Grant Agreement: 644047

INtegrated TOol chain for model-based design of CPSs

SysML Foundations for INTO-CPS

Deliverable Number: D2.3a

Version: 1.0

Date: December 2017

Public Document

http://into-cps.au.dk

Contributors:

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

Editors:

Frank Zeyda, UY

Reviewers:

Hajer Saada, UTRC
Adrian Pop, LIU
Carl Gamble, 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	01-11-2017	Frank Zeyda	Draft for internal review
1.0	15-12-2017	Frank Zeyda	Final version

Abstract

In our third and final instalment of the INTO-CPS deliverable on SysML Foundations, we complement the earlier work with general techniques that can be used to reason about co-simulation models specified using the INTO-SysML profile and implemented via the Functional Mock-up Interface (FMI). We explain our techniques by applying them to industrial case studies from the railways domain and smart buildings control. Both are part of the four major case studies of the INTO-CPS project. In doing so, we illustrate how mechanical theorem proving can add value to the analysis of co-simulation models, establishing both well-formedness of the architecture and behavioural properties of the co-simulation system. Our major result is to lay the foundations of a compositional verification approach for FMI based on Hoare logic and refinement techniques.

Contents

1 Introduction

This report constitutes the final deliverable on SysML Foundations of the INTO-CPS project. It complements the earlier deliverables on this topic by proposing a general technique for proof-based analysis of co-simulations that considers both architectural and behavioural properties of an FMI model. The technique is illustrated by way of two case studies, one from from railways and another one from the area of smart buildings control.

1.1 Background and Motivation

Co-simulation techniques are popular in the design of cyber-physical systems (CPS) [18]. Such systems are typically engineered using a variety of languages and tools that adopt complementary paradigms; examples are physics-related models, control laws, and sequential, concurrent and real-time programs. This diversity makes CPS generally difficult to analyse and study. The Functional Mock-up Interface (FMI) Standard [11] has been proposed to alleviate that problem and has since been successfully used in industry. It addresses the challenge of interoperability, coupling different simulators and their high-level control components via a bespoke FMI API[1].

While (co)simulation is currently the predominant approach to analyse CPS, this report describes a proof-based complementary technique that uses mathematical reasoning and logic. Simulation is useful in helping engineers to understand modelling implications and spot design issues, but cannot provide universal guarantees of correctness and safety. It is usually impossible to run an exhaustive number of simulations as a way of testing the system. For these reasons, it is 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.

Proof-based techniques, on the other hand, hold the promise of making universal claims about systems. They can potentially abstract from particular simulation scenarios, parametrisations of models, and interaction patterns used for testing. In traditional software engineering, they have been successfully used to validate the correctness of implementations against abstract requirements models [5]. Yet, their application to CPS is fraught with difficulties: the heterogeneous combination of languages used in typical descriptions of CPS raises issues of semantic integration and complexity in reasoning

[1] **A**bstract **P**rogramming **I**nterface

about those models. The aspiring ideal of any verification technique is a compositional approach, and such approaches are still rare for CPS [30].

1.2 Contribution

This deliverable reports on a novel technique for proof-based analysis of co-simulation models of CPS, describe by SysML UML models such as the one in Fig. 3 on page 24. We discuss proofs of various relevant properties of a co-simulation model, such as architectural well-formedness and safety properties of the abstract FMI system model. We illustrate our technique firstly through an industrial case study from railways that is based on a real system and provided by ClearSy, a consultancy company for the application of formal methods to safety-critical systems (http://www.clearsy.com/). The case study is described in detail in INTO-CPS Deliverable D1.3 in general terms. Here, we focus on mechanically encoding it in our proof tool Isabelle/UTP [16], tailored for refinement-based reasoning about multi-paradigm languages. We moreover examine a second example from buildings environmental control, which is described in detail in INTO-CPS Deliverable D1.1d.

The added contributions of this deliverable in comparison to the preceding INTO-CPS Deliverable D2.2a are summarised as follows.

1. We show how well-definedness proofs of FMI co-simulation models can be formulated and automated in a proof assistant.
2. We describe a general compositional approach for proof-based analysis of CPS via a refinement technique.
3. We illustrate our approach by way of two industrial case studies.
4. We present proofs of invariant and safety properties of our model of the FMI co-simulation for the railways system.
5. We evaluate industrial use and potential automation for our technique.

The remainder of the report is structured as follows. In Section 2, we review preliminary material. Section 3 gives a general account and overview of our technique, and in Section 4 we introduce the railways case study, including detailed descriptions of relevant models. In Section 5, we report on its formal encoding and proof-based analysis. Section 6 is then dedicated to our second example on smart buildings control, and Section 7 examines ramifications of our approach, its industrial application, and related work. Lastly, in Section 8 we conclude the report and outline future work.

2 Preliminaries

We begin by discussing relevant preliminary material: Section 2.1 introduces the INTO-SysML Profile, Section 2.2 explains the FMI Standard, and in Section 2.3 we give a brief overview of Isabelle/UTP, our proof system.

2.1 The INTO-SysML Profile

The Systems Modelling Language (SysML) [33] is a graphical notation for systems engineering, defined as both a specialisation and extension of the Unified Modelling Language (UML) [32]. It is realised by using UML's profile mechanism, which provides a generic technique for customising UML models for particular application domains and platforms.

There exist various commercial and open-source tools for SysML model creation and design. These include IBM's Rational Rhapsody Designer [23], Atego's Modeler [2], Modeliosoft's Modelio [28], as well as the Eclipse-based Papyrus [17] modelling environment. They support model-based engineering and have been used successfully in industry to model complex systems.

A SysML block is a structural element that represents a general component of the system, describing either functional, physical, or human behaviour. The SysML Block Definition Diagram (BDD) shows how blocks are assembled into architectures; it is the analogue of a UML Class Diagram. A BDD illustrates how the system is composed from its blocks using associations and other composition relations.

A SysML Internal Block Diagram (IBD) allows a designer to refine internal block structures; it is similar to UML's Composite Structure Diagram, which shows the internal structure of a class. In an IBD, parts are assembled to define how they collaborate to realise the block's overall behaviour.

The INTO-SysML profile, described in INTO-CPS Deliverable D2.1a, customises SysML for architectural modelling for FMI co-simulation. It embraces the many themes of the INTO-CPS project: tool interoperability, semantic heterogeneity, holistic modelling, and co-simulation, and provides the modelling gateway for the INTO-CPS approach. It specialises blocks to represent different types of components of a CPS. These constitute the building blocks of hierarchical descriptions of CPS architectures. A component can be a logical or conceptual unit of the system: software or a physical entity.

INTO-SysML has two types of diagrams, Architecture Structure Diagrams

(ASD) and Connection Diagrams (CD), specialising SysML BDDs and IBDs. The Modelio tool, mentioned above, has been extended as part of the INTO-CPS project to support the INTO-SysML profile. For an example of an INTO-SysML Connection Diagram, we refer to Fig. 3 on page 24.

In our railways case study, the block models are written in Modelica and VDM-RT. Modelica, developed by the Modelica Association, is an object-oriented equation-based language for modelling, simulating, and analysis of multi-domain dynamic systems. Both open-source and commercial tools exist that support the Modelica language specification. These include, for instance, OpenModelica, JModelica, Dymola, SimulationX, MappleSim, and Wolfram SystemModeler. Modelica supports both control law diagrams and explicit mathematical models via Differential Algebraic Equations (DAE). VDM-RT is a bespoke language and verification method for developing concurrent real-time systems. It is an extension of the Vienna Development Method (VDM) that includes objects and time.

2.2 FMI in a Nutshell

The FMI Standard [3] has been developed jointly by the Modelica Association and several industrial partners. It addresses the challenge of coupling different simulators. It achieves this by defining an API that prescribes the interaction of simulation components in co-simulations.

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 MA drives the simulation by performing data exchange between the FMUs, and managing their initialisation, stepping, and error handling. The MA also determines the step size of simulations, either statically, or dynamically for each step. This is important to ensure fidelity of the simulation with respect to the underlying real-world system.

The conceptual view of an FMI architecture entails one master algorithm and several FMUs that wrap tool and vendor-specific simulation components. The FMI Standard 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 MA 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 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.

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 model of concurrency [37]; 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.

2.3 Isabelle/UTP

Isabelle/UTP is a theorem prover implemented in the Isabelle proof assistant, on top of Higher-Order Logic (HOL). It supports proof in the context of Hoare and He's Unifying Theories of Programming (UTP) [21]. This is a general and unifying framework to define programming language semantics. It adopts a predicative approach that represents computational models as relations over a theory-specific alphabet of variables. These 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.

To give an example, we consider the predicate

$$ok \wedge y \neq 0 \Rightarrow ok' \wedge z' = x \,\mathbf{div}\, y \quad \text{where } \mathbf{div} \text{ is division.} \tag{1}$$

It models the partial assignment $z := x \,\mathbf{div}\, y$. Here, x, y and z are program variables of type integer or real. Primed variables are used to refer to the program state after execution, and unprimed variables to the program state before execution. We point out that ok is a special boolean variable that models program termination. Hence ok being true signifies that the program has started, and ok' being true signifies that it has terminated.

The above predicate (1) admits, for example, the observation[2] $\{ok \rightsquigarrow true, x \rightsquigarrow 6, y \rightsquigarrow 2, z' \rightsquigarrow 3, ok' \rightsquigarrow true\}$, capturing that the program starts in a state where $x = 6$ and $y = 2$, and terminates in a state where $z = 3$. It also admits the observation $\{ok \rightsquigarrow true, x \rightsquigarrow 6, y \rightsquigarrow 0, ok' \rightsquigarrow false\}$, capturing that the program may not terminate if started in a state where $y = 0$. Predicates specify in this way the observations that can be made of a computation

[2]We represent observations as bindings records. Variables not mentioned in the binding can have arbitrary values.

within a particular computational paradigm or model. Here, for instance, the paradigm is sequential programming under total correctness.

In a partial correctness semantics, *ok* would not be needed. For the semantic theory of a process algebra, we may in contrast need additional variables that account for traces of interactions with the environment. During INTO-CPS, we have encoded and mechanised several UTP theories in Isabelle/UTP that can be used to reason about languages relevant to the design of co-simulation models, including Modelica and VDM-RT.

For our proof technique presented in this deliverable, it is sufficient to limit our theory to partial-correctness computations, being modelled by predicates over program state variables only. This already provides a suitable model to validate, for instance, the rules of Hoare logic [20]. This is important as we use those later on in our verification technique.

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 computations as familiar pre- and postcondition pairs (P, Q). The refinement of specifications into programs is simply reverse implication.

We extend the notion of contract to *reactive contracts*, as described in INTO-CPS Deliverable D2.3b. 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. The pre- and postcondition play similar roles as in sequential programs, specifying the states from which the program is guaranteed to terminate (P) and its behaviours upon termination (Q). The pericondition characterises nonterminating albeit nondivergent behaviours, since reactive computations may not terminate and yet do useful things by interacting and communicating with their environment. The three predicates typically impose constraints on a trace variable *tr* recording interactions.

We summarise a few salient issues about Isabelle/UTP. In order to faithfully encode the UTP logic, Isabelle/UTP uses a deep model of predicates based on lenses [16] rather than directly translating them into HOL predicate. In practical terms, this requires a specialised treatment of variables and alphabets, as well as operators, which have to be 'lifted' from HOL into Isabelle/UTP. Most of this is hidden from the user; however, a few notations need to be memorised to write predicates and programs in Isabelle/UTP.

Firstly, the `alphabet` command has to be employed to introduce variables and state spaces. It is part of our tool and accepts similar arguments as Isabelle/HOL's `record` command to define new record types (we show an example of this in Section 5). Secondly, variables have to be decorated with $\$v$ or $\&v$ to indicate whether they refer to plain or relational state spaces. Lastly, operators frequently have to be subscripted, as in $\&x =_{\mathtt{u}} \&y$. This is because they are logically distinct from the corresponding HOL operators, with the subscripted operators denoting lifted versions thereof.

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

3 Proof-based Analysis of SysML Models

In this section, we outline our general approach to reason about SysML models of FMI co-simulations. Verification of architectural properties is the concern of Section 3.1. Section 3.2 introduces a behavioural model of FMI co-simulations, and, in Section 3.3, we discuss its refinement.

3.1 Architectural Model

The architectural model of an FMI co-simulation captures its structure in terms of FMU components and their mutual connections via ports. Each FMU provides a set of input and output ports to exchange data. Ports have names and types, identifying them within the model and specifying the kind of data that they transmit. The formal characterisation of an FMI co-simulation architecture conceptually involves the following elements.

- A finite set of FMU components.
- A finite set of input ports of the FMUs.
- A finite set of output ports of the FMUs.
- Initial values for each of the input ports.
- Possible parameters for each of the FMUs.
- A graph encoding port connections of the FMI system.
- A graph recording internal direct dependencies of FMUs.

```
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 1: Isabelle/HOL mechanisation of an FMI architecture.

Fig. 1 illustrates how we represent the above information in Isabelle/HOL. FMUs are elements of a 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. 1 yields a sequence of all FMUs of the co-simulation. Recording a sequence rather than a set is deliberate. Firstly, it guarantees finiteness of the elements; and secondly, it is useful later on to define iterative operators over FMUs. For example, our FMI model described in INTO-CPS Deliverable D2.3c makes frequent use of iterated constructs, which we define via folding of the respective operator. An iterated sequence is, for instance, used to set FMU parameters prior to simulation.

The model we use here is also used in INTO-CPS Deliverable D2.3c for additional verifications. Our overall approach does not require the construction of several models.

Well-formedness constraints for `FMUs` are that the sequence is injective and that its range is equal to the universe (carrier set) of the `FMI2COMP` type. Both can be effectively checked via automatic proof in Isabelle/HOL.

Ports are formally characterised by pairs of type `FMI2COMP` × `VAR`, where `VAR` encodes variables as name/type pairs. Again, using sequences facilitates iterative operations over inputs and outputs where this is required in our model. An example is the definition of the model of master algorithms, which need to read the outputs of all FMUs and distribute them to their connected input ports. Details of this are in INTO-CPS Deliverable D2.3c. For well-formedness, we require that the sequences for inputs and outputs are both injective, and that they are disjoint.

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`.

For both cases above, we also require that the value x (of type `VAL`) agrees with the type of the variable v. Like those on FMUs, these constraints are easy to check automatically within the Isabelle proof system.

Port dependency graph Connections between FMUs are encoded by the function `pdg`, which maps output ports to their connected input ports. The definition of `pdg` for particular FMI co-simulations can be directly inferred from the Connection Diagram (CD) of the corresponding INTO-SysML model. A constraint on `pdg` is that its domain must be a subset of the elements of `outputs`, and its range must be equal to the elements of `inputs`. This ensures that every input is connected to an output. Furthermore, we require type conformity: for each connected input $inp \in \texttt{range}(\texttt{pdg}\,(out))$ of some output out, inp and out must have the same port type. We recall that ports are encoded by pairs of type $\texttt{FMI2COMP} \times \texttt{VAR}$ where elements of type `VAR` record type information, so that conformity can be iteratively checked for all maplets of the underlying relation of `pdg`.

Internal direct dependencies Lastly, the `idd` constant records direct feed-through dependencies *within* FMU components, mapping FMU outputs to a list of dependent inputs. Hence, for well-formedness, the function must map output ports of an FMU to inputs ports of the *same* FMU. Direct dependencies can give rise to algebraic loops in the overall co-simulation, namely in feedback architectures. This is an issue for simulation stability and must be avoided in well-formed models. Absence of algebraic loops is established by taking the union of the port dependency graph (`pdg`) and internal direct dependencies (`idd`), and requiring that the resulting graph is acyclic. A proof of this can generally be automated for finite relations, though in Isabelle it involves computation of the transitive closure of $\texttt{pdg} \cup \texttt{idd}$.

We conclude that all properties for well-formedness we mentioned above can be automatically proved after instantiating the abstract constants in Fig. 1 with concrete values of particular FMI models. The main reason for this is that all functions and relations are finite.

Property of	**Isabelle Definition**

Parameters	`definition wf_parameters :: "(FMI2COMP × VAR × VAL) list ⇒ bool" where`
	`"wf_parameters params ⟷`
	`(∀(f, x, v)∈set params. ¬ (∃v'. (f, x, v') ∈ set params ∧ v ≠ v'))"`

Initial values	`definition wf_initialValues :: "(PORT × VAL) list ⇒ bool" where`
	`"wf_initialValues inits ⟷`
	`(∀((f, x), v)∈set inits. ¬ (∃v'. ((f, x), v') ∈ set inits ∧ v ≠ v'))"`

Inputs & Outputs	`definition wf_inputs :: "bool" where`
	`"wf_inputs ⟷ distinct inputs"`
	`definition wf_outputs :: "bool" where`
	`"wf_outputs ⟷ distinct outputs"`
	`definition inputs_outputs_disjoint :: "bool" where`
	`"inputs_outputs_disjoint ⟷ (set inputs) ∩ (set outputs) = {}"`

Port Type Conformance	`definition pdg_conformant :: "bool" where`
	`"pdg_conformant ⟷ conformant pdg"`
	`definition idd_conformant :: "bool" where`
	`"idd_conformant ⟷ conformant pdg"`

Control Graph	`definition wf_control_graph :: "bool" where` `"wf_control_graph ⟷ acyclic (set pdg ∪ set idd)"`

Table 1: Well-formedness properties encoded in Isabelle.

Table 1 summarises the encoding of several well-formedness properties in Isabelle/HOL. For most of them, the proof effort is linear in the size of the mathematical structure. Absence of algebraic loops is a more challenging property, as it relies on symbolic computation of the transitive closure, which is a costly operation. We automate this proof using Isabelle's `eval` tactic: rather than performing symbolic rewrite steps, it executes verified ML code to evaluate terms. While this is efficient for relations up to few hundreds of port connections, there exist more complex FMI models with hundreds of FMUs and even thousands of port connections. This pushes Isabelle's built-in code for computing acyclicity via the closure to its limits.

To mitigate the above-noted efficiency issue in analysing large FMI systems, we resort to a complementary solution in which the computation of the transitive closure is 'outsourced' to an external C++ algorithm that employs more efficient techniques, such as [35] running on optimised machine code. The issue here is trust: how do we ascertain that the result of the external algorithm is correct? One possible solution is to verify the algorithm within HOL and use Isabelle's code-generation facilities to produce a certified implementation. This requires merely to trust Isabelle's code generator — we note that use of `eval` already requires that level of trust.

Instead of verifying the algorithm, another solution that we adopt is to verify the *result* of the algorithm each time it is executed. Proving that some relation c is (a super-set of) the closure of some relation r requires less com-

putational effort than calculating the closure itself. Key in using this strategy is the following introduction law that we have proved in Isabelle/HOL:

```
lemma acyclic_witnessI:
"(∃C. acyclic_witness C R) ⟹ acyclic R"
```

The predicate `acyclic_witness` $c\ r$ captures that relation c is both irreflexive and a valid candidate for the positive closure of r, meaning that $r^+ \subseteq c$. An equivalent characterisation, more convenient for proof, is given below.

```
definition acyclic_witness:
"acyclic_witness C R ⟷ R ⊆ C ∧ C O R ⊆ C ∧ irrefl C"
```

Above, `C O R` is the relational composition of `C` and `R`. We observe that the right-hand of this definition requires less effort to show for given `C` and `R` than actually computing the closure of `R`. Our strategy is hence to first apply the introduction lemma above to transform a goal of the form `acyclic R` into a witness proof (∃ `C`. `acyclic_witness C R`); then invoke our external algorithm for calculating $\texttt{R}^+$ and use the result as a witness for `C`; and lastly unfold the definition of `acyclic_witness` and prove the three residual conjuncts.

The advantages of this approach are that (a) we do not have to invest effort in verifying a particular algorithm and can even exchange algorithms without safety implications, and (b), unlike using `eval`, we do not have to trust the code generator of Isabelle. A disadvantage is that the witness proof can still be slow due to large enumerated set terms emerging. To mitigate this, we advocate the use of specialised proof tactics for each of the residual conjuncts; moreover, we have implemented an efficient and robust reconstruction of the HOL relation from the algorithm's output that avoids, for instance, re-parsing of the result produced by the algorithm.

The two key conclusions of our work on verifying structural properties of FMI co-simulation models are that (a) a formal encoding can be produced on the basis of the INTO-SysML model, and (b) proof of well-formedness of that model can be fully automated within Isabelle/HOL without having to resort to external tools for model-checking, even for very large systems. This is by either using Isabelle's built-in features for code generation and evaluation, or alternatively interfacing with external high-performance algorithms. The construction of the formal encoding of the architecture is currently manual, but fundamentally it can be automated as all information needed for this is in the respective INTO-SysML ASD and CD. We expect that, only for the identification of internal direct dependencies, user input is needed.

We next consider the more challenging aspect of formally characterising co-simulation behaviour by way of an abstract computational model.

3.2 FMI Model

In this section, we describe an abstract model of FMI co-simulations: it makes visible the structure of the system in terms of FMUs, but omits detail of how co-simulation behaviour in FMUs is realised, perhaps for instance, by physics-related models or real-time programs. Instead, we consider specifications of FMUs that may be discrete abstractions of continuous models and capture sequential programs as *specification statements*. The rational for this approach is that proofs of system-level safety properties are much easier at this level of abstraction. Refinement is used in a separate phase to transform abstract FMU models into executable FMU co-simulations that entail continuous and algorithmic behaviours; this can be done stepwise and component-wise, with high-level strategies and laws enabling automation. This separate phase is addressed in the next section.

To give an example, rather than considering the exact position of a train on a railway network, we may record the track segment on which the train is located. Instead of describing its behaviour using equations of motion, we may merely observe that trains change track segment consistently with the railway layout while respecting red signals. This discrete abstraction allows us to formulate and prove safety properties of the system, such as trains do not derail or collide. Yet it manages to avoid the complexity of a hybrid model that additionally aggregates continuous dynamics and reactive and real-time aspects. The latter we introduce via refinement in a compositional manner. We shall explore this example in more detail in Section 5.

The essence of the execution paradigm of FMI co-simulations is that they proceed in lock-step. Data is exchanged only at the beginning of each simulation step, and there is no sharing of data while simulators compute their results. The following pattern captures this design.

$$FMI_{spec} \mathrel{\widehat{=}} Init \;;\; \mu X \bullet (FMU_1 \parallel FMU_2 \parallel \ldots \parallel FMU_n) \;;\; step!w \longrightarrow X$$

Above, *Init* corresponds to some initialisation operation of the co-simulation, and the FMU_i are abstract models of FMU behaviours for $i \in \{1, \ldots, n\}$. We note that all FMUs operate on a shared centralised state σ that includes also simulation time (*time*). *Init* and the FMU_i are relational computations on that state, hence do not communicate or interact with the environment. Communication with the environment is via the channel *step* that allows us to observe the state of the co-simulation while it executes. This ensures that the recursion is guarded and does not diverge. More importantly, it enables an environment to perform validation of the co-simulation behaviour

by communicating on the channel *step* and thereby obtaining the sequence of states that the co-simulation passes through. Both proof and model-checking techniques can thus be supported by our model.

The parallel composition $FMU_1 \parallel FMU_2 \parallel \ldots \parallel FMU_n$ captures a co-simulation step and must have the following properties: (a) FMUs write to disjoint parts of the state space *except for time* which can be modified by all FMUs; and (b) the composition only terminates when all FMUs have completed their computation. Both these properties hold for our definition of concurrency, which is based on parallel-by-merge [21]. This already ensures lock-step progress of simulation cycles. A property of merging parallel behaviours is that any progress of time is possible as long as all FMUs agree on it; otherwise, our model becomes infeasible and is thus not well-formed.

A healthiness condition for FMU specifications is that we can only observe behaviours where time increases, hence time cannot go backwards. This does not preclude master algorithms that perform roll-back, since we consider roll-back as an implementation mechanism for optimised execution. We note that even in such MAs, time goes forward between co-simulation steps.

Our model makes two assumptions. The first one is that simulations do not terminate; the second one is that FMUs can agree on a step-size in each simulation step. While it is the master algorithm that is responsible in a concrete simulation to calculate that step size, in the abstract model it is simply a conjunction of (possibly nondeterministic) timed behaviours. The conjunction emerges from the definition of parallel composition, and where multiple step sizes are possible, may still exhibit nondeterminism.

Reasoning about abstract co-simulations Dealing with a centralised rather than distributed state considerably simplifies reasoning about co-simulation models. The technique we use here is invariant-based reasoning. If we need to show some safety property S of a co-simulation model, expressed as a property of the centralised state σ, our aim is to find an invariant I implying S that is preserved by each co-simulation step.

Since the FMU models are purely relational, Hoare logic is sufficient for proof. Having identified a suitable invariant I of the system, our aim is to prove that both the initialisation and parallel step operations preserve it:

$$\{true\}Init\{I\} \tag{2}$$

$$\{I\}FMU_1 \parallel FMU_2 \parallel \ldots \parallel FMU_n\{I\} \tag{3}$$

Via refinement laws for recursion and reactive contracts, we can then establish that all observations w communicated through the channel *step* must

satisfy the invariant I. Formally, this is expressed as the refinement of a reactive contract whose precondition is *true* and whose postcondition is *false*, namely because our FMI model neither diverges nor terminates.

$$[true \mid (\forall\, evt \in tt \bullet \exists\, w{::}\sigma \bullet evt = step.w \wedge I) \diamond false] \sqsubseteq FMU_{spec}$$

A specialised law is used to prove the above refinement from the earlier provisos (2) and (3). The pericondition of the reactive contract effectively states that all events evt of the contributed trace tt must have the form $step.w$ for some w that satisfies the invariant I. Monotonicity of reactive-contracts *in the pericondition* with respect to refinement implies that the above refinement still holds if we replace I by the safety property S, given that $S \sqsubseteq I$. (We recall that the definition of refinement for plain predicates is (universal) reverse implication; see Section 2.3.)

Our key observation here is that the above gives us a proof strategy to reason about co-simulations as minimal reactive models, namely that only require a single channel *step*. The essence of such reasoning is to find an invariant I that implies the safety property S, and to prove provisos (2) and (3). The proofs can be done in the simplified context of classical Hoare logic, rather than some reactive and hybrid language and semantic calculus.

The proofs may nonetheless profit from ways of modularising the effort by decomposing I into component invariants I_1, I_2, and so on, that hold for particular FMUs, as well as a shared invariant I_s that relates the state of all FMUs, such that $I \Leftrightarrow (\forall\, i \bullet I_i) \wedge I_s$. We shall not examine those strategies in more detail here but revert our attention to this in Section 5.

We next examine the transformation of the abstract FMI co-simulation model into a concrete one.

3.3 Refinement Strategy

A concrete behavioural model of FMI co-simulations is proposed in [7], and we have mechanised that model in INTO-CPS Deliverable D2.3c. The purpose of refinement here is hence to transform an abstract model of an FMI co-simulation into a concrete one by virtue of a series of correctness-preserving model transformations. Such transformations are typically formalised by refinement laws. Refinement may proceed piece-wise, meaning we can examine the transformation of components (such as the FMU_i) in isolation.

Refinement is often motivated by subdividing and modularising the effort involved in verifying a complex system. Refinement laws only need to be proved

once, while they can be applied in combination using high-level strategies to yield a wide spectrum of possible implementation designs from a single specification. Here, the issues that refinement must address are:

1. Transforming abstract specifications of FMUs (the FMU_i models) into concrete implementations FMU_i^c that introduce (continuous) dynamics and, where applicable, algorithmic behaviours.
2. Introduce a design corresponding to the reactive model of an FMI master algorithm, as detailed in INTO-CPS Deliverable D2.3c.

The first refinement step (1) yields a model of the form:

$$FMI_{ref} \mathrel{\widehat{=}} Init^c \; ; \; \mu X \bullet (FMU_1^c \parallel FMU_2^c \parallel \ldots \parallel FMU_n^c) \; ; \; step!w \longrightarrow X$$

where each abstract FMU_i specification has been replaced by a concrete implementation FMU_i^c such that $FMU_i \sqsubseteq FMU_i^c$ for $i \in \{1, \ldots, n\}$. The compositionality principle then justifies the claim that $FMI_{spec} \sqsubseteq FMI_{ref}$. For continuous components, this step involves data refinement [8] in which additional state components for continuous variables are introduced into the model, such as the position, velocity and acceleration of a train. The concrete variables are linked to abstract variables via *gluing invariants* — a technique well understood and used in refinement-based verification techniques and tools [1, 29]. For instance, a gluing invariant may link a train's precise physical position to its abstract location on a track segment.

For FMUs that carry out some discrete computational behaviour, we may use both operational and data refinement to elicit the design of the underlying algorithm. In either case, the refinement may involve a change in language: for instance, the implementation model may be expressed not just using operators for plain relational sequential programs, but timed languages and the Hybrid Relational Calculus (HRC) [14]. This requires laws that justify an *embedding* of such models into a simple relational computation where time has been promoted to a (state) variable of the co-simulation. The issue of linking theories is addressed in detail in INTO-CPS Deliverable D2.3d.

The second stage (2) of the refinement is the introduction of the FMI co-simulation design and master algorithm. The key concern here is to distribute the centralised state into FMUs while establishing communication patterns that allow FMUs to exchange data between simulation steps. This is realised by high-level laws for *Circus* [6] that, for instance, introduce channels corresponding to the `fmi2Get` and `fmi2Set` methods of the FMI API that enable MAs to obtain outputs and set inputs of FMUs. The master algorithm emerges from such refinements by applying laws that encapsulate

the shared data transmitted between FMUs into a single process. Laws that are useful for this have, for instance, already been proposed in [38].

We note that although stage (2) of the refinement appears tedious, there is overall more potential for automating it via tactics than stage (1). This is so because FMI_{ref} above is already a sufficiently concrete target to guide subsequent refinements, if, for instance, additional information is given mapping state components to FMUs and ports. Such information can be provided by engineers who, crucially, do not have to be experts in formal techniques.

3.4 Final Considerations

We have presented techniques for analysing co-simulation models, targeting both architectural well-formedness and abstract behavioural models. Our key observations are that proofs about architectural properties are feasible to fully automate in Isabelle/HOL, even for large-scale models. This is thanks to Isabelle's open architecture and support via tactics that exploit code generation. Behavioural models are much more challenging to analyse, and for those we have presented an abstract view that reduces complexity to reason about relational computations that include time as a state component: they neither need to be concerned with continuous dynamics, nor with reactive behaviours. MAs emerge through refinement.

In addition, we outlined a clear path to moving from an abstract to a concrete FMI co-simulation model via refinement.This approach shares similarities with techniques such as [38] and can profit from laws about reactive processes and *Circus* [6] that have already been proved elsewhere.

The focus of our contribution in this deliverable is to formulate and prove properties of the abstract model of a co-simulation.

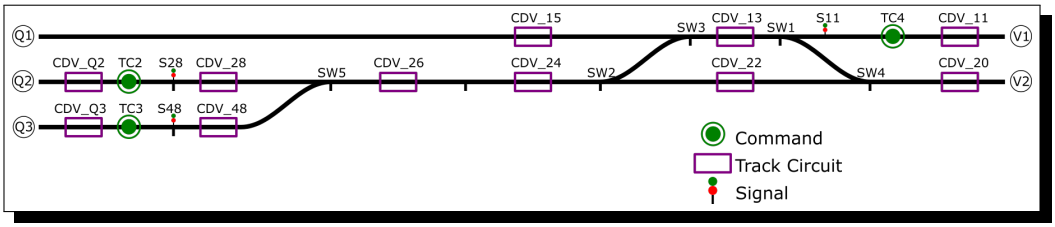

Figure 2: Railway interlocking layout of our case study.

4 Railways Case Study

This section presents an overview of the railways case study that we use to illustrate our technique for reasoning about FMI co-simulation models in Section 5. We briefly describe all relevant models in what follows.

4.1 Introduction

Our case study considers part of a tramway station. Its railway interlocking layout is presented in the diagram of Fig. 2. Trains enter the interlocking at the points V1, Q2 and Q3, and then issue a telecommand to request a route. Telecommand stations are denoted by the three green dots, and the possible routes for trains are V1 → Q1, V1 → Q2, V1 → Q3, Q2 → V2 and Q3 → V2. We consider scenarios where two trains arrive at different entry points and request a route.

Access to the interlocking is controlled by the 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 signal 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 controller is in essence an automaton whose functional behaviour is defined in *relay ladder logic* (RLL) [25]. Ladder models are graphical representations tailored for relay-based hardware implementations. The inputs of the interlocking controller are boolean vectors for the *chemin de*

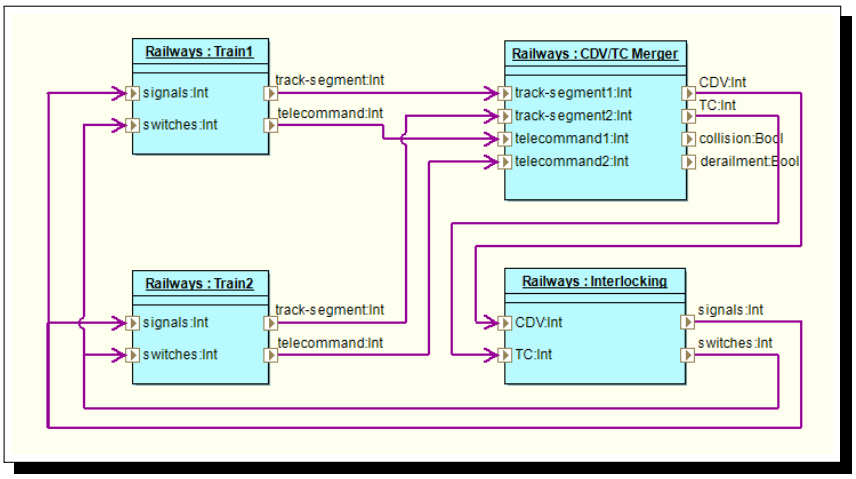

Figure 3: FMI co-simulation architecture for the railways system.

voie (CDV) and telecommand. The CDV 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.

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 telecommand 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.

4.2 System Overview

A high-level view of the system as a co-simulation architecture is given in Fig. 3. Altogether, there are four FMU components. Two of them, Train1 and Train2, simulate the physical behaviour of both trains, which includes the action of the train driver in setting 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 both 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

Deliverable D1.3, define the train physics and their control behaviour as bond diagrams in 20-sim. The VDM-RT model of the interlocking was automatically generated from the ladder production code of the software controller. Two train models of increasing complexity exist: one that assumes trains follow a predetermined fixed route, and one that moves trains according to the setting of railway switches.

With regards to 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 20-sim train model has moreover been encoded in Modelica, for which we have a formal semantics mechanised in Isabelle/UTP [13]; for details of the semantics and embedding we refer to the INTO-CPS Deliverable D2.3b.

We have manually 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.

4.3 Behavioural Models

We next describe the behavioural model of each FMU component of the co-simulation: the Modelica train model in Section 4.3.1, the VDM-RT interlocking model in Section 4.3.2, and the CDV/TC Merger in Section 4.3.3.

4.3.1 Train Models in Modelica

Kinematics and speed control of both trains is encoded by the equations in Fig. 4. 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, given by the `position_on_track` variable. 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 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

```
/* 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 4: Train control equations in Modelica.

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. 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 towards a halt.

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 (see Fig. 5). 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.

```
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;
```

Figure 5: Modelica algorithm for calculating the train's set-point speed.

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.

In the model where trains follow a pre-determined route, we have the following single equation for train location:

```
equation
  (track_segment, position_on_track) = FollowRoute(fixed_route, total_position);
```

An additional variable `total_position` records the total distance travelled by the train. Based on the fixed route (constant `fixed_route`), we can infer both the train's track segment and relative position on the track, using the `FollowRoute(_)` function whose definition we omit.

In addition to the above, we also have an equation that generates the telecom-

mand signal when trains traverse tracks equipped with a telecommand station. We omit a discussion of its straightforward definition here, too, referring to Appendix A for the complete Modelica train model.

In summary, our Modelica model of the case study collaterally achieved unification of the simple and full train model of the original 20-sim control law. We also managed to factor out the railway topology into a collection of model constants, most of them being tables and maps. This makes our model potentially applicable to arbitrary architectures. For experimentation, we defined a third train model with a minimal topology, containing only three linear tracks and one signal; it turned out useful to study our analysis approach and we call this the 'three-tracks' model.

4.3.2 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 activated (locked). A fundamental safety property is that two different routes can only be activated simultaneously if their paths do not intersect. Moreover, signals and point switches have to be set consistently with the activated routes at any given time.

To capture the core algorithmic behaviour of the interlocking system, 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 sequential tasks:

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

The sequential program logic that performs the locking of routes (task 1) is included in Fig. 6. We note that `hwi` is a VDM++ 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'

```
-- 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 6: Extract form the VDM-RT algorithm for locking routes.

```
/* Relay Deactivation */
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;
```

Figure 7: VDM-RT algorithm for resetting routes.

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. Where there is contention of two trains requesting intersecting routes, the sequential program logic ensures that one of them is given precedence to proceed, while the other has to wait.

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 the code block in Fig. 7. The conditions $\neg$ `hwi.CDV(i)` here are used to determine that a train has reached the last track segment `i` of a route. We note that `false` signifies presence rather than absence of a train.

Tasks (3) and (4) deal with the positioning of railway switches and the setting of signals. The VDM-RT code in Fig. 8, for instance, carries out the positioning of switches, based on the relay configuration. We declare a VDM type `SWITCH_POSITION` for the possible states of a railway switch. Such can be either `STRAIGHT` or `DIVERGING`. The VDM-RT code for setting signals is similar. Each activated route causes the enabling of one of the three signals. Since there are five routes, three of them cause the same signal to be activated. As an additional safety measure, before flagging a green light, we moreover check that switches have been aligned correctly to prevent derailment. We omit the programme code for signal setting (task 4) but refer to Appendix B for a complete model of the interlocking controller.

While our software implementation retains the core logic of the hardware

```
/* 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>;
```

Figure 8: VDM-RT algorithm for positioning railways switches.

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.

4.3.3 CDV/TC Merger Model

Our co-simulation design requires a separate FMU that generates the combined CDV and telecommand vectors of both trains. These are inferred from the current track segment and telecommand requests of each individual train. The relevant equation for the CDV signal is presented below.

```
/* Calculates the CDV[11] signal from the track segment of each train. */
equation
  for i in 1:11 loop
    CDV[i] = not (i == train1_segment or i == train2_segment);
  end for;
```

Once again, the negation reflects that track segments are occupied when their CDV value is `false`. The computation of the TC signal is very similar. The `for`-loop yields a conjunction of all iterated behaviours.

An important supplementary task of the CDV/TC Merger FMU is to generate signals that check whether safety requirements are met. These signals here are `collision` and `derailment`; they are both boolean outputs of the model, and their definition is recaptured below.

```
equation
  collision =
    train1_segment > 0 and
    train2_segment > 0 and train1_segment == train2_segment;
  derailment = train1_segment == -1 or train2_segment == -1;
```

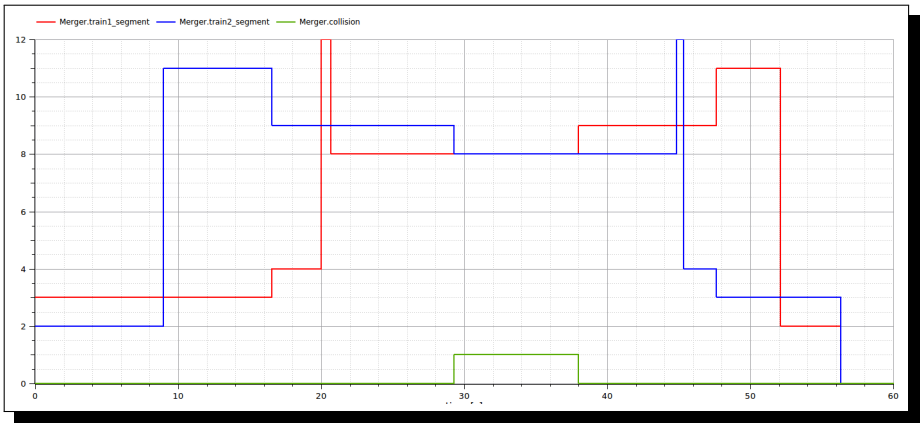

Figure 9: Co-simulation of the railways system after injecting a fault.

To explain this equation, we note that the track segment of a train, given by `train1_segment` and `train2_segment`, under normal conditions takes values in the range 1 to 13 — that is, when the train is within the boundaries of the interlocking. A track segment becomes 0 when the train leaves the interlocking, and -1 when it derails due to crossing a badly-aligned switch.

As part of INTO-CPS Deliverable D1.3, a working co-simulation of our model has been produced. A simulation scenario is demonstrated in Fig. 9, where a fault has been injected into the interlocking controller. The blue and read lines illustrate the movement of trains on the railway tracks; the green light probes the `collision` signal of the CDV/TC Merger FMU and thus indicates that a train collision occurs on track segment 8.

4.4 Final Considerations

We have presented a model of the INTO-CPS railways case study as a basis for our analysis technique. This involved, in particular, reformulation of the train model in Modelica and extracting the core logic of the VDM-RT software controller. While our model can be simulated (see Fig. 9), co-simulation does not provide universal guarantees that a violation of safety properties can never occur, depending on the parameters of the model such as initial track and train speed, acceleration and deceleration, and so on. In the next section, we examine a formalisation of the co-simulation model in Isabelle/UTP that allows for such proofs.

```
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 10: Instantiation of the railways FMI architecture in Isabelle/HOL.

5 Mechanisation in Isabelle/UTP

In this section, we describe our mechanisation of the railways co-simulation, as presented in detail in the previous section. This includes the FMI architecture (Section 5.1) and behavioural model (Section 5.2). Property verification, including safety requirements, is the subject of Section 5.3, and in Section 5.4 we conclude with some final considerations on the mechanisation work.

5.1 Railways Architecture

We have already presented our abstract and general model of FMI co-simulation architectures in Section 3.1. There, we introduced a given type `FMI2COMP` for FMU components, and several uninterpreted constants (see Fig. 1). For instantiation with a concrete model, we next show how we endow both of these with concrete values for our railways example.

First of all, to instantiate the abstract type `FMI2COMP`, we make use of an Isabelle/HOL `axiomatization` that introduces concrete symbolic constants for each of the FMUs. Here, we give them the names `train1`, `train2`, `merger` and `interlocking`. Fig. 10 includes the corresponding mechanisation fragment. Our assumptions are added: the first one, `fmus_distinct`, captures that the FMU identifiers are distinct; and the second one, `FMI2COMP_def`, that they precisely define the values (carrier set) of the `FMI2COMP` type. In other words, there are no values other than those four. We note that the built-in HOL function `distinct` l asserts that a list l is injective.

From the assumption `fmus_distinct`, we moreover prove a set of corollaries that establish various inequalities between FMU identifiers, such as, for instance, `train1` $\neq$ `train2`, and so on. These corollaries are used in the well-formedness proofs and tactics we discuss later on.

We note that Isabelle `axiomatization`s 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. 3) 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 system.

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

The well-formedness caveat `range FMUs` = `FMI2COMP` can be easily discharged, using unfolding of definitions and simplification alone.

Simulation `parameters` are of type `FMI2COMP` × `VAR` × `VAL`. To encode the type `VAR`, which, as mentioned above, represents variables as name/type records, we represent names as HOL `string`s, and types as elements of an Isabelle type `typerep`. Fundamentally, HOL's logic and type system cannot fully support types as values; a limited facility exists in Isabelle by way of a (monomorphic) datatype `typerep` that records the structure of any HOL type. Making use of it here makes our model extensible in terms of the port types that can be supported.

Concrete parameters of the railways co-simulation are the routes of the trains and their maximum speed. Fig. 11 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 [41] 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 [3].

Next, we consider the input and output ports of the control diagram in Fig. 3. We recall that ports are modelled by pairs of type `FMI2COMP` × `VAR`.

```
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 11: Parameters definition of the railways instantiation.

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. In both cases, we are using sequences rather than sets. As mentioned in Section 3.1, this is to (a) ensure finiteness by construction, and (b) to provide an easy way to formulate iterative operations over inputs and outputs.

For each input port in the list `inputs`, an initial value has to be provided. We manually extract such values from the Modelica train model and VDM-RT interlocking model, and encode them via `initialValues`, namely as a list of elements of type `ports` × `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. 11, we omit it here referring to [39] for details.

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. 1), we represent these functions by their finite graph: that is, as a relation of type (`port` × `port`) `list`. The reason

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

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, using the following recursive definition:

```
fun fun_of_rel :: "('a × 'b) list ⇒ ('a ⇒ 'b set)" where
"fun_of_rel []           x = {}" |
"fun_of_rel ((a, b) # t) x = (if x = a then {b} else {}) ∪ (fun_of_rel t x)"
```

While functional representations are more useful in the behavioural model of master algorithms (see INTO-CPS Deliverable D2.3c), 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. 12. 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. Althought `pdg` was encoded by hand here, its construction can fundamentally be automated from the Connection Diagram of the INTO-SysML model.

A direct feed-through dependency arises in an FMU if some output is driven with zero latency by some input, without delays or integrates in the signal path. We record such dependencies for all FMUs via the constant `idd`, whose type, as mentioned, is the same as the one of `pdg`. A structural difference between `pdg` and `idd` is that maplets in `pdg` connect different FMUs, hence are of the shape $((fmu_1, out), (fmu_2, in))$ with $fmu_1 \neq fmu_2$, whereas maplets in `idd` model connections within the same FMU, hence must be of the shape $((fmu, in), (fmu, out))$. Direct dependencies are generally determined by examining the particular FMU model. For control laws such as 20-sim models, it is possible to automate their detection using signal flow analysis.

In the reminder of this section, we discuss our mechanised proof of well-formedness of the instantiated railways co-simulation architecture.

Proof of well-formedness For well-formedness of the instantiation, we have to discharge all of the constraints in Fig. 1. Hence, we first verify that

there is only a single parameter value for each FMU and variable tuple within the `parameters` list. This amounts to showing that:

```
"(∀(f, x, v)∈set parameters. ¬ (∃v'. (f, x, v')∈set parameters ∧ v ≠ v'))"
```

The function `set` is used above to obtain the elements of a list. Because the `parameters` list is typically short, automatic proof via Isabelle's `auto` tactic is normally sufficient to discharge this conjecture efficiently.

Next, we consider the input and output ports of the architectural diagram. One caveat is that the lists `inputs` and `outputs` have to be injective. This is formally captured by the predicates `distinct inputs` and `distinct outputs`. Because port lists may be large, we automate proof here by virtue of a custom tactic that uses a collection of bespoke rewrite laws. This provides a notable speed-up in comparison to `auto` that appears to become more significant as models grow in the number of FMUs and port connections.

Another caveat is that `inputs` and `outputs` have to be disjoint. Again, a specialised tactic aids to carry out this proof efficiently by pre-selecting appropriate rewrite laws. We note that the use of small sets of engineered rewrite laws has throughout proved useful to automate and speed-up proofs about architectural properties. We lastly also use it to show that the list `initialValues` contains precisely one entry for each input port in `inputs`.

In summary, well-definedness properties of the constants `FMUs`, `parameters`, `initialValues`, `inputs` and `outputs` can be efficiently discharged by virtue of system-logic tactics — ones that do not have to resort to evaluation via code generation. Those tactics, we have specified using the Eisbach language and tool [26]; their definition can be found in the Isabelle report [39] too.

What remains to be shown is well-formedness of the port dependency graph `pdg` and direct dependency relation `idd`. An easy property to verify is type conformance of port connections: outputs are connected to input ports of the same FMI type. Conformance is specified via an inductive predicate `coformant` over lists of `PORT` pairs as follows:

```
inductive conformant :: "(PORT × PORT) list ⇒ bool" where
"conformant []" |
"type (name p1) = type (name p2) ⟹
  conformant l ⟹ conformant ((p1, p2) # l)"
```

It captures that for each element (`p1`, `p2`), the port types `p1` and `p2` must be the same. The proof effort here is linear in the size of the `pdg` relation, hence feasible to automate via HOL laws and tactics as before. More challenging is to establish the absence of algebraic loops in Isabelle. As mentioned before,

this amounts to showing that the relation $(\texttt{set}\,\texttt{pdg}) \cup (\texttt{set}\,\texttt{idd})$ is acyclic. While we can easily formulate this as a conjecture, using the built-in predicate $\texttt{acyclic}\,r$, its proof is beyond the capabilities of the `auto` tactic.

As explained in Section 3.1, we propose two complementary approaches to achieve automation nonetheless. Firstly, Isabelle's `eval` tactic is able to evaluate $\texttt{acyclic}\,r$ for enumerated relations r. Verified ML code is thus used to compute the transitive closure of r and check that it is irreflexive. Although the algorithm used for this is not optimised, it executes sufficiently fast for our railways example to discharge the caveat for acyclicity within one second. Imminent downsides are that (a) this approach does not scale well for larger models, and (b) that we have to trust the Isabelle code generator.

To overcome both ramifications, we implemented our alternate solution: instead of calculating the transitive closure within Isabelle, we outsource that computation to an external and optimised C++ algorithm. Since we did not verify that algorithm itself, trust is established by validating the algorithm's result on a per invocation basis. For larger models, proving the assumptions used is overall a less complex task than calculating the closure 'from scratch' by way of evaluation tactics.

In conclusion, well-formedness of the railways architecture can be established fully automatically in Isabelle, with the definitions and proofs requiring approximately 12 seconds of processing time. Towards scalability, we have presented various solutions to deal with the complexity arising in larger models. A major contribution is an external C++ tool that interfaces with Isabelle to outsource the computation of transitive closures, while supporting multiple efficient algorithms such as [35]; the tool is part of Isabelle/UTP [15].

In the next section, we specify the behavioural model of the railways co-simulation to enable analysis and proof of behavioural properties.

5.2 Abstract FMI Model

We begin by specifying the centralised state space of the abstract FMI model. For this, we use the `alphabet` command of Isabelle/UTP in order to introduce the necessary variables along with their types. Some of these variables correspond to state components of FMUs, while others represent shared data between FMUs. Conceptually, we consider variables for sharing data as part of the state of the FMU that outputs (writes) the data.

Fig. 13 presents the FMU state of the railways system as we have specified

```
alphabet railways_state =
  (* Train FMUs *)
  current_track₁ :: "int"
  current_track₂ :: "int"
  telecommand₁ :: "bool vector"
  telecommand₂ :: "bool vector"
  (* Merger FMU *)
  cdv :: "bool vector"
  tc :: "bool vector"
  (* Interlocking FMU *)
  relays :: "bool vector"
  signals :: "bool vector"
  switches :: "switch vector"
  (* Simulation Time *)
  time :: "TIME"
```

Figure 13: Definition of the centralised FMU state.

it in Isabelle/UTP. The train state is given by the variables `current_track`$_1$ and `current_track`$_2$ of type `int`eger, recording the track segment on which the trains are currently positioned. These variables moreover fulfil the dual purpose of being outputs of the train FMUs. We recall that we do not record precise physical positions of the trains at this level of abstraction yet, but use discrete approximations of them. The variables `telecommand`$_1$ and `telecommand`$_2$ are two further outputs of the train FMUs that generated the required telecommands for the interlocking. We make use of a type `'a vector` to specify their values. That type is part of Isabelle/UTP to encode fixed-size vectors over some type HOL `'a` (here `bool`).

As already noted in Section 4, train locations are recorded as integers whose values are in the range 1–13 whenever the train is within the boundaries of the interlocking (there are 13 track segments). Locations may besides take a special value 0 when the train has left the interlocking, or -1 when it has derailed due to crossing a wrongly aligned railway switch.

The merger FMU does not have a state: its sole purpose is to combine the `current_track`$_{[1/2]}$ and `telecommand`$_{[1/2]}$ input signals, to yield the respective vectors `cdv` and `tc` as its outputs, read by the interlocking.

The interlocking FMU has the state component `relays` to record the state of relays for route locking. Its outputs are `signals` and `switches`, both of which feed back into the train FMUs to control the trains' speed and determine their path, which is affected by the configuration of railways switches.

Instantiating the FMI specification pattern described in Section 3.2 yields

the following reactive computation for the railways example.

$$Init\ ;\ \left(\ \mu X \bullet \left(\ \left(\begin{array}{l} \{\texttt{current\_track}_1, \texttt{telecommand}_1\ \texttt{time}\}\ \mathit{Train}_1 \\ \|\ \{\texttt{current\_track}_2, \texttt{telecommand}_2, \texttt{time}\}\ \mathit{Train}_2 \\ \|\ \{\texttt{cdv}, \texttt{tc}, \texttt{collisions}, \texttt{derailment}, \texttt{time}\}\ \mathit{Merger} \\ \|\ \{\texttt{signals}, \texttt{switches}, \texttt{time}\}\ \mathit{Interlocking} \end{array}\right)\ ;\ \mathit{step}!(w{::}\texttt{railways\_state}) \longrightarrow X\ \right)\ \right)$$

Each parallel computation (corresponding to one of the FMUs) additionally must specify the set of variables that it writes to. The only variable that all computations jointly modify is `time`, modelling simulation time. Constraints on the `time` variable are imposed by individual FMUs to reflect the simulation step size they are willing to admit.

In the remainder of the section, we examine the behavioural model of the trains, merger, and interlocking FMUs in detail.

5.2.1 Train FMU Model

Abstractly, trains are modelled by a nondeterministic computation: in each simulation step, either the train location remains the same, or otherwise the train moves to the next track segment. The assumption our abstraction thus makes is that simulation steps are sufficiently small so that we cannot observe trains advancing more than on track (we note that this is an issue for refinement to show). To formalise the train model, we firstly specify two HOL functions `NextTrackQV` and `NextTrackVQ`. These determine the subsequent track segments of a moving train, moving either left-to-right (`QV`) or right-to-left (`VQ`). The Isabelle definition of `NextTrackQV` is given in Fig. 14. Elements $n := k$ there are the maplets of a function that associates each track segment $n \in \{1\,..\,13\}$ with a follow-up track segment k, while considering the setting of switches (`sw`). We note that this function is in direct correspondence to a constant in the Modelica model that encodes the track layout.

Another utility function we define is `MustWait`. It determines when a train must wait due to a red signal:

```
definition MustWait :: "bool vector ⇒ int ⇒ bool" where
"MustWait sig track ⟷
  (track = 1 ∧ sig[1]) ∨ (track = 2 ∧ sig[2]) ∨ (track = 3 ∧ sig[3])"
```

The parameters of the function are the state of signals (`sig`) and the current track segment (`track`). There exist only three tracks with signals, hence this function can be defined by a simple boolean expression. We assume that

```
definition NextTrackQV :: "switch vector ⇒ int ⇒ int" where
"NextTrackQV sw = undefined (
   1 := 10,
   2 := 11,
   3 := 0,
   4 := if sw[1] = STRAIGHT then 3 else 13,
   5 := if sw[3] = STRAIGHT then 4 else -1,
   6 := 0,
   7 := if sw[4] = STRAIGHT then 6 else -1,
   8 := if sw[2] = STRAIGHT then 7 else 12,
   9 := 8,
   10 := if sw[5] = STRAIGHT then 9  else -1,
   11 := if sw[5] = DIVERGING then 9 else -1,
   12 := if sw[3] = DIVERGING then 4 else -1,
   13 := if sw[4] = DIVERGING then 6 else -1
)"
```

Figure 14: HOL function that encapsulates the track layout.

```
"MoveTrain (track, dir, sig, sw) =
  (if track =  0 then  0 else
  (if track = -1 then -1 else
  (if MustWait sig track then track
   else (case dir of
    QtoV ⇒ NextTrackQV sw track |
    VtoQ ⇒ NextTrackVQ sw track))))"
```

Figure 15: HOL carries out the movement of trains.

when a track is not guarded by a signal, trains are permitted to proceed. The function thus succinctly characterises the behaviour of the train driver.

To complete our train model, the function `MoveTrain` included in Fig. 15 ties together both functions discussed above. It takes the track segment (`track`), direction of train travel (`dir`), and the state of signals (`sig`) and switches (`sw`) as arguments. Its definition caters for the case that the train may have to wait, and also considers the case that the train has either left the interlocking (first conditional) or derailed (second conditional). The values `QtoV` and `VtoQ` are introduced by a datatype declaration for train directions.

We lastly define a UTP relational computation that corresponds to the behavioural model of the train FMU, here for the first train:

```
definition Train1 :: "TIME ⇒ railways_state hrel" where
"Train1 ε = (
  ($current_track₁´ =ᵤ $current_track₁ ∨
   $current_track₁´ =ᵤ MoveTrainᵤ($current_track₁, «dir₁», $signals, $switches)ₐ)
  ∧ $telecommand₁´ =ᵤ ... ∧ $time´ ≤ᵤ $time + «ε»)"
```

The epsilon here specifies the largest step size that the train FMU is willing to perform. Such can be predetermined from the maximum speed of the train and minimum track length, to make later refinement possible. The disjunction represents the nondeterministic composition of two computations, one keeping the value of $\texttt{current\_track}_1$ and one changing it. The formula calculating the value of $\texttt{telecommand}_1$ is omitted for brevity.

An analogue definition is provided for the second train. We next turn our attention to the encoding of the merger FMU.

5.2.2 Merger FMU Model

In each simulation step, the merger processes the $\texttt{current\_track}_{[1/2]}$ and $\texttt{telecommand}_{[1/2]}$ signals, and produces bit vectors for the `cdv` and `tc` outputs. For this, we encode the Modelica equation in Section 4.3.3 into a HOL function `MakeCDV` that takes two track segments `i` and `j`:

```
fun MakeCDV :: "int × int ⇒ (bool vector)" where
"MakeCDV (i, j) = (MergeCDV i (MergeCDV j (mk_vector 13 True)))"
```

This function uses a utility operation `MergeCDV` whose purpose consists of setting the `n`-th element in a vector to `False`, provided $\texttt{n} \in \{1\,..\,13\}$. Above, the function `mk_vector` constructs an initial vector of size 13, with all elements set to `True` (we recall that `False` signifies a track being occupied).

With the above, we define the relational program that corresponds to the Merger FMU as follows:

```
definition Merger :: "railways_state hrel" where
"Merger = ($cdv´ =u MakeCDVu($current_track1, $current_track2)a ∧ $tc´ =u ...)"
```

There are no constraints on `time` here since the merger admits any simulation step size. The computation of the `tc` vector has been omitted as it follows the same principle, albeit using the inputs $\texttt{telecommand}_1$ and $\texttt{telecommand}_2$.

To conclude the encoding of FMUs, we lastly consider the interlocking.

5.2.3 Interlocking FMU Model

The interlocking is from a computational point of view the most interesting FMU. In Section 4.3.2, we have already examined its cyclic behaviour. We recall that such consisted of (a) setting relays, (b) clearing relays, (c) aligning

```
definition set_switches :: "railways_state hrel" where [urel_defs]:
"set_switches = (
  (switches[1] := «STRAIGHT») ;;
  ((switches[2] := «STRAIGHT») ◁ R3 ∨ R5 ▷ᵣ (switches[2] := «DIVERGING»)) ;;
  ((switches[3] := «STRAIGHT») ◁ R1 ▷ᵣ (switches[3] := «DIVERGING»)) ;;
  (switches[4] := «STRAIGHT») ;;
  ((switches[5] := «STRAIGHT») ◁ R2 ∨ R3 ▷ᵣ (switches[5] := «DIVERGING»)))"
```

Figure 16: Relational computation for setting railways switches.

railway switches, and (d) setting signals. Our semantic model for VDM-RT (see INTO-CPS Deliverable D2.2b) allows for a direct translation of the respective sequential code fragments into Isabelle/UTP.

An example of encoding that considers task (c) for setting railways switches is presented in Fig. 16. It encodes the VDM-RT instructions in Fig. 8. We note that the conditional statements are translated using UTP's infix notation $P \lhd b \rhd Q$, and for accessing the elements of the `relays` state component, we introduce the syntactic abbreviation `R1-R5`. Less these notational changes, the code is in direct correspondence to the VDM-RT model and profits from our framework directly supporting imperative languages.

While tasks (a), (b) and (d) have analogue definitions and are, therefore, not discussed further here, a point of interest is to specify permissible changes to the `time` variable. As the interlocking is implemented using a cyclic executive (periodic thread), time increases by a fixed amount in each simulation step, determined by the period of the thread. We can deduce that period by looking at the VDM-RT program. A further step (e) of the sequential algorithm is hence to increment time accordingly, and this is achieved by a statement $\texttt{time} := \texttt{time} + \mathit{period}$.

This concludes our account of the FMI model of the railways example. We note that the full architectural and abstract co-simulation model can be found as part of the Isabelle/UTP distribution [15]. We are ready now to examine proof-based analysis of the mechanised railways co-simulation.

5.3 Property Verification

For analysis and verification, we prove relevant invariant properties of the abstract co-simulation model. As explained in Section 3.2, these properties can subsequently be lifted to reactive contracts that constrain observable behaviours of co-simulation steps by an environment.

```
definition railways_type_inv :: "railways_state upred" where
"railways_type_inv = (
  &current_track₁ ∈ᵤ «{-1..11}» ∧ #ᵤ(&telecommand₁) =ᵤ 4 ∧
  &current_track₂ ∈ᵤ «{-1..11}» ∧ #ᵤ(&telecommand₂) =ᵤ 4 ∧
  #ᵤ(&cdv) =ᵤ 11 ∧ #ᵤ(&tc) =ᵤ 4 ∧
  #ᵤ(&relays) =ᵤ 5 ∧ #ᵤ(&signals) =ᵤ 3 ∧ #ᵤ(&switches) =ᵤ 5)"
```

Figure 17: Global typing invariant of the centralised railways model.

We distinguish *local invariants* of FMUs that can be proved in isolation, and *shared invariants* that require the consideration of multiple FMUs at the same time. Our proof strategy first consists of identifying relevant local invariants of each FMU and proving that the respective FMU preserves them, using our mechanisation of Hoare logic in Isabelle/UTP.

A particular kind of local invariant concerns typing of state components. For instance, train locations, given by the state component $\texttt{current\_track}_{[1/2]}$, must either be in the range 1-13, or otherwise in the set $\{0, -1\}$. We proved that train FMUs satisfy this invariant. Similar type invariants have also been checked for the other FMUs, and their conjunction implies the global typing invariant of the system included in Fig. 17.

The typing invariant in Fig. 17 is indeed also a shared invariant, since we have to consider the behaviour of all FMUs simultaneously to establish it. The proof, however, can be carried out in a modular fashion, since it is possible to decompose this global invariant into conjuncts I_1, I_2, and so on, that are local invariants of the respective FMUs.

The above observation leads us to further distinguish shared invariants into decomposable and non-decomposable (holistic) ones. The former are easier to prove since they facilitate a compositional approach in which a property proved for individual FMUs can be lifted to a property of their parallel composition, by virtue of the (basic) Hoare-logic rules for parallelism. Namely, all that needs to be shown for $\{I_s\}FMU_1 \parallel FMU_2 \parallel \ldots \parallel FMU_n\{I_s\}$ is that $\{I_i\}FMU_1\{I_i\}$ for $1 \leq i \leq n$, supposing that $I_s \Leftrightarrow I_1 \wedge I_2 \wedge \ldots \wedge I_n$.

Proof of the invariant property in Fig. 17 provides a first validation of our model that prevents, for instance, undefined terms from arising. For instance, it guarantees that vectors have the expected size, so that indexed access is always well defined. A more interesting local invariant constrains the possible settings of relays by the interlocking FMU. At any point in time, relays must be set so that the routes they enable do not cross each other. This is a fundamental property that reduces the possible 2^5 relay combinations to

```
definition relays_excl_inv :: "railways_state upred" where
"relays_excl_inv = (
  (#u(&relays) =u 5) ∧
  (R1 ⇒ ¬ R2 ∧ ¬ R4) ∧
  (R2 ⇒ ¬ R1 ∧ ¬ R3 ∧ ¬ R4 ∧ ¬ R5) ∧
  (R4 ⇒ ¬ R1 ∧ ¬ R2 ∧ ¬ R3 ∧ ¬ R5) ∧
  (R3 ⇒ ¬ R2 ∧ ¬ R4 ∧ ¬ R5) ∧
  (R5 ⇒ ¬ R2 ∧ ¬ R3 ∧ ¬ R4))"
```

Figure 18: Local (behavioural) exclusion invariant of the interlocking.

only 10 valid ones that may occur during execution.

The encoding of the above relay exclusion property is presented in Fig. 18. We remind the reader that `R1-R5` are abbreviations for indexed access of the `relays` vector, belonging to the state of the interlocking. We mechanically verified that the interlocking FMU preserves this local invariant. The proof was aided by tactics for Hoare-logic reasoning about invariants.

We summarise that proofs of local and decomposable shared invariants admit a modular approach that reduces the proof effort by focusing on FMUs in isolation. However, holistic properties such as safety requirements are inherently non-decomposable; we discuss how to tackle them next.

Holistic Invariants Holistic invariants relate the state components of multiple FMUs and, importantly, cannot be proved compositionally. Safety requirements such as non-collision and non-derailment are such properties.

By way of example, we encode the safety requirements of the railways co-simulation as follows.

```
definition safety_req :: "railways_state upred" where [upred_defs]:
"safety_req =
  (¬ derailed1 ∧ ¬ derailed2 ∧
  (present1 ∧ present2 ⇒ &current_track1 ≠u &current_track2))"
```

Above, $\texttt{present}_1$ and $\texttt{present}_2$ abbreviate the predicates $\texttt{current\_track}_1 \neq 0$ and $\texttt{current\_track}_2 \neq 0$. Analogously, $\texttt{derailed}_1$ and $\texttt{derailed}_2$ abbreviated the predicates $\texttt{current\_track}_1 = -1$ and $\texttt{current\_track}_2 = -1$.

The predicate `safety_req` is actually not an invariant per se. To prove it, we require the stronger predicate in Fig. 19: the shared invariant of the trains. Intuitively, the train invariant relates train positions to relay configurations of

```
definition train1_inv :: "railways_state upred" where
[urel_defs]: "train1_inv = (
  &current_track₁ =ᵤ «initial_track₁» ∨
  (&relays[route₁]ᵤ ∧ &current_track₁ ∈ᵤ «set (routes!route₁)») ∨
  &current_track₁ =ᵤ «final_track₁» ∨
  &current_track₁ =ᵤ 0)"
```

Figure 19: Shared invariant of the first train.

the interlocking. Namely, if a train has left the first track of its route, the corresponding relay must be activated, unless the final track was reached.

Unlike the typing invariant in Fig. 17, the train invariant cannot be decomposed into local invariants as it links the state of the train and interlocking FMUs, involving the variables $\texttt{current\_track}_{[1/2]}$ and `relays`. One of our results is that nonetheless some modular reasoning is possible: to establish the train invariant, it is not necessary to consider the particular implementation of the interlocking, but instead assume various local invariants I_{ilock} of the interlocking within the post-condition of the Hoare triple.

$$\{\texttt{train1\_inv} \wedge I_{ilock}\}\texttt{Train1}\{I_{ilock} \Rightarrow \texttt{train1\_inv}\}$$

The relay exclusion invariant in Fig. 18 alone is, however, not enough to characterise I_{lock}: we also need invariants that, for instance, establish the correct setting of railways switches and activity of the merger FMU. Details of this more challenging proof are in the Isabelle report [40].

To conclude, we have shown that mechanically proving properties of co-simulations in Isabelle/UTP is feasible using our model. Yet, it is a highly human-driven task that requires finding appropriate invariants and discharging proofs that they are preserved by the FMUs. Specialised Hoare rules for parallel composition proved to be of value to retain some level of compositionality in proofs that involve holistic and thus inherently non-decomposable shared invariants. Our conclusion is that proving properties of co-simulations is reducible to finding local and shared invariants, and in most cases these can be proved compositionally for FMUs.

5.4 Final Considerations

We have shown that our technique to reason about FMI co-simulations can be applied to a real-life industrial example from railways, and that it is indeed

possible to prove high-level safety properties of it. Importantly, our model abstracts from the particular simulation scenario in such a way that proved results remain valid for any permissible route combination of the trains. This is something that co-simulation tools naturally have difficulty to validate, and where model-checking likewise reaches its limits due to state explosion. We thereby can add value to an analysis that is exclusively based on simulation and model checking.

To obtain the above level of generality, careful design and proof engineering turned out to be key factors: we need to find suitable local and shared invariants, and proofs that they are preserved. Whereas well-formedness of an FMI architecture can be automatically proved even for larger-size models, proving properties of abstract behavioural models of co-simulations requires human expertise and knowledge.

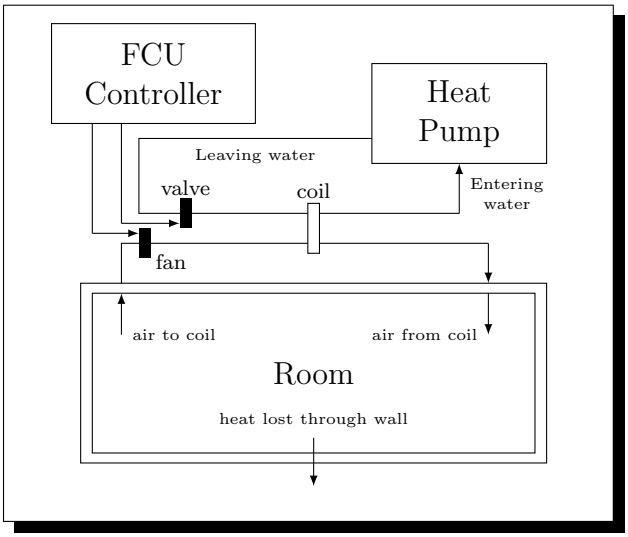

Figure 20: A diagram of the fan coil unit system.

6 The FCU Case Study

In this section, we discuss the mechanisation of our second case study - the fan coil unit (FCU), described in INTO-CPS Deliverable D1.1d, in Isabelle/UTP. Here, we use a variation of the strategy presented in Section 3. We also discuss the properties that we have proved about the model.

We first give an overview of the FCU case study and the properties that we prove, in Section 6.1. Then, in Section 6.2, we discuss the Modelica model on which we base our mechanisation of the FCU case study. After that, in Section 6.3, we discuss our mechanisation of the FCU in Isabelle/UTP, and then we examine the proofs of its properties, in Section 6.4.

6.1 Overview of FCU Case Study

The FCU example considers a system for heating a room by passing water from a heat pump through a coil, where heat is transferred to room air blown through the coil by a fan. The flow of air and water through the coil are controlled by the speed of the fan and the opening of a valve, respectively. The fan speed and valve setting are controlled by a software controller, the design of which is the focus of this case study. Heat transferred to the room is gradually lost over time through the walls of the room. Fig. 20 shows the overall layout of the FCU system.

The water passing through the coil is heated by a heat pump to a constant temperature. We refer to the water leaving the heat pump as the *leaving water temperature*, LWT. The rate of heat transfer through the coil, Q_{in}, is determined by the difference between LWT and the room air temperature, RAT, using the following equation,

$$Q_{in} = \epsilon \times fanSpeed \times \dot{m}_{at} \times c_{air} \times (LWT - RAT), \tag{4}$$

where

- ϵ is a constant representing the effectiveness of the coil,
- $fanSpeed$ is the setting of the fan's speed, ranging from 0 to 1,
- $\dot{m}_{at}$ is the maximum air flow rate that the fan can produce, and
- c_{air} is the specific heat of air.

The heat transfer, Q_{in}, corresponds to the heat loss of water flowing through the coil. It is hence moreover related to the temperature EWT of the water entering the heat pump by the following equation,

$$Q_{in} = valveOpen \times \dot{m}_{wt} \times c_{water} \times (LWT - EWT), \tag{5}$$

where

- $valveOpen$ is the setting of the valve, ranging from 0 to 1,
- $\dot{m}_{wt}$ is the maximum water flow rate through the coil, and
- c_{water} is the specific heat of water.

In addition to the heat transfer through the coil, the room air temperature is also affected by heat lost through the wall, Q_{out}. Such is determined by the difference between RAT and the surface temperature T_{isurf} of the wall on the inside of the room, and governed by the following equation:

$$Q_{out} = h_{air} \times A_{wall} \times (RAT - T_{isurf}), \tag{6}$$

where

- h_{air} is the heat transfer coefficient for the air, and
- A_{wall} is the surface area of the wall.

The overall rate of change in RAT is then determined using the following differential equation,

$$\frac{dRAT}{dt} = \frac{Q_{in} - Q_{out}}{\rho_{air} \times c_{air} \times v_{air}}, \tag{7}$$

where

- ρ_{air} is the density of air,
- c_{air} is, as above, the specific heat of air, and
- v_{air} is the volume of the air in the room.

The equations presented above describe the movement of heat in the room. The movement of heat in the wall is described by the following pair of differential equations that relate the surface temperature of the wall inside the room, T_{isurf}, and surface temperature outside the room, T_{osurf}:

$$\frac{d\,T_{isurf}}{d\,t} = (h_i * A_{wall} * (RAT - T_{isurf}) + (T_{osurf} - T_{isurf})/R)/C \quad (8)$$

$$\frac{d\,T_{osurf}}{d\,t} = (h_o * A_{wall} * (OAT - T_{osurf}) + (T_{isurf} - T_{osurf})/R)/C \quad (9)$$

where

- h_i is the heat transfer coefficient for the inside of the room,
- h_o is the heat transfer coefficient for the outside of the room,
- R is the thermal resistance of the wall,
- C is the thermal capacity of the wall, and
- OAT is the outside air temperature.

Together, these equations define the physical system that we aim to regulate. The FCU controller sets the values of *fanSpeed* and *valveOpen* to heat the room to a desired set-point temperature. Control is achieved by a PID controller, where the output of the PID must be split into values for *fanSpeed* and *valveOpen*. We discuss this in more detail in Section 6.2, where we consider the Modelica model of the system and its design as a co-simulation.

Our aim for this case study is to prove that certain desired properties are ensured by the construction of the FCU controller, given the properties captured in the above models of the room and wall. There are four such properties here. The first one is that the difference between the temperature of the water leaving the heat pump, LWT, and entering the heat pump, EWT, must be less than 5°C. This is required to minimise the strain on the heat pump and ensure that it can consistently heat the water to a constant temperature. This first property may be stated mathematically as follows:

$$LWT - EWT < 5 \quad (10)$$

This statement of the property relies on the fact that, since the FCU is being used to heat the room, LWT will always be higher than the room air temperature, and hence EWT will always be lower than LWT. For clarity, we state all temperatures in degrees Celsius. Since only differences of temperatures are used in the equations it is not necessary for us to explicitly use Kelvin, though it is the respective SI unit.

The second property we require is that the speed of the fan must always be at least 10% of its maximum speed. This is to ensure comfort via circulation of fresh air, since fresh air is added to the air passing through the coil. We note that since the latter has a negligible effect on the air temperature, it is not represented in the equations for the room. This property may be stated mathematically as follows.

$$fanSpeed \geq 0.1 \tag{11}$$

The third property is that, at a steady state, the room air temperature, RAT, must be within 1°C of the set-point, $RATSP$. This limits the amount of fluctuation that is allowed to occur after the room air temperature has reached the set-point. The notion of steady state is not formally defined here but may be taken as a state in which the variables of the model remain within fixed ranges. For RAT, we have the following inequality:

$$|RATSP - RAT| < 1 \text{ at steady state} \tag{12}$$

The fourth and last property is that, when the room air temperature is more than 1°C below the set-point, the valve must be more than 15% open. This ensures that the water flow round the system will be sufficient to ensure optimal heat pump operation when the heat pump is operating to raise the room air temperature to be close to the set-point. The mathematical statement of this property is as follows.

$$RATSP - RAT > 1 \Rightarrow valveOpen > 0.15 \tag{13}$$

The four properties above give rise to invariants of our system that we subsequently prove in Isabelle/UTP. First, however, we discuss the Modelica model of the FCU case study on which we have based our Isabelle/UTP encoding and proofs, as well as some modifications that we made to the original model in INTO-CPS Deliverable D1.1d to ensure that the properties discussed in this section can actually be guaranteed. As it turned out, the original model had to be modified as it did not satisfy the desired properties.

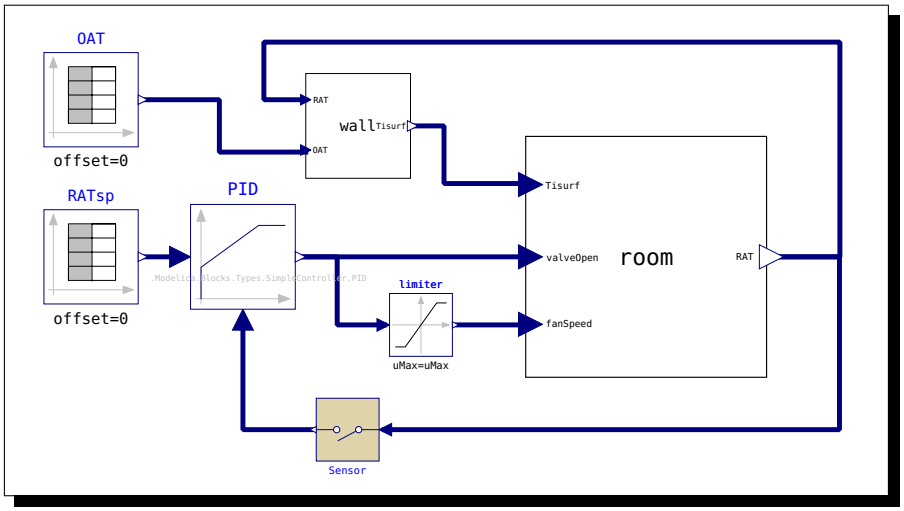

Figure 21: Graphical view of the original FCU model in OpenModelica.

6.2 FCU Modelica Model

A block diagram of the FCU Modelica model is depicted Fig. 21. It contains components labelled as `room` and `wall`, that are modelled by the equations presented in the previous section.

The `room` component takes as inputs the values for the surface temperature of the wall inside the room, T_{isurf}, the valve control, *valveOpen*, and the fan speed control, *fanSpeed*. It outputs the room air temperature, RAT.

The `wall` component takes the RAT value from the `room`, along with the outside air temperature, OAT. The `wall` outputs the value for T_{isurf} that is passed to the `room`. The OAT values for the `wall` are supplied by a table of values for each point in time. The lowest OAT value in the table is -1°C and the highest is 14.22°C.

The remaining components of the model shown in Fig. 21 comprise the FCU controller. The room air temperature is sampled every 15 minutes by the sensor shown at the bottom of the diagram. This sampled temperature is passed to a PID controller, which also receives the set-point value for the room temperature, which, like OAT, is supplied from a table of values for the purposes of simulation. The table of values varies the set-point between 21°C during working hours and 16°C outside working hours.

The PID controller computes a control value to move the room air temperature toward the set-point. The PID limits this control value to be between

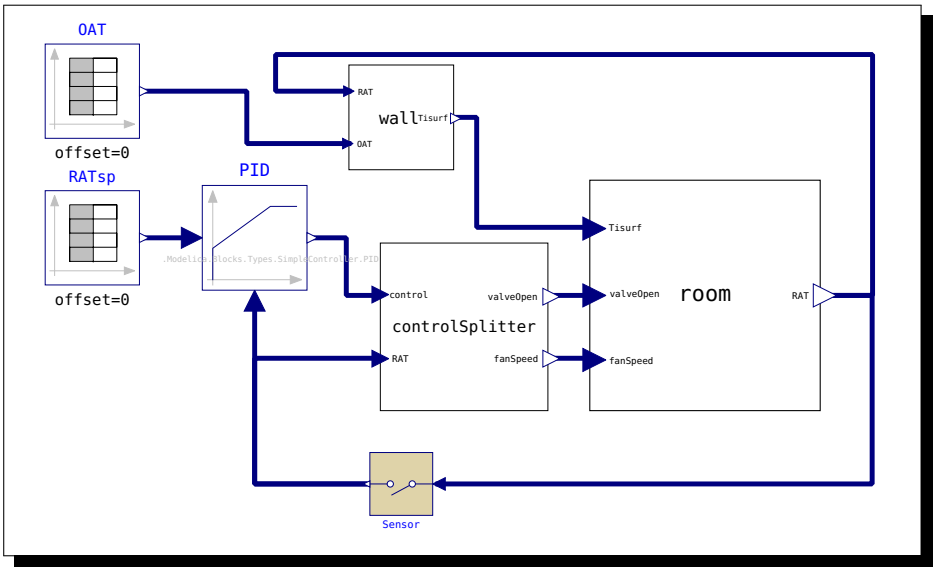

Figure 22: Graphical view of our modified FCU model in OpenModelica.

0.01 and 0.75. The control value is used for the values for *fanSpeed* and *valveOpen*, with the *fanSpeed* value passed through a limiter to ensure it is at least 0.1; this is crucial for the property in Eq. 11 to hold.

Our model performs well in simulations, but during verification we found that we could not prove the first required property ($LWT - EWT < 5$) in Eq 10. Further investigation revealed that this was due to the fact that, from equations (4) and (5),

$$LWT - EWT = \frac{\epsilon \times \dot{m}_{at} \times c_{air} \times fanSpeed \times (LWT - RAT)}{valveOpen \times \dot{m}_{wt} \times c_{water}}. \tag{14}$$

Substituting this into the statement of the property and rearranging the terms yields the following. then yields,

$$\frac{fanSpeed}{valveOpen} < \frac{5 \times \dot{m}_{wt} \times c_{water}}{\epsilon \times \dot{m}_{at} \times c_{air} \times (LWT - RAT)}. \tag{15}$$

The first property is thus equivalent to imposing a bound on the ratio between *fanSpeed* and *valveOpen*. Since the output from the PID controller is used for both *fanSpeed* and *valveOpen* in the original model, the correct ratio cannot be easily established *a priori*.

In order to ensure that the first property is fulfilled, we made some modifications to the OpenModelica model. The block diagram for our modified model

is shown in Fig. 22. It has a new component, labelled as `controlSplitter`, that computes the values for *fanSpeed* and *valveOpen* from the output of the PID controller. The temperature value measured by the sensor is passed to the `controlSplitter` to compute the bound on the ratio between *fanSpeed* and *valveOpen*.

The value for the bound is computed as in Eq. 15, with 5 replaced by 4.5 to account for changes in RAT in between samplings by the sensor. The value output by the PID controller is then used as the value for *fanSpeed*, since that determines how RAT changes.

The *fanSpeed* value is bounded below by 0.1, to ensure the second property is fulfilled, and above by the computed bound, to ensure that *valveOpen* can be set to a value that is not greater than 1. The *valveOpen* value is obtained by dividing the value for *fanSpeed* by the computed bound. This ensures that the first property is fulfilled, provided that the room temperature does not change by too much from the value recorded by the sensor. We will see later that we can obtain a proof for how much the room temperature may change that can be used to ensure this property holds.

Next, we discuss how the modified OpenModelica model is mechanised in Isabelle/UTP, so that we can analyse it and prove properties of it.

6.3 Mechanisation in Isabelle/UTP

Our Isabelle/UTP model of the FCU follows the approach in Section 3.2. The components representing FMUs act on a centralised state space, `st_fcu`, which contains the variables of all the components of the model, with each FMU only updating the variables that belong to it.

The definition of `st_fcu` is shown in the left-hand diagram of Fig. 23. The first four variables belong to the PID controller: `control` represents the output from the PID controller, while `previousuI`, `previousuDin` and `gain` are used internally. The next five variables correspond to the values *valveOpen*, *fanSpeed*, T_{isurf}, T_{osurf} and EWT in the equations presented in Section 6.1. The room air temperature is represented by two variables: `RAT_out`, the actual temperature of the room, and `RAT_sensor`, the temperature recorded by the sensor. The set-point and outside temperatures are also stored in state variables, namely `RATSP` and `OAT`. Finally, there is a variable `sensor_timeout` to measure how close the sensor is to taking the next sample of the room air temperature. All of the state variables are of type real.

```
alphabet st_fcu =
  control :: real
  previousuI :: real
  previousuDin :: real
  gain :: real
  valveOpen :: real
  fanSpeed :: real
  Tisurf :: real
  Tosurf :: real
  EWT :: real
  RAT_out :: real
  RAT_sensor :: real
  RATSP :: real
  OAT :: real
  sensor_timeout :: real
```

```
abbreviation "mdotwt :: real ≡ 0.1"
abbreviation "LWT :: real ≡ 40.0"
abbreviation "eps :: real ≡ 0.4"
abbreviation "cWater :: real ≡ 4181.0"
abbreviation "Awall :: real ≡ 60.0"
abbreviation "hAir :: real ≡ 4.0"
abbreviation "mdotat :: real ≡ 0.5"
abbreviation "rohAir :: real ≡ 1.204"
abbreviation "vAir :: real ≡ 300.0"
abbreviation "cAir :: real ≡ 1012.0"
```

Figure 23: The alphabet and constants for our Isabelle/UTP FCU model.

The constants in the model are declared as abbreviations in Isabelle. The right-hand diagram in Fig. 23 shows the definitions of various constants used in the model of the room. Other constants are similarly declared for use in the other parts of the model.

The components of the model are defined using separate Isabelle definitions. The variables are updated using assignment statements based on the equations of the model. Differential equations are handled by discretising them using Euler's method, with a fixed time step, which is set to 0.01 minutes in our model. We use minutes as the time units in our mechanised model to align it with the Modelica model.

An example of how components of the Modelica model are represented in Isabelle/UTP is the definition of `WallModel`, shown below, which implements the equations for transfer of heat through the wall. It assigns new values to the state variables `Tisurf` and `Tosurf`, using discretisations of Eq. 8 and Eq. 9.

```
definition WallModel :: "st_fcu hrel" where
[urel_defs]: "WallModel =
  Tisurf, Tosurf :=
    (&Tisurf + «stepsize» * («Tisurf_der» (&RAT_out)ₐ (&Tisurf)ₐ (&Tosurf)ₐ)),
    (&Tosurf + «stepsize» * («Tosurf_der» (&OAT)ₐ (&Tisurf)ₐ (&Tosurf)ₐ))
```

The derivatives of `Tisurf` and `Tosurf` are defined separately as functions `Tisurf_der` and `Tosurf_der`, the definitions of which are shown below.

```
definition Tisurf_der :: "real ⇒ real ⇒ real ⇒ real" where
[urel_defs]: "Tisurf_der RAT Tis Tos = (hi * aWall * (RAT - Tis) + (Tos - Tis) / R) / CC"
definition Tosurf_der :: "real ⇒ real ⇒ real ⇒ real" where
[urel_defs]: "Tosurf_der oat Tis Tos =((ho * aWall * (oat - Tos) + (Tis - Tos) / R) / CC)"
```

The model of the room is similarly represented in our Isabelle/UTP encoding by `RoomModel`. It assigns new values to `RAT_out` and `EWT`, with `RAT_out`'s value determined using the Euler method.

```
definition RoomModel :: "st_fcu hrel" where
[urel_defs]: "RoomModel =
  RAT_out, EWT :=
    (&RAT_out + «stepsize» *
      («RAT_der» («Qin_der» (&RAT_out)ₐ (&fanSpeed)ₐ) - («Qout_der» (&RAT_out)ₐ (&Tisurf)ₐ))
    («LWT» - («Qin_der» (&RAT_out)ₐ (&fanSpeed)ₐ) / (&valveOpen * «mdotwt» * «cWater»)))
```

We similarly define `Sensor` to model the behaviour of the sensor, `LimPID` to model the behaviour of the PID controller that limits its output to a predefined range, and `ControlSplitter` to model our `controlSplitter` component in the Modelica model. These definitions have been created by carefully examining the OpenModelica definitions of the components.

The above functions defined in Isabelle/UTP characterise FMUs that together form the FCU system, composed together as described in Section 3. However, our proofs only involve local invariants, so we do not need to consider the composition of the FMUs. The local invariants are sufficient to establish those invariants globally, as we discuss in the next section.

6.4 Proofs of Properties in Isabelle/UTP

The properties of our model that we have proved are stated, as before, using Hoare logic. We formalise the preconditions that we assume to hold for various components of the model and the postconditions that are established. In the preconditions, we require bounds on the state variables. We use the simulations to inform our choices of values for these bounds and, where possible, we use more general bounds for the variables to obtain statements about the model with wider applicability.

We use 21°C for the set-point, since that is the set-point value used during working hours in the simulation, and hence the case we focus on in our verification. We assume the room air temperature is between 0°C and 29°C. The room air temperature in the simulation falls within this range and it is wide enough to cover all realistic cases in which the FCU may be used to heat a room. We assume the internal surface temperature of the wall

```
theorem UTRC_FCU_001:
  "⦇&RAT_out <u 34 ∧
    &RAT_sensor ∈u {&RAT_out-0.6..&RAT_out+0.3}u ∧
    &control ∈u {«min_ctrl_value»..«max_ctrl_value»}u⦈
   (ControlSplitter ;; RoomModel)
   ⦇(«LWT» - &EWT) <u 5⦈u"
```

Figure 24: The statement of the first property in Isabelle/UTP (Eq. 10).

falls within a similar range, as it converges to the room air temperature in simulation.

The form in which we state the properties can be seen in the statement of the first property, `UTRC_FCU_001`, shown in Fig. 24. It holds whenever `RAT_out` is less than 34°C, and the value measured by the sensor, `RAT_sensor`, is no more than 0.6°C below and 0.3°C above the room temperature value. It is also required that `control`, the value output by the PID controller, be within the bounds the PID controller restricts it to, which are 0.01 and 0.75 in our model. The statement of the property then captures that $LWT - EWT < 5$ is established by the execution of `ControlSplitter` followed by `RoomModel`.

The consideration of the sequential composition of the components, rather than parallel composition as in Section 3.2 and the railways example, is justified since the postcondition established by the first component is true for the second component. All that then needs to be ensured is that the precondition of the property is not affected by the execution of the first step. The precondition on the `RAT_out` value is much more general than the assumptions on the room air temperature discussed above. The precondition on the value of `control` is ensured by the operation of `LimPID`, and we have proved that this is always the case, although we omit the proof here.

The relationship between `RAT_out` and `RAT_sensor` in the precondition of `UTRC_FCU_001` is justified by the lemma `RAT_out_movement_bounds`, recaptured in Fig. 25. It has a precondition similar to `UTRC_FCU_001`, but `RAT_out` is restricted to be between 0°C and 29°C and `Tisurf` is restricted to the same range, in accordance with the assumptions discussed above. When this precondition is fulfilled, the composition of `ControlSplitter` and `RoomModel` ensures that `RAT_out` does not increase by more than 0.0004°C and does not decrease by more than 0.0002°C within a 0.01 minute time step. Since the sensor is updated every 15 minutes, the sensor value never moves outside the required bounds.

```
lemma RAT_out_movement_bounds:
  "{&Tisurf ≥u 0 ∧ &Tisurf ≤u 29 ∧
    &RAT_out =u «x» ∧ &RAT_out <u 29 ∧ &RAT_out >u 0 ∧
    &RAT_sensor ∈u {&RAT_out-0.6..&RAT_out+0.3}u ∧
    &control ∈u {«min_ctrl_value»..«max_ctrl_value»}u}
   (ControlSplitter ;; RoomModel)
   {&RAT_out <u «x» + 0.0004 ∧ &RAT_out >u «x» - 0.0002}u"
```

Figure 25: A lemma ensuring bounds on the change in `RAT_out`.

```
theorem UTRC_FCU_002:
  "{-52 <u &RAT_sensor ∧ 40 >u &RAT_sensor}
   ControlSplitter
   {&fanSpeed ≥u 0.1}u"
  by (simp add: urel_defs, hoare_auto)
```

Figure 26: Encoding and proof of the second property in Isabelle/UTP.

The second property, `UTRC_FCU_002`, is stated as shown in Fig. 26. It says that `ControlSplitter` ensures `fanSpeed` will always be greater than 0.1 when `RAT_sensor` is between −52°C and 40°C. These bounds on the temperature measured by the sensor are wide enough to be consistent with our assumption about the room air temperature. The lower bound arises from the fact that `ControlSplitter` prioritises imposing the correct ratio to ensure the first property ($LWT - EWT < 5$) over ensuring that `fanSpeed` is less than 0.1. For temperature values much lower than −52°C, the ratio forces `fanSpeed` below 0.1.

For the third property, we also need to consider changes across multiple time steps to account for drift away from the set-point after the temperature has already converged. To deal with these, we establish further local invariants that can be used in proofs concerning other components of the model.

`Tisurf_Tosurf_invariant`, shown in Fig. 27, is an example of a lemma establishing such an invariant. It establishes that `Tisurf` and `Tosurf` are within their ranges when `RAT_out` is within 1°C of the set-point and `OAT` is between −10°C and 15°C. The ranges established by this lemma can be used in proofs of how the temperature changes in the room, since that depends on the value of `Tisurf`.

For the fourth property, we have proved `UTRC_FCU_004`, shown below. This states that, when `control` is within the range that `LimPID` restricts it to, and `RAT_sensor` is between −66°C and 24°C, `valveOpen` will always be greater

```
lemma Tisurf_Tosurf_invariant:
  "⦇&RAT_out <u 22 ∧ &RAT_out >u 20 ∧
    &Tisurf ≥u 0 ∧ &Tisurf ≤u 23 ∧
    &Tosurf ≥u -10 ∧ &Tosurf ≤u 23 ∧
    &OAT ≤u 15 ∧ -10 ≤u &OAT ⦈
   (WallModel)
   ⦇&Tisurf ≥u 0 ∧ &Tisurf ≤u 23 ∧
   &Tosurf ≥u -10 ∧ &Tosurf ≤u 23 ∧
   &RAT_out <u 22 ∧ &RAT_out >u 20⦈u"
```

Figure 27: A lemma establishing an invariant about `Tisurf` and `Tosurf`.

```
theorem UTRC_FCU_004:
  "⦇&control ∈u {«min_ctrl_value»..«max_ctrl_value»}u ∧
    &RAT_sensor <u 24 ∧ &RAT_sensor >u -66 ⦈
   (ControlSplitter)
   ⦇&valveOpen >u 0.15⦈u"
```

Figure 28: The statement of the fourth property in Isabelle/UTP.

than 0.15. As stated earlier, the bounds on `control` are always ensured. The lower bound on `RAT_sensor` is consistent with our assumptions about the room air temperature. The upper bound is sufficient to fulfil the property, since it is only required to hold when the room air temperature is more than 1°C below the set-point, which, as mentioned earlier, we take to be 21°C. It may be possible to obtain a proof for higher set-points if the setting of `control` by `LimPID` is considered in more detail.

The lemmas and properties that we have verified in Isabelle/UTP have been proved by first expanding definitions of functions and constants using Isabelle's simplifier and then applying a proof tactic called `hoare_auto`. The latter works by first applying Hoare-logic laws to break down the goal statement into simpler statements, and then trying to obtain an automatic proof using Isabelle's standard proof tactics along with definitions and laws from Isabelle/UTP.

For some proofs, `hoare_auto` alone is sufficient. This is true of `UTRC_FCU_002`, as can be seen in Fig. 26. For others, `hoare_auto` produces a list of subgoals stated in terms of standard mathematics. These can often be proved by manual application of mathematical laws but we have found that the `approximation` tactic [22] is very useful for proving subgoals that appear in our model. This tactic allows for proofs of inequalities about real numbers when ranges are given for each of the input variables. This requires us to supply the variable ranges in the correct form but when they have

```
fix control_v RAT_sensor_v :: real
assume control_assms: "1 ≤ control_v * 10" "control_v * 4 ≤ 3"
assume RAT_out_assms: "RAT_sensor_v < 24" "- 66 < RAT_sensor_v"
have RAT_out_range: "RAT_sensor_v ∈ {-66..24}" using RAT_out_assms by simp
have control_range: "control_v ∈ {0.1..max_ctrl_value}" using control_assms by simp
show "112887 < control_v * (161920 - 4048 * RAT_sensor_v) * 20"
  using RAT_out_range control_range by (approximation 100)
```

Figure 29: The proof of the first subgoal of `UTRC_FCU_004`.

been supplied, the proofs proceed automatically. This can be seen in Fig. 29, which illustrates the proof of one of the subgoals that `hoare_auto` generates for `UTRC_FCU_004`. The assumptions that `hoare_auto` provides are used to prove the aforementioned ranges on the variables.

6.5 Final Considerations

As a second case study to illustrate our analysis and verification technique for FMI, described in Section 3, we have presented our mechanisation of the FCU case study and proofs of some desired safety properties of it. We have based our mechanisation on the OpenModelica model of the FCU but found that it required some changes since it did not satisfy one of the properties. This testifies the added value of our mechanised theory and shows that attempting formal proofs with tools such as Isabelle/UTP can feed-back into on-going design work conducted using modelling and simulation tools.

Our proofs make use of the automation provided by the `hoare_auto` and `approximation` tactics, which has enabled us to experiment with the necessary preconditions of the properties to obtain bounds on state variables. In some cases the bounds obtained are more general than those observed in simulation. The proofs can therefore grant greater certainty of the correctness of properties in more general cases than can be considered in individual simulations. An Isabelle report of our FCU example can be found in the Isabelle/UTP distribution on GitHub: https://github.com/isabelle-utp/utp-main.

7 Discussion and Related Work

In this section, we discuss ramifications of our approach (Section 7.1), as well as its application in an industrial context (Section 7.2) and relevant related work (Section 7.3).

7.1 Ramifications

Our technique is based on the premise that a discrete abstraction can be found and used to reason about the continuous co-simulation model. For the railways case study, this is indeed so: the only potential safety hazard occurs when a signal changes from green to red and a moving train is present on the track controlled by that signal. Violation of safety in the continuous model can occur if the train is moving too fast and located too close to the signal at the moment when the signal change takes place.

A proof that the above scenario cannot arise in the concrete physical train model is an issue for refinement, which introduces the continuous train state and behaviour. Proofs about that model profit from our semantics of continuous and hybrid behaviours in Modelica in terms of the hybrid relational calculus (HRC) [19]. We have mechanised that calculus in Isabelle/UTP too, using the Multivariate Analysis and HOL-ODE theory libraries [24].

More specifically, to verify that trains do not overrun red signals, we impose a maximum speed threshold on trains; this also reflects real-world requirements of railways traffic. Since all signals are red initially, and any change of a signal from green to red only takes place once an initial track has been vacated (no new trains arrive in our scenario), for correct refinement we merely have to establish that a train can halt in time on its first track. A proof of this has been mechanised in Isabelle/UTP too, and is reported in one of our publications [42]. It makes use of restrictions on the length of the initial track segments and maximum speed of trains.

The above confirms the conceptual viability of our verification technique. They can be done compositionally, not having to consider the continuous behaviour of the entire FMI co-simulation.

Another point to consider is soundness. This is, for instance, important for certification evidence. The only place where unsound proofs can enter our verification approach is through Isabelle axiomatisations and the use of the `eval` tactic, though the latter is less likely to cause real issues since this tactic

is part of Isabelle/HOL itself and tested by a large community of users.

Nonetheless, we have made an effort to mitigate both potential sources of inconsistency. First, by proposing a sound definitional schema (Fig. 10) in the one place where an `axiomatizaion` is required, and, secondly, by verifying the result of our external C++ algorithm for closure computation, rather than trusting it. This means that we are principally not forced to employ Isabelle code generation, even for models of larger size.

7.2 Industrial Perspective

From an industrial perspective, we claim that our approach can add value to techniques that use co-simulation tools alone. For one, our experience is that once a co-simulation model has been developed, the construction of the architectural model in Isabelle/HOL is mostly straightforward, and fundamentally automatable from the INTO-SysML diagram: merely knowledge about internal direct dependencies has to be provided by the engineer. Well-formedness proofs of the architectural model can moreover be fully automated, using the tactics that we have presented in Section 5 and selecting one of the two strategies for discharging caveats about the control graph.

Reasoning about the behavioural FMI model is more challenging, and does require expertise and knowledge in formal modelling and theorem proving. However, our results show that a cleanly developed co-simulation model translates more easily into a mechanised model. The fact that we only have to consider relational computations limits expertise to being familiar with state-based verification methods and laws.

A basic knowledge in Isabelle/UTP and the refinement calculus [29] ought, in practice, suffice to formulate the abstract co-simulation model. For proof, we have to elicit and prove suitable invariants of the FMUs in order to validate, for instance, holistic and safety properties. Despite this, there are ways to aid the formal engineer in writing those models and conducting the proofs. For instance, the centralised FMU state is partially derivable from the INTO-SysML model; and bespoke tactics for Hoare logic turn out useful to subdivide and structure proofs. Finding local FMU invariants adds value to the analysis, as it helps the engineer to understand the assumptions and commitments that are made by various parts of the system (FMUs).

The last step in the proof is the refinement into a continuous model. Techniques of data refinement are applicable here out-of-the-box. Once again, it is universally acknowledged that refinement is a human-driven process,

though, in certain cases, it can be automated with high-level tactics and strategies. Automation typically requires the specification and target to be of a particular known shape. There exist various examples in the literature of refinement strategies that profit from well-defined target models [31, 38, 27], and our reactive model of an FMI co-simulation in INTO-CPS Deliverable D2.3c satisfies this property, too.

7.3 Related Work

First to mention is Broman's formalisation of FMI co-simulations [10]. Broman's model is very concrete, directly encoding the behaviour of FMI interface functions such as `fmi2Get`, `fmi2Set` and `fmi2doStep` for each FMU. This is because his work work focusses in the first instance on verifying properties of master algorithms rather than particular co-simulations instances. In contrast, our technique aims to be more abstract: we consider the FMI interface and master algorithm as artefacts introduced through refinement.

Apart from this, there are similarities between Broman's and our approach in that they are both based on a relational view that admits a notion of contract for abstraction. In the report [36], Tripakis and Broman 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. In comparison, we benefit form the UTP as a lingua franca to formalise and integrate semantic theories of various heterogeneous languages.

Second to mention, our previous work [7] proposes a concrete reactive model of FMI co-simulations. That model — like Broman's — aims to faithfully represent the FMI interface and can likewise potentially be used to verify properties of master algorithms. The language used there is *Circus*, a process algebra for state-based reactive systems, and the technique for verification is either model-checking or algebraic refinement. We could have used that model directly to formulate properties of co-simulations but this would have made proofs much more difficult. Yet, as explained, our approach is based on refinement and the final step of the refinement results in a model that indeed matches the one in [7].

Several other works exist that propose MAs [4, 34, 12, 9], but usually they do not provide a semantic underpinning to prove that the algorithm or co-simulation instance satisfies general and particular properties.

8 Conclusion

We have presented a novel technique for reasoning about FMI co-simulation architectures and behavioural models defined via the INTO-SysML profile. We have illustrated that technique by way of elaborate case studies, supplied by our industrial collaborators for the INTO-CPS project. Our technique is based on abstraction: we use a relational view of FMUs that abstracts from reactive behaviours as well as the API imposed by the FMI. This allows us to focus on the fundamental properties of a co-simulation, while introducing details into the model view refinement that preserves those properties.

The concrete target of our refinement is the *Circus* model of an FMI co-simulation, as described in [7] and mechanised in INTO-CPS Deliverable D2.3c. A refinement strategy that allows us to produce such a model from the one described in this report is similar to that in [38] and can moreover reuse some of its laws. The fundamental issue is localisation of states into FMUs and replacing the single synchronisation on the *step* channel by multiple synchronisations on the channels $fmi2Get$, $fmi2Set$ and $fmi2doStep$ that corresponding to method calls.

As future work, we first suggest the development of a tool that supports the user of our technique in automatically generating the Isabelle/UTP architectural model, as well as a sketch of the behavioural model. The formal developer can use the sketch as a starting point, completing it with a detailed encoding of functional behaviours of FMUs. Secondly, elements of the refinement strategy from abstract into concrete FMU models ought be explored for a larger spectrum of case studies and examples, beyond the ones we presented in this report. Both these works could be tackled by the INTO-CPS Association.

References

[1] J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, New York, NY, USA, 1st edition, 2010.

[2] Atego (acquired by ptc). Integrity Modeler. See https://www.ptc.com/en/products/plm/plm-products/integrity-modeler, 2017.

[3] 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/.

[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] J. Bicarregui, J. Fitzgerald, P. G. Larsen, and J. Woodcock. Industrial practice in formal methods: A review. In *FM 2009: Formal Methods: Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings*, Lecture Notes in Computer Science, pages 810–813. Springer, November 2009.

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

[7] 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.

[8] W.-P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and Their Comparison*. Cambridge University Press, New York, NY, USA, 1st edition, 2008.

[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 and J. Woodcock. *Towards Verification of Cyber-Physical Systems with UTP and Isabelle/HOL*, volume 10160 of *Lecture Notes in Computer Science*, pages 39–64. Springer, 2017.

[15] S. Foster and F. Zeyda. Isabelle/UTP: A verification toolbox for Isabelle/HOL based on Unifying Theories of Programming. Tool website: https://www-users.cs.york.ac.uk/~simonf/utp-isabelle/, 2017.

[16] 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.

[17] The Eclipse Foundation. Papyrus Modeling environment. Tool website: https://www.eclipse.org/papyrus/, 2015.

[18] 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.

[19] 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, December 2016.

[20] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, October 1969.

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

[22] J. Hölzl. Proving inequalities over reals with computation in Isabelle/HOL. In *International Workshop on Programming Languages for Mechanized Mathematics Systems*, pages 38–45, 2009.

[23] IBM. Rational Rhapsody Designer for Systems Engineers. See http://www-03.ibm.com/software/products/en/ratirhapdesiforsystengi, 2017.

[24] F. Immler and J. Hölzl. Numerical Analysis of Ordinary Differential Equations in Isabelle/HOL. In *ITP 2012*, volume 7406 of *LNCS*, pages 377–392. Springer, August 2012.

[25] C. T. Jones. *Programmable Logic Controllers: The Complete Guide to the Technology*. Patrick Turner Publishing Company, 1st edition, 1996.

[26] D. Matichuk, T. Murray, and M. Wenzel. Eisbach: A Proof Method Language for Isabelle. *Journal of Automated Reasoning*, 56(3):261–282, March 2016.

[27] A. Miyazawa and A. Cavalcanti. Refinement Strategies for Safety-Critical Java. In *Proceedings of SBMF 2015, 18th Brazilian Symposium on Formal Methods*, volume 9526 of *Lecture Notes in Computer Science*, pages 93–109. Springer, September 2016.

[28] Modeliosoft. modelio: the open source modeling environment. Tool website: https://www.modelio.org/, 2011-2017.

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

[30] P. Nuzzo, A. L. Sangiovanni-Vincentelli, D. Bresolin, L. Geretti, and T. Villa. A Platform-Based Design Methodology With Contracts and Related Tools for the Design of Cyber-Physical Systems. *Proceedings of the IEEE*, 103(11):2104–2132, November 2015.

[31] M. Oliveira, F. Zeyda, and A. Cavalcanti. A tactic language for refinement of state-rich concurrent specifications. *Science of Computer Programming*, 76(9):792–833, September 2011.

[32] OMG. Unified Modelling Language™, Version 2.5. Technical report, Object Management Group, March 2015. Download from http://www.omg.org/spec/UML/.

[33] OMG. OMG Systems Modeling Language™, Version 1.5. Technical report, Object Management Group, May 2017. Download from http://www.omg.org/spec/SysML/.

[34] 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.

[35] Klaus Simon. An improved algorithm for transitive closure on acyclic digraphs. *Theoretical Computer Science*, 58(1):325–346, June 1988.

[36] 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.

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

[38] F. Zeyda and A. Cavalcanti. Laws of mission-based programming. *Formal Aspects of Computing*, 27(2):423–472, March 2015.

[39] F. Zeyda and A. Cavalcanti. Mechanisation of the FMI Railways Architecture: Isabelle Proof Document. Technical Report INTO-CPS Project, University of York, York, YO10 5GH, UK, November 2017. Available from http://.

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

[41] F. Zeyda, S. Foster, and L. Freitas. An Axiomatic Value Model for Isabelle/UTP. In *Proceedings of UTP 2016*, volume 10134 of *Lecture Notes in Computer Science*, pages 155–175. Springer, June 2016.

[42] F. Zeyda, J. Ouy, S. Foster, and A. Cavalcanti. Formalising Cosimulation Models. In *Proceedings of CoSim-CPS 2017*, Lecture Notes in Computer Science. Springer, 2017. Currently under publication.

A Modelica Train Model

```
within Railways.Full;

model FullTrain "Physics model of the train and behavioural
model of the driver."
  /* Constants */
  import Railways.Common.Topology.*;

  /* Functions */
  import Railways.Common.TrainControl.*;

  /* Parameters */

  /* Note that fixed_route does not control the path in the full
train model. */
  parameter Integer fixed_route(start = V1Q1);

  /* Initial speed of the train. */
  parameter Real initial_speed(start = 0, unit = "m.s-1");

  /* Parameters of the simulation scenario. */
  parameter Integer initial_track(start = 1);
  parameter Integer direction(start = QtoV);

  /* Maximum permitted speed of the train. */
  parameter Real max_speed(unit = "m.s-1") = 4.1;

  /* Safety margin to stop before the end of the track. */
  parameter Real safety_margin(unit = "m") = 2.0;

  /* Normal acceleration when there is no restriction on the
track. */
  parameter Real normal_acceleration(unit = "m.s-2") = 0.25;

  /* Normal deceleration when the train must stop before a red
signal. */
  parameter Real normal_deceleration(unit = "m.s-2") = -1.4;

  /* Inputs and Outputs */

  /* Signals on the track observed by the train driver. */
  Modelica.Blocks.Interfaces.BooleanInput signals[3](each start
= RED);

  /* Current configuration of railway switches. */
  Modelica.Blocks.Interfaces.IntegerInput switches[5];

  /* Track segment on which the train is currently located. */
  Modelica.Blocks.Interfaces.IntegerOutput track_segment;
```

Modelica Train Model

```
  /* Telecommand issued by the train driver to request a route.
*/
  Modelica.Blocks.Interfaces.BooleanOutput telecommand[4];

  /* Local Variables */
  Real acceleration(unit = "m.s-2");
  Real current_speed(unit = "m.s");
  Real position_on_track(unit = "m");
  Real track_length(unit = "m");
  Real setpoint_speed(unit = "m.s");

  /* Initial equations determine the speed and location of the
train at time zero. */

  initial equation
    track_segment = initial_track;
    current_speed = initial_speed;
    position_on_track = 0;

  /* Algorithms and Equations */

  /* 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.
*/

  /* Currently gravity and smooth acceleration are not
considered. */

  equation
    when abs(current_speed - setpoint_speed) < 0.001 then
      /* To avoid chattering during simulation. */
      /* This case also corresponds to engaging the brakes. */
      reinit(current_speed, setpoint_speed);
      reinit(acceleration, 0);
    end when;
    /* In reality, we would use a PID controller here. */
    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
```

Modelica Train Model

```
        /* Decelerate */
        acceleration = normal_deceleration;
      end if;
    end if;

  /* Behaviour of the train driver in defining the set-point
speed. */

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

  /* Calculation and update of the track segment. */

  equation
    when position_on_track > pre(track_length) then
      /* The donkey work is done by the NextTrack() function in
TrainControl. */
      track_segment = NextTrack(pre(track_segment),
pre(switches), direction);
      reinit(position_on_track, 0);
    end when;

  equation
    /* The track length becomes 0 when we derail or leave the
interlocking. */
    track_length = (if track_segment > 0 then
track_length_tab[track_segment] else 0);

  /* Algorithm to issue the necessary telecommand for a route
request. */

  algorithm
    /* We assume the train starts from a track with a
telecommand station. */
    telecommand :=
      /* Simulation seems to complain about an if *statement*
here?! */
      if track_segment == initial_track then
        Railways.Common.Topology.Route2TC(fixed_route)
      else TC_NONE;
end FullTrain;
```

B 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)
```

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