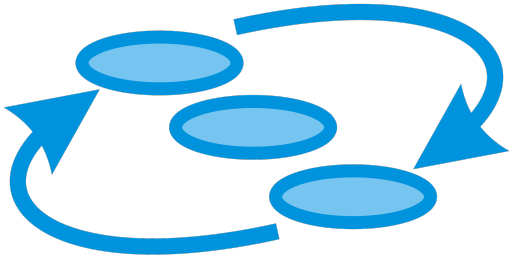
Grant Agreement: 644047

INtegrated TOol chain for model-based design of CPSs



Foundations for FMI Co-Modelling

Deliverable Number: D2.1d

Version: 0.4

Date: December 2015

Public Document

http://into-cps.au.dk

Contributors:

Nuno Amálio, UY
Ana Cavalcanti, UY
Christian König, TWT
Jim Woodcock, UY

Editors:

Christian König, TWT

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	21-05-2015	Christian König	Initial document version
0.2	09-09-2015	Nuno Amálio	Added section on FMI literature review
0.3	15-11-2015	Nuno Amálio	Added sections on FMI Semantics, formal analysis of semantics, introduction and conclusions
0.4	15-12-2015	Nuno Amálio	Revised document according to feedback coming from the internal project reviews.

Abstract

The evolving FMI is becoming a well accepted industrial standard for collaborative modelling and co-simulation. It enables the composition of different models developed using distinct tools and is seen as an important driver in enabling tool interoperability in the area of cyber-physical systems. This deliverable presents our work on defining formal foundations for FMI co-simulation in the context of the INTO-CPS project. It presents our review of the literature in the area of formal approaches to the FMI, and presents a formal semantics of the FMI described in the formal specification language CSP. The CSP model presented here is effectively a UTP model, derived from CSP's UTP semantics; UTP, however, allows for much richer models. This deliverable presents our results establishing the feasibility and a general approach to define reactive models for FMI co-simulations. The model presented here will evolve into a INTO-CSP model, a hybrid version of CSP with a UTP semantics that is under development. The model presented was formally analysed with the FDR3 refinement-checker.

Contents

1 Introduction

The Functional Mock-up Interface (FMI) [FMI14], introduced in the ITEA project MODELISAR[1] and now being further developed and maintained under the umbrella of the Modelica association[2], is a standard whose ultimate goal is collaborative design and simulation of separately developed systems.

FMI emerges from industrial needs. It tries to facilitate the cooperation between different companies, such as automotive original equipment manufacturers (OEMs) and suppliers, to enable the exchange and simulation of separately developed models. Although FMI's inception is associated with the automotive industry (supported by major companies such as Daimler or Bosch), it is a standard open to any domain and designed to embrace heterogeneity.

FMI has been applied across a variety of different domains. Project Modrio[3] applied FMI to the domains of energy and aerospace using Modelica. The more recent Acosar project[4] is currently defining advanced co-simulation interfaces for real-time systems integration.

To ensure the standard's growth, a formal development process was adopted in 2015, defining mechanisms to propose and integrate new features to be incorporated in future versions of FMI. FMI's growing importance is accompanied by the development of tools that support FMI; a list of such tools can be found at `https://www.fmi-standard.org/tools`.

In the FMI setting, a model is organised around black-box slave FMUs (Functional Mockup Units) — effectively, wrappings of model simulators — that are interconnected through their inputs and outputs [BBG+13]. FMUs are passive entities whose simulation is triggered and orchestrated by a master algorithm (MA) [BBG+13]. The simulation process is divided into simulation steps that serve as synchronisation and data exchange points; between these steps the FMUs are simulated independently.

This deliverable presents the efforts of the INTO-CPS project on the formal foundations of the FMI. The main research output reported here is a formal semantics of the FMI described in the formal specification language CSP [Hoa85], which acts as a front-end for a UTP (Unifying Theories of

[1] `https://itea3.org/project/modelisar.html`
[2] `https://www.fmi-standard.org/`
[3] https://itea3.org/project/modrio.html
[4] `https://itea3.org/project/acosar.html`

Programming [HJ98]) semantic model based on the UTP semantics of CSP. This constitutes a crucial step in defining formal semantics for FMI-based co-simulations, which is a goal of WP2. Ultimately, our semantics will be expressed in INTO-CSP, an hybrid version of CSP, also with a UTP semantics. In the work reported in this document, we make use of the FDR3 refinement checker [GRABR14], which is not considered part of the INTO-CPS tool chain, but is used solely to support the development of the foundation work presented here.

The remainder of this deliverable is organised as follows:

- Section 2 gives some background on FMI for co-simulation, the target of the formal investigation presented here, and the formal specification language CSP.
- Section 3 presents the state of the art on applications of the FMI and the formal work that has been carried out in connection with FMI.
- Section 4 presents the CSP semantics of the FMI, which includes also the specification of one simple fixed-step MA and one variable-step MA with support for rollback.
- Section 5 presents the machine-assisted formal analysis that has been undertaken to validate and verify the FMI semantics using the FDR3 model-checking tool.
- Section 6 draws the conclusions of this report.
- Appendix A presents the CSPm specifications of the FMI semantics, which were subject to a formal analysis with FDR3.

2 Background

This section gives some background on FMI for co-simulation and the formal specification language CSP.

2.1 FMI co-simulation

Modelling and simulation of complex engineering systems, known as cyber-physical systems (CPSs) [DLV12], typically involves components from different engineering fields. A global system is decomposed into subsystems; each subsystem is tackled by a team of engineers specialised in some domain using

specialised modelling and simulation tools. This approach carries the benefits of modularity, applying in practice the important engineering principle of *separation of concerns* [Par72, TOHSMS99].

A modularised model needs to be composed (or coupled) to yield a global model [KS00]. This composition can be carried out at two different levels: (a) *model-description level* (the equations that make up the subsystems are combined), and (b) *execution or simulation level* (the executed or simulated behaviours of the different models are coupled). The latter is known as *co-simulation*.

Co-simulation, an approach for the joint simulation of models developed with different tools (tool coupling) where each tool treats one part of a modular coupled problem [BCWS11], aims at tool *interoperability* [Weg96] to facilitate modelling and simulation of the intrinsincally heterogeneous CPSs. Each submodel can make use of the tool and notation that is most appropriate for the task at hand. In technical terms, co-simulation is a simulation technique for coupled problems that restricts the communication and exchange between subsystems to discrete communication points [SA12].

FMI [BOA+11, BOA+12, FMI14][5] is an evolving standard for tool coupling that intends to support a wide variety of tools to avoid the need for point-to-point solutions [BCWS11]. It was introduced to (i) facilitate exchange of dynamic models, (ii) co-simulate heterogeneous models, and (iii) protect product know-how and intellectual property [BOA+11]. FMI is a result of the MODELISAR research project [BOA+12]; it proposes a black-box model made up of interfaces to enable model exchange and co-simulation that hides the implementation details of the coupled components, protecting the intellectual property.

FMI [FMI14] is divided in two parts: *model-exchange* and *co-simulation*. Typically, model-exchange takes place when the FMU is imported into a simulation tool and handled as a black box. This section describes FMI co-simulation, a fundamental ingredient of INTO-CPS's interoperability goal.

FMI's co-simulation is based on a *master-slave architecture* [BCWS11]. A master algorithm (MA) triggers and orchestrates a collections of FMUs to bring about a co-simulation of different model parts [BBG+13]. Simulation is divided into steps that serve as synchronisation and data exchange points; between these steps the FMUs are simulated independently. A typical co-simulation scenario can be realised either as a *stand-alone* FMU which

[5] `https://www.fmi-standard.org/`

contains runnable code, or as a *tool coupling* FMU, which contains a wrapper to a native simulation tool. The two scenarios are depicted in Figure 1 (from [FMI14]). In the case of a stand-alone FMU, the whole simulation model including its solver is contained in the FMU, either as source-code or binary file. In tool-coupling, the native simulation tool, controlled by an API, is required for execution.

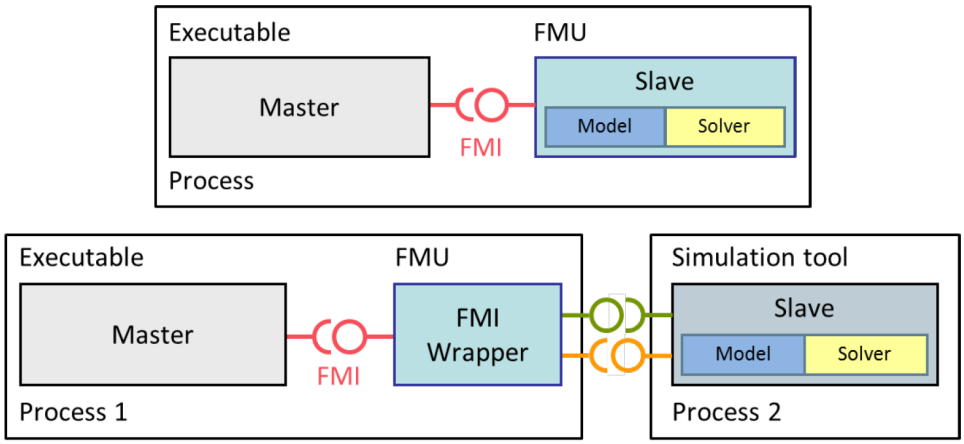


Figure 1: co-simulation with stand-alone model (top) and tool coupling (bottom) — from [FMI14]

When the co-simulation is started, the models of the co-simulation are solved independently between two discrete communication points (tc_i). At these communication points, the output signals of the models are read, and their input signals are set, by the MA. At a communication point, the models are synchronized, such that the MA waits for all models to simulate up to this communication point, before the simulation is advanced in time. A simple co-Simulation MA uses the following sequence of FMI commands:

at tc_i:
`fmi2SetXXX` on inputs
`fmi2DoStep`
at tc_{i+1}:
`fmi2GetXXX` on outputs
`fmi2SetXXX` on inputs
at tc_{i+2}:
`fmi2DoStep`
`fmi2GetXXX` on outputs

⋮

The commands above are as follows:

- The `fmi2SetXXX` and `fmi2GetXXX` commands set the input signals and retrieve the output signals of a model. The `XXX` is replaced with the data type, for example `fmi2GetReal` for real variables.
- The `fmi2DoStep` command advances the co-simulation by the time-step of the given communication step size. It returns different messages: `fmi2OK` indicates that the slave has performed the simulation up to the requested point in time, `fmi2Discard` means that only a part of the time interval could be computed successfully, while `fmi2Error` indicates that the computation could not be performed at all.

2.2 CSP

Communicating Sequential Processes (CSP) [Hoa85, Sch00, Ros10b], a formal specification language introduced by Hoare [Hoa85] that is part of a class of languages that are known as *process algebras*, aims at describing communicating processes and interaction-driven computations.

CSP's domain of discourse consists of processes, which are self-contained components with particular interfaces through which they interact with their environment. The interface of a process is described as a set of events, which describe atomic, indivisible and instantaneous actions. A process is, therefore, characterised by the events it can engage in and their ordering. CSP is supported by an underlying theory to enable reasoning and model analysis about interaction and communication in this event-based model of interaction.

In CSP, events are transmitted along *communication channels*, which carry messages of particular types. A channel has a set of associated events, corresponding to all messages that may be carried through the channel.

Process expressions are built out of events using a number of operators:

- *Event prefixing*, expressed in CSP as $e \rightarrow P$, describes a process that expects event e and then behaves as process P.
- *Interleaving*, described in CSP as $P_1 \;|||\; P_2$, defines a composition of two processes that execute in parallel without any synchronisation.

The iterated version of interleaving, applies interleaving to any number of indexed processes: $\Big|\Big|\Big| i : N \bullet P(i)$.

- *Parallel*, $P_1 \underset{A}{\parallel} P_2$, describes the composition of two processes that execute in parallel synchronising on the set A of events.
- *Sequential*, $P_1; P2$, describes a process that executes P_1 until it terminates, and then executes P_2.
- *Hiding*, $P \setminus N$, makes a set N of events internal to a process P.
- *Interrupt*, $P_1 \triangle P_2$, defines a composition that behaves like P_1, but can be interrupted by a synchronisation on one of the initial events of P_2, which then takes over.
- *Throw*, $P_1 \underset{A}{\ominus} P2$, a relatively recent CSP operator [Ros10a], defines a form of interrupt where any occurrence of an event $e \in A$ within P_1 hands control to P_2.

Every CSP process P has an alphabet αP. Its semantics is given using four models: traces, failures, divergences and infinite traces. These are understood as observations of possible executions of the process P, in terms of the events from αP that it can engage in, refuse, or lead to divergence.

3 FMI Formally: state of the art

We present a survey of the FMI-related literature to better understand the trends in the FMI related work (section 3.1) and the different ways in which it has been formalised (section 3.2).

3.1 FMI Trends

FMI has, so far, emphasised the modelling of the dynamics of physical systems [BGL+14]. This reflects the origins of the FMI as a way to achieve interoperability of simulators for models of automative suppliers. To enlarge the scope of FMI, there has been a call to make the FMI a *hybrid co-simulation* standard [BGL+14], supporting hybrid systems [MMP92] that combine continuous and discrete dynamics. [BGL+14] argues that small extensions to FMI are sufficient to satisfy the requirements of hybrid co-simulation.

FMI has been the focus of several research works. Bastian et al [BCWS11] propose a fixed-step platform-independent MA. Schierz et al [SAC12] investigate adaptive communication size control in the FMI to improve the accuracy of simulations. Feldman et al [FGP14] generate FMUs from statecharts describing components (blocks) of a SysML model. Pohlmann et al [PSR+12] generate FMUs from a UML-based DSL for real-time systems; individual FMUs are generated from model components described as real-time statecharts. Denil et al [DMMV15] generate FMI co-simulation environments from hybrid multi-models using model transformations. These works give an indication on the popularity and level of prolificacy of FMI as a subject matter, as well as the trends and concerns in the FMI related work, but we now turn to the work that is more closely related to the approach presented here.

3.2 A Formal Treatment of FMI

To highlight and uncover issues with the evolving FMI standard, and to investigate the FMI formally, Broman et al [BBG+13] develop FMI's most influential formalisation to date. This formalisation of a relevant subset of the FMI standard consists of a state-based functional model; relevant procedures of FMI's API (abstract programming interface), which acts as the interface between MA and the slave FMUs, are modelled as functions. The paper studies FMI's main subject-matter: simulations of continuous models based on numerical methods to approximate differential equations. Desirable properties of such methods are *convergence* (whether the method approximates the solution) and *stability* (whether errors are damped out). To ensure convergence, [BBG+13] investigates determinism, studying the situations under which determinate execution is ensured — that is, different runs of a MA on a given FMU network give the same results. To avoid nondeterminism and the possibility of unexpected results, the paper proposes a way to check for the absence of *algebraic loops* (convergence cannot be guaranteed in the presence of algebraic loops) by checking the acyclicity of the directed graph describing the dependencies between input and output ports of FMUs. The MAs of [BBG+13] preclude configurations with algebraic loops and are proved to be determinate. Although the model is formal, the proofs of the paper's main theorems are rigorous and grounded on the paper's formalisation, but they are not formal.

It is interesting to comment on the emphasis that [BBG+13] puts on determinism. In a hybrid co-simulation setting made up of continuous and discrete

components, nondeterminism may not necessarily be a bad thing. Continuous models may have non-determinism when they have components that rely on random numbers, for instance. Abstract models of discrete systems are often nondeterministic; that is seen as a means to achieve abstraction [HJ89]. Although nondeterminism is often at odds with model executability [HJ89], the discrete modelling community has found ways of conciliating these two opposing forces. The formal model of [BBG+13] would have to be refactored to be applicable to a world of possibly non-deterministic discrete components as all the API procedures are modelled as total functions. Given that the authors of [BBG+13] observe in [BGL+14] that the FMI should be applicable to general-purpose hybrid systems, it appears that the emphasis on determinacy should be relaxed in a hybrid co-simulation setting.

Tripakis and Broman [TB14] use the model of [BBG+13] to investigate how FMI copes with heterogeneity. Specifically, they study how components with different underlying models of computation (MoC) can be encoded as FMUs; the supported MoCs of [TB14] are state machines, discrete event and synchronous dataflow. Given the lack of support for non-determinism in [BBG+13], all the considered discrete MoCs are restricted to be deterministic.

Savicks et al [SBC14a] propose an approach to cosimulate Event-B [Abr10] models and FMUs, outlining a general co-simulation framework for Event-B that is based on the master-slave architecture of FMI. The paper gives a sketch Event-B model describing the co-simulation semantics, which discriminates between discrete and continuous simulation steps. The approach is based on a simple master algorithm and it has been applied to an industrial case study in [SBC14b]. It provides support for the simulation of FMUs within the Event-B platform Rodin [ABH+10], but it is does not wrap Event-B models as FMUs to enable their general FMI-compliant co-simulation.

4 Semantics of FMI

This section presents the formal semantics of FMI expressed in CSP. The semantics presented here targets the co-simulation API.

This version of the semantics supports a simple model of time, with FMI communication time points represented as integers to enable an analysis

with the FDR3 refinement-checker [GRABR14][6]. This section presents an idealised version of the semantics. An FDR3 optimised version, expressed in the CSP dialect CSPm and developed based on bounded types to enable model-checking with FDR3, is given in appendix A.

4.1 Base definitions

This version of the semantics represents time as integers[7]. FMU variables and their values are also represented as integers. The allowed values of a step (used in integration algorithms and a parameter of FMI-based co-simulations) are defined as natural numbers greater or equal to 1:

$$\textbf{nametype}\ Time = \mathbb{Z}$$
$$\textbf{nametype}\ Var = \mathbb{Z}$$
$$\textbf{nametype}\ Val = \mathbb{Z}$$
$$\textbf{nametype}\ Step = \{n : \mathbb{N} \mid n \geq 1\}$$

We represent environments, assignments of variables to values, as partial functions (symbol $\nrightarrow$)[8]:

$$\textbf{nametype}\ Env = Var \nrightarrow Val$$

An FMU state (St) consists of three environments, recording inputs, outputs and exposed state, respectively. Constant *emptySt* represents the empty state, made up of three empty environments.

$$\textbf{nametype}\ St = Env.Env.Env$$
$$emptySt = \{\}.\{\}.\{\}$$

The definitions above use the CSP infix dot operator (.), which builds composites out of parts in a way that is akin to cartesian products.

Function *opt* below defines optionals for some given type, which may have a value (singleton set) or not (empty set). Function *the* operates upon optionals; it yields the element of a singleton:

$$opt(X) = \{s \mid s \in \mathbb{P}\ X \wedge \#\, s \leq 1\}$$
$$the(\{x\}) = x$$

[6]FDR stands for Failures Divergences Refinement; FDR3 tool is presented in section 5.1.

[7]This is to evolve into a representation of time as real numbers when INTO-CSP comes into play.

[8]In the FDR3 version, these are represented as FDR3 maps (see FDR definitions in appendix A.1).

The *VarKind* datatype indicates the type of some variable:

$$\textbf{datatype}\ VarKind = input \mid output \mid state$$

We introduce a type to represent the allowed FMI co-simulation intervals, made up of a start and an end-times:

$$\textbf{nametype}\ CosimInt = \{(st, et) \mid st \in Time \land et \in Time \land st \leq et\}$$

The next functions retrieve the components of a pair, such as the components of a co-simulation interval (start and end times):

$$fst((st, et)) = st$$
$$snd((st, et)) = et$$

The next predicate says whether a given time point ct is within a given co-simulation interval.

$$withinCosimInt(ct, (st, et)) \Leftrightarrow st \leq ct \land ct \leq et$$

Function *nextCurrentTm* issues the next communication time point, given a current time point ct, a step-size stp and a co-simulation interval $cint$. The next time point is issued provided ct is within the given $cint$; otherwise it returns ct.

$$nextCurrentTm(ct, stp, cint) = \begin{cases} ct + stp & \text{If } withinCosimInt(ct, cint) \\ ct & \text{otherwise} \end{cases}$$

The set *FMUIdx* represents FMU indices, which are used to identify particular FMUs from a FMU collection:

$$FMUIdx = \mathbb{P}\, Int$$

We now introduce several functions to manipulate the state of an FMU. Function *updEnv* updates an environment:

$$updEnv(x, v, e) = \begin{cases} e \oplus \{x \mapsto v\} & \text{if } x \in \text{dom } e \\ e, & \text{otherwise} \end{cases}$$

Above, $\oplus$ is the function overriding operator.

Function *updSt*, with signature $updSt : St \times VarKind \times Var \times Val \to St$, updates the state of an FMU; it is defined by the following three equations:

$$updSt(ins.outs.sts, input, x, v) = updEnv(x, v, ins).outs.sts$$
$$updSt(ins.outs.sts, output, x, v) = ins.updEnv(x, v, out).sts$$
$$updSt(ins.outs.sts, state, x, v) = ins.outs.updEnv(x, v, sts)$$

Above, $ins.outs.sts$ stand for a FMU state (a member of set St) made up of environments corresponding to inputs, outputs and exposed state.The equation describe how the overall state is updated as a result of a variable of kind input, output or state by resorting to function *updEnv*.

Function *getVarVal* gets the value associated with a variable in some environment; it returns a singleton set with the proper value if there is one, or the empty set otherwise.

$$getVarVal(x, e) = \begin{cases} \{e(x)\} & \text{if } x \in \text{dom } e \\ \{\} & \text{otherwise} \end{cases}$$

Function *getValInFMUSt* extracts the value of a given variable of given kind (input, output or state) from an FMU state. It is defined by the equations:

$$getValInFMUSt(ins.outs.sts, input, x) = getVarVal(x, ins)$$
$$getValInFMUSt(ins.outs.sts, output, x) = getVarVal(x, outs)$$
$$getValInFMUSt(ins.outs.sts, state, x) = getVarVal(x, sts)$$

4.2 The FMI API in CSP

The FMI API for co-simulation is described in [FMI14] as a collection of C function signatures, comprising a name, parameters and a return value. Calls to and returns from these C functions are represented in CSP as channels; some channels have an associated CSP datatype to represent types of parameters and return values. The correspondence between the CSP channels and the FMI's API is given in table 1; the channels are described in the remainder of this section.

An interaction on the channel *instantiate* gives life to FMUs by providing an FMU with a specific state; it communicates an FMU index and an FMU state. Channel *setup* is used to set up the simulation (or experiment) with a specific co-simulation interval. Channels *set* and *get* are used to set and retrieve the values of variables (unlike the C API, our formalisation abstracts

CSP	C
channel *instantiate*	C function `fmi2Instantiate`
channel *setup*	C function `fmi2SetupExperiment`
channel *get*	C function `fmi2GetXXX` (e.g. `fmi2GetReal`)
channel *set*	C function `fmi2SetXXX` (e.g. `fmi2SetReal`)
channel *endInit*	C function `fmi2ExitInitializationMode`
channel *doStep*	C function `fmi2DoStep`
data type *DoStepOutcome*	Enumeration `fmi2Status` restricted to the values relevant to `fmi2DoStep`.
channel *doStepOutcome*	C function `fmi2DoStep`
channel *stepCancel*	C function `fmi2CancelStep`
channel *stepFinished*	Functionality of callback function `stepFinished` provided with `fmi2Instantiate` or obtained by calling `fmi2GetStaus` after `fmi2DoStep` returns `fmi2Pending`.
channel *getState*	C Function `fmi2GetFMUState`.
channel *setState*	C Function `fmi2SetFMUState`.
channel *terminate*	C function `fmi2Terminate`

Table 1: Correspondence between CSP channels and datatypes and the FMI's API for co-simulation [FMI14]

from specific value types). A synchronisation on the channel *endInit* marks the end of an FMU initialisation.

$$\textbf{channel}\ instantiate : FMUIdx.St$$
$$\textbf{channel}\ setup : FMUIdx.CosimInt$$
$$\textbf{channel}\ set : FMUIdx.VarKind.Var.Val$$
$$\textbf{channel}\ get : FMUIdx.\{output, state\}.Var.opt(Val)$$
$$\textbf{channel}\ endInit : FMUIdx$$

Above, *opt* is used to say that *get* may not yield a value — the FMU may not have the variable being queried.

A synchronisation on the channel *doStep* triggers a simulation step on an FMU; it involves an FMU index to identify the FMU, a communication time point and a step value. The possible outcomes of a *doStep* operation are captured in the CSP datatype *DoStepOutcome*: *stepOk* indicates that the FMU carried the requested step, *pending* says that the FMU is still executing the step, *discard* informs that the given step value has been rejected, and *fail* signals that the FMU could not perform the requested step. Channel

doStepOutcome carries the result of a do step operation; it takes an FMU index and a particular *DoStepOutcome*.

channel $doStep : FMUIdx.Time.Step$

datatype $DoStepOutcome = stepOk \mid pending \mid discard \mid fail$

channel $doStepOutcome : FMUIdx.DoStepOutcome$

Channels *stepCancel* and *stepFinished* are related to the asynchronous mode of FMI step simulation and the *pending* outcome of step simulation. Channel *stepCancel* is used to tell the FMU to cancel the step as the MA no longer wishes to wait for the conclusion of the step operation; channel`stepFinished` is used to inform the MA of the conclusion of the requested step operation. Both these channels take an FMU index to refer to the involved FMU.

channel $stepCancel, stepFinished : FMUIdx$

To support a rollback mechanism, FMI 2.0 introduced functions to save and restore the state of an FMU to enable recovery when an FMU rejects a given communication step-size [BBG+13]. The corresponding channels carry the index of an FMU, a communication time point and an FMU state:

channel $getState, setState : FMUIdx.Time.St$

Finally, we have a channel that communicates the termination of a particular FMU:

channel $terminate : FMUIdx$

4.3 FMI co-simulation

Figure 2 casts FMI co-simulation in a CSP and process algebra setting. The notion that a MA coordinates an execution of FMUs is described as the parallel composition (CSP operator $\|$) of an MA, synchronised on the FMI API events defined in the previous section, with a network of FMUs that run in interleaving (operator $|||$). This is described in CSP by the formula:

$$Cosimulation = MA \underset{CosimEvs}{\|} |||\, i : ifmus \bullet FMU(i)$$

Here, *MA* is any master algorithm; *FMU* is the process that captures the behaviour of a FMU as defined by the FMI standard (defined below).

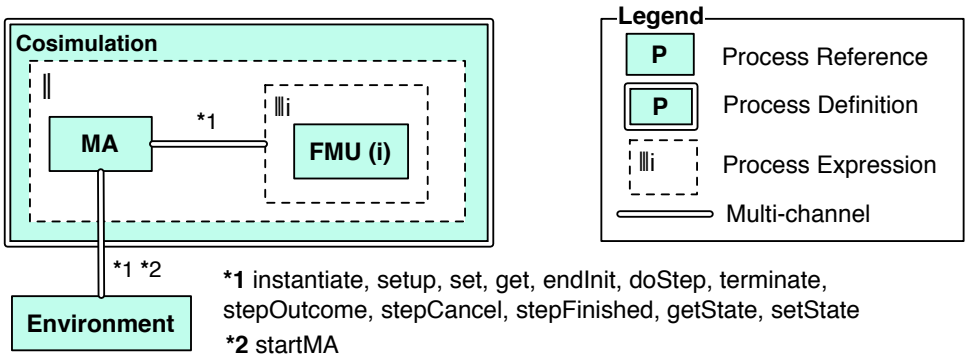


Figure 2: A pictorial description of FMI co-simulation in a CSP setting

In Figure 2, we see that the environment communicates with MAs through channel *startMA*, which signals the MA to start the simulation:

channel *startMA*

The next sections define the process that captures the behaviour of an FMU and introduce two MAs.

4.4 The *modus operandi* of an FMU

Figure 3 gives a pictorial representation of process *FMU*, which captures the overall protocol observed by an FMU in interacting with an MA and whose interface consists of the channels defined in section 4.2. Process *FMU* takes an index to identify the particular FMU being operated upon. It says that FMU simulation is interrupted by its termination, and that FMU simulation consists of the parallel composition of process *FMUSt* with process $Simulate_0$. This is defined in CSP as:

$$
\begin{array}{l}
FMU(i) = \\
\quad \text{let} \\
\qquad FMUSt = \ldots \\
\qquad Simulate = \ldots \\
\qquad Terminate = terminate.i \rightarrow SKIP \\
\quad \text{within} \\
\qquad Simulate \triangle Terminate
\end{array}
$$

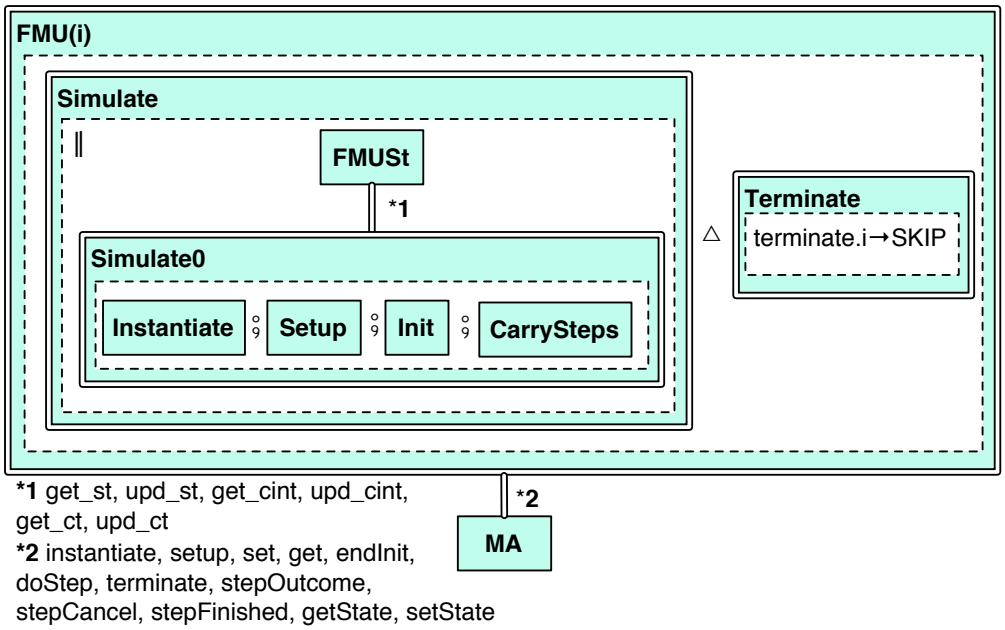


Figure 3: Pictorial description of CSP process FMU

FMUSt, *Simulate* and *Terminate* are internal processes of *FMU*. The definition says that an FMU is simulated until it is interrupted (CSP interrupt operator $\triangle$) by a termination request (event $terminate.i$).

FMU's internal process *Simulate* puts together in parallel a process that manages the state of the FMU (*FMUSt*) with a process that does the actual FMU simulation (*Simulate*0), as depicted in Fig. 3. These two processes synchronise along the channels defined for updating and retrieving the FMU state. This artificial separation of state and behaviour is necessary because CSP is a functional language, and the state of the FMU needs to be accessed by several components of FMU. INTO-CSP's imperative nature will entail the removal of such separation.

Process *FMUSt* controls a FMU state, a co-simulation interval and the simulation's current time point. Next channels read and write to these state components and are the enablers of the synchronisation portrayed in Fig. 3:

$$\textbf{channel}\ get\_st, upd\_st : St$$
$$\textbf{channel}\ get\_cint, upd\_cint : CosimInt$$
$$\textbf{channel}\ get\_ct, upd\_ct : opt(Time)$$

The set of FMU state events is thus defined as:

$$FMUStEvs = \{|upd\_st, get\_st, upd\_cint, get\_cint, upd\_ct, get\_ct|\}$$

Above, we define a set of events extensionally using $\{|\,|\}$.

Process *FMUSt* is as defined below. It controls a FMU state variable *st*, a co-simulation interval *cint* and an optional communication time point *oct*; it offers the external choice of the state-manipulation events defined above to update and retrieve state.

$$\begin{array}{l} FMUSt = \\ \quad \text{let} \\ \quad\quad FMUSt_0(st, cint, oct) = \\ \quad\quad\quad upd\_st?st' \rightarrow FMUSt_0(st', cint, oct) \\ \quad\quad\quad \Box\ get\_st!st \rightarrow FMUSt_0(st, cint, oct) \\ \quad\quad\quad \Box\ upd\_cint?cint' \rightarrow FMUSt_0(st, cint', oct) \\ \quad\quad\quad \Box\ get\_cint!cint \rightarrow FMUSt_0(st, cint, oct) \\ \quad\quad\quad \Box\ upd\_ct?oct' \rightarrow FMUSt_0(st, cint, oct') \\ \quad\quad\quad \Box\ get\_ct!oct \rightarrow FMUSt_0(st, cint, oct) \\ \quad \text{within} \\ \quad\quad FMUSt_0(emptySt, (0,0), \{\}) \end{array}$$

Process *Simulate* (Fig. 3) does the parallel composition of *FMUSt* and $Simulate_0$. The synchronisation with *FMUSt* is done through the set of events *FMUStEvs*, which is hidden so the events become internal as they pertain to the internal processing of a FMU only and are of no interest to the environment. Process $Simulate_0$ describes all the phases that a FMU has to go through, which is defined as the sequential compositions of processes *Instantiate*, *Setup*, *Init* and *CarrySteps*.

$$\begin{array}{l} Simulate = \\ \quad \text{let} \\ \quad\quad Instantiate = instantiate.i?st \rightarrow upd\_st!st \rightarrow SKIP \\ \quad\quad Setup = setup.i?cint \rightarrow upd\_cint!cint \rightarrow SKIP \\ \quad\quad Init = \\ \quad\quad\quad set.i?k?x?v \rightarrow get\_st?st \rightarrow upd\_st!updSt(st, k, x, v) \rightarrow Init \\ \quad\quad\quad \Box\ endInit.i \rightarrow SKIP \\ \quad\quad CarrySteps = \ldots \\ \quad\quad Simulate_0 = Instantiate;\ Setup;\ Init;\ CarrySteps \\ \quad \text{within} \\ \quad\quad \left(FMUSt \underset{FMUStEvs}{\|} Simulate_0 \right) \setminus (FMUStEvs) \end{array}$$

Above, process *Instantiate* is first ready to take an input on the channel *instantiate*, namely an FMU state, which is saved using channel *upd_st*. Process *Setup* waits for a communication on *setup*, providing the co-simulation interval *cint* as input, which is then stored via the channel *upd_cint*. Process *Init* keeps waiting for inputs via the channel *set* carrying a variable kind k, a variable x and a value v, which is used to update the current state of the FMU (current FMU state is obtained via channel *get_st* , which is updated through function *updSt* and then stored using channel *upd_st*), until it terminates upon receiving *endInit*.

Process *CarrySteps* below models the phases associated with a simulation step. It starts by setting up the current communication time point as the start time of the co-simulation interval (using channels *get_cint* and *upd_ct*), and proceeds with the stepping as described in $CarrySteps_0$, which gets the stored communication time point *oct* and does *CarryStep* if there is such a time point (*oct* is not empty), or terminates otherwise ($SKIP$) as there are no more time points to simulate.

$$
\begin{array}{l}
CarrySteps = \\
\quad \text{let} \\
\quad\quad CarrySteps_0 = \\
\quad\quad\quad get\_ct?oct \rightarrow \\
\quad\quad\quad\quad \left(\begin{array}{l} \text{if } oct \neq \{\} \\ \quad \text{then } get\_st?st \rightarrow \\ \quad\quad CarryStep(st, the(oct)) \\ \quad \text{else } SKIP \end{array} \right) \\
\quad \text{within} \\
\quad\quad get\_cint?cint \rightarrow upd\_ct!\{fst(cint)\} \rightarrow CarrySteps_0
\end{array}
$$

The next process, *CarryStep*, defines the preliminaries of a co-simulation step, such as setting inputs of the FMU (through channel *set*), making available the state of the FMU to enable possible step rollbacks (channel *getState*), and setting the FMU to a previous recorded state to perform a rollback (channel *setState*).

$$
\begin{array}{l}
CarryStep(st, ct) = \\
\quad set.i.input?x?v \rightarrow upd\_st!updSt(st, input, x, v) \rightarrow CarryStep(st, ct) \\
\quad \Box\ getState.i!ct!st \rightarrow CarryStep(st, ct) \\
\quad \Box\ setState.i?ct'?st' \rightarrow \\
\quad\quad \left(\begin{array}{l} ct' \leq ct\ \ \&\ upd\_st!st' \rightarrow upd\_ct!\{ct'\} \rightarrow CarryStep(st, ct') \\ \Box\ ct' > ct\ \ \&\ CarryStep(st, ct) \end{array} \right) \\
\quad \Box\ DoStep
\end{array}
$$

The next CSP process deals with actual co-simulation steps in response to the events coming along channel *doStep*. Upon a *doStep* event carrying the MA's communication time point *mct* and the step-size *stp*, it calculates the next communication time point ct'. If *mct* and *ct* are the same (MA and FMU are synchronised with respect to time), then there is a nondeterministic internal choice between the following four options: (a) the FMU carries out the co-simulation step with the given step-size *stp* (modelled by *DoStepOk*), (b) the FMU does an asynchronous execution of the co-simulation step (*DoStepPending*), (c) the FMU discards the given step-size *stp* (*DoStepDiscard*), and (d) the FMU issues an error because the step execution failed (*DoStepError*). *DoStepError* is also performed if FMU and MA are out of sync with respect to time.

$$
\begin{array}{l}
DoStep = \\
\quad doStep.i?mct?stp \rightarrow get\_ct?\{ct\} \rightarrow get\_cint?cint \rightarrow \\
\quad\quad \text{let} \\
\quad\quad\quad ct' = nextCurrentTm(ct, stp, cint) \\
\quad\quad \text{within} \\
\quad\quad\quad \text{if } mct = ct \\
\quad\quad\quad\quad \text{then} \left(\begin{array}{l} DoStepOk(ct') \\ \sqcap\ DoStepPending(ct') \\ \sqcap\ DoStepDiscard \\ \sqcap\ DoStepError \end{array} \right) \\
\quad\quad\quad\quad \text{else } DoStepError
\end{array}
$$

Above, the different responses to *doStep* are nondeterministic because the FMI sees FMUs as black-boxes — we do not know what the actual response will be. If the actual behaviour of some specific FMU is known then the nondeterminism may be removed by defining the FMU as a refinement of the general FMU process.

DoStepOk signals that the step has been carried out (event $doStepOutcome.i!stepOk$) and concludes the step.

$$
\begin{array}{l}
DoStepOk(ct') = \\
\quad doStepOutcome.i!stepOk \rightarrow get\_st?st \rightarrow FinishStep(ct', st)
\end{array}
$$

Process *FinishStep* produces non-deterministically new environments $outs'$ and sts' to represent the new values of output and state variables, respectively, that come as a result of executing the step[9] and updates the FMU

[9] As a FMU is a blackbox in this model, we do not know what are the new values of outputs and exposed state, hence the nondeterminism.

state accordingly (through channel upd_st). Following this, it does process $UpdateCT$ to check whether the new communication time point ct' is within the co-simulation bounds — depending on the result, the FMU's current communication time point is updated or voided (set to empty set). Finally, the state of the FMU may be queried (process $EnquiryFMU$), and another simulation step may be carried out, as described in process $Finalise$.

$$
\begin{aligned}
&EnquiryFMU = \\
&\quad get\_st?st \rightarrow get.i?k?x!getValInFMUSt(st, k, x) \rightarrow SKIP \\
&Finalise = \\
&\quad (EnquiryFMU;\ Finalise) \\
&\quad \Box\ CarrySteps_0 \\
&UpdateCT(ct', cint) = \\
&\quad \text{if } withinCosimInt(ct', cint) \\
&\qquad \text{then } upd\_ct!\{ct'\} \rightarrow SKIP \\
&\qquad \text{else } upd\_ct!\{\} \rightarrow SKIP \\
&FinishStep(ct', ins.outs.sts) = \\
&\quad \sqcap outs', sts' : Env \bullet \\
&\qquad upd\_st!(ins.outs'.sts') \rightarrow get\_cint?cint \\
&\qquad\quad \rightarrow UpdateCT(ct', cint);\ Finalise
\end{aligned}
$$

Above, in $FinishStep$, we use the nondeterministic choice operator $\sqcap$ to say that the values of variables $outs'$ and sts' are assigned non-deterministically. The expression $\sqcap outs', sts' : Env \bullet \ldots$ introduces those variables saying that they can have any value of their sets.

Process $DoStepDiscard$ informs the environment that the FMU rejects the given step-size by transmitting the event $doStepOutcome.i.discard$. From then onwards, the FMU offers two options: it either goes into a state where the FMU's state is queried and the simulation is finished (process $EndSim$), or has its state reset to a previous state to rollback the simulation and try

to perform it with the current time point and a new step-size.

$$
\begin{array}{l}
EndSim = \\
\quad get\_cint?cint \rightarrow (UpdateCT(snd(cint)+1, cint);\ Finalise) \\
DoStepDiscard = \\
\quad \text{let} \\
\quad\quad DoStepDiscard_0 = \\
\quad\quad\quad (EnquiryFMU;\ EndSim) \\
\quad\quad\quad \Box\ setState.i?ct'?st' \rightarrow get\_ct?\{ct\} \rightarrow \\
\quad\quad\quad\quad \left(\begin{array}{l} ct' <= ct\ \ \&\ upd\_ct!\{ct'\} \rightarrow upd\_st!st' \rightarrow SKIP \\ \Box\ ct' > ct\ \ \&\ DoStepDiscard_0 \end{array} \right) \\
\quad \text{within} \\
\quad\quad doStepOutcome.i!discard \rightarrow DoStepDiscard_0
\end{array}
$$

The next process tells the environment that the FMU is still carrying out the co-simulation step by broadcasting the event $doStepOutcome.i.pending$. From then onwards, the FMU may inform the environment that the step is finished with event *stepFinished*, followed by the conclusion of the step (*FinishStep* above) or the step may be cancelled by the environment through channel *stepCancel*, after which the simulation concludes (*EndSim* above).

$$
\begin{array}{l}
DoStepPending(ct') = \\
\quad \text{let} \\
\quad\quad DoStepPending_0 = \\
\quad\quad\quad stepFinished.i \rightarrow get\_st?st \rightarrow FinishStep(ct', st) \\
\quad\quad\quad \Box\ stepCancel.i \rightarrow EndSim \\
\quad \text{within} \\
\quad\quad doStepOutcome.i!pending \rightarrow DoStepPending_0
\end{array}
$$

The next process informs the environment that the step execution failed and ends the simulation.

$$DoStepError = doStepOutcome.i!fail \rightarrow EndSim$$

4.5 Master Algorithms

This section describes classes of master algorithms, which have a uniform structure as depicted in Fig. 4. A MA process combines subprocesses *Execute* and *Terminate* with the CSP interrupt operator ($\triangle$) to say that that MA termination interrupts MA execution. MA execution (process *Execute*) sets

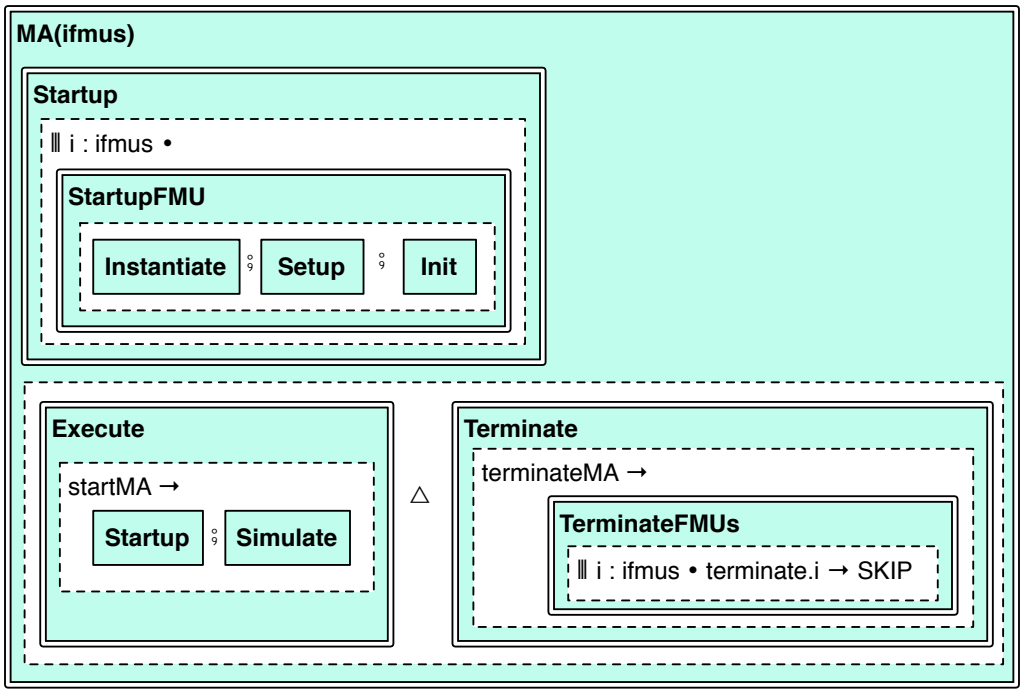


Figure 4: The underlying CSP structure of a MA

up the collection of FMUs (process *Setup*) and then simulates them (process *Simulate*).

The following start by introducing some base definitions to support the description of MAs. The, we describe in CSP two classes of MAs: simple fixed-step, and variable-step with rollback.

4.5.1 Base definitions

Time Delays. The processing of asynchronous co-simulation steps involves time delays. A MA waits for the FMU to finish some step up to a certain duration of time. To support this, we need to provide a model, which is based on the event *tock* that represents a unit of passing time; delays are based on a certain number of *tock* events that are issued. We also introduce an event *timeout* to represent the fact a certain time delay has elapsed. This is formalised in CSP by the following channels:

channel *tock*, *timeout*

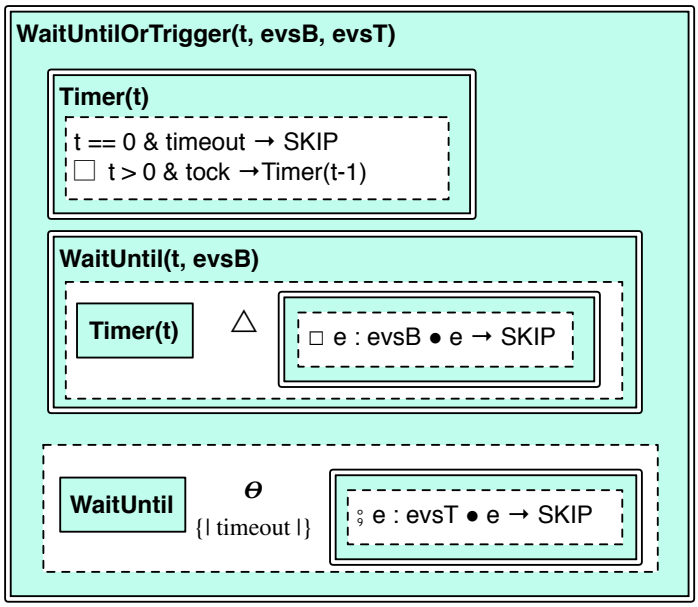


Figure 5: Process *WaitUntilOrTrigger*

Figure 5 presents process *WaitUntilOrTrigger* with its constituent processes, which underpins the processing of asynchronous FMU co-simulation steps. *WaitUntilOrTrigger* waits for a certain time delay for some event to happen, if it doesn't happen and the time delay elapses, then some event is triggered.

The next process, used by *WaitUntilOrTrigger*, defines a *Timer*. If the current time is greater than 0 (there is still some time left) and a tock is issued (an instant of time has passed), then we decrease the timer by one unit. We consider that a time duration has elapsed, when the current time reaches 0, at which point *Timer* issues the *timeout* event.

$$Timer(t) = \begin{pmatrix} t = 0 \ \& \ timeout \rightarrow SKIP \\ \Box \ t > 0 \ \& \ tock \rightarrow Timer(t-1) \end{pmatrix}$$

Process *WaitUntil*, also used by *WaitUntilOrTrigger*, waits a certain amount of time until some event, of a given set, happens:

$$\begin{array}{l} WaitUntil(t, evs) = \\ \quad \text{if } t == 0 \\ \quad\quad \text{then } Timer(t) \\ \quad\quad \text{else } Timer(t) \triangle (\Box\ e : evs \bullet e \rightarrow SKIP) \end{array}$$

Above, we say that the occurrence of one of the events interrupts the timer.

Process *WaitUntilOrTrigger* (depicted in Fig. 5) executes *WaitUntil* with the given set of break events ($evsB$), but if the *timeout* event is received then it triggers the events of the set of timeout trigger events ($evsT$).

$$
\begin{array}{l}
WaitUntilOrTrigger(t, evsB, evsT) = \\
\quad \textbf{let} \\
\quad\quad WaitUntilOrTrigger_0 = \\
\quad\quad\quad WaitUntil(t, evsB) \underset{\{|timeout|\}}{\Theta} (;\ e : evsT \bullet e \rightarrow SKIP) \\
\quad \textbf{within} \\
\quad\quad WaitUntilOrTrigger_0 \setminus \{|timeout|\}
\end{array}
$$

Above, event *timeout* is hidden because it is part of *WaitUntilOrTrigger* only, remaining, this way, invisible to the environment.

Asynchronous steps. Next process describes how a MA responds to the asynchronous execution of a FMU step. It takes an FMU index i and the number of tocks *to* wait for the FMU response. Then it does process *WaitUntilOrTrigger*, which means that it waits until event *stepFinished* occurs; if it does not occur within the given time bound, and *timeout* occurs instead, then the FMU is informed that the step is to be cancelled by transmitting event *stepCancel*.

$$
\begin{array}{l}
StepMAPending(i, to) = \\
\quad WaitUntilOrTrigger(to, \{|stepFinished.i|\}, \{|stepCancel.i|\})
\end{array}
$$

Setting the variables of a FMU. Next process may set the variables of some FMU. It takes a FMU index and a set of variables kinds; the process decides non-deterministically to set variables using channel *set* or do nothing.

$$
\begin{array}{l}
SetVarsFMU(i, VK) = \\
\quad \sqcap k : VK, x : Var, v : Val \bullet set.i!k!x!v \rightarrow SetVarsFMU(i, VK) \\
\quad \sqcap SKIP
\end{array}
$$

Process `MASetStepInputs` sets the input variables of a FMU.

$$MASetStepInputs(i) = SetVarsFMU(i, input)$$

Starting up FMUs. The next process starts up a simulation for a collection of FMUs:

$$
\begin{array}{l}
StartupFMUs(ifmus, cint) = \\
\quad \textbf{let} \\
\quad\quad StartupFMU(i) = \\
\quad\quad\quad \textbf{let} \\
\quad\quad\quad\quad Instantiate = \sqcap\, st : St \bullet instantiate.i!st \rightarrow SKIP \\
\quad\quad\quad\quad Init = SetVarsFMU(i, VarKind);\ endInit.i \rightarrow SKIP \\
\quad\quad\quad\quad Setup = setup.i!cint \rightarrow SKIP \\
\quad\quad\quad \textbf{within} \\
\quad\quad\quad\quad Instantiate;\ Setup;\ Init \\
\quad \textbf{within} \\
\quad\quad ||| \ i : ifmus \bullet StartupFMU(i)
\end{array}
$$

Above, we say that that overall startup is the interleaving (CSP operator $|||$) of the individual FMU startups.

Getting the state of a FMU. The next process retrieves outputs and exposed state of a specific FMU. It decides internally on whether to query the state of a particular FMU variable or not.

$$
\begin{array}{l}
MAGetStepOutputs(i) = \\
\quad \textbf{let} \\
\quad\quad Get(i) = \\
\quad\quad\quad \sqcap\, k : output, state, x : Var \bullet get.i!k!x?val \rightarrow Get(i) \\
\quad\quad\quad \sqcap\, SKIP \\
\quad \textbf{within} \\
\quad\quad Get(i)
\end{array}
$$

MA Termination. We introduce a channel to trigger the termination of a MA:

$$\textbf{channel}\ terminateMA$$

The following process terminates the MA and the associated collection of FMUs.

$$
\begin{array}{l}
MATerminate(ifmus) = \\
\quad \textbf{let} \\
\quad\quad TerminateFMUs = ||| \ i : ifmus \bullet terminate.i \rightarrow SKIP \\
\quad \textbf{within} \\
\quad\quad terminateMA \rightarrow TerminateFMUs
\end{array}
$$

Above, MA termination starts upon event *terminateMA*, which is followed by the termination of the individual FMUs.

FMU Events. Function *consFMIEvs* builds a set of FMI events corresponding to an indexed collection of FMUs *ifmus*, given a function that gives the individual events of some FMU (*fmuEvs*). This is defined as the distributed union of the events of the individual FMUs.

$$consFMIEvs(ifmus, fmuEvs) = \bigcup(\{fmuEvs(i) \mid i \in ifmus\})$$

Using *consFMIEvs*, we define function *CosimEvs*, which yields the set of FMI co-simulation events of an indexed collection of FMUs, and is defined using the function *FMUEvs*, giving the set of co-simulation events of a particular FMU.

$$FMUEvs(i) = \{|instantiate.i, setup.i, set.i, endInit.i, doStep.i, get.i, terminate.i, doStepOutcome.i, stepCancel.i, stepFinished.i, getState.i, setState.i|\}$$

$$CosimEvs(ifmus) = consFMIEvs(ifmus, FMUEvs)$$

4.5.2 A Generic MA

For the purpose of the FDR3 analysis, we define a process describing a generic MA that expresses the events that are expected in a FMI-based co-simulation. This gives a good basis for the validation and analysis of co-simulations based on different MAs: we should observe the expected events and no others.

Process *GenMA* captures a generic FMI co-simulation drive by a generic MA,

following the structure sketched in Fig. 4.

$$
\begin{array}{l}
GenMA(ifmus, cint, to, termEvs, discEvs) = \\
\quad \textbf{let} \\
\quad\quad TriggerTerminate = terminateMA \rightarrow SKIP \\
\quad\quad Terminate = MATerminate(ifmus) \\
\quad\quad Startup = StartupFMUs(ifmus, cint) \\
\quad\quad Execute = \\
\quad\quad\quad \textbf{let} \\
\quad\quad\quad\quad CarrySteps(ct) = \ldots \\
\quad\quad\quad\quad itermEvs = consFMIEvs(ifmus, termEvs) \\
\quad\quad\quad\quad Simulate = \\
\quad\quad\quad\quad\quad CarrySteps(fst(cint)) \underset{itermEvs}{\Theta} TriggerTerminate \\
\quad\quad\quad \textbf{within} \\
\quad\quad\quad\quad startMA \rightarrow Startup;\ Simulate \\
\quad \textbf{within} \\
\quad\quad Execute \triangle Terminate
\end{array}
$$

Above, we say that MA execution and its underlying simulation (process *Execute*) is interrupted by the termination of the MA. MA Execution starts upon event *startMA*, followed by the setup of the FMUs and their subsequent simulation. Process *Simulate* says that if one of the triggering termination events occur (set *termEvs*), then the termination of the simulation is triggered. Process *Simulate* carries the required simulation steps on the FMUs starting with the start time of the co-simulation interval.

The next process selects a step-size non-deterministically and proceeds with the simulation steps.

$$
\begin{array}{l}
CarrySteps(ct) = \\
\quad \textbf{let} \\
\quad\quad DoSteps(ct, stp) = \ldots \\
\quad \textbf{within} \\
\quad\quad \sqcap\ stp : Step \bullet DoSteps(ct, stp)
\end{array}
$$

Process *DoSteps* checks that the given communication time point *ct* is within the bounds of the co-simulation interval *cint*. If it is, it gets and saves the FMU states to enable rollback, and it carries the simulation step, which may be rollbacked, and proceeds with the next simulation action. If *ct* is not

valid, then termination is triggered.

$$
\begin{array}{l}
DoSteps(ct, stp) = \\
\quad \text{let} \\
\quad\quad GetStates = \ldots \\
\quad\quad SetStates = \ldots \\
\quad\quad CarryStep = \ldots \\
\quad\quad CarryAnother = \ldots \\
\quad\quad sDiscEvs = consFMIEvs(ifmus, discEvs) \\
\quad\quad CarryOrRollbackStep = CarryStep \Theta_{sDiscEvs} SetStates \\
\quad\quad NextSimAction = StepFMUs(ct) \sqcap CarryAnother \\
\quad \text{within} \\
\quad\quad \text{if } withinCosimInt(ct, cint) \\
\quad\quad\quad \text{then } GetStates;\ CarryOrRollbackStep;\ NextSimAction \\
\quad\quad\quad \text{else } TriggerTerminate
\end{array}
$$

Above, *CarryOrRollbackStep* carries the simulation step, but if a discard event happens it sets the FMU states to a previously recorded state to enable rollback. *NextSimAction* decides, non-deterministically, either to attempt the simulation of current ct with a new step-size (*StepFMUs*) to do the rollback or to proceed with another simulation step with the given step-size (*CarryAnother*).

To retrieve the FMU states in preparation for a possible rollback, we define the following process:

$$
\begin{array}{l}
GetStates = \\
\quad \text{let} \\
\quad\quad GetState(i) = \sqcap\ st : St \bullet getState.i.ct.st \rightarrow SKIP \\
\quad \text{within} \\
\quad\quad ||| \ i : ifmus \bullet GetState(i) \sqcap SKIP
\end{array}
$$

To set the FMUs to some state to effectuate rollback, we define the following process:

$$
\begin{array}{l}
SetStates = \\
\quad \text{let} \\
\quad\quad SetState(i) = \sqcap\ st : St \bullet setState.i.ct.st \rightarrow SKIP \\
\quad \text{within} \\
\quad\quad ||| \ i : ifmus \bullet SetState(i) \sqcap SKIP
\end{array}
$$

Process *CarryStep* does a complete simulation step on a collection of FMUs. This involves setting inputs of FMUs (*MASetStepInputs*), doing a simulation

step (*DoStep*), handling pending steps (*HandlePending*) and getting FMU outputs (*MAGetStepOutputs*).

$$
\begin{array}{l}
CarryStep = \\
\quad \text{let} \\
\quad\quad HandlePending(i) = \\
\quad\quad\quad \text{let} \\
\quad\quad\quad\quad Tocks = tock \rightarrow Tocks \sqcap SKIP \\
\quad\quad\quad\quad FinishPendingStep = \\
\quad\quad\quad\quad\quad stepCancel.i \rightarrow SKIP \sqcap stepFinished.i \rightarrow SKIP \\
\quad\quad\quad \text{within} \\
\quad\quad\quad\quad Tocks;\ FinishPendingStep \\
\quad\quad DoStep(i) = doStep.i.ct.stp \rightarrow \\
\quad\quad\quad \left(\begin{array}{l} doStepOutcome.i.stepOk \rightarrow SKIP \\ \sqcap\ doStepOutcome.i.pending \rightarrow HandlePending(i) \\ \sqcap\ doStepOutcome.i.discard \rightarrow SKIP \\ \sqcap\ doStepOutcome.i.fail \rightarrow SKIP \end{array} \right) \\
\quad\quad FullStep(i) = \\
\quad\quad\quad MASetStepInputs(i);\ DoStep(i);\ MAGetStepOutputs(i) \\
\quad \text{within} \\
\quad\quad ||| \ i : ifmus \bullet FullStep(i)
\end{array}
$$

Process *CarryAnother* gets a next communication time point ct' from the current one ct, if there is such a ct', it carries the required simulation steps for ct' with the current step-size stp otherwise the termination of the simulation is triggered:

$$
\begin{array}{l}
CarryAnother = \\
\quad \text{let} \\
\quad\quad ct' = nextCurrentTm(ct, stp, cint) \\
\quad \text{within} \\
\quad\quad \text{if } ct \neq ct' \text{ then } CarrySteps(ct', stp) \text{ else } TriggerTerminate
\end{array}
$$

4.5.3 A Simple Fixed Step Master Algorithm

We start by defining the set of events that cause the algorithm to terminate on some FMU i. This includes the step outcomes *fail* and *discard* (the algorithm is fixed step-size so it terminates when a FMU rejects the given step-size),

and the cancelling of FMU simulation steps (event *stepCancel*).

$$fsmaTermEvs(i) = \{| doStepOutcome.i.fail, doStepOutcome.i.discard, stepCancel.i |\}$$

Process fixed step master algorithm (*FSMA*), follows the MA structure outlined in Fig. 4, takes an indexed collection of FMUs to simulate *ifmus*, a co-simulation interval *cint*, the simulation's fixed step *stp* and the pending timeout duration *to*.

$$\begin{array}{l} FSMA(ifmus, cint, stp, to) = \\ \quad \text{let} \\ \quad\quad TriggerTerminate = terminateMA \rightarrow SKIP \\ \quad\quad Terminate = MATerminate(ifmus) \\ \quad\quad Startup = StartupFMUs(ifmus, cint) \\ \quad\quad termEvs = consFMIEvs(ifmus, fsmaTermEvs) \\ \quad\quad Execute = startMA \rightarrow Startup;\ Simulate \\ \quad \text{within} \\ \quad\quad Execute \triangle Terminate \end{array}$$

Above, MA execution is interrupted when the MA terminates. Process *Execute* starts upon event *startMA*, which triggers the configuration of the FMUs (*Setup*) and does the simulation (*Simulate*).

Simulate defines the required simulation stepping through the indexed collection of FMUs, starting from the first communication time point of the co-simulation interval *cint*. MA Termination is triggered upon receiving of one of the termination events.

$$\begin{array}{l} termEvs = consFMIEvs(ifmus, fsmaTermEvs) \\ Simulate = CarrrySteps(fst(cint)) \underset{termEvs}{\Theta} TriggerTerminate \end{array}$$

Stepping through the FMUs (*CarrySteps*) starts by checking that the current communication time point *ct* is within the bounds of the co-simulation interval. If it is, it carries the step and does the following step with an updated communication time point (function *nextCurrentTm*). If *ct* is not valid then there is no more stepping to do and *CarrySteps* triggers MA ter-

mination.

$$
\begin{array}{l}
CarrySteps(ct) = \\
\quad \text{let} \\
\quad\quad CarryStep = \ldots \\
\quad\quad ct' = nextCurrentTm(ct, stp, cint) \\
\quad \text{within} \\
\quad\quad \text{if } withinCosimInt(ct, cint) \\
\quad\quad\quad \text{then } CarryStep;\ CarrySteps(ct') \\
\quad\quad\quad \text{else } TriggerTerminate
\end{array}
$$

Process *CarryStep* does a complete simulation step on the FMUs. For each FMU, it sets the inputs that are required for the step (*MASetInputs*), followed by the simulation of an actual step (*Step*), followed by querying of the FMU outputs. FMU step simulation is triggered by transmitting event *doStep*, which may be sucessfull (*stepOk*), *pending*, just *fail*, or result in the rejection of the given step-size *discard*.

$$
\begin{array}{l}
CarryStep = \\
\quad \text{let} \\
\quad\quad HandleStepPending(i, to) = StepMAPending(i, to) \\
\quad\quad Step(i) = doStep.i!ct!stp \rightarrow \\
\quad\quad\quad \left(\begin{array}{l} doStepOutcome.i.stepOk \rightarrow SKIP \\ \Box\ doStepOutcome.i.pending \rightarrow HandleStepPending(i, to) \\ \Box\ doStepOutcome.i.fail \rightarrow SKIP \\ \Box\ doStepOutcome.i.discard \rightarrow SKIP \end{array} \right) \\
\quad\quad CarryStep_0(i) = MASetStepInputs(i);\ Step(i);\ MAGetStepOutputs(i) \\
\quad \text{within} \\
\quad\quad ||| \ i : ifmus \bullet CarryStep_0(i)
\end{array}
$$

We can now define a co-simulation with *FSMA*, using the FMI co-simulation CSP formula proposed in section 4.3 (Fig. 2).

$$
\begin{array}{l}
CosimFSMA(ifmus, cint, stp, to) = \\
\quad FSMA(ifmus, cint, stp, to) \underset{CosimEvs(ifmus)}{\parallel} \left|\left|\left| \ i : ifmus \bullet FMU(i) \right.\right.\right.
\end{array}
$$

This is the basis for the FDR3 analysis.

4.5.4 A Variable-Step Rollback Algorithm

We describe in CSP the rollback MA with variable step that is proposed in [BBG+13].

The sets of termination and discard events are as follows:

$$rbTermEvs(i) = \{|\, doStepOutcome.i.fail, stepCancel.i, doStepOutcome.i.pending \,|\}$$

$$discardEvs(i) = \{|\, doStepOutcome.i.discard \,|\}$$

Above, event $doStepOutcome.i.pending$ is a termination event because this MA does not support the asynchronous execution of FMU steps.

Function *decStep* decreases a step by some step increment and returns an optional: a singleton if subtraction yields a valid step, and the empty set otherwise.

$$decStep(stp, inc) = \begin{cases} \{stp - inc\} & \text{if } (stp - inc) \geq 1 \\ \{\} & \text{otherwise} \end{cases}$$

A FMI rollback algorithm needs to save the state of the FMUs under its control before executing a step. To support this, we introduce some channels to store a state of some FMU, to read the state of some FMU and to reset the state store:

channel $storeSt, readSt : FMUIdx.Time.St$
channel $reset$

We introduce set *FMUStStoreEvs* to capture the events that deal with the FMU state store.

$$FMUStStoreEvs = \{|\, storeSt, readSt, reset \,|\}$$

We introduce channels *rollback* and *finishedDoStep*, which act as internal events to the processing of simulation steps. Event *rollback* indicates that a rollback is required; *finishedDoStep* indicates that all FMUs have executed their simulation steps for some communication time point and step.

channel $rollback, finishedDoStep$

Process rollback MA (*RBMA*) takes an indexed collection of FMUs to simulate *ifmus*, a co-simulation interval *cint*, the simulation's allowed max step *maxStp* and a step increment *stpInc*; it follows the general structure of MAs

depicted in Fig. 4.

$$
\begin{array}{l}
RBMA(ifmus, cint, maxStp, stpInc) = \\
\quad \text{let} \\
\quad\quad TriggerMATerminate = terminateMA \rightarrow SKIP \\
\quad\quad Terminate = MATerminate(ifmus) \\
\quad\quad Startup = StartupFMUs(ifmus, cint) \\
\quad\quad Execute = startMA \rightarrow Startup;\ Simulate \\
\quad \text{within} \\
\quad\quad Execute \triangle Terminate
\end{array}
$$

Above, following the MA structure of Fig. 4, we say that the MA execution is interrupted by MA termination. MA execution starts upon event *startMA*, which is followed by the setup of the FMUs and their simulation.

Process *Simulate* manages the FMU state store in parallel with process $Simulate_0$ synchronised on event set *FMUStStoreEvs*, which is hidden as those events are internal to *Simulate*. Process $Simulate_0$ starts the FMU stepping (*CarrySteps*), which starts with the start time of the co-simulation interval ($fst(cint)$) and the maximum step size *maxStp*; if one of the termination events occurs while doing $Simulate_0$ then MA termination is triggered.

$$
\begin{array}{l}
Simulate = \\
\quad \text{let} \\
\quad\quad termEvs = consFMIEvs(ifmus, rbTermEvs) \\
\quad\quad Simulate_0 = \\
\quad\quad\quad CarrySteps(fst(cint), maxStp) \underset{termEvs}{\Theta} TriggerMATerminate \\
\quad \text{within} \\
\quad\quad \left(FMUSts \underset{FMUStStoreEvs}{\|} Simulate_0 \right) \setminus FMUStStoreEvs
\end{array}
$$

Process *FMUSts* controls the FMU states that the MA needs to store to

enable rollback.

$$
\begin{array}{l}
FMUSts = \\
\quad \text{let} \\
\quad\quad NotFMUSt(i, ct, st) = \\
\quad\quad\quad storeSt.i.ct.st \rightarrow IsFMUSt(i, ct, st) \\
\quad\quad\quad \Box\ reset \rightarrow NotFMUSt(i, ct, st) \\
\quad\quad IsFMUSt(i, ct, st) = \\
\quad\quad\quad readSt.i!ct!st \rightarrow IsFMUSt(i, ct, st) \\
\quad\quad\quad \Box\ reset \rightarrow NotFMUSt(i, ct, st) \\
\quad \text{within} \\
\quad\quad \underset{\{|reset|\}}{\|} \quad i : ifmus, ct : Time, st : St \bullet NotFMUSt(i, ct, st)
\end{array}
$$

Process *CarrySteps* takes a communication time point *ct* and steps-size *stp*. It checks if given *ct* is within the co-simulation interval *cint*, if it is it saves the states of the FMUs and carries the step on the FMUs for the step-size *stp* or triggers the termination of the simulation otherwise.

$$
\begin{array}{l}
CarrySteps(ct, stp) = \\
\quad \text{let} \\
\quad\quad SaveFMUSts = \\
\quad\quad\quad reset \rightarrow (\|\|\| i : ifmus \bullet getState.i?ct?st \rightarrow storeSt.i!ct!st) \\
\quad\quad CarryStep(stp) = \ldots \\
\quad \text{within} \\
\quad\quad \text{if } withinCosimInt(ct, cint) \\
\quad\quad\quad \text{then } SaveFMUSts;\ CarryStep(stp) \\
\quad\quad\quad \text{else } TriggerMATerminate
\end{array}
$$

The process that carries steps on the FMUs performs the process that does a simulation step (*DoStep*) with a process responsible for processing the next step (*DoNextStep*), synchronising on event set *evsDoStep*, which is hidden as those events are internal. Process *DoStep* does *DoStepOk*, but if a event *doStepOutcome.discard* happens then the process that carries the work associated with step-size rejection is carried out *DoStepDiscard*), which is ex-

pressed using CSP's throw operator (Θ).

$$
\begin{array}{l}
CarryStep(stp) = \\
\quad \textbf{let} \\
\quad\quad DoStepOk = \ldots \\
\quad\quad DoStepDiscard = \ldots \\
\quad\quad DoNextStep = \ldots \\
\quad\quad DoStep = \\
\quad\quad\quad \textbf{let} \\
\quad\quad\quad\quad discEvsFMUs = consFMIEvs(ifmus, discardEvs) \\
\quad\quad\quad \textbf{within} \\
\quad\quad\quad\quad DoStepOk \underset{discEvsFMUs}{\Theta} DoStepDiscard \\
\quad\quad evsDoStep = \{\!|finishedDoStep, rollback|\!\} \\
\quad \textbf{within} \\
\quad\quad \left(DoStep \underset{evsDoStep}{\|} DoNextStep \right) \setminus evsDoStep
\end{array}
$$

Process *DoStepOk* carries does the required simulation stepping on the FMUs and then it issues the event *finishedDoStep* to communication that the stepping has been carried out.

$$
\begin{array}{l}
DoStepOk = \\
\quad \textbf{let} \\
\quad\quad DoStepFMUs = \Big|\Big|\Big| i : ifmus \bullet DoStepFMU(i) \\
\quad\quad DoStepFMU(i) = MASetStepInputs(i);\ Step(i) \\
\quad\quad Step(i) = \\
\quad\quad\quad \textbf{let} \\
\quad\quad\quad\quad Step_0 = \\
\quad\quad\quad\quad\quad doStepOutcome.i.stepOk \rightarrow MAGetStepOutputs(i) \\
\quad\quad\quad\quad\quad \Box\ doStepOutcome.i.pending \rightarrow SKIP \\
\quad\quad\quad\quad\quad \Box\ doStepOutcome.i.fail \rightarrow SKIP \\
\quad\quad\quad\quad\quad \Box\ doStepOutcome.i.discard \rightarrow SKIP \\
\quad\quad\quad \textbf{within} \\
\quad\quad\quad\quad doStep.i!ct!stp \rightarrow Step_0 \\
\quad \textbf{within} \\
\quad\quad DoStepFMUs;\ finishedDoStep \rightarrow SKIP
\end{array}
$$

Process *DoStepDiscard* performs the actions associated with a step-size rejection from one of the FMUs. This amounts to either restoring the states of

the FMUs to a previous state or triggering the termination of the simulation (selected non-deterministically).

$$
\begin{array}{l}
DoStepDiscard = \\
\quad \text{let} \\
\quad\quad SetFMUStates = \\
\quad\quad\quad \text{let} \\
\quad\quad\quad\quad SetFMUState(i) = \\
\quad\quad\quad\quad\quad readSt.i?ct'?st' \rightarrow setState.i!ct'!st' \rightarrow SKIP \\
\quad\quad\quad \text{within} \\
\quad\quad\quad\quad |||\ i : ifmus \bullet SetFMUState(i) \\
\quad \text{within} \\
\quad\quad (SetFMUStates \sqcap TriggerMATerminate);\ rollback \rightarrow SKIP
\end{array}
$$

Process *DoNextStep* processes the action that follows the execution of a simulation step on the indexed collection of FMUs. It considers two cases: (a) all simulation steps executed successfully (event *finishedDoStep*) and (b) one of the FMUs discarded the step and a rollback is required (event *rollback*). The former proceeds with the simulation for the next communication time point (process *CarrySteps*), and the latter repeats the step with current communication time point but with a new smaller step-size.

$$
\begin{array}{l}
DoNextStep = \\
\quad \text{let} \\
\quad\quad ost = decStep(stp, stpInc) \\
\quad \text{within} \\
\quad\quad finishedDoStep \rightarrow CarrySteps(nextCurrentTm(ct, stp, cint), stp) \\
\quad\quad \Box\ rollback \rightarrow \\
\quad\quad\quad \text{if}\ not\ empty(ost) \\
\quad\quad\quad\quad \text{then}\ CarryStep(the(ost)) \\
\quad\quad\quad\quad \text{else}\ TriggerMATerminate
\end{array}
$$

5 FDR3 Formal Analysis of FMI Semantics

This section performs a formal analysis of the FMI semantics presented in the previous section. The analysis checks whether the semantics is accurate (validation) and satisfies certain desired properties (verification); it was performed with CSP's FDR3 tool [GRABR14], which increased our confidence in the accuracy and soundness of the semantics.

In the following, we briefly describe FDR3, present the optimised CSP version that enabled an FDR3 analysis, and present the verification and validation analysis that was carried out.

5.1 FDR3 in a nutshell

FDR (Failures Divergence Refinement) is a refinement checker for the process algebra CSP [Hoa85, Sch00, Ros10b]. FDR3 [GRABR14], the most recent version of FDR, is able to check if some processes refines another according to the denotational models of CSP, namely, traces, failures and failures divergences. This provides a powerful analysis mechanism as many questions about processes can be expressed in terms of refinement.

A refinement check is a claim of the form $SPEC \sqsubseteq IMPL$. This says that $IMPL$ refines $SPEC$. In the base CSP model of traces this equates to show that $traces(IMPL) \subseteq traces(SPEC)$: every behaviour observed in $IMPL$ can also be observed in $SPEC$. The other CSP semantic models build up on the traces model by introducing the notions of failures, refusals and divergences.

FDR is able to check important properties of concurrent systems by constructing the appropriate refinement checks, such as:

- *Deadlock-Freedom.* Deadlock arises when no further progress can be made. In a concurrent world, one process can inhibit or temporarily suspend the execution of another. Deadlocks are typically illustrated with the classic computer science example of the dining philosophers [Hoa85]. In CSP, a process is deemed deadlock-free provided it does not stop.
- *Livelock-freedom.* A livelocked system is one that diverges — it can perform only internal events becoming unresponsive to its environment. In CSP, a process is deemed livelock- or divergence-free when it is not possible that it engages in an infinite amount of internal events.
- *Determinism.* A deterministic process is one that gives the same result every time some offer is made from the environment. A deterministic process does not provide any uncertainty with respect to a particular outcome, always yielding the same result. More specifically, a deterministic process can never either diverge or have the option of performing an event or refusing it.

FDR's refinement-checking modus operandi is based on classical model-checking. This means that is performs an exhaustive exploration of all possible behaviours of a model [CW+96]. Model-checking works by refutation; to prove a property P, a model-checker will try to find a counter-example that shows $\neg P$; if none is found, then P is deemed to be true. In the context of refinement-checking, a counter-example is a trace that highlights the non-existence of a refinement between the compared processes.

The analysis presented here also used ProBE (Process Behaviour Explorer), FDR's companion tool, now integrated within FDR3. ProBE applies the formal analysis technique of *simulation* or *animation*, a form of model testing that allows tests to be executed against the model. ProBE animates CSP processes allowing the user to interactively explore a process's behaviour

5.2 The FDR3 Optmised Version of the Semantics

The semantics presented in the previous section was designed to constitute an accurate CSP model with the ultimate goal of verification using a theorem prover, such as Isabelle [NK14]. This section takes into account an optimised version of the semantics presented in the previous section to enable a practical formal model-checking analysis with FDR3.

The types based on integers defined in Section 4.1 need to be bounded. Model-checking only works with finite models; a practical analysis is feasible only when the types are restricted. In doing so, we are doing a sort of ad-hoc *abstract interpretation* [CC77] to make the model-checking based analysis practical: we create a model that is a simplified approximation of the original, but that allows us to perform inferences that apply to the original model we abstracted from.

We introduce constants `TmUB` (time upper bound), `MAXFMUs` (maximum number of FMUs), `MAXVARs` (maximum number of variables), `MINSTP` (minimum step) and `MAXSTP` (maximum step):

```
TmUB = 3
MAXFMUs = 5
MAXVARs = 1
MINSTP = 1
MAXSTP = 2
```

Based on this we bound the types `Time`, `Var` and `Val`, and the sets `FMUIdx` and `Step`:

```
nametype Time = {0..TmUB}
nametype Var = {1..MAXVARs}
nametype Val = {0..1}
FMUIdx = {1..MAXFMUs}
Step = {MINSTP..MAXSTP}
```

We map the environments of the previous section into FDR3 maps, which gives us the same semantics in a FDR3 world:

```
nametype Env = Map (Var, Val)
```

To further reduce the number of states, we allow FMU states to contain inputs and outputs only:

```
nametype St = Env.Env
```

The CSP specification of the previous section defined *St* as *Env.Env.Env* to allow for exposed variables of FMUs.

5.3 Verification

We describe the effort involved in verifying the different components that make the CSP FMI semantics presented here.

5.3.1 `FMU` Process

To support the verification of deadlock-freedom in FDR3, we introduce `Iter` to perform the iteration of a given process:

$$Iter(P) = P;\ Iter(P)$$

This keeps running processes that terminate, which is important because absence of deadlock involves checking that a process does not stop. By removing the possibility of normal termination, we can have an analysis that looks for genuine and unexpected deadlocks.

Using FDR3, we model-checked that the `FMU` process is both livelock- and deadlock-free (assertions below). Absence of deadlock is formulated using `Iter`: continuously-running FMUs must not deadlock.

```
assert FMU(1) :[divergence free]
assert Iter(FMU(1)) :[deadlock free]
```

5.3.2 Fixed Step Master Algorithm

The analysis of the fixed-step MA (FSMA) involves the following variables:

```
FSTP = 1
discEvs(i) = {| |}
```

Above, we say that the `FSTP` fixed step-size has the value 1, and that there are no step discard events (function *discEvs*) as we are in the setting of fixed-step simulation.

We start by verifying the healthiness of the generic MA that is used in the checks of FSMA. The following assertions, discharged in FDR3, check absence of both deadlock and livelock:

```
assert GenMAC({1..5}, (0, 2), fsmaTermEvs, discEvs) :[livelock free]
assert Iter(GenMAC({1..5}, (0, 2), fsmaTermEvs, discEvs)) :[deadlock free]
```

The following assertions, discharged in FDR3, check that the fixed-step master algorithm (`FSMA`), in a setting with 5 FMUs and 3 time points, is both livelock- and deadlock-free, and that only the expected events can be observed. To improve the performance of the checks, we use FDR3's diamond compression with strong bisimulation (functions *sbisim* and *diamond*).

```
FSMA1 = sbisim(diamond(FSMAC({1..5}, (0, 2), FSTP, 0)))
assert FSMA1 :[livelock free]
assert Iter(FSMA1) :[deadlock free]
assert GenMAC({1..5}, (0, 2), fsmaTermEvs, discEvs) [T= FSMA1
```

Likewise, for a setting of 3 FMUs and a delay of 3 tocks for asynchronous steps:

```
assert FSMAC({1..3}, (0, 2), FSTP, 3) :[divergence free]
assert Iter(FSMAC({1..5}, (0, 2), FSTP, 3)) :[deadlock free]
assert GenMAC({1..3}, (0, 2), fsmaTermEvs, discEvs)
   [T= FSMAC({1..3}, (0, 2), FSTP, 3)
```

We now turn to a global check to verify overall cosimulations. The next assertions, discharged in FDR3, check that the cosimulation with FSMA with two FMUs and a pending-wait time of 3 tocks is both livelock and deadlock-free, and that only the expected evens can be observed:

```
assert CosimFSMA ({1, 2}, (0, 2), FSTP, 3) :[divergence free]
assert Iter(CosimFSMA ({1, 2}, (0, 2), FSTP, 3)) :[deadlock free]
assert GenMAC({1, 2}, (0, 2), fsmaTermEvs, discEvs)
  [T= CosimFSMA ({1, 2}, (0, 2), FSTP, 3)
```

Likewise for a setting of up to 5 FMUs:

```
CoSimFSMA6 = sbisim(diamond(CosimFSMAO ({1..5}, (0, 2), FSTP)))
assert CoSimFSMA6 :[livelock free]
assert Iter(CoSimFSMA6) :[deadlock free]
GenFMI2 =
    sbisim(diamond(GenMAO({1..5}, (0, 2), fsmaTermEvs, discEvs)))
assert GenFMI2 [T= CoSimFSMA6
```

Above, `CoSimFSMA6` uses FDR3's diamond compression in the setting of strong bi-simulation. The assertions were discharged with FDR3.

5.3.3 Rollback Master Algorithm

We model-checked absence of livelock and deadlock for the rollback MA (process `RBMA`) in a setting with 3 FMUs and 3 time points:

```
assert RBMA({1..3}, (0, 2), MAXSTP, 1) :[divergence free]
assert Iter(RBMA({1..3}, (0, 2), MAXSTP, 1)) :[deadlock free]
```

The generic simulation with discard events (as defined in event set *discEvs* above) is both deadlock- and livelock-free, as stated in the assertions that follow for 3 FMUs, which were discharged in FDR3:

```
assert
  GenMAC({1..3}, (0, 2), rbTermEvs, discEvs) :[livelock free]
assert
  Iter(GenMAC({1..3}, (0, 2), rbTermEvs, discEvs)) :[deadlock free]
```

The rollback MA is less efficient than the fixed-step MA because the states of the FMUs need to be saved at each step, and this is reflected in the efficiency of the FDR3 analysis. For the rollback MA, we could only perform the required verification checks (livelock and deadlock-freedom and expected events) in settings with up to 3 FMUs and 3 time-points:

```
CoSimRBMA3 = sbisim(diamond(CoSimRBMA ({1.. 3}, (0, 2), MAXSTP, 1)))
assert Iter(CoSimRBMA4) :[divergence free]
assert Iter(CoSimRBMA4) :[deadlock free]
GenSim3 = sbisim(diamond(GenMAC({1 .. 3}, (0, 2), rbTermEvs, discardEvs)))
assert GenSim3 [T= CoSimRBMA3
```

Above, we make use FDR3's diamond compression in the setting of strong bi-simulation.

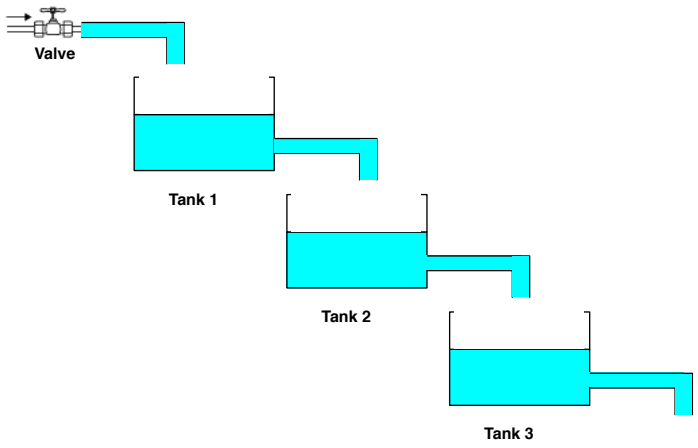


Figure 6: The three cascading Water Tanks Example

5.4 Validation

We validate the FMI semantics presented here with two examples: the three cascading water tanks, and the periodic discrete signal generator.

5.4.1 Three Water Tanks

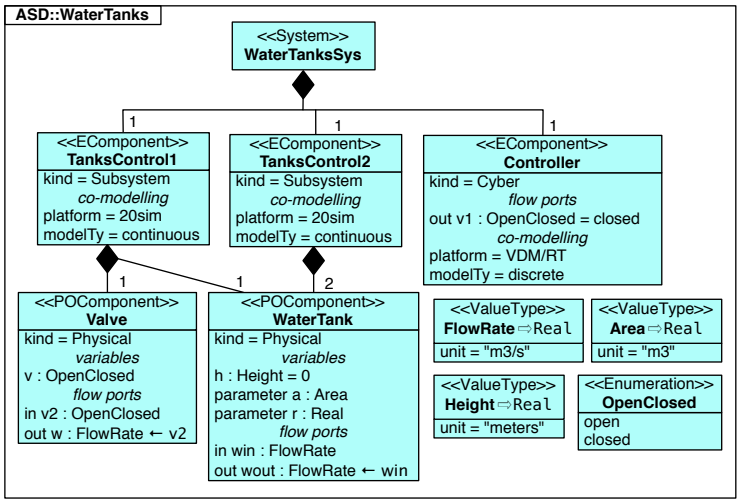
The three cascading Water Tanks System (3WTS), the running example of INTO-CPS deliverable D2.1a [APC+15] that describes the SysML/INTO-CPS profile, is given in Figure 6. The SysML/INTO-CPS architecture structure diagram (ASD) and connections diagram (CD) that describe a design of 3WTs are given in Figure 7.

The FMI interpretation of the CD (Fig. 7(b)) gives three FMUs: `TanksControl1`, `TanksControl2` and `Controller`[10]. We describe here in CSPm the casting of the SysML/INTO-CPS model of 3WTs into the FMI semantics presented here, and provide a run of 3WTs with 3 time points. We check, using FDR3, that the run is a traces refinement of a cosimulation with a FSMA.

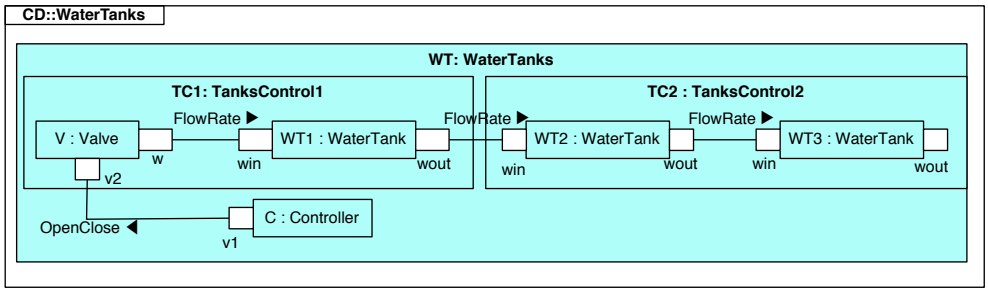
The following CSPm snippet defines the state of the three FMUs:

```
ControllerSt     = (| |).(| 1=>0 |)
TanksControl1St = (| 1=>0 |).(| 1=>0 |)
TanksControl2St =  (| 1=>0 |).(| 1=>0 |)
```

[10]It gives three FMUs because FMU nesting is not currently supported.



(a) `ASD`



(b) `CD`

Figure 7: SysML/INTO-CPS ASD and CD of the three cascading Water Tanks Example

Above, we define map extensions using the parenthesis `(| |)`. `1=>0` means that variable 1 is given value 0.

The process described in the next CSPm snippet defines a FMI co-simulation of 3WTs. The process is as follows: (i) it starts the MA, (ii) it instantiates the `Controller` FMU (FMU index 1), sets its output and ends the initialisation of this FMU, (iii) does the same for the `TanksControl1` (FMU index 2) and `TanksControl2` (FMU index 3), (iv) it does three steps on the FMUs as described in `WTsRun0` and (v) ends the cosimulation.

```
WTsSimulation =
  let
    WTsSimulation(ct) =
```

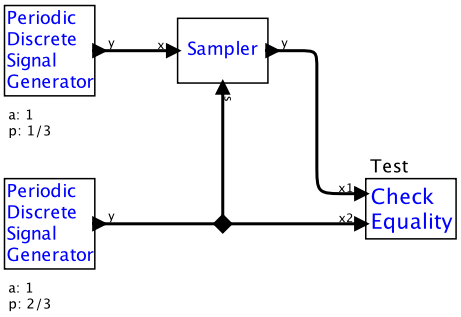


Figure 8: The periodic discrete signal generator example of [BGL+14]

```
    doStep.1.ct.1-> doStepOutcome.1.stepOk
    -> get.1.output.1?{v} -> set.2.input.1.v
    -> doStep.2.ct.1-> doStepOutcome.2.stepOk
    -> get.2.output.1?{w} -> set.3.input.1.w
    -> doStep.3.ct.1-> doStepOutcome.3.stepOk
    -> get.3.output.1?{w2}->SKIP
within
  startMA -> instantiate.1!ControllerSt -> setup.1!(0,2)
  ->set.1.output.1.1 -> endInit.1
  -> instantiate.2!TanksControl1St -> setup.2!(0,2) -> endInit.2
  -> instantiate.3!TanksControl2St -> setup.3!(0,2)
  -> endInit.3
  -> (WTsSimulation0(0); WTsSimulation0(1); WTsSimulation0(2));
     terminateMA->terminate.1
       ->terminate.2->terminate.3->SKIP
```

The next assertion, model-checked using FDR3, confirms that `WTsSimulation` is a traces refinement of a cosimulation of the fixed-step master algorithm presented in section 4.5.4.

```
CosimFSMA1 = sbisim(diamond(CosimFSMA ({1..3}, (0, 2), FSMASTP, 0)))
assert CosimFSMA1 [T= WTsSimulation
```

5.4.2 Periodic discrete signal generator

The periodic discrete signal generator example of [BGL+14] is pictured in Figure 8. It highlights a configuration with 4 FMUs.

The next CSPm snippet describes the states of these four FMUs:

```
PDSG1St          = (| |).(| 1=>0 |)
PDSG2St          = (| |).(| 1=>1 |)
```

```
SamplerSt       = (| 1 => 0, 2=>0|).(| 1=>1 |)
CheckEqualitySt = (| 1 => 0, 2=>0|).(| 1=> 0 |)
```

A CSPm process that describes a simulation of these four FMUs is as follows:

```
PDSGSimulate =
    startMA->instantiate.1!PDSG1St ->setup.1!(0,2)
    ->endInit.1
    ->instantiate.2!PDSG2St -> setup.2!(0,2)
    ->endInit.2
    ->instantiate.3!SamplerSt -> setup.3!(0,2)
    ->endInit.3
    ->instantiate.4!CheckEqualitySt -> setup.4!(0,2)
    ->endInit.4
    -- First Run gives equal
    ->doStep.1.0.1-> doStepOutcome.1.stepOk
    -> get.1.output.1?{v}
    -> doStep.2.0.1-> doStepOutcome.2.stepOk
    -> get.2.output.1?{w}
    -> set.3.input.1.v -> set.3.input.2.w
    -> doStep.3.0.1-> doStepOutcome.3.stepOk
    -> get.3.output.1?{w}
    -> set.4.input.1.w
    -> set.4.input.2.w
    -> doStep.4.0.1-> doStepOutcome.4.stepOk
    -> get.4.output.1?{1}
    -- Second Run gives not equal
    ->doStep.1.1.1-> doStepOutcome.1.stepOk
    -> get.1.output.1?{v}
    -> doStep.2.1.1-> doStepOutcome.2.stepOk
    -> get.2.output.1?{w}
    -> set.3.input.1.v -> set.3.input.2.w
    -> doStep.3.1.1-> doStepOutcome.3.stepOk
    -> get.3.output.1?{v2}->set.4.input.2.v2
    -> set.4.input.1.w
    -> doStep.4.1.1-> doStepOutcome.4.stepOk
    -> get.4.output.1?{0}
    -- Third Run gives equal
    ->doStep.1.2.1-> doStepOutcome.1.stepOk
    -> get.1.output.1?{v}
    -> doStep.2.2.1-> doStepOutcome.2.stepOk
    -> get.2.output.1?{w}
    -> set.3.input.1.v -> set.3.input.2.w
    -> doStep.3.2.1-> doStepOutcome.3.stepOk
    -> get.3.output.1?{v}
    ->set.4.input.2.w-> set.4.input.1.w
    -> doStep.4.2.1-> doStepOutcome.4.stepOk
    -> get.4.output.1?{1}
```

```
-- The termination phase
->terminate.1->terminate.2->terminate.3
->terminate.4->SKIP
```

The following assertions, which check whether the process 'PDSGSimulate' is a valid run of FSMA, were checked using FDR3

```
assert FSMAO ({1..4}, (0, 2), FSTP) [T= PDSGSimulate
assert FSMAC ({1..4}, (0, 2), FSTP, 0) [T= PDSGSimulate
```

The following global assertion (in an overall co-simulation setting) could not be checked using FDR3 due to the state explosion problem. It was checked through simulation using ProBE.

```
CosimFSMA1 = sbisim(diamond(CosimFSMAO ({1..4}, (0, 2), FSTP)))
assert CosimFSMA1 [T= PDSGSimulate
```

5.5 Experimental Results

We now present some experimental results related with model-checking in FDR3 assertions that verify and validate the CSP semantics as described above. The experiments were performed on a MacBook Pro with a 2.5 GHz Intel core i7 processor and 16GB RAM memory. The results of the experiment are summarised in table 2. Of all the assertions listed in the table, only the one related with the periodic-discrete signal generator (`PDSG`) could not be discharged with FDR3 (signalled in table 2 as ∞); as described above, this assertion was checked using ProBE; FDR's inability to discharge this assertion is due to `PDSG`'s larger state space (`PDSG` requires 4 FMUs with two variables, the extra variable constituting an unsurmountable hurdle for refinement-checking).

6 Conclusions

This deliverable presents the efforts of INTO-CPS's WP2 on the formal foundations of the FMI standard. The document gives an introduction to the FMI, emphasising FMI's co-simulation facet, surveys the FMI-related literature with a special focus on formal approaches to the FMI, develops the formal semantics of FMI expressed in the CSP process algebra, and does a formal analysis based on model-checking of semantics. The deliverable's main contribution is the CSP semantics of FMI co-simulation, which denotes

Assertion	# FMUs	# Vars	Time (sec)
`FMU` livelock	–	1	0.12
`FMU` deadlock	–	1	0.52
`CosimFSMA` livelock, 0 tocks	2	1	0.77
`CosimFSMA` deadlock, 0 tocks	2	1	0.33
`GenFMISim` $\sqsubseteq_T$ `CosimFSMA`, 0 tocks	2	1	0.27
`CosimFSMA` livelock, 3 tocks	2	1	6.2
`CosimFSMA` deadlock, 3 tocks	2	1	7.93
`GenFMISim` $\sqsubseteq_T$ `CosimFSMA`, 0 tocks	2	1	13.61
`CosimFSMA` livelock, 0 tocks	5	1	520.39
`CosimFSMA` deadlock, 0 tocks	5	1	0.09
`GenFMISim` $\sqsubseteq_T$ `CosimFSMA`, 0 tocks	5	1	397.46
`RBMA` livelock	3	1	119.63
`RBMA` deadlock	3	1	28.36
`CosimRBMA` livelock	2	1	2.32
`CosimRBMA` deadlock	2	1	0.25
`CosimRBMA` livelock	3	1	117.51
`CosimRBMA` deadlock	3	1	7.37
`3WTs`	3	1	9.76
`PDSG`	4	2	∞

Table 2: Experimental results of formally analysing FMI's CSP Semantics with FDR3 refinement-checker. Columns indicate the analysed assertions, the number of FMUs, the number of variables and the time FDR3 took to execute the assertion.

an underlying UTP semantics. As master algorithms are a a crucial component of FMI co-simulation, the document presents the formalisations of two classes of MAs: simple fixed-step MAs, and the variable-step MAs with support for rollback proposed in [BBG+13].

CSP has been chosen to express the semantics due to its appropriateness to express interaction-driven computations. FMI co-simulation revolves around the interaction between a MA with FMUs, which is precisely what is covered by the semantics presented here. This interaction is concisely expressed in the following CSP process that is portrayed in Fig. 2:

$$Cosimulation = MA \underset{CosimEvs}{\parallel} \left|\left|\left| i : ifmus \bullet FMU(i)\right.\right.\right.$$

FMI co-simulation involves an MA that synchronises with a collection of FMUs that run in interleaving, where the synchronisation is based on the FMI API for co-simulation described in CSP as communication channels.

It is interesting to compare the CSP formal semantics against [BBG+13], which is, to our knowledge, the only alternative FMI formalisation to the work presented here. A striking difference from our semantics to [BBG+13] lies in the level of fidelity with respect to the underlying FMI. The model of [BBG+13] makes many simplifications with the aim of showing the conditions under which a composition of FMUs ensures determinism; for example the mathematical model of [BBG+13] models FMU step executions as a total function that considers only two possible outcomes: success and rejection of the proposed step size. There is no consideration for the possibility that a FMU may simply fail to execute the step, or that it executes the step asynchronously and a master algorithm needs to wait for the step execution to finish, or simply that the FMU is non-deterministic (has it happens so oftenly with abstract discrete models, such as the discrete CSP model presented here, which is nondeterministic). The semantics presented here was developed to (a) gain a better understanding of the FMI and FMI-based co-simulation, (b) reason about the FMI, and (c) evaluate the impact of possible FMI extensions by studying them at the model level. These three aims require a mathematical account with high-level of fidelity. Now that we have this very general and accurate model that tells us how a FMU works, we can study the properties of classes of FMUs from our semantics similar to [BBG+13]'s deterministic study. Using the theory of CSP refinement, we can build a deterministic FMU that is proved to be a refinement of the most general FMU and that, like in [BBG+13], can only execute the step or discard the given step-size, and from then we can study deterministic compositions of FMUs as done in [BBG+13]. But the semantics presented here gives us

other possibilities also; for example, we can use it for reasoning about FMU failures and MAs that ensure resilient co-simulations, study the interaction of classes of MAs with FMUs, study MAs that tolerate asynchronous modes of FMU execution, etc.

Another difference lies in the fact that [BBG+13] does not describe the dependencies that exist between the different operations and phases that a FMU goes through in a FMI co-simulation. For example, before a FMU is allowed to execute a step it must have been set up and certain state components must have been initialised. The CSP formalisation presented here describes such dependencies. This means that unlike [BBG+13], the CSP model enables the analysis of deadlock and livelock in the interaction between a MA and FMUs and enables us to certify certain master algorithms as deadlock- and livelock-free.

The CSP formalisation presented here has ben subject to a machine-assisted analysis with the FDR3 refinement-checker, albeit in a setting of constant parameters that confine the CSP model within small bounds to avoid the state explosion problem. This was not exploited in [BBG+13]; proofs were performed without machine-assistance. In the future we would like to explore verification with the Isabelle theorem prover [NK14]. This means that we no longer have to imprison our CSP model within small bounds. Hence, the analysis becomes more general but it also requires more user intervention. We no longer have model enquiries answered by simply pushing a button; we somehow loose this straightforward feedback capability when we move into an interactive theorem proving world, where user-written proofs are required. However, it allows us to relax some restrictions of the semantics presented here and increase its accuracy, in particular, with respect to time, represented as integers in CSP model presented here; in the future, we would like to have a representation of time based on reals and its machine-representation based on floating-point numbers.

References

[ABH+10] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(6):447–466, 2010.

[Abr10] Jean-Raymond Abrial. *Modeling in Event-B: system and software engineering*. Cambridge University Press, 2010.

[APC+15] Nuno Amálio, Richard Payne, Ana Cavalcanti, Etienne Brosse, and Jim Woodcock. Foundations of the SysML profile for CPS modelling. Technical Report D2.1a, INTO-CPS project, 2015.

[BBG+13] David Broman, Christopher Brooks, Lev Greenberg, Edward A. Lee, Michael Masin, Stavros Tripakis, and Michael Wetter. Determinate composition of FMUs for co-simulation. In *EMSOFT 2013*. IEEE, 2013.

[BCWS11] Jens Bastian, Christoph Clauß, Susan Wolf, and Peter Schneider. Master for co-simulation using FMI. In *Modelica Conference*, 2011.

[BGL+14] David Broman, Lev Greenberg, Edward A. Lee, Michael Masin, Stavros Tripakis, and Michael Wetter. Requirements for hybrid cosimulation. Technical Report UCB/EECS-2014-157, EECS Department, University of California, Berkeley, 2014.

[BOA+11] Torsten Blochwitz, Martin Otter, M. Arnold, C. Bausch, C. Clauß, Hilding Elmqvist, A. Junghanns, J. Mauss, M. Monteiro, T. Neidhold, D. Neumerkel, H. Olsson, J.-V. Peetz, and S. Wolf. The functional mockup interface for tool independent exchange of simulation models. In *Modelica Conference*, 2011.

[BOA+12] T. Blochwitz, M. Otter, J. Akesson, M. Arnold, C. Clauß, H. Elmqvist, M. Friedrich, A. Junghanns, J. Mauss, D. Neumerkel, H. Olsson, and A. Viel. Functional mockup interface 2.0: The standard for tool independent exchange of simulation models. In *Modelica Conference*, 2012.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1977.

[CW+96] Edmund Clarke, Jeannette Wing, et al. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[DLV12] Patricia Derler, Edward A. Lee, and Alberto Sangiovanni Vincentelli. Modeling cyber-physical systems. *Proceedings of IEEE*, 100(1), 2012.

[DMMV15] Joachim Denil, Bart Meyers, Paul De Meulenaere, and Hans Vangheluwe. Explicit semantic adaptation of hybrid formalisms for FMI co-simulation. In *Spring Simulation Multi-Conference (SpringSim)*, 2015.

[FGP14] Yishai A. Feldman, Lev Greenberg, and Eldad Palachi. Simulating rhapsody SysML blocks in hybrid models with FMI. In *Modelica Conference*, 2014.

[FMI14] FMI development group. Functional mock-up interface for model exchange and co-simulation, 2.0. `https://www.fmi-standard.org`, 2014.

[GRABR14] Thomas Gibson-Robinson, Philip Armstrong, Alexandre Boulgakov, and A.W. Roscoe. FDR3 — A Modern Refinement Checker for CSP. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 8413 of *LNCS*, pages 187–201, 2014.

[HJ89] Ian Hayes and Cliff Jones. Specifications are not (necessarily) executable. *Software Engineering Journal*, 4(6):330–338, 1989.

[HJ98] C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice-Hall, 1998.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[KS00] R. Kübler and W. Schiehlen. Two methods of simulator coupling. *Mathematical and Computer Modelling of Dynamical Systems*, 6(2):93—113, 2000.

[MMP92] Oded Maler, Zohar Manna, and Amir Pnueli. From timed to hybrid systems. In *REX Workshop*, pages 447–484. Springer, 1992.

[NK14] Tobias Nipkow and Gerwin Klein. *Concrete Semantics: With Isabelle/HOL*. Springer, 2014.

[Par72] David Lodge Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[PSR+12] Uwe Pohlmann, Wilhelm Schäfer, Hendrik Reddehase, Jens Röckemann, and Robert Wagner. Generating functional mockup units from software specifications. In *Modelica Conference*, 2012.

[Ros10a] A. W. Roscoe. CSP is expressive enough for π. In *Reflections on the Work of C.A.R. Hoare*. Springer, 2010.

[Ros10b] A. W. Roscoe. *Understanding Concurrent Systems*. Springer, 2010.

[SA12] Tom Schierz and Martin Arnold. Stabilized overlapping modular time integration of coupled differential-algebraic equations. *Applied Numerical Mathematics*, 62:1491–1502, 2012.

[SAC12] Tom Schierz, Martin Arnold, and Christoph Clauß. Co-simulation with communication step size control in an FMI compatible master algorithm. In *Modelica Conference*, 2012.

[SBC14a] Vitaly Savicks, Michael Butler, and John Colley. Co-simulating Event-B and continuous models via FMI. In *Summer Simulation Multiconference*, 2014.

[SBC14b] Vitaly Savicks, Michael Butler, and John Colley. Co-simulation environment for rodin: Landing gear case study. In *ABZ*, LNCS. Springer, 2014.

[Sch00] Steve Schneider. *Concurrent and Real-Time Systems*. Wiley, 2000.

[TB14] Stavros Tripakis and David Broman. Bridging the semantic gap between heteregeneous modeling formalisms and FMI. Technical Report UCB/EECS-2014-30, EECS Department, University of California, Berkeley, 2014.

[TOHSMS99] Peri Tarr, Harold Ossher, William Harrison, and Jr Stanley M. Sutton. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *ICSE*, 1999.

[Weg96] Peter Wegner. Interoperability. *ACM Computing Surveys*, 28(1):285–287, 1996.

A CSPm of FMI semantics

This appendix presents the CSPm definitions corresponding to an optimised version of the CSP specification presented in section 4. The CSPm definitions were the basis of a formal analysis with the FDR3 refinement-checker.

A.1 Base definitions

```
-- CSP semantics of FMI standard (version 2.0)
--
-- Module that provides the base definitions of semantics
--
-- Nuno Amalio
-- (from an initial version of Ana Cavalcanti and Jim Woodcock)

-- Constant time upper bound
TmUB = 3

-- Maximum number of FMUs
MAXFMUs = 5

MAXVARs = 1

-- Type 'Time'
nametype Time = {0..TmUB} --Int
nametype Var = {1..MAXVARs}  -- This represents variables
nametype Val = {0..1}  -- This represents values

datatype VarKind = input | output | state

-- Defines optionals of some given type
opt(X) = {s | s<- Set(X), card(s) <= 1}

nametype Env = Map (Var, Val)

-- This version supports environments with inputs and outputs only.
-- A third environment would cover exposed state.
```

```
nametype St = Env.Env

emptySt = emptyMap.emptyMap

-- Gets the element of a singleton
the({x}) = x

-- nil constant to stand for an empty set
nil = {}

MAXSTP = 2
Step = {1..MAXSTP}

-- Set of indices corresponding to FMUs
FMUIdx = {1..MAXFMUs}

-- A co-simulation interval with a start and an end time
nametype CosimInt = {(st, et) | st <- Time, et <- Time, st <= et}

-- Functions 'fst' and 'snd'
fst ((st, et)) = st
snd ((st, et)) = et

-- Channel to instantiate FMU
channel instantiate : FMUIdx.St

-- Channel to set up experiment
channel setup : FMUIdx.CosimInt

-- Channel to set the state of a FMU
channel set : FMUIdx.VarKind.Var.Val

-- Channel to get the state of a FMU
channel get         : FMUIdx.{output, state}.Var.opt(Val)

-- Channel to inform FMU that initialisation has finished
channel endInit     : FMUIdx

-- The FMU does a step
channel doStep      : FMUIdx.Time.Step

-- The FMU terminates
channel terminate   : FMUIdx

-- Represents outcome of a 'doStep' event
datatype DoStepOutcome = stepOk | pending | discard | fail

-- What the 'doStep' event returns to the environment
channel doStepOutcome : FMUIdx.DoStepOutcome
```

```
-- When call to 'doStep' is asynchronous (outcome 'pending'),
-- MA may cancel step or FMU may say that step has finished
channel stepCancel, stepFinished : FMUIdx
channel getState, setState       : FMUIdx.Time.St

-- The FMI extension proposed by Broman et al 2013
-- The null step means infinity
channel getMaxStepSize : FMUIdx.opt(Step)

--
-- Keeps executing a process
Iter(P) = P; Iter(P)

--
-- Iterates a process through a sequence
IterPSeq (P, elems) =
  if null(elems)
    then SKIP
    else P(head(elems)); IterPSeq(P, tail(elems))

--
-- Updates variable 'x' with value 'v' in a environment
updEnv (x, v, e) =
  if mapMember(e, x) then mapUpdate(e, x, v)
  else e

--
-- Function that updates the state
updSt (ins.outs, input, x, v) =
  updEnv(x, v, ins).outs
updSt (ins.outs, output, x, v) =
  ins.updEnv(x, v, outs)
updSt (ins.outs, state, x, v) =
  ins.outs

-- Gets value of a map. If it exists returns singleton,
-- otherwise returns empty set
getVarVal(x, map) =
  if mapMember(map, x)
    then {mapLookup(map, x)}
    else {}

--
-- Function that gets the value of a variable in a FMU state
getValInFMUSt (ins.outs, input, x) = getVarVal(x, ins)
getValInFMUSt (ins.outs, output, x) = getVarVal(x, outs)
-- optimised version does not support state variables
getValInFMUSt (ins.outs, state, x) = {}
```

```
-- Predicate saying if given ct is within a cosimulation interval
withinCosimInt(ct, (st, et)) = st <= ct and ct <= et

-- Issues next communication time point for simulation
-- given current ct, a step and cosimulation interval
-- It issues next time point (ct') only when ct is valid
nextCurrentTm (ct, stp, cint) =
  if withinCosimInt(ct, cint)
    then ct+stp
    else ct

-- Process that terminates a FMU
TerminateFMU(i) = terminate.i -> SKIP
```

A.2 FMU Process

```
-- CSP semantics of FMI standard (version 2.0)
--
-- Module that defines process FMU describing FMI's API
-- for cosimulation
--
-- Nuno Amalio
-- (from an initial version of Ana Cavalcanti and Jim Woodcock)

include "fmi_base.csp"

-- Process 'FMU'
-- FMU interface defined by FMI API for cosimulation
FMU(i) =
  let
    Simulate =
      let
        -- FMU is set up with the given co-simulation interval
        -- and then proceeds with the initialisation
        Setup(st) = setup.i?cint -> InitAndCarrySteps(st, cint)
        -- The FMU intitialises and steps
        InitAndCarrySteps (st, cint) =
          let
            -- 'cint' parameter because of bug in FDR
            CarrySteps (st, ct, cint) =
              let
                DoStep (st) =
                  let
                    DoStepOk (ct') =
                      doStepOutcome.i!stepOk
                        -> FinishStep (st, ct')
                    FinishStep (ins.outs, ct') =
```

```
            let
              FinishStep0 (st) =
                get.i?k?x!getValInFMUSt (st, k, x)
                  -> FinishStep0 (st)
                  [] CarrySteps (st, ct', cint)
            within
              |~| outs' : Env @
                FinishStep0 (ins.outs')
          DoStepPending (ct') =
            let
              DoStepPending0 =
                stepFinished.i -> FinishStep (st, ct')
                [] stepCancel.i -> TerminateFMU(i)
            within
              doStepOutcome.i!pending -> DoStepPending0
          DoStepDiscard =
            let
              DoStepDiscard0 =
                TerminateFMU(i)
                [] setState.i?ct'?st' ->
                  (ct' <= ct & CarrySteps(st', ct', cint)
                   [] ct' > ct & DoStepDiscard0)
            within
              doStepOutcome.i!discard -> DoStepDiscard0
          DoStepError =
            doStepOutcome.i!fail -> TerminateFMU(i)
        within
          doStep.i?mct?stp->
            let
              ct' = nextCurrentTm (ct, stp, cint)
            within
              if mct == ct
              then DoStepOk(ct')
                |~| DoStepPending(ct')
                |~| DoStepDiscard
                |~| DoStepError
              else DoStepError
      CarrySteps0 (st) =
        set.i.input?x?v
          -> CarrySteps0 (updSt(st, input, x, v))
        [] DoStep (st)
        [] getState.i!ct!st -> CarrySteps0 (st)
        [] setState.i?ct'?st' ->
          (ct' <= ct & CarrySteps (st', ct', cint)
           [] ct' > ct & CarrySteps0 (st))
within
  if withinCosimInt(ct, cint)
    then CarrySteps0(st)
    else SKIP
```

```
            within
              set.i?k?x?v ->
                InitAndCarrySteps(updSt(st, k, x, v), cint)
              [] endInit.i -> CarrySteps(st, fst(cint), cint)
        within
          -- A FMU is instantiated when it is given a state
          instantiate.i?st -> Setup (st)
      within
        -- The FMU simulates and is interrupted when terminated
        Simulate /\ TerminateFMU(i)

--Checks that FMU does not engage in infinite internal work
assert FMU(1) :[divergence free]

-- Checks that continuously running a FMU does not deadlock
assert Iter(FMU(1)) :[deadlock free]
```

A.3 Common Definitions to support Master Algorithms

```
-- The CSP semantics of the FMI standard (version 2.0)
--
-- Common part of CSP specifications of master algorithms (MAs)
--
-- Nuno Amalio

--
-- Channels to start and terminate the MA
channel startMA, terminateMA

-- Channels used by Timer (asynchronous simulation steps)
channel tock, timeout

--
-- 'Timer' process issues event 'timeout' when time has elapsed
Timer(t) =
    t == 0 & timeout -> SKIP
    [] t > 0 & tock -> Timer(t-1)

-- Process that waits for a given time until some event of
-- a set happens (set 'evs').
--
-- If it doesn't happen it triggers events associated with 'timeout'
WaitUntilOrTrigger (t, evsB, evsT) =
  let
    WaitUntil =
        t == 0 & Timer(t)
        [] t > 0 & Timer (t) /\ ([] e : evsB @ e -> SKIP)
```

```
    WaitUntilOrTrigger0 =
      WaitUntil [| {|timeout|} |> (; e : evsT @ e->SKIP)
  within
    WaitUntilOrTrigger0 \{|timeout|}

-- Process that enables a MA to set FMU states variables
-- 'SetVarsFMU ' gets FMU index and a set of variable kinds
-- (VK, input, output or state)
SetVarsFMU (i, VK) =
    (|~| k : VK, x : Var, v : Val @ set.i!k!x!v
      -> SetVarsFMU(i, VK))
    |~| SKIP

-- Sets inputs of a a given FMU (i is FMU index)
MASetStepInputs (i) = SetVarsFMU (i, {input})

-- Gets the outputs of a given FMU (i is FMU index)
MAGetStepOutputs (i) =
  let
    Get(i) =
      |~| k : {output, state}, x : Var @
        get.i!k!x?val -> Get(i)
      |~| SKIP
  within
    Get(i)

-- starts up FMUs: instantiates them, sets experiments
--  and initialises them
-- Non-optimised version with interleaving (right way of doing it)
StartupFMUs (ifmus, cint) =
  let
    StartupFMU(i) =
      let
        Instantiate  = |~| st : St @ instantiate.i!st -> SKIP
        Init  = SetVarsFMU (i, VarKind); endInit.i -> SKIP
        Setup  = setup.i!cint -> SKIP
      within
        Instantiate; Setup; Init
  within
    ||| i : ifmus @ StartupFMU(i)

--
-- Efficient version of 'StartupFMUs' (above) without interleaving
-- Optmised version to take the most out of FDR3
StartupFMUsE (ifmus, cint)  =
  let
    StartupFMU(i) =
      let
        Instantiate  = |~| st : St @ instantiate.i!st -> SKIP
```

```
        Init  = SetVarsFMU (i, VarKind); endInit.i -> SKIP
        SetUp  = setup.i!cint -> SKIP
      within
        Instantiate; SetUp; Init
  within
    IterPSeq(StartupFMU, seq(ifmus))

-- MA terminates, telling FMUs to terminate also
-- This is the non-optimised version (the right way of doing it)
MATerminate(ifmus) =
  let
    TerminateFMUs =
      ||| i : ifmus @ terminate.i -> SKIP
  within
    terminateMA->TerminateFMUs

-- Optmised MA terminate (without interleaving), telling FMUs
-- to terminate also.
-- Imposes ordering on terminations to get most out of FDR3
MATerminateE(ifmus) =
  let
    TerminateFMUs =
      let
        TerminateFMU(i) =
          terminate.i -> SKIP
      within
        IterPSeq(TerminateFMU, seq(ifmus))
  within
    terminateMA->TerminateFMUs

-- This takes care of the asynchronous processing of a do-step
-- It waits for some time until stepFinished happens.
-- If it doesn't happen then step is cancelled upon timeout
StepMAPending(i, to) =
  WaitUntilOrTrigger(to, {|stepFinished.i|}, <stepCancel.i>)

StepPendingNoWait(i) = stepCancel.i -> SKIP

-- Builds all possible events of collection of indexed FMUs,
-- from a set of individual FMU events
consFMIEvs(ifmus, fmuEvs) = Union({fmuEvs(i) | i <- ifmus})

-- Relevant events of a co-simulation of FMUs
FMUEvs (i) = {|instantiate.i, setup.i, set.i, endInit.i, doStep.i,
  get.i, terminate.i, doStepOutcome.i, stepCancel.i, stepFinished.i,
  getState.i, setState.i|}

-- Builds set of all events that a FMU may engage in
CosimEvs (ifmus) = consFMIEvs(ifmus, FMUEvs)
```

```
--Represent generic instantiate and setup events
SetupEvs(cint)(i)  = {|instantiate.i, setup.i.cint, set.i,
  endInit.i|}
StepEvs(ct, stp)(i) = {|set.i, get.i, doStep.i.ct.stp,
  doStepOutcome.i, stepCancel.i, stepFinished.i|}

-- Allowed communication time points in a cosimulation interval
Valcts (cint) = {ct | ct<-Time, ct >= fst(cint), ct <= snd(cint)}

-- The Wait version of the step pending
StepPendingWait(i, to) = StepMAPending(i, to)

StepPendingCancel(i, to) = to == 0 & StepPendingNoWait(i)

-- Triggers termination of MA
TriggerMATerminate = terminateMA->SKIP

--
-- A Generic MA that is used to check that the simulations
-- produce the expected events and no others.
GenMA(ifmus, cint, termEvs, discEvs) =
  let
    Terminate = MATerminateE(ifmus)
    Execute =
      let
        CarrySteps(ct) =
          let
            DoSteps (ct, stp) =
              let
                GetStates =
                  let
                    GetState(i) =
                      |~| st : St @ getState.i.ct.st -> SKIP
                  within
                  IterPSeq(GetState, seq(ifmus)) |~| SKIP
                SetStates =
                  let
                    SetState(i) =
                      |~| st : St @ setState.i.ct.st -> SKIP
                  within
                  IterPSeq(SetState, seq(ifmus)) |~| SKIP
                CarryStep =
                  let
                    HandlePending(i) =
                      let
                        Tocks = tock->Tocks |~| SKIP
                        FinishPendingStep =
                          stepCancel.i->SKIP
```

```
                        |~| stepFinished.i->SKIP
                    within
                      Tocks; FinishPendingStep
                  DoStep(i) =
                    doStep.i.ct.stp->
                      (doStepOutcome.i.stepOk -> SKIP
                      |~| doStepOutcome.i.pending
                        -> HandlePending(i)
                      |~| doStepOutcome.i.discard -> SKIP
                      |~| doStepOutcome.i.fail -> SKIP)
                  FullStep(i) =
                    MASetStepInputs(i); DoStep(i);
                      MAGetStepOutputs (i)
                within
                  -- The more efficient version. The right way
                  -- of doing it would resort to interleaving
                  IterPSeq(FullStep, seq(ifmus))
              CarryAnother =
                let
                  ct' = nextCurrentTm (ct, stp, cint)
                within
                  if ct != ct'
                    then DoSteps(ct', stp)
                    else TriggerMATerminate
              CarryOrRollbackStep =
                CarryStep
                  [| consFMIEvs(ifmus, discEvs) |> SetStates
              NextSimAction = CarrySteps(ct) |~| CarryAnother
            within
              if withinCosimInt(ct, cint)
                then GetStates; CarryOrRollbackStep; NextSimAction
                else TriggerMATerminate
        within
          |~| stp : Step @ DoSteps(ct, stp)
      itermEvs = consFMIEvs(ifmus, termEvs)
      Simulate =
        CarrySteps (fst(cint)) [| itermEvs |> TriggerMATerminate
      Startup = StartupFMUsE (ifmus, cint)
    within
      startMA -> Startup; Simulate
  within
    (Execute /\ Terminate) \{|terminateMA|}

-- Complete generic MA
GenMAC(ifmus, cint, termEvs, discEvs) =
  GenMA(ifmus, cint, termEvs, discEvs)

-- Optimised generic MA
GenMAO(ifmus, cint, termEvs, discEvs) =
```

```
  GenMA(ifmus, cint, termEvs, discEvs)
```

A.4 Simple Fixed Step Master Algorithm

```
-- The CSP semantics of the FMI standard (version 2.0)
--
-- This covers a fixed step master algorithm
--
-- Nuno Amalio

include "fmi_ma_common.csp"
include "fmi_fmu.csp"

-- Termination events of fixed-step MA with no support for rollback
-- Algorithm terminates if FMU issues a 'discard'
fsmaTermEvs(i) = {|doStepOutcome.i.fail, doStepOutcome.i.discard,
  stepCancel.i|}

-- A simple fixed step master algorithm (FSMA).
-- It takes a number of FMUs to coordinate (ifmus),
-- a cosimulation interval (cint), a step (stp)
-- and a timeout (to) to process asynchronous steps
FSMA(ifmus, cint, stp, to, StepPending) =
  let
    Simulate =
      let
        CarrySteps(ct) =
          let
            CarryStep =
              let
                Step(i) =
                  let
                    Step0 =
                      doStepOutcome.i.stepOk -> SKIP
                      [] doStepOutcome.i.pending->
                        StepPending(i, to)
                      [] doStepOutcome.i.fail -> SKIP
                      [] doStepOutcome.i.discard -> SKIP
                  within
                    doStep.i!ct!stp -> Step0
                CarryStep0(i) =
                  MASetStepInputs(i); Step(i); MAGetStepOutputs(i)
              within
                -- Does all the steps in an optimised formulation
                -- without interleaving
                IterPSeq(CarryStep0, seq(ifmus))
            ct' = nextCurrentTm(ct, stp, cint)
```

```
            within
              if withinCosimInt(ct, cint)
                then CarryStep; CarryStep(ct')
                else SKIP
        within
          CarryStep(fst(cint)) [| termEvs |> SKIP
      Startup = StartupFMUsE(ifmus, cint)
      Execute = startMA -> Startup; Simulate
      Terminate = MATerminateE(ifmus)
      termEvs = consFMIEvs(ifmus, fsmaTermEvs)
    within
      (Execute; Terminate) \ {|terminateMA|}

-- Complete FSMA
FSMAC(ifmus, cint, stp, to) =
  FSMA(ifmus, cint, stp, to, StepPendingWait)

-- Optimal FSMA (without support for pending)
FSMAO(ifmus, cint, stp) =
  FSMA(ifmus, cint, stp, 0, StepPendingCancel)

-- Defines what a cosimulation with a FSMA is
CosimFSMA(ifmus, cint, stp, to) =
  FSMAC(ifmus, cint, stp, to)
    [| CosimEvs(ifmus) |] (||| i : ifmus @ FMU(i))

-- Defines what a cosimulation with optimised FSMA
-- It allows more normal states non-optimised version
CosimFSMAO(ifmus, cint, stp) =
  FSMAO(ifmus, cint, stp)
    [| CosimEvs(ifmus) |] (||| i : ifmus @ FMU(i))
```

A.5 Checks of the Fixed Step Master Algorithm

```
-- Tests of the fixed-step master algorithm (FSMA)

include "fmi_fsma.csp"

-- STEP used to perform checks on FSMA
FSTP = 1
discEvs(i) = {| |}

--Checks that generic MAs are livelock and deadlock free
assert
  GenMAC({1..5}, (0, 2), fsmaTermEvs, discEvs) :[livelock free]
assert
  Iter(GenMAC({1..5}, (0, 2), fsmaTermEvs, discEvs)) :[deadlock free]
```

```
transparent sbisim, diamond

FSMA1 = sbisim(diamond(FSMAC({1..5}, (0, 2), FSTP, 0)))

-- Checks whether some continuously executing FSMA is deadlock free
-- that does not support asynchronous 'doStep's
assert FSMA1  :[divergence free]
assert Iter(FSMA1 ) :[deadlock free]

assert
  GenMAC({1..5}, (0, 2), fsmaTermEvs, discEvs)  [T= FSMA1

-- Checks whether some continuously executing process is deadlock free
-- that supports asynchronous 'doStep's
assert FSMAC({1..3}, (0, 2), FSTP, 3) :[divergence free]
assert Iter(FSMAC({1..3}, (0, 2), FSTP, 3)) :[deadlock free]

assert GenMAC({1..3}, (0, 2), fsmaTermEvs, discEvs)
  [T= FSMAC({1..3}, (0, 2), FSTP, 3)

assert CosimFSMA ({1}, (0, 2), FSTP, 0) :[livelock free]
assert Iter(CosimFSMA ({1}, (0, 2), FSTP, 0)) :[deadlock free]

assert CosimFSMA ({1, 2}, (0, 2), FSTP, 0) :[livelock free]
assert Iter(CosimFSMA ({1, 2}, (0, 2), FSTP, 0)) :[deadlock free]
-- Checks that only expected events are observed from  co-simulation
assert GenMAC({1, 2}, (0, 2), fsmaTermEvs, discEvs)
  [T= CosimFSMA ({1, 2}, (0, 2), FSTP, 0)

assert CosimFSMA ({1, 2}, (0, 2), FSTP, 3) :[livelock free]
assert Iter(CosimFSMA ({1, 2}, (0, 2), FSTP, 3)) :[deadlock free]

-- Checks that only expected events are observed from co-simulation
assert GenMAC({1, 2}, (0, 2), fsmaTermEvs, discEvs)
  [T= CosimFSMA ({1, 2}, (0, 2), FSTP, 3)

-- Tests of the fixed-step master algorithm (FSMA)

include "fmi_fsma.csp"

-- Some sample FMU states to enable FDR model checking
initst1 = (| 1=>0 |).(| 1=>1 |)
initst2 = (| 1=>0 |).(| 1=>1 |)

-- A MA run that does three sucessful steps with one FMU
FST1 = startMA->instantiate.1!initst1->setup.1!(0,2)
  ->set.1.input.1.1->endInit.1
  ->doStep.1.0.1->doStepOutcome.1.stepOk->get.1.output.1.{1}
  ->doStep.1.1.1->doStepOutcome.1.stepOk
```

```
  ->doStep.1.2.1->doStepOutcome.1.stepOk
  ->terminate.1->SKIP

-- MA run that does two sucessul and
--  one unsucessful step with one FMU
FST2 = startMA->instantiate.1.initst1->setup.1!(0,2)
  ->set.1.input.1.1->endInit.1->doStep.1.0.1
  ->doStepOutcome.1.stepOk->doStep.1.1.1
  ->doStepOutcome.1.stepOk->doStep.1.2.1
  ->doStepOutcome.1.fail
  ->terminate.1->SKIP

-- A MA run with 2 FMUs that does three sucessful steps
FST3 = startMA->instantiate.1!initst1->setup.1!(0,2)
    ->endInit.1
    ->instantiate.2!initst2->setup.2!(0,2)
    ->endInit.2
    ->doStep.1.0.1->doStepOutcome.1.stepOk->get.1.output.1.{1}
    ->doStep.2.0.1->doStepOutcome.2.stepOk
    ->doStep.1.1.1->doStepOutcome.1.stepOk->get.1.output.1.{1}
    ->doStep.2.1.1->doStepOutcome.2.stepOk
    ->doStep.1.2.1->doStepOutcome.1.stepOk->get.1.output.1.{1}
    ->doStep.2.2.1->doStepOutcome.2.stepOk
    ->terminate.1->terminate.2->SKIP

-- A MA run with two FMUs where last step can be okay, fail or discard
FST4 = startMA->instantiate.1.initst1
  ->setup.1!(0,2)->set.1.input.1.1->endInit.1
  ->instantiate.2.initst2
  ->setup.2!(0,2)->endInit.2
  ->doStep.1.0.1->doStepOutcome.1.stepOk->get.1.output.1?{v}
  ->set.2.input.1.v->doStep.2.0.1->doStepOutcome.2.stepOk
  ->doStep.1.1.1->doStepOutcome.1.stepOk->get.1.output.1?{v}
  ->set.2.input.1.v->doStep.2.1.1->doStepOutcome.2.stepOk
  ->doStep.1.2.1->doStepOutcome.1.stepOk->get.1.output.1?{v}
  ->set.2.input.1.v->doStep.2.2.1->
    (doStepOutcome.2.stepOk->SKIP [] doStepOutcome.2.fail->SKIP);
      terminate.1->terminate.2->SKIP

-- A MA run where the last step is asynchronous and is cancelled
FST5 = startMA->instantiate.1.initst1
  ->setup.1!(0,2)->set.1.input.1.1->endInit.1
  ->instantiate.2.initst2
  ->setup.2!(0,2)->endInit.2
  ->doStep.1.0.1->doStepOutcome.1.stepOk->get.1.output.1?{v}
  ->set.2.input.1.v->doStep.2.0.1->doStepOutcome.2.stepOk
  ->doStep.1.1.1->doStepOutcome.1.stepOk->get.1.output.1?{v}
  ->set.2.input.1.v->doStep.2.1.1->doStepOutcome.2.stepOk
  ->doStep.1.2.1->doStepOutcome.1.stepOk->get.1.output.1?{v}
```

```
  ->set.2.input.1.v->doStep.2.2.1
  ->doStepOutcome.2.pending->tock->tock->tock->stepCancel.2
  ->terminate.1->terminate.2->SKIP

-- MA run with FS.
-- Three steps of FMU2 are asynchronous but sucessful (no timeout)
FST6 = startMA-> instantiate.1.initst1
  ->setup.1!(0,2)->set.1.input.1.1->endInit.1
  ->instantiate.2.initst2->setup.2!(0,2)->endInit.2
  ->doStep.1.0.1->doStepOutcome.1.stepOk->get.1.output.1?{v}
  ->set.2.input.1.v->doStep.2.0.1->doStepOutcome.2.pending
  ->stepFinished.2
  ->doStep.1.1.1->doStepOutcome.1.stepOk->get.1.output.1?{v}
  ->set.2.input.1.v->doStep.2.1.1->doStepOutcome.2.pending
  ->tock->stepFinished.2
  ->doStep.1.2.1->doStepOutcome.1.stepOk->get.1.output.1?{v}
  ->set.2.input.1.v->doStep.2.2.1->doStepOutcome.2.pending
  ->tock->tock->stepFinished.2
  ->terminate.1->terminate.2->SKIP

-- STEP used to perform checks on FSMA
FSTP = 1
discEvs(i) = {| |}

-- Checks that tests is traces refinement of 'FSMA'
assert FSMAC({1}, (0, 2), FSTP, 0) [T= FST1

-- Checks that tests is traces refinement of 'FSMA'
assert FSMAC({1}, (0, 2), FSTP, 0) [T= FST2

-- Checks that tests is traces refinement of 'FSMA'
assert FSMAC({1, 2}, (0, 2), FSTP, 0) [T= FST3

-- Checks that tests is traces refinement of 'FSMA'
assert FSMAC({1, 2}, (0, 2), FSTP, 0) [T= FST4

-- Checks that tests is traces refinement of 'FSMA'
assert FSMAC({1, 2}, (0, 2), FSTP, 3) [T= FST5

-- Checks that tests is traces refinement of 'FSMA'
assert FSMAC({1, 2}, (0, 2), FSTP, 3) [T= FST6

-- Checks tests is a traces refinement of
-- a 'FSMA' based cosimulation
assert CosimFSMA({1}, (0, 2), FSTP, 0) [T= FST1

-- Checks test is traces refinement of
-- a 'FSMA' based cosimulation
assert CosimFSMA({1}, (0, 2), FSTP, 0) [T= FST2
```

```
transparent sbisim, diamond

-- Checks that test is a traces refinement of
-- a 'FSMA' based cosimulation
CosimFSMA1 = sbisim(diamond(CosimFSMA({1, 2}, (0, 2), FSTP, 0)))
assert CosimFSMA1 [T= FST3

-- Checks that test is traces refinement of
-- a 'FSMA' based cosimulation
CosimFSMA2 = sbisim(diamond(CosimFSMA({1, 2}, (0, 2), FSTP, 3)))
assert CosimFSMA2 [T= FST4

-- Checks that test is a traces refinement of
-- a 'FSMA' based cosimulation
assert CosimFSMA2 [T= FST5

-- Checks that test is a traces refinement of
-- a 'FSMA' based cosimulation
assert CosimFSMA2 [T= FST6

-- Checks that test is a traces refinement of
-- a 'FSMA' based cosimulation
CosimFSMA3 = sbisim(diamond(CosimFSMA ({1..3}, (0, 2), FSTP, 0)))

-- Checks deadlock for processes that keep executing cosimulation.
assert CosimFSMA3 :[livelock free]
assert Iter(CosimFSMA3 ) :[deadlock free]

assert GenMAC({1..3}, (0, 2), fsmaTermEvs, discEvs) [T= CosimFSMA3

-- This check requires 4 FMUs in the optimised version
CosimFSMA4 = sbisim(diamond(CosimFSMAO({1..4}, (0, 2), FSTP)))

-- Checks deadlock for processes that keep executing cosimulation.
assert CosimFSMA4 :[livelock free]
assert Iter(CosimFSMA4) :[deadlock free]

GenFMI1 =
  sbisim(diamond(GenMAO({1..4}, (0, 2), fsmaTermEvs, discEvs)))
assert GenFMI1 [T= CosimFSMA4

-- Checks if we can handle more FMUs with optmised version without
-- asynchonous calls. This is done for 5 FMUs
CoSimFSMA5 = sbisim(diamond(CosimFSMAO ({1..5}, (0, 2), FSTP)))
assert CoSimFSMA5 :[livelock free]
assert Iter(CoSimFSMA5) :[deadlock free]

GenFMI2 =
```

```
  sbisim(diamond(GenMAO({1..5}, (0, 2), fsmaTermEvs, discEvs)))
assert GenFMI2 [T= CoSimFSMA5
```

A.6 Rollback Master Algorithm

```
-- The CSP semantics of the FMI standard (version 2.0)
--
-- This covers a MA with support for rollback as defined
-- in paper Broman-et-al13
--
-- Nuno Amalio

include "fmi_ma_common.csp"
include "fmi_fmu.csp"

-- The termination events of the rollback MA
rbTermEvs(i) = {|doStepOutcome.i.fail, stepCancel.i,
  doStepOutcome.i.pending|}

discEvs(i) = {|doStepOutcome.i.discard|}

decStep (stp, inc) =
  if (stp-inc) >= 1
    then {stp-inc}
    else {}

-- Channel to update the state of FMUs kept by MA
channel storeSt, readSt : FMUIdx.Time.St
channel reset

FMUStStoreEvs = {| storeSt, readSt, reset|}

channel rollback, finishedDoStep

--
-- The Rollback MA of Broman13
RBMA (ifmus, cint, maxStp, stpInc) =
  let
    FMUSts =
      let
        NotFMUSt (i, ct, st) =
          storeSt.i.ct.st -> IsFMUSt(i, ct, st)
          [] reset -> NotFMUSt (i, ct, st)
        IsFMUSt (i, ct, st) =
          readSt.i!ct!st -> IsFMUSt (i, ct, st)
          [] reset -> NotFMUSt (i, ct, st)
      within
```

```
    [| {reset} |] i : ifmus, ct : Time, st : St @
      NotFMUSt (i, ct, st)
CarrySteps (ct, stp) =
  let
    SaveFMUSts =
      let
        SaveFMUSts0(sifmus) =
          if null(sifmus)
            then SKIP
            else
              getState.head(sifmus)?ct?st
                -> storeSt.head(sifmus)!ct!st
                  -> SaveFMUSts0(tail(sifmus))
      within
        reset->SaveFMUSts0(seq(ifmus))
    CarryStep(stp) =
      let
        DoStepOk =
          let
            DoStepFMUs = IterPSeq(DoStepFMU, seq(ifmus))
            DoStepFMU(i) = MASetStepInputs(i); Step(i)
            Step(i) =
              let
                Step0 =
                  doStepOutcome.i.stepOk -> MAGetStepOutputs (i)
                    [] doStepOutcome.i.pending -> SKIP
                    [] doStepOutcome.i.fail -> SKIP
                    [] doStepOutcome.i.discard -> SKIP
              within
                doStep.i!ct!stp -> Step0
          within
            DoStepFMUs; finishedDoStep->SKIP
        DoStepDiscard =
          let
            SetFMUStates =
              let
                SetFMUState (i) =
                  readSt.i?ct'?st'->
                    setState.i!ct'!st' -> SKIP
              within
                IterPSeq(SetFMUState, seq(ifmus))
          within
            (SetFMUStates |~| TriggerMATerminate); rollback->SKIP
        DoNextStep =
          let
            ost = decStep(stp, stpInc)
          within
            finishedDoStep->
              CarrySteps (nextCurrentTm(ct, stp, cint), stp)
```

```
                [] rollback ->
                  if not empty(ost)
                    then CarryStep (the(ost))
                    else TriggerMATerminate
            DoStep =
              let
                discEvsFMUs = consFMIEvs(ifmus, discEvs)
              within
                DoStepOk [| discEvsFMUs |> DoStepDiscard
            evsDoStep = {|finishedDoStep, rollback|}
          within
            (DoStep [| evsDoStep |] DoNextStep) \ evsDoStep
      within
        if withinCosimInt(ct, cint)
          then SaveFMUSts; CarryStep(stp)
          else TriggerMATerminate
    Simulate =
      let
        termEvs = consFMIEvs(ifmus, rbTermEvs)
        Simulate0 =
          CarrySteps(fst(cint), maxStp)
            [| termEvs |> TriggerMATerminate
      within
        (FMUSts [| FMUStStoreEvs |] Simulate0) \ FMUStStoreEvs
    Terminate = MATerminateE(ifmus)
    Startup = StartupFMUsE (ifmus, cint)
    Execute = startMA-> Startup; Simulate
  within
    (Execute /\ Terminate) \{|terminateMA|}

RBMA1 = sbisim(diamond(RBMA({1..3}, (0, 2), MAXSTP, 1)))

assert RBMA1 :[divergence free]
assert Iter(RBMA1) :[deadlock free]

initst1 = (| 1=>0 |).(| 1=>0 |)
initst2 = (| 1=>0 |).(| 1=>1 |)

-- A MA run that does is not capable of agreeing on a step
RBT1 = startMA->instantiate.1!initst1->setup.1!(0,2)
  ->set.1.input.1.1->endInit.1
  ->getState.1.0.(| 1=>1 |).(| 1=>0 |)
  ->doStep.1.0.2->doStepOutcome.1.discard
  ->setState.1.0.(| 1=>1 |).(| 1=>0 |)
  ->doStep.1.0.1->doStepOutcome.1.discard
  ->terminate.1->SKIP

-- A MA run that does two sucessful steps with support for rollback
RBT2 = startMA->instantiate.1!initst1->setup.1!(0,2)
```

```
  ->set.1.input.1.1->endInit.1
  ->getState.1.0.(| 1=>1 |).(| 1=>0 |)
  ->doStep.1.0.2->doStepOutcome.1.stepOk
  ->get.1.output.1.{1}
  ->getState.1.2.(| 1=>1 |).(| 1=>1 |)
  ->doStep.1.2.2->doStepOutcome.1.stepOk
  ->get.1.output.1.{1}->terminate.1->SKIP

-- A MA run where the first proposed step is rejected
RBT3 = startMA->instantiate.1!initst1->setup.1!(0,2)
  ->set.1.input.1.1->endInit.1
  ->getState.1.0.(| 1=>1 |).(| 1=>0 |)
  ->doStep.1.0.2->doStepOutcome.1.discard
  ->setState.1.0.(| 1=>1 |).(| 1=>0 |)
  ->doStep.1.0.1->doStepOutcome.1.stepOk
  ->get.1.output.1.{1}
  ->getState.1.1.(| 1=>1 |).(| 1=>1 |)
  ->doStep.1.1.1->doStepOutcome.1.stepOk
  ->get.1.output.1.{1}
  ->getState.1.2.(| 1=>1 |).(| 1=>1 |)
  ->doStep.1.2.1->doStepOutcome.1.stepOk
  ->get.1.output.1.{1}->terminate.1->SKIP

-- A MA run with two FMUs
-- First proposed step on 2nd FMU is rejected
RBT4 = startMA->instantiate.1!initst1->setup.1!(0,2)
  ->set.1.input.1.1->endInit.1
  ->instantiate.2!initst2->setup.2!(0,2)
  ->endInit.2
  ->getState.1.0.(| 1=>1 |).(| 1=>0 |)
  ->getState.2.0.initst2
  ->doStep.1.0.2->doStepOutcome.1.stepOk
  -- 2nd FMU Discards
  ->doStep.2.0.2->doStepOutcome.2.discard
  ->setState.1.0.(| 1=>1 |).(| 1=>0 |)
  ->setState.2.0.initst2
  ->doStep.1.0.1->doStepOutcome.1.stepOk
  ->get.1.output.1.{1}
  ->set.2.input.1.1->doStep.2.0.1
  ->doStepOutcome.2.stepOk
  ->getState.1.1.(| 1=>1 |).(| 1=>1 |)
  ->getState.2.1.(| 1=>1 |).(| 1=>1 |)
  ->doStep.1.1.1->doStepOutcome.1.stepOk
  ->get.1.output.1?{0}
  ->set.2.input.1.0->doStep.2.1.1
  ->doStepOutcome.2.stepOk
  ->getState.1.2.(| 1=>1 |).(| 1=>0 |)
  ->getState.2.2.initst2
  ->doStep.1.2.1->doStepOutcome.1.stepOk
```

```
  ->get.1.output.1?{1}
  ->set.2.input.1.1->doStep.2.2.1
  ->doStepOutcome.2.stepOk
  ->terminate.1->terminate.2->SKIP

RBMA2 = sbisim(diamond(RBMA({1}, (0, 2), MAXSTP, 1)))

RBMA3 = sbisim(diamond(RBMA({1, 2}, (0, 2), MAXSTP, 1)))

transparent sbisim, diamond

-- Checks if RBT1, RBT2 and RBT3 are traces refinements of 'RBMA'
assert RBMA2 [T= RBT1
assert RBMA2 [T= RBT2
assert RBMA2 [T= RBT3

assert RBMA3 [T= RBT4

--Defines what a co-simulation with a RBMA is
CoSimRBMA (ifmus, cint, maxStp, stpInc) =
  RBMA (ifmus, cint, maxStp, stpInc)
    [| CosimEvs (ifmus) |] (||| i : ifmus @ FMU(i))

assert
  CoSimRBMA ({1}, (0, 2), MAXSTP, 1) :[divergence free]
assert
  Iter(CoSimRBMA ({1}, (0, 2), MAXSTP, 1)) :[deadlock free]

assert
  GenMAC({1..3}, (0, 2), rbTermEvs, discEvs) :[divergence free]
assert
  Iter(GenMAC({1..3}, (0, 2), rbTermEvs, discEvs)) :[deadlock free]

-- Checks whether RBT1, RBT2 and RBT3 are traces refinements of 'RBMA'
assert CoSimRBMA ({1}, (0, 2), MAXSTP, 1) [T= RBT1
assert CoSimRBMA ({1}, (0, 2), MAXSTP, 1) [T= RBT2
assert CoSimRBMA ({1}, (0, 2), MAXSTP, 1) [T= RBT3

CoSimRBMA2 = sbisim(diamond(CoSimRBMA ({1.. 2}, (0, 2), MAXSTP, 1)))
assert CoSimRBMA2 :[divergence free]
assert Iter(CoSimRBMA2) :[deadlock free]

assert CoSimRBMA2 [T= RBT4

CoSimRBMA3 = sbisim(diamond(CoSimRBMA ({1.. 3}, (0, 2), MAXSTP, 1)))
assert CoSimRBMA3 :[divergence free]
assert Iter(CoSimRBMA3) :[deadlock free]

GenSim3 =
```

```
  sbisim(diamond(GenMAC({1 .. 3}, (0, 2), rbTermEvs, discEvs)))
assert GenSim3 [T= CoSimRBMA3
```

A.7 3 Water Tanks

```
-- The 3 water Tanks example of D21.a in the FMI
--
-- Simulated using a fixed step MA. We abstract away from data.
-- Connection between FMI and more abstract SysML semantics is
-- to be exploited in year 2.
--
-- Nuno Amalio

include "fmi_fsma.csp"

-- This will involve three FMIs: 'TanksControl1', 'TanksControl2',
-- 'Controller'.

-- State of the 'Controller' with one output (valve)
ControllerSt = (| |).(| 1=>0 |)

-- State of the 'TanksControl1' with one input (state of valve) and
-- one output (water outflow)
TanksControl1St = (| 1=>0 |).(| 1=>0 |)

-- State of the 'TanksControl2' with one input (water inflow) and
-- one output (water outflow)
TanksControl2St =  (| 1=>0 |).(| 1=>0 |)

-- Run of algorithm that simulates the 3 water tanks System
WTsSimulation =
  let
    WTsSimulation0(ct) =
      doStep.1.ct.1-> doStepOutcome.1.stepOk
      -> get.1.output.1?{v} -> set.2.input.1.v
      -> doStep.2.ct.1-> doStepOutcome.2.stepOk
      -> get.2.output.1?{w} -> set.3.input.1.w
      -> doStep.3.ct.1-> doStepOutcome.3.stepOk
      -> get.3.output.1?{w2}->SKIP
  within
    startMA -> instantiate.1!ControllerSt -> setup.1!(0,2)
    ->set.1.output.1.1 -> endInit.1
    -> instantiate.2!TanksControl1St -> setup.2!(0,2) -> endInit.2
    -> instantiate.3!TanksControl2St -> setup.3!(0,2)
    -> endInit.3
    -> (WTsSimulation0(0); WTsSimulation0(1); WTsSimulation0(2));
      terminate.1->terminate.2->terminate.3->SKIP
```

```
-- Checks whether 3WTsRun is a traces refinement of 'FSMA'
assert FSMAC({1..3}, (0, 2), FSTP, 0) [T= WTsSimulation

transparent sbisim, diamond

-- This check requires 3 FMUs
CosimFSMA1 = sbisim(diamond(CosimFSMA ({1..3}, (0, 2), FSTP, 0)))

-- Checks whether WTsSimulation is traces refinement of 'CoSimFSMA'
assert CosimFSMA1 [T= WTsSimulation
```

A.8 Periodic Discrete Signal Generator

```
-- The peridoc discrete signal generator (PDSG) example
-- of paper Broman et al 2014 (Requirements for Hybrid Cosimulation)
--
-- This is based on an early version of Jim Woodcock and Ana Cavalcanti
-- The 4 FMUs are modelled as black boxes
--
-- Nuno Amalio

include "fmi_fsma.csp"

-- This will involve four FMUs: 'PDSG1', 'PDSG2', 'Sampler' and
-- 'CheckEquality'.

-- State of the 'PDSG1'
PDSG1St = (| |).(| 1=>0 |)

-- State of the 'PDSG2'
PDSG2St =  (| |).(| 1=>1 |)

-- State of 'Sampler'
SamplerSt = (| 1 => 0, 2=>0|).(| 1=>1 |)

-- State of the 'CheckEqualitySt' component
CheckEqualitySt = (| 1 => 0, 2=>0|).(| 1=> 0 |)

-- The run of the algorithm that simulates the PDSG system
PDSGSimulate =
    startMA->instantiate.1!PDSG1St-> setup.1!(0,2)
    -> endInit.1
    -> instantiate.2!PDSG2St -> setup.2!(0,2)
    -> endInit.2
    -> instantiate.3!SamplerSt -> setup.3!(0,2)
    -> endInit.3
    -> instantiate.4!CheckEqualitySt -> setup.4!(0,2)
```

```
    -> endInit.4
    -- First Run gives equal
    -> doStep.1.0.1-> doStepOutcome.1.stepOk
    -> get.1.output.1?{v}
    -> doStep.2.0.1-> doStepOutcome.2.stepOk
    -> get.2.output.1?{w}
    -> set.3.input.1.v -> set.3.input.2.w
    -> doStep.3.0.1-> doStepOutcome.3.stepOk
    -> get.3.output.1?{w}
    -> set.4.input.1.w
    -> set.4.input.2.w
    -> doStep.4.0.1-> doStepOutcome.4.stepOk
    -> get.4.output.1?{1}
    -- Second Run gives not equal
    -> doStep.1.1.1-> doStepOutcome.1.stepOk
    -> get.1.output.1?{v}
    -> doStep.2.1.1-> doStepOutcome.2.stepOk
    -> get.2.output.1?{w}
    -> set.3.input.1.v -> set.3.input.2.w
    -> doStep.3.1.1-> doStepOutcome.3.stepOk
    -> get.3.output.1?{v2}-> set.4.input.2.v2
    -> set.4.input.1.w
    -> doStep.4.1.1-> doStepOutcome.4.stepOk
    -> get.4.output.1?{0}
    -- Third Run gives equal
    -> doStep.1.2.1-> doStepOutcome.1.stepOk
    -> get.1.output.1?{v}
    -> doStep.2.2.1-> doStepOutcome.2.stepOk
    -> get.2.output.1?{w}
    -> set.3.input.1.v -> set.3.input.2.w
    -> doStep.3.2.1-> doStepOutcome.3.stepOk
    -> get.3.output.1?{v}
    ->set.4.input.2.w-> set.4.input.1.w
    -> doStep.4.2.1-> doStepOutcome.4.stepOk
    -> get.4.output.1?{1}
    -- The termination phase
    -> terminate.1-> terminate.2-> terminate.3
    -> terminate.4-> SKIP

-- Checks whether 'PDSGRun' is a traces refinement of 'FSMA's
assert FSMAO ({1..4}, (0, 2), FSTP) [T= PDSGSimulate

assert FSMAC ({1..4}, (0, 2), FSTP, 0) [T= PDSGSimulate

transparent sbisim, diamond, wbisim


CosimFSMA1 = sbisim(diamond(CosimFSMAO ({1..4}, (0, 2), FSTP)))
```

```
-- Checks whether PDSGRun is a traces refinement of 'CoSimFSMA'
assert CosimFSMA1 [T= PDSGSimulate
```