Determinate Composition of FMUs for Co-Simulation

David Broman
Christopher Brooks
Lev Greenberg
Edward A. Lee
Michael Masin
Stavros Tripakis
Michael Wetter

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2013-153
http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-153.html

August 18, 2013

(C) 2013 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

Acknowledgement

This is an extended version of the accepted version of the paper appearing in the proceedings of the International Conference on Embedded Software (EMSOFT 2013).

Reference to the published version:

David Broman, Christopher Brooks, Lev Greenberg, Edward A. Lee, Michael Masin, Stavros Tripakis, Michael Wetter. Determinate Composition of FMUs for Co-Simulation, In Proceedings of the 13th International Conference on Embedded Software (EMSOFT), Montreal, Canada, September 29 - October 4, IEEE, 2013.

Determinate Composition of FMUs for Co-Simulation*

David Broman[1,2] Christopher Brooks[1] Lev Greenberg[3] Edward A. Lee[1]
Michael Masin[3] Stavros Tripakis[1] Michael Wetter[4]

{broman,cxh,eal,stavros}@eecs.berkeley.edu, {levg,michaelm}@il.ibm.com, mwetter@lbl.gov

[1]University of California, Berkeley, USA [2]Linköping University, Sweden
[3]IBM [4]LBNL, Berkeley, CA, USA

August 18, 2013

Abstract

In this paper, we explain how to achieve deterministic execution of FMUs (Functional Mockup Units) under the FMI (Functional Mockup Interface) standard. In particular, we focus on co-simulation, where an FMU either contains its own internal simulation algorithm or serves as a gateway to a simulation tool. We give conditions on the design of FMUs and master algorithms (which orchestrate the execution of FMUs) to achieve deterministic co-simulation. We show that with the current version of the standard, these conditions demand capabilities from FMUs that are optional in the standard and rarely provided by an FMU in practice. When FMUs lacking these required capabilities are used to compose a model, many basic modeling capabilities become unachievable, including simple discrete-event simulation and variable-step-size numerical integration algorithms. We propose a small extension to the standard and a policy for designing FMUs that enables deterministic execution for a much broader class of models. The extension enables a master algorithm to query an FMU for the time of events that are expected in the future. We show that a model can be executed deterministically if all FMUs in the model are either memoryless or implement one of rollback or step-size prediction. We show further that such a model can contain at most one "legacy" FMU that is not memoryless and provides neither rollback nor step-size prediction.

1 Introduction

FMI (Functional Mockup Interface) is an evolving standard for composing model components designed using distinct modeling tools [3, 4, 16, 17, 18]. Initially developed within the MODELISAR project, and currently

*This is an extended version of the accepted version of the paper appearing in the proceedings of the International Conference on Embedded Software (EMSOFT 2013). © 2013 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.
Reference to the published version: David Broman, Christopher Brooks, Lev Greenberg, Edward A. Lee, Michael Masin, Stavros Tripakis, Michael Wetter. Determinate Composition of FMUs for Co-Simulation, In Proceedings of the 13th International Conference on Embedded Software (EMSOFT), Montreal, Canada, September 29 - October 4, IEEE, 2013.
Acknowledgements: This work was supported in part by the iCyPhy Research Center (Industrial Cyber-Physical Systems, supported by IBM and United Technologies), and the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley (supported by the National Science Foundation, NSF awards #0720882 (CSR-EHS: PRET), #1035672 (CPS: Medium: Ptides), and #0931843 (ActionWebs), the Naval Research Laboratory (NRL #N0013-12-1-G015), and the following companies: Bosch, National Instruments, and Toyota). This work was also supported in part by the NSF Expeditions in Computing project *ExCAPE: Expeditions in Computer Augmented Program Engineering* and *COSMOI: Compositional System Modeling with Interfaces*. This research was supported by the Assistant Secretary for Energy Efficiency and Renewable Energy, Office of Building Technologies of the U.S. Department of Energy, under Contract No. DE-AC02-05CH11231. The first author was funded by the Swedish Research Council #623-2011-955.

supported by a number of industrial partners and tools (see `https://www.fmi-standard.org/`), FMI shows enormous promise for enabling the exchange and interoperation of model components. FMI is particularly suitable for cyber-physical systems (CPSs), where model components may represent distinct subsystems that are best designed with distinct modeling tools. The FMI standard supports both co-simulation (where a component, called an FMU (Functional Mock-up Unit), implements its own simulation algorithm) and model exchange (where an FMU describes the model sufficiently for an external simulation algorithm to execute simulation). In this paper we focus the discussion on co-simulation in the current version of the standard (version 2.0, Beta 4 [16]).

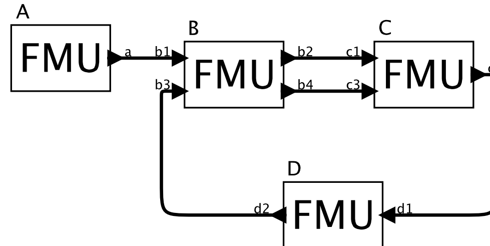

Figure 1: A model consisting of FMUs connected in a block diagram.

A *model* is a collection of interconnected FMUs, as shown in Figure 1. These FMU *slaves* are to be executed by some *master algorithm* (MA), which orchestrates the execution of the FMUs, according to its own semantics. The MA orchestrates the communication of the FMUs through their inputs and outputs, shown in the figure as black triangles. The MA may be the execution engine of an established simulation tool such as Simulink, in which case the model may also include native Simulink blocks interacting with the FMUs. The semantics implemented by the MA is that of the host tool. Each FMU may have been exported by some other tool, such as Dymola. The goal of FMI is to enable an FMU exported by one tool to interoperate with a variety of host tools, and for host tools to orchestrate interactions between FMUs exported by a variety of other tools.

In principle, FMI is capable of composing components representing timed behavior, including physical dynamics and discrete events. However, there are significant subtleties and limitations. In particular, it is possible to design FMUs and MAs that are compliant with the standard, and yet exhibit nondeterministic and unexpected behavior. In this paper, we explain how to achieve deterministic execution of FMUs. Specifically, the contributions of this paper are the following:

- We give conditions on the design of FMUs and MAs. With the current version of the standard, these conditions demand capabilities from FMUs that are optional in the standard and rarely provided by an FMU in practice (Section 3).
- We clarify and formalize a subset of the FMI standard and give constraints (*contracts*) on the expected behavior of FMUs and MAs, so that interoperation has consistent and predictable results (Sections 4).
- We propose a master algorithm for the co-simulation step that only requires capabilities of the current standard. We prove that the algorithm is convergent, determinate, and ensures maximal progress (Section 5.1).
- We propose a small extension to the standard that enables deterministic execution for a much broader class of models than possible with the current standard. Specifically, we present a new MA where all FMUs either implement rollback (which is expected to be rarely supported by FMUs) or implement our proposed extension to the standard (which is easier to support). We show further that such a model can contain at most one FMU (which we call a “legacy FMU”) that provides neither capability. We prove that this new algorithm is convergent, determinate, and ensures maximal progress (Section 5.2).

2 FMI for Co-Simulation

In this section, we explain capabilities and limitations of the FMI 1.0 and 2.0 beta 4 standards.

2.1 FMI 1.0

The method provided by FMI for advancing time in co-simulation is called `fmiDoStep`. It is implemented as a C procedure by the slave FMU. The `fmiDoStep` method is called by the MA, which coordinates data exchange between the slaves to control the entire co-simulation. In FMI 1.0, `fmiDoStep` has the following signature [17]:

```
fmiStatus fmiDoStep(
   fmiComponent c,
   fmiReal currentCommunicationPoint,
   fmiReal communicationStepSize,
   fmiBoolean newStep);
```

The `currentCommunicationPoint` argument is the current simulation time of the master, whereas `communicationStepSize` is the time step that the master proposes to the slave. If, for example, the slave encompasses a model with an ordinary differential equation (ODE), then its numerical integration algorithm should attempt to advance time by this proposed step. The slave may accept or reject the step. For instance, it may reject it if the step size is too large and hence causes a discrete event, such as a zero crossing, within the step.

The MA generally interacts with more than one slave, and uses the boolean flag `newStep` to indicate whether the proposed step is a new step, or whether the previously attempted step was rejected by some slave, and therefore is being repeated.

Although our paper is restricted to co-simulation, we point out an important class of problems in which a MA not only controls FMUs for co-simulation, but also integrates in time FMUs for model exchange or ODEs that are implemented in the simulation environment that provides the MA. In this situation, the MA needs to provide a time integration algorithm. For reasons of computational efficiency, one may want to use an adaptive step-size, multi-step time integration algorithm. Unfortunately, FMI 1.0 does not support this, as we will now show. Consider a MA that contains an adaptive (variable) step-size Runge-Kutta 2-3 solver. Let the communication point be 0 and the Runge-Kutta step size be 1.0. Then, the MA needs to obtain outputs from the FMU at times 0, 0.5, 0.75, and 1. Thus, there need to be three calls to `fmiDoStep`, e.g.,

```
fmiDoStep(component, 0.0,  0.5,  true);
fmiDoStep(component, 0.5,  0.25, ?);
fmiDoStep(component, 0.75, 0.25, ?);
```

The `newStep` argument is `true` in the first call, but what should it be in the next two calls? They cannot specify `newStep = false`, because then the slave would interpret this as restarting the previous computation, which has an earlier communication point. But they cannot specify `newStep = true` either, because if the last `fmiDoStep` is rejected by a slave, then all three steps have to be redone. "Rolling back" more than just to the previous time point (*multistep rollback*) is not possible, however, with the API of FMI 1.0.

2.2 Extensions in FMI 2.0

The FMI 2.0 beta 4 [16] standard adds more methods to the API. We will focus on two important additions, `fmiGetFMUstate` and `fmiSetFMUstate`, which allow the master to copy and later restore the complete state of an FMU slave. These procedures provide a general mechanism for rollback, but they are optional for practical reasons. In practice, an FMU may wrap a large piece of legacy software, enabling the use of that software with other modeling and simulation tools, and it may not be practical to retrieve and restore the state of such a legacy system. We will show, however, that if these procedures are not provided by an FMU, then that FMU will not be usable in some models, as the results of executing the FMU will not match the intended semantics of the FMU. We give precise conditions under which correct and deterministic execution of such FMUs is possible. In FMI 2.0 beta 4, `doStep` has a slightly modified signature [16]:

```
fmiStatus fmiDoStep(
   fmiComponent c,
   fmiReal currentCommunicationPoint,
   fmiReal communicationStepSize,
   fmiBoolean noSetFMUStatePriorToCurrentPoint);
```

The types of the arguments are the same as in version 1.0, but the name and semantics of the boolean argument are now different: by setting `noSetFMUStatePriorToCurrentPoint` to `true`, the master "promises" that it will not call `fmiSetFMUState` for time instants prior to `currentCommunicationPoint`.

Even though the API of FMI 2.0 is richer, and being able to read and write the state of an FMU provides the master with considerable power, the situation is still not entirely satisfactory, for a number of reasons. First, implementation of the rollback procedures by an FMU is optional and may be difficult to achieve in practice. Second, saving and restoring the state may incur a large overhead at run time. The API is not rich enough in general, as we will show, to prevent preventable rollbacks. Third, the interface does not quite bridge the gap between purely continuous-time models, and other modeling formalisms, such as discrete-event, synchronous-reactive, dataflow, and state-machines.

2.3 Extrapolation of Input Values

For co-simulation, time advances from a communication point t to a communication point $t + h$ when the MA calls `fmiDoStep` with step size h. When this call occurs, the FMU has been provided with input values at time t. But many integration algorithms require input values during the interval $[t, t + h]$. For example, if an implicit solver is used, then the input value at $t + h$ will also be needed in order to determine the value of the state and output at time $t + h$. To support this, the FMI standard allows for the MA to provide derivatives of the inputs at time t. The FMU will presumably then use these derivatives to extrapolate the input signals to calculate suitable approximations during the interval.

3 Requirements of FMUs

In this section we identify properties of FMUs that are required for those FMUs to be executed deterministically by a MA in an arbitrary model. Specifically, we illustrate the need for I/O dependency information and the need for rollback. Currently, both of these are optional in the standard. An FMU may or may not provide them.

3.1 I/O Dependency Information

A MA provides input data to an FMU by calling a procedure called `fmiSetXXX` provided by the FMU, where "`XXX`" is replaced by the data type of the input. For example, for a real-valued input, the MA will call `fmiSetReal`. An argument to this procedure specifies which input is being set, and another argument provides the value. To retrieve an output value from an FMU, the MA calls `fmiGetXXX`, providing as an argument the identity of the output and a pointer to a location into which to store the output. A key question in the design of an MA becomes, in which order should these procedures be called? The question is particularly subtle in models with feedback.[1]

To answer this question, it is helpful to know which outputs of an FMU depend immediately on which inputs. FMI makes it possible (although not mandatory) to provide such information. In FMI 1.0 this is done via the *DirectDependency* element in the XML-description of an FMU [3, 17, 18]. In FMI 2.0 this information can be provided using the element *ModelStructure* [4, 16]. In this section we explain by example why such information is often essential.

[1] Figure 10 on page 82 of [16] shows a statechart called the "State Machine of Calling Sequence from Master to Slave" which specifies the calling sequences of the API methods which must be supported by an FMU. Methods `fmiGetXXX` and `fmiSetXXX` appear unordered in a self-loop of the state machine, however, therefore this state machine does not help in answering the question above.

Suppose we want to create an FMU with the functionality shown in Figure 2. The figure shows a continuous-time model modeling a trivial ODE, where the input u is the derivative of the output *y1*, and the output *y2* is the input u multiplied by -5. The interesting feature of the model is that output *y2* has a *direct dependency* on input u, whereas the output *y1* does not. Specifically, at any time t in a simulation, the output *y2* depends on the value of the input *at that same time* t, whereas the output *y1* depends only on previous inputs. The equations realized by this FMU are

$$u = \frac{dy_1}{dt} \qquad y_2 = -5u.$$

In general, a MA may need to know that *y1* can be produced by the FMU even before u is known. Suppose for example, that we want to use the FMU of Figure 2 to simulate the following system of equations,

$$u = f(y_1) \qquad u = \frac{dy_1}{dt} \qquad y_2 = -5u.$$

This is shown in the model to the left of Figure 3. How should a MA execute this model? One brute force approach is to assume that the feedback loop represents an *algebraic loop*. The MA could guess the value of u at a time, call `fmiSetReal` to set the input u equal to this guess, then call `fmiGetReal` to retrieve the output values for *y1* and *y2*. It could then evaluate the function f and update the value for u. It could repeat this process until the value for u no longer changes. And in this example, indeed, the solution would converge quickly because *y1* does *not* directly depend on u.

In general, however, such an iterative technique for solving algebraic loops may not converge, and when it converges, the value it converges to may depend on the initial guess. Suppose for example that with the same FMU of Figure 2 we connect it as shown to the right of Figure 3. In this case, there is a genuine algebraic loop. Suppose the function f is the identity function. Then in this case, if the initial guess for u is zero, then the execution converges immediately. If the initial guess is anything other than zero, then the execution diverges. Suppose instead that f is given by

$$f(x) = (0.2\,x)^2.$$

In this case, the feedback loop asserts that $u = u^2$, and there are two possible solutions, $u = 0$ and $u = 1$. The solution that a brute-force MA converges to will depend on the initial guess.

Since we are interested in non-diverging and deterministic composition of FMUs, we need to distinguish these two cases and we need to reject the case with the algebraic loop. That model is not deterministic. A MA that ensures determinacy needs to know that *y1* does *not* directly depend on u, and that *y2* does. Once it knows this, it can also execute the correct case (without the algebraic loop) more efficiently, so we get an additional benefit. In particular, the direct dependency information can be used by a MA to call the `fmiGetReal` and `fmiSetReal` functions for outputs and inputs in a well-defined order. For the leftmost model of Figure 3, the MA can execute the following sequence:

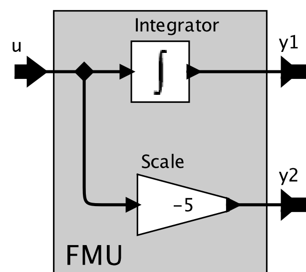

Figure 2: A continuous-time FMU where output y1 does not have a direct dependency on input u and output y2 does.

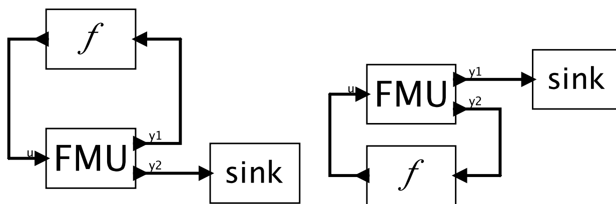

Figure 3: FMU of Figure 2 connected in feedback (left) and in an algebraic loop (right).

1. Use `fmiGetReal` to read the current value of *y1*. This value is available without knowing the current input u.
2. Evaluate f (which may itself be an FMU or it may be a native component), and use `fmiSetReal` to set the current value of u.
3. Use `fmiGetReal` to read the current value of *y2*. Notice that since the value of *y2* depends directly on the current value of u, `fmiGetReal` needs to perform a calculation in the case of *y2*.[2] Concretely, it needs to return $y_2 = -5u$.

Notice that in the sequence, `fmiGetReal` and `fmiSetReal` are called exactly once, the minimum possible number of times for any MA.

For the rightmost example of Figure 3, if the MA has direct dependency information, then it can identify the algebraic loop and either reject the model or alert the user to the potential nondeterminacy.

3.1.1 Static Analysis for Dependency Cycles

For the leftmost example of Figure 3, we were able to use I/O dependency information to identify the order in which the `fmiSetReal` and `fmiGetReal` procedures should be called at a communication point. Moreover, in this ordering, as long as there is no algebraic loop, these procedures are called exactly once, leading to efficient execution.

We can generalize this idea to models with arbitrarily complicated structure, constructing a *topological sort* of the ports in a model. Consider the model shown in Figure 1. It consists of four blocks, A, B, C, D.

Suppose that the following is known about the I/O dependencies of the blocks of Figure 1:

$$b_1 \to b_4 \quad b_3 \to b_2 \quad c_3 \to c_2 \quad d_1 \to d_2$$

where $x \to y$ means that output port y depends directly on input port x. In addition, assume that these are the only I/O dependencies. This means, in particular, that output b_2 of block B does *not* depend on input b_1, that the output of C does *not* depend on its input c_1, and so on.

In addition to the input-output dependencies induced by each block, output-input dependencies are induced by the connections in the diagram. Together all these dependencies define a directed graph whose nodes represent ports. Each edge $x \to y$ in the graph represents the fact that y depends on x. The port dependency graph for this example is shown in Figure 4.

As can be observed, the port dependency graph of Figure 4 is acyclic. If the port dependency graph of a given model does not contain cycles, then this graph can be used to derive a correct evaluation order of all ports in the model. From the port dependency graph shown in that figure, and the knowledge about which ports are inputs and which are outputs, a sequence of calls to `fmiSetXXX` and `fmiGetXXX` can be easily constructed. In general, the dependency graph resulting from such analysis gives a partial order on the calls to `fmiGetXXX` and `fmiSetXXX`, although for the example in Figure 4 the order is total.

[2]With some care to not do so prematurely, the calculation could alternatively be performed in the call to `fmiSetReal`.

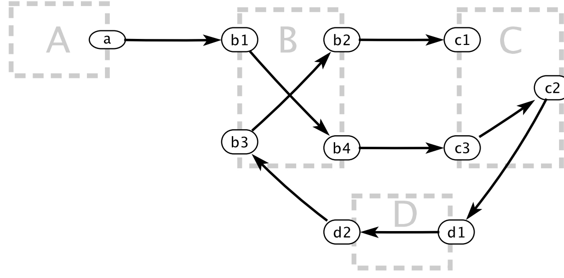

Figure 4: The port dependency graph generated from the model of Figure 1. The graph is acyclic.

3.1.2 If I/O Dependency Information is Missing

The FMI standard makes provision of I/O dependency information optional, presumably under the assumption that execution is still possible without this information, albeit less efficiently. Indeed, this is true if all `fmiSetXXX` and `fmiGetXXX` procedures are free of side effects (they make no changes to the state of the FMU), and that the corresponding mechanisms for setting inputs and retrieving outputs for native simulation components are also free of side effects. A MA can just execute the model in the same manner that it would solve algebraic loops, repeatedly invoking `fmiSetXXX` and `fmiGetXXX` until it gets convergence or gives up and declares a failure to converge.

However, these are rather strong assumptions. Even if a designer of simulation components intends to follow these guidelines, it is easy to make mistakes. Such mistakes lead to very subtle bugs that are difficult to track down. They could also result in nondeterministic models, and the nondeterminism might go unnoticed because it fails to manifest as variable behavior during testing.

3.2 The Need for Rollback

A main difference between FMI 1.0 and 2.0 is that FMI 2.0 includes functions to save and restore the state of an FMU. However, implementation of these functions by an FMU is optional, not mandatory [4, 16]. Saving and restoring the state of an FMU can be used to implement a rollback mechanism, which is often needed as we now explain.

Consider the example shown in Figure 5. The figure shows two FMUs connected in series. Consider a MA that co-simulates these two FMUs. To advance time in the simulation, the master needs to call `fmiDoStep` on both FMUs. The MA needs to pick an order to do so; it must either call `fmiDoStep` first on FMU1 and then FMU2, or vice-versa. In both cases, there is the possibility that the FMU on which `fmiDoStep` is called first *accepts* the proposed communication step size whereas the FMU on which `fmiDoStep` is called last *rejects* it. This is problematic as it requires to *roll back* the FMU which accepted the step.

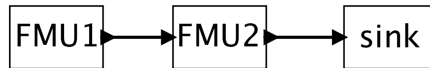

Figure 5: Two FMUs connected in series.

To make the scenario more concrete, suppose that the MA calls first `fmiDoStep` on FMU1, passing a communication step size h. It means that the MA "asks" whether the FMU can advance its local time (and correspondingly also evolve its state) from the current time t to $t + h$. If the FMU accepts the step, the implied semantics is that the state of FMU1 has now evolved to a state at future time $t + h$. If the FMU rejects the step, the implied semantics is that the state of FMU1 is still the one at time t (it remains unchanged). A third possibility is that the FMU manages to make partial progress and advance to some time $t + h'$ with $h' < h$.

Suppose for the sake of this example that FMU1 accepts h. Next, the MA calls `fmiDoStep` on FMU2, with the same communication step size, h. Suppose that FMU2 rejects the step. We are now at a situation where the state of FMU1 is at time $t+h$, while the state of FMU2 is at time t. To proceed with the simulation, the master needs to choose a new (smaller) communication step size, reset the state of FMU1 back to time t, and repeat with the new step size.

If FMU2 makes partial progress to time $t+h'$, then time has advanced to $t+h$ for FMU1 and to $t+h'$ for FMU2. If the MA chooses $t+h'$ as the next communication point, and then calls `fmiGetXXX` to retrieve the output of FMU1, FMU1 will likely respond with the wrong output value, corresponding to a future time point. Again, resetting the state ("rolling back") to time t is necessary. Note that this is true even in models without feedback, as the one shown in Figure 5.

Rolling back to time t can be achieved as follows. Before calling `fmiDoStep`, the MA uses `fmiGetFMUstate` to copy the state[3] of both FMU1 and FMU2 at time t. It then attempts to advance time by h. If this is accepted by both FMUs, the MA has succeeded. Otherwise, if say FMU2 rejects the step, then the MA can call `fmiSetFMUstate` on FMU2 to reset it back to its (copied) state at time t. However, the implementation of `fmiGetFMUstate` and `fmiSetFMUstate` is left optional in the FMI standard.

Moreover, it may appear that if *any* FMU in a model can reject a step size, then *all* FMUs in the model need to support rollback. Indeed, a first algorithm that we present in Section 5 (Algorithm 2), operates like this. We will also show, however, that with careful design and with a small extension to the standard, a model can contain some FMUs that do not support rollback (Algorithm 3).

4 FMI Formalization

In this section we formalize a core subset of FMI and propose an explicit contract between FMUs and MAs.

4.1 Function Interface

The FMI standard describes the signatures of the C functions together with informal descriptions of their meaning. In this paper, we propose a formalization of FMI, which allows to prove properties of MAs, in particular determinacy and maximal progress (see Section 5). Towards this goal, we formalize the core subset of the FMI 2.0 specification, leaving out parts that are not relevant for the discussion.

The formalization is summarized in Figure 6. It consists of a set of notations and the signatures of four (mathematical) functions, each of which corresponds directly to a C procedure defined in the FMI standard. For instance, `doStep` corresponds to `fmiDoStep`, `get` corresponds to `fmiGetXXX`, and so on. For simplicity, we do not include all parameters provided in the C functions and limit the signatures to only essential parameters. Before giving an overview of the functions, let us explain further the notation used in Figure 6. C denotes the set of all FMU instances that are coordinated by (the same) MA.[4] One such instance is an element $c \in C$. Given an instance c, S_c denotes the set of all possible states that c may be in, U_c denotes the set of input port variables of c, and Y_c denotes the set of output port variables of c. At this point we ignore typing issues, and assume a single universe of values for all variables, denoted $\mathbb{V}$.

The XML file in an FMU can (optionally) express the dependencies between input and output variables of an FMU. We model the set of all such input/output (I/O) dependencies of a given FMU instance c as a binary relation $D_c \subseteq U_c \times Y_c$. Therefore, $(u, y) \in D_c$ means that output y of c is *directly dependent* on input u of c. Directly dependent means that at a given instant in time, the value of u needs to be known to enable computation of y.

It is convenient at this point to also formalize the *connections* between FMU instances in a model. We do this using a *port mapping* function $P : U \to Y$, where U and Y are the sets of all input and output variables of all instances, respectively. P is a total function that maps every input variable to a unique output variable. This means that we assume that the model is *closed*; that is, every input is connected to

[3]In fact, the copying is done by the FMU itself which returns a pointer to the new copy of its state.

[4]Note that an FMU may be instantiated more than once in a co-simulation environment, meaning that different instances of the same FMU have separate internal state variables, but share the implementation of the FMI functions and solver.

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	$\mathbb{V}$
I/O dependency for instance c	$D_c \subseteq U_c \times Y_c$
Set of all input variables in a model	$U = \bigcup_{c \in C} U_c$
Set of all output variables in a model	$Y = \bigcup_{c \in C} Y_c$
Set of all I/O dependencies	$D = \bigcup_{c \in C} D_c$
Port mapping	$P : U \to Y$
Functions:	

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

Figure 6: Formalized model of FMI and connections between FMU instances.

some output. Note that two or more inputs may be connected to the same output. However an input is not allowed to be connected to more than one output. This is achieved by definition, since P is a function.

We now explain the functions in Figure 6. Function $\texttt{init}_c$ corresponds to `fmiInitializeSlave`. It initializes FMU instance c with given start time t, corresponding to the argument `tStart` of `fmiInitializeSlave`.[5] The function returns the initial state of the FMU instance.

Function $\texttt{set}_c$ corresponds to `fmiSetXXX`. Note that FMI has not one, but several such functions, including `fmiSetReal`, `fmiSetInteger`, and so on. Since we are ignoring data types, we formalize these using a single function that, for given FMU instance c, given current state $s \in S_c$, input variable $u \in U_c$, and value $v \in \mathbb{V}$, returns the new state of c obtained by setting u to v and keeping the rest of the state unchanged. Similarly, $\texttt{get}_c(s, y)$ returns the value of output variable y of FMU instance c at state s. Function $\texttt{get}_c$ corresponds to `fmiGetXXX`.

Note that both $\texttt{set}_c$ and $\texttt{get}_c$ are by definition free of *observable* side-effects. This means that, since $\texttt{set}_c$ and $\texttt{get}_c$ are (total) mathematical functions, given the same input arguments, they always return the same result.[6] This formalization does *not* imply that an FMU cannot use mechanisms such as value caching to improve efficiency of the implementation. This can still be done in an imperative implementation in C, provided these mechanisms do not alter the semantics. In particular, they must guarantee that the result of calling, say, $\texttt{get}_c$ multiple times without having called $\texttt{set}_c$ or $\texttt{doStep}_c$ in between, is deterministic; that is, the same value will always be returned.

Function $\texttt{doStep}_c$ takes as input the current state s of FMU instance c, and a non-negative real value $h \in \mathbb{R}_{\geq 0}$, corresponding to the `communicationStepSize` argument of `fmiDoStep`. Expression $\texttt{doStep}_c(s, h)$ returns a pair (s', h'), where s' models the new state of c at the end of the integration step, and h' models the amount by which c managed to advance time. $\texttt{doStep}_c$ must guarantee that $0 \leq h' \leq h$ (more about this in Section 4.2 below). Note that we allow $h = 0$, as well as $h' = 0$, enabling FMI to support a *superdense* model of time, which is widely acknowledged to be essential for proper modeling of hybrid systems [15, 12, 13, 2]. When $h' = h$, this indicates to a MA that the FMU is *accepting* the time step proposed by the MA. When

[5] Time is expressed here as a non-negative real value $t \in \mathbb{R}_{\geq 0}$. In FMI it is implemented as a floating-point number.

[6] Actually, the requirement on $\texttt{get}_c$ could be relaxed to allow modification of the FMU state in such a way that consecutive $\texttt{get}_c$ calls will return the same output values and the final FMU state does not depend on the order of $\texttt{get}_c$ calls.

$h' < h$, the FMU *rejects* the time step.[7] [8]

The formalization does not include explicit functions corresponding to `fmiGetFMUstate` and `fmiSetFMUstate`, which allow the MA to save and restore the state of an FMU instance. This is not a problem, as these functions can easily be modeled in our formalization. Saving a state simply means saving a particular element $s \in S_c$ of a particular instance c. Restoring the state simply means passing that saved s to subsequent calls of $\texttt{get}_c$, $\texttt{doStep}_c$, etc. This is precisely how the algorithm presented in Section 5 works.

4.2 FMU Contract

In this section we make explicit the utilization constraints of the FMI interface presented in Section 4.1. We point out that these constraints are not always explicit in the FMI standard. In fact, some of these constraints are probably not even implicitly assumed by the authors of the standard. The reason we introduce these constraints here is because they are crucial in proving the determinacy properties of the MAs presented in Section 5.

We call these utilization constraints the *FMU contract*. They consist of a set of *guarantees* that every FMU instance must provide to the caller of the functions of that instance, plus a set of *assumptions* that every FMU instance makes, that is, conditions that the caller must respect when calling these functions.

Part of the FMU contract is already given by the signature of the functions listed in Figure 6. For instance, the signature of $\texttt{doStep}_c$ implies that a caller is not allowed to call $\texttt{doStep}_c(s, h)$ with $h < 0$. A similar set of constraints includes sanity conditions, such as the fact that $\texttt{get}_c(s, y)$ can only be called when $y \in Y_c$ (i.e., variable y is indeed an output variable of c). We will not elaborate further on these and other similar constraints.

In addition to the above, we will assume that the following constraints are also part of the FMU contract:

(A0) If $\texttt{doStep}_c(s, h) = (s', h')$ then $0 \leq h' \leq h$.

(A1) If $\texttt{doStep}_c(s, h) = (s', h')$, then for any h'' where $0 \leq h'' \leq h'$, $\texttt{doStep}_c(s, h'') = (s'', h'')$ for some s''.

Assumption (A0) has been already stated above while describing the intuition of $\texttt{doStep}_c$ and is repeated here for completeness. Assumption (A1) states that if an FMU accepts a certain time step h (i.e., returns $h' = h$), or at least makes partial progress until $h' < h$, then it must accept any time step h'' smaller than or equal to h', provided the FMU is started from the same original state.

The following assumptions formalize the expected behavior of `get` and `set`. Let us first introduce some notation. Given state $s \in S_c$ of some instance $c \in C$, and given input variable $u \in U_c$, and value $v \in \mathbb{V}$, we denote by $s' = s[u := v]$ the state that is identical to s, except that s' assigns value v to variable u, whereas s may assign to u a different value.

(A2) Let $s' = \texttt{set}_c(s, u, v)$. Then $s' = s[u := v]$.

(A3) Let $v = \texttt{get}_c(s, y)$ and $v' = \texttt{get}_c(s', y)$. If $s' = s[u_1 := v_1, ..., u_k := v_k]$, and output variable y does not directly depend on any input $u_1, ..., u_k$, then $v' = v$.

The latter simply formalizes I/O dependencies.

[7] This reject is often caused by zero-crossing or another discrete change that the FMU detects. But zero-crossings or other overlooked events are sometimes detected only after an input is provided by `set`, e.g., when that input violates the validity range of the extrapolated input values in the previous `doStep`. This possibility can be handled in FMI 2.0 using the `fmiDiscard` callback which may be returned by `fmiSetXXX`. In this paper we assume `doStep` to be the only place where an FMU can reject the proposed time step. We plan to lift this assumption in the future.

[8] In this formalization we assume that FMUs can have both zero and variable communication step size. (The XML elements `canHandleVariableCommunicationStepSize` and `canHandleEvents` are both enabled.)

5 Determinate Execution

This section presents two MAs that are formally proven to have the desirable properties of termination (of an integration step) and determinacy (different runs of the algorithm produce the same result). First, we present a MA requiring all FMUs to support rollback. This is followed by a second MA that relaxes this constraint. The latter algorithm uses a proposed extension to the FMI standard, which enables the MA to query the FMU for expected future time events.

5.1 Algorithm Requiring Rollback

We first give a preprocessing algorithm that creates a list of variables, describing the order that the variables may be accessed. Recall that P denotes the mapping of input ports to output ports and D the global input-output dependency relation (see Figure 6). Let $\mathbb{X} = U \cup Y$, that is, $\mathbb{X}$ is the set of all input and output variables in the model.

Algorithm 1: *Order-Variables.*

Input: Port mapping P, global dependency relation D, and global set of variables $\mathbb{X}$.
Output: An ordered list $\bar{x}$ of variables, or error.

1. Let G be a directed graph, where the vertices are represented by port variables $\mathbb{X}$ and an edge $e \in \mathbb{X} \times \mathbb{X}$ is a variable dependency. The set of all edges E is then constructed by $E = D \cup \{(y, u) \mid u \in U \wedge P(u) = y\}$.
2. Perform a topological sort on G. If a cycle in G is found, terminate and return error. If no cycles are found, the resulting list of variables is $\bar{x}$.

In the following we assume that algorithm *Order-Variables* returns no error (i.e., G is acyclic). In that case, based on the variable list $\bar{x}$ returned by this algorithm, the MA executes, for each communication step, the algorithm *Master-Step* given below. We use the following notation: given variable $x \in \mathbb{X}$, c_x denotes the (unique) FMU instance $c \in C$ to which x belongs. That is, if $x \in U$ then c_x is the unique c such that $x \in U_c$, and if $x \in Y$ then c_x is the unique c such that $x \in Y_c$. We represent the states of all FMU instances as a mutable mapping m, mapping a FMU instance identifier $c \in C$ to a state valuation $s \in S$. Expression $m[c]$ is the current state valuation for FMU instance identifier c. We use statement $m[c] := s$ to denote that the state for c is updated to be s. Note that at each stage in the execution of the algorithm, the state mapping may formally be viewed as a function $m : C \to S$. Because the domain C is a set, the elements of m are unordered.

Algorithm 2: *Master-Step.*

Input: Set of instances C, ordered variable list $\bar{x}$, port mapping P, the maximal step size h_{max}, and a mutable state mapping m of size $|C|$.
Output: Updated state mapping m and the performed step size h.

1. Set values for all input variables:
   For each $u \in \bar{x}$ (in order) where $u \in U$ do
   (a) $y := P(u)$
   (b) $v := \texttt{get}_{c_y}(m[c_y], y)$
   (c) $m[c_u] := \texttt{set}_{c_u}(m[c_u], u, v)$
2. Save the states of all FMUs to enable rollback:
   $r := m$
3. Set communication step size to an initial default value:
   $h := h_{max}$
4. Find h acceptable by all FMUs:
   For each $c \in C$ do

(a) $(s', h') := \texttt{doStep}_c(m[c], h_{max})$
(b) $h := min(h, h')$
(c) $m[c] := s'$

5. Assert $0 \leq h \leq h_{max}$ // follows from Assumption (A0)
6. If $h < h_{max}$ then // roll back and perform step h
   For each $c \in C$ do
   (a) $(s', h') := \texttt{doStep}(r[c], h)$
   (b) Assert $h' = h$ // follows from Assumption (A1)
   (c) $m[c] := s'$
7. Return m and h.

Intuitively, in Step 4 the algorithm "sweeps" over all FMUs, attempting to perform step h_{max} on each of them. At the same time the algorithm records the smallest actually achieved step, h. At the end of Step 4, if $h = h_{max}$, then h_{max} was accepted by all FMUs, and the step is complete. Otherwise, by Assumption (A0), it must be that $h < h_{max}$. In that case the algorithm makes a second pass, performing step h on all FMUs. Thanks to Assumption (A1), this is bound to be accepted by all FMUs.

Clearly, both Algorithms 1 (Order-Variables) and 2 (Master-Step) terminate, as our models (set of FMU instances, ports, connections, etc.) are finite.

A useful notion, used in particular to prove Theorem 1 that follows, is the notion of *acceptable* time step. Let $c \in C$, $s \in S_c$ and $h \in \mathbb{R}_{\geq 0}$. We say that h is acceptable by c at state s if $\texttt{doStep}_c(s, h) = (s', h)$ for some s'. If m is a global state mapping, then we say that h is acceptable at m if for all $c \in C$, h is acceptable by c at $m[c]$.

Theorem 1 (Determinacy)**.** *Algorithm 2 (Master-Step) is determinate in the sense that, for given inputs C, P, h_{max}, and m, the returned updated output state mapping m and h are the same no matter what the ordered list $\bar{x}$ produced by Algorithm 1 (Order-Variables) is and no matter which order the instances $c \in C$ are selected in Algorithm 2.*

Proof. Denote m_1 to be the state mapping at start of Step 1 and m_2 to be the state mapping at the start of Step 2. We first show that m_2 is uniquely defined (Lemma 1).

Lemma 1. *State mapping m_2 is uniquely defined, independently of what the ordered list $\bar{x}$ produced by algorithm* Order-Variables *is.*

of Lemma 1. Let $u_1 \dots u_N$ be the sequence of topologically sorted input variables and $n \in \mathbb{N}$ be the variable index. We prove by strong mathematical induction over n that $m[c_{u_n}]$ in step 1c is uniquely defined for $n \geq 1$. (i) Basic step: Input variable u_1 is defined by output variable $y = P(u_1)$. Because the topological sort is assumed to succeed, y must be a start node in graph G (step 1 in Algorithm 1). Consequently, $\texttt{get}(m[c_y], y)$ can only have one value and therefore the value assigned for u_1 is uniquely defined. (ii) Inductive step: Assume that values in m for variables $u_1 \dots u_n$ are uniquely defined. We now show why the value for u_{n+1} is uniquely defined. Let $y = P(u_{n+1})$ be the output variable that holds the value that variable u_{n+1} will be assigned to in Step 1(c). We will show that the value assigned to u_{n+1} in Step 1(c) is unique. In addition, from assumption (A2), `set` only affects the value of the variable being assigned, therefore, the values of previously assigned variables do not change. To show that the value assigned to u_{n+1} is unique we consider two cases. Case 1: It does not exist an x such that $(x, y) \in D$, that is, y is not directly dependent on any input variables. In this case, according to assumption (A3), $\texttt{get}(m[c_y], y)$ is not affected by setting input values to instance c_y. The value for output variable y is unique and consequently is the value for u_{n+1} unique in Step 1(c). Case 2: There exist a set of directly dependent input variables $X = \{x \mid (x, y) \in D\}$. By the property of topological sort, we have that every $x \in X$ must come before both y and u_{n+1} in the sorted sequence of variables. Because each $x \in X$ corresponds to a u_k where $k \leq n$, we have by the induction hypothesis that the values for $x \in X$ are uniquely defined. Since the values for x is set before the value

$\texttt{get}(m[c_y], y)$ is retrieved, the value for y in Step 1(b) is unique and therefore also the value for u_{n+1} in Step 1(c). Finally, we can conclude that the state mapping m_2 is uniquely defined because all elements set in Step 1(c) are uniquely defined. $\square$

Since r is simply a copy of m_2, Lemma 1 also proves that the state mapping r computed at the end of Step 2 is unique, independently of what the ordered list $\bar{x}$ produced by algorithm *Order-Variables* is.

We next prove that the h computed at the end of Step 4 is uniquely defined, independently of the order in which FMU instances are chosen in the loop iteration of Step 4. Indeed, it can be easy to see that this h is equal to

$$\min\{h' \mid c \in C, (_, h') = \texttt{doStep}_c(m[c], h_{max})\}.$$

and therefore uniquely defined.

The assertion of Step 5 holds since, by Assumption (A0), $\texttt{doStep}_c(m[c], h_{max})$ always returns $0 \leq h \leq h_{max}$. At this point, if $h = h_{max}$, then the algorithm returns state mapping m_4 and h_{max}, where m_4 is equal to the current state mapping m. Then, m_4 is uniquely defined by the equality $(m_4[c], h_{max}) = \texttt{doStep}_c(m_2[c], h_{max})$, for all $c \in C$.

Otherwise, $h < h_{max}$. In this case, Step 6 is executed, which corresponds to rolling back to saved state mapping r, and performing the computed step h. By Assumption (A1), h is guaranteed to be acceptable by all FMUs. Therefore, the assertion of Step 6(b) is satisfied for all $c \in C$, and at the end of Step 6 the algorithm returns state mapping m_6 and h, where m_6 is uniquely defined by the equality $(m_6[c], h) = \texttt{doStep}_c(r[c], h)$, for all $c \in C$.

This completes the proof of Theorem 1. $\square$

As can be seen from the proof of Theorem 1 , Algorithm 2 is not only determinate, but is also *correct* in the following sense. If m_2 is the state mapping after executing Step 1, and m, h are the values returned by Algorithm 2, then for all $c \in C$, $\texttt{doStep}_c(m_2[c], h) = (m[c], h)$. This means that Algorithm 2 returns an acceptable time step h and the state mapping which would result as if all FMUs performed this step just once.

It is also desirable to show that a master algorithm achieves *maximal progress*, i.e., it achieves the *maximal* acceptable step h. We would thus like to state the following.

Theorem 2 (Maximal progress)**.** *Let m_2 be the state mapping after executing Step 1 of Algorithm 2 and let h be the step returned by Algorithm 2. There is no h' such that $h < h' \leq h_{max}$ and h' is acceptable at m_2.*

We can prove Theorem 2 provided all FMUs satisfy an additional assumption:

(P1) If $\texttt{doStep}_c(s, h) = (s', h')$ and $h' < h$, then for all $h'' > h'$, $\texttt{doStep}_c(s, h'') = (s'', h')$ for some s''.

Assumption (P1) says that every FMU makes maximal progress at the individual level. Note that (P1) is not implied by (A1). For instance, let $\texttt{doStep}_c(s, h) = (s, h')$ where

$$h' = \begin{cases} h & \text{if } h \leq 1 \\ \frac{1}{2} & \text{otherwise.} \end{cases}$$

Then c violates (P1) whereas it satisfies (A1). This example also shows why (P1) is necessary for maximal progress. If (P1) does not hold, then we can easily construct a counter-example where Algorithm 2 does not ensure maximal progress. Consider a model with a single FMU, c, and initial state s, as above. Then, starting with step size $h_{max} = 2$, Algorithm 2 will make progress only up to $\frac{1}{2}$, whereas progress up to 1 can also be made.

Proof of Theorem 2. Suppose the contrary, i.e., there is h' such that $h < h' \leq h_{max}$ and h' is acceptable at m_2. This means that for all $c \in C$, h' is acceptable by c at $m_2[c]$, i.e., for all $c \in C$, there exists s, such that $\texttt{doStep}_c(m_2[c], h') = (s, h')$. Now, since h is returned by Algorithm 2, and $h < h_{max}$, there must exist some FMU c^* and some state s^* of c^* such that $\texttt{doStep}_{c^*}(m_2[c^*], h_{max}) = (s^*, h)$. It can now be seen that this c^* contradicts Assumption (P1).

This completes the proof of Theorem 2. $\square$

We end this section with a remark. Algorithm 2 may appear wasteful in the sense that Step 4 continues attempting to perform step h_{max} on the rest of the FMUs even after it encounters an FMU which rejects h_{max}. An alternative would be to call `doStep` in Step 4(a) with h instead of h_{max}. In addition, the algorithm can keep track of which FMUs have already been executed with the right h, so that it does not re-run them. This modification can be shown to terminate (due to Assumption (A1)). It can also be shown to be determinate, provided Assumption (P1) holds. Note that Assumption (P1) is not needed for Theorem 1.

5.2 Predictable Step Sizes

The algorithm given above requires all FMUs to implement rollback. In many cases, this is impractical, particularly when an FMU wraps legacy code or serves as a wrapper for a simulation tool. Fortunately, with a small addition to the FMI standard, such FMUs can be handled in certain cases. Specifically, we propose the addition of a procedure

```
fmiStatus fmiGetMaxStepSize(
   fmiComponent c,
   fmiReal *maxStepSize);
```

where the argument returns an upper bound on the step size that the FMU can accept (or infinity if there is none). This bound could be zero to indicate the need for a zero-step-size step. We use the function

$$\texttt{getMaxStepSize}_c : S_c \to \mathbb{R}_{\geq 0} \cup \{\infty\} \tag{1}$$

to model the `fmiGetMaxStepSize` procedure. Let C_P be the set of FMU instances that implement this function. We require of these instances that

(A4) If $c \in C_P$ and $s \in S_c$ and $\texttt{getMaxStepSize}_c(s) = h$ then for all h' where $0 \leq h' \leq h$, $\texttt{doStep}_c(s, h') = (s', h')$ for some s'.

This means that an instance in C_p will accept any time step smaller than or equal to the time step returned by `getMaxStepSize`. Whether an FMU is in C_p should be indicated as a capability in the FMU XML file. Let C_R be the set of FMU instances with rollback capability, i.e., every $c \in C_R$ supports setting and getting states. Furthermore, let C_L be the set of FMU instances in a model that are not in C_R and not in C_P. We will call these FMUs "legacy FMUs". Then, we can give a MA that is determinate under the following assumption.

(A5) (a) $|C_L| \leq 1$.
(b) $C_L \cup C_R \cup C_P = C$.
(c) $C_L \cap C_R = \emptyset$ and $C_R \cap C_P = \emptyset$ and $C_P \cap C_L = \emptyset$.

That is, a model that composes FMUs has at most one legacy FMU instance, and the remaining instances all either provide predictable step sizes or support rollback.[9]

[9] If an FMU supports both predictable step sizes and rollback, our algorithm only uses its predictable step sizes capability, so we put it in set C_P and not in C_R.

Algorithm 3: *Master-Step With Predictable Step Sizes.*

Input: Set of instances C, ordered variable list $\bar{x}$, port mapping P, the maximal step size h_{max}, and a mutable state mapping m of size $|C|$.
Output: Updated state mapping m and the performed step size h.

1. Set values for all input variables:
   For each $u \in \bar{x}$ (in order) where $u \in U$ do
   (a) $y := P(u)$
   (b) $v := \texttt{get}_{c_y}(m[c_y], y)$
   (c) $m[c_u] := \texttt{set}_{c_u}(m[c_u], u, v)$
2. Find the minimal predictable communication size:
   $h := min(\{\texttt{getMaxStepSize}_c(m[c]) \mid c \in C_P\} \cup \{h_{max}\})$
3. Save the states for all instances that can perform rollback:
   (a) For each $c \in C_R$ do $r[c] := m[c]$
   (b) $doStepOnLegacy := \texttt{true}$
   (c) Goto step 5.
4. Restore states for rollback instances.
   For each $c \in C_R$ do $m[c] := r[c]$
5. Perform `doStep` on all instances with rollback:
   $h_{min} := h$
   For each $c \in C_R$ do
   (a) $(s', h') := \texttt{doStep}_c(m[c], h)$
   (b) $h_{min} := min(h', h_{min})$
   (c) $m[c] := s'$
6. If $h_{min} < h$ then $h := h_{min}$ and goto step 4.
7. Perform `doStep` on the legacy FMU (if it exists)
   If $c \in C_L$ and $doStepOnLegacy$ then
   (a) $(s', h') := \texttt{doStep}_c(m[c], h)$
   (b) $m[c] := s'$
   (c) $doStepOnLegacy := \texttt{false}$
   (d) If $h' < h$ then $h := h'$ and goto step 4.
8. Perform `doStep` on all FMUs with predictable step size:
   For each $c \in C_P$ do
   (a) $(s', h') := \texttt{doStep}_c(m[c], h)$
   (b) Assert $h' = h$ // follows from Assumption (A4)
   (c) $m[c] := s'$
9. Return m and h.

With such a MA, a collection of FMUs can always be executed deterministically if there is at most one "legacy FMU". Note that Algorithm 2 can be extended easily to also support at most one FMU that does not implement rollback.

Theorem 3 (Termination)**.** *Algorithm 3 (Master-Step With Predictable Step Sizes) terminates.*

Proof. Step 1 terminates because $\bar{x}$ is a finite list. Each of the steps 2, 3, 4, and 5 terminate because C_P and C_R are finite sets. The algorithm contains two goto statements: one at step 6 and one at step 7. Each of these goto statements can be executed at most once, which will show next. The first time step 5 is executed,

step 6 may go to step 4 if not all `doStep` succeed. They will all, however, succeed the second time step 5 is executed because of Assumption A1. Note that `doStep` is performed on all instances with the same h and that the smallest step size h_{min} is used the second time step 5 is executed (if any). If $C_L = \emptyset$ or $h' = h$ in step 7a, step 7 terminates and goes to step 8. If C_L is not empty and $h' < h$, the algorithm goes to step 4. In this case, step 6 cannot loop back to step 4, because Assumption A1 assures that all calls to `doStep` in Step 5 succeed (h updated in step 7 is necessarily smaller than the h used in previous round of step 5). Also, because $\mathit{doStepOnLegacy} = \texttt{false}$ the second time step 7 is executed, it will not loop back to step 4. Finally, step 8 terminates because C_P is finite.

This completes the proof of Theorem 3. □

Theorem 4 (Determinacy)**.** *Algorithm 3 (Master-Step With Predictable Step Sizes) is determinate in the sense that, for given inputs C, P, h_{max}, and m, the updated output state mapping m and the output step size h are the same no matter what the ordered list $\bar{x}$ produced by Algorithm 1 (Order-Variables) is and no matter the order of how $c \in C$ are selected in Algorithm 3.*

Proof. Step 1 of Algorithm 3 is identical to step 1 in Algorithm 2; determinacy for step 1 is therefore proven analogously to the proof of Algorithm 2 using Lemma 1. The step size h in step 2 and the store mapping r in step 3 are clearly uniquely defined. Variable h is only updated in step 6 and 7b, and only one time in each step. Consequently, h does not depend on the order how $c \in C$ is selected an therefore uniquely defined at the end of the algorithm.

We now consider how subsets of the mapping of m are uniquely updated. By Assumption A5 we see that the subsets C_L, C_P, and C_R are distinct. We can also observe that in each of the steps 4, 5, 7, and 8, only a distinct subset of m is updated, and in each step each element is updated only once. Finally, compared to m_2 (end of step 2), each element is updated at most once. For instances $c \in C_R$, m can be updated in step 5, but this step is always preceded by either the original m_2 state mapping or a rollback of states (step 4) to the states in m_2. An instance $c \in C_L$, is only updated once because of the guarding boolean variable *doStepOnLegacy*. Finally, for instances $c \in C_P$, updates in m is only performed once in step 8. By Assumption A4, we see that the assertion at 8b holds.

Consequently, we conclude that at the end of Step 9 the algorithm returns state mapping m_9 and h_9, where m_9 is uniquely defined by the equality $\texttt{doStep}_c(m_2[c], h_9) = (m_9[c], h_9)$, for all $c \in C$.

This completes the proof of Theorem 4. □

Analogously to the discussion of correctness for Algorithm 2, we can see in Theorem 4 that Algorithm 3 is not only determinate, but is also *correct* in the following sense: if m_2 is the state mapping after executing Step 2, and m_9, h_9 are the values returned by Algorithm 3, then for all $c \in C$, $\texttt{doStep}_c(m_2[c], h_9) = (m_9[c], h_9)$.

We next prove maximal progress for Algorithm 3. Again we assume (P1) on all FMUs. In addition, we need the following assumption for FMUs with predictable step sizes:

(P2) If $c \in C_P$ and $s \in S_c$ and $\texttt{getMaxStepSize}_c(s) = h$ then for all $h' > h$, $\texttt{doStep}_c(s, h') = (s', h)$ for some s'.

Theorem 5. *Let m_2 be the state mapping after executing Step 1 of Algorithm 3 and let h be the step returned by Algorithm 3. There is no h' such that $h < h' \leq h_{max}$ and h' is acceptable at m_2.*

Proof. Suppose the contrary, i.e., there is h' such that $h < h' \leq h_{max}$ and h' is acceptable at m_2. This means that for all $c \in C$, h' is acceptable by c at $m_2[c]$, i.e., for all $c \in C$, there exists s, such that $\texttt{doStep}_c(m_2[c], h') = (s, h')$.

Let h_P be the h computed at Step 2 of Algorithm 3, i.e.,

$$h_P = min(\{\texttt{getMaxStepSize}_c(m[c]) \mid c \in C_P\} \cup \{h_{max}\}).$$

We claim that $h' \leq h_P$. Otherwise, there is some $c \in C_P$ such that $\texttt{doStep}_c(m_2[c], h') = (s, h')$, yet $\texttt{getMaxStepSize}_c(m[c]) < h'$, which violates Assumption (P2).

Now, since h is returned by Algorithm 3, there must exist some FMU $c^* \in C$, some state s^* of c^*, and some h^*, such that $h < h^* \leq h_P$, and $\texttt{doStep}_{c^*}(m_2[c^*], h^*) = (s^*, h)$. It can now be seen that this c^* contradicts Assumption (P1).

This completes the proof of Theorem 5. □

6 Related Work

This paper focuses on co-simulation rather than model exchange because it more loosely couples simulation tools. Co-simulation enables the principle of *hierarchical heterogeneity*, pioneered in the Ptolemy Project [9], where multiple models of computation (MoCs) are combined hierarchically. Since an FMU for co-simulation includes its own simulation engine, there is no requirement that its simulation engine match precisely the semantics of the host simulator. It only has to be capable of providing the semantics of the FMI interface (a semantics that this paper clarifies). In model exchange, the host simulator semantics prevail, and any differences in semantics intended by the author of the FMU will be subjugated. In effect, this means that the author of the FMU and the author of the host simulator have to agree on the semantics. Such agreement has been shown to rarely exist even within communities working on closely related modeling techniques, as evidenced, for example, by the failure of the Hybrid Systems Interchange Format, HSIF [22, 23, 26]. Such agreement has also proven impossible to achieve in communities with broader interests, such as the UML community, where a plethora of semantic variants exist for nearly every UML notation. Co-simulation enables interoperation even in the absence of agreement about semantics.

Despite these benefits, state-of-the-art co-simulation is still quite limited. To quote from the FMI standard [17]:

> *"In contrast to classical (mono-disciplinary) simulation techniques in system dynamics, state-of-the-art master algorithms in co-simulation are even today based on constant communication step sizes and do not provide any automatic error control. Constant communication step sizes may restrict strongly the efficiency of co-simulation algorithms if the solution behavior changes considerably during time integration."*

A constant time step size co-simulation interface has for example been implemented in Ptolemy II as part of the Building Controls Virtual Test Bed [28]. The TISC [11] co-simulation environment uses a variable synchronization step size. In TISC, a simulation module can propose the length of the next time step.

An API that has been available for some time is Simulink's S-Functions. S-Functions are proprietary and not standardized, therefore not a true alternative to FMI which aims to be an open standard. S-Functions are also quite limited. For instance, being able to discard a time step, which is essential in a number of numerical simulation methods, is not possible with S-Functions [3].

S-Functions use a *split-phase* API with two distinct functions per block, as in Moore/Mealy state machines; an *Output* function computing the output of the block and a separate *Update* function that updates the state of the block. A split-phase API is also used in Ptolemy II [9], where Output and Update are called *Fire* and *Postfire*. The FMI standard, as interpreted in this paper, supports indirectly split-phase by using `fmiGetXXX` as the output function and `fmiDoStep` as the state update function. The split-phase API is interesting as it allows to decouple the computation of outputs for updating the states. This in turn enables delaying to *commit* a step, which can be useful in handling models where rollback is required, as in several examples presented above. A split-phase API can also be useful in dealing with algebraic loops or other model characteristics that require some type of fixpoint calculation. For instance, split-phase is essential in handling the semantics of synchronous block diagrams in a general way [8]. The split-phase API can be generalized to include more than one Output function, which is useful in dealing with hierarchical models [14]. Lublinerman *et al.* [14] also deal with the problem of *modular code generation*, related to the problem of automatically synthesizing FMUs from models. This synthesis problem is, however, beyond the scope of this paper.

The abstract semantics of Ptolemy include, in addition to the Fire and Postfire functions, the *FireAt* function which enables handling timed actors (both continuous-time and discrete-event). FireAt is different

in nature than Fire and Postfire, in the sense that while the latter two are implemented by the slave and called by the master, FireAt is implemented by the master and called by the slave.[10] Tripakis *et al.* [27] formalize the semantics of Ptolemy using a *deadline* function, from which the `getMaxStepSize` function has been inspired.

The above can be seen as attempts to come up with the "right" API for modeling heterogeneous systems, and in particular, continuous as well as (timed) discrete-event systems. Very relevant to this problem is the work reported by Denckla and Mosterman [7] and Mosterman *et al.* [19], who present stream- and state-based functional interfaces for a Simulink-type of language. Also, Sander and Jantsch [25] present the ForSyDe modeling framework, which provides a set of libraries for capturing heterogeneous MoCs based on the functional programming language Haskell. Broman and Siek [5] address the heterogeneous modeling problem using an embedded domain specific language (EDSL) approach.

Other approaches to tool integration apart include *coordination languages* [21] and *tool buses* [6, 10, 20, 24]. These focus on managing workflows, general-purpose distributed computation, and data exchange between concurrent tasks.

Bastian *et al.* [1] propose a MA implementation for FMI co-simulation that is designed to be platform independent. Their MA uses fixed communication step size and it assumes that FMUs are not dependent on the current output of other FMUs. By contrast, our MAs have variable communication step size and supports direct dependencies.

7 Conclusions

FMI shows enormous promise for enabling interoperability of simulation tools for CPS. We have identified some subtleties presented by the design of the API in the standard, and have offered constraints on the design of FMUs and MAs such that determinate execution of many models is ensured. We have defined a class of MAs that correctly handles models containing a mix of FMUs that support rollback, FMUs that do not support rollback but implement a proposed small extension to FMI for predictable step sizes, and at most one FMU that supports neither. Even in the case where rollback is needed, only 1-step rollback is enough for determinacy, as our master algorithms show. This is essential for efficient implementation, as multi-step rollback is very expensive.

As of this writing, the FMI 2.0 standard is still in flux. We base our analysis on the Beta 4.0 version, but the final version may introduce additional subtleties and/or problems.

An interesting direction of future work is the composition of both co-simulation and model exchange FMUs. Such heterogeneous composition is both practically important and requires new kinds of MAs.

References

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

[2] A. Benveniste, T. Bourke, B. Caillaud, and M. Pouzet. The fundamentals of hybrid systems modelers. *J. of Computer and System Sciences*, 78(3):877–910, 2012.

[3] T. Blochwitz, M. Otter, et al. The Functional Mockup Interface for Tool independent Exchange of Simulation Models. In *Proceedings of the 8th International Modelica Conference*, 2011.

[4] 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.

[5] D. Broman and J. 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.

[10] In Ptolemy, slaves (e.g. FMUs) are called *actors* and a master algorithm is called a *director*.

[6] S. Cruanes, G. Hamon, S. Owre, and N. Shankar. Tool Integration with the Evidential Tool Bus. In *VMCAI 2013*, volume 7737 of *Lecture Notes in Computer Science*, pages 275–294. Springer, 2013.

[7] B. Denckla and P. Mosterman. Stream- and state-based semantics of hierarchy in block diagrams. In *17th IFAC World Congress*, pages 7955–7960, 2008.

[8] S. Edwards and E. Lee. The semantics and execution of a synchronous block-diagram language. *Science of Computer Programming*, 48:21–42(22), July 2003.

[9] J. Eker, J. Janneck, E. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity – the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, Jan. 2003.

[10] P. Klint. The ToolBus: A Service-Oriented Architecture for Language-Processing Tools. *ERCIM News*, (70), 2007.

[11] R. Kossel, W. Tegethoff, M. Bodmann, and N. Lemke. Simulation of complex systems using modelica and tool coupling. In *5th Modelica Conference*, volume 2, pages 485–490, Sept. 2006.

[12] E. A. Lee and H. Zheng. Operational semantics of hybrid systems. In *Hybrid Systems: Computation and Control (HSCC)*, volume LNCS 3414, pages 25–53. Springer, 2005.

[13] E. A. Lee and H. Zheng. Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems. In *EMSOFT*. ACM, 2007.

[14] R. Lublinerman, C. Szegedy, and S. Tripakis. Modular code generation from synchronous block diagrams. In *POPL'09*, pages 78–89. ACM, 2009.

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

[16] MODELISAR Consortium and Modelica Association. Functional Mock-up Interface for Model Exchange and Co-Simulation – Version 2.0 Beta 4, August 10, 2012. Retrieved from `https://www.fmi-standard.org`.

[17] MODELISAR Consortium and Modelica Association. Functional Mock-up Interface for Co-Simulation, October 12, 2010. Version 1.0, Retrieved from `https://www.fmi-standard.org`.

[18] MODELISAR Consortium and Modelica Association. Functional Mock-up Interface for Model Exchange, October 12, 2010. Version 1.0, Retrieved from `https://www.fmi-standard.org`.

[19] P. J. Mosterman, J. Zander, G. Hamon, and B. Denckla. A computational model of time for stiff hybrid systems applied to control synthesis. *Control Engineering Practice*, 20(1):2 – 13, 2012.

[20] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen. The information bus: an architecture for extensible distributed systems. *SIGOPS Oper. Syst. Rev.*, 27(5):58–68, Dec. 1993.

[21] G. A. Papadopoulos and F. Arbab. Coordination models and languages. *Advances in Computers*, 46:329–400, 1998.

[22] A. Pinto, L. P. Carloni, R. Passerone, and A. Sangiovanni-Vincentelli. Interchange format for hybrid systems: abstract semantics. In *HSCC'06*, pages 491–506. Springer, 2006.

[23] A. Pinto, A. L. Sangiovanni-Vincentelli, L. P. Carloni, and R. Passerone. Interchange formats for hybrid systems: review and proposal. In *HSCC'05*, pages 526–541. Springer, 2005.

[24] J. Rushby. An evidential tool bus. In *7th international conference on Formal Methods and Software Engineering*, ICFEM'05. Springer, 2005.

[25] I. Sander and A. Jantsch. System modeling and transformational design refinement in ForSyDe. *IEEE Transactions on Computer-Aided Design of Circuits and Systems*, 23(1):17–32, 2004.

[26] O. Sokolsky, I. Lee, and R. Alur. HSIF Semantics, 2004. Available from `ftp://ftp.cis.upenn.edu/pub/rtg/public_html/papers/MS-CIS-04-05.pdf`.

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

[28] M. Wetter. Co-simulation of building energy and control systems with the building controls virtual test bed. *Journal of Building Performance Simulation*, 4:185–203, 2011.