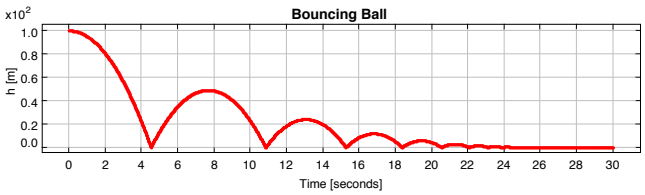
c) Step determination (lines 14 to 32): The master determines the maximum step size that is acceptable by all the FMUs. We have so-called *predictable* FMUs ($C_P \subseteq C$), which implement `getMaxStepSize`, and *unpredictable* FMUs ($C_R \subseteq C$), which do not. Conform [2], we assume that `doStep` with step size argument h returns a step size h' such that $0 \leq h' \leq h$, and `doStep` will accept any step less than or equal to that h'. For predictable FMUs, `doStep` shall accept any step less than or equal to the size returned by `getMaxStepSize`. Partial progress in `doStep` is indicated with a status flag `Discard` while acceptance of the step will yield `OK`. In addition, both `doStep` and `getMaxStepSize` may return a status code `Error` if the FMU is stuck, i.e., it cannot continue the simulation given its current state and inputs.[5]

The procedure of step determination starts by setting a default value $h = h_{max}$. We first iterate over the FMUs $c \in C_P$ to retrieve the maximum step that each of those FMUs is able to accept. The minimum in the set of all reported values $\cup\ h_{max}$, which we store in h, is the maximum step that all $c \in C_P$ are able to accept. We then iterate over all unpredictable FMUs $c \in C_R$ and invoke `doStep` to test whether they accept a step of size h. The FMUs that reject the step will report partial progress. The next simulation step is determined by taking the minimum of the reported progress among each $c \in C_R$ and our previously computed h. At this point, the unpredictable FMUs are unsynchronized and must be rolled back to the state r. In the algorithm this is denoted by performing a state update using the FMU state r instead of s (line 43). If during step determination any of the FMUs return `Error`, the variable $errorState$ is set to $true$, step determination is canceled (lines 18 and 28), and the master restores all FMUs to their previous state (lines 34 to 38).

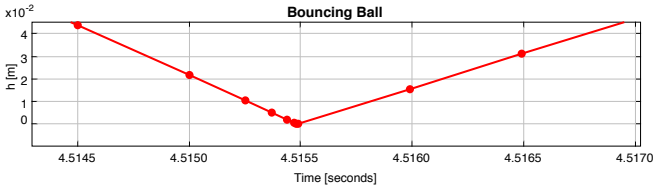
d) Step revision (lines 34 to 38): If $errorState$ is $true$, the model must be rolled back to r' (line 35). The current state s is restored to $s := r'$, where r' is the saved state from the previous iteration. From here, a smaller step h must be taken as the previous step h_p resulted in an `Error`. At line 36, function $f(h_p)$ computes h such that $0 \leq h < h_p$. How h is computed is outside of the scope of our discussion, and its behavior is left unspecified. However, the interested reader can find a detailed description of algorithms for the implementation of $f()$ in [7] and [4]. To finalize this iteration of step revision, the master backtracks the simulation time to $t := t - h_p$ and saves the computed step size $h_p := h$ (with $h := f(h_p)$) in case that also in the next iteration some FMU yields an error flag and a further refinement of the step size is needed.

e) State update (lines 39 to 55): If $errorState$ is $false$, the algorithm saves the initial state $r' := r$ (line 40) and performs a state update on all FMUs (lines 43 and 48). The master saves the computed step size $h_p := h$, advances the

[5]We assume that $\texttt{doStep}_c$ and $\texttt{getMaxStepSize}_c$ cannot return $m = Error$ at the first execution of the algorithm. If this happens, the initial condition of the FMU prevents the simulation from starting (see Algorithm 1 at line 34).



(a) Trajectory of a bouncing ball, implemented as an FMU.



(b) Step revision prevents the ball from tunneling through the surface.

Fig. 7: Simulation output obtained through FIDE.

simulation time $t := t + h$ and resets the step size to its default $h := h_{max}$. The states r' and the step size h_p will be used at the next execution of the algorithm to backtrack the simulation if needed.

VI. An Example

The QTronic Software Development Kit (SDK) [6] contains an example of a bouncing ball simulated by a fixed-step-size MA. In this example, the FMU modeling the bouncing ball enforces no control over the step size and the model suffers from a tunneling effect (the ball eventually bounces below the level of the surface). When a zero crossing is caught too late, the ball changes trajectory *after* it tunneled through the surface. In such model the behavior of the simulation is heavily dependent on the simulation engine, the master, because it controls the step size.

We now demonstrate our MA using a similar physical model. Consider an FMU that implements a bouncing ball and makes use of the FMI extensions for hybrid co-simulation presented in Section IV. In order to show how step revision improves the accuracy of the bouncing ball, the FMU returns a fixed step size $h = 0.05\,sec$ through `getMaxStepSize`. However, before returning the value of the step size, `getMaxStepSize` checks the tunneling condition (the position h of the ball results $h \leq -\varepsilon$) and returns an error flag when it detects tunneling. This way, if tunneling occurs, the master will backtrack the simulation and refine the previous step size until the ball hits the surface with a precision that is a property not emergent from the simulation, but defined by the FMU. The result of this simulation is shown in Figure 7a. In addition, Figure 7b shows the dynamics of the ball in more detail as it approaches the surface. First the FMU overshoots, detecting the zero crossing at 4.5155, and then the master backtracks to 4.515 and uses a bisection search to iteratively converge to 4.51549, right before the ball hits the surface.

[6] www.qtronic.de/en/fmusdk.html

Algorithm 1: Co-simulation MA with Step Revision.

Input: Set of instances C, ordered variable list $\bar{\mathbb{X}}$, port mapping P, the maximal step size h_{max}.

$t := t_{start}$;
$h := h_{max}$ set the initial value for the step size;
foreach $c \in C$ **do** initialize
  $s[c] := \texttt{init}_c(t)$;
end
while $t \leq t_{end}$ **do**
  foreach $u \in \bar{\mathbb{X}}$ *in order* **do** I/O ports update
    $y := P(u)$;
    $v := \texttt{get}_{c_y}(s[c_y], y)$;
    $s[c_u] := \texttt{set}_{c_u}(s[c_u], u, v)$;
  end
  $r := s$ save the current state;
  $errorState := false$;
  foreach $c \in C_P$ **do** determine the current step
    $(h', m) := \texttt{getMaxStepSize}_c(s[c])$;
    $h := min(h, h')$;
    if m = `Error` **then** this is not a valid state
      $errorState := true$;
      break;
    end
  end
  if $errorState$ = *false* **then**
    foreach $c \in C_R$ **do** determine the current step
      $(s', h', m) := \texttt{doStep}_c(s[c], h)$;
      $h := min(h, h')$;
      $s[c] := s'$;
      if m = `Error` **then** this is not a valid state
        $errorState := true$;
        break;
      end
    end
  end
  if $errorState$ **then** revise the last step
    if $t = t_{start}$ **then** quit due initialization error;
    $s := r'$ restore the previous valid state;
    $h := f(h_p)$ choose a smaller step size $h < h_p$;
    $t := t - h_p$ rollback simulation time;
    $h_p := h$ save the step size h;
  else this is a valid state
    $r' := r$ save the valid state;
    if $h < h_{max}$ **then** roll back and perform step h
      foreach $c \in C$ **do** state update
        $(s', h', m) := \texttt{doStep}_c(r[c], h)$;
        $s[c] := s'$;
      end
    else update the predictable FMUs
      foreach $c \in C_P$ **do** state update
        $(s', h', m) := \texttt{doStep}_c(r[c], h)$;
        $s[c] := s'$;
      end
    end
    $h_p := h$ save the step size h;
    $t := t + h$ advance simulation time;
    $h := h_{max}$ reinitialize the step size;
  end
end

Of course, even with step refinement, if we run the simulation long enough, we will eventually exhaust the finite precision of the numerical representation of our step size and are no longer able to represent a step small enough to prevent the ball from tunneling.

VII. Conclusions

FMI shows great promise for enabling interoperability between simulation tools for CPS. However, version 2.0 of the co-simulation standard is not suited to simulate hybrid systems. The extensions presented in [3] and [2] repair most of the shortcomings responsible for this. In this paper, we identify an important and common co-simulation scenario that is still not handled properly without a further extension to the master algorithm given in [2], namely, one which allows the simulation to revise any given simulation step immediately after it has been taken. As we have shown, the necessity of this feature is rooted in the fact that FMI does not equip FMUs with a side-effect-free evaluation function along with a separate function for updating their state.

The proposed step revision technique relies on each FMU to support rollback by implementing functions to copy and restore their state, which, so far, have been deemed optional by the FMI standard. Yet, to support step revision, co-simulation FMUs must implement these functions. Step revision allows FMUs to interpolate continuous input signals instead of extrapolating them. FMUs without extrapolation capabilities are simpler and more composable, and as demonstrated in this paper, step-size-dependent numerical approximation errors that may occur in such FMUs can be kept within bounds by a master that uses step revision.

Acknowledgments

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, 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 Academy of Finland and by NSF awards #1329759 and #1139138 and by the Clean Sky 2 JU European Union's Horizon 2020 programme under grant agreement No CS2-SYS-GAM-2014-2015-01.

References

[1] Modelica Association, "Functional Mock-up Interface for Model Exchange and Co-Simulation," Report 2.0, 2014.

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

[3] F. Cremona, M. Lohstroh, S. Tripakis, C. Brooks, and E. A. Lee, "FIDE – An FMI Integrated Development Environment," in *Symposium on Applied Computing (SAC)*, 2016.

[4] F. E. Cellier and E. Kofman, *Continuous System Simulation*. Springer, 2006.

[5] C. Ptolemaeus, Ed., *System Design, Modeling, and Simulation using Ptolemy II*. Berkeley, CA: Ptolemy.org, 2014. [Online]. Available: http://ptolemy.org/books/Systems

[6] E. Kofman and S. Junco, "Quantized-state systems: A DEVS approach for continuous system simulation," *Transactions of The Society for Modeling and Simulation International*, vol. 18, no. 1, pp. 2–8, 2001.

[7] F. Zhang, M. Yeddanapudi, and P. Mosterman, "Zero-crossing Location and Detection Algorithms for Hybrid System Simulation," in *17th IFAC World Congress, Seoul, South Korea*, 2008, pp. 7967–7972.

[8] P. Deng, F. Cremona, Q. Zhu, M. Di Natale, and H. Zeng, "A Model-based Synthesis Flow for Automotive CPS," in *Proceedings of the ACM/IEEE Sixth International Conference on Cyber-Physical Systems*, ser. ICCPS '15. New York, NY, USA: ACM, 2015, pp. 198–207.

[9] V. Galtier, S. Vialle, C. Dad, J.-P. Tavella, J.-P. Lam-Yee-Mui, and G. Plessis, "FMI-based Distributed Multi-Simulation with DACCOSIM," in *Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*. Society for Computer Simulation International, 2015, pp. 39–46.

[10] D. Broman, L. Greenberg, E. A. Lee, M. Masin, S. Tripakis, and M. Wetter, "Requirements for Hybrid Cosimulation Standards," in *Proceedings of 18th ACM International Conference on Hybrid Systems: Computation and Control (HSCC)*. ACM, 2015, pp. 179–188.

[11] J. Denil, B. Meyers, P. De Meulenaere, and H. Vangheluwe, "Explicit Semantic Adaptation of Hybrid Formalisms for FMI Co-simulation," in *Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*. Society for Computer Simulation International, 2015, pp. 99–106.

[12] B. Camus, V. Galtier, M. Caujolle, V. Chevrier, J. Vaubourg, L. Ciarletta, and C. Bourjot, "Hybrid Co-simulation of FMUs using DEV&DESS in MECSYCO," in *Symposium on Theory of Modeling & Simulation-DEVS Integrative M&S Symposium*, 2016, pp. 568–575.

[13] B. P. Zeigler, H. S. Song, T. G. Kim, and H. Praehofer, "DEVS Framework for Modelling, Simulation, Analysis, and Design of Hybrid Systems," in *International Hybrid Systems Workshop*. Springer, 1994, pp. 529–551.

[14] T. Blochwitz, M. Otter *et al.*, "Functional Mock-up Interface 2.0: The Standard for Tool independent Exchange of Simulation Models," in *Proceedings of the 9th International Modelica Conference*, 2012.

[15] O. Maler, Z. Manna, and A. Pnueli, "From Timed to Hybrid Systems," in *Real-Time: Theory and Practice, REX Workshop*. Springer-Verlag, 1992, pp. 447–484.

[16] E. A. Lee and H. Zheng, "Operational Semantics of Hybrid Systems," in *Hybrid Systems: Computation and Control (HSCC)*, M. Morari and L. Thiele, Eds., vol. LNCS 3414. Zurich: Springer-Verlag, 2005, pp. 25–53.

[17] A. B. Kahn, "Topological sorting of large networks," *Communications of the ACM*, vol. 5, no. 11, pp. 558–562, 1962.