Grant Agreement: 644047

INtegrated TOol chain for model-based design of CPSs

A Mechanised FMI Semantics

Deliverable Number: D2.3c

Version: 1.0

Date: December 2017

Public Document

http://into-cps.au.dk

Contributors:

Frank Zeyda, UY
Ana Cavaclcanti, UY
Simon Foster, UY
Jim Woodcock, UY
Julien Ouy, CLE

Editors:

Frank Zeyda, UY

Reviewers:

Casper Thule, AU
Christian Kleijn, CLP
Ken Pierce, 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	24-05-2017	Frank Zeyda	Initial document (section headers)
0.2	08-11-2017	Frank Zeyda	Draft for internal review
1.0	15-12-2017	Frank Zeyda	Final version

Abstract

In this deliverable, we present a formal semantics of FMI co-simulations and mechanisation of that semantics in our theorem prover Isabelle/UTP. This, firstly, involves embedding the *Circus* language in our proof tool. Secondly, we use that embedding to encode the processes of the FMI model. Our mechanised encoding makes precise the structure of those processes, and takes advantage of reasoning support and tactics available in Isabelle/UTP. We, thirdly, illustrate how the mechanisation can be used to encode a particular co-simulation. For illustration, we use an example from railways provided by our industrial partners. In doing so, we relate our approach here to the one in INTO-CPS Deliverable D2.3a, which considers an abstract co-simulation model. We also explain how both works are part of an integrated approach and single technique to formally encode and analyse co-simulations.

Contents

1 Introduction

This report constitutes the final deliverable on FMI modelling foundations. It extends the earlier deliverables D2.1d and D2.2d on this topic by describing a comprehensive model of FMI in the *Circus* process algebra, and a mechanisation of the model in the theorem prover Isabelle/UTP. The mechanised model is instantiated using a case study from railways (see INTO-CPS Deliverable D1.2b), and we also discuss reasoning and proof support, in particular, in relation to the verification strategy in INTO-CPS Deliverable D2.3a.

1.1 Background and Motivation

Cyber-Physical Systems (CPS) are systems that comprise both real-world entities and digital components. Modelling and designing CPSs typically requires a combination of different languages and tools that adopt complementary specification paradigms. For real-world artefacts, physics models in the form of differential equations are the norm. Digital components, such as software controllers, are typically described via control diagrams, state machines, and real-time programs. This diversity of specification and design methods makes CPS challenging to study and analyse.

Co-simulation [16] is perhaps the *de facto* technique for analysing the behaviour of CPS. It requires that models of artefacts are simulated in isolation, while master algorithms control the various simulators and thereby orchestrate the co-simulation as a whole. This, however, raises issues of interoperability between the master algorithm and simulators. The Functional Mock-up Interface (FMI) Standard [11] has been proposed to alleviate those issues, and has since been successfully used in many industrial applications.

The FMI standard prescribes how master algorithms (MA) and simulators communicate. It does so by virtue of a bespoke API that simulators have to implement, and that can be used to implement compliant master algorithms. The API enables master algorithms to exchange data between the components of a co-simulation, called FMUs (Functional Mock-up Units), perform simulation steps, and suitably deal with errors in simulators. It also allows for advanced features such as roll-back of already performed steps.

While (co)simulation is currently the predominant approach to validate CPS models, we here describe a complementary technique based on a formal model

of an FMI system. Our technique formalises both the master algorithm and the simulated FMUs, and allows for verification of their properties.

Whereas (co)simulation helps engineers to quickly gauge the implications of modelling and design decisions, our formal analysis has the potential to complement simulation with universal guarantees, both about the master algorithm and cosimulated system. The former is important since simulations depend on parameters and algorithms, and are software systems (with possible faults) in their own right. The latter is important since it is usually not possible to run an exhaustive number of simulation scenarios as a means of testing the system towards producing strong certification evidence.

For our formal modelling, we use *Circus* — a process algebra with added features for supporting stateful models. It has proved adequate and useful for modelling master algorithms [8] due to its capabilities of capturing concisely the data and control aspects of such algorithms, including data exchange between the FMUs and their concurrent execution. *Circus* models can be subjected to verification techniques. These include both model-checking approaches [17], refinement [7], and (automatic) theorem proving [24].

1.2 Contribution

In INTO-CPS Deliverable D2.3a, we present an abstract relational model of FMI co-simulations that focusses on the essence of the FMI computational paradigm. In this deliverable, we consider a concrete reactive model of FMI that, in addition, faithfully models the FMI interface as well as master algorithms. It extends and elaborates the work in the earlier INTO-CPS Deliverable D2.2c by providing a comprehensive *Circus* model that we mechanise in the theorem prover Isabelle/UTP [15].

Added contributions of this deliverable are summarised as follows.

1. We present a complete and final *Circus* model of the FMI standard for co-simulation[1];
2. We embed the *Circus* language into our theorem prover Isabelle/UTP;
3. We mechanise our FMI *Circus* model in Isabelle/UTP, using the *Circus* language embedding above;
4. We illustrate the use of our mechanisation by applying it to one of the industrial INTO-CPS case studies (a railways system);

[1]We note that we do not consider FMI for model exchange here.

5. We discuss reasoning and proof support for encoded models.

With regards to (5), we pursue a technique, based on refinement, to show compliance of master algorithms with regards to the FMI standard [1]. Unlike other approaches such as [10], we can profit from high-level algebraic laws and a stepwise approach that culminates in executable code, for both the FMUs and master algorithm.

We also explain how our work here completes the general reasoning technique presented in INTO-CPS Deliverable D2.3a. That technique proposes a refinement-based approach: we start with a discrete abstraction of a co-simulation that does not need to consider the MA and is used to establish fundamental safety properties. Our work in this deliverable fills an important gap: the transformation of an abstract FMU model into a concrete one that can be translated into code.

The rest of the report is structured as follows. In Section 2, we review preliminary material. This is the Functional Mock-up Interface (FMI), *Circus*, and the Isabelle/UTP proof tool. Section 3 presents a comprehensive *Circus* model of the FMI standard and interface, and in Section 4 we discuss its mechanisation in Isabelle/UTP. This also includes an account on embedding the *Circus* language in Isabelle/UTP. The case study and its mechanisation are described in Section 5. In Section 6, we conclude and discuss noteworthy related work.

2 Preliminaries

We begin by reviewing preliminary material: the Functional Mock-up Interface (FMI) in Section 2.1, *Circus* and CSP in Section 2.2, and Isabelle/UTP in Section 2.3.

2.1 The Functional Mock-up Interface

Co-simulation of CPSs is a popular technique in industry but poses practical challenges. The most significant one is how to couple different simulators for particular languages and models, and orchestrate their use to drive the co-simulation process. To address this problem, the FMI standard [1] has been developed jointly by the Modelica Association and several industrial partners. At its heart, it defines an API that prescribes the interaction of simulation components with tools that manage and control them.

Simulators are referred to as Functional Mock-up Units (FMUs). They are passive black-box entities (slaves) that are orchestrated by a master algorithm (MA). The master algorithm instantiates the FMUs, initialises them, sets their parameters, and performs data exchange between them during co-simulation steps. All of these tasks are facilitated by functions of the FMI API, and the FMI standard moreover specifies the protocol by which interactions for these tasks have to be preformed.

The MA is also responsible for determining the size of simulation steps, using a number of different strategies adequate for the co-simulation. Thereby, it ensures fidelity of the simulation with respect to the underlying real-world system.

The high-level conceptual view of an FMI architecture, illustrated in Fig. 1, entails one master algorithm and several FMUs that wrap vendor-specific simulation components. Typically, the master algorithm reads outputs from all FMUs and then forwards them to those FMUs that require them as inputs. After this, the MA notifies the FMUs to concurrently compute the next simulation step. Some MAs assume a fixed step size while others enquire the largest step size that the FMUs are cumulatively willing to accept. Additionally, MAs can sometimes perform roll-backs of simulation steps.

The three key aspects of the FMI paradigm are that (a) FMUs do not communicate directly with each other, so that all data exchange is carried out by the MA; (b) they proceed synchronously, following the BSP (Bulk Synchronous

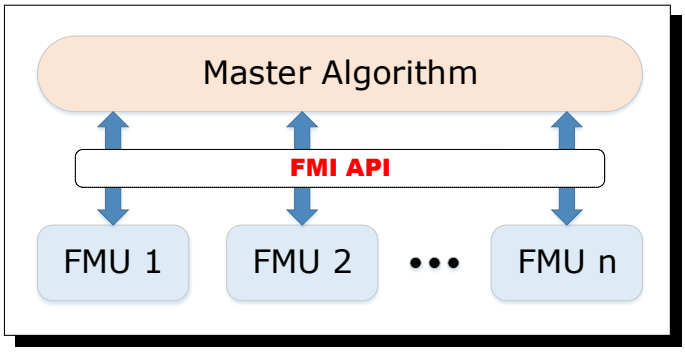

Figure 1: High-level view of the FMI design pattern.

Parallel) model of concurrency [27]; and (c) the dependency between FMUs implied by their connected input and output ports is free of algebraic loops. The third caveat ensures stability of the co-simulation, but does not exclude feedback systems as long as they do not exhibit cycles with direct dependencies. Finally, the FMI standard also constrains what kind of data can be exchanged between FMUs: real, integer, boolean and string values.

The FMI specification standard [1] is elaborate and extensive but informal. This motivates our work here to provide a formal semantics that can be used to reason about co-simulation architectures.

2.2 *Circus*

Circus is a process algebra similar to CSP [19], but with additional support for defining data operations and state. *Circus* inherits many of its process operators from CSP, including sequential and parallel composition, input and output communications on a channel c, synchronisation, external choice, interrupt, guard and recursion. A summary of *Circus* constructs relevant to our models in this deliverable is given in Table 1.

To define a process state, a *Circus* process declares a record in its **state** paragraph, whose fields define a data model. Data operations can either be written as Z operation schemas [28] or constructs from Morgan's refinement calculus [23], such as specification statements, assignments, conditionals and iteration. *Circus* has a rich set of laws that can be used for verification and refinement [7]; it is thus an ideal language for developing state-rich implementations from abstract specifications of reactive systems.

A notable trade-off in *Circus* is that the language enforces non-interference of

Name	Syntax	Description
Sequence	$A \,;\, B$	Execute A and B in sequence.
Parallel Comp.	$A \llbracket cs \rrbracket B$	Execute A and B in parallel, synchronising on the channels in the channel set cs.
External Choice	$A \,\Box\, B$	The environment decides whether to execute A or B; communication resolves the choice.
Synchronisation	$c \longrightarrow A$	Synchronisation on a (typeless) channel c.
Input Prefix	$c?x \longrightarrow A(x)$	Input a value x on a typed channel c.
Output Prefix	$c!e \longrightarrow A$	Output a value e on a typed channel c.
Guarded Action	$g \,\&\, A$	Proceed with A only if g is true.
Interrupt	$A \,\triangle\, c \longrightarrow B$	A is interrupted by synchornisation on channel c.
Recursion	$\mu\, X \bullet F(X)$	Execution recursive action F.
Assignment	$x := e$	Assignment to a state component x.

Table 1: Overview of relevant *Circus* action operators.

parallel computations; this endows it with a monotonic parallel composition operator that facilitates piecewise and compositional development.

An example of a *Circus* process *Timer* is included in Fig. 2. It is part of the FMI model that we discuss in more detail in the next section. The process defines a state record *State* that introduces two variables: *currentTime* and *stepSize* of type *TIME* (which models simulation time). It also includes a local action *Step*.

The main action after the '•' at the bottom prescribes the behaviour of the process. In our example, it first initialises the state variables and then proceeds by calling *Step*. For initialisation, we refer to the variables *ct* and *hc*, which are parameters of the process. *Step* is an external choice (operator □) that offers communication on the channels *setT*, *updateSS*, *step* and *end*. These channels are declared (with their types) by the two **channel** constructs above the process, the first introducing typed channels and the second introducing untyped channels used for synchronisation only.

The channel events here are used by co-simulation (master) algorithms to model the progression of time during co-simulation steps. The environment can change *currentTime* and *stepSize* through communication on the channels *setT* and *updateSS*, respectively. In addition, when a *step* event occurs, modelling a co-simulation step, both these values are communicated and *currentTime* is increased by *stepSize*. Lastly, an *end* event may occur only if $currentTime = tN$, where tN is a process parameter specifying the simulation end time. The **stop** action that follows effectively refuses any further

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

Figure 2: Timer process of the *Circus* FMI specification.

interaction. Otherwise, the *Step* action behaves recursively, repeating the previously described behavioural pattern.

An important modelling paradigm in *Circus* is that processes can encapsulate particular isolated concerns. They do so by restricting the events and communications that can occur in a system, namely when processes are composed in parallel and have to synchronise on their channels. The general form of parallel composition in *Circus* is $P \llbracket cs \rrbracket Q$, where cs provides the synchronisation channels. When there is no synchronisation, we also write $P \mid\mid\mid Q$ for the interleaving of two processes.

To conclude the preliminary material, we briefly describe our theorem prover Isabelle/UTP.

2.3 Isabelle/UTP

Isabelle/UTP is a theorem prover implemented within the Isabelle proof assistant and logic of HOL. It supports proof in the context of Hoare and He's Unifying Theories of Programming (UTP) [20]. This is a general and unifying framework to define programming language semantics, and we have used it to encode *Circus*, amongst other languages.

The UTP adopts a predicative approach that represents computational models as relations over a theory-specific alphabet of variables. Those variables determine the observable quantities and can, for instance, include the state

variables of a program, traces of a reactive process, or trajectories of a hybrid system. State spaces in Isabelle/UTP are modelled by record types (named tuples). In Isabelle/UTP, we use the command `alphabet` to construct such types. Below is an example that introduces three variables, `x`, `y` and `z`.

```
alphabet my_state =
  x :: "real"
  y :: "real"
  z :: "real"
```

The `alphabet` command is similar to Isabelle/HOL's built-in `record` command for introducing record types, but caters for some additional set-up in the context of UTP. We give a detailed technical explanation of it in [15]. To give an example of a predicate encoding, let us consider a model of the assignment $z := x * y$. We encode it as follows in Isabelle/UTP.

```
"$x´ =u $x ∧ $y´ =u $y ∧ $z´ =u $x * $y"
```

The above corresponds to the hand-written relational predicate

$$x' = x \wedge y' = y \wedge z' = x * y \quad .$$

Primed variables are used to refer to the values of variables after a computation has finished, and plain (unprimed) variables refer to their values at the start of a computation. Whereas the third conjunct specifies the new value of z, we note that the first two conjuncts are necessary to ensure that x and y retain their values.

The encoding illustrates a few salient points about Isabelle/UTP. First of all, variables have to be prefixed with either `&` or `$`, depending on whether they are used in the context of a plain predicate that does not allow primed variables, or in the context of a relational predicate that does so, like the one above. Secondly, operators (such as '=' above) usually have to be subscripted to delineate them from HOL operators. There are a few exceptions to this; for instance, arithmetic operators can be written as in HOL.

Important to note is that the general view of the UTP modelling computations as predicates facilitates a contractual view. For instance, more generally, predicates of the form $ok \wedge P \Rightarrow ok' \wedge Q$ specify total-correctness programs as familiar pre- and postcondition pairs (P, Q). Here, ok and ok' are special boolean variables that record whether a computation has started or terminated. The refinement of specifications into programs is simply reverse implication in this model, which we call the UTP theory of *designs*.

As part of INTO-CPS Deliverable D2.3b, we extend the notion of contract to *reactive contracts*. These enable us to specify the observable interactions

of a reactive process — that is, a processes that communicates with its environment. To write such contracts, we use the notation $[P \mid R \diamond Q]$ where P is the contract's precondition, Q its postcondition, and R its pericondition. Whereas pre- and postcondition play similar roles as in sequential programs, the pericondition characterises nonterminating albeit nondivergent behaviours, since reactive computations may not terminate and yet do useful things by meaningfully communicating and interacting with their environment. The three predicates typically impose constraints on a trace variable tr, recording histories of interactions, and refusal variable ref, recording the willingness of a process to communicate on a channel.

We summarise by noting that Isabelle/UTP is a complex tool that, at present, supports many of the theories described in the UTP book [20], as well as those relevant to the models in this report. The earlier mentioned artefacts in encoding UTP (vs HOL) predicates arise since we adopt a deep logical representation of UTP predicates in which variables are first-class entities, and operators from nominal logic can be formalised too.

This concludes our introduction of UTP and Isabelle/UTP. Next, we present a general description of our analysis and verification technique for FMI.

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

Table 2: Channels that model FMI API functions.

3 The *Circus* Model of FMI

As explained in Section 2.1, the FMI standard [1] defines an API used by master algorithms to communicate with simulators (FMUs). The FMI API consists of C functions used by the master algorithm to drive and orchestrate the co-simulation. We model these functions as *Circus* channels whose types correspond to the input and output types of those functions. The respective channels are summarised in Table 2.

To represent instances of FMUs, we use a given type *FMI2COMP*. In FMI, these are pointers to FMU-specific structures that contain the information needed to simulate them. Here, we use symbolic identifiers for them.

Variable names and values are represented by elements of the sets *VAR* and *VAL*. We do not model the FMI type system separately, which includes reals, integers, booleans, and strings; however, it is not difficult to cater for such types in *Circus*, which has a strongly-typed setting that also supports parametric and generic types. We hence pursue a shallow embedding of types that harness typing in *Circus* (and HOL) to validate FMU types. This is a simple and standard approach. Extensions to types are expected in future versions of FMI, and we are well equipped to accommodate them.

The type *FMI2ST* contains flags of FMI type `fmi2Status` that are returned by most of the API functions. We consider the values `fmi2OK`, `fmi2Error`, and `fmi2Fatal`, which indicate, respectively, that all is well, the FMU encountered an error, and the computations are irreparable for all FMUs. The extra flag `fmi2Discard` is also included in the super-type *FMI2STF*; it can only be returned by `fmi2Set` and `fmi2DoStep`. In doing so, `fmi2Set` indi-

cates that a status cannot be returned, and in the case of `fmi2DoStep` that a smaller step size is required or the requested information cannot be returned. We do not include `fmi2Warning`, used only for logging, and `fmi2Pending`, used for asynchronous simulation steps.

Our *Circus* model only captures the discrete observations of the simulation steps and their associated data exchanges. It is compatible with the view of a co-simulation as a sequence of discrete steps that define points for synchronisation and exchange of data. Modelling continuous behaviour takes place within particular FMU models, and wrapper processes are used to extract discrete simulation behaviours from those continuous models. Details of this are the subject of INTO-CPS Deliverable D2.3d on linking models.

FMUSTATE contains values that represent internal states of an FMU. It comprises all values (of parameters, inputs, buffers, and so on) needed to continue a simulation at a desired point in time. The FMU state can, for instance, be recorded by a master algorithm to support roll-back of simulation steps that have already been carried out. We note that the concrete definition of the *FMUSTATE* type depends on the particular FMU architectures.

The signature of the channels in Table 2 captures the FMI standard restrictions on the use of the API in our *Circus* model. It is, for instance, not possible to call `fmi2DoStep` with a non-positive step size, as such would raise a type error.

Unlike the FMI standard, which provides separate `fmi2Get` and `fmi2Set` functions for different types, we only define a single pair of channels. Their inconsistent use, however, results in deadlock in our model. While verification techniques using model checking and proof can show the absence of deadlocks, a proof of well-formedness of an FMI architecture, as described in INTO-CPS Deliverable D2.3a, already ensures that FMU models behave in a type-conformant manner, so that deadlock cannot occur due to this issue.

The API function `fmi2Instantiate` returns a pointer to a component, and the `null` pointer if instantiation fails. Since we do not model pointers, we use a boolean to cater for the possibility of failure. Lastly, the function `fmi2GetMaxStepSize` is not part of the standard. It is typically used in MA implementations to avoid roll-back, see the example in [10].

The overall structure of our model of a co-simulation is depicted in Fig. 3. The visible channels by an environment are `fmi2Get`, `fmi2Set`, and `fmi2DoStep`. The other channels are internal and enforce the expected control flow of a master algorithm. They are used for communication between the process *MAlgorithm* that models an MA and each process $FMUInterface(i)$ that

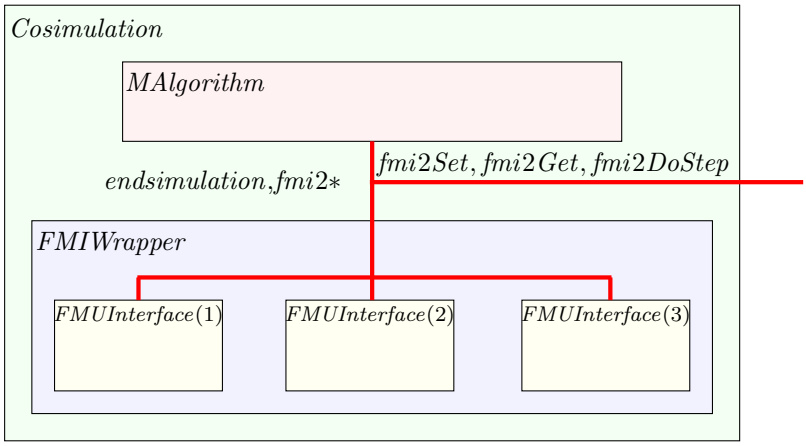

Figure 3: High-level structure of a co-simulation model.

models an FMU. We call *FMIWrapper* the collection of FMU interfaces: they execute independently and in parallel, and bridge between the models of particular simulators and the *Circus* model of the complete co-simulation as described here.

The control channel *endsimulation* is used to shutdown the simulation. Since an FMU may fail, its termination may not be carried out gracefully, via `fmi2Terminate` and `fmi2FreeInstance`. So, *endsimulation* is used to indicate the end of the experiment, causing the termination of all *Circus* processes that are part of the model.

In what follows, we describe our specifications of *MAlgorithm* (Section 3.1) and *FMUInterface* (Section 3.2), which provide a correctness criterion for these components. In Section 3.3, we describe how to construct models of specific FMUs. We recall that mechanisation and instantiation of the model with an example is the subject of Sections 4 and 5.

3.1 Master Algorithms

A master algorithm (MA) is a monolithic program that exchanges data between FMUs, determines the size of simulation steps, and handles any errors raised by an FMU. In our model, we consider each of these aspects of a master algorithm separately. The overall structure of the proposed *MAlgorithm* process is described in Fig. 4. It provides a general characterisation of the valid history of interactions of a master algorithm, as per the FMI 2.0 standard [1]. It does not commit to specific policies to define step sizes and error

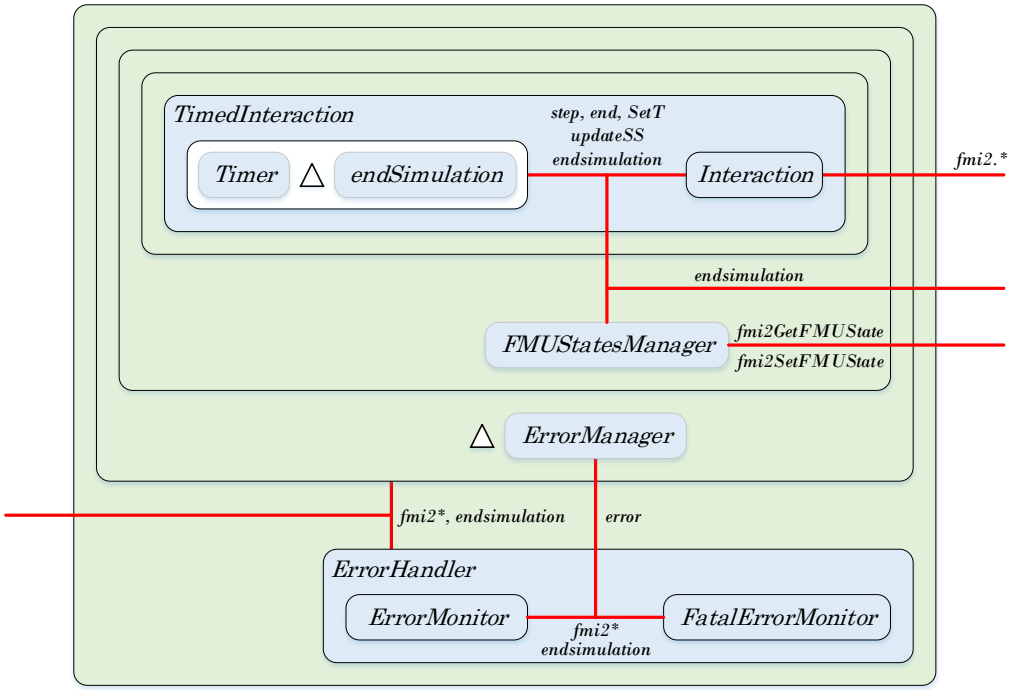

Figure 4: Structure of a model of a master algorithm.

handling in case of an API function returning `fmi2Discard`. The treatment of `fmi2Error` and `fmi2Fatal` is restricted by the standard.

MAlgorithm has three main components described next. Firstly, the **Circus** process *TimedInteraction* specifies the co-simulation steps and orchestration of the FMUs. Secondly, *FMUStatesManager* controls access to the internal state of the FMUs. And lastly, *ErrorHandler* monitors the occurrence of an `fmi2Error` or `fmi2Fatal` from the API functions.

The *TimedInteraction* process has two components. The *Timer* process was already presented in Section 2.2. It uses channels *step* and *end* to drive the *Interaction* process, which performs the orchestration of the FMUs. *Interaction* is the core process that restricts the order in which the API functions can be used by an implementation of a master algorithm. We note that algorithms with roll-backs or a variable step size can in particular make use of the channels *setT* and *updateSS* exposed by the *Timer* process to modify the current simulation time and step size. The *Timer* process can be terminated by synchronisation on *endsimulation* raised by *Interaction*.

The complete *Interaction* process is presented in Fig. 5. The only action that has been omitted is *Step*, which we discuss separately later on.

$$
\begin{array}{l}
\textbf{process}\ \mathit{Interaction} \mathrel{\widehat=} \textbf{begin} \\
\textbf{state}\ \mathit{State} == [\mathit{rinps} : \mathit{PORT} \pfun \mathit{VAL}] \\
\mathit{Instantiation} \mathrel{\widehat=} (;\ i : \mathit{FMUs} \bullet \mathit{fmi2Instantiate}.i?sc \longrightarrow \textbf{skip}) \\
\mathit{InstantiationMode} \mathrel{\widehat=} (;\ (i,x,v) : \mathit{parameters} \bullet \mathit{fmi2Set}!i!x!v?st \longrightarrow \textbf{skip}); \\
\quad \left(\begin{array}{l} ;\ i : \mathit{FMUs} \bullet \mathit{fmi2SetUpExperiment}!i!\mathit{startTime}!\mathit{stopTimeDefined} \cdots \\ \quad \cdots !\mathit{stopTime}?st \longrightarrow \textbf{skip} \end{array} \right); \\
\quad (;\ i : \mathit{FMUs} \bullet \mathit{fmi2EnterInitializationMode}.i?st \longrightarrow \textbf{skip}) \\
\mathit{InitializationMode} \mathrel{\widehat=} \\
\quad (;\ ((i,x),v) : \mathit{initialValues} \bullet \mathit{fmi2Set}!i!x!v?st \longrightarrow \textbf{skip}); \\
\quad (;\ i : \mathit{FMUs} \bullet \mathit{fmi2ExitInitializationMode}!i?st \longrightarrow \textbf{skip}) \\
\mathit{TakeOutputs} \mathrel{\widehat=} \mathit{rinps} := \varnothing\ ; \\
\quad (;\ \mathit{out} : \mathit{outputs} \bullet \mathit{fmi2Get}.(\mathit{FMU}\ \mathit{out}).(\mathit{name}\ \mathit{out})?v?st \longrightarrow \\
\quad (;\ \mathit{inp} : \mathit{pdg}(\mathit{out}) \bullet \mathit{rinps} := \mathit{rinps} \cup \{\mathit{inp} \mapsto v\}) \\
\mathit{DistributeInputs} \mathrel{\widehat=} \\
\quad (;\ \mathit{inp} : \mathit{inputs} \bullet \mathit{fmi2Set}.(\mathit{FMU}\ \mathit{inp}).(\mathit{name}\ \mathit{inp})!\mathit{rinps}(\mathit{inp})?st \longrightarrow \textbf{skip}) \\
\mathit{Step} \mathrel{\widehat=} t : \mathit{TIME};\ \mathit{ss} : \mathit{NZTIME} \bullet \cdots \text{(See Fig. 6 on page 22)} \\
\mathit{slaveInitialized} \mathrel{\widehat=} \left(\begin{array}{l} (\mathit{end} \longrightarrow \mathit{Terminated}) \ \Box \\ \mathit{step}?t?\mathit{ss} \longrightarrow \left(\begin{array}{l} \mathit{TakeOutputs}; \\ \mathit{DistributeInputs}; \\ \mathit{Step}(t,\mathit{ss}); \\ \mathit{NextStep} \end{array} \right) \end{array} \right) \\
\mathit{NextStep} \mathrel{\widehat=} \left(\begin{array}{l} (\mathit{updateSS}?d \longrightarrow \mathit{NextStep}) \\ \Box \\ (\mathit{setT}?t \longrightarrow \mathit{NextStep}) \\ \Box \\ (\mathit{slaveInitialized}) \\ \Box \\ (\mathit{Terminated}) \end{array} \right) \\
\mathit{Terminated} \mathrel{\widehat=} \\
\quad (;\ i : \mathit{FMUs} \bullet \mathit{fmi2Terminate}.i?st \longrightarrow \mathit{fmi2FreeInstance}.i?st \longrightarrow \textbf{skip}); \\
\quad \mathit{endsimulation} \longrightarrow \textbf{skip} \\
\bullet\ \mathit{Instantiation}\ ;\ \mathit{InstantiationMode}\ ;\ \mathit{InitializationMode}\ ;\ \mathit{slaveInitialized} \\
\textbf{end}
\end{array}
$$

Figure 5: Definition of the *Interaction* process.

The main action of *Interaction* is the sequential composition of *Instantiation*, *InstantiationMode*, *InitializationMode* and *slaveInitialized*. These four local actions model the phases of a co-simulation [1, page 103]. The precise behaviour of these actions depends on the configuration of the FMUs. That configuration is provided by several global constants which we refer to in the process definition, and describe next in more detail.

Before proceeding, we introduce the notion of a port. A port is a pair of an FMU identifier and variable, hence of type $FMICOMP \times VAR$. To obtain the FMU of a port, we use the function *FMU*; and to obtain its variable, we use the function *name*. A configuration is then characterised by:

1. a sequence of FMU identifiers *FMUs* of type $\mathrm{seq}(FMI2COMP)$;
2. a sequence *parameters* of type $\mathrm{seq}(FMI2COMP \times VAR \times VAL)$ that provides the parameters of FMUs and their values;
3. a sequence *inputs* of type $\mathrm{seq}(PORT)$ of input ports;
4. a sequence *outputs* of type $\mathrm{seq}(PORT)$ of output ports;
5. a sequence *initialValues* of type $\mathrm{seq}(PORT \times VAL)$ that specifies an initial value for each input;
6. a port dependency graph [10] *pdg* of type $PORT \nrightarrow \mathbb{P}(PORT)$ that maps outputs ports to their connected input ports; and
7. a function *idd* of type $PORT \nrightarrow \mathbb{P}(PORT)$ that records direct dependencies between inputs and outputs inside FMUs.

The *idd* function is not relevant for the ***Circus*** model but important to check well-formedness of an architecture. Namely, the union of both graphs *pdg* and *idd* must be acyclic to show the absence of algebraic loops. Details of this are in INTO-CPS Deliverable D2.3a.

The first action *Instantiation* (see Fig. 5) instantiates the FMUs. It is an iterated sequence of actions $fmi2Instantiate.i?sc \longrightarrow \mathbf{skip}$, where i ranges over *FMUs* and **skip** is the action that does nothing and terminates.

As per the FMI standard [1], it is valid to instantiate the FMUs in any order, and so we could potentially have the events $fmi2Instantiate.i?sc$ in interleaving. Accommodating this flexibility is a simple change from an iterated sequence $(;\, i : FMUs \bullet \ldots)$ to an iterated interleaving of actions $(|||\, i : FMUs \bullet \ldots)$. Our model captures good design practice: handling all FMUs in a specific order, which also facilitates model checking. That order is determined by the user in setting the value of *FMUs* (1).

InstantiationMode and *InitializationMode* perform the setting of parameters and initial values of inputs before calling the API function `fmi2ExitInitializationMode` that signals the start of the next phase. We recapture the definition of *InitializationMode* from Fig. 5 below.

$$
\begin{array}{l}
\mathit{InitializationMode} \mathrel{\widehat=} \\
\quad (\mathbin{;}((i,x),v) : \mathit{initialValues} \bullet \mathit{fmi2Set}!i!x!v?st \longrightarrow \mathbf{skip}); \\
\quad (\mathbin{;} i : \mathit{FMUs} \bullet \mathit{fmi2ExitInitializationMode}!i?st \longrightarrow \mathbf{skip})
\end{array}
$$

As with the *Instantiation* action, we can easily generalise the model to allow for an interleaving of the events involved.

The local action *slaveInitialized*, recaptured below, corresponds to the main phase of the co-simulation; it is driven by the *Timer* process.

$$
\begin{array}{l}
\mathit{slaveInitialized} \mathrel{\widehat=} \\
\left(\begin{array}{l}
(\mathit{end} \longrightarrow \mathit{Terminated}) \;\Box \\
\mathit{step}?t?ss \longrightarrow \left(\begin{array}{l}
\mathit{TakeOutputs}; \\
\mathit{DistributeInputs}; \\
\mathit{Step}(t, ss); \\
\mathit{NextStep}
\end{array}\right)
\end{array}\right)
\end{array}
$$

Two interactions are possible. Firstly, synchronisation on the channel *end* initiates termination of a co-simulation. Termination is carried out by the local action *Terminated*, which raises *fmi2Terminated* and *fmi2FreeInstance* events for all of FMUs, in sequence (see Fig. 5 for its definition).

The *step.t.ss* event is raised by the *Timer* process. Its purpose is to perform the next co-simulation step. Such consists of a sequence of three conceptual tasks: (a) recording the outputs of all FMUs (*TakeOutputs*); (b) forwarding their values to their connected inputs (*DistributeInputs*); and (c) telling FMUs to compute the next simulation step result (*Step*). The purpose of the fourth action *NextStep* is essentially to recurse back into *slaveInitialized*, albeit making additional interactions available, to set the current time (*setT*) and update the simulation step size (*updateSS*).

Similarly to that of *InitializationMode*, the definition of *TakeOutputs* (Fig. 5) uses an iterated sequence, now over *outputs*. Once the value v of an output port *out* is obtained, we store it inside the *rinps* state component of the process. More precisely, what we record is a tuple (inp, out) for each input *inp* connected to the output *out*. Those inputs are obtained by $pdg(out)$ as *pdg* captures the FMI diagram structure. Outputs are read using communications on *fmi2Get*, corresponding to calls of the `fmi2Get` API function.

$$
\begin{array}{l}
\textbf{process}\ \mathit{Interaction} \mathrel{\widehat=} \textbf{begin} \\
\textbf{state}\ \mathit{State} == [\,\mathit{rinps} : \mathit{PORT} \pfun \mathit{VAL}\,] \\
\ldots \\
\mathit{Step} \mathrel{\widehat=} t : \mathit{TIME};\ ss : \mathit{NZTIME} \bullet\ ;\ i : \mathrm{dom}(\mathit{FMUs}) \bullet \\
\left(\begin{array}{l}
\textbf{if}\ i = 0 \longrightarrow (\mathit{stepToComplete} \longrightarrow \mathit{fmi2DoStep.FMUs}(1).t.ss?st \longrightarrow \textbf{skip}) \\
[\!]\, i < \mathit{length}(\mathit{FMUs}) \longrightarrow \\
\quad \mu X \bullet \left(\begin{array}{l}
(\mathit{fmi2GetBooleanStatusfmi2Terminated.FMUs}(i-1)?b?st \longrightarrow X) \\
\Box \\
(\mathit{fmi2GetMaxStepSize.FMUs}(i-1)?t?st \longrightarrow X)) \\
\Box \\
(\mathit{fmi2DoStep.FMUs}(i).t.ss?st \longrightarrow \textbf{skip})
\end{array}\right) \\
[\!]\, i = \mathit{length}(\mathit{FMUs}) \longrightarrow \\
\quad \mu X \bullet \left(\begin{array}{l}
(\mathit{fmi2GetBooleanStatusfmi2Terminated.FMUs}(i-1)?b?st \longrightarrow X) \\
\Box \\
(\mathit{fmi2GetMaxStepSize.FMUs}(i-1)?t?st \longrightarrow X)) \\
\Box \\
(\mathit{stepAnalysed} \longrightarrow \textbf{skip})
\end{array}\right) \\
\textbf{fi}
\end{array}\right) \\
\ldots \\
\textbf{end}
\end{array}
$$

Figure 6: Definition of the *Step* action.

DistributeInputs uses *rinps* to set the inputs of the FMUs using the *fmi2Set* channel. *Step* proceeds with a call to *fmi2DoStep*; its definition (omitted in Fig. 5) is included in Fig. 6. Two further control events are used here: *stepToComplete* to signal that a step was initiated, and *stepAnalysed* to signal that a step has been completed. As before, FMUs are processed in the order prescribed by the *FMUs* sequence.

After the first FMU has been stepped, we enable interactions that correspond to API calls on `fmi2GetBooleanStatusfmi2Terminated` and `fmi2GetMaxStepSize`. The first one can be invoked after `fmi2DoStep` returns `fmi2Discard`, namely to check whether a slave FMU wants to terminate the co-simulation. The second one yields the maximum step size an FMU is willing to partake in.

While *Interaction* is the heart of our model of an MA, we next consider processes dealing with FMU states management and error handling.

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

Figure 7: Model of *FMUStateManager*

FMUStatesManager controls the use of the functions `fmi2GetFMUState` and `fmi2SetFMUState` for each of the FMUs. It is an interleaving of instances of the process $FMUStateManager(i)$ in Fig. 7 for all $i \in FMUs$. Once an FMU is instantiated, it is then possible to retrieve its state. After that, both gets and sets are allowed. The actual values of the state are defined in the FMUs, but recorded in the master algorithm via *fmi2GetFMUState* for later use with *fmi2SetFMUState* as defined in $FMUStateManager(i)$.

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

3.2 FMU Interfaces

The model of an FMU is simpler than the MA. It is captured by the process *FMUInterface* declared below, which takes the FMU identifier as a parameter. This process captures the control flow of an FMU, specifying, at each stage, the API functions to which it can respond. Unsurprisingly, it has some of the restrictions of a master algorithm, but it is less restrictive in that it captures just the expected capabilities of an FMU. Its purpose hence not to enforce all constraints of the standard but provide a guideline for implementations.

$$\textbf{process}\ FMUInterface \mathrel{\widehat=} i : FMUs \bullet \textbf{begin}$$

$$\textbf{state}\ State == [status : \mathbb{B}]$$
$$\ldots$$

First of all, we introduce a state component *status* to record the result of the last call to an API function.

Upon start-up of the co-simulation, the only API function that is available is `fmi2Instantiate`. The simple action below specifies this behaviour.

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

In this case, the state component *status* is updated according to the boolean b returned by *fmi2Instantiate*. If the instantiation is successful, the behaviour is described by *Instantiated*, sketched below; otherwise, it is unrestricted and specified by $RUN(FMUAPI(i))$, which allows the occurrence of any sequence of API functions, capturing that the FMU does not have to make any guarantees on its subsequent behaviour when instantiation fails.

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

- *Instantiation*

Again, if there is a fatal error, the behaviour is unrestricted. If there is no error, all functions except `fmi2Instantiate` are available, though we do not restrict their use. Finally, if there is a non-fatal error, only `fmi2FreeInstance` is possible. The main action here executes the local action *Instantation*.

While a pattern of calls is defined by a master algorithm, so that, for example, all outputs are obtained before the inputs are distributed, the FMU is passive and does not impose such a policy on its use. So, the various actions enforce precisely the restrictions in the FMI standard [1, p.105].

Although it is possible to specify a more restricted behaviour for FMUs, such a specification rules out robust FMU implementations that handle calls to the API functions that do not necessarily follow the strict pattern of a co-simulation. Such patterns are described in more detail in INTO-CPS Deliverable D2.2c.

3.3 Specific FMU Models

In the previous section, we have presented a general model for an FMU. The particular model of an FMU depends, of course, on its functionality, and must conform to a (trace) refinement of our general model. This can be proved by refinement laws for *Circus* [7].

Whereas the particular structure of the concrete FMU model is implementation dependent, we can generate a sketch of the model of an FMU using information about its structure: that is, its parameters $params_i$, inputs $cinps_i$, outputs $couts_i$, and internal state components $state_i$. FMU models in particular languages, such as Modelica and VDM-RT can be transformed into complete formal models, as explained in INTO-CPS Deliverable D2.3d.

The technique that we describe in INTO-CPS Deliverable D2.3a observes that models of FMUs can be expressed as relations on the inputs and outputs, as well as the FMU's internal state. This is also in agreement with Broman's FMI semantics [10], which uses functions in place of relations, as he restricts his attention to deterministic models only.

With the above information, we can lift a relational FMU model into a *Circus* process corresponding to its *reactive* model. Fig. 8 illustrates the shape of the resulting process. Its state includes $params_i$, $cinps_i$, $couts_i$ and $state_i$ which are fixed for $i \in FMUs$, besides the current and simulation end time.

Its structure is similar to that of the *Interaction* process used to model a master algorithm. In all cases, the interactions signal success ($fmi2OK$). If an FMU makes assumptions about its inputs, the possibility of error can be modelled as well. For example, *Instantiation* indicates success, but to explore the possibility of failure, we can define it as $fmi2Instantiate.i?b \longrightarrow \mathbf{skip}$,

$$
\begin{array}{l}
\textbf{process}\ \mathit{FMUSketch} \mathrel{\widehat=} i : \mathit{FMI2COMP} \bullet \textbf{begin} \\
\textbf{state}\ \mathit{State} == \\
\quad [\mathit{cparams}_i, \mathit{cinps}_i, \mathit{couts}_i, \mathit{cstate}_i;\ \mathit{currentTime}, \mathit{endTime} : \mathit{TIME}] \\
\mathit{Instantiation} \mathrel{\widehat=} \mathit{fmi2Instantiate}.i!\mathit{true} \longrightarrow \textbf{skip} \\
\mathit{InstantiationMode} \mathrel{\widehat=} \\
\quad \left(\begin{array}{l} \mathit{fmi2Set}.i.p_i?v!\mathit{fmi2OK} \longrightarrow \mathit{cparams}_i[p_i] := v\ ;\ \mathit{InstantiationMode} \\ \Box \\ \mathit{fmi2SetUpExperiment}.i?t_0!\mathit{true}?tN!\mathit{fmi2OK} \longrightarrow \\ \quad \left(\begin{array}{l} \mathit{currentTime}, \mathit{endTime} := t_0, tN; \\ \mathit{fmi2EnterInitializationMode}.i!\mathit{fmi2OK} \longrightarrow \textbf{skip} \end{array} \right) \end{array} \right) \\
\mathit{InitializationMode} \mathrel{\widehat=} \\
\quad \left(\begin{array}{l} \mathit{fmi2Set}.i.\mathit{inp}_i?v!\mathit{fmi2OK} \longrightarrow \mathit{cinps}_i := v\ ;\ \mathit{InitializationMode} \\ \Box \\ \mathit{fmi2ExitInitializationMode}.i!\mathit{fmi2OK} \longrightarrow \boxed{\mathit{CalculateStep}} \end{array} \right) \\
\mathit{slaveInitialized} \mathrel{\widehat=} \\
\quad \mu X \bullet \left(\begin{array}{l} \mathit{fmi2Get}.i.\mathit{out}_i!\mathit{couts}_i[\mathit{out}_i]!\mathit{fmi2OK} \longrightarrow \textbf{skip} \\ \Box \\ \mathit{fmi2Set}.i.\mathit{inp}_i?v.\mathit{fmi2OK} \longrightarrow \mathit{cinps}_i[\mathit{inp}_i] := v \\ \Box \\ \mathit{fmi2DoStep}.i?t?ss!\mathit{fmi2OK} \longrightarrow \mathit{UpdateState}) \end{array} \right) ;\ X \\
\bullet \left(\begin{array}{l} \mathit{Instantiation}\ ;\ \mathit{InstantiationMode}\ ;\ \mathit{InitializationMode}; \\ \mathit{slaveInitialized} \end{array} \right) \triangle \\
\quad \mathit{fmi2Terminate}.i!\mathit{fmi2OK} \longrightarrow \mathit{fmi2FreeInstance}.i!\mathit{fmi2OK} \longrightarrow \textbf{stop} \\
\textbf{end}
\end{array}
$$

Figure 8: Sketch of a model for a specific FMU.

which also admits the interaction $fmi2Instantiate.i.false \longrightarrow \mathbf{skip}$, in addition to $fmi2Instantiate.i.true \longrightarrow \mathbf{skip}$.

The action *CalculateStep* is obtained from the aforementioned relational FMU model. It is this action that specifies the functionality of the FMU. It corresponds to a computation that reads the values of $cparam_i$, $cinps_i$ and $cstate_i$, and sets the values of $couts_i$. It may also refer to the variables t and ss, locally introduced by the $fmi2DoStep$ communication and corresponding to the current time and step size.

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

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

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

If the state, either via setting of parameters and input or via an update, may become invalid, we can flag $fmi2Fatal$ and deadlock.

3.4 Final Considerations

We have presented a *Circus* model of FMI with focus on the behavioural and reactive aspects of master algorithms and FMUs. Our model enables us first to verify that implementations of master algorithms are conformant with the restrictions imposed by the FMI standard [1]. Secondly, it can be used to construct a concrete model of a given co-simulation, based on the relational (or, for deterministic FMUs functional) description of FMU behaviour. Such a concrete model allows us to prove properties of the particular interactions that can be observed, using the FMI API. Most interestingly, this may use the `fmi2Get` function to probe outputs during co-simulation step.

As already mentioned, our FMI *Circus* model complements the relational FMI model in INTO-CPS Deliverable D2.3a. Whereas the relational FMI model is most useful for proving universal and safety properties of co-simulations, the reactive model facilitates the design of master algorithms (which are only implicit in the relational model). It thus subsumes the relational

model, and, as explained in the previous section, we can construct specific FMUs as *Circus* processes from their relational characterisations.

In the next section, we report on our mechanisation of the *Circus* FMI model in the theorem prover Isabelle/UTP.

4 FMI Mechanisation

In this section, we present our mechanisation of the FMI *Circus* model. For this, we first discuss our embedding of *Circus* into Isabelle/UTP in Section 4.1. In Section 4.2, we explain how we formalise concrete FMI architectures. Section 4.3 then reports on our encoding of the *Circus* processes explained in the previous section. In Section 4.4, we address reasoning support. We conclude the section with some final considerations in Section 4.5.

4.1 Encoding of *Circus*

Semantically, *Circus* actions can be represented by stateful CSP processes, which we have previously integrated into Isabelle/UTP [15]. We can directly use our mechanised theory of CSP to encode *Circus* actions and processes. We next give some details on how this is done.

Circus processes are a special kind of action where the state alphabet is empty and thus corresponds to the degenerate type `unit` containing only a single value. This reflects that process state is encapsulated and internal to the process, so that it cannot be seen by an environment.

The key operator to define the semantics of a process is the following:

```
definition Process ::
  "('σ, 'φ) action ⇒
   (unit, 'φ) action" where
"Process P = state 'σ • P"
```

The function `Process` takes as its argument the main action of a process over some arbitrary state `'σ`, and yields an action over the unit state `unit`. The state hiding is carried out by the `state` operator, which changes the alphabet of a UTP action predicate to `unit`. This is achieved by first substituting the state components by some default initial values, and afterwards existentially quantifying over them. We note that the substitution step has no effect if the process initialises its state, which typically is the case.

Our *Circus* embedding also provides a treatment of local actions. Those are encoded by a nesting of Isabelle/HOL `let` expressions. We hence make use of the built-in `Let` function of HOL for our definition of the `LocalAction` constructor, as it is shown below.

```
definition LocalAction ::
  "('σ, 'φ) action ⇒
   (('σ, 'φ) action ⇒ ('σ, 'φ) action) ⇒
   ('σ, 'φ) action" where
"LocalAction = Let"
```

Generally, the signature of the `Let` function is `'a ⇒ ('a ⇒ 'b) ⇒ 'b` for arbitrary types `'a` and `'b`. The constant to be locally bound (of type `'a`) is supplied by the first argument. The second argument is a functional abstraction $(\lambda\, x \bullet A)$ in which A can refer to the local constant via the bound variable x. The semantic definition of the `Let` function is hence applying its second (function) argument to the value provided by the first.

For convenience, Isabelle/HOL provides a custom notation for terms of the form `Let` c $(\lambda\, x \bullet t)$: they are parsed and displayed as `let` x = `c in` t, so that the user does not see the application of the `Let` function. We use the same technique to support the precise syntax of *Circus* for local actions.

A third issue that begs consideration is support for recursive actions. Here, it proves sufficient for our models to support single-recursive actions only. For this, we utilise the following semantic constructor function.

```
definition RecAction ::
  "(('σ, 'φ) action ⇒
    ('σ, 'φ) action × ('σ, 'φ) action) ⇒
   ('σ, 'φ) action" where
"RecAction body =
  LocalAction (μc X • fst (body X)) (λX. snd (body X))"
```

The function `RecAction` is parametrised by a function $(\lambda\, X \bullet (A(X), B(X)))$ that maps an action X to a pair of actions $(A(X), B(X))$. The first element $A(X)$ of the pair determines the recursive shape of the action. The second element $B(X)$ allows the recursive action to be locally bound in another (successor) action. An example of this is the *Timer* process in Fig. 2 (p. 12), where the recursive *Step* action is locally bound within the main action.

We note that the $\mu_C\, X \bullet F(X)$ operator is part of the CSP embedding of Isabelle/UTP and constructs the weakest fix-point of a CSP predicate.

The three semantic constructors above enable us to define concrete *Circus* processes in our embedding. To write such processes, we, however, do not

want to use those constructors directly, as this would result in difficult to read mechanised models. Instead we define custom Isabelle syntax and term translations that mirror the process notation we use in hand-written *Circus*. By way of an example, we consider the process below which can be written directly as is for analysis by our tool. It introduces two local actions `A` and `B`, with `Act1` and `Act2` being (globally) defined elsewhere. Action `A` is recursive, while `B` invokes `A`. The main action `Main` refers to both, `A` and `B`.

```
"process P ≙
   begin
     A = Act1 ;; A and
     B = Act2 ;; A
     • Main(A, B)
end"
```

The semantic representation of the process above is as follows:

```
"P = Process
   (RecAction (λA. (Act1 ;; A,
    RecAction (λB. (Act2 ;; A, Main (A, B))))))"
```

Our parser and pretty-printer hides the latter completely from the user, and this makes our embedding convenient to use by non-Isabelle experts.

A last point of interest is how we deal with process and action parameters. Process parameters are supported easily by translating them into functional abstractions: that is, we model parametrised processes by functions from the argument type (or types, if there is more than one) to actions. We extended our parser to support the *Circus* syntax for parametric processes, too.

Parametrised actions, where they occur, can be rewritten into plain (non-parametric) actions. This is using local variables as a means to pass parameters as usual in refinement calculi. This solution works well even for non-tail-recursive actions since our model of local variables in Isabelle/UTP supports scoping: it restores the values of local variables upon exiting of their scopes.

Below is an example of a *Circus* process that is parametrised. It also illustrates a further feature of your syntax — the explicit provision of the state type of a process, via the optional `state` keyword:

```
"process P(x::nat) ≙
 begin
   state(my_state)
   A = Act1 x and
   B = Act2 ;; A
   • Main(A, B)
 end"
```

The parameter `x` here is of type `nat`, with action `A` referring to it. The process state is defined as the (HOL) type `my_state`, a record type whose definition we omit. We recall that in Isabelle/UTP, state record types are introduced using the `alphabet` rather than `record` command.

If a state type is not explicitly provided, Isabelle infers it through type checking. We note that unlike in *Circus*, we cannot impose constraints on the process state, namely by way of a *State* schema. Permissible states are hence all possible values of the underlying HOL type.

We next discuss the encoding of *Circus* operators and channels.

***Circus* operators** Most operators required for *Circus* are already available through our mechanisation of the theory of CSP [17]. Hence, we can use them out-of-the-box. Constructs that we had to additionally define are iterated sequence, iterated interleaving, and iterated parallel composition. They are all operators that fold the underlying binary CSP operator over a list of actions. We provide examples of using those operators in the next section.

***Circus* channels** In general, channels are represented by a function from the channel type `'a` to the underlying event type `'`φ. Hence, the type of a channel is `'a` $\Rightarrow$ `'`φ in HOL, and events corresponding to communications over the channel are those in the range of that function.

When we consider multiple channels, it is important that the ranges of the channel functions are disjoint and partition the event type. It is hence appropriate to introduce event types via `datatype` declarations whose constructors correspond to the declared channels. A downside of this approach is that all channels of a *Circus* model have to be declared at once, since datatypes are not extendible in the same way that, for instance, record types are.

The above is not entirely satisfactory, namely since we like to separate parts of our FMI *Circus* model into a fixed framework model that can be included *as is* and does not need modification and recompilation when new channels are introduced by the user. We next present our solution to this problem.

In general, it turns out that we require an analogue mechanism of extensible record types for datatypes. So, rather than using datatypes for events directly, we use sum types `'φ + 'ext` where `'φ` corresponds to the channels introduced by a particular *Circus* process, and `'ext` is a 'hole' (extension slot) that can be filled later on when additional channels are introduced by a subsequent process.

To give an example, we consider the channels of the *Timer* process in Fig. 2 (p. 12). The following Isabelle `datatype` introduces constructor functions for those channels and a corresponding event type `timer_event`.

```
datatype 'τ::ctime timer_event =
  setT "TIME('τ)" |
  updateSS "NZTIME('τ)" |
  step "TIME('τ) × NZTIME('τ)" |
  endc "unit"
```

We note that the datatype is parametric in the notion of time that we like to adopt (type parameter `'τ`), which gives added generality to our *Circus* encoding of the timer. We return to this point in the next section.

Importantly, we do not use the `timer_event` type directly in *Circus* processes but wrap it into a sum type with an open slot for extension. This is achieved by a prefixing operator whose Isabelle definition is presented below. It is a function applied to the type constructors of the `timer_event` datatype.

```
abbreviation timer_prefix ::
  "('a, 'τ::ctime timer_event) chan ⇒
   ('a, ('τ::ctime fmi_event) + ('τ::ctime timer_event) + 'ext) chan" where
"timer_prefix c ≡ Inr o Inl o c"
```

The resulting event type after applying the prefixing operator is the sum of three event types: one for the FMI API (`fmi_event`), one for the timer process events (`timer_event`), and an open extension type (`'ext`). The reason for inclusion of `fmi_event` is that the *Timer* process itself extends channels that are introduced prior to its declaration, namely for the FMI API.

For application of the `timer_prefix` operator, we introduce the syntax `tm:⋯`. This means, instead of writing `setT`, `updateSS`, and so on, we now write `tm:setT`, `tm:updateSS`, and so on, which carries out the prefixing and wrapping of the process event type into a suitable sum type.

Our approach provides an elegant solution for the incremental introduction of channel sets in mechanised *Circus* models, and recovers some modularity here. It also makes a case for a future version of Isabelle to incorporate extensible datatypes into the proof system.

```
consts
  FMUs :: "FMI2COMP list"
  parameters :: "(FMI2COMP × VAR × VAL) list"
  initialValues :: "(FMI2COMP × VAR × VAL) list"
  inputs :: "port list"
  outputs :: "port list"
  pdg :: "port ⇒ (port list)" -- ‹Port Dependency Graph›
  idd :: "port ⇒ (port list)" -- ‹Internal Direct Dependencies›
```

Figure 9: Isabelle/HOL mechanisation of an FMI architecture.

This concludes the description our *Circus* embedding into Isabelle/UTP. Our mechanised notation is almost identical to hand-written *Circus*, which simplifies its use by Isabelle non-experts. For proof support, we have included additional laws to reason about CSP processes that encapsulate their state (`state` construct above). We also have introduced a simplification set to unfold all definitions of the *Circus* process semantics, which is useful in proofs. The proof tactics for CSP are, in this way, directly applicable to *Circus*.

4.2 Model Configuration

Fig. 9 illustrates how we encode FMI configurations (as introduced in Section 3.1) in Isabelle/HOL. We represent FMUs identifiers as elements of a given type `FMI2COMP`, which we introduce abstractly by virtue of a HOL type declaration. Concrete identifiers for FMUs are then declared using an Isabelle `axiomatization`, along with assumptions that they are distinct — an example of this follows in Section 5.

The abstract constant `FMUs` in Fig. 9 yields a sequence of all FMUs of the co-simulation that determines the order in which FMUs are processed by master algorithms. Sequences in HOL are represented by objects of type `'a list`, where `'a` is the underlying element type. For well-formedness, the `FMUs` sequence has to be injective and include all values of the `FMI2COMP` type, so that our iterative operators process all FMUs. Proving this is the concern of the technique presented in INTO-CPS Deliverable D2.3a.

Ports are represented by pairs of type `FMI2COMP` × `VAR`, where `VAR` encodes variables as name/type pairs. We reuse the `VAR` (and `VAL`) types from the axiomatic value model proposed in [30], as this renders our model inherently extensible to support arbitrary HOL types for ports. An Isabelle `type_synonym port` is declared. Thus, `inputs` and `outputs` are of type `port list`.

Model parameters and initial values are recorded by the `parameter` and `initialValues` lists, with the following caveats:

1. For $fmu :: \texttt{FMI2COMP}$ and $v :: \texttt{VAR}$, there is at most one element (fmu, v, x) in the range of the sequence `parameters`;
2. For each input port (fmu, v) in the range of `inputs`, there is precisely one element (fmu, v, x) in `initialValues`.

In addition, we require that the value x (of type `VAL`) agrees with the type of the variable v. Like the earlier constraints, (1) and (2) above can be checked automatically within the Isabelle proof system; INTO-CPS Deliverable D2.3a explains how we automate those proofs with tactics.

We lastly look at the encoding of the constant `pdg`, which defines the control diagram of the co-simulation architecture. As explained in Section 3.1, connections between FMUs are encoded by the function `pdg`, which maps output ports to their connected input ports. In Isabelle, we encode it as a function from `port` to `port list`. We use a `list` rather than `set` in order to easily iterate over the result of applying the function to some output, as required by the `TakeOuputs` action of the *Interaction* process in Fig. 5 (p. 19); our list-based encoding moreover ensures finiteness of the connected inputs.

We conclude by noting that the concrete values for the constants in Fig. 9 can be inferred from the INTO-SysML model of a co-simulation, that is, its Architecture Structure Diagram (ASD) and Connection Diagram (CD). Fundamentally, this process can be automated. Assuming that we have shown well-formedness of the mechanised FMI architecture, using the technique in INTO-CPS Deliverable D2.3a, the next section examines the encoding of the underlying *Circus* processes.

4.3 *Circus* Processes

A mechanised version of the *Timer* process (Fig. 2 on page 12) is provided in Fig. 10. Its state is given by the HOL type `time_state` and introduced by the following alphabet declaration.

```
alphabet 'τ::ctime timer_state =
  currentTime :: "TIME('τ)"
  stepSize :: "NZTIME('τ)"
```

The `alphabet` command is part of Isabelle/UTP and an extension of Isabelle's `record` command for introducing record types. An additional behaviour of

```
definition
"process Timer(ct::TIME('τ), hc::NZTIME('τ), tN::TIME('τ)) ≙ begin
  state('τ timer_state)
  Step = (
    (tm:setT?(t : «t ≤ tN») → currentTime :=c «t») □
    (tm:updateSS?(ss) → stepSize :=c «ss») □
    (tm:step!(&currentTime)!(&stepSize) →
      currentTime :=c minu(&currentTime + §(&stepSize), «tN»)) □
    (&currentTime =u «tN») &u tm:endc → Stop) ;; Step
  • (currentTime, stepSize) :=c («ct», «hc») ;; Step
end"
```

Figure 10: Mechanised Timer process in Isabelle/UTP.

`alphabet`is to define lenses for the state components, so that they can be referenced within Isabelle/UTP predicates and relations; details of this are technical and can be found in INTO-CPS Deliverable D2.3b, as well as [15].

Two state components are introduced: `currentTime` and `stepSize`. An added feature of the mechanised model is that we represent time abstractly by virtue of an arbitrary type $'\tau$ that fulfils certain type-class membership constraints. The constructs `TIME(_)` and `NZTIME(_)` above impose those constraints on $'\tau$. They guarantee, for instance, that there exists a linear order on the elements of the time domain. They also require the existence of various arithmetic operators like $+$, $-$, $*$, and so on, with common algebraic properties. Natural, rational and real numbers are valid time domains, for instance.

We observe that the Isabelle encoding of the *Timer* process is a nearly direct image of the respective *Circus* process. There are only a few technical artefacts that we have to remember.

Firstly, when referring state components in *Circus* operators, it is usually necessary to decorate them, depending whether the underlying term is a plain predicate or relation. For instance, the occurrence of `currentTime` within output synchronisations requires an `&_` prefix since the expression is over unprimed variables only. The same applies to the right-hand side of an assignment. Secondly, where HOL values and terms are referenced, we require the double-angular brackets «_». This emphasises the difference between HOL terms and UTP terms, as the brackets here act as a coercion from the former to the latter.

As can be seen, events are prefixed by '`tm:`', which, as explained in the previous section, is necessary to facilitate later extension of the action event type when additional channels have to be declared. *Circus* constructs are

```
InstantiationMode =
  (;;c (i, x, v) : parameters •
    fmi:fmi2Set!(«i»)!(«x»)!(«v»)?(st) → Skip) ;;
  (;;c i : FMUs •
    fmi:fmi2SetUpExperiment!(«i»)!(«startTime»)
      !(«stopTimeDefined»)!(«stopTime»)?(st) → Skip) ;;
  (;;c i : FMUs •
    fmi:fmi2EnterInitializationMode.(«i»)?(st) → Skip) and
```

Figure 11: Encoded `InstantiationMode` action of the *Interaction* process.

subscripted to delineate to what UTP theory they belong. For instance, equality $t_1 =_{\mathtt{u}} t_2$ is subscripted with a '`u`' since this operator is valid in all theories. Fixed-point and assignment, on the other hand, are subscripted with a '`C`' as we use versions that are defined and valid only in the theory of CSP and *Circus*. The subscripting avoids ambiguities between UTP and HOL operators.

A more elaborate process is the *Interaction* process in Fig. 5 (p. 19), capturing the interactions of a master algorithm. Fig. 11 recaptures the encoding of the local action *InstantiationMode* of that process. We observe the use of iterated sequence to set the parameters of FMUs (`fmi2Set`), set up the co-simulation experiment (`fmi2SetUpExperiment`), and, lastly, enter initialization mode (`fmi2EnterInitializationMode`). In Fig. 11, `startTime`, `stopTimeDefined` and `stopTime` are global deferred constants of the model that can be given concrete values by particular co-simulation instances. The use of iterated *Circus* operators is very common in our model and, as mentioned before, facilitated by the use of lists rather than sets.

The encoding of actions for reading and recording outputs of FMUs, and distributing them between co-simulation steps is in Fig. 12. We make use of the state components `rinps` of the process here, whose mechanised type is a HOL partial function from `port` to `VAL`. Isabelle/UTP provides a generic update operator that enables us to set and access the elements of various map-like structures, including partial functions. Less the subscripted parentheses $(\_)_{\mathtt{a}}$, which invoke the generic application and update operator of Isabelle/UTP rather than HOL, this makes the encoding look almost exactly like our mathematical formalisation.

We recall that `pdg` in Fig. 12 is a global constant that determines the port-dependency relationships between the input and output ports of FMUs. The encoding of the *Interaction* process is, however, independent of the particular

```
TakeOutputs = rinps :=C «empty» ;;
  (;;C out : outputs •
    fmi:fmi2Get.(«FMU out»).(«name out»)?(v)?(st) →
      (;;C inp : pdg out • rinps :=C &rinps(«inp» ↦ «v»)u)) and

DistributeInputs = (;;C inp : inputs •
  fmi:fmi2Set.(«FMU inp»).(«name inp»)!(&rinps(«inp»)a)?(st) → Skip) and
```

Figure 12: `TakeOutputs` and `DistributeInputs` actions.

value of `pdg`, which is usually provided by a concrete model of a co-simulation, including master algorithm and FMUs.

An issue that we faced in the encoding of the *Interaction* process was the presence of mutual recursion in our earlier CSP model in INTO-CPS Deliverable D2.2b. This was between the `slaveInitialized` and `NextStep` local actions. We managed to break the recursion by rewriting of the model while preserving its semantics. Such rewrites can be validated with *Circus* laws and provide additional confidence that we are not changing the semantics. A similar issue arose in the encoding of the `ErrorMonitor` action, where we broke the recursion by encoding one of the local actions as a higher-order HOL function.

To summarise, we use our embedding of *Circus* to encode the reactive FMI semantic model in Section 3. While in some cases we rewrite the model to fit our restrictions that only single recursion and parameterless actions are supported, such can be subject to verification in Isabelle/HOL, too. A key achievement is that our mechanised model is very close to the syntax of the original *Circus* model and more general in supporting arbitrary time domains, such as, discrete, continuous and super-dense time.

4.4 Reasoning Support

To reason about *Circus* models, three complementary strategies are available. The first one is via algebraic refinement laws. Such laws can be proved within the Isabelle/UTP CSP semantics and enable us to transform mechanised *Circus* models in a piecewise and stepwise manner. For proofs of the laws, we have implemented automatic tactics that transform a conjecture about processes and actions within Isabelle/UTP into a pure HOL predicate, eliminating all artefacts of our UTP predicate models, such as lenses and state spaces.

Those tactics are called `rel_simp`, `rel_auto` and `rel_blast` and are defined in a separate theory of Isabelle/UTP. They use principles of transfer [21] and interpretation [14]. Many refinement laws about CSP can be proved in this manner, including basic laws for communication prefixes, external choice, recursion and parallel composition.

Once a refinement law is proved, it can be used to rewrite part of a *Circus* model. A refinement of the overall *Circus* specification in which the part is embedded is obtained exploiting monotonicity of the language constructs. We have proved various monotonicity laws. The approach described in [31] can be used to automate their application, using a tactic language as well as window inference techniques.

We note that refinement is also applicable to prove, for instance, that a trace $tr = \langle e_1, e_2, \cdots, e_n \rangle$ is exhibited by a *Circus* model. This is by turning the trace into a process $T \mathrel{\widehat{=}} e_1 \longrightarrow e_2 \longrightarrow \cdots \longrightarrow e_n$ and then composing that process in parallel with the *Circus* model, synchronising on all events in the trace. The parallel composition has to be refined by T itself, which can be checed using step laws for communication prefixes.

A second strategy is using a contractual view of reactive computations. As explained in Section 2.3, CSP processes can be characterised by way of reactive contracts, specifying a precondition, pericondition and postcondition. As formalised by **Theorem 5.6** in INTO-CPS Deliverable D2.3b, a refinement $S \sqsubseteq T$ of reactive computations can be reduced to a proof of three predicates involving $\mathbf{pre}(\_)$, $\mathbf{peri}(\_)$, $\mathbf{post}(\_)$ of S and T:

$$S \sqsubseteq T \quad \text{provided that} \quad \left(\begin{array}{l} \mathbf{pre}(S) \Rightarrow \mathbf{pre}(T) \wedge \\ \mathbf{pre}(S) \wedge \mathbf{peri}(T) \Rightarrow \mathbf{peri}(S) \wedge \\ \mathbf{pre}(S) \wedge \mathbf{post}(T) \Rightarrow \mathbf{post}(S) \end{array} \right)$$

Those predicates are easier to prove than the initial refinement conjecture on the left-hand side, and we have proved several distribution laws that enable us to evaluate $\mathbf{pre}(\_)$, $\mathbf{peri}(\_)$ and $\mathbf{post}(\_)$ for arbitrary action terms, as well as the process semantic constructors.

The contractual techniques allows us to specify, for instance, properties of the permissible histories of interactions of a *Circus* process, and then show that a concrete process exhibits those histories. It is a more powerful technique for automation than the aforementioned `rel_[simp|auto|blast]` tactics.

A third strategy is to translate the *Circus* model into CSP and use a model checking technique, as described in the previous deliverable INTO-CPS Deliverable D2.2d. This works well as long as state spaces are fairly compact.

On the other hand, the proof-based approaches work irrespective of the size of the state-space, which we abstract from in the refinement laws and reactive contracts.

4.5 Final Considerations

In this section, we have discussed both our mechanisation of *Circus* in Isabelle/UTP and the encoding of the FMI *Circus* semantics. These are key in developing reasoning techniques for FMI co-simulation models. Very importantly, we have also related our work here to the modelling and proof approach described in INTO-CPS Deliverable D2.3a. The *Circus* model can be seen as an extension of the abstract model presented there, capturing faithfully the FMI API and restrictions on FMUs and master algorithms.

Our reasoning concern, however, is different here from that in INTO-CPS Deliverable D2.3a — it is not about showing properties of FMUs but refining their abstract models into implementations from which we can, for instance, generate code. This is achieved by the reasoning approaches we summarised; similar techniques have already been used in other areas, involving, for instance, implementations of control laws [6].

In the next section, we examine a case study and its encoding using the mechanised model we described in this section.

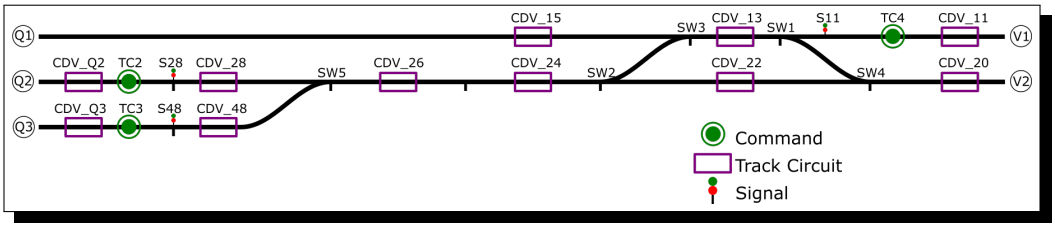

Figure 13: Railway interlocking layout of our case study.

5 Instantiation and Reasoning

In this section, we present our approach to using the FMI *Circus* model and its mechanisation. We first describe a railways case study as a running example in Section 5.1. Section 5.2 considers instantiation of the architecture, and Section 5.3 examines models of FMUs. In Section 5.4, we discuss *Circus* processes of the reactive model, and, lastly, in Section 5.5 report on analysis and proof techniques. We conclude in Section 5.6 with a few final considerations.

We note that the case study description (Section 5.1) is a summary of the case study used in INTO-CPS Deliverable D2.3a. Also, the instantiation of the FMI architecture (Section 5.2) is similar, although it is done for a complementary reason here, being an integral part of the *Circus* model. Whereas INTO-CPS Deliverable D2.3a considers *abstract* models of FMUs as relational computations, we here consider their refinement into *concrete* models, which is the contribution of Section 5.3. Lastly, Section 5.4 extends on our previous work in INTO-CPS Deliverable D2.2d by illustrating a concrete instantiation of a master algorithm and co-simulation, and our reasoning strategies in Section 5.5 are novel contributions, too.

5.1 A Railways Case Study

Our case study here considers part of a tramway station whose railway layout is presented in the diagram of Fig. 13. Trains enter the interlocking at the points V1, Q2 and Q3, and then issue a telecommand to request a route through the interlocking. Telecommand stations are denoted by the three green dots, and the possible routes for trains are $\mathsf{V1} \rightarrow \mathsf{Q1}$, $\mathsf{V1} \rightarrow \mathsf{Q2}$, $\mathsf{V1} \rightarrow \mathsf{Q3}$, $\mathsf{Q2} \rightarrow \mathsf{V2}$ and $\mathsf{Q3} \rightarrow \mathsf{V2}$. We consider co-simulation scenarios where two trains arrive at different entry points and request a route.

Access to the interlocking is controlled by three signals S11, S28 and S48. They are initially red, causing arriving trains to stop and wait on the tracks CDV_11, CDV_Q2 and CDV_Q3. When a telecommand is issued by one of the trains, the control logic of the interlocking allocates a free route, if available, sets railway switches accordingly, and then gives the respective train a green light to go ahead. Two trains are allowed to proceed simultaneously only if their routes do not intersect. This guarantees that no collision can occur due to more than one train passing through the same track segment. The correct setting of railway switches (SW1-5) additionally ensures that trains move on their allotted paths and do not derail.

The interlocking software controller is in essence an automaton implemented as a cyclic control loop in VDM-RT. The inputs of the interlocking controller are boolean vectors for the CDV and telecommand. The CDV (*chemin de voie* in French) is a bit vector of size 13 whose elements register the presence of a train on a particular track segment. Telecommand requests are likewise encoded by a bit vector TC (of size 4), in which each bit roughly corresponds to a particular route request. Since the FMI 2.0 standard [1] does not support vectors (of any type), we encode them as integer numbers when they are transmitted as data between FMU components in our model.

Outputs (actuators) of the interlocking are signals and track point switches that control the paths of trains as they move through the interlocking. We note that the interlocking controller does not see the TC of each train individually, but the combined signal from all three telecommand stations. The same is true for the CDV, which is a combined signal of all tracks. Next, we describe our FMI co-simulation model of this system.

5.1.1 System Overview

A high-level view of the system as a co-simulation is presented by the INTO-SysML connection diagram (CD) in Fig. 14. Altogether, there are four FMUs. Train1 and Train2 simulate the physical behaviour of both trains, which includes the action of the train driver controlling the speed of the trains. A third FMU Interlocking encapsulates the physical plant and the software that controls it. Lastly, we require an additional FMU CDV/TC Merger to generate the CDV signal from the trains' locations and merge their telecommands into a single vector. A supplementary function of CDV/TC Merger is to produce monitoring signals (testing probes) for collision and derailment.

The initial models for this case study, as described in detail by INTO-CPS De-

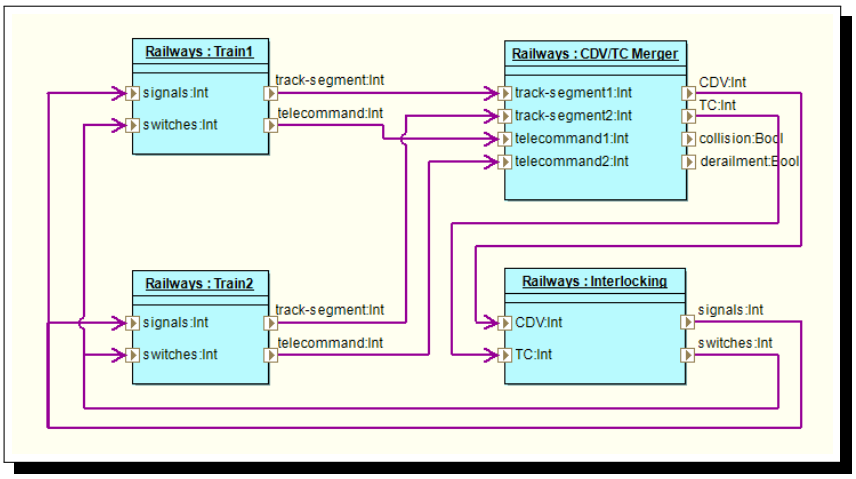

Figure 14: FMI co-simulation architecture for the railways system.

liverable D1.2b, define the train physics and their control behaviour as bond graphs in 20-sim. The VDM-RT model of the interlocking was automatically generated from the RLL[2] production code of the software controller.

For the train behaviour, we consider traction and braking actions but do not model train mass and gravity, and neither smooth acceleration and braking curves (jerk). This is justified because the influence of those factors does not alter the fundamental system dynamics. The original 20-sim train model has been encoded in Modelica, for which we have a formal semantics mechanised in Isabelle/UTP [13]; for details of that semantics we refer to the INTO-CPS Deliverable D2.3b.

We have extracted the core algorithm for setting relays, signals and switches from the interlocking software controller. This discards aspects related to driving relay actuators, since we encode relays in software rather than modelling them as hardware devices.

5.1.2 Behavioural Models

We next describe the Modelica train model and the VDM-RT Interlocking model; we omit the straightforward CDV/TC Merger model, but details of its encoding can be found in INTO-CPS Deliverable D2.3a.

[2]Relay Ladder Logic, commonly used for describing hardware controller.

```
/* Physical movement of the train. */
equation
  der(current_speed) = acceleration;
  der(position_on_track) = current_speed;

/* Control equation for acceleration and braking. */
equation
  when abs(current_speed - setpoint_speed) < 0.001 then
    /* This case corresponds to engaging the brakes. */
    reinit(current_speed, setpoint_speed);
    reinit(acceleration, 0);
  end when;
  if current_speed == setpoint_speed then
    /* To avoid chattering during simulation. */
    acceleration = 0;
  else
    if current_speed < setpoint_speed then
      /* Accelerate */
      acceleration = normal_acceleration;
    else
      /* Decelerate */
      acceleration = normal_deceleration;
    end if;
  end if;
```

Figure 15: Train control equations in Modelica.

Modelica Train Models Kinematics and speed control of both trains is encoded by the Modelica equations in Fig. 15. The first `equation` block captures motion: acceleration is the derivative (operator `der(_)`) of the train's velocity, given by the `current_speed` variable, and velocity the derivative of its relative position on the track on which it is currently located, given by the `position_on_track` variable.

The second `equation` block realises a control algorithm: acceleration is set to either zero, `normal_acceleration` or `normal_deceleration`, depending on whether the current speed is equal, below or above the set-point speed of the train, set by the driver. The latter two are suitable constants of the model. A special case is added by the `when` clause that simultaneously sets the train speed to the set-point speed and acceleration to zero if we are close to the set-point speed.

The behaviour of the train driver is captured by the following equation:

```
equation
  setpoint_speed = CalculateSpeed(track_segment, signals, max_speed);
```

The computation is carried out by the function `CalculateSpeed`, which expects the current track segment (`current_track`), signal values (`signals`), and maximum permissible speed (`max_speed`) as arguments. It then sets the set-point speed (`setpoint_speed`) to `max_speed` if there is either a green light or no signal on the track; otherwise, it sets it to zero. A Modelica

model `Topology` records (static) constants that define the railway topology, such as signal positions and track connections.

The encapsulation of algorithmic behaviours into Modelica `functions` such as `CalculateSpeed`, where possible, is deliberate. Our formal encoding later on profits from this as those functions can be naturally translated into HOL functions within the Isabelle proof system. This kind of engineering facilitates formal analysis and has a modularising ripple-on effect on proofs.

A last aspect of the train model we consider is the equations for the discontinuous variable changes that occur when a train crosses one track and enters the next. The Modelica equations for this are recaptured below.

```
equation
  when position_on_track > pre(track_length) then
    track_segment = NextTrack(pre(track_segment), pre(switches), direction);
    reinit(position_on_track, 0);
  end when;

equation track_length =
  (if track_segment > 0 then track_length_tab[track_segment] else 0);
```

The `NextTrack()` function calculates the next track segment when the train's relative position on the current track, given by the `position_on_track` variable, reaches the `track_length`. The function requires the current track, state of track points (`switches`), and travel direction as inputs, and its output is equated with the newly entered track segment after the discontinuity. Simultaneously, it also reinitialises `position_on_track` back to zero. The use of `pre(_)` statements is to refer to the system state before the discontinuity, as otherwise the equation would be self-contradictory.

VDM-RT Interlocking Model The VDM-RT interlocking controller is in essence a finite automaton whose state is determined by the configuration of five relays R1-R5, each corresponding to a particular route being enabled.

To capture the core algorithmic behaviour of the interlocking, we introduce a variable Relay to record the state of relay switches as a boolean vector. The interlocking software controller is then modelled by virtue of a cyclic executive that periodically performs the following four sequential tasks:

1. Activate (lock) routes requested by a telecommand.
2. Deactivate routes once a train has passed through them.
3. Set railways switches consistently with the enabled routes.
4. Set signals consistently with the enabled routes.

```
-- Relay Activation
if hwi.TC(4) and not hwi.TC(3) and
   not Relay(2) and not Relay(3) and
   hwi.CDV(4) and hwi.CDV(5)
then Relay(1) := true;

if hwi.TC(3) and not hwi.TC(4) and
   not Relay(1) and not Relay(3) and not Relay(4) and not Relay(5) and
   hwi.CDV(4) and hwi.CDV(8) and hwi.CDV(9) and hwi.CDV(10) and hwi.CDV(1)
then Relay(2) := true;
```

Figure 16: Extract from the VDM-RT algorithm for locking routes.

The sequential program logic that performs the locking of routes (task 1) is included in Fig. 16. We note that `hwi` is a VDM-RT object that provides the hardware interface (inputs and outputs) of the controller.

For locking (1) to occur, a telecommand must have been issued that requests the respective route; this is achieved by the condition on the bit vector TC that cumulatively records the telecommands recorded by all three telecommand stations. The constraints on Relay ensure that locked routes are non-intersecting, so that trains can pass without crossing each others' paths. Lastly, we have additional constraints on the CDV signal that ensure that the track segments of the route to be locked are not still occupied by a previous train.

Once a train has traversed a route, the respective relay has to be reset to give other trains the opportunity to pass (task 2). This is achieved by a similar program that performs assignments `Relay(i) := false` under suitable conditions, namely when relay `i` is activated and the last track of the respective root becomes vacant.

Tasks (3) and (4) deal with the positioning of railway switches and the setting of signals. Their VDM-RT implementation hence assigns the variables `Switch` and `Signals`, which correspond to outputs of the FMU. We shall not discuss the implementation of those tasks in detail but refer to Appendix A that contains the complete VDM-RT program model.

While our software model retains the core logic of the hardware realisation, it does not consider time delays incurred by the latency of relay and point actuators. Although those delays can potentially impact on safety analysis, refining our models to incorporate them would be a straightforward extension and not crucial to illustrate our technique.

```
axiomatization
  train1 :: "FMI2COMP" and
  train2 :: "FMI2COMP" and
  merger :: "FMI2COMP" and
  interlocking :: "FMI2COMP" where
  fmus_distinct: "distinct [train1, train2, merger, interlocking]" and
  FMI2COMP_def: "FMI2COMP = {train1, train2, merger, interlocking}"
```

Figure 17: Instantiation of the railways FMI architecture in Isabelle/HOL.

5.2 Architecture Instantiation

Before we consider the relational and *Circus* model of FMUs, we instantiate the FMI architecture by assigning concrete values to the abstract constants in Fig. 9 (p. 33). As previously noted, this instantiation is also discussed in INTO-CPS Deliverable D2.3a, where it is serves to establish well-formedness of an architecture. In this deliverable, it is a key ingredient for defining the concrete behaviour of a master algorithm. We hence summarise it, focusing on aspects relevant to the work here.

To instantiate the abstract type `FMI2COMP`, we make use of an Isabelle/HOL `axiomatization` that introduces concrete symbolic constants for each of the FMUs. In this case, we call them `train1`, `train2`, `merger` and `interlocking`. Fig. 17 includes the corresponding mechanisation fragment.

We note that Isabelle `axiomatizations` can potentially cause logical inconsistencies in HOL theories. Here, however, the particular shape of the assumptions, being that of an enumerated type, guarantees consistency.

The INTO-SysML diagram (Fig. 14) is encoded by providing definitions for the seven constants: `FMUs`, `parameters`, `initialValues`, `inputs`, `outputs`, `pdg` and `idd`. We note that those definitions, unlike the previous one, are conservative — no user-level axioms are hence required to formulate them.

As mentioned earlier on, `FMUs` is a sequence of all FMUs, hence we require its range to include all the values of the `FMI2COMP` type. Below we recapture its definition for the railways co-simulation system.

```
overloading
  railways_FMUs ≡ "FMUs :: FMI2COMP list"
begin
  definition railways_FMUs :: "FMI2COMP list" where
  "railways_FMUs = [train1, train2, merger, interlocking]"
end
```

```
overloading
  railways_parameters ≡ "parameters :: (FMI2COMP × VAR × VAL) list"
begin
  definition railways_parameters :: "(FMI2COMP × VAR × VAL) list" where
  "railways_parameters = [
    (train1, $max_speed:{fmi2Real}u, InjU (4.16::real)),
    (train2, $max_speed:{fmi2Real}u, InjU (4.16::real)),
    (train1, $fixed_route:{fmi2Integer}u, InjU V1Q2),
    (train2, $fixed_route:{fmi2Integer}u, InjU Q3V2)]"
end
```

Figure 18: Parameter definition of the railways instantiation.

We recall that simulation `parameters` are of type `FMI2COMP` × `VAR` × `VAL`. Concrete parameters of the railways co-simulation are the routes of the trains and their maximum speed. Fig. 18 illustrates parameter instantiation in Isabelle. The function `InjU` is used to construct a value of type `VAL` from some arbitrary HOL value; it is part of our universal value model [30] and allows us to encode arbitrary HOL values as elements of a universal type `VAL`. Our syntax for constructing variables is $\$name{:}\{type\}$. The types `fmi2Integer` and `fmi2Real` are synonyms for the HOL types `int` and `real`, encoding integers and real numbers, respectively. Indeed, we introduce such synonyms for all permissible FMI port types *as per* the FMI standard 2.0 [1].

Next, we consider the input and output ports of the control diagram in Fig. 14 on page 42. We recall that ports are modelled by pairs of type `FMI2COMP` × `VAR`.

An extract of the definition of `inputs` is recaptured below.

```
overloading
  railways_inputs ≡ "inputs :: (FMI2COMP × VAR) list"
begin
  definition railways_inputs :: "(FMI2COMP × VAR) list" where
  "railways_inputs = [
    (train1, $signals:{fmi2Integer}u),
    (train1, $switches:{fmi2Integer}u),
    (train2, $signals:{fmi2Integer}u),
    (train2, $switches:{fmi2Integer}u), ...]"
end
```

For brevity, we only list the input ports of `train1` and `train2` and omit those of `merger` and `interlocking`. The encoding of `outputs` is similar.

For each input port in the list `inputs`, an initial value has to be provided. We extract such values from the Modelica train model and VDM-RT interlocking

```
definition pdg :: "(PORT × PORT) list" where
"pdg = [
  ((train1, $track_segment:{fmi2Integer}ᵤ), (merger, $track_segment1:{fmi2Integer}ᵤ)),
  ((train2, $track_segment:{fmi2Integer}ᵤ), (merger, $track_segment2:{fmi2Integer}ᵤ)),
  ((train1, $telecommand:{fmi2Integer}ᵤ), (merger, $telecommand1:{fmi2Integer}ᵤ)),
  ((train2, $telecommand:{fmi2Integer}ᵤ), (merger, $telecommand2:{fmi2Integer}ᵤ)), ...]"
```

Figure 19: Extract of the definition of port dependency graph (`pdg`).

model, and encode them via `initialValues`, namely as a list of elements of type `port` × `VAL` where `port` is synonymous for `FMI2COMP` × `VAR`. For instance, the initial value for `signals` is [`False`, `False`, `False`], capturing that all signals show initially red. Since the Isabelle definition follows the same schema as in Fig. 18, we omit it here referring to Appendix B, which includes the complete mechanised model of the architecture.

Lastly, we consider the encoding of FMU port connections and internal dependencies, recorded by the constants `pdg` and `idd`. While their type in our mechanised model is that of a HOL function that maps outputs to their connected (or dependent) inputs (see Fig. 9), we represent these functions by their finite graph: that is, as a relation of type (`port` × `port`) `list`. The reason for this is that it simplifies (inductive) proofs of associated well-formedness properties. We note that such a graph-based model can be easily converted into a pure function, as described in INTO-CPS Deliverable D2.3a. While functional representations are more useful in the behavioural model of master algorithms (see INTO-CPS Deliverable D2.3a), from here on we assume that `pdg` and `idd` are (finite) relations, encoded as lists of maplets.

An extract of the definition of the constant `pdg` for the railways architecture is presented in Fig. 19. The pairs included in the list on the right-hand side account for the connection of the Train1 and Train2 FMUs with the CVD/TC Merger FMU. Other connections are omitted for brevity. The construction of `pdg` can be automated from the Connection Diagram of the INTO-SysML model and hence does not require expertise in formal modelling.

Since the definition of `idd` is not relevant for the *Circus* model, we refer to INTO-CPS Deliverable D2.3a for it. That deliverable also discusses how well-formedness constraints on the instantiated constants are mechanically discharged, using automatic tactics and reasoning tools.

5.3 FMU Models

In this section, we discuss the relational behavioural model of FMUs. They correspond to the data operation *CalculateStep* in the sketch of a model for a specific FMU (Fig. 8 on page 26). Once we have defined the relational FMU models, we use the sketch process to turn them into reactive FMU models. Based on the information about FMUs including parameters, inputs, outputs, and state, this lifting can be done in a uniform manner and does not require human guidance and expertise in formal modelling. Though currently the lifting and construction of the sketch process is done manually, we outline opportunities for its automation near the end of the section.

We recall that the models we consider here are *concrete* relational descriptions of FMUs used in implementations, and hence not the same as those in INTO-CPS Deliverable D2.3a, which deal with *abstract* descriptions of FMUs. Our technique requires showing that the former refine the latter.

5.3.1 Modelica Train FMUs

The Modelica train model is formalised in the Hybrid Relational Calculus (HRC) [18], which we have semantically embedded into Isabelle/UTP. Details of that embedding can be found in INTO-CPS Deliverable D2.3b. Below, we present the model that focusses on the situation when the train is stopping due to an approaching red signal. The full train behaviour can be encoded in a similar way. We formalise this situation using shorter variable names *acc*, *vel* and *pos* for `acceleration`, `current-speed` and `position-on-track` in Fig. 15. We note that *normal-deceleration* below is negative and determines the rate at which the train reduces its speed as a result of braking forces being applied.

$$BrakingTrain \mathrel{\widehat{=}} \left(\begin{array}{l} acc := normal\text{-}deceleration \,; \\ vel := max\text{-}speed \,; \\ pos := 0 \,; \\ \left\langle \begin{pmatrix} \dot{acc} \\ \dot{vel} \\ \dot{pos} \end{pmatrix} = \begin{pmatrix} 0 \\ acc \\ vel \end{pmatrix} \right\rangle \textbf{until}_h \ (vel \leq 0) \,; \\ acc := 0 \end{array} \right)$$

We first assign initial values to the continuous variables, and this effectively creates initial conditions for the ODE that describes the train motion. We

then evolve the continuous variables, according to the ODE, until the velocity reaches 0. This is achieved by the $\langle \dot{x} = f(t, x) \rangle$ operator of the HRC (a definition if this operator can be found in INTO-CPS Deliverable D2.2c). The P **until**$_h$ b construct is a pre-emption operator that interrupts the continuous evolution P once the condition b is met. After this, we set the acceleration to 0, so that the train halts and does not start moving backwards.

The above hybrid relation encodes the kinetic and control equations in the diagram of Fig. 15, albeit only considering deceleration. For the complete train model, we require an additional variable for the set-point speed and equations for calculating it from the signal vector.

Below, we show how the braking-train model is formalised in Isabelle/UTP:

```
"BrakingTrain =
   (c:accel, c:vel, c:pos) :=ᵣ («normal_deceleration», «max_speed», «0») ;;
   ⟨{&accel, &vel, &pos} • train_ode(ti)⟩ₕ untilₕ ($vel´ ≤ᵤ 0) ;; c:accel :=ᵣ 0"
```

The variable `c` constitutes the state space of the model, being a product of three continuous variables of type `real`. They are likewise introduced using the `alphabet` command discussed earlier. The differential equation of the train is captured by the constant `train_ode`, which we define separately.

The encoding above enables us to verify that trains are able to halt in time when approaching a track that is protected by a red signal. A sketch of that mechanised proof is in Fig. 20. The proof is formulated as a refinement conjecture whose left-hand side specifies that the acceleration (`accel′`) eventually becomes 0 while the position of the train travelled (`pos′`) is bound by the length of the track, here 44 meters.

Proofs like the above are important building blocks for showing that our concrete (continuous) train model refines the abstract (discrete) train model in INTO-CPS Deliverable D2.3a.

A last step is that we have to lift the continuous train model into a relational computation. The general approach for this is described in INTO-CPS Deliverable D2.3d on linking theories and based on extracting a before-after state predicate for a given simulation start and end time.

We conclude by noting that our encoding utilises the Multivariate Analysis package [22] of Isabelle, which provides a precise encoding of real numbers as Cauchy sequences and several operators from the integral and differential calculus, as well as proof support. We use that package to encode ordinary differential equations (ODEs) in the hybrid relational calculus.

```
theorem braking_train_pos_le:
 "($c:accel´ =u 0 ∧ ⌈$pos´ <u 44⌉h) ⊑ BrakingTrain" (is "?lhs ⊑ ?rhs")
proof -
  -- {* Solve ODE, replacing it with an explicit solution: @{term train_sol}. *}
  have "?rhs =
    c:accel, c:vel, c:pos := «-1.4», «4.16», «0» ;;
    {&accel,&vel,&pos} ←h «train_sol»(&accel,&vel,&pos)a(«time»)a until h ($vel´ ≤u 0) ;;
    c:accel := 0"
  by (simp only: BrakingTrain_def train_sol)
  -- {* Set up initial values for the ODE solution using assigned variables. *}
  also have "... =
    {&accel,&vel,&pos} ←h «train_sol(-1.4,4.16,0)(time)» until h ($vel´ ≤u 0) ;; c:accel := 0"
  by (simp add: assigns_r_comp usubst unrest alpha, literalise, simp)
```

Figure 20: The braking train scenario encoded in Isabelle/UTP.

5.3.2 VDM-RT Interlocking FMU

For the encoding of the interlocking FMU, we first declare an alphabet that introduces the variables for the FMU inputs, outputs, state and time.

```
alphabet ilock_state =
  (* Input Variables *)
  cdv :: "bool vector"
  tc :: "bool vector"
  (* Output Variables *)
  signals :: "bool vector"
  switches :: "switch vector"
  (* Interlocking State *)
  relays :: "bool vector"
  (* Simulation Time *)
  time :: "TIME"
```

We make use of the `vector` type to represent fixed vectors of booleans and switch configurations. The type `switch` is introduced to represent the possible orientations of a railway switch. Railways switches can either be set to `STRAIGHT` or `DIVERGING`. Lastly, the variable `time` records simulation time; we use it to impose restrictions on admissible simulation step sizes.

We note that the above is in direct correspondence with the VDM-RT model of the software controller of the FMU. The complete VDM-RT model of the controller is included in Appendix A.

The four major sequential tasks discussed in Section 5.1.2 can now be directly translated into a sequential program within Isabelle/UTP that corresponds to the periodic behaviour of the controller in each execution cycle. For instance, Fig. 21 includes the encoding of the relay activation operations in Fig. 16 on page 45. Conditional **if** statements are encoded here using the infix syntax $P \lhd b \rhd Q$ of the UTP [20]. Here, b is the condition, and P and Q correspond to

```
definition set_relays :: "ilock_state hrel" where
[urel_defs]: "set_relays =
  ((relays[1] := true) ◁ TC[4] ∧ ¬ TC[3] ∧ ¬ R2 ∧ ¬ R4 ∧
    (*¬ CDV[3] ∧*) CDV[4] ∧ CDV[5] ▷ᵣ II) ;;
  ((relays[2] := true) ◁ TC[3] ∧ ¬ TC[4] ∧ ¬ R1 ∧ ¬ R3 ∧ ¬ R4 ∧ ¬ R5 ∧
    (*¬ CDV[3] ∧*) CDV[4] ∧ CDV[8] ∧ CDV[9] ∧ CDV[10] ∧ CDV[1] ▷ᵣ II) ;;
  ((relays[4] := true) ◁ TC[3] ∧ ¬ TC[4] ∧ ¬ R1 ∧ ¬ R2 ∧ ¬ R3 ∧ ¬ R5 ∧
    (*¬ CDV[3] ∧*) CDV[4] ∧ CDV[8] ∧ CDV[9] ∧ CDV[11] ∧ CDV[2] ▷ᵣ II) ;;
  ((relays[3] := true) ◁ TC[1] ∧ ¬ R2 ∧ ¬ R4 ∧ ¬ R5 ∧
    (*¬ CDV[1] ∧*) CDV[10] ∧ CDV[9] ∧ CDV[8] ∧ CDV[7] ∧ CDV[6] ▷ᵣ II) ;;
  ((relays[5] := true) ◁ TC[2] ∧ ¬ R2 ∧ ¬ R3 ∧ ¬ R4 ∧
    (*¬ CDV[2] ∧*) CDV[11] ∧ CDV[9] ∧ CDV[8] ∧ CDV[7] ∧ CDV[6] ▷ᵣ II)"
```

Figure 21: Encoding of the VDM-RT code fragment for setting relays.

the **then** and **else** branch. We also note that `CDV[i]` and `TC[i]` are a custom syntax to access the elements of the vectors `cdv` and `tc` of the state.

The other sequential fragments `clear_relays`, `set_switches` and `set_relays` are encoded in a similar fashion. We point to the Isabelle technical report [29] for their definition.

We, lastly, have a statement that determines the admissible increase of simulation time, via the `time` variable. Here, this is modelled as an assignment that (deterministically) increases `time` by the period of the periodic thread of the cyclic executive for the interlocking program. This gives us the following complete relational FMU model of the interlocking:

```
definition ilock_cycle :: "ilock_state hrel" where [urel_defs]:
"ilock_cycle =
  (set_relays ;; clear_relays ;; set_switches ;; set_signals ;; time := (&time + δ))"
```

Above, δ is introduced as a constant for the thread period. This means that the simulation step size of the entire co-simulation is determined by the interlocking FMU, and corresponding to the execution period of the hardware controller of the interlocking. The process lifting discussed in the next section ensures that we respect admissible simulation step sizes, raising `fmi2Discard` signals when necessary.

The CDV/TC Merger FMU is encoded in a similar manner. Its key functionality is to set the elements of the `cdv` and `tc` vectors according to the `current_track`$_{[1/2]}$ and `telecommand`$_{[1/2]}$ signals produced by the train FMUs.

We summarise by observing that the FMU models in this section are expressed as pure data operations in a relational setting. They are moreover

refinements of the abstract FMU models discussed in INTO-CPS Deliverable D2.3a. The key difference is that we are dealing with continuous and implementation models here, and that the state of FMUs is localised, meaning that each FMU has its own state. In comparison, state in the abstract models in INTO-CPS Deliverable D2.3a is both discretised and centralised. We use data refinement techniques [23] to discharge the refinement proofs.

To conclude our modelling example, we finally look at the underlying *Circus* processes of the co-simulation.

5.4 *Circus* Processes

The process model lifts the relational FMU programs into reactive ones that can interact with a master algorithm and the environment. In the following, we discuss the reactive models of the FMU processes, master algorithm, and the composite FMI model that integrates all of them. The complete mechanised model can be found in the report `https://github.com/isabelle-utp/utp-main/blob/master/fmi/railways_model.pdf`.

5.4.1 FMU Processes

For the FMU processes, we use precisely the lifting approach described in Section 3.3. Hence, for each FMU we create a sketch process like the one in Fig. 8 (p. 26). For this, the state components *cparams*, *cinputs*, *coutputs* and *cstate* of the sketch process are instantiated to the respective concrete parameters, inputs, outputs and state components of the relational FMU model that it encapsulates.

The definition of the data operation(s) for the FMU are then directly added to the sketch process. We recall that *Circus* provides a rich language to model data operations on its state, including programming constructs and specification statements. The data operation that characterises the FMU behaviour is used to replace *CalculateStep* in the sketch process, whereby defining its reactive behaviour according to the lifted (relational) FMU model.

For languages that involve constructs that are not supported in *Circus*, we follow the approach described in INTO-CPS Deliverable 2.3d on linking theories. That approach explains how we extract a relational model, for instance, from a continuous description in the hybrid relational calculus.

5.4.2 Master Algorithm

For our railways example, we encode a fixed-step-size master algorithm with no roll-back. Our concrete MA model aggregates the abstract model of a master algorithm described in Section 3.1, albeit imposing additional constraints on the possible interactions. For instance, we introduce a *FixedStepTimer* process that composes the *Timer* process in Fig. 2 in parallel with another process that restricts time increments to match a given fixed step size (here, this is the δ of the interlocking FMU model).

The fixed-step MA also integrates directly the *Interaction* process (Fig. 5, p. 19) that describes the general interaction patterns of a master algorithm, albeit forcing termination upon `fmi2DoStep` returning `fmi2Discard`, which indicates that the simulation step size is too large for some FMU.

5.4.3 Composite FMI Model

The complete model is the composition of the sketch processes of the lifted (concrete) relational FMU models in interleaving, in parallel with the fixed-step master algorithm. It is a process of the following shape:

$$RailwaysCosim \mathrel{\widehat{=}} FixedStepMA \;[\![\; FMI\_API \;]\!]\; \left(\begin{array}{l} Train_1\_FMU \\ |||\; Train_2\_FMU \\ |||\; Merger\_FMU \\ |||\; Interlocking\_FMU \end{array} \right)$$

Communication between the master algorithm and FMUs is via the channels of the FMI API, for which we introduce the above channel set FMI_API.

This concludes our FMI co-simulation model of the railways case study. We next consider analysis and proof within its mechanisation.

5.5 Analysis and Proofs

Our first concern for analysis is to validate properties of the master algorithm implementation. We note that functional properties of co-simulations are addressed in INTO-CPS Deliverable D2.3a.

Secondly, we can prove that our model is divergence free. Reactive contracts allow us to directly express this property, via the pre-condition of a reactive

design that corresponds to the master algorithm. More precisely, we write this as a conjecture $\mathbf{pre}(P)$ for some *Circus* process P; the distribution laws for $\mathbf{pre}(P)$, described in INTO-CPS Deliverable D2.3b allows us then to evaluate and prove this conjecture in Isabelle/UTP.

Thirdly, concrete master algorithms must be a refinement of our abstract MA model in Section 3.1. To show this, we use refinement laws of *Circus* that we have mechanised in Isabelle/UTP. The proof is facilitated here by the fact that our concrete model aggregates the abstract model while imposing additional constraints, but in general our technique is not constrained to a particular shape of a master algorithm: the developer is free to propose and validate his own implementations.

Discharging the abovementioned refinement proof establishes that the concrete MA is conformant with our abstract model, and hence the FMI specification and standard.

A fourth property we address is that the abstract model of a co-simulation, described in INTO-CPS Deliverable D2.3a, is refined by the concrete (reactive) model presented in this deliverable. This is done via a refinement strategy that transforms the abstract model in such a way that the master algorithm emerges from the refinement. High-level objectives of this refinement strategy are enumerated as follows:

1. Localise the (central) state of the abstract co-simulation into FMUs;
2. Introduce communications that carry out parameter setting and local initialisations; and
3. Introduce channels for FMUs to exchange data; these are later on to become the channels of the FMI API.

The master algorithm emerges through refinement from applying specialised and high-level *Circus* laws that realise the above steps (2) and (3).

5.6 Final Considerations

We have illustrated how our *Circus* model of FMI co-simulations can be instantiated with an industrial case study from ClearSy (France), one of our project partners. This allows for proof-based analysis of the co-simulation model. The focus of such analysis here complements the approach proposed in INTO-CPS Deliverable D2.3a that targets functional properties of co-simulations, and provides additional guarantees of correctness. Our approach

facilitates refinement all the way down to code, which can then be translated into executable languages such as C/C++ and Java.

Our experiments show that Isabelle's code generation comes in useful here, as it enables us to verify tools that efficiently perform symbolic manipulations and refinements of, for instance, *Circus* models.

Our technique brings together many aspects of our foundational work on FMI co-simulations, relying on the semantic embedding of languages for describing co-simulation models, encoding of CPS and *Circus*, and the linking of theories. Isabelle/UTP has proved to be an invaluable tool to mechanise and reason about this ensemble of languages.

6 Related Work and Conclusion

In this deliverable, we have presented our mechanised semantics for FMI, using the state-rich process algebra *Circus*. We thereby have introduced a *reactive* model of FMI co-simulations that refines the abstract relational model in INTO-CPS Deliverable D2.3a. Both models together allow for a refinement-based development technique from abstract specifications of co-simulations to executable co-simulation models that can be simulated by tools.

Our approach described here can verify that a concrete co-simulation is faithful to its abstract model and retains all properties that have been proved of the former. For the railways example, we show, for instance, in INTO-CPS Deliverable D2.3a that trains cannot derail or collide. That property holds for a discrete abstraction of the co-simulation state and model, and here we establish that it carries over to the concrete reactive model, too, that considers continuous behaviours and FMU implementation models.

The Isabelle theories of our mechanisation can be found in the Isabelle/UTP repository on Github: `https://github.com/isabelle-utp/utp-main/fmi`. A report of the FMI mechanisation is available from `https://github.com/isabelle-utp/utp-main/blob/master/fmi/fmi_report.pdf`. The Isabelle theory sources of the FMI model and railways case study together amount to approximately 4,300 lines of definitions and proof scripts.

The approach of applying refinement techniques to hybrid systems has been explored in other contexts, such as the B Method for formal verification. An extension of it, called Hybrid Event B is proposed by Banach et al. in [3]. Similarities exist to the Hybrid Relational Calculus in that restrictions are made in Hybrid Event B that enforce continuous evolutions and discrete state changes to strictly alternate. Both approaches use piecewise continuous functions as a mathematical foundation. (Hybrid) Event B is, however, more restrictive in its modelling patterns than *Circus*, imposing a static execution model based on action systems [2]. Whether the FMI paradigm can be expressed within it is open to investigation.

Here, we have also shown how our FMI semantics can be used to very master algorithms. Broman has a similar objective in [5]. For this, he defines a semantics that is based on functions rather than relations. Our relational FMU model can encode this semantics, but additionally can represent nondeterminism in FMUs. The issue of proposing new master algorithms is prevalent in many other works [4, 25, 12, 9], which emphasises the need and value of

verification techniques like the one we developed.

Tripakis and Broman in [26] further explore the idea of mapping other formalisms into their model to facilitate descriptions of FMUs in various languages, including finite state machines, and discrete-event as well as synchronous data flow actor models. Our work benefits form the UTP as a lingua franca to formalise and integrate semantic theories of various heterogeneous languages, making such an integration easier to carry out.

Future Outlook

Future work will elicit refinement laws and strategies that are tailored and relevant for particular applications. We are also planning to encode a wider repository of master algorithms that, for instance, support variable step size and roll-back. Tool support for automatic generation of the *Circus* model will aid engineers without expert knowledge in formal modelling or proof to apply our technique, and future work on refinement strategies and tactics shall further reduce the need for such domain-specific knowledge, fully realising the potential for automation that we pointed out in Section 5.6.

We see the core future potential of our approach in paving the way for tools that automatically generate code for co-simulations and master algorithms from formal models that have been constructed, analysed and rigorously validated using our technique. This provides crucial certification evidence, namely that co-simulations used to examine and test CPSs are correct.

The synergy of (a) properties proved of particular co-simulations, as explained in INTO-CPS Deliverable D2.3a, and (b) the assurance that co-simulations, when they constitute part of certifications evidence, are based on correct implementations (the topic of this deliverable), is what we believe will strengthen the case for certifying safety-critical CPS in challenging areas of deployment in the future, such as autonomous vehicles and UAVs.

References

[1] Modelica Association. Functional Mock-up Interface for Model Exchange and Co-Simulation. Technical Report Document Version 2.0, Linköping University (Sweden), July 2014. Available from http://fmi-standard.org/downloads/.

[2] R.-J. Back and R. Kurki-Suonio. A New Paradigm for the Design of Concurrent Systems. VII(6), 1987.

[3] R. Banach, M. Butler, S. Qin, N. Verma, and H. Zhu. Core Hybrid Event-B I: Single Hybrid Event-B machines. *Science of Computer Programming*, 105(Supplement C):92–123, July 2015.

[4] J. Bastian, C Clauß, S. Wolf, and P. Schneider. Master for Co-Simultion Using FMI. In *International Modelica Conference*, pages 115–120, March 2011. Available from http://publica.fraunhofer.de/documents/N-162331.html.

[5] D. Broman, L. Greenberg, E. A. Lee, M. Masin, S. Tripakis, and M. Wetter. Requirements for Hybrid Cosimulation Standards. In *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control*, HSCC 2015, pages 179–188. ACM, April 2015.

[6] A. Cavalcanti, P. Clayton, and C. O'Halloran. From control law diagrams to Ada via *Circus*. *Formal Aspects of Computing*, 23(4):465–512, July 2011.

[7] A. Cavalcanti, A. Sampaio, and J. Woodcock. A Refinement Strategy for *Circus*. *Formal Aspects of Computing*, 15(2):146–181, November 2003.

[8] A. Cavalcanti, J. Woodcock, and N. Amalio. Behavioural Models for FMI Co-simulations. In *Proceeings of ICTAC 2016*, volume 9965 of *Lecture Notes in Computer Science*, pages 255–273. Springer, October 2016.

[9] J. Denil, B. Meyers, P. De Meulenaere, and H. Vangheluwe. Explicit Semantic Adaptation of Hybrid Formalisms for FMI Co-Simulation. In *Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M & S Symposium*, DEVS '15, pages 99–106. Society for Computer Simulation International, 2015.

[10] D. Broman et al. Determinate Composition of FMUs for Co-simulation. In *Proceedings of EMSOFT 2013*, pages 2:1–2:12. IEEE Press, September 2013.

[11] T. Blochwitz et al. The Functional Mockup Interface for Tool independent Exchange of Simulation Models. In *Proceedings of the 8th International Modelica Conference*, pages 105–114, March 2011.

[12] Y. A. Feldman, L. Greenberg, and E. Palachi. Simulating Rhapsody SysML Blocks in Hybrid Models with FMI. In *Proceedings of the 10th International Modelica Conference*, pages 43–52, March 2014.

[13] S. Foster, B. Thiele, A. Cavalcanti, and J. Woodcock. Towards a UTP Semantics for Modelica. In *Proceedings of UTP 2016, Revised Selected Papers*, volume 10134 of *Lecture Notes in Computer Science*, pages 44–64. Springer, June 2017.

[14] S. Foster, F. Zeyda, and J. Woodcock. Isabelle/UTP: A Mechanised Theory Engineering Framework. In *Unifying Theories of Programming: 5th International Symposium, UTP 2014*, volume 8963 of *Lecture Notes in Computer Science*, pages 21–41. Springer, May 2015.

[15] S. Foster, F. Zeyda, and J. Woodcock. Unifying Heterogeneous State-Spaces with Lenses. In *Proceedings of ICTAC 2016*, volume 9965 of *Lecture Notes in Computer Science*, pages 295–314. Springer, October 2016.

[16] C. Gomes, C. Thule, D. Broman, P. G. Larsen, and H. Vangheluwe. Co-simulation: State of the art. *ArXiv e-prints*, arXiv:1702.00686, February 2017.

[17] O. Grumberg and H. Veith. *25 Years of Model Checking: History, Achievements, Perspectives*, volume 5000 of *Lecture Notes in Computer Science*. Springer, 2008.

[18] J. He and L. Qin. A Hybrid Relational Modelling Language. In *Concurrency, Security, and Puzzles: Essays Dedicated to Andrew William Roscoe on the Occasion of His 60th Birthday*, volume 10160 of *Lecture Notes in Computer Science*, pages 124–143. Springer, 2016.

[19] T. Hoare. *Communicating Sequential Processes*. Prentice Hall Series in Computer Science. Prentice-Hall, Upper Saddle River, NJ, USA, April 1985.

[20] T. Hoare and J. He. *Unifying Theories of Programming*. Prentice Hall Series in Computer Science. Prentice-Hall, Upper Saddle River, NJ, USA, April 1998.

[21] B. Huffman and O. Kunčar. Lifting and Transfer: A Modular Design for Quotients in Isabelle/HOL. In *Proceedings of Certified Programs*

and Proofs: Third International Conference, CPP 2013, volume 8307 of *Lecture Notes in Computer Science*, pages 131–146. Springer, December 2013.

[22] F. Immler and J. Hölzl. Numerical Analysis of Ordinary Differential Equations in Isabelle/HOL. In *Proceedings of ITP 2012*, volume 7406 of *Lecture Notes in Computer Science*, pages 377–392. Springer, 2012.

[23] C. Morgan. *Programming from Specifications*. Prentice Hall Series in Computer Science. Prentice-Hall, Hertfordshire, HP2 4RG, UK, January 1996.

[24] M. Oliveira, A. Cavalcanti, and J. Woodcock. A UTP semantics for *Circus*. *Formal Aspects of Computing*, 21(1):3–32, February 2009.

[25] U. Pohlmann, W. Schäfer, H. Reddehase, J. Röckemann, and R. Wagner. Generating Functional Mockup Units from Software Specifications. In *Proceedings of the 9th International Modelica Conference*, pages 765–774, September 2012.

[26] S. Tripakis and D. Broman. Bridging the Semantic Gap Between Heterogeneous Modeling Formalisms and FMI. Technical Report UCB/EECS-2014-30, EECS Department, University of California, Berkeley, April 2014.

[27] L. G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[28] J. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof*. Prentice Hall Series in Computer Science. Prentice-Hall, Upper Saddle River, NJ, USA, April 1996.

[29] F. Zeyda, S. Foster, and A. Cavalcanti. Mechanisation of the Railways FMI Cosimulation, November 2017.

[30] F. Zeyda, S. Foster, and L. Freitas. An Axiomatic Value Model for Isabelle/UTP. In *Unifying Theories of Programming, 6th International Symposium*, volume 10134 of *Lecture Notes in Computer Science*, pages 155–175. Springer, June 2016.

[31] F. Zeyda, M. Oliveira, and A. Cavalcanti. Mechanised support for sound refinement tactics. *Formal Aspects of Computing*, 24(1):127–160, January 2012.

A VDM-RT Interlocking Controller

Interlocking.vdmrt

```
class Interlocking

types

SWITCH_POSITION = <STRAIGHT> | <DIVERGING>

instance variables

private hwi : HardwareInterface;

private Relay : seq of bool;

private Switch : seq of SWITCH_POSITION;

operations

-- Constructor for Interlocking
public Interlocking: HardwareInterface ==> Interlocking
Interlocking(hardware) ==
(
  hwi := hardware;
  Relay := [false,false,false,false,false];
  Switch :=
[<DIVERGING>,<DIVERGING>,<DIVERGING>,<DIVERGING>,<DIVERGING>]
);

-- Control loop
public Step: () ==> ()
Step() ==
(
    -- Relay Setting
    if hwi.TC(4) and not hwi.TC(3) and not Relay(2) and not Relay
(3) and hwi.CDV(4) and hwi.CDV(5)
       then Relay(1) := true;
    if hwi.TC(3) and not hwi.TC(4) and not Relay(1) and not Relay
(3) and not Relay(4) and not Relay(5) and hwi.CDV(4) and hwi.CDV
(8) and hwi.CDV(9) and hwi.CDV(10) and hwi.CDV(1)
       then Relay(2) := true;
    if hwi.TC(3) and not hwi.TC(4) and not Relay(1) and not Relay
(2) and not Relay(3) and not Relay(5) and hwi.CDV(4) and hwi.CDV
(8) and hwi.CDV(9) and hwi.CDV(11) and hwi.CDV(2)
       then Relay(4) := true;
    if hwi.TC(1) and not Relay(2) and not Relay(4) and not Relay
(5) and hwi.CDV(10) and hwi.CDV(9) and hwi.CDV(8) and hwi.CDV(7)
```

Page 1

VDM-RT Interlocking Controller

Interlocking.vdmrt

```
and hwi.CDV(6)
       then Relay(3) := true;
    if hwi.TC(2) and not Relay(2) and not Relay(3) and not Relay
(4) and hwi.CDV(11) and hwi.CDV(9) and hwi.CDV(8) and hwi.CDV(7)
and hwi.CDV(6)
       then Relay(5) := true;

    /* Relay Clearing */
    if Relay(1) and not hwi.CDV(5) then Relay(1) := false;
    if Relay(2) and not hwi.CDV(1) then Relay(2) := false;
    if Relay(3) and not hwi.CDV(6) then Relay(3) := false;
    if Relay(4) and not hwi.CDV(2) then Relay(4) := false;
    if Relay(5) and not hwi.CDV(6) then Relay(5) := false;

    /* Switch Positioning */
    Switch(1) := <STRAIGHT>;
    if Relay(1)
      then Switch(3) := <STRAIGHT>
      else Switch(3) := <DIVERGING>;
    if Relay(3) or Relay(5)
      then Switch(2) := <STRAIGHT>
      else Switch(2) := <DIVERGING>;
    Switch(4) := <STRAIGHT>;
    if Relay(2) or Relay(3)
      then Switch(5) := <STRAIGHT>
      else Switch(5) := <DIVERGING>;

    /* Signal Settings */
    hwi.signals(1) := Relay(3) and Switch(5) = <STRAIGHT> and
Switch(2) = <STRAIGHT> and Switch(4) = <STRAIGHT>;
    hwi.signals(2) := Relay(5) and Switch(5) = <DIVERGING> and
Switch(2) = <STRAIGHT> and Switch(4) = <STRAIGHT>;
    hwi.signals(3) := (Relay(1) and Switch(1) = <STRAIGHT> and
Switch(3) = <STRAIGHT>)
     or (Relay(2) and Switch(1) = <STRAIGHT> and Switch(3) =
<DIVERGING> and Switch(2) = <DIVERGING> and Switch(5) =
<STRAIGHT>)
     or (Relay(4) and Switch(1) = <STRAIGHT> and Switch(3) =
<DIVERGING> and Switch(2) = <DIVERGING> and Switch(5) =
<DIVERGING>);

    /* Switches Actuators */
    hwi.switches(1) := if Switch(1) = <STRAIGHT> then true else
false;
```

Page 2

VDM-RT Interlocking Controller

Interlocking.vdmrt

```
    hwi.switches(2) := if Switch(2) = <STRAIGHT> then true else
false;
    hwi.switches(3) := if Switch(3) = <STRAIGHT> then true else
false;
    hwi.switches(4) := if Switch(4) = <STRAIGHT> then true else
false;
    hwi.switches(5) := if Switch(5) = <STRAIGHT> then true else
false;
);

-- 10Hz control loop
thread periodic(1E8, 0, 0, 0)(Step);

end Interlocking
```

Page 3

B Mechanised Railways Architecture

jEdit - Architecture_D2_3c.thy

```
1  (*****************************************************************************)
2  (* Project: The Isabelle/UTP Proof System                                    *)
3  (* File: Architecture_D2_3c.thy                                              *)
4  (* Authors: Frank Zeyda and Simon Foster (University of York, UK)            *)
5  (* Emails: frank.zeyda@york.ac.uk and simon.foster@york.ac.uk                *)
6  (*****************************************************************************)
7
8  section {* Railways Architecture *}
9
10 theory Architecture_D2_3c
11 imports fmi
12 begin recall_syntax
13
14 subsection {* Preliminaries *}
15
16 subsubsection {* Track Segments *}
17
18 definition "CDV1 = (1::fmi2Integer)"
19 definition "CDV2 = (2::fmi2Integer)"
20 definition "CDV3 = (3::fmi2Integer)"
21 definition "CDV4 = (4::fmi2Integer)"
22 definition "CDV5 = (5::fmi2Integer)"
23 definition "CDV6 = (6::fmi2Integer)"
24 definition "CDV7 = (7::fmi2Integer)"
25 definition "CDV8 = (8::fmi2Integer)"
26 definition "CDV9 = (9::fmi2Integer)"
27 definition "CDV10 = (10::fmi2Integer)"
28 definition "CDV11 = (11::fmi2Integer)"
29
30 subsubsection {* Available Routes *}
31
32 definition "V1Q1 = (1::fmi2Integer)"
33 definition "V1Q2 = (2::fmi2Integer)"
34 definition "Q2V2 = (3::fmi2Integer)"
35 definition "V1Q3 = (4::fmi2Integer)"
36 definition "Q3V2 = (5::fmi2Integer)"
37
38 subsubsection {* Signal Encoding *}
39
40 definition "RED == False"
41 definition "GREEN == True"
42
43 fun signals :: "(bool × bool × bool) ⇒ fmi2Integer" where
44 "signals (s1, s2, s3) =
45   (if s1 then 1 else 0) +
46   (if s2 then 2 else 0) +
47   (if s3 then 4 else 0)"
48
49 subsubsection {* Switch Encoding *}
50
```

07/11/17 23:06 :: page 1

Mechanised Railways Architecture

jEdit - Architecture_D2_3c.thy

```
51  definition "STRAIGHT == False"
52  definition "DIVERGING == True"
53
54  fun switches :: "(bool × bool × bool × bool × bool) ⇒ fmi2Integer" where
55  "switches (sw1, sw2, sw3, sw4, sw5) =
56    (if sw1 then 1 else 0) +
57    (if sw2 then 2 else 0) +
58    (if sw3 then 4 else 0) +
59    (if sw4 then 8 else 0) +
60    (if sw5 then 16 else 0)"
61
62  subsection {* Model Instantiation *}
63
64  subsubsection {* Railways FMUs *}
65
66  axiomatization
67    train1 :: "FMI2COMP" and
68    train2 :: "FMI2COMP" and
69    merger :: "FMI2COMP" and
70    interlocking :: "FMI2COMP" where
71    fmus_distinct: "distinct [train1, train2, merger, interlocking]" and
72    FMI2COMP_def: "FMI2COMP = {train1, train2, merger, interlocking}"
73
74  overloading
75    railways_FMUs ≡ "FMUs :: FMI2COMP list"
76  begin
77    definition railways_FMUs :: "FMI2COMP list" where
78    "railways_FMUs = [train1, train2, merger, interlocking]"
79  end
80
81  paragraph {* Proof Support *}
82
83  lemma fmu_simps [simp]:
84  "train1 ≠ train2"
85  "train1 ≠ merger"
86  "train1 ≠ interlocking"
87  "train2 ≠ train1"
88  "train2 ≠ merger"
89  "train2 ≠ interlocking"
90  "merger ≠ train1"
91  "merger ≠ train2"
92  "merger ≠ interlocking"
93  "interlocking ≠ train1"
94  "interlocking ≠ train2"
95  "interlocking ≠ merger"
96  using fmus_distinct apply (auto)
97  done
98
99  subsubsection {* Parameters *}
100
```

07/11/17 23:06 :: page 2

Mechanised Railways Architecture

jEdit - Architecture_D2_3c.thy

```
101 overloading
102   railways_parameters ≡ "parameters :: (FMI2COMP × VAR × VAL) list"
103 begin
104   definition railways_parameters :: "(FMI2COMP × VAR × VAL) list" where
105   "railways_parameters = [
106     (train1, $max_speed:{fmi2Real}ᵤ, InjU (4.16::real)),
107     (train2, $max_speed:{fmi2Real}ᵤ, InjU (4.16::real)),
108     (train1, $fixed_route:{fmi2Integer}ᵤ, InjU V1Q2),
109     (train2, $fixed_route:{fmi2Integer}ᵤ, InjU Q3V2)]"
110 end
111
112 subsubsection {* Inputs and Outputs *}
113
114 overloading
115   railways_inputs ≡ "inputs :: PORT list"
116 begin
117   definition railways_inputs :: "PORT list" where
118   "railways_inputs = [
119     (train1, $signals:{fmi2Integer}ᵤ),
120     (train1, $switches:{fmi2Integer}ᵤ),
121     (train2, $signals:{fmi2Integer}ᵤ),
122     (train2, $switches:{fmi2Integer}ᵤ),
123     (merger, $track_segment1:{fmi2Integer}ᵤ),
124     (merger, $track_segment2:{fmi2Integer}ᵤ),
125     (merger, $telecommand1:{fmi2Integer}ᵤ),
126     (merger, $telecommand2:{fmi2Integer}ᵤ),
127     (interlocking, $CDV:{fmi2Integer}ᵤ),
128     (interlocking, $TC:{fmi2Integer}ᵤ)]"
129 end
130
131 overloading
132   railways_outputs ≡ "outputs :: PORT list"
133 begin
134   definition railways_outputs :: "PORT list" where
135   "railways_outputs = [
136     (train1, $track_segment:{fmi2Integer}ᵤ),
137     (train1, $telecommand:{fmi2Integer}ᵤ),
138     (train2, $track_segment:{fmi2Integer}ᵤ),
139     (train2, $telecommand:{fmi2Integer}ᵤ),
140     (merger, $CDV:{fmi2Integer}ᵤ),
141     (merger, $TC:{fmi2Integer}ᵤ),
142     (merger, $collision:{fmi2Boolean}ᵤ),
143     (merger, $derailment:{fmi2Boolean}ᵤ),
144     (interlocking, $signals:{fmi2Integer}ᵤ),
145     (interlocking, $switches:{fmi2Integer}ᵤ)]"
146 end
147
148 subsubsection {* Initial Values *}
149
150 text {* The following constants have to be defined as appropriate. *}
```

07/11/17 23:06 :: page 3

Mechanised Railways Architecture

jEdit - Architecture_D2_3c.thy

```
151
152 definition "initialSignals = InjU (signals (RED, RED, RED))"
153 definition "initialSwitches =
154   InjU (switches (STRAIGHT, STRAIGHT, STRAIGHT, STRAIGHT, STRAIGHT))"
155 definition "initialTrack1 = InjU CDV3"
156 definition "initialTrack2 = InjU CDV2"
157
158 overloading
159   railways_initialValues ≡ "initialValues :: (PORT × VAL) list"
160 begin
161   definition railways_initialValues :: "(PORT × VAL) list" where
162   "railways_initialValues = [
163     ((train1, $signals:{fmi2Integer}ᵤ), initialSignals),
164     ((train1, $switches:{fmi2Integer}ᵤ), initialSwitches),
165     ((train2, $signals:{fmi2Integer}ᵤ), initialSignals),
166     ((train2, $switches:{fmi2Integer}ᵤ), initialSwitches),
167     ((merger, $track_segment1:{fmi2Integer}ᵤ), initialTrack1),
168     ((merger, $track_segment2:{fmi2Integer}ᵤ), initialTrack2),
169     ((merger, $telecommand1:{fmi2Integer}ᵤ), undefined),
170     ((merger, $telecommand2:{fmi2Integer}ᵤ), undefined),
171     ((interlocking, $CDV:{fmi2Integer}ᵤ), undefined),
172     ((interlocking, $TC:{fmi2Integer}ᵤ), undefined)]"
173 end
174
175 subsubsection {* Port Dependency Graph (‹pdg›) *}
176
177 definition pdg :: "(PORT × PORT) list" where
178 "pdg = [
179   ((train1, $track_segment:{fmi2Integer}ᵤ), (merger, $track_segment1:{fmi2Integer}ᵤ)),
180   ((train2, $track_segment:{fmi2Integer}ᵤ), (merger, $track_segment2:{fmi2Integer}ᵤ)),
181   ((train1, $telecommand:{fmi2Integer}ᵤ), (merger, $telecommand1:{fmi2Integer}ᵤ)),
182   ((train2, $telecommand:{fmi2Integer}ᵤ), (merger, $telecommand2:{fmi2Integer}ᵤ)),
183   ((merger, $CDV:{fmi2Integer}ᵤ), (interlocking, $CDV:{fmi2Integer}ᵤ)),
184   ((merger, $TC:{fmi2Integer}ᵤ), (interlocking, $TC:{fmi2Integer}ᵤ)),
185   ((interlocking, $signals:{fmi2Integer}ᵤ), (train1, $signals:{fmi2Integer}ᵤ)),
186   ((interlocking, $signals:{fmi2Integer}ᵤ), (train2, $signals:{fmi2Integer}ᵤ)),
187   ((interlocking, $switches:{fmi2Integer}ᵤ), (train1, $switches:{fmi2Integer}ᵤ)),
188   ((interlocking, $switches:{fmi2Integer}ᵤ), (train2, $switches:{fmi2Integer}ᵤ))
189 ]"
190
191 subsubsection {* Internal Direct Dependencies (‹idd›) *}
192
193 definition idd :: "(PORT × PORT) list" where
194 "idd = [
195   ((merger, $track_segment1:{fmi2Integer}ᵤ), (merger, $CDV:{fmi2Integer}ᵤ)),
196   ((merger, $track_segment2:{fmi2Integer}ᵤ), (merger, $CDV:{fmi2Integer}ᵤ)),
197   ((merger, $telecommand1:{fmi2Integer}ᵤ), (merger, $TC:{fmi2Integer}ᵤ)),
198   ((merger, $telecommand2:{fmi2Integer}ᵤ), (merger, $TC:{fmi2Integer}ᵤ)),
199   ((interlocking, $CDV:{fmi2Integer}ᵤ), (interlocking, $signals:{fmi2Integer}ᵤ)),
200   ((interlocking, $CDV:{fmi2Integer}ᵤ), (interlocking, $switches:{fmi2Integer}ᵤ)),
```

07/11/17 23:06 :: page 4

Mechanised Railways Architecture

jEdit - Architecture_D2_3c.thy

```
201   ((interlocking, $TC:{fmi2Integer}ᵤ), (interlocking, $signals:{fmi2Integer}ᵤ)),
202   ((interlocking, $TC:{fmi2Integer}ᵤ), (interlocking, $switches:{fmi2Integer}ᵤ))
203 ]"
204 end
```

07/11/17 23:06 :: page 5