



Grant Agreement: 644047

INtegrated TOol chain for model-based design of CPSs



## **Final Semantics of VDM-RT**

Technical Note Number: D2.2b

Version: 1.1

Date: December 2016

Public Document

<http://into-cps.au.dk>

**Contributors:**

Simon Foster, UY  
Ana Cavalcanti, UY  
Samuel Canham, UY  
Ken Pierce, UY  
Jim Woodcock, UY

**Editors:**

Simon Foster, UY

**Reviewers:**

Victor Bandur, AU  
Bernhard Thiele, LIU  
John Fitzgerald, 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 | 10-06-2016 | Simon Foster | Initial document version            |
| 0.2 | 09-08-2016 | Simon Foster | Completion of main semantic content |
| 1.0 | 29-10-2016 | Simon Foster | Final draft for review              |
| 1.1 | 05-12-2016 | Simon Foster | Final version for submission        |

## Abstract

The deliverable reports on our work towards the creation of a novel formal semantics for the real-time modelling language VDM-RT. The focus of this report is on the real-time aspects of the language, having previously dealt with the object-oriented aspects. We provide a denotational semantics for the real-time parts of the language in a semantic framework called Unifying Theories of Programming via an intermediate concurrent and real-time modelling language called *CML* (COMPASS Modelling Language). We create the core VDM-RT expression model, and use *CML* processes to describe the behaviour of classes, objects, threads, and busses. We then show how our semantics has been validated using the *CML* interpreter in the *Symphony* tool, and summarise the work towards mechanisation of the denotational model in Isabelle/HOL.

# Contents

|   |           |
|---|-----------|
| <b>Glossary</b>                                   | <b>7</b>  |
| <b>1 Introduction</b>                             | <b>8</b>  |
| <b>2 Background</b>                               | <b>9</b>  |
| 2.1 VDM-RT . . . . .                              | 9         |
| 2.2 Unifying Theories of Programming . . . . .    | 13        |
| 2.3 Reactive Designs . . . . .                    | 14        |
| 2.4 COMPASS Modelling Language . . . . .          | 16        |
| 2.5 Isabelle/UTP . . . . .                        | 19        |
| <b>3 CML Extension</b>                            | <b>21</b> |
| 3.1 Universe for VDM-SL . . . . .                 | 21        |
| 3.2 Expressions . . . . .                         | 23        |
| <b>4 VDM-RT Semantics in CML</b>                  | <b>24</b> |
| 4.1 Overview . . . . .                            | 24        |
| 4.2 Types . . . . .                               | 27        |
| 4.3 Channels . . . . .                            | 28        |
| 4.4 Classes and Objects . . . . .                 | 28        |
| 4.5 Class Threads . . . . .                       | 31        |
| 4.6 CPUs . . . . .                                | 32        |
| 4.7 Busses . . . . .                              | 33        |
| 4.8 Statements . . . . .                          | 35        |
| 4.9 System Instantiation . . . . .                | 38        |
| <b>5 Validation of semantics</b>                  | <b>40</b> |
| <b>6 Mechanisation in Isabelle/UTP</b>            | <b>42</b> |
| <b>7 Areas for future work</b>                    | <b>45</b> |
| 7.1 Dynamic topologies . . . . .                  | 45        |
| 7.2 Passive and active classes . . . . .          | 46        |
| 7.3 Timed specifications . . . . .                | 46        |
| 7.4 Model checking in FDR3 . . . . .              | 47        |
| 7.5 Timed state machines for RTT-MBT . . . . .    | 47        |
| 7.6 Encoding FMUs . . . . .                       | 48        |
| <b>8 Conclusion</b>                               | <b>48</b> |
| <b>A VDM-RT Symphony Model</b>                    | <b>48</b> |
| A.1 Infrastructure Processes . . . . .            | 48        |
| A.2 Water Tank Controller Instantiation . . . . . | 50        |
| <b>B Lenses</b>                                   | <b>53</b> |
| B.1 Introduction . . . . .                        | 53        |
| B.2 Background and related work . . . . .         | 54        |
| B.2.1 Unifying Theories of Programming . . . . .  | 54        |

|       |   |    |
|-------|---|----|
| B.2.2 | Isabelle/HOL . . . . .                      | 55 |
| B.2.3 | Mechanised state spaces . . . . .           | 56 |
| B.3   | Lenses . . . . .                            | 57 |
| B.3.1 | Lens laws . . . . .                         | 57 |
| B.3.2 | Concrete lenses . . . . .                   | 58 |
| B.3.3 | Lens algebraic operators . . . . .          | 58 |
| B.4   | Unifying state-space abstractions . . . . . | 61 |
| B.4.1 | Alphabetised predicate calculus . . . . .   | 61 |
| B.4.2 | Meta-logical operators . . . . .            | 63 |
| B.5   | Relational laws of programming . . . . .    | 65 |
| B.6   | Parallel-by-merge . . . . .                 | 66 |
| B.7   | Conclusions . . . . .                       | 68 |

## Glossary

|                   |  |
|-------------------|--|
| <i>CML</i>        | COMPASS Modelling Language, a formal language for modelling systems of systems based on <i>Circus</i> and VDM.             |
| <i>Circus</i>     | a formal modelling language for state-rich concurrent systems building on CSP and with a UTP semantics.                    |
| <i>CircusTime</i> | <i>Circus</i> extended with primitives for real-time modelling.  |
| CPU               | Central Processing Unit.   |
| CSP               | Communicating Sequential Processes, a process calculus created by Tony Hoare.  |
| CyPhyCircus       | a version of <i>Circus</i> for Cyber-Physical Systems modelling.   |
| FMI               | Functional Mockup Interface, a language for describing the composition of heterogeneous system models.                     |
| FMU               | Functional Mockup Unit, an encapsulated constituent model of an FMI network.   |
| HOL               | Higher Order Logic.  |
| Isabelle          | a generic proof-assistant usually associated with HOL.   |
| RTT-MBT           | A toolkit for model-based testing of real time systems including support for automated test generation and model checking. |
| SUT               | System Under Test.   |
| Symphony          | a development environment for CML, including a model simulator.  |
| UTP               | Unifying Theories of Programming, a framework for reasoning about formal semantics.  |
| VDM               | Vienna Development Method.   |
| VDM++             | Object-oriented dialect of VDM.  |
| VDM-RT            | Real-time dialect of VDM.  |
| VDM-SL            | Standardised dialect of imperative VDM.  |

# 1 Introduction

This deliverable supplements our previous semantics of object-oriented data structures [39] with a semantics for real-time threads in the real-time modelling language VDM-RT [47]. VDM-RT is a real-time dialect of the VDM formal modelling language [9] that can be applied to the specification of discrete controllers for Cyber-Physical Systems (CPSs). VDM-RT is object-oriented, with models defined as classes that are instantiated as objects. It supports concurrency through threading, and communication between threads through message passing over shared busses. The real-time features of the language comprise abstractions for deployment of objects to compute units (CPUs), which are connected by busses, and the time taken to evaluate expressions, which advance a global “wall clock” to predict the computation time of a model.

We use Hoare and He’s *Unifying Theories of Programming* [46, 18] (UTP) to give a denotational semantics to VDM-RT; details of the UTP can be found in our previous deliverables [39, 16]. VDM-RT is a discrete real-time language, which leads us to employ the UTP theory of timed reactive designs as the semantic model, as embodied in the COMPASS Modelling Language (*CML*). Thus, we use the constructs of *CML* to describe VDM-RT objects, threads, CPUs, and busses, together with actions that encode their orchestrated execution. In order to accomplish this, we also extend *CML* with a universe type for VDM-RT, and also timed expressions that cause language constructs like assignment to expend time during execution. Our aim for the final year of INTO-CPS is to integrate the *CML* constructs in our final target language, *CyPhyCircus*, which will additionally enable description of continuous time behaviour, and thus allow composition of VDM-RT discrete controllers with a continuous plant. Information about work towards creation of *CyPhyCircus* can be found in sister deliverable D2.2c [16].

Our semantics of VDM-RT is based on a pattern commonly employed in the INTO-CPS project to describe the discrete time component of a cyber-physical system. Such a “cyber component” consists of one or more controller objects, each of which owns a number of sensors and actuators through which to interact with the physical components. The topology of such a cyber component is thus fixed at instantiation, and there is no necessity to support dynamic object creation, which thus favours the use of static *CML* processes to represent objects and threads. Limiting ourselves to static topologies enables the application of static analysis techniques like model checking [40, 61, 6], which typically requires a tractable state space.

The structure of this deliverable is as follows. In Section 2 we give background for the deliverable, briefly describing VDM-RT, the UTP, and *CML*. In Section 3 we conservatively extend *CML* with the VDM-RT universe and timed expressions. In Section 4 we give our semantics of VDM-RT as a translation into *CML*. We employ *CML* processes to represent VDM-RT objects, CPUs, and busses. In Section 5 we provide some validation of our semantics through a hand translation of the water tanks VDM-RT controller to *CML*, and explore its simulation in the *CML* development environment, *Symphony*. In Section 6 we describe our work on mechanisation of the VDM-RT semantics in *Isabelle/UTP*. In Section 7 we explore a number of areas of future work for our semantics. Finally, in Section 8 we conclude the deliverable.



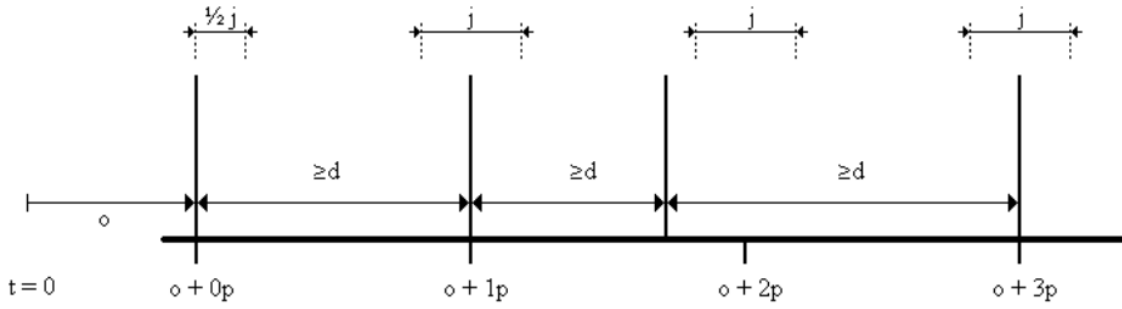


Figure 1: Periodic thread with period  $p$ , jitter  $j$ , delay  $d$ , and offset  $o$

## 2 Background

In this section we give the background to this deliverable, introducing VDM-RT, UTP, *CML*, and other foundational concepts.

### 2.1 VDM-RT

**Overview** The Vienna Development Method (VDM) is a state-based formal method that was originally designed in the 1970s to give semantics to programming languages [50]. Models in VDM have a persistent state, described through a rich set of datatypes (sets, sequences, mappings, etc.). Functionality is described through operations that modify the state. The core specification language, called VDM-SL, has been standardized as ISO/IEC 13817-1 [49]. As part of the standardisation process, a full denotational semantics has been defined for VDM-SL (due to Larsen et al. [53]), as well as a proof theory and comprehensive set of proof rules [8].

Models in VDM-SL can be structured into modules. Each module has its own state, which is global to the module, and functionality. Data and functionality can be exported and imported between modules.

In the 1990s a new dialect, called VDM++ [27], was defined adding object-orientation and concurrency features. VDM++ retained all the core features of VDM-SL (datatypes, operations, pre- and post-conditions, invariants, etc.) but replaced the notion of modules with classes and objects. In the 2000s another language extension was defined that included abstractions for modelling real-time embedded software [66]. This work led to the dialect used in INTO-CPS, VDM-RT (VDM Real Time).

There are two industrial-strength tools for VDM, the commercial VDMTools and the open source Overture<sup>1</sup>. Overture is used in INTO-CPS and also forms part of the Crescendo baseline technology. This latter allows co-simulation between VDM-RT and 20-sim models [28]. The main focus of VDM-RT is thus simulation, whereby the Overture interpreter steps a model forward a certain number of time units and calculates the resulting state of the system. This state can then be shared with a composed continuous model for the purpose of co-simulating an entire Cyber-Physical system.

<sup>1</sup><http://www.overturetool.org/>

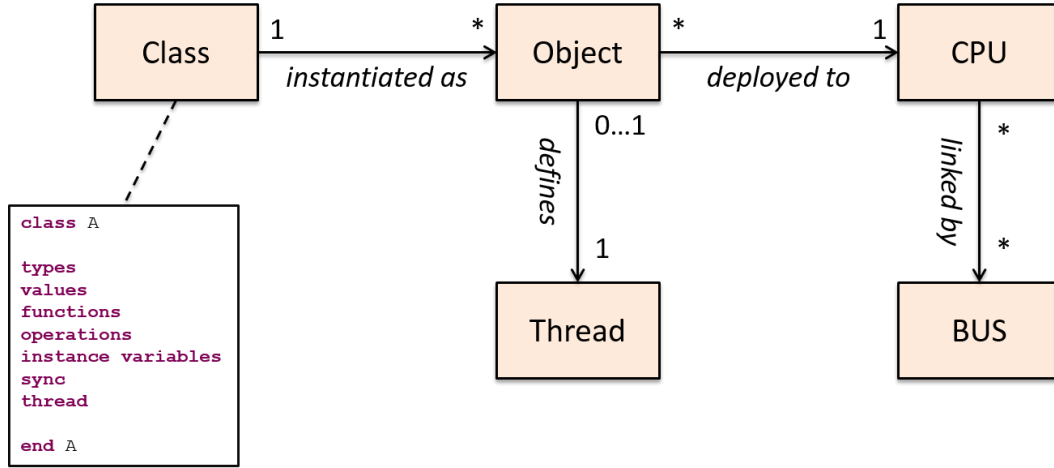


Figure 2: Relationship of VDM-RT concepts

All models in VDM-RT are built from classes, which are instantiated as objects. Variables can have a class as a datatype, or use the datatype system as defined in VDM-SL. Concurrency in VDM-RT is based on threads which can be executed on different modelled CPUs. Each class may define a thread, and once an object of that class is created, its thread can be started. A thread will terminate once its work is finished. There are also notations for defining threads that will call an operation periodically, as illustrated in Figure 1. Periodic threads execute every  $p$  time units, with an initial start time at offset  $o$ . Jitter  $j$  can be used to allow imprecision around the execution time of events, and delay  $d$  gives a minimal distance between two events. Threads can also be set to execute sporadically, that is, at non-deterministic intervals. Such real-time features of VDM-RT are based around a global “wall clock” that records the time, in nanoseconds, since the start of the simulation. All expressions in a model have an associated evaluation time, which causes the clock to advance.

VDM-RT has built-in abstractions for compute nodes, represented by CPU objects. A special **System** class is used to define CPU objects. Other objects in the system can be *deployed* to a CPU. When objects on different CPUs communicate, they must do so via a BUS object, which incurs a time penalty. Both CPU and BUS objects have a notion of their own speed. A class diagram relating the key elements introduced here is given in Figure 2. Threads communicate via sharing of objects. When a thread makes a call to an object operation, the interpreter has to determine whether the object is local or remote. If it is local, then the operation can simply be executed in the context of the present CPU. However, if it is remote then a message must be conveyed to the remote CPU via a bus. The present thread’s execution is paused whilst the target operation is executed, assuming the operation is synchronous. Operations can also be specified to be asynchronous, meaning they do not return a value and the calling thread need not pause during its execution.

The amount of time each expression takes to evaluate (i.e. the amount of time by which the wall clock is updated) is, by default, two simulated cycles of the CPU. Objects deployed to faster CPUs will take less (simulated) time to execute. Similarly, for CPUs connected by faster buses, their objects will incur a smaller time penalty when communicating. The amount of time an expression, or set of expressions, takes to execute can be altered in two ways. Using a **cycles** statement, which can be used to increase or decrease the

```

class Controller

instance variables
  levelSensor    : LevelSensor;
  valveActuator  : ValveActuator;

operations
  public Controller : LevelSensor * ValveActuator ==> Controller
  Controller(l,v)==
    (levelSensor := l; valveActuator := v);

  private loop : () ==> ()
  loop() ==
    duration(40)
    (
      let level : real = levelSensor.getLevel()
      in
      (
        if (level >= HardwareInterface.maxlevel)
        then valveActuator.setValve(true);

        if (level <= HardwareInterface.minlevel)
        then valveActuator.setValve(false);
      );
    );

  thread periodic(10E6,0,0,0)(loop);

end Controller

```

Figure 3: Example controller class for Water Tanks pilot study in VDM-RT

simulated cycles, or using a **duration** expression, which directly sets the time taken in nanoseconds (independent of the CPU). This is useful when measurements can be made on real hardware in order make the timing predictions of the model more accurate.

Figure 3 shows an example VDM-RT class corresponding to a controller for a simple water tank system [?]. It has two instance variables that correspond to a sensor for checking the tank water level (**LevelSensor**) and an actuator for turning on and off an evacuation valve on the tank (**ValveActuator**). There are also two operations: **Controller**, which is the class constructor that sets initial values for the sensor and actuator, and **loop** that describes the periodic thread of the class, which is set to execute every 10 milliseconds. The body of **loop** has a specified execution time of 40 nanoseconds using the **duration** statement. Within this a local variable **level** is created that obtains the present tank level from the sensor. If the level goes above **maxLevel**, which is a static variable of the **HardwareInterface** class, then the valve is turned on, and when it drops below **minLevel** it is turned off. This example also serves to partly illustrate the VDM-RT pattern that we use for our semantics. We will discuss this in more detail in Section 4.1.

**Semantics** There are two existing formal semantics for VDM-RT, both of which are operational in nature. The first [47] gives a structural operational semantics to a simplified version of VDM-RT. This semantics describes the behaviour of real-time threads being concurrently executed on CPUs, and exchanging call messages locally and remotely via

busses. A particular focus of this semantics is co-simulation of a discrete time VDM-RT model with an external continuous time model. Execution of a VDM-RT model is divided into a number of variable length time slots ( $t \in \mathbb{R}_{\geq 0}$ ) which correspond to critical regions. The length of a given time slot is calculated by determining the minimal time step that all system threads are willing, by agreement, to collectively expend.

The semantics identifies two kinds of variables: *LVar* which are variables local to a thread, and *IOVar* which are variables shared with the continuous model. Writes by a thread to one or more *IOVars* are atomic, in the sense that only one thread in each time slot may make such writes. This is ensured by locking of an *IOVar* the first time a thread makes a write to it, and until the end of the time slot, the thread effectively has exclusive ownership. This semantics does not explicitly consider the state of individual objects, in that only threads have variable valuations associated with them. Moreover, periodic threads do not exhibit timing variance constraints like clock jitter. This semantics provides the basis of the Overture VDM-RT interpreter.

The second formal semantics [54] gives a more comprehensive operational semantics (61 structural operational rules, and more than 20 utility functions), in which most of the constructs of concrete VDM-RT are considered. It explicitly handles the state of objects, in addition to associated threads, and atomic synchronisation of the states at the end of duration statements. It considers evaluation times for expressions, which are factored in when executing operators like assignments, and adds concepts like clock jitter to periodic threads. Moreover it also identifies and fixes a compositionality problem with the prior semantics of the duration statement. Specifically, in the Overture VDM-RT interpreter, nested durations are effectively ignored, and the overall execution time is simply based on the top-level duration. Thus, the execution time of any statement can only be determined by its context, which can of course be arbitrary, and thus the semantics is partly non-compositional. The new semantic model [54] thus reinterprets the duration statement to represent an atomic deadline operator, such that nested durations must sum to less than the overall deadline. We discuss the implications of this change further in Section 4.8. We have mechanised this semantics as described in our previous deliverable [39], and also corrected a number of errors and inconsistencies in the rules. However, reasoning about this operational semantics at the level of statements is difficult since all operational deduction rules are highly context dependent.

In this deliverable, taking input from both existing semantics and also the VDM language manual [52], we provide a novel denotational semantics for VDM-RT based on the timed reactive language *CML*. Our denotational semantics allows us to prove algebraic laws about VDM-RT programs without relying upon induction over an abstract syntax tree. This, therefore, equips us with more reasoning capability for the purpose of mechanical program verification. There also exists a corresponding operational semantics for *CML*, which is derived from the denotational semantics [11]. Moreover, our semantic model is defined within the context of Hoare and He's Unifying Theories of Programming [46] (UTP). This allows it to be formally linked to other semantic models and theories, such as those for continuous time and hybrid systems.

$$P ; (Q ; R) = (P ; Q) ; R \quad (1)$$

$$P ; \mathbf{false} = \mathbf{false} \quad (2)$$

$$(P \triangleleft b \triangleright Q) ; R = (P ; R) \triangleleft b \triangleright (Q ; R) \quad (3)$$

$$\mathbf{while} \ b \ \mathbf{do} \ P = (P ; \mathbf{while} \ b \ \mathbf{do} \ P) \triangleleft b \triangleright \mathbb{I} \quad (4)$$

$$P ; Q = \exists x_0. P[x/x_0] ; P[x'/x_0] \quad (5)$$

$$(P \wedge b) ; Q = P ; (b' \wedge Q) \quad (6)$$

$$\mathbb{I}_{\{x, x'\} \cup A} = (x = x') \wedge \mathbb{I}_A \quad (7)$$

Table 2: UTP Algebraic Laws of Predicative Programming

## 2.2 Unifying Theories of Programming

Unifying Theories of Programming [46] (UTP) is a mathematical framework for describing and unifying the formal semantics of programming and modelling languages. It has previously been applied to the creation of semantic models for a variety of languages, including Safety-Critical Java [20], SysML [56], Simulink [14], and *CML* [71]. During these developments a large library of theories of programming has been built up, and we make use of these in our semantics for VDM-RT. Moreover, UTP enables us to describe formal links between VDM-RT and the other INTO-CPS notations, which will in turn allow us to have a tool-chain that is semantically well founded.

Programs in the UTP are given denotational semantics using *alphabetised predicates* ( $P$ ) that define the relation between before variables ( $x$ ) and after variables ( $x'$ ) in the predicate's alphabet  $\alpha(P)$ . The calculus provides the operators typical of first-order logic, such as connectives  $\wedge, \vee, \neg, \Rightarrow$  and quantification  $\forall x. P, \exists x. P, [P]$ , where  $[P]$  represents the universal closure of  $P$ , that is a universal quantification over all the variables in  $\alpha(P)$ . UTP predicates are ordered by a refinement partial order  $P \sqsubseteq Q$  that equates to universal closure of reverse implication  $[Q \Rightarrow P]$ . Detailed tutorials on the UTP alphabetised predicate calculus are available [18, 35], and so we concentrate only on the crucial elements here.

Imperative programs can be described using relational operators such as sequential composition  $P ; Q$ , if-then-else conditional  $P \triangleleft b \triangleright Q$  (for condition  $b$ ), non-deterministic choice  $\sqcap$ , assignment  $x :=_A v$  (for expression  $v$  and alphabet  $A$ ), and skip  $\mathbb{I}_A$  (do nothing and identify all variables) all of which are given predicative interpretations. For such imperative programs, the refinement operator  $P \sqsubseteq Q$  corresponds to behavioural refinement, where the refined program  $Q$  is more deterministic than  $P$ . This also induces a complete lattice on programs, where **true**, the most non-deterministic program represents the bottom of the lattice, and **false**, the miraculous program, is the top. Recursive and iterative constructions can then be specified using lattice and fixed-point operators, such as  $\sqcap, \mu X. P$ , and the derived **while**  $b$  **do**  $P$ . A collection of algebraic laws that can be proved about such imperative and predicate operators is shown in Table 2 (see [46] and [70]).

Law 1 demonstrates the associativity of sequential composition. Law 2 shows that the miraculous program **false** is a right annihilator of sequential composition. Law 3 shows how sequential composition distributes through if-then-else conditional. Law 4 shows how a while loop can be unfolded by making a copy of the body. Law 5 allows the extraction of an intermediate variable  $x_0$  in a sequential composition through the use of an existential quantification. Law 6 shows how a conjoined conditional predicate  $b$  can be transferred to a postcondition on the other side of the sequential composition.

Aside from such programming operators, denotations can also be given to assertional reasoning calculi such as the Hoare calculus triple  $\{p\}Q\{r\}$ , and weakest precondition calculus  $P \text{ wp } q$ . Moreover, UTP provides a way of linking operational semantics to denotational semantics [46] by describing the transition relation  $(\sigma, P) \rightarrow (\rho, Q)$ , for state configuration predicates  $\sigma$  and  $\rho$  and programs  $P$  and  $Q$ , as a refinement statement –  $\sigma' ; P \sqsubseteq \rho' ; Q$ . From such a definition we can derive a set of structural operational laws for our target language as theorems, and thus obtain an operational semantics that is sound with respect to the denotational semantics.

The UTP predicate calculus thus provides a rich language for both defining and reasoning about semantics of programs, specifications, and models, in the algebraic, denotational, and operational flavours. Building on the core imperative constructs, the UTP also allows the specification of more complex language aspects using UTP theories. A *UTP theory* isolates an aspect of a language, such as object orientation, real-time, or concurrency, to allow its independent study. A language's denotational semantics can then be constructed by composition of the underlying building block UTP theories.

This is important for Cyber-Physical Systems, which make use of a wide variety of heterogeneous programming and modelling paradigms [32]. In this deliverable we will build our semantics of VDM-RT's real-time threads on top the UTP theory of *timed reactive designs*, embodied in the formal modelling language *CML*, which we describe in Section 2.4.

## 2.3 Reactive Designs

VDM-RT programs are reactive in nature: the threads, objects, CPUs, and busses all have the ability to interact with their environment in various ways. Thus to give them a semantics we need a suitable UTP theory, which is provided by the theory of *reactive designs* [18, 60]. A reactive design, of the form  $\mathbf{R}(P \vdash Q)$ , is a specification for a reactive program consisting of an assumption  $P$  and commitment  $Q$ . The turnstile  $\vdash$  constructs a UTP design – a total correctness specification – and  $\mathbf{R}$  is a healthiness function that makes the specification reactive. The assumption and commitment are relations (or predicates) whose alphabet consists of the following variables.

- $wait, wait' : \mathbb{B}$  – describe whether the previous or current process, respectively, are in an intermediate state. An intermediate state occurs when a process is waiting for interaction with its environment, and when time is passing.
- $tr, tr' : \text{seq } \Sigma^E$  – describe the history of events that were performed after the previous or current process, respectively.  $\Sigma^E$  is the event alphabet of the process, that is the set of events, such as inputs and outputs, that the process can engage in.

$$\begin{aligned}
\mathbf{R1}(P) &\triangleq P \wedge tr \leq tr' \\
\mathbf{R2}(P) &\triangleq P[\langle \rangle, tr' - tr / tr, tr'] \triangleleft tr \leq tr' \triangleright P \\
\mathbf{R3}(P) &\triangleq \mathbb{I}_{rea} \triangleleft wait \triangleright P \\
\mathbf{R} &\triangleq \mathbf{R3} \circ \mathbf{R2} \circ \mathbf{R1} \\
\\ 
\mathbf{Chaos} &\triangleq \mathbf{R}(\mathbf{false} \vdash \mathbf{true}) \\
\mathbf{Miracle} &\triangleq \mathbf{R}(\mathbf{true} \vdash \mathbf{false}) \\
\mathbb{I}_{rea} &= \mathbf{R}(\mathbf{true} \vdash \mathbb{I}) \\
x :=_{rea} v &\triangleq \mathbf{R}(\mathbf{true} \vdash x := v)
\end{aligned}$$

Table 3: Reactive design definitions

- $v, v' : \Sigma$  – describe the valuation of program variables before and after the current program, respectively.  $\Sigma$  is the program alphabet.

The behaviour of these observational variables is constrained by the healthiness condition  $\mathbf{R}$ , seen above, which is a monotone idempotent function and itself consists of the composition of three healthiness conditions presented in Table 3.  $\mathbf{R1}$  states that the trace can only get longer, or more formally that  $tr$  is monotonically increasing.  $\mathbf{R2}$  states that a process is history independent: if we remove the history in  $tr$  the process still has the same behaviour. We remove the history by replacing  $tr$  with  $\langle \rangle$ , and  $tr'$  with  $tr' - tr$ , that is the trace of events the current process has contributed. Our presentation of  $\mathbf{R2}$  differs slightly from the literature, in that we only remove the history when  $tr \leq tr'$  as the conditional ensures. This modification retains the standard meaning, but ensures that  $\mathbf{R1}$  and  $\mathbf{R2}$  are independent.

$\mathbf{R1}$  and  $\mathbf{R2}$  together ensure that the reactive behaviour of a process contributes an extension  $tt$  to the trace, which the following theorem demonstrates:

**Theorem 2.1 ( $\mathbf{R1-R2}$  trace contribution)**

$$\mathbf{R1}(\mathbf{R2}(P)) = (\exists tt \bullet P[\langle \rangle, tt / tr, tr'] \wedge tr' = tr \frown tt)$$

The theorem shows that  $\mathbf{R1-R2}$  processes ensure that there exists a trace extension  $tt$ , and  $tr'$  is the prior history appended with this extension.

Finally, healthiness condition  $\mathbf{R3}$  ensures that intermediate states are appropriately respected. If a prior process is in an intermediate state, denoted by *wait*, then the following process must behave as the reactive skip  $\mathbb{I}_{rea}$ . Otherwise, the process exhibits its own behaviour ( $P$ ). The reactive skip has the intuitive definition  $\mathbb{I}_{rea} = \mathbf{R}(\mathbf{true} \vdash \mathbb{I})$ , where  $\mathbb{I}$  is the relational identity. Thus  $\mathbb{I}_{rea}$  has a true precondition, and its postcondition simply identifies all variables.

Since UTP designs form a complete lattice, and the reactive healthiness condition  $\mathbf{R}$  is monotone, we can show that reactive designs also form a complete lattice. The bottom of the lattice is the process **Chaos**, defined in Table 3, which corresponds to the program that makes no guarantees about its behaviour other than that the trace monotonically



| Construct                            | Description  |
|--------------------------------------|--|
| <b>Skip</b>                          | terminates immediately with no activity  |
| <b>Stop</b>                          | deadlocks immediately with no activity; allows time passage  |
| <b>Wait</b> $e$                      | wait $e$ time units and then terminate   |
| <b>Wait</b> $[m, n]$                 | non-deterministically waits between $m$ and $n$ units  |
| $c!e \rightarrow P$                  | offer output of $e$ on channel $c$ , then behave as $P$  |
| $c.e \rightarrow P$                  | offer event on channel $c$ parametrised by $e$ , then behave as $P$  |
| $c?x \rightarrow P(x)$               | accept input for bound variable $x$ of on channel $c$  |
| $P \square Q$                        | external choice; choose between $P$ and $Q$ when one offers an event   |
| $P ; Q$                              | sequential composition of $P$ and $Q$  |
| $P \triangleleft b \triangleright Q$ | conditional; behave as $P$ if $b$ is true, otherwise behave as $Q$   |
| $\mu X \bullet P(X)$                 | recursive behaviour; calculates fixed point of $X = P(X)$  |
| <b>while</b> $b$ <b>do</b> $P$       | while loop; iterates $P$ while $b$ is true   |
| $x :=_{\text{rea}} e$                | instantaneously assign $e$ to variable $x$   |
| <b>var</b> $x : \tau \bullet P(x)$   | creates a local variable $x$ of type $\tau$ scoped to $P$  |
| $P \triangleright^e Q$               | timeout; behaves as $P$ initially, if $P$ does not perform an event or terminate before $e$ time units have elapsed then behave like $Q$ |
| $P \blacktriangleright m$            | deadline; $P$ must terminate before $m$ units have passed  |
| $P \parallel Q$                      | interleave the behaviour of $P$ and $Q$ (no communication)   |
| $P \llbracket A \rrbracket Q$        | parallel composition of $P$ and $Q$ ; synchronisation permitted on events in $A$   |
| $P \setminus A$                      | hide the events of $P$ specified in $A$ such that they become internal events  |

Table 4: *CML* process constructs

increases. The top of the lattice is **Miracle**, the process which violates its postcondition and is thus impossible to execute.

Since a reactive design's postcondition specifies both intermediate states (*wait'*), and final states (*wait'*) we introduce the following derived syntax [13]:  $\mathbf{R}(P \vdash Q \diamond R) \triangleq \mathbf{R}(P \vdash Q \triangleleft \text{wait}' \triangleright R)$ . Here,  $Q$  denotes the so-called “pericondition”, that is, the predicate that is satisfied by intermediate behaviours, and  $R$  is the postcondition satisfied by final states.

## 2.4 COMPASS Modelling Language

The semantic model we use for VDM-RT is *timed reactive designs*, a form a specification construct that allows the specification of reactive behaviour with discrete timing constraints. This UTP theory is used to give a semantics to the COMPASS Modelling Language [29, 11] (*CML*), which is a combination of VDM-SL and the process algebra CSP. We give VDM-RT a UTP semantics by using *CML*, whose constructs are directly defined using the UTP.

*CML* describes the behaviour of a system in terms of a collection of *processes* that communicate with each other over shared synchronous channels. These channels can optionally carry data, and are the only way that processes can communicate. Each process addition-



ally has a state structure which can be described using the data structures of VDM-SL, such as numerics, sets, sequences, records, and maps. This state is encapsulated, such that only constructs within the process' scope can query the state variable. A *CML* process is described using a sequence of paragraphs that we summarise below:

- the **types** paragraph, which describes the data structures the process uses;
- the **values** paragraph, which describes static defined values of the process;
- the **state** paragraph, which defines a collection of typed state variables;
- the **operations** paragraph, which defines a collection of (non-reactive) operations that act on the state variables. They can be specified implicitly using pre- and postconditions;
- the **actions** paragraph, which defines a collection of reactive actions that can both manipulate the state and interact with the environment using channels;
- the main action,  $\bullet P$ , which describes the top-level behaviour of a process using a composition of internal actions and operations.

The reactive behaviour of *CML* actions is described using constructs from CSP, a number of which are illustrated in Table 4. The various parallel operators, such as  $P \parallel Q$  and  $P[A] \parallel Q$ , permit the events from either  $P$  or  $Q$ , but the state is not shared – both act on an independent copy of the state variables. This ensures there can be no race conditions on state variables between processes. In addition to the usual constructs of CSP, *CML* also provides various timing operators, such as the delay statement, **Wait**  $n$ , where  $n$  is an expression of type  $\mathbb{N}$ , and the timeout statement  $P \stackrel{n}{\triangleright} Q$ . *CML*'s time domain is abstract, in that time events do not have a concrete interpretation as a specific length of time. For VDM-RT, the base time unit is the nanosecond and thus by default we will interpret *CML* time events to be the passage of 1 nanosecond.

Another key aspect of *CML* processes to be observed is that they satisfy the *maximal progress* assumption [43]. There are essentially three kinds of activity that a process can engage in: invisible internal activity, such as state manipulation, visible communication events synchronised over one or more processes, and the passage of time. The maximal progress assumption states that the internal activity of a process occurs infinitely faster than can be observed by the passage of time. Thus the presence of executable internal events in a process prevent time from passing. Moreover, hiding an event  $a$  in a process using  $P \setminus \{a\}$  converts  $a$  to an internal action, and thus ensures it too will happen before the passage of time. Only once all possible internal activity has been executed – the process has maximally progressed – can time pass. This assumption largely ensures that time is of a deterministic nature, and is important for VDM-RT as it ensures, for example, that pending state updates must take place before time can pass.

Details of the denotational and operational semantics of *CML* can be found in our previous COMPASS deliverables [11]. We here briefly summarise the semantic model, and describe the core constructs.

A timed reactive design is a reactive design whose event alphabet  $\Sigma^E$  contains a distinguished event called *tock* that records the discrete passage of time in the trace. Assuming a set  $\Sigma_U^E$  of user events, then we have  $\text{tock} : \mathbb{P} \Sigma_U^E \rightarrow \Sigma^E$  and  $\Sigma^E = \Sigma_U^E \cup \{\text{tock}.n \mid n \in \mathbb{P} \Sigma_U^E\}$ . The parameter of *tock* denotes the set of refusals at the end of each clock cycle. We

$$\begin{aligned}
\text{Stop} &\triangleq R(\text{true} \vdash \text{events}(\mathbf{tt}) = \langle \rangle \diamond \text{false}) \\
a \rightarrow \text{Skip} &\triangleq R \left( \text{true} \vdash \begin{array}{l} \text{events}(\mathbf{tt}) = \langle \rangle \wedge a \notin \text{refusals}(\mathbf{tt}) \\ \diamond \mathbf{tt} = \text{idleprefix}(\mathbf{tt}) \frown \langle a \rangle \wedge v' = v \wedge a \notin \text{refusals}(\mathbf{tt}) \end{array} \right) \\
\text{Wait } n &\triangleq R \left( \text{true} \vdash \begin{array}{l} \text{events}(\mathbf{tt}) = \langle \rangle \wedge \# \mathbf{tt} < n \\ \diamond \text{events}(\mathbf{tt}) = \langle \rangle \wedge \# \mathbf{tt} = n \wedge v' = v \end{array} \right)
\end{aligned}$$

Table 5: Example *CML* language semantics

abbreviate  $tr' - tr$  by  $\mathbf{tt}$ , an expression denoting the portion of the trace that the present process has contributed. The semantics use a number of functions on traces that we summarise below.

- $\text{events}(\mathbf{tt})$  – denotes the trace  $\mathbf{tt}$  excluding tock events, but retaining the order of other events;
- $\text{refusals}(\mathbf{tt})$  – denotes the set of all events that are refused by at least one tock event in the trace;
- $\text{idleprefix}(\mathbf{tt})$  – denotes the maximal prefix of  $\mathbf{tt}$  that contains only tock events, that is, the time events that occur in the initial idle period.

We can then define some example *CML* language constructs in Table 5. **Stop** is the process that immediately deadlocks: it has a true precondition, its pericondition ensures that no events can occur (though time can pass), and it cannot terminate since the postcondition is false. An event prefix,  $a \rightarrow \text{Skip}$ , has a true precondition, and its pericondition is that no CSP events occur in the trace, and yet that  $a$  is not refused anywhere. The use of  $\text{events}(\mathbf{tt})$  ensures that tock events are, however, possible in the trace, meaning that time can pass whilst we are waiting for an  $a$  event. The postcondition of event prefix states using three conjuncts, firstly, that the contributed trace consists of a period of time passing after which the event  $a$  is performed; secondly that all state variables remain unaltered; and finally that  $a$  is not refused anywhere during the idle period. A delay, **Wait**  $n$ , states in the pericondition that no communication events occur, and that the length of the trace is less than  $n$ . Since the trace can consist only of tock events, this means that less than  $n$  units of time have passed. The postcondition states, likewise, that no communication events can occur, the length of the trace equals  $n$ , so sufficient time has passed, and finally that the state variables remain constant.

This denotational semantics is then used to derive, and thus validate the soundness of, a corresponding operational semantics for *CML* processes [11]. This operational semantics was then used to implement the simulator in *Symphony*<sup>2</sup> [21, 22], an integrated development environment for *CML* based on the Overture platform.

*CML*'s sister language, *CircusTime* [68, 67], which shares a similar semantic model based on timed reactive designs and has many of the same operators, has been in used a similar context to give a semantics to Safety-Critical Java [17, 20]. We adapt many of the ideas from this work to create our semantics for VDM-RT, since many of the static analysis

<sup>2</sup><http://symphonytool.org/>

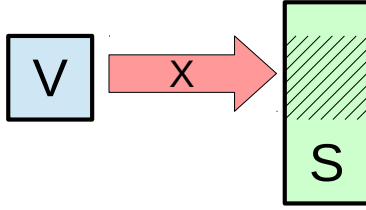


Figure 4: A typical lens

$$\begin{aligned}
 \text{get}(\text{puts } v) &= v & (\text{PutGet}) \\
 \text{put}(\text{puts } v') v &= \text{puts } v & (\text{PutPut}) \\
 \text{puts}(\text{gets } s) &= s & (\text{GetPut})
 \end{aligned}$$

Figure 5: Lens laws

techniques developed there are also of benefit to us.

## 2.5 Isabelle/UTP

*Isabelle/UTP* [38, 72, 36] is a mechanisation of the UTP semantic framework in the proof assistant Isabelle/HOL [59]. It allows us to define UTP theories within the alphabetised relational calculus, whilst taking advantages of Isabelle's type checker, and then mechanically prove associated theorems, such as algebraic laws. Such laws can then be applied to program verification tasks in Isabelle.

An alphabetised relation is essentially a set of possible observations that can be made of the model, such as the set of possible input and output mappings. Our model of alphabetised predicates, therefore, is  $\alpha \text{ upred} \triangleq \alpha \Rightarrow \text{bool}$ , where  $\alpha$  is a suitable type for modelling the alphabet, that corresponds to the state space. This means that we can easily implement the usual operators of boolean algebra and complete lattices by lifting the corresponding HOL notions on sets. Similarly, relational operators like composition  $P ; Q$  can also be obtained by lifting the corresponding HOL functions.

Variables in the state space  $\alpha$  are modelled abstractly using *lenses* [31, 30], which are perhaps best known in the functional programming world. A lens  $V \Longrightarrow S$ , for view type  $V$  and source type  $S$ , identifies  $V$  with a subregion of  $S$ . This is illustrated in Figure 4, where the hatched region denotes the portion of  $S$  that  $V$  corresponds to. Lenses can be used to abstract many types of data structure. For example, if  $S$  is a record type, then  $V$  might be a particular field, or if  $S$  is a function type, then  $V$  might be an element of the domain. A lens consists of two functions: *get* that extracts a view from a larger source, and *put* that puts back an updated view. Moreover the behaviour of lenses is constrained by a number of algebraic laws which are summarised in Figure 5. Since lenses are semantic rather than syntactic entities, we cannot compare them just using (in)equality, and thus we introduce further operators. Lens equivalence,  $X \approx Y$ , states that lenses  $X$  and  $Y$  view precisely the same region of the source, though these views may have different types. Lens independence,  $X \bowtie Y$ , states that the two lens views are independent: manipulating the source type using  $X$  has no effect on the region identified by  $Y$  and vice-versa. Such operators can be used as the basis for comparison of variables.

We have mechanised a theory of lenses in Isabelle during this project, including an algebra that allows us to variously compose lenses in the style of separation algebra [12]. For example, the sum lens  $X \oplus Y$  represents the lens that simultaneously views the regions characterised by both lenses  $X$  and  $Y$ . For more details please see our recent paper [38], which has been adapted into this deliverable and can be found in Appendix B.

|   |                                    |
|---|------------------------------------|
| $(\exists x \bullet P \wedge x = e) = P[e/x]$ | if $x \in \text{mwb-lens}, x \# e$ |
| $x := e ; P = P[e/x]$                         |                                    |
| $x := e ; x := f = x := f$                    | if $x \# f$                        |
| $x := e ; y := f = y := f ; x := e$           | if $x \bowtie y, x \# f, y \# e$   |

Table 6: Isabelle/UTP laws

We model variables as abstract views on program state spaces with a uniform semantic interface. A variable  $x : (\tau, \alpha) \text{ uvar}$  is a lens that views a particular subregion of type  $\tau$  in  $\alpha$ , which affords a very general state model. The main advantage lenses thus provide us with is an abstract notion of variables and state space in UTP predicates, such that a wide variety of different representations are possible. A commonly employed model for state spaces is that of Isabelle/HOL records, with fields to model variables, as these afford a large amount of built-in proof automation, which thus aids the program verification effort. Our use of lenses enables a more abstract characterisation, and ensures that any lens-based model for variables can be applied to proven laws of *Isabelle/UTP*, including for example partial function maps to encode a deeper predicate model with variable names as first-class citizens.

Mechanisation of the predicate calculus requires that we can specify meta-logical provisos, such as  $x \notin \text{fv}(P)$ , that is, that variable  $x$  is not free in  $P$ , and also variable substitution. Alphabetised predicates are principally semantic rather than syntactic entities, and so these notions cannot be specified using, for example, recursive functions that depend on an abstract syntax tree. Instead, we leverage our theory of lenses to define weaker semantic notions. For free variables, we introduce the concept of *unrestriction*, written  $x \# P$  for some variable (lens)  $x$ . A predicate  $P$  is unrestricted by variable  $x$  if the valuation of  $P$  does not depend on  $x$ . For example, the predicate  $y > 10$  is unrestricted by  $x$ , assuming  $x \bowtie y$ , since the value of  $x$  clearly has no bearing on the truth value of the predicate.

Substitution is also introduced semantically using the notation  $\sigma \dagger P$ , where  $\sigma : \alpha \rightarrow \alpha$  is a homogeneous substitution function on the state space. Application of a substitution to a predicate updates all possible observations using the function. The most basic substitution is the identity *id*, which maps all variables to their present value. We can also write  $\sigma(x \mapsto_s e)$ , which updates a substitution such that  $x$  takes the value of expression  $e$ . We also introduce the short-hand  $[x_1 \mapsto_s e_1, \dots, x_n \mapsto_s e_n] = \text{id}(x_1 \mapsto_s e_1, \dots, x_n \mapsto_s e_n)$ . A substitution  $P[e_1, \dots, e_n/x_1, \dots, x_n]$  of  $n$  expressions to corresponding variables is then expressed as  $[x_1 \mapsto_s e_1, \dots, x_n \mapsto_s e_n] \dagger P$ . This model allows us to obtain the usual laws of substitution, such as  $(P \wedge Q)[v/x] = (P[v/x] \wedge Q[v/x])$ .

With a complete relational calculus and associated meta-logical operators defined we are able to mechanise all the usual laws of predicate calculus, relation algebra, Kleene algebra, and other typical laws of programming, such as those in Table 2 and Table 6. The latter shows how we specify meta-logical provisos in *Isabelle/UTP*. For example the last law, commutativity of assignments, requires that  $x$  and  $y$  be different variables, specified using lens independence, that  $x$  is not free in  $f$  and that  $y$  is not free in  $e$ .

So far we have mechanised several hundreds of such algebraic laws, which provides the foundation for automated reasoning about programs and models. These can be seen by viewing our Isabelle/UTP git repository<sup>3</sup>. Moreover, we have also created a number of proof tactics for predicate calculus (**pred-tac**) and relational calculus (**rel-tac**), which also greatly aid the proof effort. When these are combined with Isabelle's built-in automated proof facilities [10] like the **auto** deduction tactic, the **sledgehammer** automated theorem prover integration, and the **nitpick** counterexample generator, *Isabelle/UTP* greatly aids the effort of mechanising UTP theories and eventually applying them to program verification.

Thus far we have used *Isabelle/UTP* to mechanise a number of UTP theories, including designs, reactive processes, the hybrid relational calculus [34, 16], and separation logic [69]. In this deliverable we will show how we can use *Isabelle/UTP* to prove laws of timed reactive designs and *CML*.

### 3 CML Extension

In this section we provide a number of extensions to the core *CML* language to cater for specific features of VDM-RT. We provide a new universe type for VDM-SL, and also the ability to specify timed expressions.

#### 3.1 Universe for VDM-SL

A prerequisite of our semantics is a representation of the universe of constructible VDM-SL values, which we call  $\mathbb{U}_{sl}$ . We will use this in our CML-based semantics of VDM-RT to represent parameters that are passed from object to object for method calls. This will allow us to have generic infrastructure processes for CPUs and busses which both simplifies translation, and also reduces the number of processes that need specific static analysis.

Since we are mechanising our semantics in Isabelle/HOL, the universe must be representable within the HOL logical system. A possible candidate universe is the von Neumann universe  $V_{\omega+\omega}$  [72]. The von Neumann universe  $V_i$  is inductively defined for some index  $i$  by repeated application of the power-set for ordinal indices  $\beta$ , and generalised union for limit ordinals  $\lambda$ .

$$V_0 \hat{=} \emptyset \quad V_{\beta+1} \hat{=} \mathbb{P}(V_\beta) \quad V_\lambda \hat{=} \bigcup_{\beta < \lambda} V_\beta$$

Each limit ordinal index corresponds to the union of all sets constructed up to that level. In HOL, every finite type is representable by some  $V_n$  (for  $n \in \mathbb{N}_{>0}$ ), and every infinite type by some  $V_{\omega+n}$ . For example,  $\mathbb{N}$  can be represented by  $V_\omega$  and  $\mathbb{R}$  by  $V_{\omega+1}$ . Then,  $V_{\omega+\omega}$ , which is the limit of these sets, contains all possible types that are constructible in HOL. It is, therefore, impossible to formalise  $V_{\omega+\omega}$  in HOL and thus this cannot be our universe. Nevertheless, since it is the universe of ordinary mathematics, we will use it to

<sup>3</sup>Please see <https://github.com/isabelle-utp/utp-main/>

characterise the base logic world  $\mathbb{W} \triangleq V_{\omega+\omega}$ , which could be either HOL or some other equipollent logic.

Fortunately, VDM-SL does not require the full generality of the von Neumann universe, as its type constructors are mainly of a finite nature. In particular, the power set operator is a finite power set operator. The constructible VDM-SL types  $\mathbb{T}_{\text{sl}}$  include the following types, for  $A, B, T_i \in \mathbb{T}_{\text{sl}}$ :

1. booleans ( $\underline{\mathbb{B}} \in \mathbb{T}_{\text{sl}}$ );
2. natural numbers ( $\underline{\mathbb{N}} \in \mathbb{T}_{\text{sl}}$ );
3. integers ( $\underline{\mathbb{Z}} \in \mathbb{T}_{\text{sl}}$ );
4. rational numbers ( $\underline{\mathbb{Q}} \in \mathbb{T}_{\text{sl}}$ );
5. real numbers ( $\underline{\mathbb{R}} \in \mathbb{T}_{\text{sl}}$ );
6. products ( $A \times B \in \mathbb{T}_{\text{sl}}$ );
7. sequences ( $\text{seq of } A \in \mathbb{T}_{\text{sl}}$ );
8. finite sets ( $\text{set of } A \in \mathbb{T}_{\text{sl}}$ );
9. finite maps ( $\text{map } A \text{ to } B \in \mathbb{T}_{\text{sl}}$ );
10. records ( $R :: f_1 : T_1 \cdots f_n : T_n \in \mathbb{T}_{\text{sl}}$ );
11. union types ( $A \mid B \in \mathbb{T}_{\text{sl}}$ ).

We distinguish a type code name  $\underline{A}$  from its characteristic set  $A$ . The first observation is that  $\mathbb{B}, \mathbb{N}, \mathbb{Z}$ , and  $\mathbb{Q}$ , being countable sets, clearly all have a cardinality no greater than that of  $\mathfrak{c}$ , the cardinality of the continuum, which corresponds to  $|\mathbb{R}|$ . Moreover, it is well known that taking the product, union, and finite power set of a set does not increase its cardinality. Sequences and finite maps can be also effectively encoded as finite sets of pairs, and records are simply products. The one remaining type is  $\mathbb{R}$ , which, by Cantor's famous diagonalisation proof, we know can be encoded as infinite sequences of bits, which in turn is equivalent to  $\mathbb{P}\mathbb{N}$ . Thus all the types of VDM-SL can be injected into  $\mathbb{P}\mathbb{N}$ , or equivalently the von Neumann set  $V_{\omega+1}$ . We have formalised in Isabelle/HOL and proved that all the given types are injectable, which then provides the mathematical basis for our semantics.

We define a number of functions that allow manipulation of values in the universe. We define a function  $\llbracket - \rrbracket^{\text{sl}} : \mathbb{T}_{\text{sl}} \rightarrow \mathbb{P}\mathbb{W}$ , which gives the carrier set for a VDM-SL type. Then we define a function  $\text{inj}_{\tau} : \llbracket \tau \rrbracket^{\text{sl}} \rightarrow \mathbb{U}_{\text{sl}}$  that injects a value of type  $\tau$  into the VDM-SL universe. We also define  $\text{cast}_{\tau} : \mathbb{U}_{\text{sl}} \rightarrow \llbracket \tau \rrbracket^{\text{sl}}$  that casts a universe value of the type  $\tau$ . Together these two functions satisfy the following axiom:

$$x \in \llbracket \tau \rrbracket^{\text{sl}} \implies \text{cast}_{\tau}(\text{inj}_{\tau}(x)) = x$$

That is, if we inject a value of type  $\tau$  and then cast it back to  $\tau$ , the same value is returned. These functions effectively allow us to perform type erasure and reinstatement on a value, which is necessary to construct call messages in our *CML* semantics.



### 3.2 Expressions

A VDM program variable  $x$  pairs a name ( $x_n \in \text{VarId}$ ) with a type ( $x_\tau \in \mathbb{T}_{\text{sl}}$ ). The state of a program  $\sigma$  is a finite partial function from variable names to values  $\Sigma \triangleq \{f : \text{VarId} \mapsto \mathbb{U}_{\text{sl}} \mid \forall x \bullet f(x) : x_\tau\}$ . A VDM-SL expression of type  $\tau$  is then modelled as a partial function from state bindings to values, that is  $\mathcal{E}_{\text{sl}}^\tau \triangleq \{f : \Sigma \mapsto \mathbb{U}_{\text{sl}} \mid \forall \sigma \bullet f(\sigma) \in \llbracket \tau \rrbracket^{\text{sl}}\}$ . This semantic characterisation of expressions means we can give semantics to expression operators by lifting corresponding functions of the underlying logic, such as Isabelle/HOL. We thus define combinators for expression liftings.

$$\begin{aligned} \perp_{\text{sl}} &= \emptyset \\ \text{lit}(v) &= \{\sigma \mapsto v \mid \sigma \in \Sigma\} \\ \text{uop}(f, e) &= \{\sigma \mapsto f(e(\sigma)) \mid \sigma \in \text{dom}(e) \wedge e(\sigma) \in \text{dom}(f)\} \\ \text{bop}(f, e_1, e_2) &= \left\{ \sigma \mapsto f(e_1(\sigma), e_2(\sigma)) \mid \begin{array}{l} \sigma \in \text{dom}(e_1) \cap \text{dom}(e_2) \wedge \\ (e_1(\sigma), e_2(\sigma)) \in \text{dom}(f) \end{array} \right\} \\ \text{trop}(f, e_1, e_2, e_3) &= \left\{ \sigma \mapsto f \left( \begin{array}{c} e_1(\sigma), \\ e_2(\sigma), \\ e_3(\sigma) \end{array} \right) \mid \begin{array}{l} \sigma \in \text{dom}(e_1) \cap \text{dom}(e_2) \cap \text{dom}(e_3) \wedge \\ (e_1(\sigma), e_2(\sigma), e_3(\sigma)) \in \text{dom}(f) \end{array} \right\} \end{aligned}$$

Expression  $\perp_{\text{sl}}$  is the undefined expression. Expression  $\text{lit}(v)$  takes a value in the underlying logic,  $v \in \mathbb{W}$ , and constructs a literal expression; it is constant for every state. Combinators  $\text{uop}(f)$ ,  $\text{bop}(f)$ , and  $\text{trop}(f)$  construct unary, binary, and ternary expressions for base logic functions of type  $\mathbb{W}^n \mapsto \mathbb{W}$ , where  $n \in \{1..3\}$ . Using these combinators, we can lift suitable functions from the base logic like so:

$$\begin{aligned} \llbracket e + f \rrbracket &= \text{bop}(+, e, f) \\ \llbracket e - f \rrbracket &= \text{bop}(-, e, f) \\ \llbracket \text{inj}_\tau(e) \rrbracket &= \text{uop}(\text{inj}_\tau, e) \\ \llbracket \text{cast}_\tau(e) \rrbracket &= \text{uop}(\text{inj}_\tau, e) \end{aligned}$$

Moreover, we can also characterise the expression definedness construct, and use this to define VDM-SL assignment:

$$\begin{aligned} \mathcal{D}(e) &\triangleq \{\sigma \mapsto (\sigma \in \text{ran}(e)) \mid \sigma \in \Sigma\} \\ x :=_{\text{sl}} e &\triangleq \mathbf{R}(\mathcal{D}(e) \vdash x := e) \end{aligned}$$

The definedness condition for an expression is the set of states that are in the domain of the expression's characteristic partial function. For example, division is defined only when the denominator is non-zero, and thus we would have that  $\mathcal{D}(x/y) = (y \neq 0)$ . This then allows us to define assignment as a reactive design, where the precondition is that the assigned expression is defined. This construct will abort for states where the expression is undefined.

A VDM-RT expression pairs a VDM-SL expression of type  $\mathbb{N}$ , representing the execution time, with a VDM-SL expression of type  $\tau$ , i.e.  $\mathcal{E}_{\text{rt}}^\tau \triangleq \mathcal{E}_{\text{sl}}^{\mathbb{N}} \times \mathcal{E}_{\text{sl}}^\tau$ . Then  $e_t$  denotes the time expression, and  $e_v$  the underlying VDM-SL expression.

## 4 VDM-RT Semantics in CML

In section we describe the semantic mapping from VDM-RT into *CML*.

### 4.1 Overview

We focus our semantics of VDM-RT on the subset of the language used by the case studies and pilot studies of the INTO-CPS project. These models follow a pattern that consists of a static object topology with the following components:

- a **World** class, which describes the overall model;
- a **System** class, which describes the system’s configuration and topology, in terms of its controller, sensors, and actuators, together with their construction and deployment on CPUs;
- several user-defined **Controller** classes, corresponding to each of the discrete controllers;
- several user-defined **Sensor** and **Actuator** classes, which are each owned by a controller;
- several CPU objects, onto which the controllers are deployed;
- several bus objects, that connect together CPUS and provide the infrastructure for remote operation calls between objects.

The structure of a typical **System** class topology is shown in Table 7. Our water tank example in Figure 3 shows a typical controller linked to sensors and actuators. We assume there is no sharing of sensors and actuators between controllers, and there is no dynamic creation of additional objects – the topology is fixed by the **System** class. These restrictions will also provide more scope for formal analysis of the models using model checking and theorem proving. Nevertheless, several extensions are possible to consider dynamic topologies, that we discuss in Section 7. For example, our previous deliverable [39] gives semantics to object-oriented VDM-RT data structures with multiple inheritance. Such data structures are “passive” because they do not possess threads. This kind of object creation can be readily supported, since passive objects are simply records with associated operations.

The general approach of our semantics is to describe each of the above objects using *CML* processes, with operation invocation described using CSP events. In a sense, therefore, all of the objects are assumed to be “active”, meaning they exhibit autonomous reactive behaviour, which can, for example, take the form of periodic threads and asynchronous operation calls<sup>4</sup>. Unlike [47] and [54] we do not invoke the copy rule to execute local synchronous calls, to avoid the complexities associated with multiple object state contexts. Specifically, each operation must run within the context of its object process, which is where the state is encapsulated. Moreover our active classes do not support

<sup>4</sup>It is also possible to identify the class of “passive” objects which have no reactive behaviour; however for now VDM-RT makes no obvious distinction of these cases and so neither does our semantics.



```

system System

instance variables
  public static ctrl1 : [Ctrl1] := nil;
  ...
  public static ctrln : [Ctrln] := nil;

  public static cpu1 : CPU := new CPU(sp1, s1);
  ...
  public static cpun : CPU := new CPU(spn, sn);

operations
  public System : () ==> System
  (
    let act1-1 = new Actuator1-1(...),
      ...
      act1-k1 = new Actuator1-k1(...),
      sen1-l1 = new Sensor1-1(...),
      ...
      sen1-ln = new Sensor1-ln(...)
    in
      ctrl1 := new Ctrl1(act1-1,...,act1-k1,sen1-1,...,sen1-l1);

    cpu1.deploy(ctrl1);

    let actn-1 = new Actuatern-1(...),
      ...
      actn-kn = new Actuatern-kn(...),
      senn-1 = new Sensorn-1(...),
      ...
      senn-ln = new Sensorn-ln(...)
    in
      ctrln := new Ctrl1(actn-1,...,actn-kn,senn-1,...,senn-ln);

    cpun.deploy(ctrln);
  );

end System

```

Table 7: Typical structure of the System class

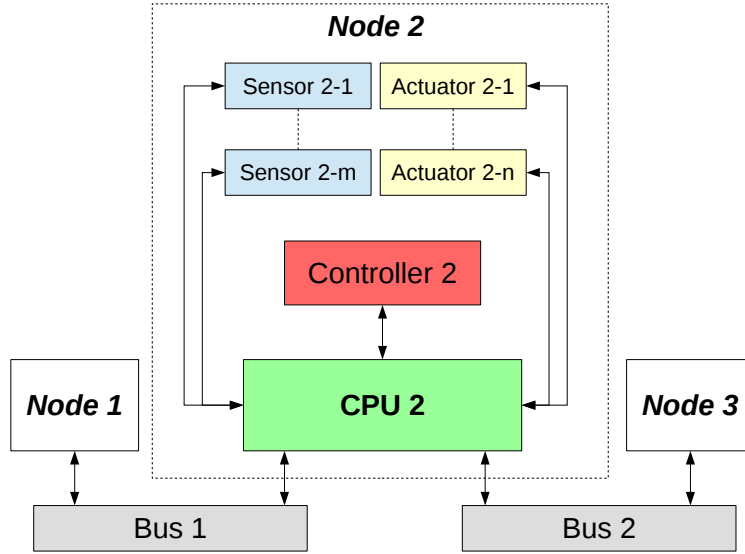


Figure 6: Overview of the VDM-RT *CML* process architecture

more advanced object oriented features like inheritance to avoid the need for behavioural subtyping [55]. Such object oriented features are supported for passive classes [39].

The process structure is exemplified in Figure 6. Each box denotes a *CML* process, and the arrows denote communication channels between them. Each connected VDM-RT compute node, consisting of a CPU and its various objects, becomes a composite *CML* process, such as *Node 2*. The CPU process schedules execution of the node’s threads and handles operation calls between objects. All communication is passed through the CPU – other communications between node objects are not permitted. Each controller object will usually encapsulate a periodic thread that gives the overall behaviour of the node in terms of operation calls to the sensors, actuators, and potentially controllers running on different CPUs. Each sensor and actuator will mainly consist of a collection of operations, together with threads to handle calls to these operations. Inter-CPU communication is made possible through the bus processes.

The overall lifecycle of a given thread is shown in Figure 7. The CPU creates a new unique identifier for the given thread, to enable addressing of operation call return data. The new thread first obtains permission to execute, since only one thread can be active on the CPU at any one time, and so any other active thread must first yield. Once permission is granted, the thread retrieves the central object state and places it into its own local cache. The body of the thread is then executed, which may expend time, and could require further execute-yield cycles, for example when synchronous operation calls are made. Once the body is complete, the thread synchronises its cache back with the central state, yields control back to the CPU and finally terminates.

The denotational semantics is specified via a function that maps a VDM-RT model written using our pattern to a *CML* program. In defining this function, we use an environment that captures the object/CPU dependency tree defined in the VDM-RT model. We do not formalise here the construction of this environment, but broadly, the tree should specify which controller runs on which CPU, and which sensors and actuators these controllers own. The *CML* process defined by the semantics uses standard infrastructure

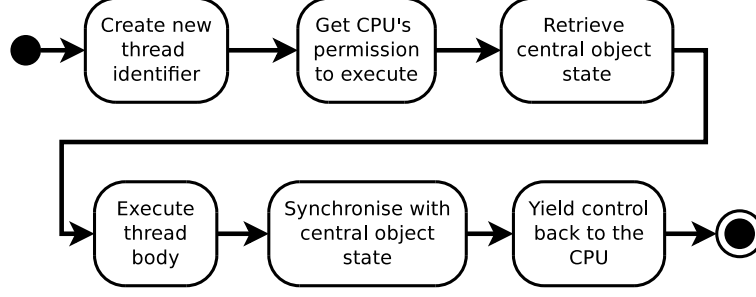


Figure 7: Abstract thread lifecycle

processes to represent CPUs and buses, and bespoke processes generated for each controller, sensor, and actuator object following the interface. Composing these processes using an appropriate topology of abstractions and channel links then gives the overall system definition.

We specify the semantic function as a collection of infrastructure and template *CML* processes. The template processes contain meta-tags of the form  $\langle k \rangle$  that should be replaced with appropriate texts at the point of generation. For example, each VDM-RT object has a name  $\langle Obj_k \rangle$ , and this name will form part of the name for the corresponding object *CML* process. We define the following three semantic functions for VDM-RT classes:

- $\llbracket - \rrbracket^{\text{cls}}$  – maps a VDM-RT class to a *CML* process;
- $\llbracket - \rrbracket^{\text{thr}}$  – maps a VDM-RT thread to a *CML* action;
- $\llbracket - \rrbracket_d^{\text{stm}}$  – maps a VDM-RT statement block to a *CML* action fragment, using variable  $d$  to encode the duration context.

We will now proceed to describe the semantic model in terms of the *CML* process artefacts.

## 4.2 Types

We introduce a number of *CML* types for the VDM-RT processes, which are part of the standard infrastructure and thus need not be generated. These include:

- $\mathbb{U}_{\text{sl}}$  – the VDM-SL data universe that we defined in Section 3.1;
- $SCall = SCall :: sobj : ObjId sthr : ThrId parm : \mathbb{U}_{\text{sl}}$  – a record representing a synchronous call, with return information including the source object (*sobj*), thread (*sthr*), and call parameters (*parm*);
- $ACall = ACall :: parm : \mathbb{U}_{\text{sl}}$  – a record represents an asynchronous call, consisting of only parameters (*parm*);
- $Call = SCall \mid ACall$  – a call of either kind;
- $BusCallMsg :: tobj : ObjId topr : OpId call : Call$  – a bus message for a call, with a target object and operation;

- $\mathbf{BusRetMsg} :: robj: ObjId \ rthr: ThrId \ rval: \mathbb{U}_{sl}$  – a bus return message, with a return object, thread, and value.

Additionally, several types must be generated to represent model specific artefacts. These include

- $\mathbf{CPUId}$  – a finite type of all extant CPU names, encoded as an enumeration;
- $\mathbf{ObjId}$  – a finite type of all extant object names, encoded as an enumeration;
- $\mathbf{OpId}$  – a finite type of all extant operation names, encoded as an enumeration;
- $\mathbf{ThrId}$  – the type of allocated thread identifiers, which in our semantics is simply a natural number  $\mathbb{N}$ , but could be given a more sophisticated structure .

### 4.3 Channels

We also introduce a number of infrastructure *CML* channels in our models which are described below.

- $\mathbf{yield} : ()$  – offered by a thread that is willing to yield to other threads;
- $\mathbf{exec} : ()$  – used to instruct a thread to begin or resume execution;
- $\mathbf{newThr} : ThrId$  – used to request and allocate a thread identifier for a new thread;
- $\mathbf{call} : ObjId \times OpId \times (SCall \mid ACall)$  – used by threads to make operation calls;
- $\mathbf{lCall} : ObjId \times OpId \times (SCall \mid ACall)$  – used to delegate a call to a local operation;
- $\mathbf{ret} : ObjId \times ThrId \times \mathbb{U}_{sl}$  – used by an operation thread to communicate its return value to the CPU;
- $\mathbf{lRet} : ObjId \times ThrId \times \mathbb{U}_{sl}$  – used by a CPU to return a value to a local object;
- $\mathbf{rCall} : CPUId \times \mathbf{BusCallMsg}$  – used to send a remote call via a bus;
- $\mathbf{rRet} : CPUId \times \mathbf{BusRetMsg}$  – used to send return messages via a bus;
- $\mathbf{cCall} : CPUId \times \mathbf{BusCallMsg}$  – used by the bus to forward call messages to a CPU;
- $\mathbf{cRet} : CPUId \times \mathbf{BusRetMsg}$  – used by the bus to forward return messages to a CPU.

Additionally, the following channels must be generated on a per-model basis:

- $\mathbf{getState-}\langle Class \rangle : State-}\langle Class \rangle$  – used by threads of class *Class* to get the present central state;
- $\mathbf{syncState-}\langle Class \rangle : State-}\langle Class \rangle \rightarrow State-}\langle Obj \rangle$  – used by threads to synchronise their internal state with the central state.

### 4.4 Classes and Objects

Class processes encapsulate actions corresponding to the operations and threads of the corresponding VDM-RT class. Object processes will effectively create copies of their corresponding class process, but with an allocated object name. In our semantics we do

not directly consider static instance variables and functions, which can be represented by similar constructs in *CML*. We assume that each class can be allocated a unique name represented by meta-tag  $\langle \text{Class}_k \rangle$ . Additionally, each object process of a particular class maintains its own central state. The various threads running within the class's context copy and synchronise with this state using the channels  $\text{getState-}\langle \text{Class}_k \rangle$  and  $\text{syncState-}\langle \text{Class}_k \rangle$ .

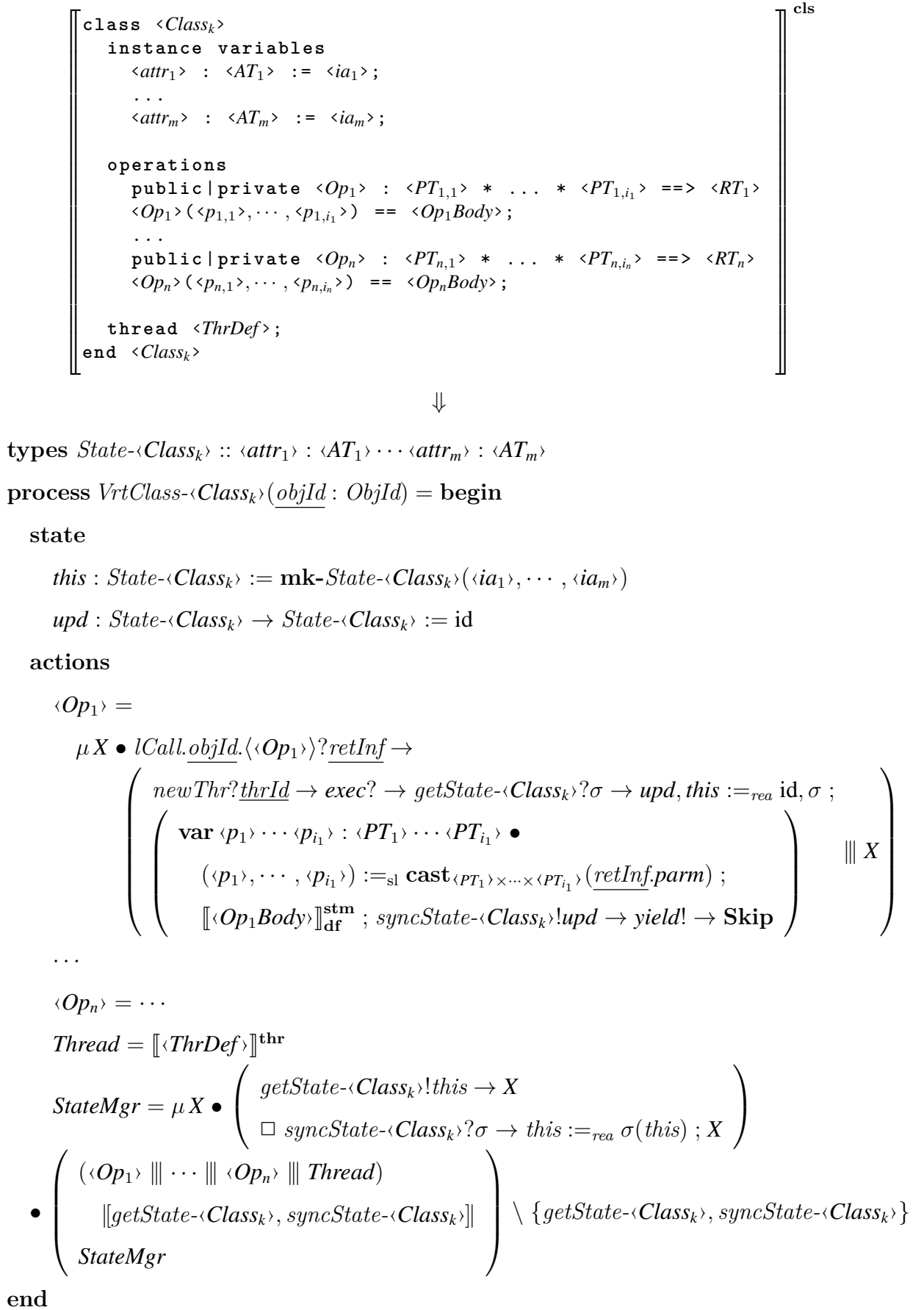
A semantic mapping for a typical class process is shown in Figure 8. The generated class process is parameteric over an object name  $\text{objId}$  that will be given when the class is instantiated. The state of the class, that is, the collection  $m$  of instance variables, is represented by the record type  $\text{State-}\langle \text{Class}_k \rangle$  where  $\langle \text{attr}_i \rangle$  is the instance variable name, and  $\langle \text{AT}_m \rangle$  its type. Thus the class process,  $\text{VrtClass-}\langle \text{Class}_k \rangle$  has a state variable  $\text{this}$  of the state type, with initial values taken from the instance variable initial values  $\langle \text{ia}_i \rangle$ . The state variable  $\text{upd}$  records the updates that have been made to the state as a total function on an existing state. It can then be applied to the central state when a thread needs to synchronise.

We assume there are  $n$  operations, named  $\langle \text{Op}_1 \rangle \dots \langle \text{Op}_n \rangle$ , with operation bodies  $\langle \text{Op}_1 \text{Body} \rangle \dots \langle \text{Op}_n \text{Body} \rangle$ , respectively, which are encoded by respective actions that encode their thread behaviour. Additionally a class can optionally have a thread action,  $\text{Thread}$ , whose semantics will be described in the next section. The  $\text{StateMgr}$  action manages the central state for the class stored in state variable  $\text{this}$ . It can send the current central state to another action using the  $\text{getState-}\langle \text{Class}_k \rangle$  channel, and can synchronise the state with an action using the  $\text{syncState-}\langle \text{Class}_k \rangle$ . In the latter case the state manager updates the value of  $\text{this}$  by applying the update function to it. The main action of the process interleaves the operation and thread actions, and composes them with the state manager. The state manipulation channels are then hidden to ensure that state updates occur urgently: the clock cannot advance whilst a pending state update remains.

We now describe the operation actions in more detail. Each operation action first waits for a call request on the local call channel,  $\text{lCall}$ , parametrised with the respective instantiated object and operation names, and carrying argument data in  $e$ . A replication then occurs, whereby the behaviour of the operation thread is parallel composed with a copy of the operation action by a recursive reference, which means that another local call can be made and additional threads created if necessary. Thus an unbounded number of threads can be created for each operation.

In the operation thread, a new thread name is created through a request on channel  $\text{newThr}$  which is placed into variable  $\text{thrId}$ . The thread then waits for permission to execute from the CPU on the  $\text{exec}$  channel. Once this is granted, the thread gets the current central state, sets the update to the empty update id, and the value of  $\text{this}$  to the retrieved state. A collection of local variables are created to represent the operation parameters. These parameters are populated by casting the input parameter expression to the appropriate type. If one of the parameters is a class type, then this should be mapped to the type  $\text{ObjId}$ , so that the parameter corresponds to the name of an object.

After this the body of the operation is executed, the semantics of which is generated using the semantic function  $\llbracket - \rrbracket_{\text{df}}^{\text{stm}}$ , where the **df** parameter denotes that the statement block is not in the context of a duration statement. After the execution of the body, the thread synchronises its state, yields to the CPU, and finally terminates with **Skip**.



$$\begin{aligned}
\llbracket \langle \text{ThrBody} \rangle \rrbracket^{\text{thr}} &= \left( \begin{array}{l} \text{newThr?} \underline{\text{thrId}} \rightarrow \text{exec?} \rightarrow \text{getState-}\langle \text{Class}_k \rangle ? \sigma \rightarrow \\ \text{upd, this} :=_{\text{rea}} \text{id}, \sigma ; \llbracket \langle \text{ThrBody} \rangle \rrbracket_{\text{df}}^{\text{stm}} ; \\ \text{syncState-}\langle \text{Class}_k \rangle ! \text{upd} \rightarrow \text{yield!} \rightarrow \text{Skip} \end{array} \right) \\
\llbracket \text{periodic}(\langle \text{period} \rangle, \langle \text{jitter} \rangle, \langle \text{delay} \rangle, \langle \text{offset} \rangle)(\langle \text{ThrBody} \rangle) \rrbracket^{\text{thr}} &= \\
&\quad \text{Wait} [\langle \text{offset} \rangle, \langle \text{offset} \rangle + \langle \text{jitter} \rangle] ; \\
&\quad \mu X \bullet \left( \begin{array}{l} \text{newThr?} \underline{\text{thrId}} \rightarrow \text{exec?} \rightarrow \text{getState-}\langle \text{Class}_k \rangle ? \sigma \rightarrow \text{upd, this} :=_{\text{rea}} \text{id}, \sigma ; \\ \left( \begin{array}{l} ((\llbracket \langle \text{PTBody} \rangle \rrbracket_{\text{df}}^{\text{stm}} ; \text{syncState-}\langle \text{Class}_k \rangle ! \text{upd} \rightarrow \text{Skip}) \blacktriangleright \langle \text{period} \rangle) \\ \parallel \text{Wait} [\min(\langle \text{delay} \rangle, \langle \text{period} \rangle - \langle \text{jitter} \rangle), \langle \text{period} \rangle + \langle \text{jitter} \rangle] \end{array} \right) ; \\ \text{yield!} \rightarrow X \end{array} \right) \\
\llbracket \text{sporadic}(\langle \text{delay} \rangle, \langle \text{bound} \rangle, \langle \text{offset} \rangle)(\langle \text{ThrBody} \rangle) \rrbracket^{\text{thr}} &= \\
&\quad \text{Wait} [\langle \text{offset} \rangle, \langle \text{offset} \rangle + \langle \text{bound} \rangle] ; \\
&\quad \mu X \bullet \left( \begin{array}{l} \text{newThr?} \underline{\text{thrId}} \rightarrow \text{exec?} \rightarrow \text{getState-}\langle \text{Class}_k \rangle ? \sigma \rightarrow \text{upd, this} :=_{\text{rea}} \text{id}, \sigma ; \\ \left( \begin{array}{l} ((\llbracket \langle \text{STBody} \rangle \rrbracket_{\text{df}}^{\text{stm}} \rightarrow \text{syncState-}\langle \text{Class}_k \rangle ! \text{upd} \rightarrow \text{Skip}) \blacktriangleright \langle \text{bound} \rangle) \\ \parallel \text{Wait} [\langle \text{delay} \rangle, \langle \text{bound} \rangle] \end{array} \right) ; \\ \text{yield!} \rightarrow X \end{array} \right)
\end{aligned}$$

Figure 9: Thread semantics

## 4.5 Class Threads

Class threads can either be procedural, periodic, or sporadic in nature. The semantics of these different kinds of threads are shown in Figure 9. A procedural thread is the simplest kind of thread: it simply follows the usual thread protocol and begins executing as soon as the process is instantiated.

A periodic thread action behaves similarly, but its execution is driven by timing constraints. These constraints are specified in terms of the following semantic parameters:

- $\langle \text{offset} \rangle$  – gives the lower bound on the thread's first execution time;
- $\langle \text{period} \rangle$  – gives the usual period between the thread's execution;
- $\langle \text{jitter} \rangle$  – gives the amount of time variance that is allowed around a single event;
- $\langle \text{delay} \rangle$  – gives the minimum delay between two periodic events.

The action first waits between  $\langle \text{offset} \rangle$  and  $\langle \text{offset} \rangle + \langle \text{jitter} \rangle$  nanoseconds, which gives the start time of the first event, and then enters its iterative behaviour. The thread first waits

```

process VrtCpu(cpu : CPUId, depl :  $\mathbb{F}$  ObjId) = begin

  state

    lastThr : ThrId := 0

  actions

    ThrSched =  $\mu X \bullet exec! \rightarrow yield? \rightarrow X$ 

    ThrMgr =  $\mu X \bullet newThr!(lastThr + 1) \rightarrow lastThr :=_{rea} lastThr + 1 ; X$ 

    OpMgr =  $\dots$  (see Table 8)

    • (ThrSched ||| ThrMgr ||| OpMgr)

end

```

Figure 10: VDM-RT CPU infrastructure process

for permission from the CPU to begin executing. Once this is granted, it synchronises the state and executes the periodic thread body, which has a deadline of  $\langle period \rangle$ . Upon completion of the body, the thread synchronises the state, and then yields. In parallel with the body execution, the clock waits for between  $\min(\langle delay \rangle, \langle period \rangle - \langle jitter \rangle) -$  the least of the minimum delay and the period minus the jitter – and  $\langle period \rangle + \langle jitter \rangle$  nanoseconds. This ensures the correct timing constraints for the next execution of the periodic thread. Once the thread is completed and sufficient time has passed for the next execution, the action recurses.

A sporadic thread simply specifies that an event can occur at any non-deterministically selected instant between  $\langle delay \rangle$  and  $\langle bound \rangle$  time units, along with a given starting offset. It has a similar behaviour, though the inter-execution wait period is between  $\langle delay \rangle$  and  $\langle bound \rangle$ .

## 4.6 CPUs

A CPU orchestrates the execution of the objects which are deployed on it. It provides facilities for scheduling threads, and delegating operation calls to appropriate object processes. The CPU infrastructure process can be seen in Figure 10. The *VrtCpu* process is parametrised over an identifier for the CPU, and finite set *depl* that contains a record of all the deployed objects. The single state variable *lastThr* holds the previously allocated thread identifier; it is incremented each time a new thread is created. This is a simplistic semantics for thread identified allocation, and could be replaced with a more sophisticated algorithm that similarly ensures uniqueness.

The CPU process has three actions that define the behaviour of the thread scheduler (*ThrSched*), thread identifier manager (*ThrMgr*) and the operation call manager (*OpMgr*). The thread scheduler *ThrSched* simply ensures that only one thread is active at a time by offering *exec* events, and then awaiting a *yield* event which must come from the same thread. The thread identifier manager offers a new identifier over *newThr* by incrementing *lastThr*; following this, it also increments the state variable. The behaviour



$$\begin{aligned}
OpMgr = & \mu X \bullet call?(o, f, c) \rightarrow \\
& \left( \begin{array}{l} lCall!(o, f, c) \rightarrow X \\ \triangleleft o \in depl \triangleright \\ rCall.cpu!mk-BusCallMsg(o, f, c) \rightarrow X \end{array} \right) \\
& \square ret?(p, t, v) \rightarrow \\
& \left( \begin{array}{l} lRet!(p, t, v) \rightarrow X \\ \triangleleft p \in depl \triangleright \\ rRet.cpu!mk-BusRetMsg(p, t, v) \rightarrow X \end{array} \right) \\
& \square cCall.cpu?mk-BusCallMsg(o, f, c) \rightarrow lCall!(o, f, c) \rightarrow X \\
& \square cRet.cpu?mk-BusRetMsg(p, t, v) \rightarrow lRet!(p, t, v) \rightarrow X
\end{aligned}$$

Table 8: Operation call and return manager

of the operation manager is rather more involved, and thus it is expanded in Table 8, which we describe shortly.

The CPU process main action interleaves the behaviour of *ThrSched*, *ThrMgr*, and *OpMgr*. Thus, composing a suitably instantiated CPU process with a collection of deployed objects allows the CPU to orchestrate the execution of the objects. Such an instantiation and composition is illustrated in Section 4.9.

The operation manager is shown in Table 8. It consists of a recursive external choice over various possibilities for calls to and returns from object processes and busses. The first two cases deal with local operation call and return requests. The first possibility is the receipt of a call request from a local deployed object on channel *call*, which provides an object identifier *o*, operation identifier *f*, and call information *c*. If the object to be called is deployed locally ( $o \in depl$ ) then the call is simply passed on to the corresponding operation using channel *lCall*. If the object is not deployed then it is forwarded to an appropriate bus using the *rCall* channel, parametrised by the current CPU identifier and a new bus call message. Secondly, if a return is received from an object process on channel *ret*, with return object *p*, thread *t*, and value *v*, then if *p* is local, the return value *v* is sent to it. If the return object is not locally deployed then a remote return message is sent to a bus via *rRet*.

The final two cases deal with call and return requests coming remotely from busses. If a call message is received from a bus via *cCall* then it is treated just like a local call. Similarly, if a return message is received from a bus, then a local return is communicated.

## 4.7 Busses

A bus manages passing call and return messages between different CPUs, and can potentially introduce delays into their transmission. The bus process is defined in Table 9. In our semantics, bus processes connect two CPUs and are one-way. In order to produce a two-way CPU, two bus processes must be composed, as illustrated by *VrtBus2*. Essentially a bus is a FIFO queue for call and return messages with delays on when the

**process**  $VrtBus(cpu1 : CPUId, cpu2 : CPUId, delay : \mathbb{N}, sdepl : \mathbb{F} ObjId, tdepl : \mathbb{F} ObjId) =$   
**begin**  
    **state**  $mq : seq((BusCallMsg \mid BusRetMsg) \times \mathbb{N}) := \langle \rangle$   
    **actions**  
         $BusLane = \mu X \bullet$ 

$$\left( \begin{array}{l} \left( \begin{array}{l} (rCall.cpu1?m : (m.tobj \in tdepl) \rightarrow \\ \quad mq :=_{rea} mq \hat{\wedge} \langle (m, delay) \rangle ; X) \square \\ (rRet.cpu2?m : (m.robj \in sdepl) \rightarrow \\ \quad mq :=_{rea} mq \hat{\wedge} \langle (m, delay) \rangle ; X) \end{array} \right) \\ \bigtriangledown^1 \left( \begin{array}{l} \text{while } (\#mq > 0 \wedge snd(hd(mq)) = 0) \\ \quad \left( \begin{array}{l} (cRet.cpu1!(fst(hd(mq))) \rightarrow \text{Skip} \\ \quad \triangleleft \text{is-BusRetMsg}(fst(hd(mq))) \triangleright \\ \quad cCall.cpu2!(fst(hd(mq))) \rightarrow \text{Skip} \end{array} \right) ; \\ \quad mq :=_{rea} tl(mq) \end{array} \right) \\ \quad mq := map(\lambda(m, d).(m, d - 1))mq ; X \end{array} \right) \end{array} \right)$$
  
         $\bullet BusLane$   
**end**  
  
**process**  $VrtBus2(cpu1 : CPUId, cpu2 : CPUId, delay : \mathbb{N}, sdepl : \mathbb{F} ObjId, tdepl : \mathbb{F} ObjId) =$   
 $VrtBus(cpu1, cpu2, delay, sdepl, tdepl) \parallel VrtBus(cpu2, cpu1, delay, tdepl, sdepl)$

Table 9: VDM-RT one- and two-way bus processes

$$S ::= \text{skip} \mid \text{this}.x :=_{\text{rt}} e \mid x :=_{\text{rt}} e \mid \text{scall}(x, o, f, e) \mid \text{acall}(o, f, e) \mid \text{duration}(e)(S) \\ \mid S ; S \mid \text{if } e \text{ then } S \text{ else } S \mid \text{while } e \text{ do } S \mid \text{return } e$$

Table 10: VDM-RT Statements

messages are forwarded. The same queue can be shared for both types of messages as the queue effectively allocates a timestamp to each message.

The process is parametrised over identifiers for the connected CPUs ( $cpu1$ ,  $cpu2$ ), a delay parameter for message transmission ( $delay$ ), and two sets of object identifiers, one for the source CPU ( $sdepl$ ) and one for the target CPU ( $tdepl$ ). It is assumed that these two sets are disjoint. Recording of the objects deployed on the CPUs ensures that a bus is able to decide whether it can handle a remote call to a particular object.

The state of the bus consists of a single state variable  $mq$ , which represents the timed message queue. It consists of a sequence of call and return messages. Each of these is paired with a natural number indicating the delay left to pass before the bus can forward the message.

The main behaviour of the bus is described by the action *BusLane*. The bus waits to receive remote call messages from  $cpu1$  on channel  $rCall$  and remote return messages from  $cpu2$  on channel  $rRet$ . Such messages must target objects deployed on the source or target CPU, respectively, denoted by their membership in  $tdepl$  or  $sdepl$ . When such a call is received, the message is added to the queue, together with the delay value. Once no more messages can be received, and a timeout of one unit occurs, the bus proceeds to forward all messages and update times.

The algorithm first iterates while the message queue is non-empty, and the message at the head of the queue has a zero delay: it is ready for delivery. This being the case, the message is forwarded to the source or target CPU, depending on whether it is a return or call message. The message is then removed from the head of the queue. Once no more deliverable messages exist, the bus decrements the delays of all pending messages in the queue. It does this by applying the map function which subtracts 1 from the second element of each item in the queue.

## 4.8 Statements

We consider a core subset of VDM-RT statements that are shown in Table 10. Since we assume a static topology of active objects, we do not consider the **new** statement directly, though if object-oriented data structures are necessary to model the (passive) data structures of a VDM-RT model, then the semantics of our previous deliverable can be combined with our work here. On the other hand, if dynamic creation of active objects with threads is required, then please see the future work section (Section 7).

We assume a distinction is known between assignments to instance variables, which are qualified by **this**, and assignments to dynamically created local variables, such as parameters. The synchronous or asynchronous nature of each operation can be statically determined by examining the operation's signature, and thus this information can be

known at the call site. Thus we have both constructs for synchronous calls (**scall**) and asynchronous calls (**acall**). The statement structure follows the protocol described in Section 4.1, and appropriately yields to the CPU and synchronises state. All the statement semantics assume that prior to beginning, permission to execute has already been obtained, which is ensured by both thread and operation contexts.

The duration statement, **duration**( $e$ )( $S$ ), denotes an atomic region whose internal state updates only become visible after its completion. In terms of its behaviour with respect to timing parameter  $e$ , two possible interpretations of its semantics exist:

- $e$  denotes a time override: all previously known delay information for encapsulated statements is discarded, and the block's state behaviour is revealed only after  $e$  time units;
- $e$  denotes a strict deadline: if the encapsulated statements fail to complete within  $e$  time units, then the program behaviour is miraculous (or aborting).

This question is discussed at length in [54], Sections 4.1.2 and 4.7. The issue with the former interpretation is that it is non-compositional: in order to know the semantics of any statement one needs to know whether it is in the context of a duration or not, so that delays can be encoded appropriately. Thus it is difficult to develop VDM-RT programs in a modular way, since it is always necessary to know this contextual information. Nevertheless, this is the behaviour of the current Overture VDM-RT interpreter. The latter interpretation in contrast is compositional, and supports nested durations since an outer duration can simply sum up all the encapsulated time delays and compare to its deadline. The problem with strict deadlines, is that a real-time operating system is needed to ensure they can be honoured, which is beyond the scope of INTO-CPS.

Thus, in our semantics we opt for the current execution semantics of the Overture tool. However, we also note that deadlines and various other timing operators are supported by *CML*, and the latter interpretation can easily be taken without affecting our semantic model outside of the statement semantics. Nevertheless, a necessary consequence of this choice is that generation of the semantics of statements does require contextual information. We give the semantics of statements in terms of a semantic interpretation function  $\llbracket P \rrbracket_d^{\text{stm}}$ , where  $P$  is a statement block and parameter  $d$  can either take the value **df** or **dt**. If in the context of a duration statement,  $d$  will take the value **dt**, otherwise it has the value **df**.

The semantic interpretation rules for statements are shown in Table 11. We assume that the meta-tag  $\langle \text{Class}_k \rangle$ , along with the variables  $\text{objId}$  and  $\text{thrId}$ , are brought into scope by the enclosing class definition. Moreover, if the enclosing scope defines an operation, then we assume it brings the variable  $\text{retInf}$ , containing return information, into scope as well. Essentially the difference between the **dt** and **df** versions of the rules is that the former ignores timing information. We thus mainly consider only the **df** rules. Assignment of expression  $e$  to an instance variable  $x$  firstly waits  $e_t$  units, and then assigns the expression body  $e_v$  to the corresponding field in the state variable  $\text{this}$ . Additionally, since this is an instance variable, the update is added to the  $\text{upd}$  function to allow state synchronisation later. The behaviour of local variable assignment is identical, except that  $\text{upd}$  is not updated as the assignment is not visible outside the thread.

A synchronous operation call to operation  $f$  of object  $o$  first waits  $e_t$  time units, for

|   |              |   |
|---|--------------|---|
| $\llbracket \text{skip} \rrbracket_{\text{df}}^{\text{stm}}$                                  | $\triangleq$ | <b>Skip</b>   |
| $\llbracket \text{this}.x :=_{\text{rt}} e \rrbracket_{\text{df}}^{\text{stm}}$               | $\triangleq$ | <b>Wait</b> $e_t$ ; $\text{this}.x :=_{\text{sl}} e_v$ ; $\text{upd} := (\lambda \sigma \bullet (\text{upd}(\sigma))(x := e_v))$  |
| $\llbracket \text{this}.x :=_{\text{rt}} e \rrbracket_{\text{dt}}^{\text{stm}}$               | $\triangleq$ | $\text{this}.x :=_{\text{sl}} e_v$ ; $\text{upd} := (\lambda \sigma \bullet (\text{upd}(\sigma))(x := e_v))$  |
| $\llbracket x :=_{\text{rt}} e \rrbracket_{\text{df}}^{\text{stm}}$                           | $\triangleq$ | <b>Wait</b> $e_t$ ; $x :=_{\text{sl}} e_v$  |
| $\llbracket x :=_{\text{rt}} e \rrbracket_{\text{dt}}^{\text{stm}}$                           | $\triangleq$ | $x :=_{\text{sl}} e_v$  |
| $\llbracket \text{scall}(x, o, f, e) \rrbracket_{\text{df}}^{\text{stm}}$                     | $\triangleq$ | <b>Wait</b> $e_t$ ; $\text{call}!(o, f, \text{mk-SCall}(\underline{\text{objId}}, \underline{\text{thrId}}, \text{inj}_{e_\tau}(e_v))) \rightarrow \text{yield!} \rightarrow$<br>$\text{lRet}.\underline{\text{objId}}.\underline{\text{thrId}}?r \rightarrow \text{exec?} \rightarrow$<br>$\text{getState-}\langle \text{Class}_k \rangle?\sigma \rightarrow \text{this} :=_{\text{rea}} \text{upd}(\sigma) ; x :=_{\text{rea}} r$ |
| $\llbracket \text{scall}(x, o, f, e) \rrbracket_{\text{dt}}^{\text{stm}}$                     | $\triangleq$ | $\text{call}!(o, f, \text{mk-SCall}(\underline{\text{objId}}, \underline{\text{thrId}}, \text{inj}_{e_\tau}(e_v))) \rightarrow \text{yield!} \rightarrow$<br>$\text{lRet}.\underline{\text{objId}}.\underline{\text{thrId}}?r \rightarrow \text{exec?} \rightarrow$<br>$\text{getState-}\langle \text{Class}_k \rangle?\sigma \rightarrow \text{this} :=_{\text{rea}} \text{upd}(\sigma) ; x :=_{\text{rea}} r$                     |
| $\llbracket \text{acall}(o, f, e) \rrbracket_{\text{df}}^{\text{stm}}$                        | $\triangleq$ | <b>Wait</b> $e_t$ ; $\text{aCall}!(o, f, \text{mk-ACall}(\text{inj}_{e_\tau}(e_v))) \rightarrow \text{Skip}$  |
| $\llbracket \text{acall}(o, f, e) \rrbracket_{\text{dt}}^{\text{stm}}$                        | $\triangleq$ | $\text{aCall}!(o, f, \text{mk-ACall}(\text{inj}_{e_\tau}(e_v))) \rightarrow \text{Skip}$  |
| $\llbracket \text{duration}(e)(P) \rrbracket_{\text{df}}^{\text{stm}}$                        | $\triangleq$ | <b>Wait</b> $e_t$ ; $\llbracket P \rrbracket_{\text{dt}}^{\text{stm}} ; \text{yield!} \rightarrow \text{Wait } e_v ;$<br>$\text{syncState-}\langle \text{Class}_k \rangle!\text{upd} \rightarrow \text{exec?} \rightarrow$<br>$\text{getState-}\langle \text{Class}_k \rangle?\sigma \rightarrow \text{this} :=_{\text{rea}} \sigma ; \text{upd} :=_{\text{rea}} \text{id}$   |
| $\llbracket \text{duration}(e)(P) \rrbracket_{\text{dt}}^{\text{stm}}$                        | $\triangleq$ | $\llbracket P \rrbracket_{\text{dt}}^{\text{stm}}$  |
| $\llbracket \text{return } e \rrbracket_{\text{df}}^{\text{stm}}$                             | $\triangleq$ | <b>Wait</b> $e_t$ ; $\text{ret}!(\underline{\text{retInf.sobj}}, \underline{\text{retInf.sthr}}, \text{inj}_{e_\tau}(e_v)) \rightarrow \text{Skip}$   |
| $\llbracket \text{return } e \rrbracket_{\text{dt}}^{\text{stm}}$                             | $\triangleq$ | $\text{ret}!(\underline{\text{retInf.sobj}}, \underline{\text{retInf.sthr}}, \text{inj}_{e_\tau}(e_v)) \rightarrow \text{Skip}$   |
| $\llbracket P ; Q \rrbracket_{\text{df}}^{\text{stm}}$  | $\triangleq$ | $\llbracket P \rrbracket_{\text{df}}^{\text{stm}} ; \text{syncState-}\langle \text{Class}_k \rangle!\text{upd} \rightarrow \text{yield!} \rightarrow \text{exec?} \rightarrow$<br>$\text{getState-}\langle \text{Class}_k \rangle?\sigma \rightarrow \text{this} :=_{\text{rea}} \text{upd}(\sigma) \rightarrow \llbracket Q \rrbracket_{\text{df}}^{\text{stm}}$   |
| $\llbracket P ; Q \rrbracket_{\text{dt}}^{\text{stm}}$  | $\triangleq$ | $\llbracket P \rrbracket_{\text{dt}}^{\text{stm}} ; \llbracket Q \rrbracket_{\text{dt}}^{\text{stm}}$   |
| $\llbracket \text{if } e \text{ then } P \text{ else } Q \rrbracket_{\text{df}}^{\text{stm}}$ | $\triangleq$ | <b>Wait</b> $e_t$ ; $(P \triangleleft e_v \triangleright Q)$  |
| $\llbracket \text{if } e \text{ then } P \text{ else } Q \rrbracket_{\text{dt}}^{\text{stm}}$ | $\triangleq$ | $P \triangleleft e_v \triangleright Q$  |
| $\llbracket \text{while } e \text{ do } P \rrbracket_{\text{df}}^{\text{stm}}$                | $\triangleq$ | <b>Wait</b> $e_t$ ; <b>while</b> $e_v$ <b>do</b> $(\llbracket P \rrbracket_{\text{df}}^{\text{stm}} ; \text{Wait } e_t)$  |
| $\llbracket \text{while } e \text{ do } P \rrbracket_{\text{dt}}^{\text{stm}}$                | $\triangleq$ | <b>while</b> $e_v$ <b>do</b> $\llbracket P \rrbracket_{\text{dt}}^{\text{stm}}$   |

Table 11: Statement semantics

parameter expression evaluation, and then offers the call information of *call*, together with the current object identifier *objId*, thread identifier *thrId*, and parameter data  $e_v$ . It then yields to the CPU, and awaits a *lRet* event, parametrised by the object identifier, thread identifier, and return data. Once this is received, it waits for permission to resume execution, and once this is granted, synchronises the state and assigns the return data to variable  $x$ .

An asynchronous call is much simpler than a synchronous call, as there is no need to pause the current thread execution. Instead, a delay is issued for the parameter evaluation and then the *aCall* event is offered with the call parameters.

The semantics of the duration statement is sensitive to whether it is in the context of another duration statement or not. If it is – indicated by the presence of **dt** – then the semantic function ignores the duration statement. Otherwise, the statement is top-level and so indicates a timed atomic region. The semantics is to first wait for the duration expression,  $e$ , to be evaluated. The body of the duration statement is then executed, which must be instantaneous by the presence of the **dt** parameter. Once the body is completed, the duration statement yields, and then waits for the period of the duration  $e_v$  to elapse. After sufficient time has passed, the duration statement can reveal its state, and thus performs a state synchronisation. Note that due to the maximal progress property of *CML*, this synchronisation must happen before any more time is permitted to elapse in the system. Permission to execute is then sought, and finally the new state is obtained, and the update function is reset to the identity function. We do not directly give semantics to the **cycles** statement as these can be rewritten to duration statements by statically applying multiplication by the associated CPU's clock resolution.

The semantics of return is simple: a delay is issued for the return expression evaluation, and then the expression is offered over the *ret* channel.

The remaining semantics consider the imperative combinators. Sequential composition is just relational composition when in the context of a duration statement. When not in the context of a duration, sequential composition yields and synchronises the state in between the two statements. This is because it is then within a non-atomic section, and so each statement's effect must be immediately propagated to the central state. The *if* statement is simply a conditional, though with an appropriate delay to evaluate the conditional. Finally the while statement also must delay for evaluation, but in this case must do so once after each iteration.

## 4.9 System Instantiation

The final stage for giving a semantics to a VDM-RT system is instantiation of objects, CPUs, and busses, and connecting them together using the appropriate topology. This is exemplified in the collection of *CML* process in Figure 11. Each of the  $i \leq m$  objects to be created in the static topology of the INTO-CPS pattern,  $\langle Obj_i \rangle$  of class  $\langle ClassOfObj_i \rangle$ , is allocated a process  $VrtObj\text{-}\langle Obj_i \rangle$  that instantiates the class process with the object's name.

Each compute node  $\langle CPU_i \rangle$ , for  $i \leq n$ , is also allocated a process  $VrtNode\text{-}\langle CPU_i \rangle$  that composes the set of objects to be deployed with a CPU process, instantiated with the

```

process  $VrtObj\text{-}\langle Obj_1 \rangle = VrtClass\text{-}\langle ClassOfObj_1 \rangle(\langle \langle Obj_1 \rangle \rangle)$ 
...
process  $VrtObj\text{-}\langle Obj_m \rangle = VrtClass\text{-}\langle ClassOfObj_m \rangle(\langle \langle Obj_m \rangle \rangle)$ 

process  $VrtNode\text{-}\langle CPU_1 \rangle =$ 

$$\left( \begin{array}{l} (VrtObj\text{-}\langle Obj_1 \rangle \parallel \dots \parallel VrtObj\text{-}\langle Obj_j \rangle) \\ \llbracket exec, yield, newThr, call, lCall, ret, lRet \rrbracket \\ VrtCpu(\langle \langle CPU_1 \rangle \rangle, \{ \langle \langle Obj_1 \rangle \rangle, \dots, \langle \langle Obj_j \rangle \rangle \}) \end{array} \right) \setminus \left\{ \begin{array}{l} exec, yield, newThr \\ call, lCall, ret, lRet \end{array} \right\}$$

...
process  $VrtNode\text{-}\langle CPU_n \rangle = \dots$ 

process  $VrtSystem =$ 

$$\left( \begin{array}{l} (VrtNode\text{-}\langle CPU_1 \rangle \parallel \dots \parallel VrtNode\text{-}\langle CPU_n \rangle) \\ \llbracket cCall, cRet, rCall, rRet \rrbracket \\ \left( \begin{array}{l} VrtBus2(\langle \langle CPU_1 \rangle \rangle, \langle \langle CPU_2 \rangle \rangle, 1000, \{ \langle \langle Obj_1 \rangle \rangle, \dots, \langle \langle Obj_j \rangle \rangle \}, \{ \dots \}) \\ \parallel VrtBus2(\langle \langle CPU_2 \rangle \rangle, \langle \langle CPU_3 \rangle \rangle, 2000, \dots, \dots) \\ \parallel \dots \end{array} \right) \end{array} \right)$$


```

Figure 11: VDM-RT system instantiation



node name and set of object names. The channels for scheduling, time, and internal calls are then hidden, which ensures their events occur urgently within the CPU.

Finally, the actual system process *VrtSystem* is created, which composes the set of all compute nodes with a set of bus processes. Each bus connects two CPUs, identifies a delay, and specifies the objects deployed on each CPU.

## 5 Validation of semantics

In order to validate the semantics, that is, in order to demonstrate that it exhibits our expected behaviour, we have hand translated the CPU and bus infrastructure processes, along with the `controller`, `levelSensor`, and `valveActuator` objects from the year 2 version of the “Three Tanks” pilot study [?] to *CML*. The associated *CML* code can be found in Appendix A. We then applied the *Symphony* interpreter[21], which implements the *CML* operational semantics [11], to explore the behaviour of the model.

Our translation into *Symphony CML* requires that a few simplifications be made to the model. Firstly, the VDM-RT universe,  $\mathbb{U}_{\text{si}}$ , cannot properly be created in *Symphony* without extension and so we opted to use the  $\mathbb{R}$  type as our universe (which has the same cardinality). Secondly, timed expressions are not directly supported, and so when generating statement semantics we manually inserted concrete time values for the appropriate **Wait** statements. Other than these two caveats, the model corresponds to the mathematics contained in this deliverable, though with a slightly different syntax.

The encoding was useful for several reasons. Firstly, we were able to type check the model, ensuring correct typing of variables, expressions, and channels. Secondly, we were able to simulate the model using the *Symphony* interpreter, which also allowed some debugging of the semantics. For example, in a prior version of our semantics call messages were not removed from the active operation map during a remote call, and the simulator allowed us to spot this.

An example simulation can be seen in Figure 12, in which we have the `WaterTank1` process from Appendix A.2 loaded, which has the three objects loaded onto a single CPU. For the purpose of this simulation we expose the channels *exec*, *call*, and *yield* so that we can observe the internal interaction of the objects. The pane to the top-right (“Observable ...” tab) gives the sequence of events that we have stepped through so far. In the first step the periodic thread of `controller` is created, a new identifier (1) is allocated to it using the *newThr* channel, and the thread requests permission to execute on *exec*. Then, it makes a call to the `levelSensor` object, operation `getLevel`, with call information corresponding to the calling object, thread, and parameter data (in this case 0 as we are using reals as the universe and the operation takes no parameters). The periodic thread then yields, to wait for the `getLevel` thread to execute. A new thread is created for the latter, and this requests permission to execute. It then needs to evaluate an internal expression which takes some time, and so a *tock* event occurs next. Then, the operation has completed and so it returns its value (0) on the *ret* channel. This thread then yields, and the periodic thread of `controller` then becomes live again. It makes a further call to `setValve` on `valveActuator`, yields, and then the corresponding operation thread, is created, gains permission to execute, and takes some time to evaluate an internal expression.



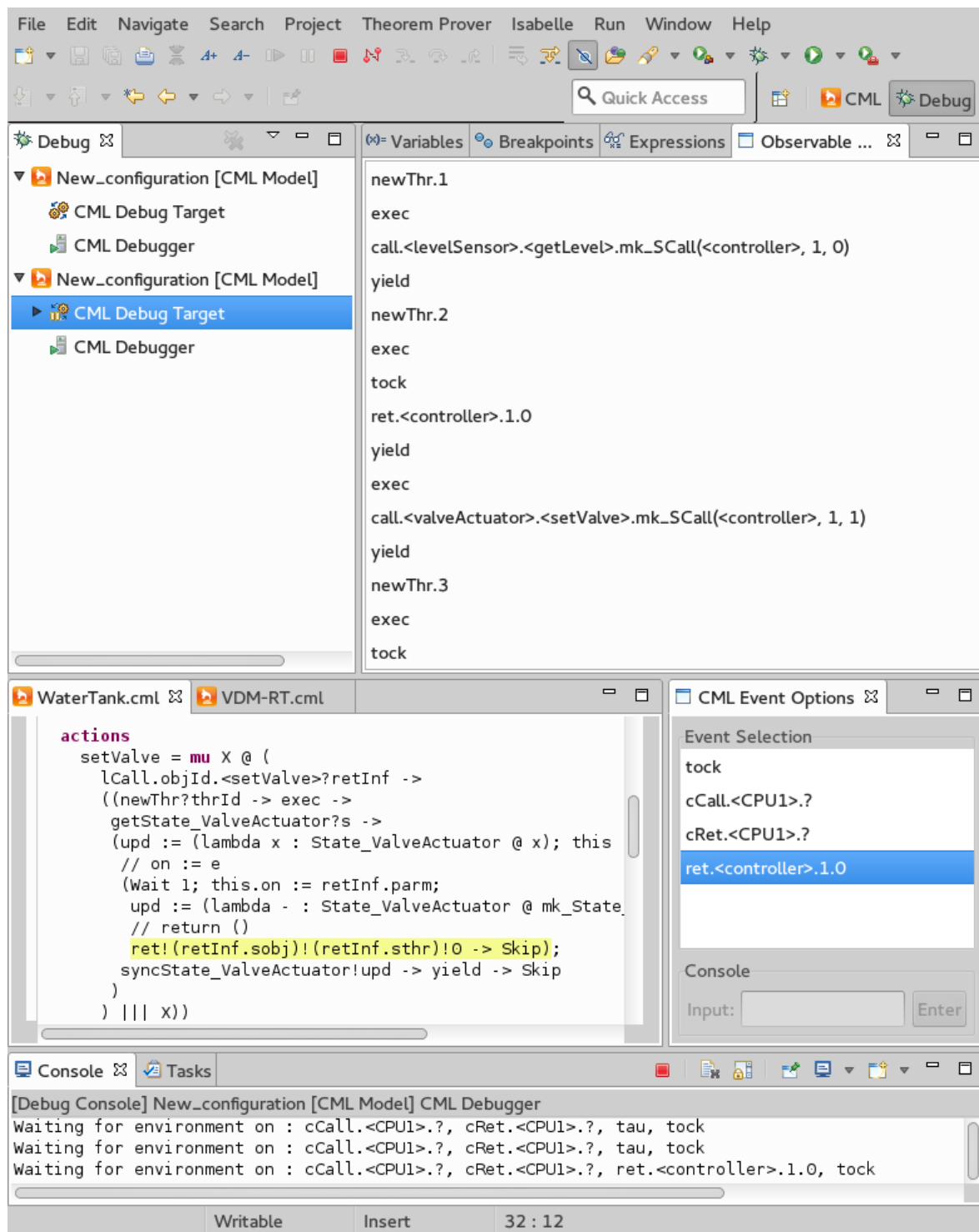
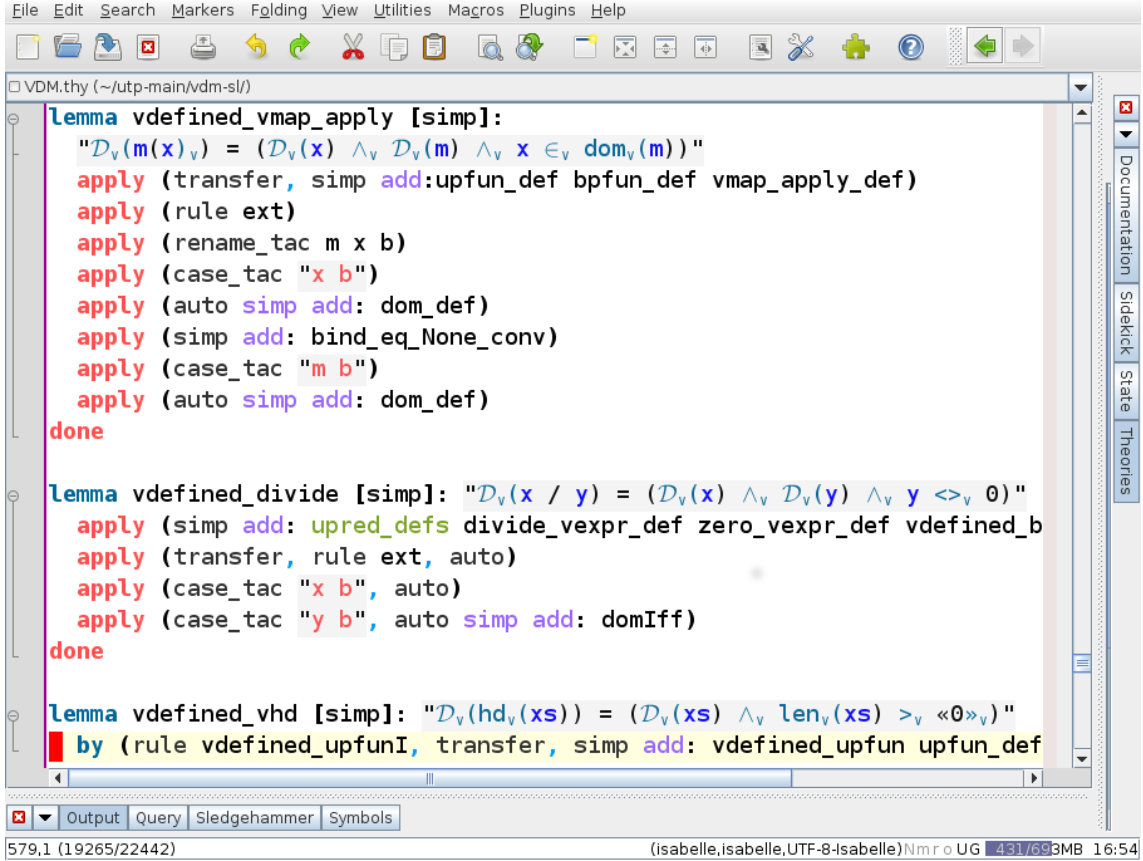


Figure 12: Trace of the water tanks system running on a single CPU



```

VDM.thy (~:/utp-main/vdm-sl/)
lemma vdefined_vmap_apply [simp]:
  "Dv(m(x)v) = (Dv(x) ∧v Dv(m) ∧v x ∈v domv(m))"
  apply (transfer, simp add: upfun_def bpfun_def vmap_apply_def)
  apply (rule ext)
  apply (rename_tac m x b)
  apply (case_tac "x b")
  apply (auto simp add: dom_def)
  apply (simp add: bind_eq_None_conv)
  apply (case_tac "m b")
  apply (auto simp add: dom_def)
done

lemma vdefined_divide [simp]: "Dv(x / y) = (Dv(x) ∧v Dv(y) ∧v y <>v 0)"
  apply (simp add: upred_defs divide_vexpr_def zero_vexpr_def vdefined_b)
  apply (transfer, rule ext, auto)
  apply (case_tac "x b", auto)
  apply (case_tac "y b", auto simp add: domIff)
done

lemma vdefined_vhd [simp]: "Dv(hdv(xs)) = (Dv(xs) ∧v lenv(xs) >v «0»v)"
  by (rule vdefined_upfunI, transfer, simp add: vdefined_upfun upfun_def)

```

579,1 (19265/22442) (isabelle,isabelle,UTF-8-Isabelle)Nm r o UG 431/693MB 16:54

Figure 13: Proof of definedness predicates in *Isabelle/UTP*

At the end of this sequence of actions, the active section of the *CML* file is shown in the bottom-left pane, and the events that are being offered is shown in the bottom-right pane. Currently, the `setValve` operation is about to return by communication on the `ret` channel. Additional events include the `tock` event (as time could pass before the return occurs), and also the events for communicating with an external bus (there is no composed bus in the model).

In addition to the simple case of `WaterTank1`, we also define `WaterTank2`, which is the same except for the fact that the `LevelSensor` object is deployed on a remote CPU. Simulating this allows the delayed exchange of call and return messages between the two CPUs to be observed.

## 6 Mechanisation in Isabelle/UTP

In addition to the validation, we have also made substantial progress towards the mechanisation of our semantics in *Isabelle/UTP*. We have mechanised the universe for VDM-RT,  $\mathbb{U}_{sl}$ , and proved injectivity theorems for the majority of VDM-SL type equivalents in Isabelle. In particular, this includes the real numbers which necessitated the mechanisation of Cantor's proof that reals can be represented as infinite binary sequences, which are

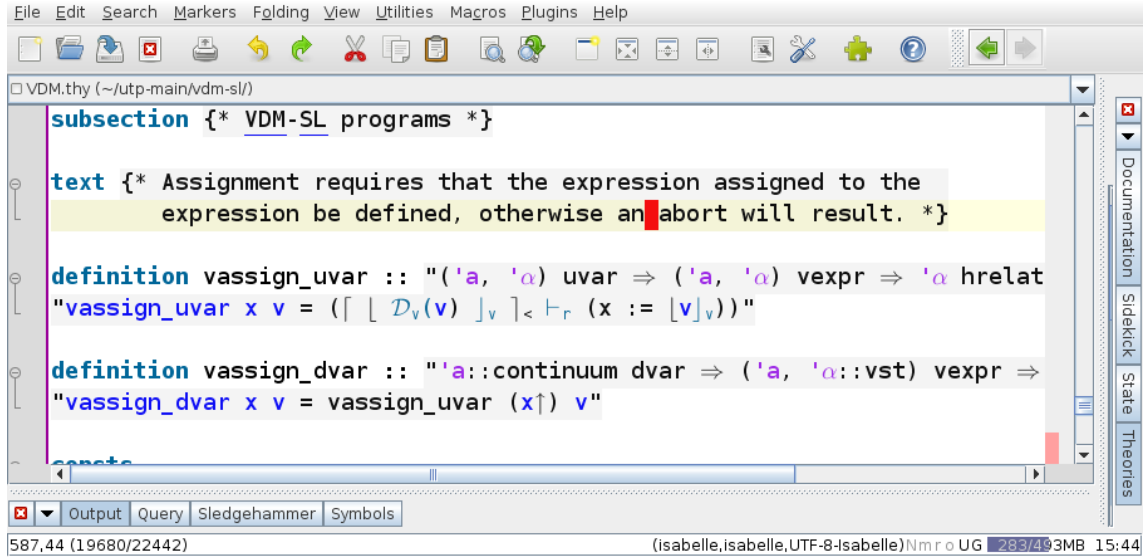


Figure 14: VDM-SL assignment in Isabelle/UTP

the equivalent to  $\mathbb{P}\mathbb{N}^5$ .

As we mentioned in Section 2.5, we already mechanised the theory of designs and reactive designs, and this allows us to represent the semantics of VDM-SL expressions and statements. In particular, it allows us to account for expression (un)definedness. We define a model for VDM-SL expressions,  $(\tau, \alpha) \text{ vexpr}$ , for return type  $\tau$  and alphabet type  $\alpha$ , based on partial functions. We then use this to define the constructs of Section 3.2, along with all the functions for manipulating numbers, sequences, sets, maps etc., which is done by lifting corresponding functions in Isabelle/HOL. HOL functions are total and do not give an explicit account of undefinedness: an arbitrary but defined value is returned for such cases. Thus in order to account for undefinedness we introduce the following additional lifting partial functions:

$$\begin{aligned}
 \text{vuop} &: (\sigma \rightarrow \tau) \rightarrow (\sigma, \alpha) \text{ vexpr} \rightarrow (\tau, \alpha) \text{ vexpr} \\
 \text{vbop} &: (\sigma_1 \times \sigma_2 \rightarrow \tau) \rightarrow (\sigma_1, \alpha) \text{ vexpr} \rightarrow (\sigma_2, \alpha) \text{ vexpr} \rightarrow (\tau, \alpha) \text{ vexpr}
 \end{aligned}$$

These functions lift unary and binary partial functions to VDM-SL expressions. They are both strict in the sense that  $\text{vuop } f \perp_v = \text{vbop } f \perp_v e = \text{vbop } f e \perp_v$ , that is, if either parameter is undefined, then the whole expression is also undefined. We can then define partial operations, like division, by lifting as the following definitions demonstrate:

$$\begin{aligned}
 \llbracket e/f \rrbracket &= \text{vbop } \{(m, n) \mapsto m/n \mid n \neq 0\} e f \\
 \llbracket \text{hd}(e) \rrbracket &= \text{uop } \{xs \mapsto \text{hd}(xs) \mid xs \neq []\} e \\
 \llbracket f(x) \rrbracket &= \text{vbop } \{(m, k) \mapsto m(k) \mid k \in \text{dom}(m)\} f x
 \end{aligned}$$

We define division by restricting the domain of the function to those whose denominator is non-zero. Similarly for the `hd` function that takes the head of a list, we disallow non-empty lists. Finally, for application of a map  $f$  to a key  $x$ , we require that  $x$  is in the domain of  $f$ . Naturally, operators that are not partial can be trivially lifted. We can then use such

<sup>5</sup>This mechanised proof is located at [https://github.com/isabelle-utp/utp-main/blob/master/utls/Real\\_Bit.thy](https://github.com/isabelle-utp/utp-main/blob/master/utls/Real_Bit.thy)

```

lemma hd_nil_abort:
  fixes x :: "('a, 'α) uvar"
  shows "(x :=v hdv([]v)) = true"
  by rel_tac

text {* Here we augment the set of design weakest precondition laws
  with the VDM assignment operator *}

theorem wpd_vdm_assign [wp]:
  fixes x :: "('a, 'α) uvar"
  shows "(x :=v v) wpD r = ([ $\mathcal{D}_v(v)$ ]v ∧ r[[v]v/x])"
  by (simp add: vassign_uvar_def wp)

lemma wp_calc_test_1:
  "[[ uvar x; uvar y ]] ⇒ (y :=v hdv(&vx)) wpD true
  = [ $\mathcal{D}_v(\&_v x) \wedge_v \text{len}_v(\&_v x) >_v \ll 0 \gg_v$ ]v"
  by (simp add: wp usubst)

lemma wp_calc_test_2:
  "[[ uvar x; uvar y ]] ⇒ (y :=v 1 / hdv(&vx)) wpD true
  = [ $\text{len}_v(\&_v x) >_v \ll 0 \gg_v \wedge_v \text{hd}_v(\&_v x) <>_v 0$ ]v"
  by (simp add: wp usubst)

```

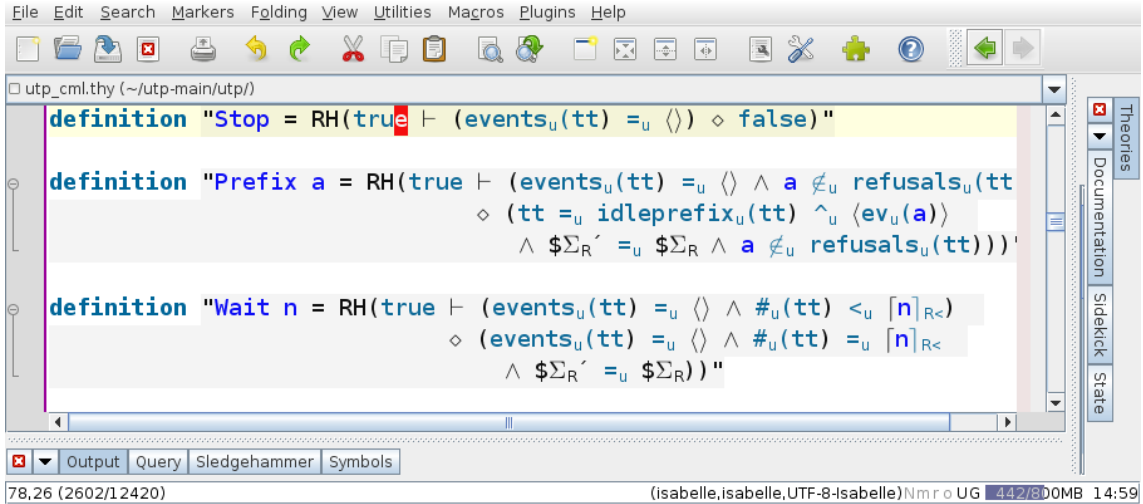
Figure 15: VDM-SL assignment experiments

definitions to prove definedness theorems in Isabelle as illustrated in Figure 13. Amongst other laws, we prove that definedness of  $x/y$  depends on definedness of  $x$ , definedness of  $y$ , and  $y$  being non-zero.

We then define VDM-SL assignment, as illustrated in Figure 14. The definition is near identical to that of Section 3.2 except that we have to drop the VDM-SL expression  $\mathcal{D}_v(v)$  to a UTP expression using the  $[-]_v$  operators to ignore definedness values (the definedness predicate is always defined), and then we lift this precondition expression to a relation using  $[-]_<$ .

The definition of assignment can be used to prove some intuitive properties about VDM-SL program fragments, as shown in Figure 15. We first show that assignment of  $\text{hd}([])$ , which is an undefined expression, to a variable equates to the relation **true**, an abort. Secondly, we prove a weakest precondition law about VDM-SL assignment that states, for  $x := v$  to satisfy postcondition  $r$ , it must at least be the case that  $v$  is defined, and  $r$  is true when  $v$  is substituted for  $x$ . This law can then be applied to show, for example, when VDM-SL programs do not abort (i.e. the associated proof obligations). Lemma `wp_calc_test_1` states that  $y := \text{hd}(x)$  does not abort provided  $x$  is defined, and that the length of  $x$  is greater than 0. Lemma `wp_calc_test_2` states that  $y := 1/\text{hd}(x)$  does not abort if, in addition, the head of  $x$  is not zero. Thus the mechanisation could in theory be used in the future to validate the proof obligations generated by the *Overture* tool.

Aside from the VDM-SL operators, we have also made progress towards mechanisation of *CML* itself. We have mechanised a substantial number of laws associated with reactive processes, designs, and reactive designs. Moreover we have begun to mechanise the

Figure 16: CML operator definitions in *Isabelle/UTP*

operators of *CML*, as illustrated in Figure 16, using the reactive triple notation [13]. Note that **RH** corresponds to **R** as we described in Section 2.3 (R being a commonly used name). As can be seen, we are able to mimick closely the syntax, though sometimes require *u* subscripts to differentiate operators from their HOL counterparts. Finally, we are also able to prove algebraic laws about *CML*. In Figure 17 we show part of the proof for **Wait** *m* ; **Wait** *n* = **Wait** *m* + *n*.

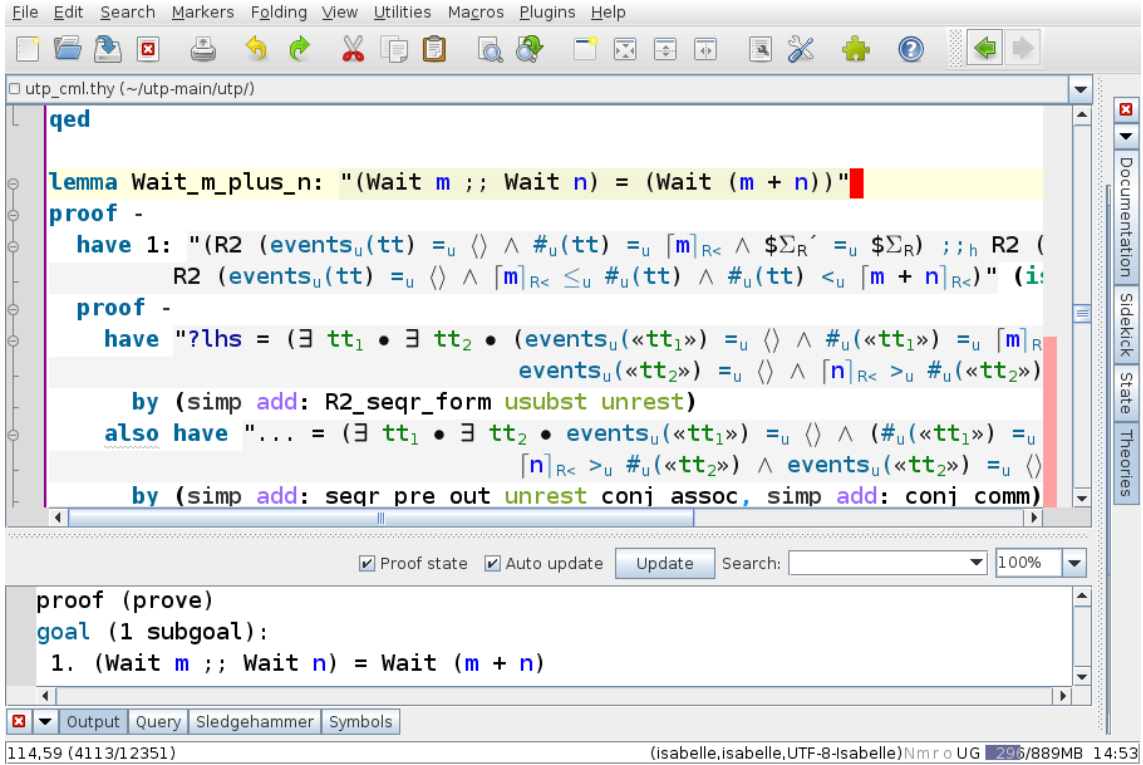
## 7 Areas for future work

### 7.1 Dynamic topologies

Our current semantics supports only a static topology of objects, though each may have an unbounded number of threads. As per our specified pattern, all controllers, sensors, and actuators are created at the beginning and no further objects can be created. This has the advantage of tractability, in that the system cannot grow arbitrarily and is thus easier to analyse. However, this may be considered too restrictive for some applications. The present semantic framework can thus be appropriately adapted to allow dynamic object topologies.

The first step to allow dynamic topologies would be to alter the definition of *ObjId* to an infinite rather than a finite type. This would then allow an arbitrary number of object identifiers, and thus objects. Moreover a separate type *ClassId* is necessary to support naming of classes as first class citizens, which can either be finite or infinite. Then the **new** statement, for creating a new object of a specified class, can be implemented.

There are various ways to support construction of new objects, and in particular how they act with respect to CPUs and threads. A relatively simple approach is a version of the **new** statement that spawns a new copy of the object process on the same CPU as the call site. Objects passed as parameters are simply identifiers, as before, and this avoids the complexities of object mobility. If mobility is required, then it will be necessary to adopt one of the appropriate *Circus* extensions for the semantic model [65, 58].

Figure 17: Part of the proof of a **Wait** law

## 7.2 Passive and active classes

VDM-RT classes can either be “passive” or “active” in nature. An active class has threads associated with it, whereas a passive class does not. Combining active classes and object-oriented features such as inheritance is non-trivial as it requires consideration of how active behaviour should be inherited in subclasses. One possible solution is to introduce a form of behavioural subtyping [55], whereby the thread of a subclass must refine that of the corresponding superclass. This would require substantial proof machinery to support checking of such conformances. When considering features like VDM-RT’s multiple inheritance, where two active classes could be inherited, things are even more complicated.

An alternative approach is to provide inheritance only for passive classes, such that a class may only extend a passive class and not an active one. Doing this would require the VDM-RT type checker to be able to distinguish whether a class is active or passive. Semantics of object-orientation for passive classes can follow our previous deliverable [39]. Passive objects can then be passed around like normal data structures, without the complexity of considering active behaviour.

## 7.3 Timed specifications

VDM-SL contains a specification statement that is adopted from refinement calculus [57], and takes the form of `[ext f pre p post q]`. It is a non-deterministic statement that the code should, provided that precondition `p` is satisfied, fulfil obligation `q`, within the state

variable frame  $\mathbf{f}$ . The statement can be used to aid in refining implicit operation contracts into explicit code.

For VDM-RT, it may also be helpful to encode timing constraints in the specifications, such as hard deadlines, which would allow the expression of timing budgets. We could for instance modify the specification statement to `[ext f pre p post q wcet t]`, where  $t$  is an expression denoting the worst case execution time in terms of the input variables. Fulfilling such behaviour may require deployment of VDM-RT code onto a real-time operating system.

## 7.4 Model checking in FDR3

FDR3 [40] is a model checker that allows the verification of CSP [45] processes through the specification of suitable refinement conjectures. The input language of FDR3 is a CSP dialect called *CSP-M*. The *Circus* modelling language, which forms the foundation for *CML*, has previously been abstracted to CSP-M for the purpose of model checking *Circus* specifications [61, 6, 7]. A similar approach could be applied to *CML* which could then be applied to model checking VDM-RT models. For example, FDR3's deadlock freedom checker could allow us to locate remote operations that recursively call each other. Such a translation would require the representation of time, which in CSP-M could be done using *tock*-CSP, a subset of the untimed language with a distinguished event to represent time.

## 7.5 Timed state machines for RTT-MBT

RTT-MBT is a toolkit for model-based testing which forms part of the INTO-CPS tool chain [5]. It provides facilities for both production of test suites, based on an abstract system model and high-level requirements, and also for model checking. A test system consists of a *System Under Test* (SUT), which describes the part of the system we are testing, and a *Test Environment*, which provides a context for the SUT. The main language for describing the SUT's abstract behaviour is a form of hierarchical timed state machines, which include timers that introduce waits, delays and resets. Once such an abstract model has been created, RTT-MBT can semi-automatically construct a suitable test suite, and use it to verify a concrete implementation.

VDM-RT models are likewise timed, and it would be highly advantageous if they could be used as an input language to RTT-MBT. In the context of INTO-CPS, this would allow us to use a VDM-RT model to provide testing and model checking for corresponding optimised deployed controller code. In order to do this, it would be necessary to convert a VDM-RT model to a timed automaton. In general this is not possible, for example since transcendental functions like those of trigonometry, if present in a controller, cannot easily be mapped. However, if a simpler controller specification is present, as is the case for many of the pilot studies and case studies, then this could be performed by suitably abstracting the variables to be exposed by the timed automata. The denotational semantics presented in this deliverable could provide a basis for this mapping, when combined with the *CML* operational semantics [11].



## 7.6 Encoding FMUs

Sister deliverable D2.2d [19], and the associated paper [15], presents a *Circus*-based semantics for FMI. FMUs are represented as processes that expose channels to orchestrate simulation, such as *fmi2Set*, which sets the FMU state, *fmi2Get*, which retrieves the current state, and *fmi2DoStep* that advances a model by a certain amount of time. In order to give formal semantics to entire INTO-CPS multi-models it will be necessary to cast the various model semantics into a form with this interface. For the VDM-RT semantics, this could be done by adding special objects that interface with the FMI channels. We will consider this in the final year of INTO-CPS, as part of our work on the *lingua franca* *CyPhyCircus*, which will subsume *CML*.

## 8 Conclusion

We have presented a denotational semantics for VDM-RT based on a pattern for Cyber-Physical Systems in the INTO-CPS project using the *CML* formal real-time modelling language. We gave semantic mappings for VDM-RT constructs, including classes, objects, operations, threads, CPUs, and busses, all of which become forms of *CML* process. We then validated this semantics through an encoding into the *Symphony CML* tool. Finally we showed our work towards mechanisation of *CML* in *Isabelle/UTP*. Though Task 2.2 ends with this deliverable, next year we will use the semantics contained herein to develop INTO-CPS multi-models using *CyPhyCircus*.

# Appendices

## A VDM-RT Symphony Model

### A.1 Infrastructure Processes

```
types
  Univ = real
  ThrId = nat
  SCall :: sobj : ObjId  sthr : ThrId  parm : Univ
  ACall :: parm : Univ
  Call = SCall | ACall
  BusCallMsg :: tobj : ObjId  topr : OpId  call : Call
  BusRetMsg  :: robj : ObjId  rthr : ThrId  rval : Univ

/* The decDelay function decrements the remaining delay associated
   with each message in the queue. We implement it this way as
   VDM-SL has no map function built-in. */

functions
  mDelay: seq of ((BusCallMsg | BusRetMsg) * nat) -> nat
  mDelay(xs) == len xs
```



```

decDelay: seq of ((BusCallMsg | BusRetMsg) * nat) ->
    seq of ((BusCallMsg | BusRetMsg) * nat)
decDelay(xs) ==
    if (xs = [])
    then []
    else [mk_((hd(xs)).#1, (hd(xs)).#2-1)] ^ decDelay(tl(xs))
measure mDelay

channels
yield
exec
newThr : ThrId
call : ObjId * OpId * Call
lCall : ObjId * OpId * Call
ret : ObjId * ThrId * Univ
lRet : ObjId * ThrId * Univ
rCall : CPUId * BusCallMsg
rRet : CPUId * BusRetMsg
cCall : CPUId * BusCallMsg
cRet : CPUId * BusRetMsg

process VrtCpu = cpu : CPUId, depl: set of ObjId @ begin
state
    lastThr : ThrId := 0
actions
    ThrSched = mu X @ (exec -> yield -> X)
    ThrMgr = mu X @ (newThr!(lastThr + 1) -> lastThr := lastThr + 1 ; X)
    OpMgr =
        mu X @
            (call?o?f?c ->
                (if (o in set depl)
                    then (lCall!o!f!c -> X)
                    else (rCall.cpu!(mk_BusCallMsg(o, f, c)) -> X))

            [] ret?p?t?v ->
                (if (p in set depl)
                    then (lRet!p!t!v -> X)
                    else (rRet.cpu.mk_BusRetMsg(p, t, v) -> X))
            [] (cCall.cpu?b -> lCall!(b.tobj)!(b.topr)!(b.call) -> X)
            [] (cRet.cpu?r -> lRet!(r.robj)!(r.rthr)!(r.rval) -> X)
        )
    @ (ThrSched ||| ThrMgr ||| OpMgr)
end

process VrtBus = cpu1 : CPUId, cpu2 : CPUId, delay : nat,
    sdepl : set of ObjId, tdepl : set of ObjId @ begin
state mq : seq of ((BusCallMsg | BusRetMsg) * nat) := []

actions
    BusLane = mu X @ ((rCall.cpu1?m:(m.tobj in set tdepl) ->
        mq := mq ^ [mk_(m, delay)] ; X
    [] rRet.cpu2?m:(m.robj in set sdepl) ->
        mq := mq ^ [mk_(m, delay)] ; X)
    [_1_> while (len(mq) > 0 and (hd(mq)).#2 = 0)
        do ((if (is_BusRetMsg((hd(mq)).#1))
            then cRet.cpu1!((hd(mq)).#1) -> Skip
            else cCall.cpu2!((hd(mq)).#1) -> Skip);
        mq := tl(mq));

```

```
mq := decDelay(mq); X)
```

```
@ BusLane
end
```

## A.2 Water Tank Controller Instantiation

```
types
  CPUId = <CPU1> | <CPU2> | <CPU3>
  ObjId = <controller> | <levelSensor> | <valveActuator>
  OpId = <getLevel> | <setValve>
  State_LevelSensor :: level : real
  State_ValveActuator :: on : real
  State_Controller ::

channels
  getState_LevelSensor : State_LevelSensor
  syncState_LevelSensor : State_LevelSensor -> State_LevelSensor
  getState_ValveActuator : State_ValveActuator
  syncState_ValveActuator : State_ValveActuator -> State_ValveActuator
  getState_Controller : State_Controller
  syncState_Controller : State_Controller -> State_Controller

process VrtClass_LevelSensor = objId : ObjId @ begin
  state
    this : State_LevelSensor := mk_State_LevelSensor(0)
    upd : State_LevelSensor -> State_LevelSensor
      := (lambda x : State_LevelSensor @ x)
  actions
    getLevel = mu X @ (
      lCall.objId.<getLevel>?retInf ->
      ((newThr?thrId -> exec ->
        getState_LevelSensor?s ->
        (upd := (lambda x : State_LevelSensor @ x); this := s;
          // return level
          (Wait 1; ret!(retInf.sobj)!(retInf.sthr)!(this.level) -> Skip);
          syncState_LevelSensor!upd -> yield -> Skip
        )
      ) ||| X))

    StateMgr = mu X @ (getState_LevelSensor!this -> X
      [] syncState_LevelSensor?s -> this := s(this) ; X)

  @ ((getLevel)
    [|{getState_LevelSensor, syncState_LevelSensor}|]
    StateMgr) \ \ {getState_LevelSensor, syncState_LevelSensor}
end

process VrtClass_ValveActuator = objId : ObjId @ begin
  state
    this : State_ValveActuator := mk_State_ValveActuator(0)
    upd : State_ValveActuator -> State_ValveActuator
      := (lambda s : State_ValveActuator @ s)
    time : nat := 0

  actions
```

```

setValve = mu X @ (
  lCall.objId.<setValve>?retInf ->
  ((newThr?thrId -> exec ->
    getState_ValveActuator?s ->
    (upd := (lambda x : State_ValveActuator @ x); this := s;
      // on := e
      (Wait 1; this.on := retInf.parm;
        upd := (lambda - : State_ValveActuator
          @ mk_State_ValveActuator(retInf.parm));
          // return ()
          ret!(retInf.sobj)!(retInf.sthr)!0 -> Skip);
        syncState_ValveActuator!upd -> yield -> Skip
      )
    ) ||| X))
StateMgr = mu X @ (getState_ValveActuator!this -> X
  [] syncState_ValveActuator?s -> this := s(this) ; X)
@ ((setValve)
  [|{getState_ValveActuator, syncState_ValveActuator}|]
  StateMgr) \ \ {getState_ValveActuator, syncState_ValveActuator}
end

process VrtClass_Controller = objId : ObjId @ begin
  values
    maxLevel: real = 10
    minLevel: real = 5
  state
    this : State_Controller := mk_State_Controller()
    upd : State_Controller -> State_Controller
      := (lambda s : State_Controller @ s)
    time : nat := 0
  actions
    Thread =
      // Wait for offset period (in this case 0)
      Wait 0;
      mu X @
        (newThr?thrId -> exec ->
          getState_Controller?s ->
          (upd := (lambda x : State_Controller @ x); this := s;

          ((dcl level : real @

            // level := levelSensor.getLevel()
            call!<levelSensor>!<getLevel>
              !mk_SCall(<controller>, thrId, 0) ->
            yield ->
            lRet.objId.thrId?r -> (
              exec -> getState_Controller?s -> this := upd(s);
              level := r;

              // if (level <= minLevel) then valveActuator.setValve(true)
              if (level <= minLevel)
                then (call!<valveActuator>!<setValve>
                  !mk_SCall(objId, thrId, 1) ->
                  yield -> lRet.objId.thrId?r ->
                  exec -> getState_Controller?s ->
                  this := upd(s))

              // if (level >= maxLevel) then valveActuator.setValve(false)

```

```

        else if (level >= maxLevel)
        then (call!<valveActuator>!<setValve>
              !mk_SCall(objId, thrId, 0) ->
              yield -> lRet.objId.thrId?r ->
              exec -> getState_Controller?s ->
              this := upd(s)))));

    // Yield, and wait for the end of the duration (4)
    yield -> Wait 4; syncState_Controller!upd -> exec ->

    /* Synchronise state, and wait the remaining
       time before iterating */
    getState_Controller?s -> this := s;
    upd := (lambda x : State_Controller @ x);
    syncState_Controller!upd -> Skip) ||| Wait 20); yield -> X)

StateMgr = mu X @ (getState_Controller!this -> X
                  [] syncState_Controller?s -> this := s(this) ; X)

@ (Thread
  [|{getState_Controller, syncState_Controller}||
   StateMgr) \ \ {getState_Controller, syncState_Controller}
end

process VrtObj_levelSensor = VrtClass_LevelSensor(<levelSensor>)
process VrtObj_valveActuator = VrtClass_ValveActuator(<valveActuator>)
process VrtObj_controller = VrtClass_Controller(<controller>)

process VrtNode_CPU1 =
  ((VrtObj_controller ||| VrtObj_levelSensor ||| VrtObj_valveActuator)
   [|{exec, yield, newThr, call, ret, lCall, lRet}||
    VrtCpu(<CPU1>, {<levelSensor>, <valveActuator>, <controller>})
   ) \ \ {exec, yield, newThr, call, ret, lCall, lRet}

process VrtNode_CPU2 =
  ((VrtObj_controller ||| VrtObj_valveActuator)
   [|{exec, yield, newThr, call, ret, lCall, lRet}||
    VrtCpu(<CPU2>, {<valveActuator>, <controller>})
   ) \ \ {exec, yield, newThr, call, ret, lCall, lRet}

process VrtNode_CPU3 =
  (VrtObj_levelSensor
   [|{exec, yield, newThr, call, ret, lCall, lRet}||
    VrtCpu(<CPU3>, {<levelSensor>})
   ) \ \ {exec, yield, newThr, call, ret, lCall, lRet}

// Standard deployment; all objects running on same CPU
process WaterTank1 = VrtNode_CPU1

/* Alternative deployment: LevelSensor running on a different CPU with a
   bus with associated delay of 2 connecting them */

process WaterTank2 =
  (VrtNode_CPU2 ||| VrtNode_CPU3)
  [|{rCall, rRet, cCall, cRet}||
   VrtBus(<CPU2>, <CPU3>, 2, {<valveActuator>, <controller>}, {<levelSensor>})

```

## B Lenses

### B.1 Introduction

Predicative programming [41] is a unification technique that uses predicates to describe abstract program behaviour and executable code alike. Programs are denoted as logical predicates that characterise the observable behaviours as mappings between the state before and after execution. Thus one can apply predicate calculus to reason about programs, as well as prove the algebraic laws of programming themselves [44]. These laws can then be applied to construct semantic presentations for the purpose of verification, such as operational semantics, Hoare calculi, separation logic, and refinement calculi, to name a few [2, 23]. This further enables the application of automated theorem provers to build program verification tools, an approach which has seen multiple successes [1, 51].

Modelling the state space of a program and manipulation of its variables is a key problem to be solved when building verification tools [64]. Whilst relation algebra, Kleene algebra, quantales, and related algebraic structures provide excellent models for point-free laws of programming [33, 3], when one considers point-wise laws for operators that manipulate state, like assignment, additional behavioural semantics is needed. State spaces can be heterogeneous — that is consisting of different representations of state and variables. For example, separation logic [12] considers both the store, a static mapping from names to values, and the heap, a dynamic mapping from addresses to values. Nevertheless, one would like a uniform interface for different variable models to facilitate the definition and use of generic laws of programming. When considering parallel programs [46], one also needs to consider subdivision of the state space into non-interfering regions for concurrent threads, and their eventual reconciliation post execution. Moreover, we have the overarching need for meta-logical operators on state, like variable substitution and freshness, that are often considered informally but are vital to express and mechanise many laws of programming [44, 41, 46].

In this paper, we propose *lenses* [30] as a unifying solution to state-space modelling. Lenses provide a solution to the view-update problem in database theory [31], and are similarly applied to manipulation of data structures in functional programming [26]. They employ well-behaved *get* and *put* functions to identify a particular view of a source data structure, and allow one to perform transformations on it independently of the wider context.

Our contribution is an extension of the theory of lenses that allows their use in modelling variables as abstract views on program state spaces with a uniform semantic interface. We define a novel lens algebra for manipulation of variables and state spaces, including separation-algebra-style operators [12] such as state (de)composition, that enable abstract reasoning about program operators that modify state spaces in sophisticated ways. Our algebra has been mechanised in Isabelle/HOL [59] and includes a repository of verified lens laws.

We apply the lens algebra to model heterogeneous state space models within the context of Hoare and He’s Unifying Theories of Programming [46, 18] (UTP), a predicative programming framework with an incremental and modular approach to denotational model construction. Therein, we use lenses to semantically model UTP variables and the predi-

$$\begin{array}{llll}
x := v & \triangleq & x' = v \wedge y' = y & P ; Q \triangleq \exists x_0 \bullet P[x_0/x'] \wedge Q[x_0/x] \\
(P \triangleleft b \triangleright Q) & \triangleq & (b \wedge P) \vee (\neg b \wedge Q) & P^* \triangleq \nu X \bullet P ; X
\end{array}$$

Table 12: Imperative programming in the alphabetised relational calculus

cate calculus' meta-logical functions, with no need for explicit abstract syntax, and thence provide a purely algebraic basis for the meta-logical laws, predicate calculus laws, and the laws of programming. We have further used Isabelle/HOL to mechanise a large repository of UTP laws; this both validates the soundness of our lens-based UTP framework and, importantly, paves the way for future program verification tools<sup>6</sup>.

The structure of our paper is as follows. In §B.2, we provide background material and related work. In §B.3, we present a mechanised theory of lenses, in the form of an algebraic hierarchy, concrete instantiations, and algebraic operators, including a useful equivalence relation. This theory is standalone, and we believe has further applications beyond modelling state. Crucially, all the constructions we describe require only a first-order polymorphic type system which makes it suitable for Isabelle/HOL. In §B.4, we apply the theory of lenses to show how different state abstractions can be given a unified treatment. For this, we construct the UTP's relational calculus, associated meta-logical operators, and prove various laws of programming. Along the way, we show how our model satisfies various important algebraic structures to validate its adequacy. We also use lenses to give an account to parallel state in §B.6. Finally, in §B.7, we conclude.

## B.2 Background and related work

### B.2.1 Unifying Theories of Programming

The UTP [46] is a framework for defining denotational semantic models based on an alphabetised predicate calculus. A program is denoted as a set of possible observations. In the relational calculus, imperative programs are in view and thus observations consist of before variables  $x$  and after variables  $x'$ . This allows operators like assignment, sequential composition, if-then-else, and iteration to be denoted as predicates over these variables, as illustrated in Table 12. From these denotations, algebraic laws of programming can be proved, such as those in Table 13, and more specialised semantic models developed for reasoning about programs, such as Hoare calculi and operational semantics. UTP also supports more sophisticated modelling constructs; for example concurrency is treated in [46, Chapter 7] via the *parallel-by-merge* construct  $P \parallel_M Q$ , a general scheme for parallel composition that creates two copies of the state space, executes  $P$  and  $Q$  in parallel on them, and then merges the results through the merge predicate  $M$ . This is then applied to UTP theory of communication in Chapter 8, and henceforth to give a UTP semantics to the process calculus CSP [18, 45].

Mechanisation of the UTP for the purpose of verification necessitates a model for the

<sup>6</sup>For supporting Isabelle theories, including mechanised proofs for all laws in this paper, see <http://cs.york.ac.uk/~simonf/ictac2016>

predicate and relational calculi [37, 72] that must satisfy laws such as those in Table 13. LP1 and LP2 are point-free laws, and can readily be derived from algebras like relation algebra or Kleene algebra [33]. The remaining laws, however, are point-wise in the sense that they rely on the predicate variables. Whilst law LP3 can be modelled with KAT [2] (Kleene Algebra with Tests) by considering  $b$  to be a test, the rest explicitly reference variables. LP4 and LP5 require that we support quantifiers and substitution. LP6 additionally requires we can specify free variables. Thus, to truly provide a generic algebraic foundation for the UTP, a more expressive model supporting these operators is needed.

### B.2.2 Isabelle/HOL

Isabelle/HOL [59] is a proof assistant for Higher Order Logic. It includes a functional specification language, a proof language for discharging specified goals in terms of proven theorems, and tactics that help automate proof. Its type system supports first-order parametric polymorphism, meaning types can carry variables – e.g.  $\alpha$  list for type variable  $\alpha$ . Built-in types include total functions  $\alpha \Rightarrow \beta$ , tuples  $\alpha \times \beta$ , booleans **bool**, and natural numbers **nat**. Isabelle also includes partial function maps  $\alpha \multimap \beta$ , which are represented as  $\alpha \Rightarrow \beta$  **option**, where  $\beta$  **option** can either take the value **Some** ( $v : \beta$ ) or **None**. Function  $\text{dom}(f)$  gives the domain of  $f$ ,  $f(k \mapsto v)$  updates a key  $k$  with value  $v$ , and function  $\text{the} : \alpha \text{ option} \Rightarrow \alpha$  extracts the valuation from a **Some** constructor, or returns an underdetermined value if **None** is present.

Record types can be created using **record**  $\mathcal{R} = f_1 : \tau_1 \cdots f_n : \tau_n$ , where  $f_i : \tau_i$  is a field. Each field  $f_i$  yields a query function  $f_i : \mathcal{R} \Rightarrow \tau_i$ , and update function  $f_i\text{-upd} : (\tau_i \Rightarrow \tau_i) \Rightarrow (\mathcal{R} \Rightarrow \mathcal{R})$  with which to transform  $\mathcal{R}$ . Moreover Isabelle provides simplification theorems for record instances  $\langle f_1 = v_1 \cdots f_n = v_n \rangle$ :

$$f_i \langle \cdots f_i = v \cdots \rangle = v \quad f_i\text{-upd } g \langle \cdots f_i = v \cdots \rangle = \langle \cdots f_i = g(v) \cdots \rangle$$

The HOL logic includes an equality relation  $=_\alpha : \alpha \Rightarrow \alpha \Rightarrow \text{bool}$  that equates values of the same type  $\alpha$ . In terms of tactics, Isabelle provides an equational simplifier **simp**, generalised deduction tactics **blast** and **auto**, and integration of external automated provers using the **sledgehammer** tool [10].

$$(P ; Q) ; R = P ; (Q ; R) \quad (\text{LP1})$$

$$P ; \text{false} = \text{false} ; P = \text{false} \quad (\text{LP2})$$

$$\text{while } b \text{ do } P = (P ; \text{while } b \text{ do } P) \triangleleft b \triangleright \text{II} \quad \text{if } \forall x \bullet x' \notin \text{fv}(b) \quad (\text{LP3})$$

$$P ; Q = \exists x_0 \bullet P[x_0/x] ; Q[x_0/x'] \quad (\text{LP4})$$

$$x := e ; P = P[e/x] \quad (\text{LP5})$$

$$(x := e ; y := f) = (y := f ; x := e) \quad \text{if } x \neq y, x \notin \text{fv}(f), y \notin \text{fv}(e) \quad (\text{LP6})$$

Table 13: Typical laws of programming



Our paper does not rely on detailed knowledge of Isabelle, as we present our definitions and theorems mathematically, though with an Isabelle feel. Technically, we make use of the **lifting** and **transfer** packages [48] that allow us to lift definitions and associated theorems from super-types to sub-types. We also make use of Isabelle’s **locale** mechanism to model algebraic hierarchies as in [33].

### B.2.3 Mechanised state spaces

Several mechanisations of the UTP in Isabelle exist [24, 25, 37, 72] that take a variety of approaches to modelling state; for a detailed survey see [72]. A general comparison of approaches to modelling state was made in [64] which identifies four models of state, namely state as functions, tuples, records, and abstract types, of which the first and third seem the most prevalent.

The first approach models state as a function  $\text{Var} \Rightarrow \text{Val}$ , for suitable value and variable types. This approach is taken by [62, 37, 23, 72], and requires a deep model of variables and values, in which concepts such as typing are first-class. This provides a highly expressive model with few limitations on possible manipulations [37]. However, [64] highlights two obstacles: (1) the machinery required for deep reasoning about program values is heavy and *a priori* limits possible constructions, and (2) explicit variable naming requires one to consider issues like  $\alpha$ -renaming. Whilst our previous work [72] effectively mitigates (1), at the expense of introducing axioms, the complexities associated with (2) remain. Nevertheless, the approach seems necessary to model dynamic creation of variables, as required, for example, in modelling memory heaps in separation logic [12, 23].

The alternative approach uses records to model state; a technique often used by verification tools in Isabelle [1, 24, 25, 2]. In particular, [24] uses this approach to create a shallow embedding of the UTP and library of laws<sup>7</sup> which, along with [62], our work is inspired by. A variable in this kind of model is abstractly represented by pairing the field-query and update functions,  $f_i$  and  $f_i\text{-upd}$ , yielding a nameless representation. As shown in [24, 25, 2], this approach greatly simplifies automation of program verification in comparison to the former functional approach through directly harnessing the polymorphic type system and automated proof tactics. However, the expense is a loss of flexibility compared to the functional approach, particularly in regards to decomposition of state spaces and handling of extension as required for local variables [64]. Moreover, those employing records seldom provide general support for meta-logical concepts like substitution, and do not abstractly characterise the behaviour of variables.

Our approach generalises all these models by abstractly characterising the behaviour of state and variables using lenses. Lenses were created as an abstraction for bidirectional programming and solving the view-update problem [30]. They abstract different views on a data space, and allow their manipulation independently of the context. A lens consists of two functions: **get** that extracts a view from a larger source, and **put** that puts back an updated view. [26] gives a detailed study of the algebraic lens laws for these functions. Combinators are also provided for composing lenses [31, 30]. They have been practically applied in the *Boomerang* language<sup>8</sup> for transformations on textual data structures.

<sup>7</sup>See archive of formal proofs: <https://www.isa-afp.org/entries/Circus.shtml>

<sup>8</sup>Boomerang home page: <http://www.seas.upenn.edu/~harmony/>



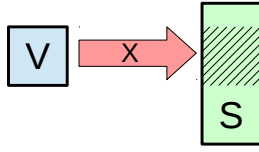


Figure 18: Visualisation of a simple lens

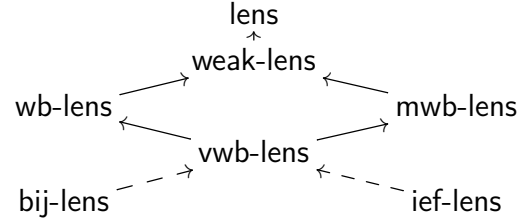


Figure 19: Lens algebraic hierarchy

Our lens approach is indeed related to the state-space solution in [64] of using Isabelle locales to characterise a state type abstractly and polymorphically. A difference though is the use of explicit names, where our lenses are nameless. Moreover, the core lens laws [26] bear a striking resemblance to Back’s variable laws [4], which he uses to form the basis for the meta-logical operators of substitution, freshness, and specification of procedures.

### B.3 Lenses

In this section, we introduce our lens algebra, which is later used in §B.4 to give a uniform interface for variables. The lens laws in §B.3.1 and composition operator of §B.3.3 are adapted from [30, 26], though the remaining operators, such as independence and sublens, are novel. All definitions and theorems have been mechanically validated<sup>6</sup>.

#### B.3.1 Lens laws

A lens  $X : V \Rightarrow S$ , for source type  $S$  and view type  $V$ , identifies  $V$  with a subregion of  $S$ , as illustrated in Figure 18. The arrow denotes  $X$  and the hatched area denotes the subregion  $V$  it characterises. Transformations on  $V$  can be performed without affecting the parts of  $S$  outside the hatched area. The lens signature consists of a pair of total functions<sup>9</sup>  $get_X : S \Rightarrow V$  that extracts a view from a source, and  $put_X : S \Rightarrow V \Rightarrow S$  that updates a view within a given source. When speaking about a particular lens we omit the subscript name. The behaviour of a lens is constrained by one or more of the following laws [26].

$$\begin{aligned}
 get(put\ s\ v) &= v && \text{(PutGet)} \\
 put(put\ s\ v')\ v &= put\ s\ v && \text{(PutPut)} \\
 put\ s\ (get\ s) &= s && \text{(GetPut)}
 \end{aligned}$$

PutGet states that if we update the view in  $s$  to  $v$ , then extracting the view yields  $v$ . PutPut states that if we make two updates, then the first update is overwritten. GetPut states that extracting the view and then putting it back yields the original source. These laws are often grouped into two classes [30]: *well-behaved lenses* that satisfy PutGet and GetPut, and *very well-behaved lenses* that additionally satisfy PutPut. We also identify *weak lenses* that satisfy only PutGet, and *mainly well-behaved lenses* that satisfy PutGet and PutPut but not GetPut. These weaker classes prove useful in certain contexts,

<sup>9</sup>Partial functions are sometimes used in the literature, e.g. [31]. We prefer total functions, as these circumvent undefinedness issues and are at the core of Isabelle/HOL.

notably in the map lens implementation (see §B.3.2). Moreover [26, 30] also identify the class of *bijective lenses* that satisfy PutGet and also the following law.

$$put\ s\ (get\ s') = s' \quad (\text{StrongGetPut})$$

StrongGetPut states that updating the view completely overwrites the state, and thus the source and view are, in some sense, equivalent. Finally we have the class of *ineffectual lenses* whose views do not effect the source. Our complete algebraic hierarchy of lenses is illustrated in Figure 19, where the arrows are implicative.

### B.3.2 Concrete lenses

We introduce lenses that exemplify the above laws and are applicable to modelling different kinds of state spaces. The function lens (fl) can represent total variable state functions  $\text{Var} \Rightarrow \text{Val}$  [37], whilst the map lens (ml) can represent heaps [23]. The record lens (rl) can represent static variables [25, 2].

**Definition B.1 (Function, Map, and Record lenses)**

$$\begin{array}{ll} get_{fl(k)} \triangleq \lambda f. f(k) & put_{fl(k)} \triangleq \lambda f\ v. f(k := v) \\ get_{ml(k)} \triangleq \lambda f. \text{the}(f(k)) & put_{ml(k)} \triangleq \lambda f\ v. f(k \mapsto v) \\ get_{rl(f_i)} \triangleq f_i & get_{rl(f_i)} \triangleq \lambda r\ v. f_i\text{-upd}(\lambda x. v)\ r \end{array}$$

The (total) function lens  $fl(k)$  focusses on a specific output associated with input  $k$ . The *get* function applies the function to  $k$ , and the *put* function updates the valuation of  $k$  to  $v$ . It is a very well-behaved lens:

**Theorem B.1 (The function lens is very well-behaved)**

**Proof B.1** *Included in our mechanised Isabelle theories<sup>6</sup>.*

The map lens  $ml(k)$  likewise focusses on the valuation associated with a given key  $k$ . If no value is present at  $k$  then *get* returns an arbitrary value. The map lens is therefore not a well-behaved lens since it does not satisfy GetPut, as  $f(k \mapsto \text{the}(f(k))) \neq f$  when  $k \notin \text{dom}(f)$  since the maps have different domains.

**Theorem B.2 (The map lens is mainly well-behaved)**

Finally, we consider the record lens  $rec(f_i)$ . As mentioned in §B.2.3, each record field yields a pair of functions  $f_i$  and  $f_i\text{-upd}$ , and associated simplifications for record instances. Together these can be used to prove the following theorem:

**Theorem B.3 (Record lens)** *Each  $f_i : \mathcal{R} \Rightarrow \tau_i$  yields a very well-behaved lens.*

This must be proved on a case-by-case basis for each field in each newly defined record; however the required proof obligations can be discharged automatically.

### B.3.3 Lens algebraic operators

Lens composition  $X \circ Y : V_1 \Rightarrow S$ , for  $X : V_1 \Rightarrow V_2$  and  $Y : V_2 \Rightarrow S$  allows one to focus on regions within larger regions. The intuition in Figure 20 shows how composition of  $X$

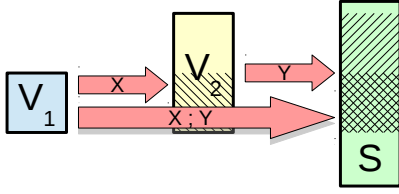


Figure 20: Lens composition visualised

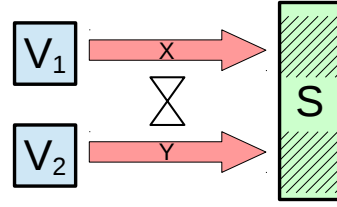


Figure 21: Lens independence visualised

and  $Y$  yields a lens that focuses on the  $V_1$  subregion of  $S$ . For example, if a record has a field which is itself a record, then lens composition allows one to focus on the inner fields by composing the lenses for the outer with those of the inner record. The definition is given below.

### Definition B.2 (Lens composition)

$$put_{X;Y} \triangleq \lambda s v. put_Y s (put_X (get_Y s) v) \quad get_{X;Y} \triangleq get_X \circ get_Y$$

The **put** operator of lens composition first extracts view  $V_2$  from source  $S$ , puts  $v : V_1$  into this, and finally puts the combined view. The **get** operator simply composes the respective **get** functions. Lens composition is closed under all lens classes ( $\{\text{weak}, \text{wb}, \text{mwb}, \text{vwb}\}$ -lens). We next define the unit lens,  $\mathbf{0} : \text{unit} \Rightarrow S$ , and identity lens,  $\mathbf{1} : S \Rightarrow S$ .

### Definition B.3 (Unit and identity lenses)

$$put_{\mathbf{0}} \triangleq \lambda s v. s \quad get_{\mathbf{0}} \triangleq \lambda s. () \quad put_{\mathbf{1}} \triangleq \lambda s v. v \quad get_{\mathbf{1}} \triangleq \lambda s. s$$

The unit lens view is the singleton type **unit**. Its **put** has no effect on the source, and **get** returns the single element  $()$ . It is thus an ineffectual lens. The identity lens identifies the view with the source, and it is thus a bijective lens. Lens composition and identity form a monoid. We now consider operators for comparing lenses which may have different view types, beginning with lens independence.

**Definition B.4 (Lens independence)** Lenses  $X : V_1 \Rightarrow S$  and  $Y : V_2 \Rightarrow S$  are independent, written  $X \bowtie Y$ , provided they satisfy the following laws:

$$put_X (put_Y s v) u = put_Y (put_X s u) v \quad (\text{LI1})$$

$$get_X (put_Y s v) = get_X s \quad (\text{LI2})$$

$$get_Y (put_X s u) = get_Y s \quad (\text{LI3})$$

Intuitively, two lenses are independent if they identify disjoint regions of the source as illustrated in Figure 21. We characterise this by requiring that the **put** functions of  $X$  and  $Y$  commute (LI1), and that the **put** functions of each lens has no effect on the result of the **get** function of the other (LI2, LI3). For example, independence of function lenses follows from inequality of the respective inputs, i.e.  $\text{fl}(k_1) \bowtie \text{fl}(k_2) \Leftrightarrow k_1 \neq k_2$ . Lens independence is a symmetric relation, and it is also irreflexive ( $\neg(X \bowtie X)$ ), unless  $X$  is ineffectual.

The second type of comparison between two lenses is containment.

**Definition B.5 (Sublens relation)** Lens  $X : V_1 \Rightarrow S$  is a sublens of  $Y : V_2 \Rightarrow S$ , written  $X \preceq Y$ , if the equation below is satisfied.

$$X \preceq Y \triangleq \exists Z : V_1 \Rightarrow V_2. Z \in \text{wb-lens} \wedge X = Z ; Y$$

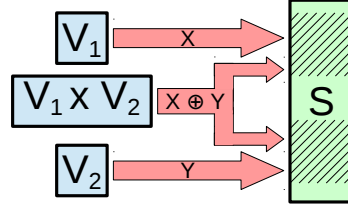


Figure 22: Lens sum visualised

The intuition of sublens is simply that the source region of  $X$  is contained within that of  $Y$ . The definition is explained by the following commuting diagram:

$$\begin{array}{ccc} & S & \\ x \nearrow & & \nwarrow y \\ V_1 & \xrightarrow{Z} & V_2 \end{array}$$

Intuitively,  $Z$  is a “shim” lens that identifies  $V_1$  with a subregion of  $V_2$ . Focusing on region  $V_1$  in  $V_2$ , followed by  $V_2$  in  $S$  is the same as focusing on  $V_1$  in  $S$ . The sublens relation is transitive and reflexive, and thus a preorder. Moreover  $\mathbf{0}$  is the least element ( $\mathbf{0} \preceq X$ ), and  $\mathbf{1}$  is the greatest element ( $X \preceq \mathbf{1}$ ), provided  $X$  is well-behaved. Sublens orders lenses by the proportion of the source captured. We have also proved the following theorem relating independence to sublens:

**Theorem B.4 (Sublens preserves independence)**

*If  $X \preceq Y$  and  $Y \bowtie Z$  then also  $X \bowtie Z$*

We use sublens to induce an equivalence relation  $X \approx Y \triangleq X \preceq Y \wedge Y \preceq X$ . It is a weaker notion than homogeneous HOL equality = between lenses as it allows the comparison of lenses with differently-typed views. We next prove two correspondences between bijective and ineffectual lenses.

**Theorem B.5 (Bijective and ineffectual lenses equality equivalence)**

$$X \in \text{ief-lens} \Leftrightarrow X \approx \mathbf{0} \quad X \in \text{bij-lens} \Leftrightarrow X \approx \mathbf{1}$$

The first law states that ineffectual lenses are equivalent to  $\mathbf{0}$ , and the second that bijective lenses are equivalent to  $\mathbf{1}$ . Showing that a lens is bijective thus entails demonstrating that it characterises the whole state space, though potentially with a different view type. We lastly describe lens summation.

**Definition B.6 (Lens sum)**

$$\text{put}_{X \oplus Y} \triangleq \lambda s (u, v). \text{put}_X (\text{put}_Y s v) u \quad \text{get}_{X \oplus Y} \triangleq \lambda s. (\text{get}_X s, \text{get}_Y s)$$

The intuition is given in Figure 22. Given independent lenses  $X : V_1 \Rightarrow S$  and  $Y : V_2 \Rightarrow S$ , their sum yields a lens  $V_1 \times V_2 \Rightarrow S$  that characterises both subregions. The combined *put* function executes the *put* functions sequentially, whilst the *get* extracts both values simultaneously. A notable application is to define when a source can be divided into two disjoint views  $X \bowtie Y$ , a situation we can describe with the formula  $X \oplus Y \approx \mathbf{1}$ , or equivalently  $X \oplus Y \in \text{bij-lens}$ , which can be applied to framing or division of a state space for parallel programs (see §B.6). Lens sum is closed under all lens classes. We also introduce two related lenses for viewing the left and right of a product source-type, respectively.

**Definition B.7 (First and second lenses)**

$$\begin{aligned}
put_{fst} &\triangleq (\lambda(s, t)u.(u, t)) & get_{fst} &\triangleq fst \\
put_{snd} &\triangleq (\lambda(s, t)u.(s, u)) & get_{snd} &\triangleq snd
\end{aligned}$$

We then prove the following lens sum laws:

**Theorem B.6 (Sum laws)** *Assuming  $X \bowtie Y$ ,  $X \bowtie Z$ , and  $Y \bowtie Z$ :*

$$\begin{aligned}
X \oplus Y &\approx Y \oplus X & X \oplus (Y \oplus Z) &\approx (X \oplus Y) \oplus Z \\
X \oplus \mathbf{0} &\approx X & (X \oplus Y) \mathbin{\circ} Z &= (X \mathbin{\circ} Z) \oplus (Y \mathbin{\circ} Z) \\
X &\preceq X \oplus Y & fst \oplus snd &= \mathbf{1} \\
X \oplus Y &\bowtie Z & \text{ if } X \bowtie Z \text{ and } Y \bowtie Z
\end{aligned}$$

Lens sum is commutative, associative, has  $\mathbf{0}$  as its identity, and distributes through lens composition. Naturally, each summand is a sublens of the whole, and it preserves independence as the next law demonstrates. The remaining law demonstrates that a product is fully viewed by its first and second component.

## B.4 Unifying state-space abstractions

In this section, we apply our lens theory to modelling state spaces in the context of the UTP's predicate calculus. We construct the core calculus (§B.4.1), meta-logical operators (§B.4.2), apply these to the relational laws of programming (§B.5), and finally give an algebraic basis to parallel-by-merge (§B.6). We also show that our model satisfies various important algebras, and thus justify its adequacy.

### B.4.1 Alphabetised predicate calculus

Our model of alphabetised predicates is  $\alpha \Rightarrow \text{bool}$ , where  $\alpha$  is a suitable type for modelling the alphabet, that corresponds to the state space. We do not constrain the structure of  $\alpha$ , but require that variables be modelled as lenses into it. For example, the record lens **rl** can represent a typed static alphabet [24, 25, 2], whilst the map lens **ml** can support dynamically allocated variables [23]. Moreover, lens composition can be used to combine different lens-based representations of state. We begin with the definition of types for expressions, predicates, and variables.

**Definition B.8 (UTP types)**

$$\begin{aligned}
(\tau, \alpha) \text{ uexpr} &\triangleq (\alpha \Rightarrow \tau) & \alpha \text{ upred} &\triangleq (\text{bool}, \alpha) \text{ uexpr} \\
(\alpha, \beta) \text{ urel} &\triangleq (\alpha \times \beta) \text{ upred} & (\tau, \alpha) \text{ uvar} &\triangleq (\tau \Rightarrow \alpha)
\end{aligned}$$

All types are parametric over alphabet type  $\alpha$ . An expression  $(\tau, \alpha) \text{ uexpr}$  is a query function mapping a state  $\alpha$  to a given value in  $\tau$ . A predicate  $\alpha \text{ upred}$  is a boolean-valued expression. A (heterogeneous) relation is a predicate whose alphabet is  $\alpha \times \beta$ . A variable  $x : (\tau, \alpha) \text{ uvar}$  is a lens that views a particular subregion of type  $\tau$  in  $\alpha$ , which affords a very general state model. We already have meta-logical functions for variables,

in the form of lens equivalence  $\approx$  and lens independence  $\bowtie$ . Moreover, we can construct variable sets using operators  $\mathbf{0}$  which corresponds to  $\emptyset$ ,  $\oplus$  which corresponds to  $\cup$ ,  $\mathbf{1}$  which corresponds to the whole alphabet, and  $\preceq$  that can model set membership  $x \in A$ . Theorem B.6 justifies these interpretations. We define several core expression constructs for literals, variables, and operators, from which most other operators can be built.

**Definition B.9 (UTP expression constructs)**

$$\begin{aligned}
\text{lit} : \tau &\Rightarrow \tau \text{ uexpr} & \text{var} : (\tau, \alpha) \text{ uvar} &\Rightarrow (\tau, \alpha) \text{ uexpr} \\
\text{lit } k &\triangleq \lambda s. k & \text{var } x &\triangleq \lambda s. \text{get}_x s \\
\text{uop} : (\tau &\Rightarrow \phi) &\Rightarrow (\tau, \alpha) \text{ uexpr} &\Rightarrow (\phi, \alpha) \text{ uexpr} \\
\text{uop } f \text{ } v &\triangleq \lambda s. f(v(s)) \\
\text{bop} : (\tau &\Rightarrow \phi \Rightarrow \psi) &\Rightarrow (\tau, \alpha) \text{ uexpr} &\Rightarrow (\phi, \alpha) \text{ uexpr} \Rightarrow (\psi, \alpha) \text{ uexpr} \\
\text{bop } f \text{ } u \text{ } v &\triangleq \lambda s. f(u(s))(v(s))
\end{aligned}$$

A literal **lit** lifts a HOL value to an expression via a constant  $\lambda$ -abstraction, so it yields the same value for any state. A variable expression **var** takes a lens and applies the *get* function on the state space  $s$ . Constructs **uop** and **bop** lift functions to unary and binary operators, respectively. These lifting operators enable a proof tactic for predicate calculus we call **pred-tac** [37] that uses the **transfer** package [48] to compile UTP expressions and predicates to HOL predicates, and afterwards apply **auto** or **sledgehammer** to discharge the resulting conjecture. Unless otherwise stated, all theorems below are proved in this manner.

The predicate calculus' boolean connectives and equality are obtained by lifting the corresponding HOL functions, leading to the following theorem:

**Theorem B.7 (Boolean Algebra)** *UTP predicates form a Boolean Algebra*

We define the refinement order on predicates  $P \sqsubseteq Q$ , as usual, as universally closed reverse implication  $[Q \Rightarrow P]$ , and use it to prove the following theorem.

**Theorem B.8 (Complete Lattice)** *UTP predicates form a Complete Lattice*

This provides suprema ( $\bigsqcup$ ), infima ( $\bigsqcap$ ), and fixed points  $(\mu, \nu)$  which allow us to express recursion. The bottom of the lattice is **true**, the most non-deterministic specification, and the top is **false**, the miraculous program. Next we define the existential and universal quantifiers using the lens operation *put*:

**Definition B.10 (Existential and universal quantifiers)**

$$\exists x \bullet P \triangleq (\lambda s. \exists v. P(\text{put}_x s v)) \quad \forall x \bullet P \triangleq (\lambda s. \forall v. P(\text{put}_x s v))$$

The quantifiers on the right-hand side are HOL quantifiers. Existential quantification  $(\exists x \bullet P)$  states that there is a valuation for  $x$  in state  $s$  such that  $P$  holds, specified using *put*. Universal quantification is defined similarly and satisfies  $(\forall x \bullet P) = (\neg \exists x \bullet \neg P)$ . We derive universal closure  $[P] \triangleq \forall \mathbf{1} \bullet P$ , that quantifies all variables in the alphabet ( $\mathbf{1}$ ). Alphabetised predicates then form a Cylindric Algebra [42], which axiomatises the quantifiers of first-order logic.

**Theorem B.9 (Cylindric Algebra)** *UTP predicates form a Cylindric Algebra; the following laws are satisfied for well-behaved lenses  $x$ ,  $y$ , and  $z$ :*

$$(\exists x \bullet \text{false}) \Leftrightarrow \text{false} \tag{C1}$$

$$P \Rightarrow (\exists x \bullet P) \quad (\text{C2})$$

$$(\exists x \bullet (P \wedge (\exists x \bullet Q))) \Leftrightarrow ((\exists x \bullet P) \wedge (\exists x \bullet Q)) \quad (\text{C3})$$

$$(\exists x \bullet \exists y \bullet P) \Leftrightarrow (\exists y \bullet \exists x \bullet P) \quad (\text{C4})$$

$$(x = x) \Leftrightarrow \mathbf{true} \quad (\text{C5})$$

$$(y = z) \Leftrightarrow (\exists x \bullet y = x \wedge x = z) \quad \text{if } x \bowtie y, x \bowtie z \quad (\text{C6})$$

$$\mathbf{false} \Leftrightarrow \left( \begin{array}{l} (\exists x \bullet x = y \wedge P) \wedge \\ (\exists x \bullet x = y \wedge \neg P) \end{array} \right) \quad \text{if } x \bowtie y \quad (\text{C7})$$

**Proof B.2** Most proofs are automatic, the one complexity being C4 which we have to split into cases for (1)  $x \bowtie y$ , when  $x$  and  $y$  are different, and (2)  $x \approx y$ , when they're the same. We thus implicitly assume that variables cannot overlap, though lenses can. C6 and C7 similarly require independence assumptions.

From this algebra, the usual laws of quantification can be derived [42], even for nameless variables. Since lenses can also represent variable sets, we can also model quantification over multiple variables such as  $\exists x, y, z \bullet P$ , which is represented as  $\exists x \oplus y \oplus z \bullet P$ , and then prove the following laws.

**Theorem B.10 (Existential quantifier laws)**

$$(\exists A \oplus B \bullet P) = (\exists A \bullet \exists B \bullet P) \quad (\text{Ex1})$$

$$(\exists B \bullet \exists A \bullet P) = (\exists A \bullet P) \quad \text{if } B \preceq A \quad (\text{Ex2})$$

$$(\exists x \bullet P) = (\exists y \bullet Q) \quad \text{if } x \approx y \quad (\text{Ex3})$$

Ex1 shows that quantifying over two disjoint sets or variables equates to quantification over both. Ex2 shows that quantification over a larger lens subsumes a smaller lens. Finally Ex3 shows that if we quantify over two lenses that identify the same subregion then those two quantifications are equal.

In addition to quantifiers for UTP variables we also provide quantifiers for HOL variables in UTP expressions,  $\exists x \bullet P$  and  $\forall x \bullet P$ , that bind  $x$  in a closed  $\lambda$ -term. These are needed to quantify logical meta-variables, which are often useful in proof. This completes the specification of the predicate calculus.

#### B.4.2 Meta-logical operators

We next move onto the meta-logical operators, first considering fresh variables, which we model by a weaker semantic property known as *unrestriction* [62, 37].

**Definition B.11 (Unrestriction)**

$$x \# P \Leftrightarrow (\forall s, v \bullet P(\text{put}_x s v) = P(s))$$

Intuitively, lens  $x$  is unrestricted in  $P$ , written  $x \# P$ , provided that  $P$ 's valuation does not depend on  $x$ . Specifically, the effect of  $P$  evaluated under state  $s$  is the same if we change the value of  $x$ . It is thus a sufficient notion to formalise the meta-logical provisos for the laws of programming. Unrestriction can alternatively be characterised as predicates whose satisfy the fixed point  $P = (\exists x \bullet P)$  for very well-behaved lens  $x$ . We now show some of the key unrestricted laws.



**Theorem B.11 (Unrestriction laws)**

$$\begin{array}{llll}
\text{U1 } \frac{-}{\mathbf{0} \# P} & \text{U2 } \frac{x \preceq y \quad y \# P}{x \# P} & \text{U3 } \frac{x \# P \quad y \# P \quad x \bowtie y}{(x \oplus y) \# P} \\
\text{U4 } \frac{-}{x \# \mathbf{true}} & \text{U5 } \frac{x \# P \quad x \# Q}{x \# P \wedge Q} & \text{U6 } \frac{x \# P \quad x \# Q}{x \# (P = Q)} & \text{U7 } \frac{x \# P}{x \# \neg P} \\
\text{U8 } \frac{x \bowtie y}{x \# y} & \text{U9 } \frac{x \in \mathbf{mwb-lens}}{x \# (\exists x \bullet P)} & \text{U10 } \frac{x \bowtie y \quad x \# P}{x \# (\exists y \bullet P)} & \text{U11 } \frac{-}{x \# [P]}
\end{array}$$

Laws U1–U3 correspond to unrestriction of multiple variables using the lens operations; for example U2 states that sublens preserves unrestriction. Laws U4–U7 show that unrestriction distributes through the logical connectives. Laws U8–U11 show the behaviour of unrestriction with respect to variables. U8 states that  $x$  is unrestricted in variable expression  $y$  if  $x$  and  $y$  are independent. U9 and U10 relate to unrestriction over quantifiers; the proviso  $x \in \mathbf{mwb-lens}$  means, for example, that a law is applicable to variables modelled by maps. Finally U11 states that all variables are unrestricted in a universal closure.

We next introduce substitution  $P[v/x]$ , which is also encoded semantically using homogeneous substitution functions  $\sigma : \alpha \Rightarrow \alpha$  over state space  $\alpha$ . We define functions for application, update, and querying of substitutions:

**Definition B.12 (Substitution functions)**

$$\begin{aligned}
\sigma \dagger P &\triangleq \lambda s. P(\sigma(s)) \\
\sigma(x \mapsto_s e) &\triangleq (\lambda s. \mathbf{put}_x(e(s)))(\sigma(s)) \\
\langle \sigma \rangle_s x &\triangleq (\lambda s. \mathbf{get}_x(\sigma(s)))
\end{aligned}$$

Substitution application  $\sigma \dagger P$  takes the state, applies  $\sigma$  to it, and evaluates  $P$  under this updated state. The simplest substitution,  $\mathbf{id} \triangleq \lambda x. x$ , effectively maps all variables to their present value. Substitution lookup  $\langle \sigma \rangle_s x$  extracts the expression associated with variable  $x$  from  $\sigma$ . Substitution update  $\sigma(x \mapsto_s e)$  assigns the expression  $e$  to variable  $x$  in  $\sigma$ . It evaluates  $e$  under the incoming state  $s$  and then puts the result into the state updated with the original substitution  $\sigma$  applied. We also introduce the short-hand  $[x_1 \mapsto_s e_1, \dots, x_n \mapsto_s e_n] = \mathbf{id}(x_1 \mapsto_s e_1, \dots, x_n \mapsto_s e_n)$ . A substitution  $P[e_1, \dots, e_n/x_1, \dots, x_n]$  of  $n$  expressions to corresponding variables is then expressed as  $[x_1 \mapsto_s e_1, \dots, x_n \mapsto_s e_n] \dagger P$ .

**Theorem B.12 (Substitution query laws)**

$$\langle \sigma(x \mapsto_s e) \rangle_s x = e \quad (\text{SQ1})$$

$$\langle \sigma(y \mapsto_s e) \rangle_s x = \langle \sigma \rangle_s x \quad \text{if } x \bowtie y \quad (\text{SQ2})$$

$$\sigma(x \mapsto_s e, y \mapsto_s f) = \sigma(y \mapsto_s f) \quad \text{if } x \preceq y \quad (\text{SQ3})$$

$$\sigma(x \mapsto_s e, y \mapsto_s f) = \sigma(y \mapsto_s f, x \mapsto_s e) \quad \text{if } x \bowtie y \quad (\text{SQ4})$$

SQ1 and SQ2 show how substitution lookup is evaluated. SQ3 shows that an assignment to a larger lens overrides a previous assignment to a small lens and SQ4 shows that independent lens assignments can commute. We next prove the laws of substitution application.



**Theorem B.13 (Substitution application laws)**

$$\begin{aligned}
\sigma \dagger x &= \langle \sigma \rangle_s x & \text{(SA1)} \\
\sigma(x \mapsto_s e) \dagger P &= \sigma \dagger e & \text{if } x \nmid P \text{ (SA2)} \\
\sigma \dagger \mathbf{uop} f v &= \mathbf{uop} f (\sigma \dagger v) & \text{(SA3)} \\
\sigma \dagger \mathbf{bop} f u v &= \mathbf{bop} f (\sigma \dagger u) (\sigma \dagger v) & \text{(SA4)} \\
(\exists y \bullet P)[e/x] &= (\exists y \bullet P[e/x]) & \text{if } x \bowtie y, y \nmid e \text{ (SA5)}
\end{aligned}$$

These laws effectively subsume the usual syntactic substitution laws, for an arbitrary number of variables, many of which simply show how substitution distributes through expression and predicate operators. SA2 shows that a substitution of an unrestricted variable has no effect. SA5 captures when a substitution can pass through a quantifier. The variables  $x$  and  $y$  must be independent, and furthermore the expression  $e$  must not mention  $y$  such that no variable capture can occur. Finally, we will use unrestricted and substitution to prove the one-point law of predicate calculus [41, §3.1].

**Theorem B.14 (One-point)**

$$(\exists x \bullet P \wedge x = e) = P[e/x] \quad \text{if } x \in \mathbf{mwb-lens}, x \nmid e$$

**Proof B.3** *By predicate calculus with **pred-tac**.*

The one-point law states that a quantification can be eliminated if precisely one value for the quantified variable is specified. We state the requirement “ $x$  does not appear in  $e$ ” with unrestricted. Thus we have now constructed a set of meta-logical operators and laws which can be applied to the laws of programming, all the while remaining within our algebraic lens framework and mechanised model. Indeed, all our operators are deeply encoded first-class entities in Isabelle/HOL.

**B.5 Relational laws of programming**

We now show how lenses can be applied to prove the common laws of programming within the relational calculus, by augmenting the alphabetised predicate calculus with relational variables and operators. Recall that a relation is simply a predicate over a product state:  $(\alpha \times \beta) \mathbf{upred}$ . Input and output variables can thus be specified as lenses that focus on the before and after state, respectively.

$$\textbf{Definition B.13 (Relational variables)} \quad \llbracket x \rrbracket = x \circ \mathbf{fst} \quad \llbracket x' \rrbracket = x \circ \mathbf{snd}$$

A variable  $x$  is lifted to an input variable  $x$  by composing it with **fst**, or to an output variable  $x'$  by composing it with **snd**. We can then proceed to define the operators of the relational calculus.

**Definition B.14 (Relational operators)**

$$\begin{aligned}
P ; Q &\triangleq \exists v \bullet P[v/\mathbf{1}'] \wedge Q[v/\mathbf{1}] & \mathbf{II} &\triangleq (\mathbf{1}' = \mathbf{1}) \\
P \triangleleft b \triangleright Q &\triangleq (b \wedge P) \vee (\neg b \wedge Q) & x := v &\triangleq \mathbf{II}[v/x]
\end{aligned}$$

The definition of sequential composition is similar to the standard UTP presentation [46], but we use  $\mathbf{1}$  and  $\mathbf{1}'$  to represent the input and output alphabets of  $Q$  and  $P$ , respectively. Skip ( $\mathbb{I}$ ) similarly uses  $\mathbf{1}$  to state that the before state is the same as the after state. We then combine  $\mathbb{I}$  with substitution to define the assignment operator. Note that because  $x$  is a lens, and  $v$  could be a product expression, this operator can be used to represent multiple assignments. We also describe the if-then-else conditional operator  $P \triangleleft b \triangleright Q$ . Sequential composition and skip, combined with the already defined predicate operators, provide us with the facilities for describing point-free while programs [2], which we illustrate by proving that alphabetised relations form a quantale.

**Theorem B.15 (Unital quantale)** *UTP relations form a unital quantale; that is they form a complete lattice and in addition satisfy the following laws:*

$$\begin{aligned} (P ; Q) ; R &= P ; (Q ; R) & P ; \mathbb{I} &= P = \mathbb{I} ; P \\ P ; \left( \bigsqcap_{Q \in \mathcal{Q}} Q \right) &= \bigsqcap_{Q \in \mathcal{Q}} (P ; Q) & \left( \bigsqcap_{P \in \mathcal{P}} P \right) ; Q &= \bigsqcap_{P \in \mathcal{P}} (P ; Q) \end{aligned}$$

This is proved in the context of Armstrong's Regular Algebra library [2], which also derives a proof that UTP relations form a Kleene algebra. This in turn allows definition of iteration using **while**  $b$  **do**  $P \triangleq (b \wedge P)^* \wedge (\neg b')$ , where  $b'$  denotes relational converse of  $b$ , and thence to prove the usual laws of loops. We next describe the laws of assignment.

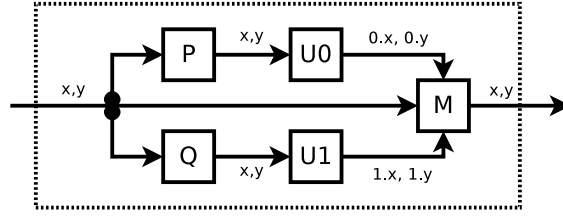
**Theorem B.16 (Assignment laws)**

$$\begin{aligned} x := e ; P &= P[e/x] & \text{(ASN1)} \\ x := e ; x := f &= x := f & \text{if } x \# f \text{ (ASN2)} \\ x := e ; y := f &= y := f ; x := e & \text{if } x \bowtie y, x \# f, y \# e \text{ (ASN3)} \\ x := e ; (P \triangleleft b \triangleright Q) &= (x := e ; P) \triangleleft b[e/x] \triangleright \\ &\quad (x := e ; Q) & \text{if } \mathbf{1}' \# b \text{ (ASN4)} \end{aligned}$$

We focus on ASN3 that demonstrates when assignments to  $x$  and  $y$  commute, and models law LP6 on page 55. Thus we have illustrated how lenses provide a general setting in which the laws of programming can be proved, including those that require meta-logical assumptions.

## B.6 Parallel-by-merge

We further illustrate the flexibility of our model by implementing one of the more complex UTP operators: parallel-by-merge. Parallel-by-merge is a general schema for parallel composition as described in [46, Chapter 7]. It enables the expression of sophisticated forms of parallelism that merge the output of two programs into a single consistent after state. It is illustrated in Figure 23 for two programs  $P$  and  $Q$  acting on variables  $x$  and  $y$ . The input values are fed into  $P$  and  $Q$ , and their output values are fed into predicates  $\mathbf{U0}$  and  $\mathbf{U1}$ . The latter two rename the variables so that the outputs from both programs can be distinguished by the merge predicate  $M$ .  $M$  takes as input the variable values

Figure 23: Pictorial representation of parallel-by-merge  $P \parallel_M Q$ 

before  $P$  and  $Q$  were executed, and the respective outputs. It then implements a specific mechanism for reconciling these outputs depending on the semantic model of the target language. For example, if  $P$  and  $Q$  both yield event traces as in CSP [45, 18], then only those traces that are consistent will be permitted.

Lenses can be used to define the merge predicate and post-state renamings  $U0$  and  $U1$ . The merge predicate takes as input three copies of the state: the outputs from  $P$  and  $Q$ , and the before state of the entire computation. Thus if the state has type  $A$  then  $M : ((A \times A) \times A, A) \text{ urel}$ , and similarly  $U0, U1 : (A, (A \times A) \times A) \text{ urel}$ . We thus give syntax to refer to indexed variables  $n.x$ , and prior variables  $<x$ , that give the input values, using the following lens compositions:

**Definition B.15 (Separated and prior variables)**

$$\llbracket 0.x \rrbracket = x \circ \text{fst} \circ \text{fst} \quad \llbracket 1.x \rrbracket = x \circ \text{snd} \circ \text{fst} \quad \llbracket <x \rrbracket = x \circ \text{snd}$$

Lenses  $0.x$  and  $1.x$  focus on the first and second elements of the tuple's first element, and  $<x$  focusses on the second element. We now define  $U0$  and  $U1$ :

**Definition B.16 (Separating simulations)**

$$U0 \triangleq 0.1' = 1 \wedge <1' = 1 \quad U1 \triangleq 1.1' = 1 \wedge <1' = 1$$

$U0$  and  $U1$  copy the before value of the whole state into both their respective indexed variables, and also the prior state. We can now describe parallel-by-merge, given a suitable basic parallel composition operator  $\parallel$  which could, for example, be plain conjunction or design parallel composition (see [46, Chapter 3]):

**Definition B.17 (Parallel-by-merge)**

$$P \parallel_M Q \triangleq ((P ; U0) \parallel (Q ; U1)) ; M$$

We also define predicate  $\text{swap}_m \triangleq 0.x, 1.x := 1.x, 0.x$  that swaps the left and right copies, and then prove the following generalised commutativity theorem:

**Theorem B.17 (Commutativity of parallel-by-merge)** *If  $M ; \text{swap}_m = M$  then  $P \parallel_M Q = Q \parallel_M P$ .*

This theorem states that if a merge predicate is symmetric, the resulting parallel composition is commutative. In the future we will also show the other properties of parallel composition [46], such as associativity and units. Nevertheless, we have shown that lenses enable a fully algebraic treatment of parallelism.

## B.7 Conclusions

We have presented an enriched theory of lenses, with algebraic operators and lens comparators, and shown how it can be applied to generically modelling the state space of programs in predicative semantic frameworks. We showed how lenses characterise variables, express meta-logical properties, and enrich and validate the laws of programming. The theory of lenses is general, and we believe it has many applications beyond program semantics, such as verifying bidirectional transformations [30]. We have also defined various other useful lens operations, such as lens quotient which is dual to composition. Space has not allowed us to cover this, but we claim this is useful for expressing the contraction of state spaces. Further study of the algebraic properties of these operators is in progress.

Overall, lenses have proven to be a useful abstraction for reasoning about state, in terms of properties like independence and combination. We have used our model to prove several hundred laws of predicate and relational calculus from the UTP book [46] and other sources [41, 18, 63]. We have also mechanised the Hoare calculus and a weakest precondition calculus that support practical program verification. Although details were omitted for brevity, lenses enable definition of operators like alphabet extension and restriction, through the description of alphabet coercion lenses that are used to represent local variables and methods. We are currently exploring links with Back's variable calculus [4].

In future work we will to apply lenses to additional theories of programming, such as hybrid systems [34] and separation logic [69], especially since our lens algebra resembles a separation algebra. Moreover, we will use our UTP theorem prover<sup>10</sup> to apply our database of programming laws to build practical verification tools for a variety of semantically rich languages [63], in particular for the purpose of analysing heterogeneous Cyber-Physical Systems [34]. We also plan to integrate our work with the existing *Isabelle/Circus* [25] library<sup>7</sup> to further improve verification support for concurrent and reactive systems.

---

<sup>10</sup>See our repository at [github.com/isabelle-utp/utp-main/tree/shallow.2016](https://github.com/isabelle-utp/utp-main/tree/shallow.2016)

## References

- [1] E. Alkassar, M. Hillebrand, D. Leinenbach, N. Schirmer, and A. Starostin. The Verisoft approach to systems verification. In *VSTTE 2008*, volume 5295 of *LNCS*, pages 209–224. Springer, 2008.
- [2] A. Armstrong, V. Gomes, and G. Struth. Building program construction and verification tools from algebraic principles. *Formal Aspects of Computing*, 28(2):265–293, 2015.
- [3] A. Armstrong, G. Struth, and T. Weber. Program analysis and verification based on Kleene Algebra in Isabelle/HOL. In *ITP*, volume 7998 of *LNCS*. Springer, 2013.
- [4] R.-J. Back and V. Preoteasa. Reasoning about recursive procedures with parameters. In *Proc. Workshop on Mechanized Reasoning About Languages with Variable Binding*, MERLIN '03, pages 1–7. ACM, 2003.
- [5] Victor Bandur, Peter Gorm Larsen, Kenneth Lausdahl, Sune Wolff, Carl Gamble, Adrian Pop, Etienne Brosse, Jörg Brauer, Florian Lapschies, Marcel Groothuis, and Christian Kleijn. User Manual for the INTO-CPS Tool Chain. Technical report, INTO-CPS Deliverable, D4.1a, December 2015.
- [6] A. Beg and A. Butterfield. Development of a prototype translator from Circus to CSPm. In *Proc. 9th Intl. Conf. on Open Source Systems and Technologies (ICOSST)*, pages 16–23. IEEE, December 2015.
- [7] M. M. A. Beg. *Translating from “State-Rich” to “State-Poor” Process Algebras*. PhD thesis, Department of Computer Science, Trinity College Dublin, April 2016.
- [8] Juan Bicarregui, John Fitzgerald, Peter Lindsay, Richard Moore, and Brian Ritchie. *Proof in VDM: A Practitioner’s Guide*. FACIT. Springer-Verlag, 1994. ISBN 3-540-19813-X.
- [9] D. Bjørner and C.B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *Lecture Notes in Computer Science*. Springer-Verlag, 1978.
- [10] J. C. Blanchette, L. Bulwahn, and T. Nipkow. Automatic proof and disproof in Isabelle/HOL. In *FroCoS*, volume 6989 of *LNCS*, pages 12–27. Springer, 2011.
- [11] Jeremy Bryans, Samuel Canham, and Jim Woodcock. CML definition 4. Technical report, COMPASS Deliverable, D23.5, March 2014. Available at <http://www.compass-research.eu/>.
- [12] C. Calcagno, P. O’Hearn, and H. Yang. Local action and abstract separation logic. In *LICS*, pages 366–378. IEEE, July 2007.
- [13] Samuel Canham and Jim Woodcock. Three approaches to timed external choice in UTP. In *Unifying Theories of Programming*, volume 8963, pages 1–20. Springer, 2015.
- [14] A. Cavalcanti, A. Mota, and J. Woodcock. Simulink timed models for program verification. In *Theories of Programming and Formal Methods*, volume 8051 of *LNCS*, pages 82–99. Springer, 2013.

- [15] A. Cavalcanti, J. Woodcock, and N. Amalio. Behavioural models for FMI cosimulations. In *Proc. 13th Intl. Conf. on Theoretical Aspects of Computing (ICTAC)*, volume 9965 of *LNCS*. Springer, 2016.
- [16] Ana Cavalcanti, Simon Foster, Bernhard Thiele, and Jim Woodcock. Initial version of the semantics for continuous-time modelling. Technical report, INTO-CPS Deliverable, D2.2c, December 2016.
- [17] Ana Cavalcanti, Andy Wellings, and Jim Woodcock. The safety-critical java memory model formalised. *Formal Aspects of Computing*, 25(1):37–57, 2013.
- [18] Ana Cavalcanti and Jim Woodcock. A Tutorial Introduction to CSP in Unifying Theories of Programming. In Ana Cavalcanti, Augusto Sampaio, and Jim Woodcock, editors, *Refinement Techniques in Software Engineering*, volume 3167 of *Lecture Notes in Computer Science*, pages 220–268. Springer Berlin / Heidelberg, 2006.
- [19] Ana Cavalcanti and Jim Woodcock. Initial semantics for FMI. Technical report, INTO-CPS Deliverable, D2.2d, December 2016.
- [20] Ana Cavalcanti, Frank Zeyda, Andy Wellings, Jim Woodcock, and Kun Wei. Safety-critical java programs Circus models. *Real-Time Systems*, 49(5):614–667, 2013.
- [21] Joey W. Coleman, Anders Kaels Malmos, Luis Diogo Couto, Peter Gorm Larsen, Richard Payne, Simon Foster, Uwe Schulze, and Adalberto Cajueiro. Fourth release of the COMPASS tool — symphony ide user manual. Technical Report D31.4a, COMPASS Deliverable, D31.4a, September 2014.
- [22] Joey W. Coleman, Anders Kaels Malmos, Peter Gorm Larsen, Jan Peleska, Ralph Hains, Zoe Andrews, Richard Payne, Simon Foster, Alvaro Miyazawa, Cristiano Bertolini, and André Didier. COMPASS Tool Vision for a System of Systems Collaborative Development Environment. In *Proceedings of the 7th International Conference on System of System Engineering, IEEE SoSE 2012*, pages 451–456, July 2012.
- [23] B. Dongol, V. Gomes, and G. Struth. A program construction and verification tool for separation logic. In *MPC 2015*, volume 9129 of *LNCS*, pages 137–158. Springer, 2015.
- [24] A. Feliachi, M.-C. Gaudel, and B. Wolff. Unifying theories in Isabelle/HOL. In *UTP 2010*, volume 6445 of *LNCS*, pages 188–206. Springer, 2010.
- [25] A. Feliachi, M.-C. Gaudel, and B. Wolff. Isabelle/Circus: a process specification and verification environment. In *VSTTE 2012*, volume 7152 of *LNCS*, pages 243–260. Springer, 2012.
- [26] S. Fischer, Z. Hu, and H. Pacheco. A clear picture of lens laws. In *MPC 2015*, pages 215–223. Springer, 2015.
- [27] John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat, and Marcel Verhoef. *Validated Designs for Object-oriented Systems*. Springer, New York, 2005.
- [28] John Fitzgerald, Peter Gorm Larsen, and Marcel Verhoef, editors. *Collaborative Design for Embedded Systems – Co-modelling and Co-simulation*. Springer, 2014.



- [29] John Fitzgerald, Peter Gorm Larsen, and Jim Woodcock. Foundations for Model-based Engineering of Systems of Systems. In M. Aiguier et al., editor, *Complex Systems Design and Management*, pages 1–19. Springer, January 2014.
- [30] J. Foster. *Bidirectional programming languages*. PhD thesis, University of Pennsylvania, 2009.
- [31] J. Foster, M. Greenwald, J. Moore, B. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.*, 29(3), May 2007.
- [32] S. Foster, A. Miyazawa, J. Woodcock, A. Cavalcanti, J. Fitzgerald, and P. Larsen. An approach for managing semantic heterogeneity in systems of systems engineering. In *Proc. 9th Intl. Conf. on Systems of Systems Engineering*. IEEE, 2014.
- [33] S. Foster, G. Struth, and T. Weber. Automated engineering of relational and algebraic methods in Isabelle/HOL. In *RAMICS 2011*, volume 6663 of *LNCS*, pages 52–67. Springer, 2011.
- [34] 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*, June 2016. To appear.
- [35] S. Foster and J. Woodcock. Unifying theories of programming in Isabelle. In *IC-TAC 2013 School on Software Engineering*, volume 8050 of *LNCS*, pages 109–155. Springer, 2013.
- [36] S. Foster, F. Zeyda, and J. Woodcock. Isabelle/UTP: A mechanised theory engineering framework. In David Naumann, editor, *Proc. 5th Intl. Symposium on Unifying Theories of Programming (UTP 2014)*, volume 8963 of *LNCS*, pages 21–41. Springer, 2014.
- [37] S. Foster, F. Zeyda, and J. Woodcock. Isabelle/UTP: A mechanised theory engineering framework. In *Proc. 5th Intl. Symp. on Unifying Theories of Programming*, volume 8963 of *LNCS*, pages 21–41. Springer, 2014.
- [38] 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.
- [39] Simon Foster, Ana Cavalcanti, Kenneth Lausdahl, Ken Pierce, and Jim Woodcock. Initial Semantics of VDM-RT. Technical report, INTO-CPS Deliverable, D2.1b, December 2015.
- [40] Thomas Gibson-Robinson, Philip Armstrong, Alexandre Boulgakov, and A.W. Roscoe. FDR3 — A Modern Refinement Checker for CSP. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 8413 of *Lecture Notes in Computer Science*, pages 187–201, 2014.
- [41] E. C. R. Hehner. *A Practical Theory of Programming*. Springer, 1993.
- [42] L. Henkin, J. Monk, and A. Tarski. *Cylindric Algebras, Part I*. North-Holland, 1971.
- [43] M. Hennessy and T. Regan. A process algebra for timed systems. *Information and Computation*, 117(2):221–239, March 1995.

- [44] C.A.R. Hoare, I.J. Hayes, He Jifeng, C.C. Morgan, A.W. Roscoe, J.W. Sanders, I.H. Sørensen, J.M. Spivey, and B.A. Sufrin. The laws of programming. *Communications of the ACM*, 30(8):672–687, August 1987. see Corrigenda in *Communications of the ACM*, 30(9): 770.
- [45] Tony Hoare. *Communicating Sequential Processes*. Prentice-Hall International, Englewood Cliffs, New Jersey 07632, 1985.
- [46] Tony Hoare and Jifeng He. *Unifying Theories of Programming*. Prentice-Hall, 1998.
- [47] J. Hooman and M. Verhoef. Formal semantics of a VDM extension for distributed embedded systems. In D. Dams, U. Hannemann, and M. Steffen, editors, *Concurrency, Compositionality, and Correctness, Essays in Honor of Willem-Paul de Roever*, volume 5930 of *Lecture Notes in Computer Science*, pages 142–161. Springer-Verlag, 2010.
- [48] B. Huffman and O. Kunčar. Lifting and transfer: A modular design for quotients in Isabelle/HOL. In *CPP*, volume 8307 of *LNCS*, pages 131–146. Springer, 2013.
- [49] Information technology – Programming languages, their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language, December 1996.
- [50] C. B. Jones. The early search for tractable ways of reasoning about programs. *IEEE, Annals of the History of Computing*, 25(2):26–49, 2003.
- [51] G. Klein et al. seL4: Formal verification of an OS kernel. In *Proc. 22nd Symp. on Operating Systems Principles (SOSP)*, pages 207–220. ACM, 2009.
- [52] Peter Gorm Larsen, Kenneth Lausdahl, Nick Battle, John Fitzgerald, Sune Wolff, Shin Sahara, Marcel Verhoef, Peter W. V. Tran-Jørgensen, and Tomohiro Oda. VDM-10 Language Manual. Technical Report TR-001, The Overture Initiative, [www.overturetool.org](http://www.overturetool.org), April 2013.
- [53] Peter Gorm Larsen and Wiesław Pawłowski. The Formal Semantics of ISO VDM-SL. *Computer Standards and Interfaces*, 17(5–6):585–602, September 1995.
- [54] Kenneth Lausdahl, Joey W. Coleman, and Peter Gorm Larsen. Semantics of the VDM Real-Time Dialect. Technical Report ECE-TR-13, Aarhus University, April 2013.
- [55] B. H. Liskov and J. M. Wing. A behavioural notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.
- [56] A. Miyazawa, L. Lima, and A. Cavalcanti. Formal models of sysml blocks. In *15th Intl. Conf. on Formal Engineering Methods (ICFEM)*, volume 8144 of *LNCS*, pages 249–264. Springer, 2013.
- [57] Carroll Morgan. *Programming from Specifications*. Prentice-Hall, London, UK, 1990.
- [58] Gerard Ekembe Ngondi and Jim Woodcock. UTP semantics of reactive processes with continuations. In *6th Intl. Symp. on Unifying Theories of Programming (UTP)*, LNCS. Springer, 2016.



- [59] T. Nipkow, M. Wenzel, and L. C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
- [60] 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.
- [61] M. V. M Oliveira, A. C. A. Sampaio, and M. S. C. Filho. Model-checking Circus state-rich specifications. In *11th Intl. Conf. on Integrated Formal Methods*, volume 8739 of *LNCS*, pages 39–54. Springer, 2014.
- [62] Marcel Oliveira, Ana Cavalcanti, and Jim Woodcock. Unifying theories in ProofPower-Z. In *Unifying Theories of Programming*, volume 4010 of *LNCS*, pages 123–140. Springer, 2007.
- [63] Marcel Oliveira, Ana Cavalcanti, and Jim Woodcock. A UTP semantics for Circus. *Formal Aspects of Computing*, 21:3–32, February 2009.
- [64] N. Schirmer and M. Wenzel. State spaces – the locale way. In *SSV 2009*, volume 254 of *ENTCS*, pages 161–179, 2009.
- [65] Xinbei Tang and Jim Woodcock. Travelling processes. In Dexter Kozen, editor, *7th Intl. Conf. on Mathematics of Program Construction (MPC)*, volume 3125 of *Springer*, pages 381–399. Springer, 2004.
- [66] Marcel Verhoef, Peter Gorm Larsen, and Jozef Hooman. Modeling and Validating Distributed Embedded Real-Time Systems with VDM++. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods*, Lecture Notes in Computer Science 4085, pages 147–162. Springer-Verlag, 2006.
- [67] Kun Wei. Reactive designs of interrupts in Circus Time. In *10th Intl. Colloquium on Theoretical Aspects of Computing (ICTAC)*, volume 8049 of *LNCS*, pages 373–390. Springer, 2013.
- [68] 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.
- [69] J. Woodcock, S. Foster, and A. Butterfield. Heterogeneous semantics and unifying theories. In *7th Intl. Symp. on Leveraging Applications of Formal Methods, Verification, and Validation (ISoLA)*, 2016. To appear.
- [70] J. C. P. Woodcock and A. L. C. Cavalcanti. A Tutorial Introduction to Designs in Unifying Theories of Programming. In E. A. Boiten, J. Derrick, and G. Smith, editors, *IFM 2004: Integrated Formal Methods*, volume 2999 of *Lecture Notes in Computer Science*, pages 40 – 66. Springer-Verlag, 2004. Invited tutorial.
- [71] Jim Woodcock. Engineering UToPiA - Formal Semantics for CML. In Cliff Jones, Pekka Pihlajasaari, and Jun Sun, editors, *FM 2014: Formal Methods*, volume 8442 of *Lecture Notes in Computer Science*, pages 22–41. Springer International Publishing, 2014.
- [72] F. Zeyda, S. Foster, and L. Freitas. An axiomatic value model for Isabelle/UTP. In *Proc. 6th Intl. Symp. on Unifying Theories of Programming*, 2016. To appear.