Proceedings Paper:

Cavalcanti, A. L. C. orcid.org/0000-0002-0831-1976, Woodcock, J. C. P. orcid.org/0000-0001-7955-2702 and Amalio, N. orcid.org/0000-0001-8751-5039 (2016) Behavioural Models for FMI Co-simulations. In: Sampaio, A. C. A. and Wang, F., (eds.) International Colloquium on Theoretical Aspects of Computing. Lecture Notes in Computer Science . Springer , pp. 255-273.

https://doi.org/10.1007/978-3-319-46750-4_15

Behavioural Models for FMI Co-simulations

Ana Cavalcanti, Jim Woodcock, and Nuno Amálio

University of York

Abstract. Simulation is a favoured technique for analysis of cyber-physical systems. With their increase in complexity, co-simulation, which involves the coordinated use of heterogeneous models and tools, has become widespread. An industry standard, FMI, has been developed to support orchestration; we provide the first behavioural semantics of FMI. We use the state-rich process algebra, *Circus*, to present our modelling approach, and indicate how models can be automatically generated from a description of the individual simulations and their dependencies. We illustrate the work using three algorithms for orchestration. A stateless version of the models can be verified using model checking via translation to CSP. With that, we can prove important properties of these algorithms, like termination and determinism, for example. We also show that the example provided in the FMI standard is not a valid algorithm.

Keywords: verification, modelling, *Circus*, CSP

1 Introduction

The Functional Mock-up Interface (FMI) [12] is an industry standard for co-simulation: collaborative simulation of separately developed models. It has been applied across a variety of domains, including automotive, energy, aerospace, and real-time systems integration; dozens of tools support the standard.

An FMI co-simulation [4] is organised around black-box slave FMUs (Functional Mockup Units): effectively, wrappings of models that are interconnected through their inputs and outputs. FMUs are passive entities whose simulation is triggered and orchestrated by a master algorithm. A simulation is divided into steps that serve as synchronisation and data exchange points; between these steps, the FMUs are simulated independently. The master algorithm communicates with the FMUs via a number of functions that compose the FMI API.

Here, we present the first behavioural formal semantics for FMI-based co-simulations. We use *Circus* [21], a state-rich process algebra that combines Z [26] for data modelling and CSP [23] for behavioural specification. We characterise formally master algorithms and FMUs that make appropriate use of the FMI API. These abstract models of a co-simulation can be automatically generated from the number of FMUs, their inputs and outputs and dependencies.

The general models can be used to verify specific master algorithms and the adequacy of simulation models for FMUs. We have verified a classic algorithm from the FMI standard for Simulink [19], and a more robust algorithm that caters for FMU failures [4]. This revealed that the example in the standard implicitly assumes that FMUs do not raise fatal errors; it is not a valid algorithm.

Circus models, with abstracted state, can be translated to CSP and verified using the FDR3 model checker [16]. We prove important properties discussed in the FMI literature, like termination and determinism using the FDR3 model checker. Richer models can be verified using a *Circus* theorem prover [14]. Given a choice of master algorithm and formal models of the FMUs, our work can also be used to prove properties of an overall system described by the separate simulations. *Circus* can currently cater only for discrete-time models. On the other hand, a continuous time extension of *Circus* that can be used to give semantics to continuous-systems simulations [13] is under development.

Broman [4] has presented the most influential formalisation of FMI to date: a state-based model of the three main API functions that set and get FMU variables and trigger a simulation step with two master algorithms and a proof of core properties. Our model of a co-simulation also has its interface defined by the interactions corresponding to the simulation steps and the exchange of data associated with them. Our behavioural model covers a large portion of the FMI API, defining valid patterns for its usage and error treatment.

Sects 2 and 3 describe FMI for co-simulation and *Circus*. Sect. 4 describes the *Circus* semantics of FMI. The specification and verification of master algorithms and co-simulations is discussed in Sect. 5. Sect. 6 presents our conclusions.

2 FMI

Modelling and simulating cyber-physical systems (CPSs) [10] involves different engineering fields: a global system with components tackled by domain engineers using specialised tools. Co-simulation [18] involves tool interoperability for modelling and simulating heterogeneous components. FMI avoids the need for tool-specific integration, by exchanging dynamic models, co-simulating heterogeneous models, and protecting intellectual property. We deal with co-simulation, but we can also reason about simulations with model exchange.

A master algorithm orchestrates a collection of FMUs that may be standalone, containing runnable code, or be coupled, in which case it contains a wrapper to a simulation tool. Like FMI, our model is agnostic to the particular realisation of an FMU, and does not cover any communication infrastructure that may be in place to support distributed co-simulation. We assume that communication between the master algorithm and the various FMUs is reliable.

When the co-simulation is started, the models of the FMUs are solved independently between two discrete communication points defined by a step. For that, the master algorithm reads the outputs of the FMUs, sets their inputs, and then waits for all FMUs to simulate up to the defined communication point, before advancing the simulation time. Master algorithms differ in their approach to handling the definition of the step sizes and any simulation errors.

Although the FMI standard does not specify any particular master algorithms, or the technology for development of FMUs, it specifies an API that can be used to orchestrate the various simulations. Restrictions on the use of the API functions specify, indirectly and informally, how a master algorithm can be

$\mathbf{channel} : \mathit{setT} : \mathit{TIME};\ \mathit{updateSS} : \mathit{NZTIME};\ \mathit{step} : \mathit{TIME} \times \mathit{NZTIME};\ \mathit{end}$
$\mathbf{process}\ \ \mathit{Timer} \mathrel{\widehat=} \mathit{ct}, \mathit{hc}, \mathit{tN} : \mathit{TIME} \bullet \mathbf{begin}$
$\mathbf{state}\ \mathit{State} == [\mathit{currentTime}, \mathit{stepSize} : \mathit{TIME}]$
$\mathit{Step} =$
$\quad \mathit{setT}?t : t \leq \mathit{tN} \longrightarrow \mathit{currentTime} := t;\ \mathit{Step}$
$\quad \Box\ \mathit{updateSS}?\mathit{ss} \longrightarrow \mathit{stepSize} := \mathit{ss};\ \mathit{Step}$
$\quad \Box\ \mathit{step}!\mathit{currentTime}!\mathit{stepSize} \longrightarrow \mathit{currentTime} := \mathit{currentTime} + \mathit{stepSize};\ \mathit{Step}$
$\quad \Box\ \mathit{currentTime} = \mathit{tN}\ \&\ \mathit{end} \longrightarrow \mathbf{Stop}$
$\bullet\ \mathit{currentTime}, \mathit{stepSize} := \mathit{ct}, \mathit{hc};\ \mathit{Step}$
$\mathbf{end}$

Fig. 1. *Circus* specification of a *Timer* process

defined and how an FMU may respond. Our model captures a significant subset of the FMI API, and defines formally validity for algorithms and FMUs.

3 *Circus*

The main construct of *Circus* is a process, used to specify a system and its components. Processes communicate with each other via channels. Communications are instantaneous and synchronous events. A process can have a state, defined using a Z schema, and a behaviour, defined using an action.

To illustrate *Circus*, Fig. 1 presents the model of a *Timer* from a valid master algorithm. *Timer* takes as parameters the current time ct, the step size hc, and the end time tN of the simulation. Although it is possible to set up experiments without an end time, we restrict ourselves to experiments that are time bounded.

Timer's state contains two components: *currentTime* and *stepSize*. Its behaviour is defined by the action at the end. After initialising *currentTime* and *stepSize* using ct and hc, it calls the local action *Step*. It takes inputs on channels *setT* and *updateSS* to update the current time and step size. The channel declarations define the type of the values that can be communicated through them: *TIME* is the set of natural numbers, and *NZTIME* excludes 0. Step sizes cannot be 0. It uses a channel *step* to output the current time and step size. After a communication on *step*, the current time is advanced to the next simulation step; at the end of the experiment ($\mathit{currentTime} = \mathit{tN}$), it synchronises on *end*.

The action *Step* offers communications on the above channels in external choice ($\Box$). The time t input through *setT* cannot exceed the end time tN of the simulation. The offer of synchronisation on *end* is guarded by $\mathit{currentTime} = \mathit{tN}$ and only becomes available if this condition holds. After the event *end*, the timer deadlocks: behaves like the action **Stop**.

Processes can also be defined by combination of other processes. For example, the specification of the process *TimedInteractions* below combines three processes *Timer*, *endSimulation* and *Interaction*.

$$\mathit{TimedInteractions} \mathrel{\widehat=} t0, \mathit{tN} : \mathit{TIME} \bullet \left(\begin{array}{l} (\mathit{Timer}(t0, 1, \mathit{tN})\ \Delta\ \mathit{endSimulation}) \\ \quad [\![\{\!|\ \mathit{step}, \mathit{end}, \mathit{setT}, \mathit{updateSS}, \mathit{endsimulation}\ |\!\}]\!] \\ \mathit{Interaction} \end{array}\right) \setminus \{\!|\ \mathit{step}, \mathit{end}, \mathit{setT}, \mathit{updateSS}\ |\!\}$$

`fmi2Get`	$FMI2COMP.VAR.VAL.FMI2ST$
`fmi2Set`	$FMI2COMP.VAR.VAL.FMI2STF$
`fmi2DoStep`	$FMI2COMP.TIME.NZTIME.FMI2STF$
`fmi2Instantiate`	$FMI2COMP.Bool$
`fmi2SetUpExperiment`	$FMI2COMP.TIME.Bool.TIME.FMI2ST$
`fmi2EnterInitializationMode`	$FMI2COMP.FMI2ST$
`fmi2ExitInitializationMode`	$FMI2COMP.FMI2ST$
`fmi2GetBooleanStatusfmi2Terminated`	$FMI2COMP.Bool.FMI2ST$
`fmi2GetMaxStepSize`	$FMI2COMP.TIME.FMI2ST$
`fmi2Terminate`	$FMI2COMP.FMI2ST$
`fmi2FreeInstance`	$FMI2COMP.FMI2ST$
`fmi2GetFMUState`	$FMI2COMP.FMUSTATE.FMI2ST$
`fmi2SetFMUState`	$FMI2COMP.FMUSTATE.FMI2ST$

Table 1. Channels that model FMI API functions

TimedInteractions has two parameters: a start and an end time $t0$ and tN. It uses *Timer* defined above with arguments $t0$, 1, and tN. *Timer* can be interrupted (Δ) by the process *endSimulation*. It, however, runs in parallel ($[\![\,]\!]$) with the process *Interaction*. They synchronise on communications on *step*, *end*, *setT*, *updateSS*, and *endsimulation*, but otherwise proceed independently. The process that results from the parallelism hides ($\setminus$) communications on *step*, *end*, *setT*, and *updateSS*, which are used just internally by *Timer* and *Interaction*.

A complete account of *Circus* can be found in [8]. We explain any extra notation not explained here as needed.

4 A model of FMI

The FMI API consists of functions used by the master algorithm to orchestrate the FMUs. In our model, these functions are defined as channels whose types correspond to the input and output types of the functions; see Table 1.

We use the given type *FMI2COMP* to represent an instance of an FMU. In FMI, these are pointers to an FMU-specific structure that contains the information needed to simulate it. Here, we use identifiers for such components.

Valid variable names and values are represented by the sets *VAR* and *VAL*. We do not model the FMI type system, which includes reals, integers, booleans, characters, strings, and bytes; however, it is not difficult to cater for this type system. Extensions to the type system are expected in future versions of FMI.

The type *FMI2ST* contains flags of the FMI type `fmi2Status` that are returned by the API functions. We include `fmi2OK`, `fmi2Error`, and `fmi2Fatal`, which indicate, respectively, that all is well, the FMU encountered an error, and the computations are irreparable for all FMUs. The extra flag `fmi2Discard` is also included in the superset *FMI2STF*; it can only be returned by `fmi2Set` and `fmi2DoStep`. `fmi2Set` indicates that a status cannot be returned, and in the case of `fmi2DoStep` that a smaller step size is required or the requested information cannot be returned. We do not include `fmi2Warning`, used for logging, and `fmi2Pending`, used for asynchronous simulation steps.

FMUSTATE contains values that represent an internal state of an FMU. It comprises all values (of parameters, inputs, buffers, and so on) needed to continue a simulation. It can be recorded by a master algorithm to support rollback.

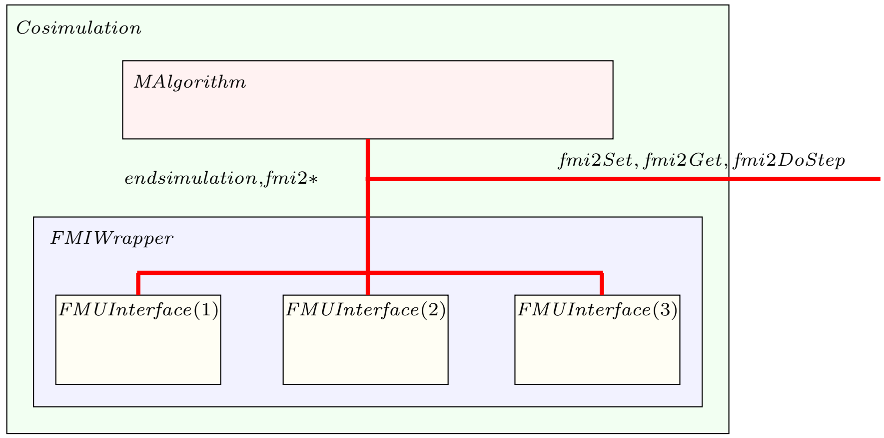

Fig. 2. Structure of a co-simulation model

The signature of the channels impose restrictions on the use of the API. It is not possible to call `fmi2DoStep` with a non-positive step size. Given a particular configuration of FMUs, we can define the types of the `fmi2Get` and `fmi2Set` channels so that setting or getting a variable that is not in the given FMU is undefined. Without this fine tuning, such attempts lead to deadlocks in our model: a check for deadlock freedom ensures the absence of such problems. The API actually includes specialised `fmi2Get` and `fmi2Set` functions for each data type available. As already said, we do not cater for the FMI type system.

The function `fmi2Instantiate` returns a pointer to a component, and null if the instantiation fails. Since we do not model pointers, we use a boolean to cater for the possibility of failure. The function `fmi2GetMaxStepSize` is not part of the standard; we use it to implement the rollback algorithm in [4].

The overall structure of our models of a co-simulation is shown in Fig. 2. The visible channels are `fmi2Get`, `fmi2Set`, and `fmi2DoStep`. So, we can use our model to verify properties of co-simulations that can be described in terms of these interactions, and involving variables from any of the FMUs involved.

The other channels enforce the expected control flow of a master algorithm. They are used for communication between the process *MAlgorithm* that models a master algorithm and each process *FMUInterface*(i) that models the FMU identified by i. We call *FMIWrapper* the collection of FMU interfaces: they execute independently in parallel, that is, in interleaving.

The control channel *endsimulation* is used to shutdown the simulation. Since an FMU may fail, its termination may not be carried out gracefully (with `fmi2Terminate` and `fmi2FreeInstance`). So, *endsimulation* is used to indicate the end of the experiment in all cases and shutdown the model processes.

In what follows, we describe our specifications of *MAlgorithm* (Sect. 4.1) and *FMUInterface* (Sect. 4.2), which provide a correctness criterion for these components. In Sect. 4.3, we describe how to construct models of specific FMUs. Applications of our models are described in Sect. 5.

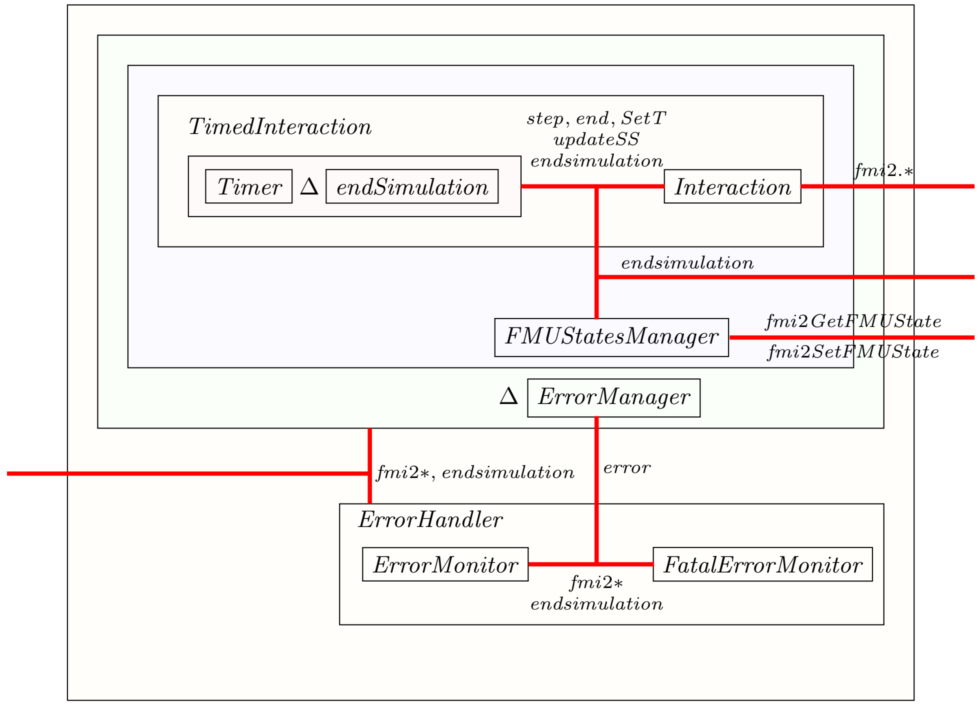

Fig. 3. Structure of a model of a master algorithm

4.1 Master algorithms

A master algorithm is a monolithic program that defines the connections between the FMUs and the time of the simulation steps, and handles any errors raised by an FMU. In our model, we consider each of these aspects of a master algorithm separately. The overall structure of the *MAlgorithm* process is described in Fig. 3. It provides a general characterisation of the valid history of interactions of a master algorithm. It does not commit to specific policies to define step sizes and error handling in case an API function returns `fmi2Discard`. The treatment of `fmi2Error` and `fmi2Fatal` is restricted by the standard.

MAlgorithm has three main components described next. *TimedInteractions* specifies the co-simulation steps and orchestration of the FMUs. *FMUStates-Manager* controls access to the internal state of the FMUs. *ErrorHandler* monitors the occurrence of an *fmi2Error* or *fmi2Fatal* from the API functions.

TimedInteractions has two components. *Timer* is presented in Sect. 3. It uses *step* and *end* to drive the *Interaction* process, which defines the orchestration of the FMUs. This is the core process that restricts the order in which the API functions can be used. *Timer* also exposes channels *setT* and *updateSS* to allow *Interaction* to define algorithms will rollback or a variable step size. The timer can be terminated by the signal *endsimulation* raised by *Interaction*.

Interaction is the sequential composition of *Instantiation*, *InstantiationMode*, *InitializationMode*, and *slaveInitialized*, which correspond to states that define the stages of a co-simulation [12, p.103]. The definitions of these processes depend on the configuration of the FMUs. Given such a configuration, they can be automatically generated as indicated below. A configuration is characterised by a sequence of FMU identifiers ($FMUs : \mathrm{seq}\, FMI2COMP$), and sequences that define the parameters and their values ($parameters : \mathrm{seq}(FMI2COMP \times VAR \times VAL)$), inputs and their initial values ($inputs : \mathrm{seq}(FMI2COMP \times VAR \times VAL)$), outputs ($outputs : \mathrm{seq}(FMI2COMP \times VAR)$), and an input/output port dependency graph [4] *pdg*. Some of this information is also needed to generate automatically a sketch of the models of the FMUs (see Sect. 4.3).

The port dependency graph *pdg* is a relation between outputs and inputs defined by a pair of type $FMI2COMP \times VAR$. The graph establishes how the inputs of each of the FMUs depend on the outputs of the others. It must be acyclic, and this can be automatically checked using the CSP model checker. Using the port dependency graph, once we retrieve the outputs, via the `fmi2Get` function, we know how to provide the inputs, via the `fmi2Set` function.

Instantiation, defined below, instantiates the FMUs. It is an iterated sequential composition (;) of actions $fmi2Instantiate.i?sc \longrightarrow \mathbf{Skip}$, where i comes from *FMUs* and **Skip** is the action that terminates immediately.

InstantiationMode and *InitializationMode* allow the setting up of parameters and initial values of inputs before calling the API function that signals the start of the next phase. We show below *InitializationMode*. For an element *inp* of *inputs*, we use projection functions *FMU*, *name* and *val* to get its components.

$$(;\ inp : inputs \bullet fmi2Set!(FMU\ inp)!(name\ inp)!(val\ inp)?st \longrightarrow \mathbf{Skip});$$
$$(;\ i : FMUs \bullet fmi2ExitInitializationMode!i?st \longrightarrow \mathbf{Skip})$$

We can easily generalise the model to allow an interleaving of the events involved. The value of such a generalisation, however, is unclear (and it harms the possibility of automated verification via model checking).

The process *slaveInitialized* is sketched in Fig. 4; it is driven by the *Timer*. Its state contains a component *rinps*: a function that records, for each FMU identifier a function from the names of its inputs to values. This function is defined by taking the value of each output from the FMUs, and updating *rinps* to record that value for the inputs associated with the output in the port dependency graph. If the *Timer* signals the end, *slaveInitialized* finishes. Otherwise, it collects the outputs, distributes the inputs, and carries out a step.

Similarly to that of *InitializationMode*, the definition of *TakeOutputs* uses an iterated sequence, now over *outputs*: the sequence of pairs that identify an FMU and an output name. Once the value v of an output *out* is obtained, it is assigned to each input *inp* in the sequence $pdf(out)$ associated with *out* in the port dependency graph *pdg*. We use $\oplus$ to denote function overriding.

DistributeInputs uses *inp* to set the inputs of the FMUs using $fmi2Set$. *Step* proceeds with the calls to $fmi2DoStep$ and if all goes well, recurses back to the *Main* action of *slaveInitialized*. Their definitions are omitted for brevity.

FMUStatesManager controls the use of the functions `fmi2GetFMUState` and `fmi2SetFMUState` for each of the FMUs. It is an interleaving of instances of the

$\mathbf{process}\ \ slaveInitialized \mathrel{\widehat=}$
$\mathbf{state}\ \ State == [rinps : FMI2COMP \mathrel{+\!\!\!\to} (VAR \mathrel{+\!\!\!\to} VAL)]$
$\ldots$
$TakeOutputs \mathrel{\widehat=}$
$\ ;\ out : outputs \bullet fmi2Get.(FMU\ out).(name\ out)?v \longrightarrow$
$\quad ;\ inp : pdg(out) \bullet$
$\qquad rinps := rinps \oplus \{\,(FMU\ inp) \mapsto ((rinps\,(FMU\ inp)) \oplus \{(name\ inp) \mapsto v\})\,\}$
$Main \mathrel{\widehat=}\ end \longrightarrow \mathbf{Skip}$
$\qquad\qquad \Box\ step?t?hc \longrightarrow TakeOutputs;\ DistributeInputs;\ Step$
$\bullet\ Main$
$\mathbf{end}$

Fig. 4. Sketch of *slaveInitialized*

$\mathbf{process}\ \ FMUStatesManager \mathrel{\widehat=} i : FMI2COMP \bullet \mathbf{begin}$

$AllowAGet \mathrel{\widehat=} fmi2GetFMUState.i?s?st \longrightarrow AllowsGetsAndSets(s)$

$AllowsGetsAndSets \mathrel{\widehat=} s : FMUSTATE \bullet$
$\quad fmi2GetFMUState.i?t?st \longrightarrow AllowsGetsAndSets(t)$
$\quad \Box\ fmi2SetFMUState.i!s?st \longrightarrow AllowsGetsAndSets(s)$

$\bullet\ fmi2Instantiate.i?b \longrightarrow AllowAGet$

$\mathbf{end}$

Fig. 5. Model of *FMUStateManager*

process $FMUStateManager(i)$ in Fig. 5 for each of the FMUs. Once an FMU is instantiated, then it is possible to retrieve its state. After that, both gets and sets are allowed. The actual values of the state are defined in the FMUs, but recorded in the master algorithm via $fmi2GetFMUState$ for later use with $fmi2SetFMUState$ as defined in $FMUStateManager(i)$.

For complex internal states, model checking can become infeasible (although we have managed it for simple examples). To carry out verifications that are independent of the values of the internal state of the FMUs, we need to adjust only this component. Some examples, explored in the next section, are properties of algorithms that do not support retrieval and resetting of the FMU states, determinism and termination of algorithms, and so on.

The *ErrorHandler* process contains two components: monitors for $fmi2Error$ and $fmi2Fatal$. If any of the API functions returns an error, they signal that to the *ErrorManager* via a channel *error*. Upon an error, the *ErrorManager* interrupts the main flow of execution. In the case of an $fmi2Fatal$ error, the simulation is stopped via *endsimulation*. In the case of an $fmi2Error$, a call to $fmi2FreeInstance$ is allowed, before the simulation is ended.

4.2 FMU interfaces

The model of a valid FMU is simpler. It captures the control flow of an FMU, specifying, at each stage, the API functions to which it can respond. Unsurprisingly, it has some of the restrictions of a master algorithm, but it is much more lax, in that it captures just the expected capabilities of an FMU.

At first, the only API function that is available is `fmi2Instantiate`. The simple action below specifies this behaviour.

$$\begin{array}{l} \mathit{Instantiation} = \\ \quad \mathit{fmi2Instantiate}.i?b \longrightarrow \left(\begin{array}{l} b \mathbin{\&} \mathit{status} := \mathit{fmi2OK};\ \mathit{Instantiated} \\ \Box \\ \neg\ b \mathbin{\&} \mathit{status} := \mathit{fmi2Fatal};\ \mathit{RUN}(\mathit{FMUAPI}(i)) \end{array} \right) \end{array}$$

A state component *status* records the result of the last call to an API function. In this case, it is updated based on the boolean b returned by *fmi2Instantiate*. If the instantiation is successful, the behaviour is described by *Instantiated*, sketched below; otherwise, it is unrestricted: specified by $\mathit{RUN}(\mathit{FMUAPI}(i))$, which allows the occurrence of any API functions, in any order.

$$\begin{array}{rl} \mathit{Instantiated} = & \mathit{status} = \mathit{fmi2Fatal} \mathbin{\&} \mathit{RUN}(\mathit{FMUAPI}(i)) \\ & \Box\ \mathit{status} \notin \{\mathit{fmi2Error}, \mathit{fmi2Fatal}\} \mathbin{\&} \\ & \quad \left(\begin{array}{l} \mathit{fmi2Get}.i?n?v?st \longrightarrow \mathit{status} := st;\ \mathit{Instantiated} \\ \Box\ \mathit{fmi2DoStep}.i?t?hc?st \longrightarrow \mathit{status} := st;\ \mathit{Instantiated} \\ \Box \cdots \end{array} \right) \\ & \Box\ st \neq \mathit{fmi2Fatal} \mathbin{\&} \mathit{fmi2FreeInstance}!i?st \longrightarrow \cdots \end{array}$$

Again, if there is a fatal error, the behaviour is unrestricted. If there is no error, all functions except `fmi2Instantiate` are available. Finally, if there is a non-fatal error, only `fmi2FreeInstance` is possible.

While a pattern of calls is defined by a master algorithm, so that, for example, all outputs are obtained before the inputs are distributed, the FMU is passive and does not impose such a policy on its use. So, the various actions enforce only the restrictions in the standard [12, p.105].

Although it is possible to specify a more restricted behaviour for FMUs, such a specification rules out robust FMU implementations that handle calls to the API functions that do not necessarily follow the strict pattern of a co-simulation. Next, we describe how to generate FMU models that follow a more restricted pattern that is adequate for use with valid master algorithms.

4.3 Specific FMU models

In the previous section, we have presented a general model for an FMU. The particular model of an FMU depends, of course, on its functionality, and must conform to (trace refine) our general model. This can be proved via model checking for stateless models of FMUs that do not offer the facility to retrieve and set its internal state. In this case, the models do not offer the choices of communications $\mathit{fmi2GetFMUState}.i?st$ and $\mathit{fmi2SetFMUState}.i?st$. The availability of such facilities is defined by capability flags of the FMU.

We can, however, generate a sketch of the model of an FMU using information about its structure: lists of parameters p_i, inputs inp_i, and outputs out_i.

$$
\begin{array}{l}
\textbf{process}\ \ \mathit{FMUSketch} \mathrel{\widehat=} i : \mathit{FMI2COMP} \bullet \textbf{begin} \\[1ex]
\textbf{state}\ \mathit{State} = [\mathit{currentTime}, \mathit{endTime} : \mathit{TIME};\ \mathit{cp}_i, \mathit{cinp}_i, \mathit{cev}_i, \mathit{cout}_i] \\[1ex]
\mathit{Instantiation} = \mathit{fmi2Instantiate}.i!\mathit{true} \longrightarrow \textbf{Skip} \\[1ex]
\mathit{InstantiationMode} = \\
\quad \mathit{fmi2Set}.i.p_i?v!\mathit{fmi2OK} \longrightarrow \mathit{cp}_i := v;\ \mathit{InstantiationMode} \\
\quad \Box\ \mathit{fmi2SetUpExperiment}.i?t0!\mathit{true}?tN!\mathit{fmi2OK} \longrightarrow \\
\qquad \mathit{currentTime}, \mathit{endTime} := t0, tN; \\
\qquad \mathit{fmi2EnterInitializationMode}.i!\mathit{fmi2OK} \longrightarrow \textbf{Skip} \\[1ex]
\mathit{InitializationMode} = \\
\quad \mathit{fmi2Set}.i.\mathit{inp}_i?v!\mathit{fmi2OK} \longrightarrow \mathit{cinp}_i := v;\ \mathit{InitializationMode} \\
\quad \Box\ \mathit{fmi2ExitInitializationMode}.i!\mathit{fmi2OK} \longrightarrow \mathit{UpdateState} \\[1ex]
\mathit{slaveInitialized} = \\
\quad \mathit{fmi2Get}.i.\mathit{out}_i!\mathit{cout}_i!\mathit{fmi2OK} \longrightarrow \mathit{slaveInitialized} \\
\quad \Box\ \mathit{fmi2Set}.i.\mathit{inp}_i?v.\mathit{fmi2OK} \longrightarrow \mathit{cinp}_i := v;\ \mathit{slaveInitialized} \\
\quad \Box\ \mathit{fmi2DoStep}.i?t?\mathit{ss}!\mathit{fmi2OK} \longrightarrow (\mathit{UpdateState};\ \mathit{slaveInitialized}) \\[1ex]
\bullet\ \mathit{Instantiation};\ \mathit{InstantiationMode};\ \mathit{InitializationMode}; \\
\quad (\mathit{slaveInitialized}\ \Delta \\
\qquad \mathit{fmi2Terminate}.i!\mathit{fmi2OK} \longrightarrow \mathit{fmi2FreeInstance}.i!\mathit{fmi2OK} \longrightarrow \textbf{Stop}) \\[1ex]
\textbf{end}
\end{array}
$$

Fig. 6. Sketch of a model for a specific FMU

This information is used to construct a master algorithm (see Sect. 4.1). Fig. 6 shows the sketch of a *Circus* process with the FMU behaviour. Its state includes components cp_i, $cinp_i$, and $cout_i$, besides the current and end simulation time.

Its structure is similar to that of the *Interaction* process used to model a master algorithm. In all cases, the interactions flag success ($\mathit{fmi2OK}$). If an FMU makes assumptions about its inputs, the possibility of error can be modelled. For example, *Instantiation* indicates success, but to explore the possibility of failure, we can define it as $\mathit{fmi2Instantiate}.i?b \longrightarrow \textbf{Skip}$. The action *UpdateState* is left unspecified. It is this action that specifies the functionality of the FMU. It can be automatically generated if there is a more complete model of the FMU. For example, [7] shows the case if a discrete-time Simulink model is available.

If the FMU supports retrieval and update of its state, we need to add the following choices to *InstantiationMode*, *InitializationMode*, and *slaveInitialized*.

$$
\begin{array}{l}
\ldots \\
\Box\ \mathit{fmi2GetFMUState}.i!\theta\,\mathit{State}!\mathit{fmi2OK} \longrightarrow \cdots \\
\Box\ \mathit{fmi2SetFMUState}.i?s?\mathit{st} \longrightarrow \theta\,\mathit{State} := s;\ \cdots
\end{array}
$$

Via $\mathit{fmi2GetFMUState}$, it outputs the whole state record, that is, $\theta\,\mathit{State}$, and via $\mathit{fmi2SetFMUState}$, we can update it.

If the state, either via setting of parameters and input or via an update, may become invalid, we can flag $\mathit{fmi2Fatal}$ and deadlock. For example, we consider the test case shown in Fig. 7 taken from [5]. It has been designed to show that

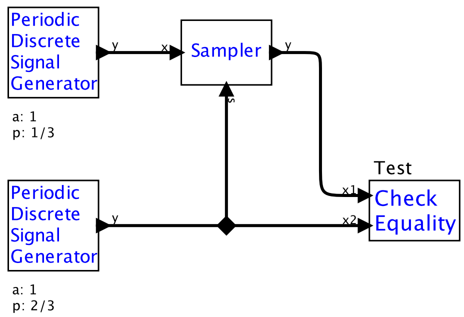

Fig. 7. Test case for sampling of discrete event signals [5]

components with discrete timed behaviour coordinate their representation of time. There are three main components: two periodic discrete signal generators, both generating the same signal, one with period one time unit and the other two time units; and a discrete sampler. The test criterion is that the output of the Sampler should equal the output of the second periodic discrete signal generator at all superdense times. There is an implicit constraint that the period p should not be 0; therefore, we specify its *InstantiationMode* action as follows.

$$
\begin{array}{l}
\mathit{InstantiationMode} = \\
\quad \mathit{fmi2Set}.i.a?v!\mathit{fmi2OK} \longrightarrow a := v \longrightarrow \mathit{InstantiationMode} \\
\quad \Box\ \mathit{fmi2Set}.i.p?v!\mathit{fmi2OK} \longrightarrow p := v \longrightarrow \mathit{InstantiationMode} \\
\quad \Box\ p \neq 0\ \&\ \mathit{fmi2SetUpExperiment}.i?t0!\mathit{true}?tN!\mathit{fmi2OK} \longrightarrow \\
\qquad \mathit{currentTime}, \mathit{endTime} := t0, tN; \\
\qquad \mathit{fmi2EnterInitializationMode}.i!\mathit{fmi2OK} \longrightarrow \mathbf{Skip} \\
\quad \Box\ p = 0\ \&\ \mathit{fmi2SetUpExperiment}.i?t0!\mathit{true}?tN!\mathit{fmi2Fatal} \longrightarrow \mathbf{Stop}
\end{array}
$$

In this case, if the experiment is set up when p is 0, we have a fatal error.

An FMU model generated as just explained trace refines $\mathit{FMUInterface}(i)$. This means that all possible histories of interactions of the FMU are possible for $\mathit{FMUInterface}(i)$ and, therefore, valid according to that criterion. We have proofs of refinement for all FMUs in Fig. 7 and for a data-flow network.

5 Evaluation: verification applications

In this section, we show how we can use our formal semantics for FMI to verify master algorithms and to study system properties via their co-simulations. For automation, our semantics can be translated from *Circus* to `CSPM` (the input language for the model checker FDR3), using a strategy similar to that of [20], so that it can be both model checked in FDR3 and executed in ProBe (FDR's process behaviour explorer), for suitably chosen model parameters.

5.1 Master algorithms

As well as giving a correctness criterion for a master algorithm, the model presented in Sect. 4 gives an indication of how to construct models for particular algorithms. We consider here three examples.

Classic brute-force The simplest algorithm uses a fixed step size, has no access to the state of the FMUs, and queries them for termination if `fmi2Discard` is flagged. To model this algorithm, we define a process *ClassicMAlgorithm* with the same structure shown in Fig. 3, but more specific components.

ClassicMAlgorithm uses a simple timer that does not use *setT* or *updateSS*. For the *FMUStatesManager*, we use a simple process that just terminates immediately. Finally, for *Interaction*, we use the parallel composition of *Interaction* itself with a process *DiscardMonitor*, whose main action is *Monitor* defined below, followed by an action *Terminated* that shuts down the FMUs.

$$\begin{array}{l} \mathit{Monitor} \mathrel{\widehat=} \\ \quad \mathit{fmi2DoStep?i?t?hc?st} : \mathit{st} \neq \mathit{fmi2Discard} \longrightarrow \mathit{Monitor} \\ \quad \Box\ \mathit{fmi2DoStep?i?t?hc.fmi2Discard} \longrightarrow \\ \quad\quad \left(\begin{array}{l} \mathit{fmi2GetBooleanStatusfmi2Terminated.i.true?st} \longrightarrow \mathit{ToDiscard} \\ \Box\ \mathit{fmi2GetBooleanStatusfmi2Terminated.i.false?st} \longrightarrow \mathit{Monitor} \end{array}\right) \\ \quad \Box\ \mathit{stepAnalysed} \longrightarrow \mathit{Monitor}\ \Box\ \mathit{step?t?hc} \longrightarrow \mathit{Monitor} \\ \quad \Box\ \mathit{end} \longrightarrow \mathbf{Skip} \end{array}$$

Monitor ignores all flags *st* returned by *fmi*2*DoStep* except *fmi*2*Discard*. If this flag is returned, it queries the FMU using *fmi*2*GetBooleanStatusfmi*2*Terminated*. If the FMU requests termination, *Monitor* behaves like *ToDiscard* whose simple definition we omit. In *ToDiscard*, when completion of the step is indicated via either a *stepAnalysed* or a *step*?*t*?*hc* event, the co-simulation is terminated. The signal *stepAnalysed* is not part of the *Interaction* interface, but is used to indicate that `fmi2DoStep` has been carried out for all FMUs, and we are now in a position to decide how to continue with the co-simulation.

Since *ClassicMAlgorithm* has the same structure as *MAlgorithm*, we can prove refinement by considering each of the components in isolation. While proof of refinement by model checking for the whole model is not feasible, it is feasible for the individual components. In the sequel, we use the same approach to analyse more complex algorithms. It is also feasible to prove that *ClassicMAlgorithm* terminates, but otherwise does not deadlock, and is deterministic.

The example in the FMI standard is a classic algorithm with a fixed step and handling of `fmi2Discard`, but does not include error management. So, its specification does not include the *ErrorHander* and the *ErrorManager*. Model checking can show that this is not a valid algorithm. A simple counterexample shows that it continues and calls `fmi2Instantiate` a second time even after the first call returns an `fmi2Fatal` flag. This is explicitly ruled out in the standard.

Simulink This is a widely used tool for simulation based on control law diagrams [19]. A popular solver uses a variable-step policy based on change rate

process *VaryStep* $\widehat{=}$ *threshold* : *VAL*; *initialSS* : *NZTIME* • **begin**

state
$State = [oldOuts, newOuts : (FMI2COMP \times VAR) \rightarrow\!\!\!\!\!+ VAL;\ currentSS : NZTIME]$

Init
$State'$

$\mathrm{dom}\ oldOuts' = \mathrm{ran}\ outputs \wedge \mathrm{ran}\ oldOuts = \epsilon \wedge newOuts' = \varnothing$
$currentSS' = initialSS$

$Monitor \mathrel{\widehat{=}} ;\ out : outputs \bullet$
$fmi2Get.(FMU\ out).(name\ out)?nv?st \longrightarrow newOuts := newOuts \oplus \{out \mapsto nv\}$

$Adjust \mathrel{\widehat{=}}$ **if** $delta(oldOuts, newOuts) \geq threshold \longrightarrow$
$currentSS := newstep(delta(oldOuts, newOuts), currentSS);$
$updateSS!currentSS \longrightarrow$ **Skip**
$[]\ delta(oldOuts, newOuts) > threshold \longrightarrow$ **Skip**
fi

$Step = Monitor;\ Adjust;\ Step$

$\bullet\ Init;\ (Step\ \triangle\ endSimulation)$

end

Fig. 8. Model of *VaryStep*

of the state. To model this algorithm, we use a process *SimulinkMAlgorithm*, which is similar to *ClassicMAlgorithm*, but has another monitor *VaryStep*, specified in Fig. 8. It is composed in parallel with *Interaction* to define a process *VariableStepInteraction* used in *SimulinkMAlgorithm*.

VaryStep takes as parameters a *threshold* for change and the initial value of the step size *initialSS*. Taking a simple approach, we define a state that records the old (*oldOuts*) and new (*newOuts*) values of the outputs, besides the current step size *currentSS*. After the state is initialised (using the action *Init*) to record undefined (ϵ) old values for the outputs, no new values (empty function $\varnothing$), and the initial step size, the monitor steps by recording the new output values (*Monitor*) and then changing the step size (*Adjust*). Adjustment is based just on a comparison between the old and new values defined by an (omitted) function *delta*. If the *threshold* is reached, a new step size is defined by another function *newstep* and informed to the *Timer*.

We have established that *SimulinkMAlgorithm* is valid, that is, it refines *MAlgorithm*, by proving that the new *VariableStepInteraction* refines *Interaction*. We have also proved termination, deadlock freedom, and determinism.

Rollback In the same way as illustrated by *VaryStep* in Fig. 8, we can model a sophisticated algorithm suggested in [4]. We define a *Rollback* monitor that has the same structure as *VaryStep*. Its *Monitor* (a) saves the state using *fmi2GetFMUState* before each step of co-simulation, and (b) queries the maximum step size that each FMU is prepared to take. This uses an extra FMI API function `fmi2GetMaxStepSize`. In *Adjust*, if any of the maximum values returned is lower than that originally proposed, the states of the FMUs are reset using *fmi2SetFMUState*, and the time as well as the step size are adjusted (using *setT* and *updateSS*). We have again proved validity, termination, and determinism.

In [4], determinism is also based on the FMU states, which are visible via `fmi2Get` and `fmi2Set`. On the other hand, that work considers determinism with respect to the order of retrieval and update of variables and execution of the FMUs. In our models, this order is fixed. To establish determinism in that sense, we need to consider a highly parallel model with all valid execution orders respecting the port dependency graph. This is the approach in [7], where verification uses theorem proving. The approach taken here is more amenable to model checking and sufficient to verify sequential implementations of simulations.

As explained in the previous section, the definition of *Interaction* is determined by structural information about the FMUs configuration. Using that information, and a choice of master algorithm (fixed or variable step, treatment of `fmi2Discard`, and so on), we can obtain a model. For the FMUs, in the previous section, we have explained how to derive (sketches of) models.

5.2 Co-simulations

Our semantics is also useful for analysis using FDR of the FMU compositions in co-simulations for deadlock, livelock, and determinism. We have done this verification, for instance, for the discrete event signal example in Fig. 7.

The semantics can also be used to validate the results of co-simulation runs. For example, Fig 9 describes a short scenario involving two co-simulation steps. We specify it using `CSP-M`, rather than *Circus*, and write the traces refinement (`[T=`) assertion we use for verification. The assertion says that this scenario is a possible trace of the model: it is a correct co-simulation run. (We may check this by noting that the final two operations set the same inputs for FMU 4 (Check Equality)—the FMU that checks equality in the simulation model.) To facilitate model checking, we use numbers for the names of the variables. With this approach, we validate our model against an actual co-simulation.

Moreover, we can go further and check behavioural correctness too. The specification of an FMI composition $\mathcal{C}$ is an assertion over traces of events corresponding to the FMI API, principally `doStep`, `get`, and `set`. A similar technique is used for specification of processes in CSPm based on traces of events [17], and in CCS, using temporal logic over actions [3].

An alternative is to use a more abstract composition of FMUs $\mathcal{A}$ as a specification. $\mathcal{A}$ can be used as an oracle in testing the simulation: do a step of $\mathcal{C}$ and then compare it with a step of $\mathcal{A}$. $\mathcal{A}$ and $\mathcal{C}$ can be used even more directly in our model by carrying out a refinement check in FDR3.

Consider a dataflow process taken from [17, p.124] and depicted in Fig. 10 that computes the weighted sums of consecutive pairs of inputs. So, if the input is $x_0, x_1, x_2, x_3, \ldots$, then the output is $(a*x_0+b*x_1), (a*x_1+b*x_2), (a*x_2+b*x_3), \ldots$, for weights a and b. The network has two external channels, *left* and *right*, and three internal channels. $X2$ multiplies an input on channel *left*1 by a and passes

```
DSynchronousEventsSpec =
  -- Set parameters
  fmi2Set.1.1.1.fmi2OK -> fmi2Set.1.2.1.fmi2OK ->
  fmi2Set.2.1.1.fmi2OK -> fmi2Set.2.2.2.fmi2OK ->
  -- Set initial values of inputs
  fmi2Set.3.1.1.fmi2OK -> fmi2Set.3.2.1.fmi2OK ->
  fmi2Set.4.1.1.fmi2OK -> fmi2Set.4.2.1.fmi2OK ->
  -- Steps
  fmi2Get.1.1.1.fmi2OK -> fmi2Get.2.1.1.fmi2OK -> fmi2Get.3.3.1.fmi2OK ->
  fmi2Set.3.1.1.fmi2OK -> fmi2Set.3.2.1.fmi2OK ->
  fmi2Set.4.2.1.fmi2OK -> fmi2Set.4.1.1.fmi2OK ->
  fmi2DoStep.1.0.2.fmi2OK -> fmi2DoStep.2.0.2.fmi2OK ->
  fmi2DoStep.3.0.2.fmi2OK -> fmi2DoStep.4.0.2.fmi2OK ->
  fmi2Get.1.1.1.fmi2OK -> fmi2Get.2.1.1.fmi2OK -> fmi2Get.3.3.1.fmi2OK ->
  fmi2Set.3.1.1.fmi2OK -> fmi2Set.3.2.1.fmi2OK ->
  fmi2Set.4.2.1.fmi2OK -> fmi2Set.4.1.1.fmi2OK ->
  fmi2DoStep.1.2.2.fmi2OK -> fmi2DoStep.2.2.2.fmi2OK ->
  fmi2DoStep.3.2.2.fmi2OK -> fmi2DoStep.4.2.2.fmi2OK -> SKIP

assert Cosimulation(0,2) [T= SynchronousEventsSpec
```

Fig. 9. Scenarios for Fig 7: sampling of discrete event signals

the result to $X3$ on *mid*. $X3$ multiplies an input on the *left*2 channel by b and adds the result to the corresponding value from the *mid* channel. $X1$ duplicates its inputs and passes them to the other two processes (since all values except the first and last are used twice), where the multiplications can be performed in parallel. A little care needs to be taken to get the order of communications on the *left*1 and *left*2 channels right, otherwise a deadlock soon ensues.

The CSP specification of this network remembers the previous input.

$$
\begin{array}{l}
DFProc(a, b) = left?x \longrightarrow P(x) \\
P(x) = left?y \longrightarrow right!(a * x + b * y) \longrightarrow P(y)
\end{array}
$$

The key part of the main FMU in this specification is shown in Fig 11.

Once the slave FMU has been initialised, the master algorithm can instruct it to perform a simulation step (`fmi2DoStep`). The FMU fetches the state item, gets the next input, fetches the parameters a and b, performs the necessary computation, and stores it as the current output.

We have been able to encode both the specification and implementation of the data flow network, with small values for *maxint*, and check behavioural refinement. We have identified the problem alluded to above, in getting the communications on *left*1 and *left*2 in the wrong order; issues to do with determinism concerning hidden state in our model; and termination issues to do with the end of the experiment and closing down resources. We have also been able to demonstrate in a small way the consistency of the semantics model.

The transformation from *Circus* to `CSPM` corresponding to the FMI API requires the identification of barrier synchronisations that correspond to the `doStep` commands. An appropriate strategy is outlined in [6].

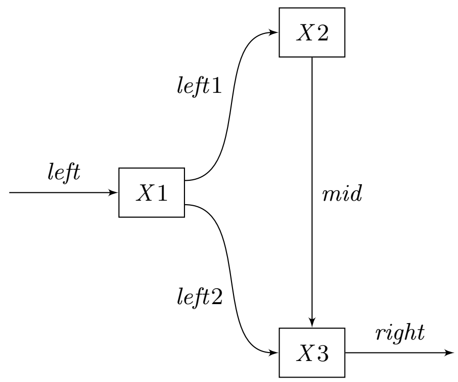

Fig. 10. A data-flow example

```
DFSPECFMUProc(i) =
  let
    slaveInitialized(hc) =
      ...
      []
      fmi2DoStep.i?t?ss!fmi2OK -> (UpdateState; slaveInitialized(ss))
    UpdateState =
      get.i.1?x:INPUTVALP ->
        getinput.i.1?y:INPUTVALP ->
          getparam.i.1?a:PARAMVAL -> getparam.i.2?b:PARAMVAL ->
            setoutput.i.1!(a*x+b*y) -> SKIP
  within
    Instantiation; InstantiationMode(eps,eps);
    InitializationMode; slaveInitialized(0)
```

Fig. 11. Data flow specification

6 Conclusions

We have provided a comprehensive model of the FMI API, characterising formally valid master algorithms and FMUs. We can use our models to prove validity of master algorithms and FMU models. For stateless models, model checking is feasible, and we can use that to establish properties of interest of algorithms and FMU models. For state-rich models, we need theorem proving.

Given information about the network of FMUs and a choice of master algorithm, it is possible to construct a model of their co-simulation automatically for reasoning about the whole system. This is indicated by how our models are defined in terms of information about parameters, inputs, and so on, for each FMU, and about the FMU connections. A detailed account of the generation process and its mechanisation are, however, left as future work.

We have discussed a few example master algorithms. This includes a sophisticated rollback algorithm presented in [4] using a proposed extension of the FMI. It uses API functions to get and set the state of an FMU. In [4], this algorithm uses a `doStep` function that returns an alternative step size, in case the input step size is not possible. Here, instead, we use an extra function that can get the alternative step size. This means that our standard algorithms respect the existing signature of the `fmi2DoStep` function. As part of our future work, we plan to model one additional master algorithm proposed in [4].

There has been very practical work on new master algorithms, generation of FMUs and simulations, and hybrid models [2, 22, 11, 9]. Tripakis [25] shows how components with different underlying models (state machines, synchronous data flow, and so on) can be encoded as FMUs. Savicks [24] presents a framework for co-simulation of Event-B and continuous models based on FMI, using a fixed-step master algorithm and a characterisation of simulation components as a class specialised by Event-B models or FMUs. This work has no semantics for the FMI API, but supplements reasoning in Event-B with simulation of FMUs.

Pre-dating FMI, the work in [15] presents models of co-simulations using timed automata, with validation and verification carried out using UPPAAL, and support for code generation. It concentrates on the combination of one continuous and one discrete component using a particular orchestration approach. The work in [5] discusses the difficulties for treatment of hybrid models in FMI.

There are several ways in which our models can be enriched: definition of the type system, consideration of asynchronous FMUs, sophisticated error handling policies that allow resetting of the FMU states, and increased coverage of the API. FMI includes capability flags that define the services supported by FMUs, like asynchronous steps, and retrieval and update of state, for example. We need a family of models to consider all combinations of values of the capability flags. We have explained here how a typical combination can be modelled.

Our long-term goal is to use our semantics to reason about the overall system composed of the various simulation models. In particular, we are interested in hybrid models, involving FMUs defined by languages for discrete and for continuous modelling. To cater for models involving continuous FMUs, we plan to use a *Circus* extension [13]. Using current support for *Circus* in Isabelle [14], we may also be able to explore code generation from the models. We envisage fully automated support for generation and verification of models and programs.

Acknowledgements The work is funded by the EU INTO-CPS project (Horizon 2020, 664047). Ana Cavalcanti and Jim Woodcock are also funded by the EPSRC grant EP/M025756/1. Anonymous referees have made insightful suggestions. No new primary data were created during this study.

References

1. Abrial, J.R.: Modeling in Event-B—System and Software Engineering. Cambridge University Press (2010)
2. Bastian, J., Clauß, C., Wolf, S., Schneider, P.: Master for co-simulation using FMI. In: Modelica Conference (2011)
3. Bradfield, J.C., Stirling, C.: Verifying temporal properties of processes. In: Baeten, J.C.M., Klop, J.W. (eds.) CONCUR '90, Theories of Concurrency: Unification and Extension. LNCS, vol. 458, pp.115–125. Springer (1990)

4. Broman, D., et al.: Determinate composition of FMUs for co-simulation. In: ACM SIGBED Intl Conf. on Embedded Software. IEEE (2013)
5. Broman, D., et al.: Requirements for Hybrid Cosimulation Standards. In: 18th Intl Conf. on Hybrid Systems: Computation and Control. pp.179–188. ACM (2015)
6. Butterfield, A., Sherif, A., Woodcock, J.C.P.: Slotted *Circus*: A UTP-family of reactive theories. In: Intl Conf. on Integrated Formal Methods. LNCS, vol. 4591, pp.75–97. Springer-Verlag (2007)
7. Cavalcanti, A.L.C., Clayton, P., O'Halloran, C.: From Control Law Diagrams to Ada via *Circus*. Formal Aspects of Computing 23(4), 465–512 (2011)
8. Cavalcanti, A.L.C., Sampaio, A.C.A., Woodcock, J.C.P.: A Refinement Strategy for *Circus*. Formal Aspects of Computing 15(2–3), 146–181 (2003)
9. Denil, J., et al.: Explicit semantic adaptation of hybrid formalisms for FMI co-simulation. In: Spring Simulation Multi-Conference (2015)
10. Derler, P., Lee, E.A., Vincentelli, A.S.: Modeling cyber-physical systems. Procs of IEEE 100(1) (2012)
11. Feldman, Y.A., Greenberg, L., Palachi, E.: Simulating Rhapsody SysML blocks in hybrid models with FMI. In: Modelica Conference (2014)
12. FMI development group: Functional mock-up interface for model exchange and co-simulation, 2.0. https://www.fmi-standard.org (2014)
13. Foster, S., et al.: Towards a UTP semantics for Modelica. In: Unifying Theories of Programming. LNCS, Springer (2016)
14. Foster, S., Zeyda, F., Woodcock, J.C.P.: Isabelle/UTP: A Mechanised Theory Engineering Framework. In: Naumann, D. (ed.), Unifying Theories of Programming, LNCS, vol. 8963, pp.21–41. Springer (2015)
15. Gheorghe, L., et al.: A Formalization of Global Simulation Models for Continuous/Discrete Systems. In: Summer Computer Simulation Conf. pp.559–566. Society for Computer Simulation International (2007)
16. Gibson-Robinson, T., et al.: FDR3—A Modern Refinement Checker for CSP. In: Tools and Algorithms for the Construction and Analysis of Systems. pp. 187–201 (2014)
17. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall (1985)
18. Kübler, R., Schiehlen, W.: Two methods of simulator coupling. Mathematical and Computer Modelling of Dynamical Systems 6(2), 93–113 (2000)
19. The MathWorks, Inc.: Simulink, `www.mathworks.com/products/simulink`
20. Oliveira, M.V.M., Cavalcanti, A.L.C.: From *Circus* to JCSP. In: 6th Intl Conf. on Formal Engineering Methods. LNCS, vol. 3308, pp.320–340. Springer (2004)
21. Oliveira, M.V.M., Cavalcanti, A.L.C., Woodcock, J.C.P.: A UTP Semantics for *Circus*. Formal Aspects of Computing 21(1-2), 3–32 (2009)
22. Pohlmann, U., et al.: Generating functional mockup units from software specifications. In: Modelica Conference (2012)
23. Roscoe, A.W.: Understanding Concurrent Systems. Texts in Computer Science, Springer (2011)
24. Savicks, V., et al.: Co-simulating Event-B and Continuous Models via FMI. In: Summer Simulation Multiconference. pp. 37:1–37:8. Society for Computer Simulation International (2014)
25. Tripakis, S.: Bridging the semantic gap between heterogeneous modeling formalisms and FMI. In: Intl Conf. on Embedded Computer Systems: Architectures, Modeling, and Simulation. pp. 60–69. IEEE (2015)
26. Woodcock, J.C.P., Davies, J.: Using Z—Specification, Refinement, and Proof. Prentice-Hall (1996)