Formalised Cosimulation Models

Frank Zeyda[1], Julien Ouy[2], Simon Foster[1], and Ana Cavalcanti[1]

[1] University of York, Department of Computer Science, York, YO10 5GH, UK
[2] ClearSy, Parc de la Duranne, 320 Avenue Archimède Les Pléiades III — Bât A, 13857 Aix-en-Provence Cedex 3, France

frank.zeyda@york.ac.uk simon.foster@york.ac.uk ana.cavalcanti@york.ac.uk

Abstract. Cosimulation techniques are popular in the design and early testing of cyber-physical systems. Such systems are typically composed of heterogeneous components and specified using a variety of languages and tools; this makes their formal analysis beyond simulation challenging. We here present work on formalised models and proofs about cosimulations in our theorem prover Isabelle/UTP illustrated by an industrial case study from the railways sector. Novel contributions are a mechanised encoding of the FMI framework for cosimulation, simplification and translation of (case-study) models into languages supported by our proof system, and an encoding of an FMI instantiation.

1 Introduction

Cosimulation techniques are popular in the design of cyber-physical systems (CPS) [10]. Such systems are typically specified using a variety of languages and tools that adopt complementary modelling paradigms, such as physical models, control laws, and sequential, concurrent and real-time programming. The industrial standard FMI (Functional Mock-up Interface) [6] addresses the challenge of coupling different simulators and simulations. It defines an API used to implement master algorithms that mitigate issues of interoperability.

Our aim is to complement cosimulation with proof-based techniques. Simulation is useful in helping engineers to understand modelling implications and spot design issues, but cannot provide universal guarantees of correctness and safety. This is due to the complexity of CPS in considering continuous behaviours as well as real-world interactions, and the impracticality of running an exhaustive number of simulations. It is besides often not clear how the evidence provided by simulations is to be qualified, since simulations depend on parameters and algorithms, and are software systems (with possible faults) in their own right.

Challenges in analysing heterogeneous CPS *formally* are multifarious. Firstly, we have to consider the semantics of various modelling approaches and languages. Secondly, we have to consolidate those semantic models to enable us to reason about the system as a whole. And thirdly, realistic industrial systems are often difficult to tackle by formal approaches due their complexity and level of detail. A key challenge remains to find abstractions that make the system tractable for

formal analysis, and at the same time not forfeit fidelity, so that formal analysis can justify and support claims about the real system under scrutiny.

In this paper, we outline our approach to address the above challenges using an industrial application from railways. The example is a system developed by ClearSy and involves control models of trains in 20-sim [22] and an implementation of the interlocking system in VDM-RT [18]. We first show how the initial models of the industrial example can be simplified and expressed in notations for which we have a precise semantics: these are Modelica for dynamic systems modelling, and VDM-RT for concurrent real-time programming. We then present work on encoding the models in our theorem prover Isabelle/UTP [9]. Part of this is also a mechanised reactive and timed model of the FMI framework, which we formulate in the *Circus* [2] process algebra for state-rich reactive systems.

Our contributions in this paper are summarised as follows. Firstly, we simplify an industrial railways application and case study and reformulate it in notations which we have embedded into our Isabelle/UTP theorem prover; secondly, we encode the FMI framework for cosimulation in Isabelle/UTP; and thirdly, we encode key parts of the railways model together with examples of proofs.

The rest of the paper is organised as follows. In Section 2, we review preliminary material: Modelica and VDM-RT, and the *Circus* language. FMI and its mechanisation are described in Section 3. Then, Section 4 discusses the railways case study and our simplified FMI model of it, and Section 5 details our encoding in Isabelle/UTP, including obstacles and challenges we faced. Lastly, in Section 6 we conclude and outline areas of future work.

2 Preliminaries

We proceed by reviewing preliminary material.

2.1 Modelica and VDM-RT

Modelica [19] is a language for continuous systems modelling. It is applicable to a large spectrum of domains, including physical, electrical, and control systems. Various tools exist that provide simulation support for Modelica.

Modelica allows control models to be expressed either in explicit equational form or as control laws. Continuous behaviours are described by virtue of Differential Algebraic Equations (DAE) [21]. We may also define discontinuous state changes when some condition (guard) becomes true.

An example of a Modelica *control law* relevant to our railways example is given in Fig. 1. The control law models the deceleration of a moving body — in our case the braking of a train. The model consists of two integrators, *Velocity* and *Position* that calculate the velocity and travelled distance of the body. We introduce a control switch to set its acceleration to a fixed value of $-1.4ms^{-2}$ when the velocity is greater than zero, and otherwise to $0ms^{-2}$.

We can simulate this model to confirm that the body stops after $v_{init}^2/(2a)$ metres, where v_{init} is a particular initial speed of the body and a the deceleration.

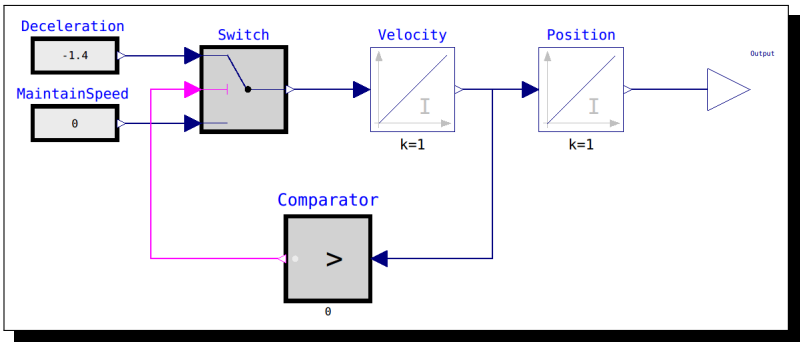


Fig. 1. Modelica control law of a decelerating moving object.

However, simulation cannot prove that this is indeed the case in every scenario. A safety concern is that a train must always be able to stop in time to not overrun a red signal or ill-set point. We discuss a more realistic train model in Section 4 that enables us to examine such proofs in the context of a cosimulation with two trains moving independently on a given track layout.

Control laws such as the one in Fig. 1 are interpreted as equational systems in Modelica and flattened into a single large system of simultaneous equations, some of which correspond to connecting wires of the control diagram, and others to the specification of subcomponents. Modelica also provides limited support for `algorithms` formulated in sequential statements. Yet, their semantics does not account for execution time, concurrency, and effects on data sharing. Support for those features is provided by VDM-RT, which we briefly discuss next.

VDM-RT [18] is a real-time extension of the VDM language [17], which supports sequential program development from model-based specifications. It has a precise semantics that enables correctness proofs. Verification of implementations is supported by tools such as Overture [5]. Beyond VDM, VDM-RT has features to model execution time and concurrency. It also adds system entities that correspond to clocks, CPUs, threads and communication buses.

Our technique to reason about Modelica and VDM-RT specifications is based on a unifying semantic framework, the Unifying Theories of Programming (UTP) of Hoare and He [13]. We have mechanised the UTP inside the Isabelle/HOL theorem prover [9], and within that mechanisation also encoded the semantics of various languages for CPS modelling and design, including Modelica [8], Simulink and VDM-RT [7]. A third language is used as a front-end to the UTP to tie our models together: the process algebra *Circus* [2]. We summarise it next.

2.2 The *Circus* language

Circus is a process algebra similar to CSP [12], but with additional support for defining data operations and state. *Circus* inherits from CSP, for instance, sequential and parallel composition, input and output communications on a chan-

Name	Syntax	Description
Sequence	$A \; ; \; B$	Execute A and B in sequence.
Parallelism	$A \; [\![\; cs \;]\!] \; B$	Execute A and B in parallel, synchronising on the channels in the channel set cs.
External Choice	$A \; \Box \; B$	The environment decides whether A or B is executed; communication resolves the choice.
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.
Assignment	$x := e$	Assignment to a state component x.
Stop	**stop**	Stop and refuse any further communication.

Table 1. Overview of relevant *Circus* operators on actions.

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

Fig. 2. Timer process of the *Circus* FMI specification.

nel, external choice, interrupt, and recursion. A summary of operators relevant to our models in this paper is included in Table 1.

To define a process state (**state** paragraph), a *Circus* process declares a record whose fields define a data model. Data operations can either be written as Z operation schemas [23] or constructs from Morgan's refinement calculus [20], such as specification statements, assignment, conditionals and iteration. A notable trade-off in *Circus* is that the language enforces non-interference of parallel computations; this endows it with a rich set of laws that can be used to verify *Circus* implementations against abstract (non-executable) *Circus* models.

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 model 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 *Timer*. *Step* is an external choice (operator $\Box$) that offers communication on the channels $setT$, $updateSS$, *step* and *end*. These channels are declared (with their types) by the **channel** construct right above the process.

The channel events here are used by simulation (master) algorithms to model the progression of time during cosimulation steps. The environment can change *currentTime* and *stepSize* through communication on the channels $setT$ and $updateSS$, respectively. When a *step* event occurs, modelling a cosimulation 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 interaction. Otherwise, the *Step* action behaves recursively, repeating the previously described communication behaviour.

3 FMI and its Mechanisation

The conceptual view of an FMI cosimulation entails a master algorithm (MA) to orchestrate the cosimulation, and several Functional Mock-up Units (FMUs) that wrap tool and vendor-specific simulation components. The FMI standard [1] not only specifies the API by which MAs must communicate with the FMUs, but also how control and exchange of data must be realised. Typically, the master algorithm reads outputs from all FMUs and then forwards them to the FMUs that require them as inputs. After this, the MA notifies the FMUs to concurrently compute the next simulation step. Some master algorithms assume a fixed step size while others enquire the largest step size that the FMUs are cumulatively willing to accept. MAs sometimes also perform roll-backs of already performed simulation steps, and suitably deal with errors raised during cosimulation. We hence have a design space of possible implementations of master algorithms.

Our model of FMI formalises a cosimulation (including the MA and FMUs) as a collection of *Circus* processes. There exist processes that specify the interaction of master algorithms with FMUs, as well as processes that describe the behaviour of *particular* master algorithms. For illustration, the top-level abstract architecture of master algorithms is depicted in Fig. 3. Each box corresponds to a *Circus* process and the **thick** red lines between them highlight internal and external communications. The basic (non-composite) processes of the *Circus* model are *Timer*, *Interaction*, *FMUStatesManager*, *ErrorManager*, *ErrorMonitor* and *FatalErrorMonitor*. Their surrounding boxes are parallel compositions.

The *Timer* process has already been discussed in Section 2.2: its purpose is to ensure that simulation time increases in accordance with the current time and step size. More complex is the *Interactions* process, which determines the order in which FMI functions that initialise FMUs, read their outputs (`fmi2Get`), set their inputs (`fmi2Set`), and invoke the next simulation step (`fmi2DoStep`) must be called. Function calls are modelled by communication events on special channels prefixed with *fmi*2. Restrictions on the permissible order of FMI function calls, as defined in the FMI standard [1], are thus captured by the *observable event traces* in our process model. For instance, the *Interaction* process includes

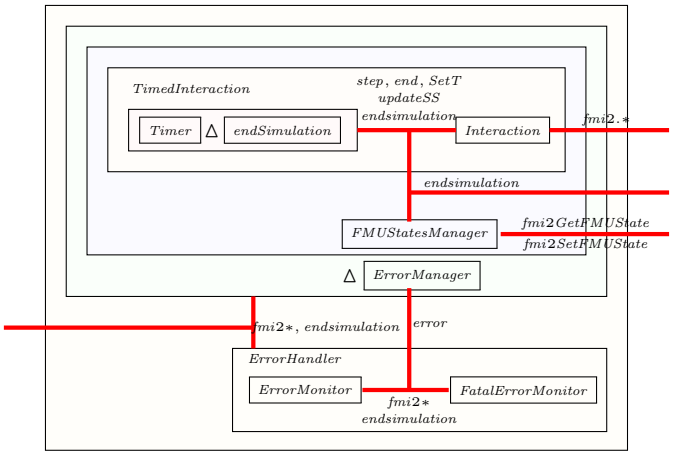


Fig. 3. Overview of the abstract *Circus* model for master algorithms.

local actions *TakeOutputs* and *DistributeInputs* that correspond to phases of the control cycle of a master algorithm, whereas *FMUStatesManages* prescribes the use of functions `fmi2GetFMUState` and `fmi2SetFMUState` to obtain and set FMU states during roll-back. An in-depth discussion of the *Circus* model can be found in [3]; in the remainder of the section, we report on its mechanisation.

In essence, we translate *Circus* notations into corresponding operators in our embedding of *Circus* in Isabelle/UTP. To give an example, a mechanised version of the *Timer* process from Fig. 2 is presented in Fig. 4. We note, however, that this (and other) mechanised processes do not have a *State* definition since the process state is implicit in the variables used within actions.

One challenge that we faced was the encoding of mixed prefixes of inputs and outputs on the same channel. These, we translate into a single input communication. This solution requires us to supply an input variable for each output (`out1` and `out2` in Fig. 4). That input is, however, preconditioned to only accept a particular value, thereby emulating the behaviour of an output. Another challenge is the encoding of recursive actions, such as `Step` in Fig. 4. In general, our tool rewrites local actions into a chain of HOL `let` statements, and fixed-point predicates are used to encode single-recursive actions. Parameters are dealt with via hidden stack variables, which allows for an elegant treatment of scopes.

To give another example, the mechanised encoding of the `TakeOuputs` action of the *Interaction* process in Fig. 3 is recaptured below.

```
TakeOutputs = <rinp::(port × VAL) list> := ⟨⟩ ;;
  (;; out : outputs •
    fmi:fmi2Get.[out₁](«FMU out»).[out₂](«name out»)?(v)?(st) →ᶜ
      (;; inp : pdg out • <rinp> := &<rinp> ^ᵤ ⟨(«inp», «v»)ᵤ⟩))
```

It reads the outputs of all FMUs (first iterated sequence `;;`) and stores them in a the state component `rinp` of the process, so that they can subsequently be

```
definition
"process Timer(ct::TIME('τ), hc::NZTIME('τ), tN::TIME('τ)) ≙ begin
  Step =
    (tm:setT?(t : «t ≤ tN») →C (<currentTime> := «t») ;; Step) □
    (tm:updateSS?(ss) →C (<stepSize> := «ss») ;; Step) □
    (tm:step![out1]($<currentTime>)![out2]($<stepsize>) →C
      (<currentTime::'τ> :=
        minu(&<currentTime> + §(&<stepSize::'τ pos>), «tN»)) ;; Step) □
    ((&<currentTime> =u «tN») &u tm:endc →C Stop)
  • (<currentTime>, <stepSize> := «ct», «hc») ;; Step
end"
```

Fig. 4. Mechanised Timer process in Isabelle/UTP.

forwarded to the FMUs requiring them, prior to initiating the next simulation step. The HOL type of `rinp` is a list of pairs whose first component is an FMU port, and whose second component is a permissible FMI value.

We note that `pdg` is a global constant that determines the port-dependency relationships between the input and output ports of FMUs. Our mechanised model introduces it via an Isabelle `constant` definition, alongside other global constants to determine the identifiers of FMUs, their parameters, initial values, and so on. Isabelle `constant`s are uninterpreted, so that concrete FMI instantiations can define suitable value for these constants. An example of this is given later on in Section 5, where we consider our railways case study.

The complete mechanised FMI model can be found in [24]. Our contribution is that (a) we embedded the syntax and semantics of *Circus* into Isabelle/UTP on top of its UTP CSP model, and (b) achieved a direct correspondence between *Circus* notations and Isabelle constructs of the mechanisation.

4 The Railways Case Study

Our case study considers an existing tramway station. Its railway layout is presented in the diagram of Fig. 5. Trains enter the interlocking at the points Q2, Q3 and V1, and then issue a telecommand to request a route. Telecommand stations are denoted by the green dots, and possible routes through the railway network are Q2→V2, Q3→V2, V1→Q1, V1→Q2 and V1→Q3.

Access to the interlocking is controlled by the signals S28, S48 and S11. They are initially set to red causing trains arriving on the tracks CDV_Q2, CDV_Q3 and CDV_11 to stop and wait. When a telecommand is issued by a train, the control logic of the interlocking allocates a free route, if available, and then gives the respective train a green light to go ahead. No other train is allowed to proceed meanwhile. This guarantees that no collision can occur, namely due to multiple trains passing through the same track segment. The control logic also caters for the setting of track points (SW1-5) so that trains move on the allotted paths.

The inputs of the interlocking controller are the CDV and telecommand boolean vectors. The CDV is a bit vector whose elements register the presence

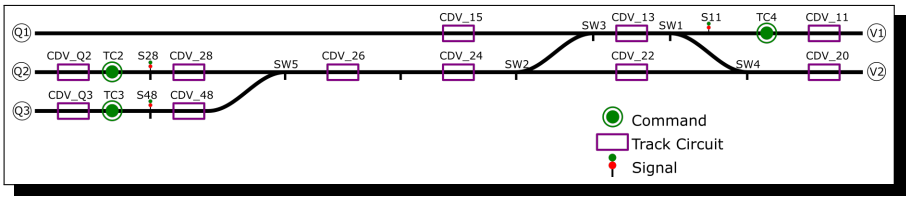
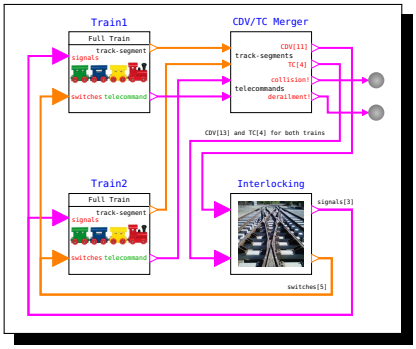


Fig. 5. Railway interlocking layout of the case study




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

/* Control equation for acceleration and braking of the train. */
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;
```

Fig. 6. FMI cosimulation (left) and train control equations (right).

of a train on a particular track segment. Telecommand requests are likewise encoded by bit vectors where each bit corresponds to a particular route request. Outputs (actuators) of the interlocking are signals and track point switches that control the paths of trains when they proceed through the interlocking.

A high-level view of the system as a cosimulation is given in the left diagram of Fig. 6. There are four FMU components. Two of them, Train1 and Train2, simulate the physical behaviour of the trains, which includes the actions of the train driver in adjusting 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 merge the CDV and telecommand outputs of both trains into single boolean vectors. A supplementary function of CDV/TC Merger is to calculate monitoring signals for collision and derailment.

The initial models for this case study define the train physics and their control behaviour by bond diagrams in 20-sim, and the interlocking in VDM-RT. To make those models tractable for formal analysis, we have simplified and encoded them in Modelica (for which we have a mechanised semantics). We hence consider traction and braking actions but do not model train mass and gravity, and neither smooth acceleration and braking curves (jerk). This simplification is justified because the influence of the more precise model does not alter the fundamental system behaviour and is negligible in analytical terms.

The kinematics and speed control of both trains is encoded by the equations in the right-hand diagram of Fig. 6. The first `equation` block captures motion:

```
algorithm
  setpoint_speed := 0;
  if current_track > 0 then
    signalID := Topology.signal_tab[current_track];
    if signalID <> NONE then
      if signals[signalID] == GREEN then
        setpoint_speed := max_speed;
      else
        setpoint_speed := 0;
      end if;
    else
      setpoint_speed := max_speed;
    end if;
  end if;
```

Fig. 7. Modelica function (body) for calculating the train set-point speed.

acceleration is the derivative (`der(_)` operator) of velocity, and velocity the derivative of position. While an accurate physics model of the train would be expressed in terms of traction and braking forces, the assumption of constant train mass and Newton's law entitles us to consider acceleration alone.

The second `equation` block realises a simplified control algorithm: train 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` statement that simultaneously sets the train speed to the set-point speed and acceleration to zero if we are close to the set-point speed. This is to avoid chattering during simulation and can also be thought of as 'engaging the brakes' when the train approaches zero speed while decelerating. We note that this equational characterisation is partly equivalent to the control law in Fig. 1, with the added feature of considering not merely braking but also up-regulation of the train speed. For formal analysis, the explicit version in Fig. 6 is more suitable as it is formulated in terms of derivatives rather than integrals, making the conversion into an ODE or DAE easier.

The behaviour of the 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 or no signal on the current track; otherwise, it sets it to zero (see Fig. 7).

Encapsulation of algorithmic behaviours into Modelica `function`s, where possible, is a deliberate refactoring. Our encoding profits from this as those functions can be naturally encoded as HOL functions into the proof system. This kind of engineering has a modularising ripple-on effect on subsequent proofs.

A last aspect of the train model considers equations for the discontinuous variable changes that occur when the train reaches the end of a track and enters

the next track. The Modelica equations for this are given 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 resets `position_on_track` back to zero.

In addition to the above, we also need an equation that generates telecommand signals when the train is on a track equipped with a telecommand station, but we omit its straightforward definition here.

As already mentioned, the VDM-RT interlocking has also been simplified from the production code in hardware. To capture its essential behaviour, we introduce a variable Relay to record the state for relay switches that, in real hardware, record the locking of a particular route for a train that requests it. Below is an extract of the sequential program logic that performs the locking.

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

For the locking to occur, a telecommand must have been issued that actually requests the respective route; this is achieved by the condition on the bit vector TC that cumulatively records the telecommands issued by all three telecommand stations. The constraints on Relay ensure that locked routes are non-intersecting, so that trains can pass without crossing each other's 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.

While our software implementation retains the core logic of the hardware realisation, it does not consider time delays incurred by the latency of relay and point actuators. We hence assume that both process signals quickly enough to disregard such delays. For relays, delays are in fact not an issue since all they cause is a (very small) delay in trains obtaining permission to proceed.

5 Encoding in Isabelle/UTP

We consider two aspects of the encoding here: the mechanised FMI system model (Section 5.1) and the continuous train FMUs (Section 5.2). All our Isabelle theories are available at: https://github.com/isabelle-utp/utp-main.

5.1 FMI System Model

The FMI system model introduces concrete definitions for uninterpreted constants of the abstract FMI model described in Section 3. These constants determine the names of the FMUs, their input and output ports, initial values and parameters, and graphs that capture internal and external port dependencies. The latter two are relevant to establish the absence of algebraic loops within the cosimulation architecture. Instantiation of the model for a particular cosimulation is realised by an `axiomatization` in Isabelle/UTP, as shown below.

```
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}"
```

Here, we introduce the constants `train1`, `train2`, `merger` and `interlocking` of a given (abstract) type `FMU2COMP`, together with axiomatic constraints that ensure that (a) the constants are distinct, and (b) there exist no other values in the type `FMU2COMP`.

An extract of the port dependency graph of our system is sketched below:

```
definition pdg :: "port relation" where
"pdg = {
  (* External Dependencies *)
  ((train1, $track_segment:{int}ᵤ), (merger, $track_segment1:{int}ᵤ)),
  ((train2, $track_segment:{int}ᵤ), (merger, $track_segment2:{int}ᵤ)), ...,
  (* Internal Dependencies *)
  ((merger, $track_segment1:{int}ᵤ), (merger, $CDV:{int}ᵤ)),
  ((merger, $track_segment2:{int}ᵤ), (merger, $CDV:{int}ᵤ)), ...
}"
```

External dependencies correspond to connection arrows in Fig. 6, and internal dependencies arise from direct signal feed-through within FMUs. Above, we can see that a direct internal dependency exists between the inputs and outputs of the `merger` block. There is, however, no such dependency between inputs and outputs of the `train` FMUs due to integrator behaviours (Fig. 1). For this reason, our feedback system does not contain an algebraic loop. We have proved this by using Isabelle's code evaluation framework and tactics; it amounts to showing that the `pdg` is acyclic. The proof of this can be found in the report [24], too.

5.2 Continuous Train Model

Continuous and hybrid behaviour is given a semantics in terms of the hybrid relational calculus (HRC) [11]. We have mechanised this calculus in Isabelle/UTP using the Multivariate Analysis and HOL-ODE theory libraries [15].

The hybrid relational calculus extends the UTP with continuous variables, which are encoded using timed traces . A timed trace, as illustrated in Fig. 8, is a partial function $\mathbf{tt} : \mathbb{R}_{\geq 0} \nrightarrow \Sigma$, such that $\mathrm{dom}(\mathbf{tt}) = [0, \ell)$, for some $\ell : \mathbb{R}_{\geq 0}$, and $\mathbf{tt}$ is piecewise continuous. Type Σ is a topological space that defines the entire

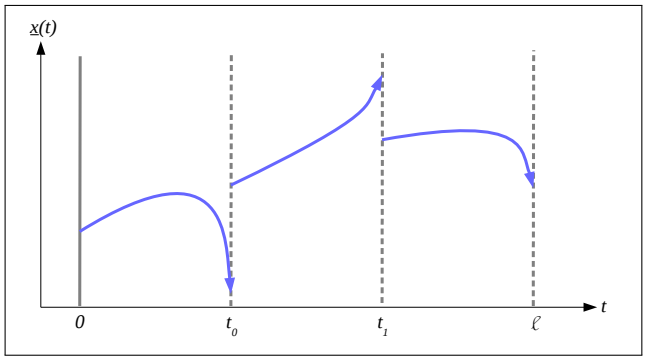


Fig. 8. Piecewise continuous function modelling a timed trace.

continuous state type, accommodating all continuous variables. Typically, Σ is associated with a vector space of type $\mathbb{R}^n$.

A continuous variable is decorated with an underscore $\underline{x}$ to distinguish it from a discrete variable. Like timed traces, continuous variables are functions on time. A key feature of the hybrid relational calculus is the ability to perform discrete assignments to continuous variables. This is achieved by pairing each continuous variable $\underline{x}$ with an assignable discrete copy variable x, such that $x = \underline{x}(0)$ holds for the before state, and $x' = \lim_{t \to \ell} \underline{x}(t)$ holds for the after state, for any $\ell > 0$.

Hybrid relations are constructed using common programming operators, such as assignment and sequential composition, plus various operators to specify continuous evolutions. For instance, we adopt the interval operator $\lceil P \rceil$ from the Duration Calculus [4] in order to specify constraints on the continuous variables during evolution, such that P must be satisfied at every instant.

With the above, we define evolution operator $\underline{x} \leftarrow f(x_0, t) \mathrel{\widehat{=}} \lceil \underline{x} = f(x_0, t) \rceil$. It specifies that continuous variable $\underline{x}$ evolves according to the function f, whose parameters are the initial values x_0 and time t, for any evolution duration ℓ. We also have $\underline{x} \leftarrow_n f(x_0, t)$, which presumes a definite duration $\ell = n$. Lastly, we define the pre-emption operator P **until**$_b$ b that permits evolutions according to P until the condition b becomes true; it thus imposes constraints on the possible durations ℓ after which control passes to the next hybrid computation.

The Multivariate Analysis package [15] of Isabelle provides a precise encoding of real numbers as Cauchy sequences and several operators from the integral and differential calculus. We use that package and our interval operator to encode ordinary differential equations (ODEs) in the hybrid relational calculus. Namely, $\langle\, \dot{x} = f(t, x) \,\rangle$ specifies that the derivative of $x(t)$ is given by $f(t, x(t))$ — a function of the current time and continuous state. Using Immler's HOL-ODE package [16] we can certify symbolic solutions to initial value problems, and thus reduce $\langle\, \dot{x} = f(t, x) \,\rangle$ to a function evolution $x \leftarrow g(x_0, t)$ where g is the solution to $\dot{x}(t) = f(t, x(t))$ with initial condition x_0.

We describe below part of the Modelica train model from Section 4 in the hybrid relational calculus. We focus on the situation when the train is stop-

ping due to an approaching red signal. The other behaviours 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*. 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.

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

We first assign initial values to the continuous variables, and this effectively creates initial conditions for the ODE. We then evolve the continuous variables, according to the ODE, until the velocity reaches 0. 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 right diagram of Fig. 6, 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. Those, however, are not differential equations and can likewise use the interval operator previously described.

We have encoded the example in Isabelle/UTP and mechanised a proof (see Fig. 9) that the train stops before the track ends, that is, $\lceil pos < 44 \rceil$ holds, where $44m$ is the track length of CDV_Q2 in Fig. 5. For the sake of brevity, we elide details of the proof, other than the first four steps. The proof proceeds as follows.

1. Solve the ODE *symbolically* to obtain a function evolution statement. This requires us to show Lipschitz continuity of the ODE so that, via the Picard Lindelöf theorem, there is precisely one such solution;
2. Use the assigned values to obtain the set of initial conditions;
3. Calculate the precise time at which the velocity reaches zero; here, that is approximately after 2.97 seconds;
4. Prove that the position at every earlier instant is less than 44 metres.

The final step requires that we solve a polynomial inequality

$$(104/25) * t - (7/10) * t^2 < 44$$

which includes the solution for the position derivative. In Isabelle, this can be done using the lesser-known *approximate* tactic [14], which safely employs a floating-point approximation to prove the conjecture with respect to the reals.

Our analysis has proceeded directly at the level of the Modelica train model, and our next aim shall be to transfer this result to the FMI cosimulation model of the entire system. For this, the train models are wrapped into *Circus* processes corresponding to the train FMUs in the left diagram of Fig. 6. This is on-going work; our initial results provide evidence that our semantic theories and reasoning framework is up to the challenge of proving properties in this context.

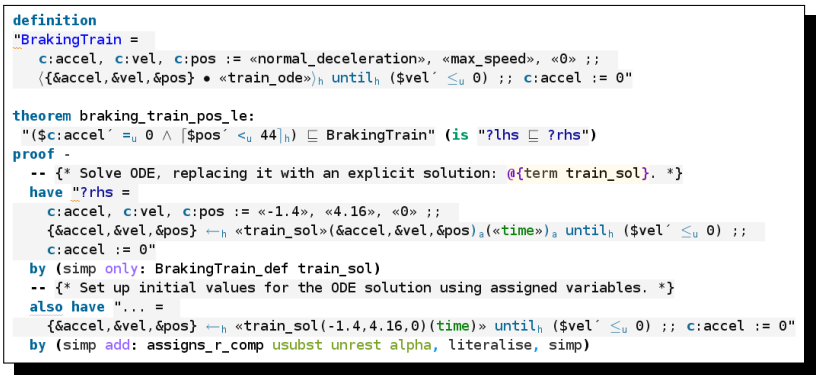


Fig. 9. The braking train scenario encoded in Isabelle/UTP.

6 Conclusion

We have reported on some initial results in formalising and mechanising FMI cosimulations in our theorem prover Isabelle/UTP.

The relevance of our project is to enable proofs about cosimulated systems, as well as the cosimulation itself. Such proofs may, for instance, entail behavioural correctness and safety properties, such as trains never collide. We also envisage proofs that validate the suitability of simulations to observe faults or — vice versa — provide tangible evidence for their absence. While the details of how this can be done touches upon open research problems, the models and their encoding described here are a first important step into this direction.

A collateral contribution is to provide an encoding of the *Circus* language, as this was required to mechanise the semantics of the FMI framework. While our proof system currently offers support for CSP, the *Circus* language poses further challenges related to the representation of *Circus* processes and actions, dynamic channel declarations, and specialised *Circus* operators.

Future work will address the completion of our models and investigate proof strategies and laws to reason about the cosimulation as a whole. In addition, we aim to elicit and verify properties of master algorithms that hold independently of the simulators and structure of the simulated model as an FMI system.

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. A. Cavalcanti, A. Sampaio, and J. Woodcock. A Refinement Strategy for *Circus*. *Formal Aspects of Computing*, 15(2):146–181, November 2003.
3. A. Cavalcanti, J. Woodcock, and N. Amalio. Behavioural Models for FMI Co-simulations. In *Theoretical Aspects of Computing — ICTAC 2016*, volume 9965 of *LNCS*, pages 255–273. Springer, October 2016.

4. Z. Chaochen, T. Hoare, and A. P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, December 1991.
5. P. G. Larsen et al. Tutorial for Overture/VDM-RT. Technical Report TR-005, September 2015. http://overturetool.org/documentation/tutorials.html.
6. T. Blochwitz et al. The Functional Mockup Interface for Tool independent Exchange of Simulation Models. In *Proc. of the 8th Int. Modelica Conference*, 2011.
7. S. Foster, A. Cavalcanti, S. Canham, K. Pierce, and J. Woodcock. Final Semantics of VDM-RT. Deliverable 2.2b, INTO-CPS Project, H2020 Grant 644047, December 2016. http://projects.au.dk/fileadmin/D2.2b_Final_VDM-RT_Semantics.pdf.
8. 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 *LNCS*, pages 44–64. Springer, June 2017.
9. S. Foster, F. Zeyda, and J. Woodcock. Isabelle/UTP: A Mechanised Theory Engineering Framework. In *Proceedings of UTP 2014*, volume 8963 of *LNCS*, pages 21–41. Springer, May 2014.
10. 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.
11. 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 *LNCS*, pages 124–143. Springer, 2016.
12. T. Hoare. *Communicating Sequential Processes*. Prentice-Hall, April 1985.
13. T. Hoare and J. He. *Unifying Theories of Programming*. Prentice-Hall, April 1998.
14. J. Hölzl. Proving Inequalities over Reals with Computation in Isabelle/HOL. In *Proceedings of ACM SIGSAM PLMMS 2009*, pages 38–45, August 2009.
15. J. Hölzl, F. Immler, and B. Huffman. Type Classes and Filters for Mathematical Analysis in Isabelle/HOL. In *Proceedings of ITP 2013*, volume 7998 of *LNCS*, pages 279–294. Springer, July 2013.
16. F. Immler and J. Hölzl. Numerical Analysis of Ordinary Differential Equations in Isabelle/HOL. In *Proceedings of ITP 2012*, volume 7406 of *LNCS*, pages 377–392. Springer, August 2012.
17. C. B. Jones. *Systematic Software Development using VDM*. Prentice-Hall, 1990.
18. K. Lausdahl, M. Verhoef, P. G. Larsen, and S. Wolff. Overview of VDM-RT Constructs and Semantic Issues. In *Proceedings of the 8th Overture Workshop*, volume 1224 CS-TR, pages 57–67, September 2010.
19. Modelica Association. *Modelica® – A Unified Object-Oriented Language for Systems Modeling, Language Specification, Version 3.4*, April 2017. Available from https://www.modelica.org/documents/.
20. C. Morgan. *Programming from Specifications*. Prentice-Hall, January 1996.
21. L. Petzold. Differential/Algebraic Equations are not ODEs. *SIAM Journal on Scientific and Statistical Computing*, 3(3):367–384, 1982.
22. J. v. Amerongen, C. Kleijn, and C. Gamble. Continuous-Time Modelling in 20-sim. In *Collaborative Design for Embedded Systems: Co-modelling and Co-simulation*, pages 27–59. Springer, Berlin, Heidelberg, March 2014.
23. J. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof*. Prentice-Hall, April 1996.
24. F. Zeyda, S. Foster, and A. Cavalcanti. Mechanisation of the FMI. Technical report, University of York, UK, June 2017. Available from https://github.com/isabelle-utp/utp-main/blob/master/fmi/fmi_report.pdf.