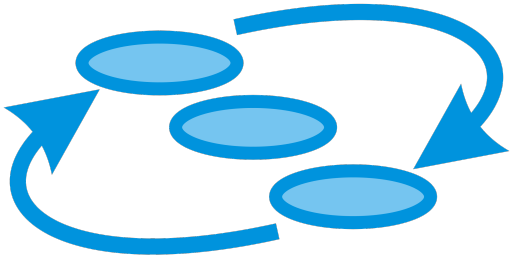
Grant Agreement: 644047

INtegrated TOol chain for model-based design of CPSs



Foundations for FMI Co-modelling

Deliverable Number: D2.2d

Version: 1.0

Date: 5th December 2016

Public Document

http://into-cps.au.dk

Contributors:

Ana Cavalcanti, UY
Jim Woodcock, UY

Editors:

Jim Woodcock, UY

Reviewers:

Stylianos Basagiannis, UTRC
Bernhard Thiele, LIU
Richard Payne, UNEW

Consortium:

Aarhus University	AU	Newcastle University	UNEW
University of York	UY	Linköping University	LIU
Verified Systems International GmbH	VSI	Controllab Products	CLP
ClearSy	CLE	TWT GmbH	TWT
Agro Intelligence	AI	United Technologies	UTRC
Softeam	ST		

Document History

Ver	Date	Author	Description
0.1	17-08-2016	Jim Woodcock	Initial version with writing plan
0.2	30-10-2016	Jim Woodcock	Draft for internal review
0.3	05-12-2016	Jim Woodcock	Draft, including internal reviewers' comments
1.0	12-12-2016	Jim Woodcock	Final version for external review

Abstract

The objective of Task T2.4 is to provide formal foundations for co-modelling with the Functional Mockup Interface (FMI). In Year 2, we have created the first behavioural semantics for the FMI standard. 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.

Contents

Objectives for Task T2.4

The overall objective for the task is to provide formal foundations for co-modelling with FMI. The objectives in Year 2 were to define an INTO-CPS FMI-based semantics for a collection of models connected as a graph of dependencies described in SysML.

1 Introduction

The Functional Mock-up Interface (FMI) [13] 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. To ensure the standard's growth, there is a formal development process; dozens of tools support FMI.

An FMI co-simulation [5] 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* [22], a state-rich process algebra that combines Z [27] for data modelling and CSP [24] 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 [20], and a more robust algorithm that caters for FMU failures [5, p.106]. This revealed that the example in the standard implicitly assumes that FMUs do not raise fatal errors; it is not a valid algorithm. Indeed, the standard does point out that error handling is implemented in a very rudimentary way; however, it is the only example

of a master algorithm in the standard.

Circus models, with abstracted state, can be translated to CSP and verified using the FDR3 model checker [17]. 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 [15]. 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 [14] is under development.

Broman [5] 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.

Sections 2 and 3 describe FMI for co-simulation and *Circus*. Section 4 describes the *Circus* semantics of FMI. The specification and verification of master algorithms and co-simulations is discussed in Section 5. Section 6 presents our conclusions. Three appendices contain working code for our CSP models: Appendix A describes the CSP model of an FMI master algorithm; Appendix B describes the CSP model of a data-flow machine implemented in the FMI architecture (see p. 24); and Appendix C describes the CSP code for an FMI mode of a periodic discrete signal generator taken from [6] (see page 19). All three models are accepted and analysed by FDR 3.4.0.

2 FMI

Modelling and simulating cyber-physical systems (CPSs) [11] involves different engineering fields: a global system with components tackled by domain engineers using specialised tools. Co-simulation [19] involves tool interoperability for modelling and simulating heterogeneous components. Each model can make use of the tool and notation that is most appropriate for the task at hand. 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 [3].

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 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. The specification of an action can combine Z schemas that specify data operations over the state and CSP constructs.

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

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

Figure 1: *Circus* specification of a *Timer* process

that are time bounded. It is not clear what is the practical use of non-terminating experiments (as opposed, of course, to non-terminating control systems themselves).

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 ($currentTime = tN$), 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 $currentTime = tN$ 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*.

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

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 ($\triangle$) 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 [9]. 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`,

`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

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.

Our model only captures the discrete observations of the simulation steps and their associated data exchanges. It is compatible with the view of a co-simulation as a sequence of discrete steps that define points for synchronisation and exchange of data.

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.

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.

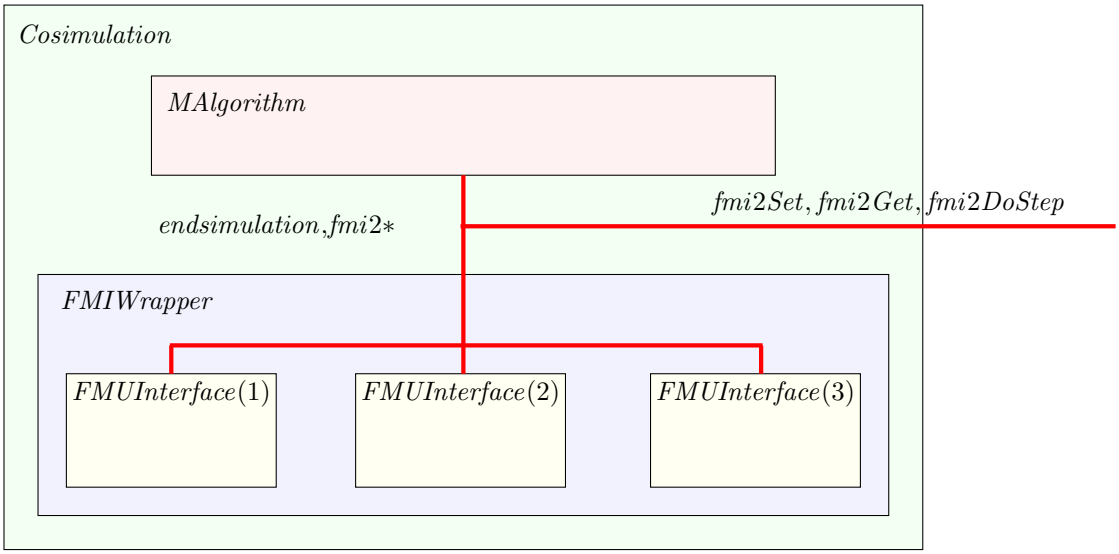


Figure 2: Structure of a co-simulation model

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 [5].

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

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 proposed *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. *FMUStatesManager* 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 Section 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*.

The structure of *Interaction* is the sequential composition of *Instantiation*, *InstantiationMode*, *InitializationMode*, and *slaveInitialized*, which correspond to states that define the stages of a co-simulation [13, 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 ($FMUs : \text{seq}\, FMI2COMP$), and sequences that define the parameters and their values ($parameters : \text{seq}(FMI2COMP \times VAR \times VAL)$), inputs and their initial values ($inputs : \text{seq}(FMI2COMP \times VAR \times VAL)$), outputs ($outputs : \text{seq}(FMI2COMP \times VAR)$), and an input/output port dependency

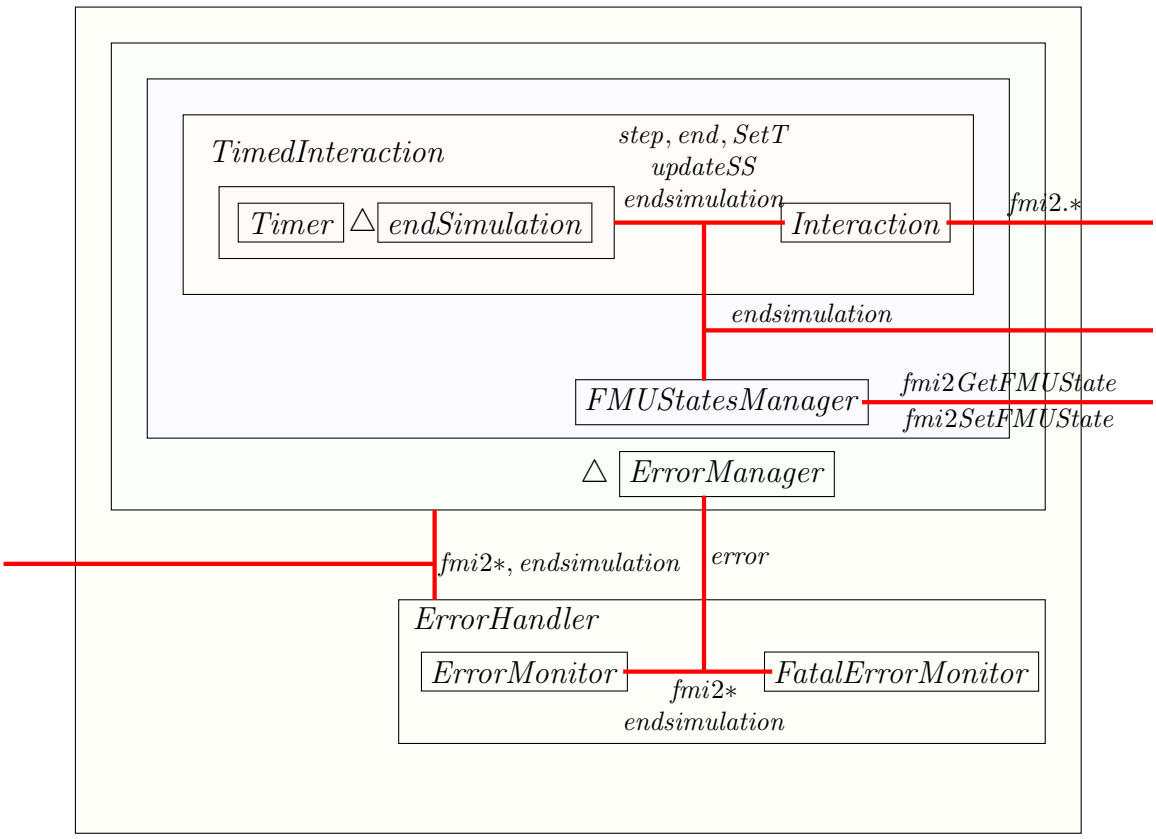


Figure 3: Structure of a model of a master algorithm

graph [5] *pdg*. Some of this information is also needed to generate automatically a sketch of the models of the FMUs (see Section 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.

$$; \ i : FMUs \bullet fmi2Instantiate.i?sc \longrightarrow \mathbf{Skip}$$

Our model is restrictive: it is valid to instantiate the FMUs in any order, and so we can have the events $fmi2Instantiate.i?sc$ in interleaving. Accommodating this flexibility is a simple change from an iterated sequence to an iterated interleaving, but our model captures good practice: handling all FMUs in a specific order.

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 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

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

Figure 4: Sketch of *slaveInitialized*

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

$$
\begin{array}{l}
\textbf{process}\ \ FMUStatesManager \mathrel{\widehat=} i : FMI2COMP \bullet \textbf{begin} \\
AllowAGet \mathrel{\widehat=} fmi2GetFMUState.i?s?st \longrightarrow AllowsGetsAndSets(s) \\
AllowsGetsAndSets \mathrel{\widehat=} s : FMUSTATE \bullet \\
\quad fmi2GetFMUState.i?t?st \longrightarrow AllowsGetsAndSets(t) \\
\quad \Box \\
\quad fmi2SetFMUState.i!s?st \longrightarrow AllowsGetsAndSets(s) \\
\bullet\ fmi2Instantiate.i?b \longrightarrow AllowAGet \\
\textbf{end}
\end{array}
$$

Figure 5: Model of *FMUStateManager*

FMU.

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

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

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 $RUN(FMUAPI(i))$, which allows the occurrence of any API functions, in any order.

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

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 [13, 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.

InstantiationMode terminates immediately if there has been an error. Otherwise, it permits the values of any variables of the FMU i to be set, and its state to be retrieved and set until a call is made to `fmi2SetUpExperiment`. After that, the action terminates, and the action *Instantiated* is called.We assume here that any FMU can have its internal state set and retrieved.

The actions *Instantiated*, *InitializationMode* and *slaveInitialized* are similar.

The interrupting action *Terminated* is triggered by $fmi2Terminate$, but it is available only if there has been no error. Otherwise, all actions, terminate. If the error was not fatal, $fmi2FreeInstance$ becomes the only available function.

$$
\begin{aligned}
&Terminated = \\
&\quad status \notin \{fmi2Error, fmi2Fatal\} \;\&\; fmi2Terminate.i?st \longrightarrow Remove \\
&\quad \Box \\
&\quad status = fmi2Error \;\&\; Remove
\end{aligned}
$$

The simple action *Remove* offers $fmi2FreeInstance$. The signal *endsimulation* terminates an FMU at any time.We are not imposing a restriction that after a non-fatal error the FMU must be terminated. We consider the possibility that the state of the FMU may have been retrieved and then reset.

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 $fmi2GetFMUState.i?st$ and $fmi2SetFMUState.i?st$. 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. This information is used to construct a master algorithm (see Section 4.1). Fig. 6 on 62 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 ($fmi2OK$). 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 $fmi2Instantiate.i?b \longrightarrow \mathbf{Skip}$. 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, [8] 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*.

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

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

If the state, either via setting of parameters and input or via an update, may become invalid, we can flag $fmi2Fatal$ and deadlock. For example, we consider the test case shown in Fig. 7 on 63 taken from [6]. It has been designed to show that 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.[1]

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

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 [21], so that it can be both model checked in FDR3 and executed in ProBe (FDR's process behaviour explorer), for suitably chosen model parameters.

[1] The full model is contained in Appendix C. The machine-readable code can be cut and pasted into a file for input to FDR 3.4.0.

5.1 Master algorithms

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

5.1.1 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.

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

Monitor ignores all flags *st* returned by *fmi2DoStep* except *fmi2Discard*. If this flag is returned, it queries the FMU using

$$fmi2GetBooleanStatusfmi2Terminated$$

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.

5.1.2 Simulink

This is a widely used tool for simulation based on control law diagrams [20]. A popular solver uses a variable-step policy based on change rate 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 action *VariableStepInteraction* refines

Interaction. We have also proved termination, deadlock freedom, and determinism.

5.1.3 Rollback

In the same way as illustrated by *VaryStep* in Fig. 8, we can model a sophisticated algorithm suggested in [5]. We define a *Rollback* monitor that has the same structure as *VaryStep*. Its *Monitor* (a) saves the state using *fmi*2*GetFMUState* 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 *fmi*2*SetFMUState*, and the time as well as the step size are adjusted (using *setT* and *updateSS*). We have again proved validity, termination, and determinism.

In [5], 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 [8], 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 of the FMU compositions in co-simulations. For this, we can use the FDR model checker [17] to check for the structural; properties of 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 [18], and in CCS, using temporal logic over actions [4].

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 [18, 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 $left1$ by a and passes the result to $X3$ on *mid*. $X3$ multiplies an input on the $left2$ 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 $left1$ and $left2$ channels right, otherwise a deadlock soon ensues.

The little network of processes is suitable for implementation as a composition of five FMUs (the three processes in Fig. 10 and a source and a sink).

The CSP specification of this network remembers the previous input.

$$DFProc(a, b) = left?x \longrightarrow P(x)$$
$$P(x) = left?y \longrightarrow right!(a * x + b * y) \longrightarrow P(y)$$

The key part of the main FMU in this specification is shown in Fig 11; (there is also a sink and a source).

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. This simple protocol could reflect more sophisticated behaviour, as required.

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. The CSP code for this problem is contained in Appendix B; it uses the master algorithm that can be found in Appendix A. 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 [7].

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 [5] using a proposed extension of the FMI. It uses API functions to get and set the state of an FMU. In [5], 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 [5].

There has been very practical work on new master algorithms, generation of FMUs and simulations, and hybrid models [2, 23, 12, 10]. Tripakis [26] shows how components with different underlying models (state machines, synchronous data flow, and so on) can be encoded as FMUs. Savicks [25] 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. within the Event-B platform Rodin It has been applied to an industrial case study [25]. It provides support for the simulation of FMUs within the Event-B platform Rodin, but it is does not wrap Event-B models as FMUs to enable their general FMI-compliant co-simulation.

Pre-dating FMI, the work in [16] 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 [6] 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 [14]. Using current support for *Circus* in Isabelle [15], 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.

7 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] Blochwitz, T., et al. The Functional Mockup Interface for Tool independent Exchange of Simulation Models. In: Proceedings of the 8th International Modelica Conference, 2011.

[4] 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)

[5] Broman, D., et al.: Determinate composition of FMUs for co-simulation. In: ACM SIGBED Intl Conf. on Embedded Software. IEEE (2013)

[6] Broman, D., et al.: Requirements for Hybrid Cosimulation Standards. In: 18th Intl Conf. on Hybrid Systems: Computation and Control. pp.179–188. ACM (2015)

[7] 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)

[8] 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)

[9] 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)

[10] Denil, J., et al.: Explicit semantic adaptation of hybrid formalisms for FMI co-simulation. In: Spring Simulation Multi-Conference (2015)

[11] Derler, P., Lee, E.A., Vincentelli, A.S.: Modeling cyber-physical systems. Procs of IEEE 100(1) (2012)

[12] Feldman, Y.A., Greenberg, L., Palachi, E.: Simulating Rhapsody SysML blocks in hybrid models with FMI. In: Modelica Conference (2014)

[13] FMI development group: Functional mock-up interface for model exchange and co-simulation, 2.0. `https://www.fmi-standard.org` (2014)

[14] Foster, S., et al.: Towards a UTP semantics for Modelica. In: Unifying Theories of Programming. LNCS, Springer (2016)

[15] 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)

[16] 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)

[17] 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)

[18] Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall (1985)

[19] Kübler, R., Schiehlen, W.: Two methods of simulator coupling. Mathematical and Computer Modelling of Dynamical Systems 6(2), 93–113 (2000)

[20] The MathWorks, Inc.: Simulink, `www.mathworks.com/products/-simulink`

[21] 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)

[22] 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)

[23] Pohlmann, U., et al.: Generating functional mockup units from software specifications. In: Modelica Conference (2012)

[24] Roscoe, A.W.: Understanding Concurrent Systems. Texts in Computer Science, Springer (2011)

[25] 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)

[26] 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)

[27] Woodcock, J.C.P., Davies, J.: Using Z—Specification, Refinement, and Proof. Prentice-Hall (1996)

A CSP Model of an FMI Master Algorithm

```
channel idle

IDLE = idle -> IDLE

--------------------
-- model of time --
--------------------
endOfTime        = 5
TIME             = { 0..endOfTime }
NZTIME           = diff(TIME,{0})
eps              = -1

---------------------
-- Model of values --
---------------------
topVal    = 8
VAL       = { -1..topVal }

---------------------------------------
-- Inputs to the translation process --
---------------------------------------
startTime        = 0
-- We do not support stopTimeDefined = false, either here or in the COE.
stopTimeDefined  = true
stopTime         = 5

-- State management
canGetAndSetFMUState(pdsgfmu1)         = false
canGetAndSetFMUState(pdsgfmu2)         = false
canGetAndSetFMUState(samplerfmu)       = false
canGetAndSetFMUState(checkequalityfmu) = false

-- Port indices, used to model variable names, in the context of each FMU.
INDEX            = { 1..5 }

-- Return type of FMI API functions
datatype FMI2STATUSFULL = fmi2OK | fmi2Discard | fmi2Error | fmi2Fatal

FMI2STATUS = {fmi2OK, fmi2Error, fmi2Fatal}

ErrorFlags = {fmi2Error,fmi2Fatal}

-----------------------
-- FMI API functions --
-----------------------
channel fmi2Get                            : FMI2COMPONENT . INDEX . VAL . FMI2STATUSFULL
channel fmi2Set                            : FMI2COMPONENT . INDEX . VAL . FMI2STATUS
-- The step size in an fmi2DoStep call cannot be 0.
channel fmi2DoStep                         : FMI2COMPONENT . TIME . NZTIME . FMI2STATUSFULL
channel fmi2Instantiate                    : FMI2COMPONENT . Bool
-- This function returns a component, and null if the instantiation
-- fails. We do not model pointers, so in our model we use a boolean
-- to cater for the possibility that instantiation may fail.
channel fmi2SetUpExperiment                : FMI2COMPONENT . TIME . Bool . TIME . FMI2STATUS
channel fmi2EnterInitializationMode        : FMI2COMPONENT . FMI2STATUS
channel fmi2ExitInitializationMode         : FMI2COMPONENT . FMI2STATUS
channel fmi2GetBooleanStatusfmi2Terminated : FMI2COMPONENT . Bool . FMI2STATUS
channel fmi2GetMaxStepSize                 : FMI2COMPONENT . NZTIME . FMI2STATUS
channel fmi2Terminate                      : FMI2COMPONENT . FMI2STATUS
channel fmi2FreeInstance                   : FMI2COMPONENT . FMI2STATUS
channel fmi2GetFMUState                    : FMI2COMPONENT -- . FMUSTATE
                                             . FMI2STATUS
channel fmi2SetFMUState                    : FMI2COMPONENT -- . FMUSTATE
                                             . FMI2STATUS

FMIAPI = {| fmi2Get,
            fmi2Set,
            fmi2DoStep,
            fmi2Instantiate,
            fmi2SetUpExperiment,
            fmi2EnterInitializationMode,
            fmi2ExitInitializationMode,
            fmi2GetBooleanStatusfmi2Terminated,
            fmi2GetMaxStepSize,
            fmi2Terminate,
            fmi2FreeInstance,
            fmi2GetFMUState,
            fmi2SetFMUState |}

----------------------------
```

```
-- Timer control channels --
----------------------------
channel end
channel step                        : TIME . NZTIME
channel setT                        : TIME
channel updateSS                    : NZTIME

-------------------------------------
-- Memory manager control channels --
-------------------------------------
channel get                         : FMI2COMPONENT . INDEX . VAL
channel set                         : FMI2COMPONENT . INDEX . VAL
channel getctime,getetime           : FMI2COMPONENT . TIME
channel setctime,setetime           : FMI2COMPONENT . TIME
channel startstep                   : FMI2COMPONENT
channel stopstep                    : FMI2COMPONENT

-----------------------------------------
-- Controller for FMU status channels --
-----------------------------------------
channel setstatus                   : FMI2COMPONENT . FMI2STATUSFULL
channel getstatus                   : FMI2COMPONENT . FMI2STATUSFULL
channel stepAnalysed
channel stepToComplete

-----------------------------------------------------------------
-- Control channel to shutdown the simulation.
-- We note that, since an FMU may fail, its termination may
-- not be carried out gracefully with fmi2Terminate and
-- fmi2FreeInstance.  This channel is used to indicate the
-- end of the experiment in all cases and shutdown the
-- model processes.
-----------------------------------------------------------------
channel endsimulation

-----------------------------------------------------------
-- If any of the API functions returns an error, no further
-- calls to API functions should take place. This is ensured
-- by flagging the error via fatalError or error.
-----------------------------------------------------------
channel fatalError
channel error

----------------------
-- Master Algorithm --
----------------------
-- This is a general characterisation of the valid history of interactions
-- traces of a master algorithm. It does not commit to specific policies to
-- define step size and error treatment, for example.
MAlgorithm(t0,tN) =
  ( ( ( TimedInteractions(t0,tN)
          [|{|endsimulation,fmi2Instantiate,fmi2SetFMUState,fmi2GetFMUState|}|]
        AllFMUStatesManager)
      /\ ErrorManager )
        [| union(FMIAPI,{endsimulation,error,fatalError}) |]
      ErrorHandler)
    \ {error,fatalError}

TimedInteractions(t0,tN) =
  ( (Timer(t0,1,tN) /\ endSimulation) [| {|step,end,setT,updateSS,endsimulation|} |] Interaction)
    \ {|step,end,setT,updateSS,stepToComplete|}

ErrorManager = FatalError [] ErrorManagement

ErrorHandler = ErrorMonitor [| union(FMIAPI,{endsimulation}) |] FatalErrorMonitor

-- stepToComplete should not be in the specification. It is there because
-- Interaction is used to specify other algorithms.

endSimulation = endsimulation -> SKIP

-- The state manager is kept as a separate component, because it is
-- the component that raises difficulties with model checking.

FixedStepNoRollbackMAlgorithm(t0,hc,tN) =
  ( ( ( FixedStepTimedInteractions(t0,hc,tN)
          [|{|endsimulation,fmi2Instantiate,fmi2SetFMUState,fmi2GetFMUState|}|]
        ExampleStateManager)
      /\ ErrorManager )
        [| union(FMIAPI,{endsimulation,error,fatalError}) |]
      ErrorHandler)
    \ {error,fatalError}
```

```
FixedStepTimedInteractions(t0,hc,tN) =
  ( (FixedStepTimerNoRollBack(t0,hc,tN) /\ endSimulation)
      [| {|step,end,setT,updateSS,endsimulation|} |]
    FixedStepInteraction(hc) )
    \ {|step,end,setT,updateSS,stepAnalysed,stepToComplete|}

-- The refinement below follows by monotonicity of
-- the refinements proved in the sequel.
-- assert MAlgorithm(0,endOfTime) [T= FixedStepNoRollbackMAlgorithm(0,2,endOfTime)

-- assert FixedStepNoRollbackMAlgorithm(0,2,endOfTime); IDLE :[deadlock free]

-- assert FixedStepNoRollbackMAlgorithm(0,2,endOfTime) :[deterministic]

StandardFixedStepNoRollbackMAlgorithm(t0,hc,tN) =
  FixedStepTimedInteractions(t0,hc,tN)
    [|{|endsimulation,fmi2Instantiate,fmi2SetFMUState,fmi2GetFMUState|}|]
  ExampleStateManager

-- The refinement below follows by monotonicity of
-- the refinements proved in the sequel.
-- assert MAlgorithm(0,endOfTime) [T= StandardFixedStepNoRollbackMAlgorithm(0,2,endOfTime)

-- assert StandardFixedStepNoRollbackMAlgorithm(0,2,endOfTime); IDLE :[deadlock free]

-- assert StandardFixedStepNoRollbackMAlgorithm(0,2,endOfTime) :[deterministic]

VariableStepNoRollbackMAlgorithm(t0,tN) =
  ( ( ( VariableStepNoRollbackTimedInteractions(t0,tN)
          [|{|endsimulation,fmi2Instantiate,fmi2SetFMUState,fmi2GetFMUState|}|]
        ExampleStateManager)
      /\ ErrorManager )
        [| union(FMIAPI,{endsimulation,error,fatalError}) |]
      ErrorHandler)
    \ {error,fatalError}

VariableStepNoRollbackTimedInteractions(t0,tN) =
  ( (VariableStepTimerNoRollBack(t0,2,tN) /\ endSimulation)
      [| {|step,end,setT,updateSS,endsimulation|} |]
    VariableStepNoRollbackInteraction )
    \ {|step,end,setT,updateSS,stepAnalysed,stepToComplete|}

-- The refinement below follows by monotonicity of
-- the refinements proved in the sequel.
-- assert MAlgorithm(0,endOfTime) [T= VariableStepWithRollbackMAlgorithm(0,2,endOfTime)

-- assert VariableStepNoRollbackMAlgorithm(0,endOfTime); IDLE :[deadlock free]

-- assert VariableStepNoRollbackMAlgorithm(0,endOfTime) :[deterministic]

VariableStepWithRollbackMAlgorithm(t0,hc,tN) =
  ( ( ( VariableStepWithRollbackTimedInteractions(t0,hc,tN)
          [|{|endsimulation,fmi2Instantiate,fmi2SetFMUState,fmi2GetFMUState|}|]
        AllFMUStatesManager)
      /\ ErrorManager )
        [| union(FMIAPI,{endsimulation,error,fatalError}) |]
      ErrorHandler)
    \ {error,fatalError}

VariableStepWithRollbackTimedInteractions(t0,hc,tN) =
  ( (VariableStepTimerWithRollBack(t0,t0,2,tN) /\ endSimulation)
      [| {|step,end,setT,updateSS,endsimulation|} |]
    VariableStepWithRollbackInteraction(hc) )
    \ {|step,end,setT,updateSS,stepAnalysed,stepToComplete|}

-- The refinement below follows by monotonicity of
-- the refinements proved in the sequel.
-- assert MAlgorithm(0,endOfTime) [T= VariableStepWithRollbackMAlgorithm(0,2,endOfTime)

-- assert VariableStepWithRollbackMAlgorithm(0,2,endOfTime); IDLE :[deadlock free]

-- assert VariableStepWithRollbackMAlgorithm(0,2,endOfTime) :[deterministic]

--------------------
-- General timer --
--------------------
-- It allows roolbacks (setT), variable step size (updateSS),
-- as well as indicating the steps (step) and the end
-- (end) of the simulation
Timer(ct,hc,tN) =
  let
    T(t,ss) =
      setT?t: { vt | vt <- TIME, vt <= tN} -> T(t,ss)
```

```
      []
      updateSS?nhc: NZTIME -> T(t,nhc)
      []
      step!t!ss -> T(min(t+ss,tN),ss)
      []
      t == tN & end -> STOP
    within
      T(ct,hc)

-------------------------
-- Simple example timer --
-------------------------
FixedStepTimerNoRollBack(t0,hc,tN) =
  let FSTNRB(t) =
    if t <= tN then
      step.t.hc -> FSTNRB(t+hc)
    else
      end -> STOP
  within
    FSTNRB(t0)

-- We use the general components to give trace specifications to models
-- of a particular algorithm.
-- assert Timer(0,2,endOfTime) [T= FixedStepTimerNoRollBack(0,2,endOfTime)

--------------------
-- Simulink timer --
--------------------
VariableStepTimerNoRollBack(t0,hc,tN) =
  let
    VSTNRB(t,ss) =
      if t <= tN then
        step.t.ss -> VSTNRB(t+ss,ss)
        []
        updateSS?nhc: NZTIME -> VSTNRB(t,nhc)
      else
        end -> STOP
  within
    VSTNRB(t0,hc)

-- assert Timer(0,2,endOfTime) [T= VariableStepTimerNoRollBack(0,2,endOfTime)

------------------
-- Broman timer --
------------------
VariableStepTimerWithRollBack(t0,pt,hc,tN) =
  let VSTWRB(t,p,ss) =
      if t <= tN then
        step.t.ss -> VSTWRB(t+ss,t,ss)
        []
        updateSS?nhc: NZTIME -> VSTWRB(t,p,nhc)
        []
        setT.p -> VSTWRB(p,p,ss)
      else
        end -> STOP
  within
    VSTWRB(t0,pt,hc)

-- assert Timer(0,2,endOfTime) [T= VariableStepTimerWithRollBack(0,0,2,endOfTime)

-----------------------------------------------------------
-- General interaction pattern for a master algorithm --
-----------------------------------------------------------
-- This pattern can be autoamtically generated for a
-- given set of FMUs, as indicated by the use of the
-- inputs to such a procedure that we instantiate above
-- for a particular example.

LifeCycle =
  let
    Instantiation =
      ; i: FMI2COMPONENTseq @ fmi2Instantiate.i?sc -> SKIP

    InstantiationMode(params) =
      if params == <> then
        ( ( ; i : FMI2COMPONENTseq @
      fmi2SetUpExperiment!i!startTime!stopTimeDefined!stopTime?st ->
      setstatus.i!st -> SKIP ) ;
          ( ; i : FMI2COMPONENTseq @
      fmi2EnterInitializationMode.i?st ->
              setstatus.i!st -> SKIP ) )
      else
        let (i,x,v) = head(params) within
```

```
  fmi2Set!i!x!v?st -> setstatus.i!st -> InstantiationMode(tail(params))

    InitializationMode(inits) =
      if inits == <> then
        ( ; i : FMI2COMPONENTseq @
      fmi2ExitInitializationMode!i?st -> setstatus.i!st -> SKIP )
      else
        let (i,x,v) = head(inits) within
  fmi2Set!i!x!v?st -> setstatus.i!st -> InitializationMode(tail(inits))

    slaveInitialized =
      end -> Terminated
      []
      step?t?hc -> TakeOutputs(outputs,<>,t,hc)

    TakeOutputs(outs,vals,t,hc) =
      let
        TO(os,vs) =
          if os == <> then
            DistributeInputs(portdeps,vs,t,hc)
          else
            let (i,n) = head(os) within
              fmi2Get.i.n?v?st -> TO(tail(os),vs^<v>)
      within
        TO(outs,vals)

    DistributeInputs(pds,vals,t,hc) =
      let
        DI(ps) =
          if ps == <> then
            Step(t,hc)
          else
            let (pos,(i,n)) = head(ps) within
           fmi2Set.i.n!index(vals,pos)?st -> setstatus.i!st ->
             DI(tail(ps))
      within
        DI(pds)


    Step(t,hc) = let
--      -- The guard is just to FDR'2 benefit.
--      -- The process cannot reach this stage with hc == 0.
--      -- See below for a proof
--      hc != 0 & (
--      stepToComplete ->
--        ( ; i: FMI2COMPONENTseq @
--            ( ( fmi2DoStep.i.t.hc?st -> setstatus.i!st -> SKIP;
--
--                )
--          )
--    ) ;
--        -- The above can be proved by commenting in the following lines.
--        -- []
--        -- hc == 0 & printme.10 -> STOP
      Iterations(i) = hc != 0 & (
        if i == 0 then
          stepToComplete -> fmi2DoStep.1.t.hc?st -> setstatus.1!st -> Iterations(1)
        else if i < numfmus then
         (fmi2GetBooleanStatusfmi2Terminated.i?b?st -> Iterations(i)
          []
          fmi2GetMaxStepSize.i?t?st -> Iterations(i)
          []
          fmi2DoStep.(i+1).t.hc?st -> setstatus.(i+1)!st -> Iterations(i+1))
        else (fmi2GetBooleanStatusfmi2Terminated.i?b?st -> Iterations(i)
              []
              fmi2GetMaxStepSize.i?t?st -> Iterations(i)
              []
              stepAnalysed -> SKIP)
        )
      within
        Iterations(0);
-- JIM: This is too specific.
        -- (getstatus.1?st1 -> getstatus.2?st2 -> getstatus.3?st3 -> getstatus.4?st4 ->
        --   if st1 == fmi2OK and st2 == fmi2OK and st3 == fmi2OK and st4 == fmi2OK then
        --     NextStep
        (getstatus.1?st1 -> getstatus.2?st2 -> getstatus.3?st3 ->
          if st1 == fmi2OK and st2 == fmi2OK and st3 == fmi2OK then
            NextStep
          else -- If there is a fatal error, it is blocked by the StatusMonitor.
            -- In the presence of fmi2Discard or fmi2Error, we may continue.
            -- In fact, in the current version of this model, an error leads to a
            -- cancellation of the co-simulation. This is managed by the StatusMonitor.
            -- NextStep []
```

```
          Terminated)

-- assert RUN(diff(Events,{|printme|})) [T= FixedStepNoRollbackMAlgorithm(0,2,endOfTime); IDLE

    -- This is used to ensure that fmi2GetBooleanStatusfmi2Terminated and
    -- fmi2GetMaxStepSize can only happen afer an fmi2DoStep.
--     StepManagement =
--       let
--         BeforeStep = fmi2DoStep?i?t?hc?st -> InStep(i)
--         -- There may be other API functions that can take place here.
--         InStep(i) = fmi2GetBooleanStatusfmi2Terminated.i?b?st -> InStep(i)
--                     []
--                     fmi2GetMaxStepSize.i?t?st -> InStep(i)
--                     []
--                     ([] e: {|step,updateSS,setT,end|} @ e -> BeforeStep) --  step?t?hc:{ss | ss <- TIME, ss!= 0} -> BeforeStep
--                     []
--                     fmi2DoStep?i?t?hc?st -> InStep(i)
--       within
--         step?t?hc:{ss | ss <- TIME, ss!= 0} -> BeforeStep

    NextStep = updateSS?d -> NextStep
               []
               setT?t -> NextStep
               []
               slaveInitialized

    Terminated =
      ( ; i: FMI2COMPONENTseq @
            fmi2Terminate.i?st -> setstatus.i!st ->
            fmi2FreeInstance.i?st -> setstatus.i!st -> SKIP );
        endsimulation -> SKIP

  within
    Instantiation ; InstantiationMode(parameterValues); InitializationMode(initialValues); slaveInitialized

-- This is needed just because LifeCycle uses the status to decide whether to proceed to the next step or not.
StatusStore(i,s) =
  setstatus.i?st -> StatusStore(i,st) -- (if less(st,s) then StatusStore(st) else StatusStore(st))
  []
  getstatus.i!s -> StatusStore(i,s)

FatalErrorMonitor = let

    -- The parameter here is duplicating the information that is in the StatusStore.
    Monitor(st) =
      fmi2Get?i?n?v?st -> StopFatal(st)
      []
      fmi2Set?i?n?v?st -> StopFatal(st)
      []
      fmi2GetFMUState?i?st -> StopFatal(st) -- Monitor -- ?s?st -> StopFatal(st)
      []
      fmi2SetFMUState?i?st -> StopFatal(st) -- Monitor -- ?s?st -> StopFatal(st)
      []
      fmi2SetUpExperiment?i?t?b?hc?st -> StopFatal(st)
      []
      fmi2EnterInitializationMode?i?st -> StopFatal(st)
      []
      fmi2ExitInitializationMode?i?st -> StopFatal(st)
      []
      fmi2GetBooleanStatusfmi2Terminated?i?b?st -> StopFatal(st)
      []
      fmi2DoStep?i?t?hc?st -> StopFatal(st)
      []
      fmi2Terminate?i?st -> StopFatal(st)
      []
      fmi2GetMaxStepSize?i?t?st -> StopFatal(st)
      []
      fmi2Instantiate?i?b -> (if b == true then StopFatal(fmi2OK) else StopFatal(fmi2Fatal))
      []
      fmi2FreeInstance?i?st -> StopFatal(st)

      StopFatal(st) = st == fmi2Fatal & fatalError -> STOP []  st != fmi2Fatal & Monitor(st)

  within
      Monitor(fmi2OK) /\ endsimulation -> SKIP

FatalError = fatalError -> endsimulation -> SKIP

    -- We are not treating the occurrence of fmi2Error as recoverable. So, there is no possibility
    -- of resetting the FMU and continuing.
ErrorMonitor = let

  -- The parameter here is duplicating the information that is in the StatusStore.
```

```
  Monitor(st) =
    fmi2Get?i?n?v?st -> StopError(st)
    []
    fmi2Set?i?n?v?st -> StopError(st)
    []
    fmi2GetFMUState?i?st -> StopError(st) -- Monitor -- ?s?st -> StopError(st)
    []
    fmi2SetFMUState?i?st -> StopError(st) -- Monitor -- ?s?st -> StopError(st)
    []
    fmi2SetUpExperiment?i?t?b?hc?st -> StopError(st)
    []
    fmi2EnterInitializationMode?i?st -> StopError(st)
    []
    fmi2ExitInitializationMode?i?st -> StopError(st)
    []
    fmi2GetBooleanStatusfmi2Terminated?i?b?st -> StopError(st)
    []
    fmi2DoStep?i?t?hc?st -> StopError(st)
    []
    fmi2Terminate?i?st -> StopError(st)
    []
    fmi2GetMaxStepSize?i?t?st -> StopError(st)
    []
    fmi2Instantiate?i?b -> (if b == true then StopError(fmi2OK) else StopError(fmi2Fatal))
    []
    fmi2FreeInstance?i?st -> StopError(st)

    StopError(st) = st == fmi2Error & error -> Monitor(st) [] st != fmi2Error & Monitor(st)

within
    Monitor(fmi2OK) /\ endsimulation -> SKIP

ErrorManagement = let
  Shutdown(i) = fmi2Instantiate.i?b -> ShutdownCreated(i); endsimulation -> SKIP
                []
                error -> endsimulation -> SKIP

  ShutdownCreated(i) = error -> fmi2FreeInstance.i?st -> SKIP
                       []
                       fmi2FreeInstance.i?st -> SKIP

within
 (|| i: FMI2COMPONENT @ [{error,endsimulation}] Shutdown(i))

Interaction =
    (LifeCycle
        [|{|setstatus,getstatus,endsimulation|}|]
      ((||| i: FMI2COMPONENT @ StatusStore(i,fmi2OK)) /\ endSimulation)
    ) \ {|setstatus,getstatus,error,fatalError|}

-------------------------------------------------
-- Interaction patterns for a master algorithm --
-------------------------------------------------
-----------------------------
-- Example in the standard --
-----------------------------

PossibleTerminationOnDiscard =
  Interaction
    [| {| fmi2DoStep, fmi2GetBooleanStatusfmi2Terminated, stepAnalysed, endsimulation |} |]
  DiscardTreatment

StandardPossibleTerminationOnDiscard =
  (LifeCycle [|{|setstatus,getstatus,endsimulation|}|] ((||| i: FMI2COMPONENT @ StatusStore(i,fmi2OK)) /\ endSimulation))
    \ {|setstatus,getstatus|}
    [| {| fmi2DoStep, fmi2GetBooleanStatusfmi2Terminated, stepAnalysed, endsimulation |} |]
  DiscardTreatment

-- The interest in stepAnalysed ensures that the FMU step cannot terminate
-- before its fmi2GetBooleanStatusfmi2Terminated is requested.
DiscardTreatment =
  let
    ErrorTreatment =
      fmi2DoStep?i?t?hc?st ->
        (if st == fmi2Discard then
          fmi2GetBooleanStatusfmi2Terminated.i?b?st0m ->
            ( if b == true
              then ErrorTreatment -- Although the algorithm checks this, it ignores that status until the next step.
              else ErrorTreatment )
        else
          ErrorTreatment)
      []
      stepAnalysed -> ErrorTreatment
```

```
  within
   ErrorTreatment /\ endsimulation -> SKIP

FixedStep = endsimulation -> SKIP

FixedStepInteraction(hc) =
  PossibleTerminationOnDiscard
    [ union(union(diff(FMIAPI,{|fmi2GetMaxStepSize|}),{endsimulation,end,stepAnalysed,stepToComplete}),{step.t.hc | t <- TIME})
      ||
    { endsimulation } ]
  -- This is here just to mimic the general shape of the definition of an interaction
  -- (and to allow us to cut down the behaviour of PossibleTerminationOnDiscard).
  FixedStep

-- assert Interaction [T= FixedStepInteraction(2)

StandardFixedStepInteraction(hc) =
  StandardPossibleTerminationOnDiscard
    [ union(union(diff(FMIAPI,{|fmi2GetMaxStepSize|}),{endsimulation,end,stepAnalysed,stepToComplete}),{step.t.hc | t <- TIME})
      ||
    { endsimulation } ]
  -- This is here just to mimic the general shape of the definition of an interaction
  -- (and to allow us to cut down the behaviour of PossibleTerminationOnDiscard).
  FixedStep

-- assert Interaction [T= StandardFixedStepInteraction(2)

------------------------
-- Simulink algorithm --
------------------------
-- The synchronisation between Interaction and VaryStep on updateSS
-- ensures that the update can only take place at the right point
-- of the loop.
VariableStepNoRollbackInteraction =
  PossibleTerminationOnDiscard
    [ union(diff(FMIAPI,{|fmi2GetMaxStepSize|}),{|endsimulation,end,stepAnalysed,stepToComplete,step,updateSS|})
       ||
    {| fmi2Get,updateSS,endsimulation |} ]
  VaryStep

channel delta: {(-10)..10}

-- This can be modelled much more generally and elegantly in Circus.
-- The initial step size is defined in the instantiation of the right timer.
VaryStep =
  let
    threshold = 5

    Monitor(y1,y2,y3) =
     (fmi2Get.1.1?ny1?st -> delta!(y1 - ny1) -> SKIP)
     |||
     (fmi2Get.2.1?ny2?st -> delta!(y2 - ny2) -> SKIP)
     |||
     (fmi2Get.3.3?ny3?st -> delta!(y3 - ny3) -> SKIP)

    Decide = delta?d1 -> delta?d2 -> delta?d3 ->
      if (d1 != eps and (d1 >= threshold or d1 <= -threshold)) and
         (d2 != eps and (d2 >= threshold or d2 <= -threshold)) and
         (d1 != eps and (d3 >= threshold or d3 <= -threshold))
      then updateSS!1 -> SKIP
      else SKIP

      Step = (Monitor(eps,eps,eps) [| {| delta |} |] Decide) \ {| delta |}; Step
  within
      Step /\ endsimulation -> SKIP

-- assert Interaction [T= VariableStepNoRollbackInteraction

----------------------
-- Broman algorithm --
----------------------
VariableStepWithRollbackInteraction(hcmax) =
  Interaction
    [ union(diff(FMIAPI,{|fmi2GetFMUState,fmi2SetFMUState|}),{|endsimulation,end,step,stepAnalysed,stepToComplete,setT,updateSS|})
       ||
    {| fmi2DoStep,fmi2GetBooleanStatusfmi2Terminated,fmi2GetMaxStepSize,
       fmi2GetFMUState,fmi2SetFMUState,
       step,stepAnalysed,stepToComplete,setT,updateSS,endsimulation |} ]
  Rollback(hcmax)

channel hcmin: TIME
```

```
Rollback(hcmax) =
  let
    -- In Circus, this can be much more elegant.
    Monitor =
      -- Synchronisation on step, as well as stepToComplete, ensures that a new step cannot start
      -- before the management required here is completed.
      step?t?hc -> stepToComplete ->
        (; i: FMI2COMPONENTseq @ fmi2GetFMUState.i?st -> SKIP); -- ?s?st -> SKIP) ;
        (; i: FMI2COMPONENTseq @ fmi2DoStep?i?t.hcmax?st ->  fmi2GetMaxStepSize.i?hc?st -> hcmin!hc -> SKIP)

    Decide(hcmax) =
      hcmin?h1 -> hcmin?h2 -> hcmin?h3 -> hcmin?h4 ->
        let
          newhc = min4(h1,h2,h3,h4)
        within
          if 0 < newhc and newhc < hcmax
          then (; i: FMI2COMPONENTseq @ fmi2SetFMUState.i?st -> SKIP); -- ?s?st -> SKIP);
               stepAnalysed -> setT?t -> updateSS!newhc -> SKIP
          else stepAnalysed -> SKIP

      Step = (Monitor [| {| hcmin |} |] Decide(hcmax)) \ {| hcmin |}; Step
  within
      Step /\ endsimulation -> SKIP

-- The need to hide fmi2GetFMUState,fmi2SetFMUState arises from the artificial
-- separation of the management of state, to allow model checking.
-- assert Interaction [T= (VariableStepWithRollbackInteraction(2)) \ {| fmi2GetFMUState,fmi2SetFMUState |}

-----------------------------------------------
-- General state manager for a master algorithm --
-----------------------------------------------
-- Once an FMU is instantiated, then it is possible to retrieve the
-- state. After that, both gets and sets are allowed. Error management
-- is handled by Interaction.
FMUStateManager(i) =
  let
    AllowAGet =
      fmi2GetFMUState.i?st -- ?s?st
      -> AllowsGetsAndSets -- (s)

    AllowsGetsAndSets = -- (s) =
      fmi2GetFMUState.i?st -- ?t?st
      -> AllowsGetsAndSets -- (t)
      []
      fmi2SetFMUState.i?st -- !s?st
      -> AllowsGetsAndSets -- (s)
  within
    fmi2Instantiate.i?b -> AllowAGet

AllFMUStatesManager = ( ||| i : FMI2COMPONENT @ FMUStateManager(i) ) /\ endSimulation

-----------------------------------------------
-- Simple state manager for a master algorithm --
-----------------------------------------------
-- This is a manager that should be adequate when
-- canGetAndSetFMUState(i) is false for all i.
ExampleStateManager = ( ||| i : FMI2COMPONENT @ fmi2Instantiate.i?b -> STOP ) /\ endsimulation -> SKIP

-- We cannot model check the stateful version of the algorithms, but with the state
-- actually commented out, it passes.
-- assert AllFMUStatesManager [T= ExampleStateManager

-----------------------------------------------
-- State manager for Broman's master algorithm --
-----------------------------------------------
BromanAllFMUStatesManager = AllFMUStatesManager

-- The refinement below is established by monotonicity.
-- assert FMIWrapper(0,endOfTime) [T= ExampleFMIWrapper(0,endOfTime)

---------------------------------
-- General behaviour of an FMU --
---------------------------------
FMUInterface(i) =
  let
    Instantiation =
      fmi2Instantiate.i?b -> (b != false & SKIP [] b == false & setstatus.i!fmi2Fatal -> SKIP)

    InstantiationMode = getstatus.i?st -> (
      member(st,ErrorFlags) & SKIP
      []
      not member(st,ErrorFlags) & (
        fmi2SetUpExperiment!i?starttime?stoptimedefined?stoptime?st -> SKIP
```

```
      []
      fmi2Set!i?x?v?st -> setstatus.i!st -> InstantiationMode
      []
      fmi2GetFMUState.i?st -> setstatus.i!st -> InstantiationMode
      []
      fmi2SetFMUState.i?st -> setstatus.i!st -> InstantiationMode))

  Instantiated = getstatus.i?st -> (
    member(st,ErrorFlags) & SKIP
    []
    not member(st,ErrorFlags) & (
      fmi2EnterInitializationMode.i?st -> not member(st,ErrorFlags) &  SKIP
      []
      fmi2Set!i?x?v?st -> setstatus.i!st -> Instantiated
      []
      fmi2GetFMUState.i?st -> setstatus.i!st -> Instantiated
      []
      fmi2SetFMUState.i?st -> setstatus.i!st -> Instantiated))

  InitializationMode = getstatus.i?st -> (
    member(st,ErrorFlags) & SKIP
    []
    not member(st,ErrorFlags) & (
      fmi2ExitInitializationMode.i?st -> setstatus.i!st -> SKIP
      []
      fmi2Set.i?x?v?st -> setstatus.i!st -> InitializationMode
      []
      fmi2GetFMUState.i?st -> setstatus.i!st -> InitializationMode
      []
      fmi2SetFMUState.i?st -> setstatus.i!st -> InitializationMode))

  -- While a pattern should be defined by a master algorithm,
  -- where all output are obtained before the inputs are
  -- distributed, the FMU is passive and does not impose
  -- such a policy on its use.
  slaveInitialized = getstatus.i?st -> (
    member(st,ErrorFlags) & SKIP
    []
    not member(st,ErrorFlags) & (
      fmi2Get.i?n?v?st -> setstatus.i!st -> slaveInitialized
      []
      fmi2Set.i?n?v?st -> setstatus.i!st -> slaveInitialized
      []
      fmi2DoStep.i?t?hc?st -> setstatus.i!st -> slaveInitialized
      []
      fmi2GetMaxStepSize.i?hc?st -> setstatus.i!st -> slaveInitialized
      []
      fmi2GetBooleanStatusfmi2Terminated.i?b?st -> setstatus.i!st -> slaveInitialized
      []
      fmi2GetFMUState.i?st -> setstatus.i!st -> slaveInitialized
      []
      fmi2SetFMUState.i?st -> setstatus.i!st -> slaveInitialized))

  Terminated =
    getstatus.i?st : diff(FMI2STATUS,{fmi2Error,fmi2Fatal}) -> (
      fmi2Terminate.i?st -> Remove
      []
      fmi2GetFMUState.i?st -> setstatus.i!st -> Terminated
      []
      fmi2SetFMUState.i?st -> setstatus.i!st -> Terminated)
    []
    getstatus.i.fmi2Error ->  Remove

  Remove = getstatus.i?st -> (
    st == fmi2Fatal & STOP
    []
    st != fmi2Fatal & fmi2FreeInstance.i?st -> STOP
    []
    st != fmi2Fatal & fmi2GetFMUState.i?st -> setstatus.i!st -> Remove
    []
    st != fmi2Fatal & fmi2SetFMUState.i?st -> setstatus.i!st -> Remove)

  Status(st) = setstatus.i?nst -> Status(nst)
               []
               getstatus.i.st -> Status(st)
within
  (((Instantiation;
   InstantiationMode ;
   Instantiated;
   InitializationMode ;
   slaveInitialized)
   /\
    ( Terminated /\ endSimulation )) [| {|getstatus,setstatus,endsimulation |} |] (Status(fmi2OK) /\ endSimulation )) \ {|getstatus,setstatus|}
```

```
------------------
-- Cosimulation --
------------------
Cosimulation(t0,tN) =
  ( MAlgorithm(t0,tN)
    [| union(FMIAPI,{endsimulation}) |]
    FMIWrapper(t0,tN) ) \ {|fmi2Instantiate,
                fmi2SetUpExperiment,
                fmi2EnterInitializationMode,
                            fmi2ExitInitializationMode,
                            fmi2GetBooleanStatusfmi2Terminated,
                            fmi2GetMaxStepSize,
                            fmi2Terminate,
                            fmi2FreeInstance,
                fmi2GetFMUState,
                            fmi2SetFMUState,
                            endsimulation|}

-- Refinement is established by monotonicity
FixedStepNoRollbackCosimulation(t0,hc,tN) =
  ( FixedStepNoRollbackMAlgorithm(t0,hc,tN)
    [| union(FMIAPI,{endsimulation}) |]
    ExampleFMIWrapper(t0,tN) ) \ {|fmi2Instantiate,
                        fmi2SetUpExperiment,
                        fmi2EnterInitializationMode,
                                    fmi2ExitInitializationMode,
                                    fmi2GetBooleanStatusfmi2Terminated,
                                    fmi2GetMaxStepSize,
                                    fmi2Terminate,
                                    fmi2FreeInstance,
                        fmi2GetFMUState,
                                    fmi2SetFMUState,
                                    endsimulation|}

-- assert FixedStepNoRollbackCosimulation(0,2,endOfTime) :[deterministic]

-- assert FixedStepNoRollbackCosimulation(0,2,endOfTime) ; IDLE :[deadlock free]

-- assert FixedStepNoRollbackCosimulation(0,2,endOfTime) :[livelock free]

VariableStepNoRollbackCosimulation(t0,tN) =
  (VariableStepNoRollbackMAlgorithm(t0,tN)
    [|union(FMIAPI,{endsimulation})|]
    ExampleFMIWrapper(t0,tN) ) \ {|fmi2Instantiate,
                        fmi2SetUpExperiment,
                        fmi2EnterInitializationMode,
                                    fmi2ExitInitializationMode,
                                    fmi2GetBooleanStatusfmi2Terminated,
                                    fmi2GetMaxStepSize,
                                    fmi2Terminate,
                                    fmi2FreeInstance,
                        fmi2GetFMUState,
                                    fmi2SetFMUState,
                                    endsimulation|}

-- assert VariableStepNoRollbackCosimulation(0,endOfTime) :[deterministic]

-- assert VariableStepNoRollbackCosimulation(0,endOfTime) ; IDLE :[deadlock free]

-- assert VariableStepNoRollbackCosimulation(0,endOfTime) :[livelock free]

VariableStepWithRollbackCosimulation(t0,hc,tN) =
  (VariableStepWithRollbackMAlgorithm(t0,hc,tN)
    [|union(FMIAPI,{endsimulation})|]
    ExampleFMIWrapper(t0,tN) ) \ {|fmi2Instantiate,
                        fmi2SetUpExperiment,
                        fmi2EnterInitializationMode,
                                    fmi2ExitInitializationMode,
                                    fmi2GetBooleanStatusfmi2Terminated,
                                    fmi2GetMaxStepSize,
                                    fmi2Terminate,
                                    fmi2FreeInstance,
                        fmi2GetFMUState,
                                    fmi2SetFMUState,
                                    endsimulation|}

-- assert VariableStepWithRollbackCosimulation(0,2,endOfTime) :[deterministic]

-- assert VariableStepWithRollbackCosimulation(0,2,endOfTime) ; IDLE :[deadlock free]

-- assert VariableStepWithRollbackCosimulation(0,2,endOfTime) :[livelock free]
```

```
--------------
-- Scenario --
--------------

SynchronousEventsSpec =
  AllInInternalChoice({fmi2Set.1.1.1.fmi2OK,fmi2Set.1.2.1.fmi2OK,fmi2Set.2.1.1.fmi2OK,fmi2Set.2.2.2.fmi2OK}) ;
  AllInInternalChoice({fmi2Set.3.1.1.fmi2OK,fmi2Set.3.2.1.fmi2OK,fmi2Set.4.1.1.fmi2OK,fmi2Set.4.2.1.fmi2OK}) ;
  AllInInternalChoice({fmi2Get.1.1.1.fmi2OK,fmi2Get.2.1.1.fmi2OK,fmi2Get.3.3.1.fmi2OK}) ;
  AllInInternalChoice({fmi2Set.3.1.1.fmi2OK,fmi2Set.3.2.1.fmi2OK,fmi2Set.4.1.1.fmi2OK,fmi2Set.4.2.1.fmi2OK}) ;
  (fmi2DoStep.1.0.2.fmi2OK -> SKIP) ;
  (fmi2DoStep.2.0.2.fmi2OK -> SKIP) ;
  (fmi2DoStep.3.0.2.fmi2OK -> SKIP) ;
  (fmi2DoStep.4.0.2.fmi2OK -> SKIP) ;
  AllInInternalChoice({fmi2Get.1.1.1.fmi2OK,fmi2Get.2.1.1.fmi2OK,fmi2Get.3.3.1.fmi2OK}) ;
  AllInInternalChoice({fmi2Set.3.1.1.fmi2OK,fmi2Set.3.2.1.fmi2OK,fmi2Set.4.1.1.fmi2OK,fmi2Set.4.2.1.fmi2OK}) ;
  (fmi2DoStep.1.2.2.fmi2OK -> SKIP) ;
  (fmi2DoStep.2.2.2.fmi2OK -> SKIP) ;
  (fmi2DoStep.3.2.2.fmi2OK -> SKIP) ;
  (fmi2DoStep.4.2.2.fmi2OK -> SKIP)

-- assert SynchronousEventsSpec [FD= FixedStepNoRollbackCosimulation(0,2,2)

-- assert SynchronousEventsSpec [FD= VariableStepNoRollbackCosimulation(0,2)

-- assert SynchronousEventsSpec [FD= VariableStepWithRollbackCosimulation(0,2,2)

-- Auxiliary functions
index(<v>^vals,pos) = if pos == 1 then v else index(vals,pos-1)

min(a,b) = if a >= b then b else a

min4(a,b,c,d) = min(min(a,b),min(c,d))

-- Auxiliary definitions to specify scenarios
AllInInternalChoice(s) =
  if (empty(s))
  then SKIP
  else |~| e: s @ e -> AllInInternalChoice(diff(s,{e}))

AllInInternalofExternalChoices(ss) =
  if (empty(ss))
  then SKIP
  else |~| cs: ss @ ([] e: cs @ e -> AllInInternalChoice(diff(cs,{e})))

channel printme: {(-10)..10}

channel a1, b1, c1, dx

P = (a1 -> b1 -> endsimulation -> SKIP) /\ fatalError -> endsimulation -> SKIP

Q = (c1 -> fatalError -> dx -> Q) /\ endsimulation -> SKIP

Work = P [|{endsimulation,dx,fatalError} |] Q
```

B CSP Model of a Data-Flow Machine

```
include "fmi-master.csp"

--------------------
-- Dataflow in FMI --
-- Specification   --
--------------------

-- Number of FMUS
numfmus = 3
-- FMU identifiers
sourcefmu        = 1
dfspecfmu        = 2
sinkfmu          = 3
-- Set of identifiers
FMI2COMPONENT    = {sourcefmu,dfspecfmu,sinkfmu}
FMI2COMPONENTseq = <sourcefmu,dfspecfmu,sinkfmu>
-- Port dependency graph
-- Outputs
outputs          = <(sourcefmu,1),(dfspecfmu,2)>
outputsset       = {(sourcefmu,1),(dfspecfmu,2)}
portdeps         = <(1,(dfspecfmu,1)),(2,(sinkfmu,1))>

a         = 1
b         = 1
topInput  = 2
INPUTVAL  = { 0..topInput }
topParam  = 2
PARAMVAL  = { 0..topParam }
INPUTVALP = union(INPUTVAL,{eps})
PARAMVALP = union(PARAMVAL,{eps})

channel setoutput : FMI2COMPONENT . INDEX . VAL
channel getnext,getparam,getinput,getoutput : FMI2COMPONENT . INDEX . VAL
channel setparam,setinput   : FMI2COMPONENT . INDEX . VAL

-- Parameters and their values
-- Need to check against types
parameterValues  = <(sourcefmu,1,2),(dfspecfmu,1,a),(sinkfmu,2,b)>

initialValues    = <(sourcefmu,1,0),(dfspecfmu,1,0),(sinkfmu,1,0)>

----------------
-- FMIWrapper --
----------------

FMIWrapper(t0,tN) =
  FMUInterface(sourcefmu)
  [|{endsimulation}|]
  ( FMUInterface(dfspecfmu)
    [|{endsimulation}|]
    FMUInterface(sinkfmu)
  )

ExampleFMIWrapper(t0,tN) =
  SOURCEFMU(sourcefmu)
  [|{endsimulation}|]
  ( DFSPECFMU(dfspecfmu)
    [|{endsimulation}|]
    SINKFMU(sinkfmu)
  )

--------------------
-- Generator      --
--------------------

SOURCEFMUProc(i) =
  let
    Instantiation =
      fmi2Instantiate.i!true -> SKIP
    InstantiationMode =
      fmi2SetUpExperiment.i?t0!true?tN!fmi2OK ->
        setctime.i!t0 -> setetime.i!tN ->
          fmi2EnterInitializationMode.i!fmi2OK -> SKIP
    InitializationMode =
      fmi2ExitInitializationMode.i!fmi2OK -> SKIP
    slaveInitialized(hc) =
      getoutput.i.1?out ->
        ( fmi2Get.i.1!out!fmi2OK -> slaveInitialized(hc)
          []
     fmi2DoStep.i?t?ss!fmi2OK -> (UpdateState2; slaveInitialized(ss))
```

```
)
    UpdateState2 =
      getnext.i.1?v -> setoutput.i.1!v -> SKIP
      []
      emptysourcebuffer -> ErrorProc
    ErrorProc =
      fmi2Get.i.1!0!fmi2Error -> ErrorProc
      []
      fmi2DoStep.i?t?ss!fmi2Error -> ErrorProc
  within
    Instantiation;
    InstantiationMode;
    InitializationMode;
    slaveInitialized(0)

-- State

channel emptysourcebuffer

SOURCEFMUState(i) =
  let
    SMUS(ct,et,xs,out) =
      if xs == <> then
        emptysourcebuffer -> SMUS(ct,et,xs,out)
      else
        ( setctime.i?t -> SMUS(t,et,xs,out)
          []
          setetime.i?t -> SMUS(ct,t,xs,out)
          []
          setoutput.i.1?z -> SMUS(ct,et,xs,z)
          []
          getctime.i!ct -> SMUS(ct,et,xs,out)
          []
          getetime.i!et -> SMUS(ct,et,xs,out)
          []
          getnext.i.1!head(xs) -> SMUS(ct,et,tail(xs),out)
          []
          getoutput.i.1!out -> SMUS(ct,et,xs,out)
          []
          fmi2SetFMUState.i.fmi2OK -> SMUS(ct,et,xs,out)
          []
          fmi2GetFMUState.i.fmi2OK -> SMUS(ct,et,xs,out)
        )
    within
      SMUS(0,0,<1,0>,eps) -- must all be no more than topInput values

-- Process
SOURCEFMU(i) =
  ( SOURCEFMUProc(i)
    [|{|setctime.i, setetime.i, setoutput.i, set.i,
       getctime.i, getetime.i, getoutput.i, get.i, getnext.i,
emptysourcebuffer,
       fmi2Instantiate|}|]
    fmi2Instantiate.i?b -> SOURCEFMUState(i)
  ) \ {|get.i,set.i,getctime,getetime,setctime,setetime|}
  /\
  ( (fmi2Terminate.i!fmi2OK -> fmi2FreeInstance.i!fmi2OK -> STOP) /\ (endsimulation -> SKIP) )

assert SOURCEFMUProc(1) :[deadlock free]

assert SOURCEFMUProc(1) :[livelock free]

assert SOURCEFMUProc(1) :[deterministic]

assert SOURCEFMUState(1) :[deadlock free]

assert SOURCEFMUState(1) :[livelock free]

assert SOURCEFMUState(1) :[deterministic]

assert SOURCEFMU(1) ; IDLE :[deadlock free]

assert SOURCEFMU(1) :[livelock free]

assert SOURCEFMU(1) :[deterministic]

-------------
-- Sink --
-------------
-- Main action

SINKFMUPROC(i) =
  let
```

```
    Instantiation =
      fmi2Instantiate.i!true -> SKIP
    InstantiationMode =
      fmi2SetUpExperiment.i?t0!true?tN!fmi2OK ->
        setctime.i!t0 -> setetime.i!tN ->
          fmi2EnterInitializationMode.i!fmi2OK -> SKIP
    InitializationMode =
      fmi2ExitInitializationMode.i!fmi2OK -> SKIP
    slaveInitialized(hc) =
      getoutput.i.1?out ->
        ( fmi2Set.i.1!out!fmi2OK -> slaveInitialized(hc)
          []
          fmi2DoStep.i?t?ss!fmi2OK -> ( UpdateState2; slaveInitialized(ss) ) )
    UpdateState2 =
      getinput.i?v -> set.i!v -> SKIP
  within
    Instantiation;
    InstantiationMode;
    InitializationMode;
    slaveInitialized(0)

-- State

SINKFMUState(i) =
  let
    SKFMUS(ct,et,in,x) =
      setctime.i?t -> SKFMUS(t,et,in,x)
      []
      setetime.i?t -> SKFMUS(ct,t,in,x)
      []
      set.i.1?y -> SKFMUS(ct,et,in,y)
      []
      setinput.i.1?z -> SKFMUS(ct,et,z,x)
      []
      getctime.i!ct -> SKFMUS(ct,et,in,x)
      []
      getetime.i!et -> SKFMUS(ct,et,in,x)
      []
      get.i.1!x -> SKFMUS(ct,et,in,x)
      []
      getinput.i.1!in -> SKFMUS(ct,et,in,x)
      []
      fmi2SetFMUState.i.fmi2OK -> SKFMUS(ct,et,in,x)
      []
      fmi2GetFMUState.i.fmi2OK -> SKFMUS(ct,et,in,x)
    within
      SKFMUS(0,0,eps,eps)

-- Process
SINKFMU(i) =
  ( SOURCEFMUProc(i)
    [|{|setctime.i, setetime.i, set.i, setinput.i,
        getctime.i, getetime.i, get.i, getinput.i,
        fmi2Instantiate|}|]
    fmi2Instantiate.i?b -> SINKFMUState(i)
  ) \ {|get.i,set.i,getctime.i,getetime.i,setctime.i,setetime.i,setinput.i,getinput.i|}
  /\
  ( (fmi2Terminate.i!fmi2OK -> fmi2FreeInstance.i!fmi2OK -> STOP) /\ (endsimulation -> SKIP) )

assert SINKFMUPROC(3) :[deadlock free]
assert SINKFMUPROC(3) :[livelock free]
assert SINKFMUPROC(3) :[deterministic]
assert SINKFMUState(3) :[deadlock free]
assert SINKFMUState(3) :[livelock free]
assert SINKFMUState(3) :[deterministic]
assert SINKFMU(3) ; IDLE :[deadlock free]
assert SINKFMU(3) :[livelock free]
assert SINKFMU(3) :[deterministic]

--------------------
-- DFSpec --
--------------------
-- Main action

channel answer : VAL
channel geta : PARAMVAL
channel getx : INPUTVALP
channel getb : PARAMVAL
channel gety : INPUTVAL
channel kill

DFSPECFMUProc(i) =
  let
```

```
    Instantiation =
      fmi2Instantiate.i!true -> SKIP
    InstantiationMode(p,q) =
      fmi2Set.i.1?v:PARAMVAL?fmi2OK -> setparam.i.1!v -> InstantiationMode(v,q)
      []
      fmi2Set.i.2?v:PARAMVAL?fmi2OK -> setparam.i.2!v -> InstantiationMode(p,v)
      []
      if p != eps and q != eps then
        fmi2SetUpExperiment.i?t0!true?tN!fmi2OK ->
          setctime.i!t0 ->
            setetime.i!tN ->
              fmi2EnterInitializationMode.i!fmi2OK ->
                SKIP
      else
        fmi2SetUpExperiment.i?t0!true?tN!fmi2Error ->
          fmi2FreeInstance.i!fmi2Error ->
    ErrorProc
    InitializationMode =
      fmi2Set.i.1?v:INPUTVAL!fmi2OK -> setinput.i.1!v -> InitializationMode
      []
      fmi2ExitInitializationMode.i!fmi2OK -> SKIP
    slaveInitialized(hc) =
      getoutput.i.1?out ->
        ( fmi2Get.i.1!out!fmi2OK -> slaveInitialized(hc)
          []
          fmi2Set.i.1?v:INPUTVAL!fmi2OK -> setinput.i.1!v -> slaveInitialized(hc)
          []
          fmi2DoStep.i?t?ss!fmi2OK -> (UpdateState; slaveInitialized(ss))
)
    UpdateState =
     let
Fetch =
  getinput.i.1?x:INPUTVALP ->
            get.i.1?y:INPUTVALP ->
              if y == eps then
                set.i.1!x -> kill -> SKIP
              else
                getparam.i.1?a:PARAMVAL ->
                  getparam.i.2?b:PARAMVAL ->
                    geta!a ->
                      getx!x ->
                        getb!b ->
                          gety!y ->
                            answer?z:VAL ->
                              setoutput.i.1!z ->
                                SKIP
       Calculator =
  geta?a:PARAMVAL ->
    getx?x:INPUTVAL ->
      getb?b:PARAMVAL ->
        gety?y:INPUTVAL ->
  answer!((a*x) + (b*y)) ->
    SKIP
          []
  kill -> SKIP
      within
        Fetch [|{|geta,getx,getb,gety,answer,kill|}|] Calculator
    ErrorProc =
      fmi2SetFMUState.i.fmi2Error -> ErrorProc
      []
      fmi2GetFMUState.i.fmi2Error -> ErrorProc
      []
      fmi2Set.i.1?v:INPUTVAL!fmi2Error -> ErrorProc
      []
      fmi2ExitInitializationMode.i!fmi2Error -> ErrorProc
      []
      fmi2Get.i.1!1!fmi2Error -> ErrorProc
      []
      fmi2Set.i.1?v:INPUTVAL!fmi2Error -> ErrorProc
      []
      fmi2DoStep.i?t?ss!fmi2Error -> ErrorProc
  within
    Instantiation;
    InstantiationMode(eps,eps);
    InitializationMode;
    slaveInitialized(0)

-- State

DFSPECFMUState(i) =
  let
    DFSFMUS(ct,et,p,q,in,x,out) =
      setctime.i?t:TIME -> DFSFMUS(t,et,p,q,in,x,out)
```

```
      []
      setetime.i?t:TIME -> DFSFMUS(ct,t,p,q,in,x,out)
      []
      setparam.i.1?r:PARAMVAL -> DFSFMUS(ct,et,r,q,in,x,out)
      []
      setparam.i.2?s:PARAMVAL -> DFSFMUS(ct,et,p,s,in,x,out)
      []
      setinput.i.1?z:INPUTVALP -> DFSFMUS(ct,et,p,q,z,x,out)
      []
      set.i.1?y:INPUTVALP -> DFSFMUS(ct,et,p,q,in,y,out)
      []
      setoutput.i.1?z:VAL -> DFSFMUS(ct,et,p,q,in,x,z)
      []
      getctime.i!ct -> DFSFMUS(ct,et,p,q,in,x,out)
      []
      getetime.i!et -> DFSFMUS(ct,et,p,q,in,x,out)
      []
      getparam.i.1!p -> DFSFMUS(ct,et,p,q,in,x,out)
      []
      getparam.i.2!q -> DFSFMUS(ct,et,p,q,in,x,out)
      []
      getinput.i.1!in -> DFSFMUS(ct,et,p,q,in,x,out)
      []
      get.i.1!x -> DFSFMUS(ct,et,p,q,in,x,out)
      []
      getoutput.i.1!out -> DFSFMUS(ct,et,p,q,in,x,out)
      []
      fmi2SetFMUState.i.fmi2OK -> DFSFMUS(ct,et,p,q,in,x,out)
      []
      fmi2GetFMUState.i.fmi2OK -> DFSFMUS(ct,et,p,q,in,x,out)
    within
      DFSFMUS(0,0,eps,eps,eps,eps,eps)

-- Process
DFSPECFMU(i) =
  ( DFSPECFMUProc(i)
    [|{|setctime.i, setetime.i, setparam.i, setinput.i, set.i, setoutput.i,
        getctime.i, getetime.i, getparam.i, getinput.i, get.i, getoutput.i,
kill,
        fmi2Instantiate|}|]
    fmi2Instantiate.i?b -> DFSPECFMUState(i)
  ) \ {|setctime.i, setetime.i, setparam.i, setinput.i, set.i, setoutput.i,
        getctime.i, getetime.i, getparam.i, getinput.i, get.i, getoutput.i,
geta, getx, getb, gety, answer, kill|}
  /\
  ( (fmi2Terminate.i!fmi2OK -> fmi2FreeInstance.i!fmi2OK -> STOP) /\ (endsimulation -> SKIP) )


DFSTEST =
  DFSPECFMUProc(2)
    [|{|setctime.2, setetime.2, setparam.2, setinput.2, set.2, setoutput.2,
        getctime.2, getetime.2, getparam.2, getinput.2, get.2, getoutput.2,
        fmi2Instantiate|}|]
  fmi2Instantiate.2?b -> DFSPECFMUState(2)

assert DFSPECFMUProc(2) :[livelock free]

assert DFSPECFMUProc(2) :[deterministic]

assert DFSPECFMUState(2) :[deadlock free]

assert DFSPECFMUState(2) :[livelock free]

assert DFSPECFMUState(2) :[deterministic]

assert DFSTEST :[deadlock free]

assert DFSTEST :[livelock free]

assert DFSTEST :[deterministic]

assert DFSPECFMU(2) ; IDLE :[deadlock free]

assert DFSPECFMU(2) :[livelock free]

assert DFSPECFMU(2) :[deterministic]

assert FMUInterface(dfspecfmu) [T= DFSPECFMU(dfspecfmu)

assert Cosimulation(0,2) ; IDLE :[deadlock free]
```

C CSP Model of a PDSG Sampler

```
channel idle

IDLE = idle -> IDLE

--------------------
-- model of time --
--------------------
endOfTime        = 5
TIME             = { 0..endOfTime }
NZTIME           = diff(TIME,{0})
eps              = -1

---------------------
-- Model of values --
---------------------
topVal    = 2
VAL       = { -1..topVal }

---------------------------------------
-- Inputs to the translation process --
---------------------------------------
startTime        = 0
-- We do not support stopTimeDefined = false, either here or in the COE.
stopTimeDefined  = true
stopTime         = 5

-- Number of FMUS
numfmus = 4
-- FMU identifiers
pdsgfmu1         = 1
pdsgfmu2         = 2
samplerfmu       = 3
checkequalityfmu = 4
-- Set of identifiers
FMI2COMPONENT    = {pdsgfmu1,pdsgfmu2,samplerfmu,checkequalityfmu}
FMUs = <pdsgfmu1,pdsgfmu2,samplerfmu,checkequalityfmu>

-- Port dependency graph
-- Outputs
outputs          = <(pdsgfmu1,1),(pdsgfmu2,1),(samplerfmu,3)>
outputsset       = {(pdsgfmu1,1),(pdsgfmu2,1),(samplerfmu,3)}
pdg              = <(1,(samplerfmu,1)),(2,(samplerfmu,2)),(2,(checkequalityfmu,2)),(3,(checkequalityfmu,1))>

-- Parameters and their values
parameterValues  = <(pdsgfmu1,1,1),(pdsgfmu1,2,1),(pdsgfmu2,1,1),(pdsgfmu2,2,2)>

-- Inputs and their initial values
inputs(pdsgfmu1)         = <>
inputs(pdsgfmu2)         = <>
inputs(samplerfmu)       = <1,2>
inputs(checkequalityfmu) = <1,2>
initialValues    = <(samplerfmu,1,1),(samplerfmu,2,1),(checkequalityfmu,1,1),(checkequalityfmu,2,1)>

-- Exposed variables
expvars(pdsgfmu1)         = <>
expvars(pdsgfmu2)         = <>
expvars(samplerfmu)       = <>
expvars(checkequalityfmu) = <>

-- Type that represents the state of any of the FMUs. It can be generated.
datatype FMUSTATE = three.VAL.VAL.VAL | two.VAL.VAL

-- Port indices, used to model variable names, in the context of each FMU.
INDEX            = { 1..5 }

-- Return type of FMI API functions
datatype FMI2STATUSFULL = fmi2OK | fmi2Discard | fmi2Error | fmi2Fatal

FMI2STATUS = {fmi2OK, fmi2Error, fmi2Fatal}

ErrorFlags = {fmi2Error,fmi2Fatal}

-----------------------
-- FMI API functions --
-----------------------
channel fmi2Get                          : FMI2COMPONENT . INDEX . VAL . FMI2STATUSFULL
channel fmi2Set                          : FMI2COMPONENT . INDEX . VAL . FMI2STATUS
-- The step size in an fmi2DoStep call cannot be 0.
channel fmi2DoStep                       : FMI2COMPONENT . TIME . NZTIME . FMI2STATUSFULL
channel fmi2Instantiate                  : FMI2COMPONENT . Bool
```

```
-- This function returns a component, and null if the instantiation
-- fails. We do not model pointers, so in our model we use a boolean
-- to cater for the possibility that instantiation may fail.
channel fmi2SetUpExperiment               : FMI2COMPONENT . TIME . Bool . TIME . FMI2STATUS
channel fmi2EnterInitializationMode       : FMI2COMPONENT . FMI2STATUS
channel fmi2ExitInitializationMode        : FMI2COMPONENT . FMI2STATUS
channel fmi2GetBooleanStatusfmi2Terminated : FMI2COMPONENT . Bool . FMI2STATUS
channel fmi2GetMaxStepSize                : FMI2COMPONENT . NZTIME . FMI2STATUS
channel fmi2Terminate                     : FMI2COMPONENT . FMI2STATUS
channel fmi2FreeInstance                  : FMI2COMPONENT . FMI2STATUS
channel fmi2GetFMUState                   : FMI2COMPONENT -- . FMUSTATE
                                            . FMI2STATUS
channel fmi2SetFMUState                   : FMI2COMPONENT -- . FMUSTATE
                                            . FMI2STATUS

FMIAPI = {| fmi2Get,
            fmi2Set,
            fmi2DoStep,
            fmi2Instantiate,
            fmi2SetUpExperiment,
            fmi2EnterInitializationMode,
            fmi2ExitInitializationMode,
            fmi2GetBooleanStatusfmi2Terminated,
            fmi2GetMaxStepSize,
            fmi2Terminate,
            fmi2FreeInstance,
            fmi2GetFMUState,
            fmi2SetFMUState |}

FMUAPI(i) = {| fmi2Get.i,
               fmi2Set.i,
               fmi2DoStep.i,
               fmi2Instantiate.i,
               fmi2SetUpExperiment.i,
               fmi2EnterInitializationMode.i,
               fmi2ExitInitializationMode.i,
               fmi2GetBooleanStatusfmi2Terminated.i,
               fmi2GetMaxStepSize.i,
               fmi2Terminate.i,
               fmi2FreeInstance.i,
               fmi2GetFMUState.i,
               fmi2SetFMUState.i |}
----------------------------
-- Timer control channels --
----------------------------
channel end
channel step                        : TIME . NZTIME
channel setT                        : TIME
channel updateSS                    : NZTIME

-------------------------------------
-- State manager control channels --
-------------------------------------
channel get                         : FMI2COMPONENT . INDEX . VAL
channel set                         : FMI2COMPONENT . INDEX . VAL
channel getctime,getetime           : FMI2COMPONENT . TIME
channel setctime,setetime           : FMI2COMPONENT . TIME

-----------------------------------------
-- Controller for FMU status channels --
-----------------------------------------
channel stepAnalysed
channel stepToComplete

-------------------------------------------------------------
-- Control channel to shutdown the simulation.
-- We note that, since an FMU may fail, its termination may
-- not be carried out gracefully with fmi2Terminate and
-- fmi2FreeInstance.  This channel is used to indicate the
-- end of the experiment in all cases and shutdown the
-- model processes.
-------------------------------------------------------------
channel endsimulation

-------------------------------------------------------------
-- If any of the API functions returns an error, further
-- calls to API functions is restricted. This is ensured
-- by flagging the error via the channel error.
-------------------------------------------------------------
channel error: ErrorFlags

----------------------
-- Master Algorithm --
```

```
----------------------
-- This is a general characterisation of the valid history of interactions
-- traces of a master algorithm. It does not commit to specific policies to
-- define step size and error treatment, for example.

MAlgorithm(t0,tN) =
  ( ( ( TimedInteractions(t0,tN)
          [|{|endsimulation,fmi2Instantiate|}|]
        FMUStatesManager)
      /\ ErrorManager )
        [| union(FMIAPI,{|endsimulation,error|}) |]
      ErrorHandler)
    \ {|error|}

TimedInteractions(t0,tN) =
  ( (Timer(t0,2,tN) /\ endSimulation) [| {|step,end,setT,updateSS,endsimulation|} |] Interaction \ {stepAnalysed,stepToComplete})
    \ {|step,end,setT,updateSS|}
-- stepAnalysed and stepToComplete should not be in the specification. It is there because
-- Interaction is used to specify other algorithms.

endSimulation = endsimulation -> SKIP

-----------------------------------------
-- Classic Brute-force Master Algorithm --
-----------------------------------------

-- Because fmi2Instantiate is used by FixedStepTimedInteractions, it needs to be
-- allowed by NoStateManager.
ClassicMAlgorithm(t0,hc,tN) =
  ( ( ( FixedStepTimedInteractions(t0,hc,tN)
          [|{|endsimulation,fmi2Instantiate|}|]
        NoStateManager)
      /\ ErrorManager )
        [| union(FMIAPI,{|endsimulation,error|}) |]
      ErrorHandler)
    \ {|error|}

FixedStepTimedInteractions(t0,hc,tN) =
  ( (FixedStepTimerNoRollBack(t0,hc,tN) /\ endSimulation)
      [| {|step,end,setT,updateSS,endsimulation|} |]
    FixedStepInteraction(hc,tN) )
    \ {|step,end,setT,updateSS|}

-- The refinement below follows by monotonicity of
-- the refinements proved in the sequel.
-- assert MAlgorithm(0,endOfTime) [T= ClassicMAlgorithm(0,2,endOfTime)

-- Proof of termination
assert ClassicMAlgorithm(0,2,endOfTime) \ Events [F= SKIP
assert SKIP [F= ClassicMAlgorithm(0,2,endOfTime) \Events

-- Proof of deadlock freedom, except for termination
assert ClassicMAlgorithm(0,2,endOfTime); IDLE :[deadlock free]

-- Proof of determinism, in the CSP sense.
assert ClassicMAlgorithm(0,2,endOfTime) :[deterministic]

-------------------------------
-- Standard Master Algorithm --
-------------------------------

StandardClassicMAlgorithm(t0,hc,tN) =
  FixedStepTimedInteractions(t0,hc,tN)
    [|{|endsimulation,fmi2Instantiate|}|]
  NoStateManager

-- assert MAlgorithm(0,endOfTime) [T= StandardClassicMAlgorithm(0,2,endOfTime) RiP

assert StandardClassicMAlgorithm(0,2,endOfTime) \ Events [F= SKIP
assert SKIP [F= StandardClassicMAlgorithm(0,2,endOfTime) \Events

assert StandardClassicMAlgorithm(0,2,endOfTime); IDLE :[deadlock free]

assert StandardClassicMAlgorithm(0,2,endOfTime) :[deterministic]

-------------------------------
-- Simulink Master Algorithm --
-------------------------------

SimulinkMAlgorithm(t0,tN) =
  ( ( ( VariableStepNoRollbackTimedInteractions(t0,tN)
          [|{|endsimulation,fmi2Instantiate|}|]
        NoStateManager)
```

```
      /\ ErrorManager )
        [| union(FMIAPI,{|endsimulation,error|}) |]
      ErrorHandler)
    \ {|error|}

-- We use 2 as a starting step size.
VariableStepNoRollbackTimedInteractions(t0,tN) =
  ( (VariableStepTimerNoRollBack(t0,2,tN) /\ endSimulation)
      [| {|step,end,setT,updateSS,endsimulation|} |]
    VariableStepInteraction(2,tN) )
    \ {|step,end,setT,updateSS|}

-- The refinement below follows by monotonicity of
-- the refinements proved in the sequel.
-- assert MAlgorithm(0,endOfTime) [T= SimulinkMAlgorithm(0,2,endOfTime)

assert SimulinkMAlgorithm(0,endOfTime) \ Events [F= SKIP
assert SKIP [F= SimulinkMAlgorithm(0,endOfTime) \Events

assert SimulinkMAlgorithm(0,endOfTime); IDLE :[deadlock free]

assert SimulinkMAlgorithm(0,endOfTime) :[deterministic]

-----------------------------
-- Broman Master Algorithm --
-----------------------------

VariableStepWithRollbackMAlgorithm(t0,hc,tN) =
  ( ( ( VariableStepWithRollbackTimedInteractions(t0,hc,tN)
          [|{|endsimulation,fmi2Instantiate,fmi2GetFMUState,fmi2SetFMUState|}|]
        FMUStatesManager)
      /\ ErrorManager )
        [| union(FMIAPI,{|endsimulation,error|}) |]
      ErrorHandler)
    \ {|error|}

VariableStepWithRollbackTimedInteractions(t0,hc,tN) =
  ( (VariableStepTimerWithRollBack(t0,t0,2,tN) /\ endSimulation)
      [| {|step,end,setT,updateSS,endsimulation|} |]
    VariableStepWithRollbackInteraction(hc) )
    \ {|step,end,setT,updateSS|}

-- The refinement below follows by monotonicity of
-- the refinements proved in the sequel.
-- assert MAlgorithm(0,endOfTime) [T= VariableStepWithRollbackMAlgorithm(0,2,endOfTime)

assert VariableStepWithRollbackMAlgorithm(0,2,endOfTime) \ Events [F= SKIP
assert SKIP [F= VariableStepWithRollbackMAlgorithm(0,2,endOfTime) \Events

assert VariableStepWithRollbackMAlgorithm(0,2,endOfTime); IDLE :[deadlock free]

assert VariableStepWithRollbackMAlgorithm(0,2,endOfTime) :[deterministic]

-------------------
-- General timer --
-------------------
-- It allows roolbacks (setT), variable step size (updateSS),
-- as well as indicating the steps (step) and the end
-- (end) of the simulation
Timer(ct,hc,tN) =
  let
    T(t,ss) =
      setT?t: { vt | vt <- TIME, vt <= tN} -> T(t,ss)
      []
      updateSS?nhc: NZTIME -> T(t,nhc)
      []
      step!t!ss -> T(min(t+ss,tN),ss)
      []
      t == tN & end -> STOP
    within
      T(ct,hc)

--------------------------
-- Simple example timer --
--------------------------
FixedStepTimerNoRollBack(t0,hc,tN) =
  let FSTNRB(t) =
    if t <= tN then
      step.t.hc -> FSTNRB(t+hc)
    else
      end -> STOP
  within
    FSTNRB(t0)
```

```
-- We use the general components to give trace specifications to models
-- of a particular algorithm.
assert Timer(0,2,endOfTime) [T= FixedStepTimerNoRollBack(0,2,endOfTime)

--------------------
-- Simulink timer --
--------------------
VariableStepTimerNoRollBack(t0,hc,tN) =
  let
    VSTNRB(t,ss) =
      if t <= tN then
        step.t.ss -> VSTNRB(t+ss,ss)
        []
        updateSS?nhc: NZTIME -> VSTNRB(t,nhc)
      else
        end -> STOP
  within
    VSTNRB(t0,hc)

-- assert Timer(0,2,endOfTime) [T= VariableStepTimerNoRollBack(0,2,endOfTime)

------------------
-- Broman timer --
------------------
VariableStepTimerWithRollBack(t0,pt,hc,tN) =
  let VSTWRB(t,p,ss) =
      if t <= tN then
        step.t.ss -> VSTWRB(t+ss,t,ss)
        []
        updateSS?nhc: NZTIME -> VSTWRB(t,p,nhc)
        []
        setT.p -> VSTWRB(p,p,ss)
      else
        end -> STOP
  within
    VSTWRB(t0,pt,hc)

-- assert Timer(0,2,endOfTime) [T= VariableStepTimerWithRollBack(0,0,2,endOfTime)

------------------------------------------------------------
-- General interaction pattern for a master algorithm --
------------------------------------------------------------
-- This pattern can be autoamtically generated for a
-- given set of FMUs, as indicated by the use of the
-- inputs to such a procedure that we instantiate above
-- for a particular example.

Interaction =
  let
    Instantiation =
      ; i: FMUs @ fmi2Instantiate.i?sc -> SKIP

    InstantiationMode(params) =
      if params == <> then
        ( ( ; i : FMUs @
      fmi2SetUpExperiment!i!startTime!stopTimeDefined!stopTime?st -> SKIP ) ;
          ( ; i : FMUs @
      fmi2EnterInitializationMode.i?st -> SKIP ) )
      else
        let (i,x,v) = head(params) within
  fmi2Set!i!x!v?st -> InstantiationMode(tail(params))

    InitializationMode(inits) =
      if inits == <> then
        ( ; i : FMUs @
      fmi2ExitInitializationMode!i?st -> SKIP )
      else
        let (i,x,v) = head(inits) within
  fmi2Set!i!x!v?st -> InitializationMode(tail(inits))

    slaveInitialized =
      end -> Terminated
      []
      -- In Circus, we get a sequence.
      step?t?hc -> (TakeOutputs(outputs,<>,t,hc); Step(t,hc))

    TakeOutputs(outs,vals,t,hc) =
      let
        TO(os,vs) =
          if os == <> then
            DistributeInputs(pdg,vs,t,hc)
          else
```

```
            let (i,n) = head(os) within
              fmi2Get.i.n?v?st -> TO(tail(os),vs^<v>)
      within
        TO(outs,vals)

    DistributeInputs(pds,vals,t,hc) =
      let
        DI(ps) =
          if ps == <> then
            SKIP
          else
            let (pos,(i,n)) = head(ps) within
           fmi2Set.i.n!index(vals,pos)?st ->
             DI(tail(ps))
      within
        DI(pds)

    Step(t,hc) = let
      Iterations(i) = hc != 0 & (
        if i == 0 then
          stepToComplete -> fmi2DoStep.1.t.hc?st -> Iterations(1)
        else if i < numfmus then
         (fmi2GetBooleanStatusfmi2Terminated.i?b?st -> Iterations(i)
          []
          fmi2GetMaxStepSize.i?t?st -> Iterations(i)
          []
          fmi2DoStep.(i+1).t.hc?st -> Iterations(i+1))
        else (fmi2GetBooleanStatusfmi2Terminated.i?b?st -> Iterations(i)
              []
              fmi2GetMaxStepSize.i?t?st -> Iterations(i)
              []
              stepAnalysed -> SKIP)
        )
      within
        Iterations(0);
        NextStep

    NextStep = updateSS?d -> NextStep
               []
               setT?t -> NextStep
               []
               slaveInitialized
               []
               Terminated

    Terminated =
      ( ; i: FMUs @ fmi2Terminate.i?st -> fmi2FreeInstance.i?st -> SKIP );
      endsimulation -> SKIP

  within
    Instantiation ;
    InstantiationMode(parameterValues);
    InitializationMode(initialValues);
    slaveInitialized

-------------------------------------------------
-- Interaction patterns for a master algorithm --
-------------------------------------------------
-----------------------------
-- Example in the standard --
-----------------------------

PossibleTerminationOnDiscard(tN) =
   (Interaction
      [| {| fmi2DoStep, fmi2GetBooleanStatusfmi2Terminated, end, step, stepAnalysed, fmi2Terminate, fmi2FreeInstance, endsimulation |} |]
    DiscardMonitor) \ {stepAnalysed,stepToComplete}

-- The interest in stepAnalysed ensures that the FMU step cannot terminate
-- before its fmi2GetBooleanStatusfmi2Terminated is requested.
DiscardMonitor =
  let

    -- step is now blocked.  When stepAnalysed happens, the co-simulation terminates.
    ToDiscard = fmi2DoStep?i?t?hc?st -> ToDiscard
                []
                stepAnalysed -> SKIP
                []
                end -> SKIP

    Monitor = fmi2DoStep?i?t?hc?st ->
        (if st == fmi2Discard then
          fmi2GetBooleanStatusfmi2Terminated.i?b?st0m ->
            ( if b == true
```

```
              then ToDiscard -- Although the algorithm checks this, it ignores that status until the next step.
              else Monitor)
        else
          Monitor)
      []
      stepAnalysed -> Monitor
      []
      step?t?hc -> Monitor
      []
      end -> SKIP

    Terminated = ( ; i: FMUs @ fmi2Terminate.i?st -> fmi2FreeInstance.i?st -> SKIP );
             endsimulation -> SKIP

  within
    Monitor; Terminated

FixedStep = endsimulation -> SKIP

FixedStepInteraction(hc,tN) =
  PossibleTerminationOnDiscard(tN)
    [ union(union(diff(FMIAPI,{|fmi2GetMaxStepSize|}),{endsimulation,end}),{step.t.hc | t <- TIME})
      ||
    { endsimulation } ]
  -- This is here just to mimic the general shape of the definition of an interaction
  -- and to allow us to cut down the behaviour of PossibleTerminationOnDiscard.
  FixedStep

-- stepAnalysed and stepToComplete are there just to support the definition of
-- other algorithms. They should not be in the interface.
assert Interaction \ {stepAnalysed,stepToComplete} [T= FixedStepInteraction(2,endOfTime)

------------------------
-- Simulink algorithm --
------------------------
-- The synchronisation between Interaction and VaryStep on updateSS
-- ensures that the update can only take place at the right point
-- of the loop.
VariableStepInteraction(hc,tN) =
  PossibleTerminationOnDiscard(tN)
    [ union(diff(FMIAPI,{|fmi2GetMaxStepSize|}),{|endsimulation,end,stepAnalysed,stepToComplete,step,updateSS|})
        ||
    {| fmi2Get,updateSS,endsimulation |} ]
  VaryStep(hc)

channel delta: {(-3)..3}

-- This can be modelled much more generally and elegantly in Circus.
-- The initial step size is defined in the instantiation of the right timer.
VaryStep(hc) =
  let
    threshold = 5

    Monitor(y1,y2,y3) =
     fmi2Get.1.1?ny1?st -> delta!(y1 - ny1) ->
     fmi2Get.2.1?ny2?st -> delta!(y2 - ny2) ->
     fmi2Get.3.3?ny3?st -> delta!(y3 - ny3) ->
     Monitor(ny1,ny2,ny3)

    Adjust(hc) = delta?d1 -> delta?d2 -> delta?d3 ->
      if (d1 != eps and (d1 >= threshold or d1 <= -threshold)) and
         (d2 != eps and (d2 >= threshold or d2 <= -threshold)) and
         (d1 != eps and (d3 >= threshold or d3 <= -threshold))
      then updateSS!min(hc+1,endOfTime) -> Adjust(hc+1)
      else Adjust(hc)

      Step = (Monitor(eps,eps,eps) [| {| delta |} |] Adjust(hc)) \ {| delta |}
  within
      Step /\ endsimulation -> SKIP

assert Interaction \ {stepAnalysed,stepToComplete} [T= VariableStepInteraction(2,endOfTime)

----------------------
-- Broman algorithm --
----------------------

-- Interaction is nondeterministic, in that, at the end of each step, it allows the
-- step to be terminated or the next step to proceed. To make it deterministic, we
-- need the algorithm to make a decision.  Here, we just allow termination at the end.
-- Interaction also allows the use of fmi2GetBooleanStatusfmi2Terminated, which is not
-- needed in this algorithm. To block it, we put it in the interface of Rollback.
VariableStepWithRollbackInteraction(hcmax) =
  ((Interaction [| {| end, fmi2Terminate, endsimulation |} |] (end -> RUN({| fmi2Terminate |}) /\ endSimulation) )
```

```
     [ union(diff(FMIAPI,{|fmi2GetFMUState,fmi2SetFMUState|}),{|endsimulation,end,step,stepAnalysed,stepToComplete,setT,updateSS|})
       ||
     {| fmi2DoStep,fmi2GetBooleanStatusfmi2Terminated,fmi2GetMaxStepSize,
        fmi2GetFMUState,fmi2SetFMUState,
        step,stepAnalysed,stepToComplete,setT,updateSS,endsimulation |} ]
   Rollback(hcmax)) \ {stepAnalysed,stepToComplete}

channel hcmin: TIME

Rollback(hcmax) =
  let
    -- In Circus, this can be much more elegant.
    Monitor =
      -- Synchronisation on step, as well as stepToComplete, ensures that a new step cannot start
      -- before the management required here is completed.
      step?t?hc -> stepToComplete ->
        (; i: FMUs @ fmi2GetFMUState.i?st -> SKIP); -- ?s?st -> SKIP) ;
        (; i: FMUs @ fmi2DoStep?i?t.hc?st ->  fmi2GetMaxStepSize.i?hc?st -> hcmin!hc -> SKIP)

    Decide =
      step?t?hc -> (hcmin?h1 -> hcmin?h2 -> hcmin?h3 -> hcmin?h4 ->
        let
          newhc = min4(h1,h2,h3,h4)
        within
          if 0 < newhc and newhc < hc
          then (; i: FMUs @ fmi2SetFMUState.i?st -> SKIP); -- ?s?st -> SKIP);
              stepAnalysed -> setT?t -> updateSS!newhc -> SKIP
          else stepAnalysed -> SKIP)

      Step = ((Monitor [| {| step,hcmin |} |] Decide) \ {| hcmin |}); Step
  within
      Step /\ endsimulation -> SKIP

assert (TimedInteractions(0,endOfTime) [|{|endsimulation,fmi2Instantiate|}|] FMUStatesManager)
       [T=
       (VariableStepWithRollbackTimedInteractions(0,2,endOfTime)
          [|{|endsimulation,fmi2Instantiate,fmi2GetFMUState,fmi2SetFMUState|}|]
        FMUStatesManager )

-------------------------------------------------
-- General state manager for a master algorithm --
-------------------------------------------------
-- Once an FMU is instantiated, then it is possible to retrieve the
-- state. After that, both gets and sets are allowed. Error management
-- is handled by Interaction.
FMUStateManager(i) =
  let
    AllowAGet =
      fmi2GetFMUState.i?st -- ?s?st
      -> AllowsGetsAndSets -- (s)

    AllowsGetsAndSets = -- (s) =
      fmi2GetFMUState.i?st -- ?t?st
      -> AllowsGetsAndSets -- (t)
      []
      fmi2SetFMUState.i?st -- !s?st
      -> AllowsGetsAndSets -- (s)
  within
    fmi2Instantiate.i?b -> AllowAGet

FMUStatesManager = ( ||| i : FMI2COMPONENT @ FMUStateManager(i) ) /\ endSimulation

-------------------------------------------------
-- Simple state manager for a master algorithm --
-------------------------------------------------
NoStateManager = ( ||| i : FMI2COMPONENT @ fmi2Instantiate.i?b -> STOP) /\ endSimulation

-- We cannot model check the stateful version of the algorithms, but with the state
-- actually commented out, it passes.
assert FMUStatesManager [T= NoStateManager

-------------------------------------------------
-- State manager for Broman's master algorithm --
-------------------------------------------------
BromanFMUStatesManager = FMUStatesManager

--------------------
-- Error Handling --
--------------------

-- The parameter mst is the monitored status.  It can be either fmi2Error or fmi2Fatal.
-- We are not treating the occurrence of fmi2Error as recoverable. So, there is no possibility
-- of resetting the FMU and continuing.
```

```
ErrorMonitor(mst) = let

  Monitor(st) =
    fmi2Get?i?n?v?st -> StopError(st)
    []
    fmi2Set?i?n?v?st -> StopError(st)
    []
    fmi2GetFMUState?i?st -> StopError(st) -- ?s?st -> StopError(st)
    []
    fmi2SetFMUState?i?st -> StopError(st) -- ?s?st -> StopError(st)
    []
    fmi2SetUpExperiment?i?t?b?hc?st -> StopError(st)
    []
    fmi2EnterInitializationMode?i?st -> StopError(st)
    []
    fmi2ExitInitializationMode?i?st -> StopError(st)
    []
    fmi2GetBooleanStatusfmi2Terminated?i?b?st -> StopError(st)
    []
    fmi2DoStep?i?t?hc?st -> StopError(st)
    []
    fmi2Terminate?i?st -> StopError(st)
    []
    fmi2GetMaxStepSize?i?t?st -> StopError(st)
    []
    fmi2Instantiate?i?b -> (if b == true then StopError(fmi2OK) else StopError(fmi2Fatal))
    []
    fmi2FreeInstance?i?st -> StopError(st)

    StopError(st) = st == mst & error.mst -> Monitor(st) [] st != mst & Monitor(st)

within
    Monitor(fmi2OK) /\ endsimulation -> SKIP

ErrorHandler = ErrorMonitor(fmi2Error) [| union(FMIAPI,{endsimulation}) |] ErrorMonitor(fmi2Fatal)

ErrorManager = FatalError [] ErrorManagement

FatalError = error.fmi2Fatal -> endsimulation -> SKIP

ErrorManagement = let
  Shutdown(i) = fmi2Instantiate.i?b -> ShutdownCreated(i); endsimulation -> SKIP
                []
                error.fmi2Error -> endsimulation -> SKIP

  ShutdownCreated(i) = error.fmi2Error -> fmi2FreeInstance.i?st -> SKIP
                       []
                       fmi2FreeInstance.i?st -> SKIP

within
 (|| i: FMI2COMPONENT @ [{|error,endsimulation|}] Shutdown(i))

----------------
-- FMIWrapper --
----------------

FMIWrapper(t0,tN) =
  FMUInterface(pdsgfmu1)
  [|{endsimulation}|]
  ( FMUInterface(pdsgfmu2)
    [|{endsimulation}|]
    ( FMUInterface(samplerfmu)
      [|{endsimulation}|]
      FMUInterface(checkequalityfmu) ) )

ExampleFMIWrapper(t0,tN) =
  PDSGFMU(pdsgfmu1)
  [|{endsimulation}|]
  ( PDSGFMU(pdsgfmu2)
    [|{endsimulation}|]
    ( SamplerFMU(samplerfmu)
      [|{endsimulation}|]
      CheckEqualityFMU(checkequalityfmu) ) )

-- The refinement below is established by monotonicity.
-- assert FMIWrapper(0,endOfTime) [T= ExampleFMIWrapper(0,endOfTime)

---------------------------------
-- General behaviour of an FMU --
---------------------------------
FMUInterface(i) =
  let
    Instantiation =
```

```
      fmi2Instantiate.i?b -> (b == true & Instantiated(fmi2OK) [] b == false & RUN(FMUAPI(i)))

    Instantiated(st) =
      st == fmi2Fatal & RUN(FMUAPI(i))
      []
      not member(st,ErrorFlags) & (
        fmi2SetUpExperiment!i?starttime?stoptimedefined?stoptime?st -> Instantiated(st)
        []
        fmi2Set!i?x?v?st -> Instantiated(st)
        []
        fmi2GetFMUState.i?st -> Instantiated(st) -- ?s?st -> Instantiated(st)
        []
        fmi2SetFMUState.i?st -> Instantiated(st) -- ?s?st -> Instantiated(st)
        []
        fmi2EnterInitializationMode.i?st -> Instantiated(st)
        []
        fmi2ExitInitializationMode.i?st -> Instantiated(st)
        []
        fmi2Get.i?n?v?st -> Instantiated(st)
        []
        fmi2DoStep.i?t?hc?st -> Instantiated(st)
        []
        fmi2GetMaxStepSize.i?hc?st -> Instantiated(st)
        []
        fmi2GetBooleanStatusfmi2Terminated.i?b?st -> Instantiated(st)
        []
        fmi2Terminate.i?st -> Instantiated(st))
      []
      st != fmi2Fatal & fmi2FreeInstance.i?st -> if st == fmi2Fatal
                                                 then RUN(FMUAPI(i))
                                                 else Instantiation
  within
    Instantiation /\ endSimulation

----------------------------------------
-- Periodic Discrete Signal Generator --
----------------------------------------
Instantiation(i) = fmi2Instantiate.i!true -> SKIP

PDSGFMUProc(i) =
  let
    InstantiationMode = get.i.2?p -> (
      fmi2Set.i.1?a!fmi2OK -> set.i.1!a -> InstantiationMode
      []
      fmi2Set.i.2?p!fmi2OK -> set.i.2!p -> InstantiationMode
      []
      p != 0 & fmi2SetUpExperiment.i?t0!true?tN!fmi2OK ->
        setctime.i.t0 -> setetime.i.tN ->
        fmi2EnterInitializationMode.i!fmi2OK -> SKIP
      []
      p == 0 & fmi2SetUpExperiment.i?t0!true?tN!fmi2Fatal -> STOP
      []
      fmi2SetFMUState.i!fmi2OK -> InstantiationMode -- ?s!fmi2OK -> InstantiationMode
      []
      fmi2GetFMUState.i!fmi2OK -> InstantiationMode -- ?s!fmi2OK -> InstantiationMode
      )

    -- It has no inputs, so nothing can be set.
    InitializationMode =
      fmi2ExitInitializationMode.i!fmi2OK ->
      -- This is a part that cannot be automatically generated.
      getctime.i?t0 -> getetime.i?tN ->
      t0 <= tN & get.i.2?p -> p != 0 & (
        if t0%p==0 then
          get.i.1?a -> set.i.3.a -> SKIP
        else
          set.i.3.eps -> SKIP)
      -- It has no inputs, so nothing can be set.
      []
      fmi2SetFMUState.i!fmi2OK -> InitializationMode -- ?s!fmi2OK -> InitializationMode
      []
      fmi2GetFMUState.i!fmi2OK -> InitializationMode -- ?s!fmi2OK -> InitializationMode

    slaveInitialized(hc) =
      get.i.3?y ->
        (fmi2Get.i.1!y!fmi2OK -> slaveInitialized(hc)
         []
         -- This is a part that cannot be automatically generated.
         fmi2DoStep.i?t?ss!fmi2OK ->
           (get.i.2?p ->
              p != 0 & if t%p==0 then
                         get.i.1?a -> set.i.3.a -> slaveInitialized(ss)
                       else
```

```
                        set.i.3.eps -> slaveInitialized(ss))
          []
          fmi2GetMaxStepSize.i!hc!fmi2OK -> slaveInitialized(hc)
          []
          fmi2SetFMUState.i!fmi2OK -> slaveInitialized(hc) -- ?s!fmi2OK -> slaveInitialized(hc)
          []
          fmi2GetFMUState.i!fmi2OK -> slaveInitialized(hc) -- ?s!fmi2OK -> slaveInitialized(hc)
          )
  within
    InstantiationMode;
    InitializationMode;
    (slaveInitialized(1) /\ fmi2Terminate.i!fmi2OK -> fmi2FreeInstance.i!fmi2OK -> STOP)

-- This can be defined in a more generic way in Circus
-- using the input to the model generation that describes
-- inputs, outputs, and exposed variables.
-- State
PDSGFMUState(i,t,st,three.a.p.y) =
  get.i.1!a -> PDSGFMUState(i,t,st,three.a.p.y)
  []
  get.i.2!p -> PDSGFMUState(i,t,st,three.a.p.y)
  []
  get.i.3!y -> PDSGFMUState(i,t,st,three.a.p.y)
  []
  getctime.i!t -> PDSGFMUState(i,t,st,three.a.p.y)
  []
  getetime.i!st -> PDSGFMUState(i,t,st,three.a.p.y)
  []
  set.i.1?na -> PDSGFMUState(i,t,st,three.na.p.y)
  []
  set.i.2?np -> PDSGFMUState(i,t,st,three.a.np.y)
  []
  set.i.3?ny -> PDSGFMUState(i,t,st,three.a.p.ny)
  []
  setctime.i?ct -> PDSGFMUState(i,ct,st,three.a.p.y)
  []
  setetime.i?et -> PDSGFMUState(i,t,et,three.a.p.y)
  []
  fmi2SetFMUState.i.fmi2OK -> PDSGFMUState(i,t,st,three.a.p.y) -- .three?na?np?ny!fmi2OK -> PDSGFMUState(i,t,st,three.na.np.ny) --
  []
  fmi2GetFMUState.i.fmi2OK -> PDSGFMUState(i,t,st,three.a.p.y) -- .three!a!p!y!fmi2OK -> PDSGFMUState(i,t,st,three.a.p.y) --

-- Complete process
PDSGFMU(i) =
  Instantiation(i);
  ( PDSGFMUProc(i)
      [|{|get.i,set.i,getctime,getetime,setctime,setetime,fmi2SetFMUState,fmi2GetFMUState|}|]
    PDSGFMUState(i,0,0,three.eps.eps.eps)
  ) \ {|get.i,set.i,getctime,getetime,setctime,setetime|}
  /\ endSimulation

assert FMUInterface(pdsgfmu1) [T= PDSGFMU(pdsgfmu1)

-------------
-- Sampler --
-------------
-- Main action
SamplerFMUProc(i) =
  let
    -- It has no parameters, so nothing can be set.
    InstantiationMode =
      fmi2SetUpExperiment.i?t0!true?tN!fmi2OK ->
        setctime.i.t0 -> setetime.i.tN ->
      fmi2EnterInitializationMode.i!fmi2OK -> SKIP
      []
      fmi2SetFMUState.i!fmi2OK -> InstantiationMode -- ?s!fmi2OK -> InstantiationMode
      []
      fmi2GetFMUState.i!fmi2OK -> InstantiationMode -- ?s!fmi2OK -> InstantiationMode

    InitializationMode =
      fmi2Set.i.1?x!fmi2OK -> set.i.1!x -> InitializationMode
      []
      fmi2Set.i.2?s!fmi2OK -> set.i.2!s -> InitializationMode
      []
      fmi2ExitInitializationMode.i!fmi2OK ->
      -- This is a part that cannot be automatically generated.
      getctime.i?t0 -> getetime.i?tN ->
      t0<=tN &
        get.i.2?s ->
          ( if s != eps then
              get.i.1?x -> set.i.2!x -> set.i.3!x -> SKIP
            else
              set.i.2!eps -> set.i.3!eps -> SKIP )
```

```
      []
      fmi2SetFMUState.i!fmi2OK -> InitializationMode -- ?s!fmi2OK -> InitializationMode
      []
      fmi2GetFMUState.i!fmi2OK -> InitializationMode -- ?s!fmi2OK -> InitializationMode

    slaveInitialized(hc) =
      get.i.3?y ->
        (fmi2Set.i.1?xv!fmi2OK -> set.i.1!xv -> slaveInitialized(hc)
         []
         fmi2Set.i.2?sv!fmi2OK -> set.i.2!sv -> slaveInitialized(hc)
         []
         -- This is a part that cannot be automatically generated.
         fmi2DoStep.i?t?ss!fmi2OK -> get.i.2?s ->
           ( if s != eps then
               get.i.1?x -> set.i.2!x -> set.i.3!x -> slaveInitialized(ss)
             else
               set.i.2!eps -> set.i.3!eps -> slaveInitialized(ss))
         []
         fmi2Get.i.3!y!fmi2OK -> slaveInitialized(hc)
         []
         fmi2GetMaxStepSize.i!hc!fmi2OK -> slaveInitialized(hc)
         []
         fmi2SetFMUState.i!fmi2OK -> slaveInitialized(hc) -- ?s!fmi2OK -> slaveInitialized(hc)
         []
         fmi2GetFMUState.i!fmi2OK -> slaveInitialized(hc) -- ?s!fmi2OK -> slaveInitialized(hc)
         )

  within
    InstantiationMode ;
    InitializationMode ;
    ( slaveInitialized(1) /\ fmi2Terminate.i!fmi2OK -> fmi2FreeInstance.i!fmi2OK -> STOP)

SamplerFMUState(i,t,st,three.x.s.y) =
  let
    SFMUS(t,st,three.x.s.y) =
      get.i.1!x -> SFMUS(t,st,three.x.s.y)
      []
      get.i.2!s -> SFMUS(t,st,three.x.s.y)
      []
      get.i.3!y -> SFMUS(t,st,three.x.s.y)
      []
      getctime.i!t -> SFMUS(t,st,three.x.s.y)
      []
      getetime.i!st -> SFMUS(t,st,three.x.s.y)
      []
      set.i.1?nx -> SFMUS(t,st,three.nx.s.y)
      []
      set.i.2?ns -> SFMUS(t,st,three.x.ns.y)
      []
      set.i.3?ny -> SFMUS(t,st,three.x.s.ny)
      []
      setctime.i?ct -> SFMUS(ct,st,three.x.s.y)
      []
      setetime.i?et -> SFMUS(t,et,three.x.s.y)
      []
      fmi2SetFMUState.i.fmi2OK -> SFMUS(t,st,three.x.s.y) -- .three?nx?ns?ny!fmi2OK -> SFMUS(t,st,three.nx.ns.ny) --
      []
      fmi2GetFMUState.i.fmi2OK -> SFMUS(t,st,three.x.s.y) -- .three!x!s!y!fmi2OK -> SFMUS(t,st,three.x.s.y) --
    within
      SFMUS(t,st,three.x.s.y)

-- Main process
SamplerFMU(i) =
  Instantiation(i);
  ( SamplerFMUProc(i)
      [|{|get.i,set.i,getctime,getetime,setctime,setetime,fmi2SetFMUState,fmi2GetFMUState|}|]
    SamplerFMUState(i,0,0,three.eps.eps.eps)
  ) \ {|get.i,set.i,getctime,getetime,setctime,setetime|}
  /\ endSimulation

assert FMUInterface(samplerfmu) [T= SamplerFMU(samplerfmu)

--------------------
-- Check Equality --
--------------------
-- Main action
CheckEqualityFMUProc(i) =
  let
    -- It has no parameters, so nothing can be set.
    InstantiationMode =
      fmi2SetUpExperiment.i?t0!true?tN!fmi2OK ->
        setctime.i.t0 -> setetime.i.tN ->
      fmi2EnterInitializationMode.i!fmi2OK -> SKIP
```

```
      []
      fmi2SetFMUState.i!fmi2OK -> InstantiationMode -- ?s!fmi2OK -> InstantiationMode
      []
      fmi2GetFMUState.i!fmi2OK -> InstantiationMode -- ?s!fmi2OK -> InstantiationMode

    InitializationMode =
      fmi2Set.i.1?x1!fmi2OK -> set.i.1!x1 -> InitializationMode
      []
      fmi2Set.i.2?x2!fmi2OK -> set.i.2!x2 -> InitializationMode
      []
      fmi2ExitInitializationMode.i!fmi2OK ->
      -- This is a part that cannot be automatically generated.
        (getctime.i?t0 -> getetime.i?tN -> t0 <= tN & SKIP)
      []
      fmi2SetFMUState.i!fmi2OK -> InitializationMode -- ?s!fmi2OK -> InitializationMode
      []
      fmi2GetFMUState.i!fmi2OK -> InitializationMode -- ?s!fmi2OK -> InitializationMode

    -- It has no outputs, so nothing can be gotten.
    slaveInitialized(hc) =
      fmi2Set.i.1?x1v!fmi2OK -> set.i.1!x1v -> slaveInitialized(hc)
      []
      fmi2Set.i.2?x2v!fmi2OK -> set.i.2!x2v -> slaveInitialized(hc)
      []
      -- This is a part that cannot be automatically generated.
      fmi2DoStep.i?t?ss!fmi2OK -> slaveInitialized(ss)
      []
      fmi2GetMaxStepSize.i!hc!fmi2OK -> slaveInitialized(hc)
      []
      fmi2SetFMUState.i!fmi2OK -> slaveInitialized(hc) -- ?s!fmi2OK -> slaveInitialized(hc)
      []
      fmi2GetFMUState.i!fmi2OK -> slaveInitialized(hc) -- ?s!fmi2OK -> slaveInitialized(hc)

  within
    InstantiationMode ;
    InitializationMode ;
    (slaveInitialized(1) /\ fmi2Terminate.i!fmi2OK -> fmi2FreeInstance.i!fmi2OK -> STOP)

CheckEqualityFMUState(i,t,st,two.x1.x2) =
  let
    CEFMUS(t,st,two.x1.x2) =
      get.i.1!x1 -> CEFMUS(t,st,two.x1.x2)
      []
      get.i.2!x2 -> CEFMUS(t,st,two.x1.x2)
      []
      getctime.i!t -> CEFMUS(t,st,two.x1.x2)
      []
      getetime.i!st -> CEFMUS(t,st,two.x1.x2)
      []
      set.i.1?nx1 -> CEFMUS(t,st,two.nx1.x2)
      []
      set.i.2?nx2 -> CEFMUS(t,st,two.x1.nx2)
      []
      setctime.i?ct -> CEFMUS(ct,st,two.x1.x2)
      []
      setetime.i?et -> CEFMUS(t,et,two.x1.x2)
      []
      fmi2SetFMUState.i.fmi2OK -> CEFMUS(t,st,two.x1.x2) -- .two?nx1?nx2!fmi2OK -> CEFMUS(t,st,two.nx1.nx2) --
      []
      fmi2GetFMUState.i.fmi2OK -> CEFMUS(t,st,two.x1.x2) -- .two!x1!x2!fmi2OK -> CEFMUS(t,st,two.x1.x2) --
    within
      CEFMUS(t,st,two.x1.x2)

-- Process
CheckEqualityFMU(i) =
  Instantiation(i);
  ( CheckEqualityFMUProc(i)
      [|{|get.i,set.i,getctime,getetime,setctime,setetime,fmi2SetFMUState,fmi2GetFMUState|}|]
    CheckEqualityFMUState(i,0,0,two.eps.eps)
   ) \ {|get.i,set.i,getctime,getetime,setctime,setetime|}

  /\ endSimulation

assert FMUInterface(checkequalityfmu) [T= CheckEqualityFMU(checkequalityfmu)

-------------------
-- Cosimulations --
-------------------
Cosimulation(t0,tN) =
  ( MAlgorithm(t0,tN)
    [| union(FMIAPI,{endsimulation}) |]
    FMIWrapper(t0,tN) ) \ {|fmi2Instantiate,
                fmi2SetUpExperiment,
```

```
                fmi2EnterInitializationMode,
                            fmi2ExitInitializationMode,
                            fmi2GetBooleanStatusfmi2Terminated,
                            fmi2GetMaxStepSize,
                            fmi2Terminate,
                            fmi2FreeInstance,
                fmi2GetFMUState,
                            fmi2SetFMUState,
                            endsimulation|}

-- Refinement is established by monotonicity

-------------
-- Classic --
-------------
ClassicCosimulation(t0,hc,tN) =
  ( ClassicMAlgorithm(t0,hc,tN)
    [| union(FMIAPI,{endsimulation}) |]
    ExampleFMIWrapper(t0,tN) ) \ {|fmi2Instantiate,
                        fmi2SetUpExperiment,
                        fmi2EnterInitializationMode,
                                    fmi2ExitInitializationMode,
                                    fmi2GetBooleanStatusfmi2Terminated,
                                    fmi2GetMaxStepSize,
                                    fmi2Terminate,
                                    fmi2FreeInstance,
                        fmi2GetFMUState,
                                    fmi2SetFMUState,
                                    endsimulation|}

assert ClassicCosimulation(0,2,endOfTime) :[deterministic]

assert ClassicCosimulation(0,2,endOfTime) ; IDLE :[deadlock free]

assert ClassicCosimulation(0,2,endOfTime) :[livelock free]

--------------
-- Simulink --
--------------
SimulinkCosimulation(t0,tN) =
  (SimulinkMAlgorithm(t0,tN)
    [|union(FMIAPI,{endsimulation})|]
   ExampleFMIWrapper(t0,tN) ) \ {|fmi2Instantiate,
                         fmi2SetUpExperiment,
                       fmi2EnterInitializationMode,
                                   fmi2ExitInitializationMode,
                                   fmi2GetBooleanStatusfmi2Terminated,
                                   fmi2GetMaxStepSize,
                                   fmi2Terminate,
                                   fmi2FreeInstance,
                       fmi2GetFMUState,
                                   fmi2SetFMUState,
                                   endsimulation|}

assert SimulinkCosimulation(0,endOfTime) :[deterministic]

assert SimulinkCosimulation(0,endOfTime) ; IDLE :[deadlock free]

assert SimulinkCosimulation(0,endOfTime) :[livelock free]

------------
-- Broman --
------------
VariableStepWithRollbackCosimulation(t0,hc,tN) =
  (VariableStepWithRollbackMAlgorithm(t0,hc,tN)
    [|union(FMIAPI,{endsimulation})|]
    ExampleFMIWrapper(t0,tN) ) \ {|fmi2Instantiate,
                        fmi2SetUpExperiment,
                        fmi2EnterInitializationMode,
                                    fmi2ExitInitializationMode,
                                    fmi2GetBooleanStatusfmi2Terminated,
                                    fmi2GetMaxStepSize,
                                    fmi2Terminate,
                                    fmi2FreeInstance,
                        fmi2GetFMUState,
                                    fmi2SetFMUState,
                                    endsimulation|}

assert VariableStepWithRollbackCosimulation(0,2,endOfTime) :[deterministic]

assert VariableStepWithRollbackCosimulation(0,2,endOfTime) ; IDLE :[deadlock free]

assert VariableStepWithRollbackCosimulation(0,2,endOfTime) :[livelock free]
```

```
--------------
-- Scenario --
--------------

SynchronousEventsSpec =
  AllInInternalChoice({fmi2Set.1.1.1.fmi2OK,fmi2Set.1.2.1.fmi2OK,fmi2Set.2.1.1.fmi2OK,fmi2Set.2.2.2.fmi2OK}) ;
  AllInInternalChoice({fmi2Set.3.1.1.fmi2OK,fmi2Set.3.2.1.fmi2OK,fmi2Set.4.1.1.fmi2OK,fmi2Set.4.2.1.fmi2OK}) ;
  AllInInternalChoice({fmi2Get.1.1.1.fmi2OK,fmi2Get.2.1.1.fmi2OK,fmi2Get.3.3.1.fmi2OK}) ;
  AllInInternalChoice({fmi2Set.3.1.1.fmi2OK,fmi2Set.3.2.1.fmi2OK,fmi2Set.4.1.1.fmi2OK,fmi2Set.4.2.1.fmi2OK}) ;
  (fmi2DoStep.1.0.2.fmi2OK -> SKIP) ;
  (fmi2DoStep.2.0.2.fmi2OK -> SKIP) ;
  (fmi2DoStep.3.0.2.fmi2OK -> SKIP) ;
  (fmi2DoStep.4.0.2.fmi2OK -> SKIP) ;
  AllInInternalChoice({fmi2Get.1.1.1.fmi2OK,fmi2Get.2.1.1.fmi2OK,fmi2Get.3.3.1.fmi2OK}) ;
  AllInInternalChoice({fmi2Set.3.1.1.fmi2OK,fmi2Set.3.2.1.fmi2OK,fmi2Set.4.1.1.fmi2OK,fmi2Set.4.2.1.fmi2OK}) ;
  (fmi2DoStep.1.2.2.fmi2OK -> SKIP) ;
  (fmi2DoStep.2.2.2.fmi2OK -> SKIP) ;
  (fmi2DoStep.3.2.2.fmi2OK -> SKIP) ;
  (fmi2DoStep.4.2.2.fmi2OK -> SKIP)

-- assert SynchronousEventsSpec [FD= ClassicCosimulation(0,2,2) -- RiP

-- assert SynchronousEventsSpec [FD= SimulinkCosimulation(0,2)

-- assert SynchronousEventsSpec [FD= VariableStepWithRollbackCosimulation(0,2,2)

----------------------------
-- Deterministic Scenario --
----------------------------

DSynchronousEventsSpec =
  -- Setting parameters
  fmi2Set.1.1.1.fmi2OK -> fmi2Set.1.2.1.fmi2OK -> fmi2Set.2.1.1.fmi2OK -> fmi2Set.2.2.2.fmi2OK ->
  -- Setting 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
  -- First step
  -- Take outputs
  fmi2Get.1.1.1.fmi2OK -> fmi2Get.2.1.1.fmi2OK -> fmi2Get.3.3.1.fmi2OK ->
  -- Distribute inputs
  fmi2Set.3.1.1.fmi2OK -> fmi2Set.3.2.1.fmi2OK -> fmi2Set.4.2.1.fmi2OK -> fmi2Set.4.1.1.fmi2OK ->
  -- Step
  fmi2DoStep.1.0.2.fmi2OK -> fmi2DoStep.2.0.2.fmi2OK -> fmi2DoStep.3.0.2.fmi2OK -> fmi2DoStep.4.0.2.fmi2OK ->
  -- Second step
  -- Take outputs
  fmi2Get.1.1.1.fmi2OK -> fmi2Get.2.1.1.fmi2OK -> fmi2Get.3.3.1.fmi2OK ->
  -- Distribute inputs
  fmi2Set.3.1.1.fmi2OK -> fmi2Set.3.2.1.fmi2OK -> fmi2Set.4.2.1.fmi2OK -> fmi2Set.4.1.1.fmi2OK ->
  -- Step
  fmi2DoStep.1.2.2.fmi2OK -> fmi2DoStep.2.2.2.fmi2OK -> fmi2DoStep.3.2.2.fmi2OK -> fmi2DoStep.4.2.2.fmi2OK ->
  SKIP

assert ClassicCosimulation(0,2,2) [T= DSynchronousEventsSpec

assert SimulinkCosimulation(0,2) [T= DSynchronousEventsSpec

assert VariableStepWithRollbackCosimulation(0,2,2) [T= DSynchronousEventsSpec

-- Auxiliary functions
index(<v>^vals,pos) = if pos == 1 then v else index(vals,pos-1)

max(a,b) = if a >= b then a else b
min(a,b) = if a >= b then b else a

min4(a,b,c,d) = min(min(a,b),min(c,d))

-- Auxiliary definitions to specify scenarios
AllInInternalChoice(s) =
  if (empty(s))
  then SKIP
  else |~| e: s @ e -> AllInInternalChoice(diff(s,{e}))

AllInInternalofExternalChoices(ss) =
  if (empty(ss))
  then SKIP
  else |~| cs: ss @ ([] e: cs @ e -> AllInInternalChoice(diff(cs,{e})))

channel printme: {(-10)..10}
```

$$
\begin{array}{l}
\textbf{process}\ \ FMUSketch \mathrel{\widehat=} i : FMI2COMP \bullet \textbf{begin} \\
\textbf{state}\ State = [currentTime, endTime : TIME;\ cp_i, cinp_i, cev_i, cout_i] \\
Instantiation = fmi2Instantiate.i!true \longrightarrow \textbf{Skip} \\
InstantiationMode = \\
\quad fmi2Set.i.p_i?v!fmi2OK \longrightarrow cp_i := v;\ InstantiationMode \\
\quad \Box \\
\quad fmi2SetUpExperiment.i?t0!true?tN!fmi2OK \longrightarrow \\
\qquad currentTime, endTime := t0, tN; \\
\qquad fmi2EnterInitializationMode.i!fmi2OK \longrightarrow \textbf{Skip} \\
InitializationMode = \\
\quad fmi2Set.i.inp_i?v!fmi2OK \longrightarrow cinp_i := v;\ InitializationMode \\
\quad \Box \\
\quad fmi2ExitInitializationMode.i!fmi2OK \longrightarrow UpdateState \\
slaveInitialized = \\
\quad fmi2Get.i.out_i!cout_i!fmi2OK \longrightarrow slaveInitialized \\
\quad \Box \\
\quad fmi2Set.i.inp_i?v.fmi2OK \longrightarrow cinp_i := v;\ slaveInitialized \\
\quad \Box \\
\quad fmi2DoStep.i?t?ss!fmi2OK \longrightarrow (UpdateState;\ slaveInitialized) \\
\bullet\ Instantiation;\ InstantiationMode;\ InitializationMode; \\
\quad (slaveInitialized \triangle \\
\qquad fmi2Terminate.i!fmi2OK \longrightarrow fmi2FreeInstance.i!fmi2OK \longrightarrow \textbf{Stop}) \\
\textbf{end}
\end{array}
$$

Figure 6: Sketch of a model for a specific FMU

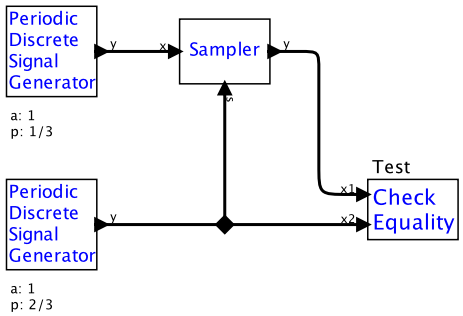


Figure 7: Test case for sampling of discrete event signals [6]

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

state

$State = [oldOuts, newOuts : (FMI2COMP \times VAR) \pfun VAL;\ currentSS : NZTIME]$

Init

$State'$

$\text{dom}\ oldOuts' = \text{ran}\ outputs \land \text{ran}\ oldOuts = \epsilon \land newOuts' = \varnothing$

$currentSS' = initialSS$

$Monitor \mathrel{\widehat=}; out : outputs \bullet$

$fmi2Get.(FMU\ out).(name\ out)?nv?st \longrightarrow newOuts := newOuts \oplus \{out \mapsto nv\}$

$Adjust \mathrel{\widehat=}$ **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

Figure 8: Model of *VaryStep*

```
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
```

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

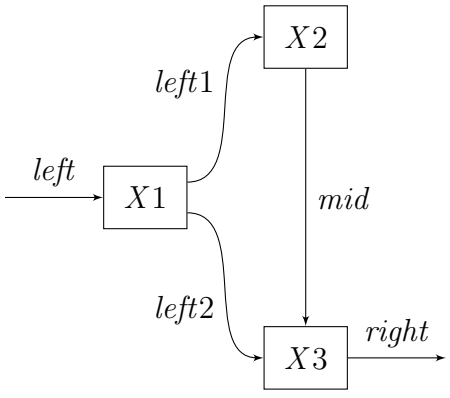


Figure 10: A data-flow example

```
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)
```

Figure 11: Data flow specification