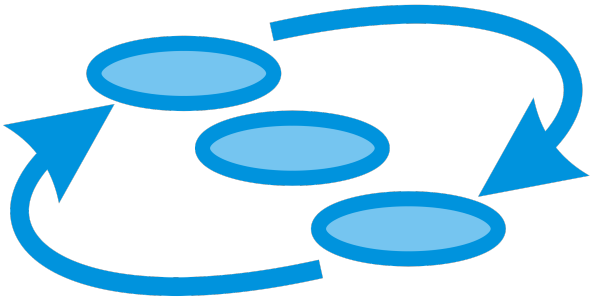
Grant Agreement: 644047

INtegrated TOol chain for model-based design of CPSs



INTO-CPS

Multi-Model Linking Semantics

Technical Note Number: D2.3d

Version: 1.2

Date: December 2017

Public Document

http://into-cps.au.dk

Contributors:

Ana Cavalcanti, UY
Simon Foster, UY
Jim Woodcock, UY
Frank Zeyda, UY

Editors:

Simon Foster, UY

Reviewers:

Luis Diogo Couto, UTRC
Adrian Pop, LIU
Martin Mansfield, NCL

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	15-05-2016	Simon Foster	Initial document version
1.0	01-11-2017	Simon Foster	Internal review version
1.1	11-12-2017	Simon Foster	Added additional materials on multi-model verification, integration of additional notations, and tools
1.2	14-12-2017	Simon Foster	Final version for submission

Abstract

In this deliverable we show how the theory linking facilities of the UTP can be used to compose multiple heterogeneous models in an FMI multi-model. We give a timed relational semantics to FMI multi-models that accounts for each FMU, the connections between them, and the master algorithm. The FMUs should be encoded as suitable UTP theories, and a linking function to a timed relational interface given. We exemplify this with examples from VDM-RT and Modelica, which are encoded using our UTP theories of timed and hybrid relations, respectively. We also show how such heterogeneous multi-models can be subjected to automated Hoare logic verification using theorem proving. Finally, we give guidelines for integration of additional notations and tools into our framework.

Contents

1 Introduction

Deliverable D2.3a [19] demonstrates a relational semantics for FMI that abstracts over the details of the API as previously modelled using *Circus* [14], which is nevertheless behaviour preserving. That semantics focusses on parallel composition of relational FMUs that relate the given input values to given output values and a time step. A technique is developed in D2.3a [19] that utilises a relational Hoare logic, built on top of Unifying Theories of Programming [13, 5] (UTP), to demonstrate that the composition of FMUs, together with constraints imposed by a master algorithm, exhibit certain desirable properties. This verification technique is automated in the Isabelle/HOL theorem prover, and has been applied to prove properties of both the railways [15] and buildings case studies [8]. It remains to show how discrete event and continuous time models can be integrated into such relational presentations.

This deliverable shows how heterogeneous models can be integrated into an FMI co-model by leveraging the UTP's theory linking capabilities. We have previously shown in Deliverable D2.2b [11] how the real-time modelling language VDM-RT, typically used in INTO-CPS to model controllers, can be given a denotational semantics in terms of the UTP theory of timed reactive designs. Moreover, in parallel with this deliverable we show in Deliverable D2.3b [7] how the dynamical systems modelling language, Modelica, can be given a semantics in terms of hybrid reactive designs. This deliverable then shows how timed reactive designs and hybrid relations can be composed in an FMI multi-model using a UTP theory of timed relations as a *lingua franca*. Our UTP theory is used to give a semantics to an FMI multi-model, including the individual FMUs, connections, and the master algorithm. It can be used to subject multi-models to verification using Isabelle/UTP [10] in order to substantiate (or refute) safety properties of a cyber-physical system.

In Section 2 we give a brief overview of theory linking in the UTP, and explain how the latter facilitates multi-model integration. In Section 3 we explain our UTP theory of timed relations, and in Section 4 show how this can be used to give a semantics to FMI multi-models. In Section 5 we show how to give associated FMU semantics to VDM-RT periodic threads, and we do the same for Modelica and the hybrid relational calculus in Section 6. Then, in Section 7 we bring these results together to show how verification of an FMI multi-model can be performed in Isabelle/UTP. Sections 8 and 9 then give guidelines for integration of additional notations and tools, respectively, into our semantic framework. Finally in Section 10 we conclude the deliverable.

2 Linking UTP Theories

The goal of the UTP is provide a unifying semantic framework for the multitude of computational theories that underlie programming languages. Computational theories can be used to account for discrete state, concurrency, communication, real-time, hybrid systems, probability, object orientation, and mobility, to name a few of the fundamental paradigmatic ideas that can be formalised in UTP. For more foundational information on the foundations of UTP, please see Section 2 of D2.3b [7].

One part of the theory engineering activity is to formalise these computational theories,

as we have done in D2.2b [11] and D2.3b [7], for example, prove laws of them, and then use these laws to create verification tools. However, another important activity is to demonstrate the links between different theories. For example, there are clearly links between real-time systems and hybrid systems, and the UTP allows us to formally describe these links. Often such links are described using Galois connections that demonstrate mutual embeddings between two theory domains.

This is a clear advantage of the UTP as a semantic framework – it not only enables us to describe semantics for heterogeneous models, but it also allows their integration. This is why UTP is so important for describing multi-models in INTO-CPS, because it allows us to give a precise mathematical account of the composition of models from different languages and theoretical domains. Moreover, as we shall see, it also allows the application of this underlying theory to verification techniques that can encompass a wide variety of heterogeneous models.

In the remainder of this deliverable we shall show how VDM-RT and Modelica can be integrated in an FMI multi-model using such theory linking capabilities.

3 Timed Relations

UTP is based on the alphabetised relational calculus. A relation is modelled as a logical predicate with two kinds of variables: initial variables (x), and final variables (x'). Thus, a program can be modelled as a relation that shows the possible final states that can result from a given initial state. A simple operator, like assignment ($x := v$), can be modelled as a predicate: $x' = v \wedge y' = y$. This states that x in the final state takes the value v, whilst any other variable y retains its previous value.

Our relational FMI model uses an extension of UTP relations that adds a special observational variable called *time* : $\mathbb{R}_{\geq 0}$ that models the passage of time using a postive real number. It is special because, unlike regular program variables, it is restricted in how it can evolve. For example, we cannot assign an arbitrary value to *time* as this would violate several physical laws. Thus, a timed relation shows both the final valuation for any program variables, and also how much time passed while computing these variables.

Technically, a timed relation is modelled using our theory of generalised reactive relations that we describe in sister deliverable D2.3b [7, Section 3] on the continuous time semantics. Briefly, a reactive relation is a UTP relation with an observational variable tr that is used to model the trace. The trace can be any instance of a "trace algebra", whose models include positive real numbers, discrete event sequences, and piecewise continuous functions. The latter are employed in our theory of hybrid relations for the semantics of Modelica. Here, we shall use the trace variable to model the *time* variable.

The healthiness condition for reactive relations is ***RR*** which is defined below.

Definition 3.1 (Reactive Healthiness Conditions)**.**

$$\textbf{\textit{R1}}(P) \triangleq P \wedge tr \leq tr'$$

$$\textbf{\textit{R2}}_c(P) \triangleq P[\epsilon, tr' - tr/tr, tr'] \lhd tr \leq tr' \rhd P$$

$$\textbf{\textit{RR}}(P) \triangleq (\exists\, ok, ok', wait, wait' \bullet \textbf{\textit{R1}}(\textbf{\textit{R2}}_c(P)))$$

R1 requires that the trace monotonically increases. ***R2***$_c$ forbids dependence upon the history of interaction before a given process began executing. We form the theory of timed relations with $tr : \mathbb{R}_{\geq 0}$, and by using the definitions of $x \leq y$, $x + y$, and $x - y$ from real number theory. Then, the healthiness condition ***R1*** has the specialised definition of time only being allowed to advance, and not reverse, as we mentioned previously. ***R2***$_c$ means that any timed relation cannot observe the specific time at which it started to execute. It can only record the amount of time that has elapsed since its execution began. This is a kind of time invariance property.

The theory of timed relations allows the description of timed imperative programs featuring the following usual programming operators, all of which are closed under ***RR***, and thus can be used together to construct well-formed timed programs.

- sequential composition $(P \,;\, Q)$, and its iterated relative $(\fatsemi\, i \in I \bullet P(i))$;
- assignment $(x :=_r v)$;
- if-then-else conditional $(P \lhd b \rhd Q)$ – (if b then P, else Q);
- non-deterministic choice $(P \sqcap Q)$ and its indexed relative $(\bigsqcap i \in I \bullet P(i)$;
- finite iteration (P^*);
- miraculous program (***false***).

As usual, we also have II_r as the degenerate assignment that updates no variables; it is the unit of sequential composition $(\mathit{II}_r \,;\, P = P \,;\, \mathit{II}_r = P)$. We also have a more general version of the assignment operator $\langle\sigma\rangle_r$, where σ is a mapping from variables to expressions, that allows multiple simultaneous assignments. For example, $x :=_r v$ is equivalent to $\langle\{x \mapsto v\}\rangle_r$, and II_r to $\langle id\rangle_r$.

We also introduce a frame operator $a : [P]$, the effect of which is to specify that timed relation P determines how state variables within a (the frame) can change. All other state variables outside of frame a retain the same values in the final state. For example, the relation $x : [x' = 7]$ states that only x can change value, and in the final state will take the value 7. It is therefore equivalent to an assignment $x :=_r 7$.

In this language of timed relations, we impose a number of restrictions on syntactically valid programs. An assigned expression v can only refer to state variables, and not the time variable. The same restriction follows for an if-then-else condition b, since b can refer only the initial values of state variables and the initial value of *time* has no meaning as enforced by ***R2***$_c$. We are also not permitted to assign to *time* as this may result in unhealthy behaviour. Finally, we require that in a framed relation $a : [P]$, P can only refer to variables within frame a.

The Kleene star operator P^* has the usual definition of a non-deterministic choice of all finite repetitions of the enclosed behaviour, as defined below.

Definition 3.2 (Kleene Star)**.**

$$P^* \triangleq \bigsqcap i \in \mathbb{N} \bullet P^i$$

Here, P^i is defined to be i sequential iterations of P, with $P^0 = \mathit{II}_r$. The timed relational operators obey the usual algebraic laws of programming and relational calculus, some of

which we enumerate below.

Theorem 3.1 (Timed Relation Laws)**.** *If P, Q, R are **RR** healthy then*

$$\begin{aligned}
\mathit{II}_r \,;\, P &= P \,;\, \mathit{II}_r = P \\
\textbf{\textit{false}} \,;\, P &= P \,;\, \textbf{\textit{false}} = \textbf{\textit{false}} \\
(P \,;\, Q) \,;\, R &= P \,;\, (Q \,;\, R) \\
x :=_r v \,;\, P &= P[v/x] \\
P \lhd \mathit{true} \rhd Q &= P \\
P \lhd \mathit{false} \rhd Q &= Q \\
(P \lhd b \rhd Q) \,;\, R &= P \,;\, R \lhd b \rhd Q \,;\, R \\
P \sqcap Q &= Q \sqcap P \\
P \sqcap \textbf{\textit{false}} &= P \\
a{:}[\mathit{II}_r] &= \mathit{II}_r \\
a{:}[P \,;\, Q] &= a{:}[P] \,;\, a{:}[Q] \\
a{:}[P] \,;\, b{:}[Q] &= b{:}[Q] \,;\, a{:}[P] \qquad a \cap b = \emptyset \\
P^* &= P \,;\, P^*
\end{aligned}$$

These laws of programming should be mostly familiar from relational calculus [12, 13]. The second to last law deserves comment, however. It states that two framed timed relations can be commuted provided their frames are disjoint. This law follows because neither of the sequentially composed programs can interfere with each others' state variables.

Additionally, we implement an operator for delays in timed relations.

Definition 3.3 (Delay Operator)**.**

$$\textbf{\textit{wait}}(n) \triangleq (st' = st \wedge time' = time + n)$$

This operator simply leaves all state variables (st) unchanged and advances the time by n. A couple of additional useful identities follow from this operator.

Theorem 3.2 (Delay Properties)**.**

$$\textbf{\textit{wait}}(0) = \mathit{II}_r \tag{3.2.1}$$

$$\textbf{\textit{wait}}(m) \,;\, \textbf{\textit{wait}}(n) = \textbf{\textit{wait}}(m + n) \tag{3.2.2}$$

$$\textbf{\textit{wait}}(m) \,;\, (P \lhd b \rhd Q) = (\textbf{\textit{wait}}(m) \,;\, P) \lhd b \rhd (\textbf{\textit{wait}}(m) \,;\, Q) \tag{3.2.3}$$

$$\textbf{\textit{wait}}(n) \,;\, x :=_r v = x :=_r v \,;\, \textbf{\textit{wait}}(n) \qquad x \notin \mathit{fv}(v) \tag{3.2.4}$$

$$a{:}[\textbf{\textit{wait}}(n)] = \textbf{\textit{wait}}(n) \tag{3.2.5}$$

Property (1) shows that a zero wait is simply the timed relational identity, since no time has elapsed. Property (2) shows that two sequential delay results in a single delay that is

the sum of the parts. Property (3) shows that a delay distributes from the left through a conditional; this follows because b cannot depend upon time. Property (4) shows that assignment and delays commute, provided the delay expression does not depend on x. At first sight this may seem counter-intuitive, but it holds because we only observe the final values of variables and not their intermediate observations. This is faithful to FMI, where we only perceive variables following a time step of the FMUs. Finally, property (5) shows that framing a delay has no effect since all state variables retain their value.

4 FMI in Timed Relations

Following D2.3a we give an abstract relational semantics to networks of FMUs. However, we here use our theory of timed relations as the basis of the semantics. An FMI multi-model consists of n FMUs with connections between them, an initialisation routine to provide initial values for input variables, and a master algorithm to orchestrate their execution. We consider a simplified model of FMUs that consist of a state space, containing the input, output, and internal variables; an instantiation routine *inst*, which is called from `fmi2Instantiate`; and a step routine *step*, which is called from `fmi2DoStep`. We represent an FMI structure using a timed relation of the following form where the n FMUs (FMU_i) are indexed by a set $I = \{1..n\}$.

Definition 4.1 (FMI Timed Relation)**.**

$$(\fatsemi\, i \in I \bullet s_i : [FMU_i.inst]) \;;\; \langle Init \rangle_r \;;$$

$$\Big(\langle Wiring \rangle_r \;;\; \Big(\prod t \in MA \bullet (\fatsemi\, i \in I \bullet s_i : [FMU_i.step(t)]) \;;\; \mathbf{wait}(t)\Big)\Big)^*$$

Such a timed relation operates over a state space consisting of all variables of all FMUs in the composition. The state space of each FMU is a set of variables s_i, and we have it that $\forall\, i, j \in I \bullet i \neq j \Rightarrow s_i \cap s_j = \emptyset$. *Init* and *Wiring* are both variable assignments that set initial values for variables, and connect FMU outputs to FMU inputs, respectively. We assume that any problematic algebraic loops have been filtered out using the verification techniques in D2.3a [19].

There are two phases to co-simulation: initialisation and execution. In the initialisation phase, each of the constituent FMUs is first instantiated. This involves executing *inst* for every FMU in the network, each of which is framed by the corresponding FMU state space s_i. Since the state spaces are disjoint, we can execute the instantiations in any order and obtain the same result. We therefore chose to do so in the order of the given indices using iterated sequential composition ($\fatsemi$), but we can also re-order them if necessary. Following instantiation, we need to provide initial values for FMU input variables which is done through the *Init* assignment. Actually, for convenience we choose to populate initial values for all output variables, since the *Wiring* assignment will copy these to the corresponding input variables.

In the execution phase, the FMUs are all stepped forward by a chosen time step t. We first prime them all for execution through the *Wiring* assignment, which provides values for the input variables. Following this, we non-deterministically pick a time t that the master algorithm is willing to permit. Then we step each of the FMUs forward by t, and

insert a delay of the given time period. Again, the FMUs operate on distinct section of the state space and therefore we can step them forward in any sequential order. This main body of execution is then iterated a finite number of times using the Kleene star operator, P^*, that corresponds to reflexive transitive closure. Thus, the semantics of an FMI network is the set of all permissible sequences of time steps allowed by the master algorithm, with the network wired together using *Wiring*, and started in initial state *Init*.

We represent the master algorithm by a set of positive reals: $MA : \mathbb{P}(\mathbb{R}_{\geq 0})$. For example, we can define a fixed step and variable step master algorithms as below.

Definition 4.2 (Sample Master Algorithms)**.**

$$\begin{aligned} \mathit{FixedStep}(t) &\triangleq \{t\} \\ \mathit{ArbStep} &\triangleq \{t \in \mathbb{R}_{\geq 0} \mid t > 0\} \end{aligned}$$

The $\mathit{FixedStep}(t)$ master algorithm permits only time steps of length t, whilst *ArbStep* permits any non-zero time step which is chosen non-deterministically.

The master algorithm, of course, is not solely responsible for picking the step size, and there must also be agreement from the FMUs. An FMU is stepped forward using $\mathit{step}(t)$, which constructs a timed relation corresponding to all t length evolutions. If t is not a valid time step for the current FMU state, then the behaviour will be ***false***: the miraculous relation. Since ***false*** is a zero for sequential composition, the entireity of the behaviour for that time step will be miraculous. Moreover, ***false*** is a unit for non-deterministic choice ($\bigsqcap$), and therefore any such miraculous time steps are effectively excluded by the overall orchestration. In other words, both the master algorithm and all FMUs must agree on the length of each time step.

We exemplify this with an adapted version of the Water Tank pilot study from D3.6 with two FMUs: one to model the continuous behaviour of the tank, and one to model the controller. We create a state-space for each of them. The tank state-space consists of two variables $\mathit{valve} : \mathbb{B}$ that models whether the valve is open or shut, and $\mathit{level} : \mathbb{R}$ that models the current water level in the tank. The controller state-space also consists of two variables: $\mathit{valveOn} : \mathbb{B}$ and $\mathit{levelSensor} : \mathbb{R}$ that correspond to an actuator and sensor for the two correspond tank entities. We then take the conjunction of the two state-space, with namespaces *tank* and *ctr* applied to the respective variable names.

The water tank relational FMI, with a particular configuration, can then be modelled as given below.

Example 4.1 (Water Tanks in relational FMI)**.**

$$
\begin{aligned}
\mathit{Init} &\triangleq \left\{ \begin{array}{l} \mathit{tank{:}valve} \mapsto \textbf{\textit{true}}, \mathit{tank{:}level} \mapsto 5, \\ \mathit{ctr{:}valveOn} \mapsto \textbf{\textit{false}}, \mathit{ctr{:}levelSensor} \mapsto 0 \end{array} \right\} \\
\mathit{Wiring} &\triangleq \{\mathit{tank{:}valve} \mapsto \mathit{ctr{:}valveOn}, \mathit{ctr{:}levelSensor} \mapsto \mathit{tank{:}level}\} \\
\mathit{MA} &\triangleq \mathit{FixedStep}(0.001) \\
\mathit{WT}(\mathit{minl}, \mathit{maxl}) &\triangleq \mathit{TankFMU.inst} \,;\, \mathit{CtrFMU.inst} \,;\, \langle \mathit{Init} \rangle_r \,; \\
&\qquad \left(\langle \mathit{Wiring} \rangle_r \,;\, \left(\prod t \in \mathit{MA} \bullet \left(\begin{array}{l} \mathit{TankFMU.step}(t) \,; \\ \mathit{CtrFMU.inst}(t) \end{array} \right) \,;\, \textbf{\textit{wait}}(t) \right) \right)^*
\end{aligned}
$$

The overall behaviour is given by the parametric timed relation $\mathit{WT}(\mathit{minl}, \mathit{maxl})$. Parameters *minl* and *maxl* represent the minimum and maximum level allowable by the water tank system, and correspond to the design parameters of this particular example. In general we can make FMI networks parametric over the design parameters. The two FMUs are modelled as timed relations, but they can be defined in different UTP theories and then mapped to timed relations. This is the requirement we have for composing heterogeneous system models.

Next, we will show how lift both VDM-RT and Modelica models into such an FMI composition.

5 Lifting VDM-RT

In D2.2b [11] we showed how to give a semantics to VDM-RT in terms of timed reactive designs and the language ***CML***. This semantics is quite complex, and deals with all aspects of the VDM-RT language, including features not used in the INTO-CPS case studies, such as concurrency. The majority of case studies and pilot studies have controllers whose behaviour is defined by a single periodic thread which executes certain stateful behaviour every time a period of time elapses. Thus, in order to support models more amenable to verification we will give a direct semantics to periodic threads in timed relations that will then be directly usable in our relational FMI semantics. We note, though, that the more complex semantics can also be supported if required since ***CML*** timed reactive designs can, in theory, be collapsed to timed relations.

In order to model periodic threads, we need to add a further observational variable: $\mathit{ctdown} : \mathbb{R}_{\geq 0}$. This is used to model the periodic thread countdown timer in the state and is required for a VDM-RT FMU. Essentially, whenever such an FMU is instructed to step forward, any remaining time on the countdown timer is first removed from the timer. If the time step causes the countdown timer to become 0, then the behaviour of the thread is executed and the countdown timer is reset. For the purposes of simplicity, we assume that the code of the periodic thread can be executed instantaneously. If delays are required then they can be inserted using the ***wait*** operator. We also disallow VDM-RT FMUs from being stepped forward beyond the remaining countdown time, as this is usually when input variables and read and output variables are written to, which information must be shared with sibling FMUs.

We then give the following definition for a VDM-RT periodic thread.

Definition 5.1 (Periodic Thread)**.**

$$\textbf{\textit{periodic}}(n, P) \triangleq \left(\begin{array}{l} (\textbf{\textit{wait}}(ctdown)\ ;\ P\ ;\ ctdown :=_r n) \sqcap \\ (\bigsqcap t \mid 0 < t \wedge t < ctdown \bullet ctdown :=_r ctdown - t\ ;\ \textbf{\textit{wait}}(t)) \end{array} \right)$$

A periodic thread is parametrised by the period $n : \mathbb{R}_{\geq 0}$ and a timed relation P, representing the body of the thread. There are two possible behaviours which are combined using the non-deterministic choice operator ($\sqcap$). The first option is to wait for the entirety of the remaining periodic countdown, execute the body of the perioidic thread P, and then reset the countdown time to n. Alternatively, the countdown timer can be reduced by any amount of time strictly greater than 0 and strictly less than the remaining periodic countdown. Any time step greater than the periodic countdown timer is thus excluded, and an attempt to do so will result in miraculous behaviour.

With this semantics for periodic threads, we can now give a semantics to the water tank controller. The main body of the VDM-RT controller is implemented by the following code[1], which is the body of a period thread with a period of 1 ms.

```
let level : real = levelSensor.getLevel()
in
(
  if (level >= HardwareInterface`maxlevel.getValue())
  then valveActuator.setValve(open);

  if (level <= HardwareInterface`minlevel.getValue())
  then valveActuator.setValve(close);
);
```

This code first obtains the present level from the level sensor, and checks its present value. If it is at or above the maximum threshold, then the valve is opened. If it is at, or below the minimum, the valve is closed. We add an error of 0.1 onto each of the thresholds to allow for the sensor latency. This behaviour can be represented by the following timed relations.

Definition 5.2 (Controller FMU)**.**

$$CtrFMU(minl, maxl).inst \triangleq ctdown :=_r 0.001$$

$$CtrFMU(minl, maxl).step \triangleq \textbf{\textit{periodic}} \left(0.001,\ \begin{array}{l} valveOn :=_r false \\ \quad \lhd levelSensor \leq minl + 0.1 \rhd \\ valveOn :=_r true \\ \quad \lhd levelSensor \geq maxl - 0.1 \rhd \mathit{II}_r \end{array} \right)$$

The links between the VDM-RT FMU and the the other FMU networks, which in the code above are provided by different VDM-RT classes and objects, are here abstracted to assignments to connector variables. The instantiation relation simply assigns an initial

[1] See `https://github.com/into-cps/case-study_single_watertank/blob/master/Models/SingleWT/Controller.vdmrt`

value to the countdown timer of 0.001, to set the initial period. The step relation implements the body of the periodic thread. Every millisecond the controller checks the status of the water level from the input variable *levelSensor*, whose value is provided by *Wiring*. If it is above the maximum, the output variable *valveOn* is assigned the value *true*. If it is below the minimum, *valveOn* is assigned the value *false*. Otherwise, all variables remain the same. Though a simple example, all the operators of the relational calculus can be utilised to formally encode imperative controller models in this way.

6 Lifting Modelica

In D2.3c [6] we give a semantics to Modelica in terms of the hybrid relational calculus and reactive designs. A hybrid relation is a form of reactive relation where the trace is a piecewise continuous function over the continuous state. Such a timed trace records at every instant the value of every continuous variable in the state space. We describe a simplified water tank with a linear ODE to represent the level in Modelica as below.

```
model Tank
  RealInput level;
  BooleanInput valve;

equation
  der(level) = if valve then 1 else -1;

end Tank;
```

This behaviour of the simplified water tank is given by the following hybrid relation.

Definition 6.1 (Water Tank Hybrid Relation)**.**

$$Tank \triangleq \langle\, level : (\lambda\, l \bullet -1) \,\rangle \lhd valve \rhd \langle\, level : (\lambda\, l \bullet 1) \,\rangle$$

If *valve* is initially true then the water level should be dropping and so we assign the derivative -1 using the ODE operator $\langle\, x : f \,\rangle$. This is a slightly specialised version of the operator given in D2.3c in that it also has an implicit frame attached. Specifically, any variable other than x is held constant during evolution. If the valve is off, then the water level will rise. Naturally, much more complicated ODEs can be assigned in a realistic system.

For FMI, a hybrid relation encodes more information than is necessary to perform time steps. Thus it is necessary to convert a hybrid relation to a timed relation to enable integration of Modelica models into relational FMI. We first define the following function, which allows us to step a hybrid relation a given time period t.

Definition 6.2 (Stepping a Hybrid Relation)**.**

$$\textbf{\textit{HyStep}}_t(P) \triangleq ((P \wedge \ell = t \wedge \textbf{\textit{rl}}(\textbf{\textit{v}}) \wedge \textbf{\textit{d}} = \textbf{\textit{d}}') \upharpoonright_r st{:}\textbf{\textit{c}})$$

Function $\textbf{\textit{HyStep}}_t(P)$ produces an untimed relation between inputs and outputs that effectively encodes the possible behaviours of the model at the time step t, subject to

particular inputs. This function simply takes the value of continuous variables at the end of an evolution, ignoring intermediate values, combined with a record of the evolution length. The relation can be primed with initial values for the variables in order to obtain the outputs at time instant t.

The definition is quite complex and uses additional operators for manipulating alphabets of the underlying relations. We first conjoin the hybrid relation P with three statements: (1) the duration length ℓ is the same as t, (2) each relational output variable is equated with the right limit of the corresponding trajectory variable, and (3) the discrete variables ($\boldsymbol{d}$) are left unchanged. We then restrict the alphabet to the continuous variables using relational alphabet restriction operator $\restriction_r$. The result is a relation consisting of only continuous state variables and no observational variables.

We then use the stepping function to define a further function ***H2T*** that converts a hybrid relation to a timed relation.

Definition 6.3 (Hybrid Relations to Timed Relations)**.**

$$\textbf{\textit{H2T}}(P) \triangleq \left(\prod t \bullet \textbf{\textit{wait}}(t) \fatsemi \textbf{\textit{HyStep}}_t(P)\right)$$

Function ***H2T*** non-deterministically selects a time step, introduces a delay of this length, and then steps the hybrid relation using ***HyStep***. The result then is a timed relation which relates the Modelica FMU's inputs to its output and the time expended.

As an example, consider the hybrid relation $\underline{x} = 5 * ti$, which states that continuous variable x at each instant has the value $5 * ti$. If we apply ***H2T*** to it, we obtain the timed relation $x' = 5 * (time' - time)$. Its behaviour is simply to assign the final value of the continuous variable at each time instant.

We can prove a useful set of laws for ***H2T***.

Theorem 6.1 (***H2T*** Identities)**.**

$$\begin{aligned}
\textbf{\textit{H2T}}(\mathit{II}_r) &= \mathit{II}_r \\
\textbf{\textit{H2T}}(P \lhd b \rhd Q) &= \textbf{\textit{H2T}}(P) \lhd b \rhd \textbf{\textit{H2T}}(Q) \\
\textbf{\textit{H2T}}(\langle \sigma \rangle_h) &= \langle \sigma \rangle_r \\
\textbf{\textit{H2T}}(\{x \mapsto f(\boldsymbol{v}, ti)\}_h) &= \left(\prod t \mid t > 0 \bullet \textbf{\textit{wait}}(t) \fatsemi x :=_r f(\boldsymbol{v}, t)\right)
\end{aligned}$$

H2T has the expected effect on a relational identity, and distributes through conditional. $\langle \sigma \rangle_h$ is a version of assignment where only continuous variables are assigned, and it maps directly to timed relation assignment. Finally, we have the case for an evolution statement $\{x \mapsto f(\boldsymbol{v}, ti)\}_h$ which states that x at every instant is determined by a given function on the initial value of the state $\boldsymbol{v}$ and time ti. The resulting timed relation non-deterministically chooses a non-zero amount of time to progress, inserts a delay of this period, and finally assigns to x the result of f at t with initial value $\boldsymbol{v}$.

Using this function, we can now define the Tank FMU for our multi-model.

Definition 6.4 (Tank FMU)**.**

$$TankFMU.inst \triangleq \mathit{II}_r$$
$$TankFMU.step \triangleq \textbf{\textit{H2T}}(Tank)$$

The instantiation for a Modelica FMU is simply II_r as there is no need to set internal variables. Then, applying the laws from Theorem 6.1 and relational calculus, we can calculate the following result.

Theorem 6.2 (Tank FMU Calculation)**.**

$$TankFMU.step = \left(\prod t \mid t > 0 \bullet \textbf{\textit{wait}}(t) \; ; \; (level :=_r level + t \lhd valve \rhd level :=_r level - t)\right)$$

The behaviour of this FMU is to step forward a strictly positive amount of time, and depending on the status of the value either increment or decrement level by t. Thus, our multi-model can now be entirely expressed in the language of timed relations, and subjected to verification using Hoare logic as detailed in deliverable D2.3a [19].

7 Multi-Model Verification

In this section we show how the formal multi-model developed in the above sections can be subjected to verification using Hoare logic. A Hoare logic triple has the form $\{\, p \,\}\, S \,\{\, q \,\}$, where S is a block of program code, and p and q are predicates on the program's state variables. The Hoare triple states that, if S is started in a state that satisfies predicate p, then when it terminates the final state will satisfy predicate q. For example, the statement $\{\, x \geq Y \,\}\, x := x + 1 \,\{\, x > Y \,\}$ is provable since if we knew that x was greater or equal to Y initially, and then we increment x by 1, then x must be greater than Y in the final state.

We have developed a Hoare logic for timed relations which allows us to prove properties of multi-models in Isabelle/UTP. The main contribution is a tactic called `fmi_hoare` that uses the rules of Hoare logic to break a desired specification of an FMI network into a collection of verification conditions for discharge. The laws we make use of are standard for Hoare logic, and in particular they can be found in related UTP publications [13, 4]. The novelty lies in both the mechanisation of the multi-model, and the tactic that enables verification.

We describe the definition of the water tank example multi-model itself in Isabelle/UTP, in terms of its principal types and definitions. We will then prove a safety property of the example using our FMI proof tactic.

Figure 1 shows the principle state space (alphabet) types for the two constituent FMUs, which are called `tank_st` and `ctr_st`, and and the overall FMI network state space, `wt_st`, which composes the two constituent state spaces. The controller state space, `ctr_st`, extends the VDM-RT state space template `vrt_st` which adds the special `ctdown` variable used to schedule executions of the periodic thread. Figure 2 shows the initialisation and wiring topology for the FMI network. The initialisation sets initial values for each of the variables in the overall state space, `wt_st`. The wiring connects

```
alphabet tank_st =
  valve   :: bool
  level   :: real

alphabet ctr_st = vrt_st +
  valveOn     :: bool
  levelSensor :: real

alphabet wt_st =
  tank  :: tank_st
  ctr   :: "ctr_st vrt_st_scheme"
```

Figure 1: Water tank state types

```
definition
  "Init = [ &tank:valve      ↦s true
          , &tank:level      ↦s 0
          , &ctr:valveOn     ↦s true
          , &ctr:levelSensor ↦s 0 ]"

definition
  "Wiring = [ &tank:valve      ↦s &ctr:valveOn
            , &ctr:levelSensor ↦s &tank:level ]"
```

Figure 2: Water tank topology and initialisation

```
definition Tank  :: "tank_st fmu" where
  "Tank = Modelica_FMU (⟨&level : λlev. -1⟩h ◁ &valve ▷h ⟨&level : λlev. 1⟩h)"

definition Ctr :: "real ⇒ real ⇒ ctr_st vrt_st_ext fmu" where
"Ctr minLevel maxLevel =
  VDMRT_FMU 0.001 (valveOn := false ◁ &levelSensor ≤u «minLevel + 0.1» ▷r
                   valveOn := true ◁ &levelSensor ≥u «maxLevel - 0.1» ▷r II)"

definition WaterTank :: "real ⇒ real ⇒ wt_st trel" where
  "WaterTank minLevel maxLevel =
     FMI Init [FMU[ctr, Ctr minLevel maxLevel], FMU[tank, Tank]] (FixedStep 0.001) Wiring"
```

Figure 3: Water tank definitions

FMU outputs to FMU inputs. In this case it connects the controller output variable `valveOn` to the tank input variable `valve`, and tank output `level` to controller input `levelSensor`.

Figure 3 has the definition of each of the FMUs and the overall network. FMU `Tank` is constructed using the function `Modelica_FMU`, which takes a hybrid relation as input, and produces a timed relation. Internally this function is implemented using the ***H2T*** function we described in Section 6. FMU `Ctr` is a parametric FMU with the minimum and maximum levels as design parameters. It is constructed using the function `VDMRT_FMU`, which is uses the ***periodic*** function from Section 5 to construct a periodic thread FMU with a period of 1 millisecond. Finally, the actual FMI network is constructed using the function `FMI`, which takes as input the initialisation, a list of FMUs, a master algorithm specification, and a wiring relation. The function `FMI[s, P]` takes as input the state-space `s` for the given FMU, and its behavioural specification `P`. Since `ctr` and `tank` are disjoint variables of the state space, these two FMUs can be executed in any order without risk of incorrect interference.

Figure 4 then shows how we go about proving properties of an FMU network using our mechanised Hoare logic. The property we want to prove is `wt_safe` at the bottom which states that, given minimum and maximum levels of 0 and 10, respectively, the tank level should never go above 10. The precondition is *true* because the instantiation and initialisation of the FMUs provides starting values for all state variables, and therefore we need assume nothing. The postcondition is effectively quantified over every evolution of the system allowed by the FMUs and master algorithm. Thus, this specification states that for any evolution it should never be the case that the tank overflows 10.

```
abbreviation WTI :: "wt_st upred" where
  "WTI ≡ &ctr:ctdown =u 0.001 ∧
         ((&tank:level <u 9.9) ∨
          (&tank:level ≥u 9.9 ∧ &tank:level <u 9.95 ∧ ¬ &ctr:valveOn) ∨
          (&tank:level ≥u 9.9 ∧ &tank:level <u 10 ∧ &ctr:valveOn))"

lemma wt_lemma_1:
  "{true} WaterTank 0 10 {WTI}r"
  unfolding WaterTank_def
  by (fmi_hoare defs: Tank_def Ctr_def Init_def Wiring_def tank_ode_1_evolve tank_ode_2_evolve)

lemma wt_safe:
  "{true} WaterTank 0 10 {&tank:level <u 10}r"
  by (rule hoare_rp_strengthen[where q'="WTI"])
     (rel_auto, rule wt_lemma_1)
```

Figure 4: Water tank verification

In order to prove this, we first have to observe that the controller will not respond to the tank reaching its maximum until up to 1ms after this occurs. This is due to the periodic nature of the VDM-RT thread, and this is why we have subtracted an error of 0.1 from the maximum level. Thus, when the water level reaches 9.9, the sensor will detect this and start to act, but the tank will rise a little more. Of course the dynamics of this model is very simple, and more complex dynamics would lead to a more realistic system and challenging verification problem.

As usual for Hoare logic proofs, we cannot directly use the property of $level < 10$ as an invariant of the system, as it is too weak. Specifically, just knowing that, at a given instant, the water level is less than 10, does not guarantee that it will also be at the next instant. The controller, for example, may not respond quick enough if the level is 9.99999, which is closer to the level than the rate at which it rises in 1ms, but still satisfies the invariant. Thus, we need an enriched invariant that characterises the states of the system more precisely, and in particular accounts for the behaviour of the controller.

`WTI` is an invariant of the water tanks system that can be directly proved. It states that the system must be in one of three states:

1. The water level is well below the limit of 10 ($level < 9.9$);
2. The water level is getting close to the limit ($9.9 < level < 9.95$), but the valve has not yet turned on ($\neg valveOn$);
3. The water level is almost, but not quite at the limit ($9.9 < level < 10$), and the valve has been switched on.

The crucial point to observe is there is a region $9.95 < level < 10$ where the valve must have been turned on, and consequently the water level must be dropping even if we enter that region. As a result the system dynamics will not allow the water level to overflow. The region below this ($9.9 < level < 9.95$) is a safe region in which water level could still rise, but it can't enter the former region without the valve controller responding. Since the system can only be in one of these three states, and each of them satisfies our original property $level < 10$, this is a sufficient invariant for us to prove. Note that we also have to conjoin the requirement that $ctdown = 0.001$, that is, at the beginning of every cycle the periodic thread needs to expend 0.001 seconds before it can execute. This is true

because we have chosen a fixed-step master algorithm. A variable step would require a more complex assumption about the periodic countdown timer.

Once the invariant is formulated in `WTI`, we can then attempt to discharge it using our tactic `fmi_hoare`. This takes as parameters the collection of all definitional equations, and then applies Hoare logic deduction rules and relational calculus to simplify the resulting verification conditions. In this instance, the resulting arithmetic properties of the system dynamics are sufficiently simple that Isabelle/HOL can discharge them automatically, and thus we show that the system satisfies `WTI`.

Finally, we can return to our original property. We first strengthen the postcondition to be `WTI`. This requires that we prove $WTI \Rightarrow level < 10$, which is indeed provable, and we discharge this using the relational calculus tactic, `rel-auto`. Finally, we can make use of the invariant property lemma, `wt_lemma_1`, to discharge the proof goal. Thus we have shown how to perform multi-model verification with heterogeneous models.

8 Integration of Additional Notations

In this section we provide guidelines for integration of additional notations into our foundational semantics model. Our relational model of FMI is fundamentally open in nature, and can be extended by additional notations and languages. The semantic domain for this is our UTP theory of timed relations, and it is necessary to map models into this domain.

In the, UTP semantic meta-models are constructed by composition of one or more UTP theories that characterise the underlying paradigms of the language. For example, the Modelica language is formalised in terms of hybrid reactive designs, a theory which combines a theory of imperative programs with termination (reactive designs), and a theory of hybrid computation (hybrid relations). There are therefore three steps to integrating a new language in this UTP setting.

(1) Theory Engineering. The language's underlying paradigms must be formalised and mechanised as UTP theories. This is the most difficult step of integration, as it requires in-depth knowledge of the mathematics that lie behind the language. However, there already exist a large number of UTP theories for various paradigms which have been developed by the UTP community. For example, there are UTP theories for concurrency [13, 5], object-orientation [17], real-time programming [18], hybrid computation [9], and probability [2], to name a few. In all likely-hood, therefore, UTP theories already exist that can be used to construct a denotational semantics for the given language. Indeed, this is the "theory supermarket" idea: that a semantics engineer can simply pick existing theories off the shelf to support construction of verification tools for their target language.

(2) Denotational Semantics. Once a semantic model has been fixed, the syntax and semantics of the language should be defined and mechanised in Isabelle/UTP. This involves giving denotations for each of the language operators in terms of the target semantic model. UTP theories invariably import standard relational operators like sequential composition, whilst creating specific operators for given paradigms, and so many such operators may be directly usable. This is the case for the hybrid relational calculus,

where the specialised operators include ODEs, continuous function evolution, and evolution interruption. Hybrid relational calculus, though, is effectively an intermediary UTP theory and the actual language we give semantics to there is Modelica, of course. Deliverable D2.3b shows how to give the Modelica block language a semantics as a number of block constructors an composition functions, the target of which is a hybrid relation. Technically at this point one can also automate the translation from the source language's abstract syntax tree into Isabelle/UTP definitions. This would allow us to subject the individual models to theorem proving, and pave the way for multi-model verification as well.

(3) Theory Linking. Finally, a function mapping the source semantic model into the theory of timed relations must be defined, like ***H2T***. In many instances such a function will be injective if the model can be losslessly encoded as a timed relation. This is the case, for example, with VDM-RT models. However, inevitably this may involve loss of some information about the model since FMI is fundamentally discrete time in nature, in that it only permits observation of a model at particular instants and not continuously. Lossy functions may be Galois connections [13], rather than injections. The linking function should demonstrate how elements of the source theory should be represented as timed relations. For Modelica this involves throwing away the trajectory information, and recording all possible snapshots of the continuous state for any given time step from a given initial state, in the function ***H2T***. Once this behaviour has been defined, cases for the different operators of the language can be proved as theorems, as shown for example in Theorem 6.1. This then gives the stepping behaviour of the individual FMUs.

Once all of these three activities are completed, the source language can be integrated into an FMI multi-model. Moreover, when one comes to prove theorems about multi-models we can proceed by decomposing the required invariant into individual FMU invariants. These FMU invariants can then be converted, through the prism of the mapping function, to a property of the individual model. This therefore enables both integration, and a holistic multi-model verification technique.

9 Integration of Additional Tools

In this section we provide complementary information on integrating additional tools into the tool-chain, building on the formal approach highlighted in Section 8. Suppose that we want to include a new tool in the INTO-CPS tool chain. There are two aspects to this: adding a new solver to the FMI architecture used for co-simulation; and adding a new notation to the foundations for INTO-CPS. We address both.

1. Our aim is to be able to include the new tool in our FMI architecture for co-simulation. For that, we require formal semantics for both (a) continuous-time and (b) reactive behaviours. The former (a) is to account for the model of FMUs, already expressed in notations such as Modelica, VDM-RT and 20-sim used by INTO-CPS, while the latter (b) accounts for the semantics of the FMI co-simulation itself, which is a reactive system. An additional aim, from the foundations work package, is to be able to prove universal properties of FMI architectures and co-simulations, for a possibly infinite set of initial values and scenarios; for this we need formal semantics.

2. Aspects (a) and (b), however, do not have to be supported within the *same* semantic theory. Since co-simulation decouples Master Algorithms (Mas) from simulated components, both of them can live in the realm of their own semantic theory. An important exception to this is HIL scenarios, where the execution time of the simulator is also important. In such a setting, (b) would also have to support reasoning about discrete time.

3. We should distinguish between easy (decidable) and difficult (non-decidable) properties of co-simulations. Decidable properties include the absence of algebraic loops, type conformance, non-divergence of MA models, and the solution of certain kinds of differential equations. Non-decidable properties include high-level safety properties and the conformance of MA implementations to their specification (in the general case, at least). The formal semantics for the new tool should be mechanised and equipped with a set of rules to evaluate decidable properties, as well as enable reasoning about non-decidable properties. Our work shows that code generation and evaluation are valuable techniques to deal with emergent complexity of models and the overhead of the proof system in reasoning about them.

4. With respect to foundational work, the formal semantics for the new tool ought to be equipped with a refinement calculus, in order to enable proofs about co-simulations, and in particular to modularise proofs that consider discrete, continuous, and reactive properties and aspects of a model. There are three major phases in this refinement approach. Discrete abstractions of models are used to prove safety properties, and are then refined into continuous models that preserve them. The co-simulation state is still centralised at that level, and data is exchanged through shared variables. The MA is introduced as yet another refinement step; namely, to distribute shared data between FMUs.

5. As the problem of reasoning about co-simulations is highly non-trivial, the key challenge a tool must address is abstraction and modularisation of proofs. The question of building semantics cannot be decoupled from the strategy to reason about co-simulations. Each of the above-mentioned stages will require their own laws and strategies that a tool should be aware of to assist the developer. The UTP and Isabelle/UTP proof tool have been shown to address this issue very well, by offering a large spectrum of semantic theories and proof strategies that could potentially be employed and linked to each other.

As an example of our methodology, consider adding an additional modelling paradigm to the INTO-CPS toolchain. As a concrete example, we addressed the task of adding probabilistic modelling and analysis. The context was the multi-modelling of robot controllers. We have already linked INTO-CPS to the UK EPSRC project, RoboCalc (see deliverable D2.1a [1] and [3]), where we restricted and extended the INTO-SysML profile for cyber-physical systems, implemented in RoboTool for modelling and analysis of RoboChart diagrams.

We extended the INTO-SysML/RoboChart language with probabilistic semantics to be analysed adding a probabilistic model checker to RoboTool. This required new semantics for the extensions and for the translation to existing probabilistic model checkers. A number of important tools use the same input format (the Reactive Modules formalism) for a range of different tools, including PRISM, IsCAS, Storm, LiQour, and MRMC.

The input format has different semantics that cover deterministic and nondeterministic Markov processes, as well as discrete and continuous time probabilistic automata.

Collaboration with the Federal University of Pernambuco in Brazil has led to implementing the translation from the new semantics, so that RoboChart diagrams are mapped to the probabilistic guarded command language, and from there to PRISM. This gives access to five different model checkers. Another route goes via the FDR model checker and on to the PRISM formalism, giving a total of 10 different routes from model syntax to tools and the different semantics for these models. The results have different analysis characteristics. The advantage of this is that a problem can be matched to the best semantics and analysis technique. Future work will address theorem proving as well as model checking.

This work was carried out as part of the EPSRC RoboCalc project [2] and the Royal Academy of Engineering's Newton Fund project on Robotics and Autonomous Systems in Brazil.

10 Conclusion

In this deliverable we have shown how the UTP's theory linking capabilities can be used to compose heterogeneous models to produce formal semantics for a heterogeneous FMI multi-model. The common semantic domain for this in our theory of timed relations that shows how variables of a model change for a given time step. Provided that a language supports output into this domain, we can formally link them together in a multi-model and subject them to verification using Hoare logic. All of theory described herein has also been mechanised in Isabelle/UTP [10].

There are several areas for future work. Currently we have shown how two different languages can be formally integrated using FMI, and in the future it would be useful to support additional notations and tools using the approaches outlines in Sections 8 and 9.

The integration and verification technique we have shown for FMI could be applied to larger examples, with more complex invariants in order to gain further experience. This would also allow us to assess the scalability of our verification platform for larger examples. Likely there will be a need for additional proof tactics and tools that deal efficiently with different classes of continuous mathematical problems that arise from such verifications, possibly building on technical results from related tools [16]. Moreover, our requirement language for FMI is currently limited to state invariants, and it would be useful to consider more expressive notations, such as modal and temporal logics.

From a technical standpoint, our verification tool, though powerful, is currently academic in nature. Thus, it would be useful to develop appropriate high-level tool support for this technology for industrialists, including model transformations from FMI, VDM-RT, Modelica, and other languages, into Isabelle/UTP in order to automate the verification process.

[2] RoboCalc Website: `https://www.cs.york.ac.uk/circus/RoboCalc/`

References

[1] Nuno Amalio, Richard Payne, Ana Cavalcanti, and Etienne Brosse. Foundations of the SysML profile for CPS modelling. Technical report, INTO-CPS Deliverable, D2.1a, December 2015.

[2] R. Bresciani and A. Butterfield. A UTP semantics of pGCL as a homogeneous relation. In *Proc. Integrated Formal Methods (IFM)*, volume 7321 of *LNCS*. Springer, 2012.

[3] A. Cavalcanti, A. Miyazawa, R. Payne, and J. Woodcock. Sound simulation and co-simulation for robotics. In *Present and Ulterior Software Engineering*, pages 173–194. Springer, September 2017.

[4] A. Cavalcanti and J. Woodcock. A tutorial introduction to designs in unifying theories of programming. In *Proc. 4th Intl. Conf. on Integrated Formal Methods (IFM)*, volume 2999 of *LNCS*, pages 40–66. Springer, 2004.

[5] A. Cavalcanti and J. Woodcock. A tutorial introduction to CSP in unifying theories of programming. In *Refinement Techniques in Software Engineering*, volume 3167 of *LNCS*, pages 220–268. Springer, 2006.

[6] Ana Cavalcanti, Simon Foster, Bernhard Thiele, and Jim Woodcock. Initial semantics of Modelica. Technical report, INTO-CPS Deliverable, D2.2c, December 2016.

[7] Ana Cavalcanti, Simon Foster, Bernhard Thiele, Jim Woodcock, and Frank Zeyda. Final Semantics of Modelica. Technical report, INTO-CPS Deliverable, D2.3b, December 2017.

[8] Luis Diogo Couto, Pasquale Antonante, Stylianos Basagiannis, Sara Falleni, Hassan Ridouane, Hajer Saada, Erica Zavaglio, and James Baxter. Building Case Study 3, (Confidential). Technical report, INTO-CPS Confidential Deliverable, D1.3d, December 2017.

[9] S. Foster, B. Thiele, A. Cavalcanti, and J. Woodcock. Towards a UTP semantics for Modelica. In *Proc. 6th Intl. Symp. on Unifying Theories of Programming*, volume 10134 of *LNCS*. Springer, June 2016. To appear.

[10] S. Foster, F. Zeyda, and J. Woodcock. Unifying heterogeneous state-spaces with lenses. In *Proc. 13th Intl. Conf. on Theoretical Aspects of Computing (ICTAC)*, volume 9965 of *LNCS*. Springer, 2016.

[11] Simon Foster, Ana Cavalcanti, Samuel Canham, Ken Pierce, and Jim Woodcock. Final Semantics of VDM-RT. Technical report, INTO-CPS Deliverable, D2.2b, December 2016.

[12] T. Hoare, I. Hayes, J. He, C. Morgan, A. Roscoe, J. Sanders, I. Sørensen, J. Spivey, and B. Sufrin. The laws of programming. *Communications of the ACM*, 30(8):672–687, August 1987.

[13] Tony Hoare and Jifeng He. *Unifying Theories of Programming*. Prentice-Hall, 1998.

[14] M. V. M. Oliveira, A. L. C. Cavalcanti, and J. C. P. Woodcock. A UTP Semantics for Circus. *Formal Aspects of Computing*, **21**(1):3 – 32, 2007.

[15] Julien Ouy and Thierry Lecomte. Railways Case Study 3, (Confidential). Technical report, INTO-CPS Confidential Deliverable, D1.3b, December 2017.

[16] André Platzer. *Logical Analysis of Hybrid Systems*. Springer, 2010.

[17] Thiago Santos, Ana Cavalcanti, and Augusto Sampaio. Object-Orientation in the UTP. In S. Dunne and B. Stoddart, editors, *UTP 2006: First International Symposium on Unifying Theories of Programming*, volume 4010 of *LNCS*, pages 20–38. Springer-Verlag, 2006.

[18] Kun Wei, Jim Woodcock, and Ana Cavalcanti. Circus Time with Reactive Designs. In *Unifying Theories of Programming*, volume 7681 of *LNCS*, pages 68–87. Springer, 2013.

[19] Frank Zeyda, Ana Cavalcanti, Jim Woodcock, and Julien Ouy. SysML Foundations: Case Study. Technical report, INTO-CPS Deliverable, D2.3a, December 2017.