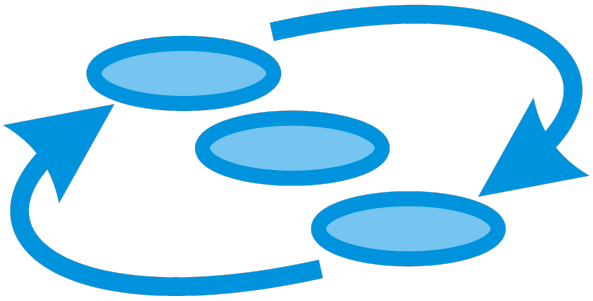
Grant Agreement: 644047

INtegrated TOol chain for model-based design of CPSs



INTO-CPS

Initial Semantics of VDM-RT

Technical Note Number: D2.1b

Version: 1.0

Date: December 2015

Public Document

http://into-cps.au.dk

Contributors:

Simon Foster, UY
Ana Cavalcanti, UY
Ken Pierce, UNEW
Kenneth Lausdahl, AU
Jim Woodcock, UY

Editors:

Simon Foster, UY

Reviewers:

Stylianos Basagiannis, UTRC
John Fitzgerald, UNEW
Bernhard Thiele, LIU

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	18-05-2015	Simon Foster	Initial document version
0.2	19-08-2015	Simon Foster	Slight changes to structure, added section of preprocessing
0.5	01-11-2015	Ken Pierce	Added introduction and VDM-RT section
0.6	14-11-2015	Simon Foster	Completed full draft for internal review
0.7	09-12-2015	Simon Foster	Completed integrating the internal review comments
1.0	14-12-2015	Ken Pierce	Integrated John Fitzgerald's review comments

Abstract

The deliverable reports on our work towards the creation of a novel formal semantics for the real-time modelling language, VDM-RT. The initial focus of this report is on the object-oriented aspects of the language, with other aspects being dealt with by subsequent deliverables. We provide a denotational semantics for the object-oriented parts of the language in a framework called Universal Theories of Programming (UTP), with particular emphasis on tackling multiple inheritance, in the form of an extended UTP theory of classes and the definition of semantic functions. We also provide preliminary experimental mechanisation in Isabelle showing how classes can be encoded, and some insight into future work in this area.

Contents

Glossary

Circus	a formal modelling language for state-rich concurrent systems building on CSP and with a UTP semantics.
CircusTime	*Circus* extended with primitives for real-time modelling.
CML	COMPASS Modelling Language, a formal language for modelling systems of systems based on *Circus* and VDM.
CSP	Communicating Sequential Processes, a process calculus created by Tony Hoare.
FMI	Functional Mockup Interface, a language for describing the composition of heterogeneous system models.
HOL	Higher Order Logic.
Isabelle	a generic proof-assistant usually associated with HOL.
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.

Symbols

$A \leftrightarrow B$	relation between A and B.
$P \,;\, Q$	sequential composition of P and Q.
$P \vdash Q$	design turnstile, with assumption P and commitment Q.
$R(\!\mid A \mid\!)$	the image of relation R under set A.
$R \oplus S$	override the relation R with S.
R^*	reflexive transitive closure of relation R.
R^+	transitive closure of relation R.
$X \mathrel{\widehat=} A$	define the name X to be A.
II	relation identity (skip).
Object	a distinguished root class of the inheritance tree.
Σ	the top-level alphabet of all symbols.
$\bigsqcap A$	non-deterministic choice of an element in A.

$\mathrm{dom}(R)$	the domain of relation R.
ϵ	Hilbert's choice operator.
iseq A	a sequence of unique elements drawn from A (injective).
$\mathbb{P}A$	power set of A.
$\prec$	strict subclass relation.
$\preceq$	subclass relation.
$\mathrm{ran}(R)$	the range of relation R.
$k \mapsto v$	maplet from k to v.

1 Introduction

The INTO-CPS project is integrating seven baseline technologies into a tool chain. This involves either transforming data from one tool and format to another, or passing data between models during co-simulation. The foundations work is providing a formal semantics for this transfer, using a common framework, UTP (Unifying Theories of Programming). This deliverable reports on semantics work covering one baseline technology, VDM-RT.

VDM-RT is a real-time dialect of the VDM formal modelling language that can be applied to the specification of discrete controllers for Cyber-Physical Systems (CPSs). VDM-RT is object-oriented, where all models are defined as classes that are instantiated as objects. It supports concurrency through threading and communication between threads through shared objects. The real-time features of the language comprise abstractions for deployment of objects to compute units, which are connected by buses, and the time taken to evaluate expressions, which advance a global "wall clock" to predict the computation time of a model.

A denotational semantics exists for the core specification language [27], and a structured operational semantics (SOS) exists for the real-time aspects [28], but there is currently no full semantic description of VDM-RT. To address this, this deliverable is the first in a series of three that looks at giving a semantics to the VDM-RT language. This first version looks primarily at object-orientation. Future versions will look at real-time and concurrency.

The structure of the deliverable is as follows. Section 2 provides essential background for this deliverable, covering VDM-RT first, then looking at Isabelle/HOL, UTP and their combination as Isabelle/UTP. Section 3 gives our overall approach to giving a UTP semantics to VDM-RT, and puts the remaining sections in context. Section 4 describes our extended calculus of class and object that is used to give an account of VDM-RT classes. Section 5 describes the UTP semantics of VDM-RT classes and objects using the class calculus described in the previous section. Section 6 gives an outline of our work so far on mechanising these semantics. Finally, Section 7 concludes the deliverable and shows the way forward. Appendix A describes an initial mechanisation of Lausdahl's [28] operational semantics of VDM-RT in Isabelle/HOL.

2 Background

2.1 VDM-RT

The Vienna Development Method (VDM) is a state-based formal method that was originally designed in the 1970s to give semantics to programming languages [26]. 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 [24]. As part of the standardisation process, a full denotational semantics has been defined for VDM-SL (due to Larsen et. al [27]), as well as a proof theory and comprehensive set of proof rules [3].

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++ [11] was defined adding

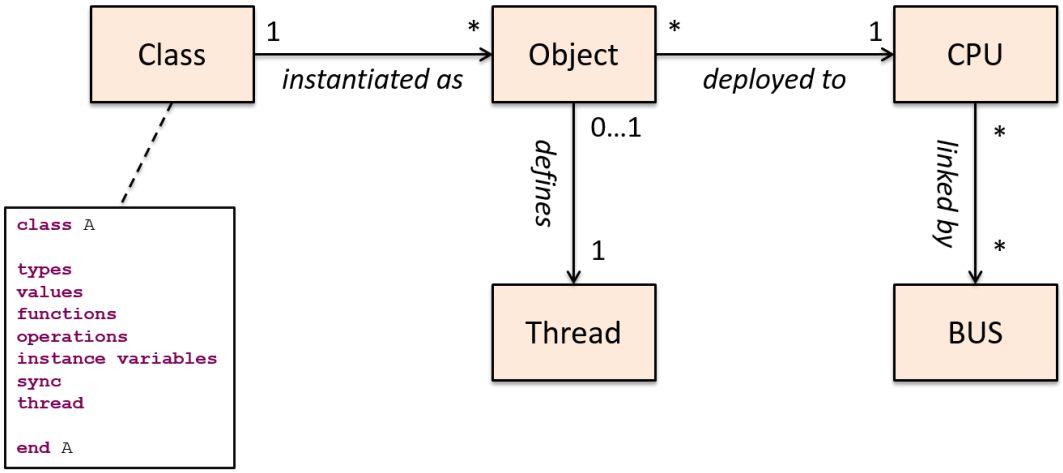


Figure 1: Relationship of VDM-RT concepts

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 [39]. 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[1] and the open source Overture[2]. Overture is used in INTO-CPS and also forms part of the Crescendo baseline technology, which allows co-simulation between VDM-RT and 20-sim models [12].

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. Each class may define a thread, and once an object of that class is created, its thread can be started. A thread without a loop will terminate once its work is finished. There is a shorthand notation for defining a thread that will call an operation periodically.

Threads communicate via shared objects. Synchronization on shared objects is specified using *permission predicates*. A permission predicate comprises an operation name and a predicate over the state of the object. If the predicate evaluates to false, calls to the operation are blocked until such time as the predicate evaluates to true. Permission predicates can also use *history counters* that yield the number of times an operation has been requested, activated or completed. A shorthand notation for mutual exclusion of operations is also provided.

The real-time features of VDM-RT are based around a global "wall clock" that records the time, in nanoseconds, since the simulation began. All expressions in a model advance the clock. 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 speed. A class diagram relating the key elements introduced here is given in Figure 1.

The amount of time each expression takes to evaluate (i.e. the amount of time the wall clock

[1] http://www.vdmtools.jp/en/
[2] http://www.overturetool.org/

is updated by) 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 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 of real hardware in order make the timing predictions of the model more accurate.

Semantics work has not kept pace with the changes introduced by the VDM++ and VDM-RT dialects. A structured operational semantics (SOS) is given for the real-time aspects of VDM-RT using a simplified language by Verhoef [38] (summarised in [22]). A more comprehensive version is given by Lausdahl et al. [28]. A mechanization of this semantics in Isabelle is reported in Appendix A. There is however currently no full semantics covering all aspects of VDM-RT. In a sense the semantics is best represented at present by the interpreter of the Overture tool. One of the developers has identified a number of ambiguities that a semantics should resolve [2]:

- Initialization of static instance variables
- Initialization order of instance variables
- Calling multiple explicit superclass constructors
- Multiple inheritance superclass initialization
- Implicit calls to default constructors
- Overridden vs local operations in super constructors
- Invariant checking during construction
- Are constructors inheritable?
- Overriding/overloading polymorphic / curried functions
- Pre-post conditions in OO state context
- Diamond inheritance

As an open source initiative, both the Overture tool and the language definitions are looked after by community members. A group of seven elected members form the Language Board (LB), oversee requests for change to the language. We have engaged the LB as part of the semantics work and will hold regular netmeetings to report progress and discuss issues as they arise. This will ensure that the semantics work will have a wide benefit to both the project and the community at large.

2.2 Isabelle/HOL

Isabelle/HOL [33] is a proof assistant for Higher-Order Logic (HOL). A proof assistant enables a theory engineer to develop theories, consisting of definitions and laws, and prove theorems about those theories. Isabelle/HOL consists of an ML-style functional programming language and a proof system to assert and prove properties of defined constructs. Functional

programming concepts include functions, algebraic datatypes, and records. A simple function for squaring an integer can be defined in Isabelle like so:

definition *square* :: *int* ⇒ *int* **where**
square n = *n* ∗ *n*

This **definition** command creates a new constant called *square*, whose type is *int* ⇒ *int*, that is a total function from integers to integers. The body of the function is declared by the equation below the type declaration follow **where**.

Mechanical proofs can be entered as a sequence of commands (`apply`-style) or using the ISAR [41] structured proof language, which supports readable proofs. Isabelle proofs are constructed by the application of tactics that can be used to variously decompose a proof goal into a collection of simpler ones, until all goals have been discharged. Proofs are correct by construction with respect to a small core of axioms as part of the *LCF architecture* that ensures relative soundness of the system. This follows since theories only introduce constants through definition, and not through the addition of further axioms, though the latter is possible (with suitable justification) if a more powerful logic than HOL is required. Nevertheless, if the definitional approach is followed, no inconsistencies can be introduced by user theories. Though a consistency proof for the HOL axioms is impossible (*à la* Gödel), they have been given a semantics in ZFC set theory [23] and therefore HOL is as consistent as one of the main foundations of mathematics.

Along with elementary deduction tactics, like backward chaining, Isabelle has a large number of automated proof tactics. This includes the equational simplifier simp, the classical reasoner blast, and the combination tactic auto. Tactics can be augmented with additional rules by placing them in appropriate theorem sets. For instance, the set simp contains simplification laws and intro contains introduction laws. Isabelle also includes a number of high-level proof tools [4] such as sledgehammer, a principled integration of third-party automated theorem provers, and nitpick, a counterexample generator. We believe this combination sets Isabelle apart as an ideal platform on which to mechanise semantics. For more information in Isabelle/HOL, please see the excellent document provided on their website[3].

In Section 6 we will use Isabelle to create a prototype mechanisation of some of the core constructs of our object-oriented language. This is engineered in the context of *Isabelle/UTP*, our UTP implementation that we describe in Section 2.4.

2.3 Unifying Theories of Programming

Unifying Theories of Programming [21] (UTP) is a mathematical framework for describing and unifying the formal semantics of programming and modelling languages. It has previously been applied to creation of semantic models for a variety of languages, including Safety-Critical Java [7], SysML [31], Simulink [8], and CML [44] (a formal modelling language for Systems of Systems). During these developments a large library of theories of programming has been built up, and we will be able to make use of these in our semantics for VDM-RT. Moreover, UTP will enable 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.

[3] http://isabelle.in.tum.de/documentation.html

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

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

$$(P \lhd b \rhd Q)\ ;\ R = (P\ ;\ R) \lhd b \rhd (Q\ ;\ R) \quad (3)$$

$$\textbf{\textit{while}}\ b\ \textbf{\textit{do}}\ P = (P\ ;\ \textbf{\textit{while}}\ b\ \textbf{\textit{do}}\ P) \lhd b \rhd \mathit{II} \quad (4)$$

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

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

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

Table 3: UTP Algebraic Laws of Predicative Programming

Programs in the UTP are given denotational semantics using *alphabetised predicates* (P) that define the relationship 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 $[Q \Rightarrow P]$. A detailed tutorial on the UTP alphabetised predicate calculus is available [15], and so we will 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 $P \lhd b \rhd Q$ (for condition b), non-deterministic choice $\sqcap$, assignment $x :=_A v$ (for expression v and alphabet A), and skip II_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 $\bigsqcap$, $\mu X.P$, 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 3 (see [21] and [43]).

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 through 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 $P\ \mathbf{wp}\ q$. Moreover, UTP provides a way of linking operational semantics to denotational semantics [21] through describing the transition relation $(\sigma, P) \rightarrow (\rho, Q)$, for state configuration predicates σ and ρ and programs P and Q, as a refinement statement – $\sigma'\ ;\ P \sqsubseteq \rho'\ ;\ Q$. From this definition

$$\frac{(\sigma, P) \to (\rho, Q)}{(\sigma, P\ ;\ R) \to (\rho, Q\ ;\ R)}\ \text{SEQ-STEP} \qquad \frac{}{(\sigma, \mathit{II}\ ;\ P) \to (\sigma, P)}\ \text{SEQ-TERM}$$

$$\frac{[\sigma \dagger c]}{(\sigma, P \lhd c \rhd Q) \to (\sigma, P)}\ \text{COND-TRUE} \qquad \frac{[\neg(\sigma \dagger c)]}{(\sigma, P \lhd c \rhd Q) \to (\sigma, Q)}\ \text{COND-FALSE}$$

$$\frac{}{(\sigma, x := v) \to (\sigma(x := \sigma \dagger v), \mathit{II})}\ \text{ASSIGN}$$

$$\frac{[\sigma \dagger c]}{(\sigma, \textbf{\textit{while}}\ c\ \textbf{\textit{do}}\ P) \to (\sigma, P\ ;\ \textbf{\textit{while}}\ c\ \textbf{\textit{do}}\ P)}\ \text{ITER-COPY}$$

$$\frac{[\neg(\sigma \dagger c)]}{(\sigma, \textbf{\textit{while}}\ c\ \textbf{\textit{do}}\ P) \to (\sigma, \mathit{II})}\ \text{ITER-TERM}$$

Table 4: Operational semantics of an imperative language in UTP

we can derive a set of structural operational laws for our imperative language, as illustrated in Table 4. The † operator used in the rules substitutes a state configuration's variable values into an expression ([21] uses a slightly different notation). In this operational semantics the relation II stands for successful termination; thus when a program to left-hand side of a sequential composition reaches this, the right-hand side can begin.

Rule SEQ-STEP describes the evolution of the left-hand side of a sequential composition. Specifically, if P in state σ can perform a transition to Q in state ρ, the a sequential composition can perform the same transition. Rule SEQ-TERM describes the situation when the left-hand side has terminated, leaving II. In this case, the right-hand side is enabled. Rule COND-TRUE describes the situation when the condition of a conditional evaluates to true under state σ. The latter is described by the universal closure of the application of the state substitution σ into condition c. Likewise, rule COND-FALSE describes the situation when the condition is false. Rule ASSIGN describes the execution of an assignment by updating the state σ with a new value for x as the state substitution applied to expression v. Rule ITER-COPY is the iteration copy rule. When the condition c evaluates to *true* then a copy of the loop body P is prepended to the while loop. This can then be executed through rule SEQ-STEP. Finally, rule ITER-TERM describes the situation when c is false and therefore the loop terminates.

As can be seen, 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 [14].

Concretely, a UTP theory consists of three components:

- an *alphabet* of observational variables that provide structure to the semantic denotations;
- a collection of operators (the *signature*) specified in terms of the alphabet;

- *healthiness conditions* – idempotent functions whose image define the theory domain.

For example a simple theory of discrete time could consists of observational variables $now, now' : \mathbb{N}$ giving the time before and after a process has executed. A basic signature could consist of the usual imperative operators to describe programs that take time to execute. We could further define a simple operator $\mathbf{wait}\, n \triangleq now' = now + n$ that allows n time units to elapse, and a healthiness condition $\boldsymbol{DT}(P) \triangleq P \wedge now \leq now'$ that ensures that time can only go forward, and is idempotent since it is conjunctive. Clearly it follows then **wait** is healthy since it satisfies this property, thus $\boldsymbol{DT}(\mathbf{wait}\, n) = \mathbf{wait}\, n$, i.e. $\mathbf{wait}\, n$ is a fixed point of $\boldsymbol{DT}$, as illustrated below:

$$\begin{aligned}
\boldsymbol{DT}(\mathbf{wait}\, n) &= \boldsymbol{DT}(now' = now + n) \\
&= now' = now + n \wedge now \leq now' \\
&= now' = now + n \wedge now \leq now + n \\
&= now' = now + n \wedge true \\
&= now' = now + n.
\end{aligned}$$

Initially for VDM-SL, we are most interested in the UTP theory of designs, that are used to specify terminating imperative programs and Meyer-style contracts with assumptions and commitments [43]. Designs can therefore be used to give an account to both implicit operations and invariants over rich state in the context of VDM-SL modules. The main construct in the signature is the design turnstile $P \vdash Q$, for relations P and Q, stating that, if the assumption P is satisfied then the commitment Q is guaranteed. The signature also contains the aborting design $\bot_D$, the miraculous design $\top_D$ (an infeasible specification), and the usual imperative constructs such as assignment $(x :=_D v)$ and sequential composition lifted into the design space.

Like their relational counterparts, the design signature is given a purely predicative interpretation, specifically $P \vdash Q \triangleq (ok \wedge P) \Rightarrow (ok' \wedge Q)$. The observational variables $ok, ok' : \mathbb{B}$ in this context are used to indicate that a program P has started executing or has successfully terminated, respectively. Designs can equivalently be characterised by two idempotent healthiness conditions called ***H1*** and ***H2***, that is to say if a predicate P is ***H1***-***H2*** healthy, then P can be rewritten in the form $pre(P) \vdash post(P)$, where the functions *pre* and *post* extract the pre and postconditions from P, respectively. For more details please see the UTP book [21] or one of the UTP tutorials [43, 15].

From this definition of designs, we can prove a number of algebraic laws that give further intuition. Some key laws of designs are presented in Table (5). Law (8) states that an unsatisfiable design is equivalent to an abort. Consequently, a design executed outside its precondition results in to abnormal termination. Law (9) states that a design specified after an abort is unreachable. Law (10) states that the non-deterministic choice between two designs conjoins the pre-conditions and disjoins the post-conditions. Law (11) states how the sequential composition of two designs can be calculated to be a single composite design. The composite design assumes that the precondition (p_1) of the first design holds, conjoined with weakest precondition of Q_1 which establishes the second design's precondition (p_2). The composite design's commitment is then simply the relational composition of the two design's commitments.

In addition to such imperative behaviour, VDM-RT also allows the representation of concurrent reactive behaviour. UTP also provides a theory for such systems called the theory of *reactive*

processes that can be used to characterise systems that engage in a series of a events, represented as a set of traces. Reactive processes are characterised by three (pairs of) observational variables:

- $tr, tr' : \text{seq}\, Event$ – describe the sequence of traces the process has executed in before and after being executed;
- $wait, wait' : \mathbb{B}$ – indicate whether the process (before) is waiting for an input from another process;
- $ref, ref' : \mathbb{P}\, Event$ – describe the sets of events that the processes refuses to engage in.

The domain of reactive processes is then characterised by a healthiness condition $\boldsymbol{R}$ that ensures traces and processes are well behaved. Reactive processes can be used to give a semantics to a wide variety of process calculi, such as Milner's CCS [30] and Hoare's CSP [20] as describe in [21] chapter 8. Moreover, reactive processes can be combined with the theory of designs to create *reactive designs* [35]. A reactive design, written $\boldsymbol{R}(P \vdash Q)$ is a reactive program with an assumption P and commitment Q. These constructs can be used to express reactive contracts in a rely-guarantee style [25]. Moreover reactive designs can also be used to express the constructs of CSP. This fact forms the basis for the *Circus* [35] language family that combines the rich-state modelling of designs with reactive behaviour of CSP. We will further consider *Circus* in section 3.

2.4 Isabelle/UTP

In order to verify the correctness of UTP-based semantic models, we need mechanical support for formalising UTP theories, proving algebraic laws, composing them to produce denotational models, and providing provably corresponding semantic bases. In a theorem prover like Isabelle, we can go even further and construct proof tactics and procedures for proving properties of theory objects (i.e. programs or models) in a particular semantic interface, such as a Hoare logic based program verifier. This then means that we have an unbroken chain from proof of program correctness to justification in terms of high-level properties in the underlying denotational semantic models and theories.

We have therefore mechanised the UTP semantic framework in *Isabelle/UTP* [15, 17]. *Isabelle/UTP* is a framework that allows the formation of theories, semi-automated proof of their properties, and theory combination to provide semantic models. It is a (relatively) deep semantic embedding of the UTP relational calculus into the HOL object logic. Unlike typical deep embeddings, it is also integrated with Isabelle's type system and automated proof tactics. Thus it allows both precise reasoning about program semantics, and also facilitates program verification. *Isabelle/UTP* facilitates mechanised theory engineering, that is the creation and

$$\textbf{false} \vdash Q = \bot_D \tag{8}$$
$$\bot_D ;\ (P \vdash Q) = \bot_D \tag{9}$$
$$(P_1 \vdash Q_1) \sqcap (P_2 \vdash Q_2) = (P_1 \wedge P_2 \vdash Q_1 \vee Q_2) \tag{10}$$
$$(p_1 \vdash Q_1) ;\ (p_2 \vdash Q2) = (p_1 \wedge (Q_1 \,\mathbf{wp}\, p_2)) \vdash (Q_1 ;\ Q_2) \tag{11}$$

Table 5: Algebraic laws of designs

```
theorem H1-H2-is-DesignD:
  assumes P is H1 and P is H2
  shows ‘P‘ = ‘¬P^f ⊢ P^t‘
proof −
  from assms have ‘P‘ = ‘$ok ⇒ H2(P)‘
    by (utp-pred-tac)
  also have ... = ‘$ok ⇒ (P^f ∨ (P^t ∧ $ok´))‘
    by (metis H2-split assms)
  also have ... = ‘$ok ∧ ¬ P^f ⇒ $ok´ ∧ P^t‘
    by (utp-pred-auto-tac)
  finally show ?thesis by (metis DesignD-def)
qed
```

Figure 2: A UTP proof using the *ISAR* language

exploration of the theories of programming with machine aided reasoning facilities. For example, all the laws shown in Tables 3 and 5 have been verified with respect to our model in this framework.

We give a model to alphabetised predicates as a derived Isabelle type, and then define the standard constructs of predicates and relations as Isabelle functions. We also give a purely semantic model to expressions, variable renaming, and substitution. We then define proof tactics that allow us to automate proofs. The majority of these tactics work by soundly translating a UTP predicate into some kind of Isabelle/HOL construct, such as sets or binary relations, for which a large number of laws have already been proven. For example *utp-pred-tac* converts a UTP predicate into a HOL predicate, and this allow standard predicate reasoning to be applied. In contrast *utp-rel-tac* can be applied to relational conjecture, through conversion to Isabelle's binary relations. Using these tactics we also prove many of the UTP's algebraic laws.

We can also here leverage *ISAR* [41]to provide readable UTP proofs, for example in Figure 2 we show a proof for one of the laws of designs relating the turnstile operator to healthiness conditions ***H1*** and ***H2***. We specify a theorem with two assumptions and a conclusion, and then open the proof environment to perform a step-by-step equational proof.Specifically, we demonstrate that if a predicate P is ***H1***-***H2*** healthy then it can be written as a design of $\neg P^f \vdash P^t$, where $\neg P^f$ extracts the precondition, and P^t extracts the postcondition. We prove this through number of steps invoking our predicate tactic, *utp-pred-tac* and Isabelle's first-order logic reasoner *metis*.

We have previously used *Isabelle/UTP* to mechanise the theory of designs, the theories of alphabetised relations, reactive processes, and CSP [16]. This then gives the basis for the state-rich reactive parts of VDM-RT. Preliminary work on mechanisation of object-orientation has also be completed [47]. This includes a large library of algebraic laws that can be applied to proving properties of our semantic models. In this deliverable we will use *Isabelle/UTP* to give a preliminary mechanisation of our theory of object-orientation.

We now give a few technical details of *Isabelle/UTP*. The *Isabelle/UTP* core consists of a model of alphabetised predicates and relations including all the standard operators of Boolean algebra, complete lattices, and the relational calculus. Our Isabelle type for UTP predicates *'m uapred* is parametric over a value model *'m*. The value model is a datatype consisting of all

the types constructable within the domain of a particular language. It is necessary for a deep semantic embedding as we cannot easily characterise a type of all the types constructable in HOL (a universe type) without axiomatic extension, only cardinality bounded approximations of such a universe. Nevertheless, the presence of this parameter need not concern most users, but all UTP types of necessity carry this extra $'m$ parameter.

The model-centric approach means we have an open syntax tree for UTP predicates, meaning the new operators can be defined as functions. This is vital for the UTP where new operators are created in every new UTP theory. Isabelle's open parser architecture also ensures maximum flexibility in the creation of an AST with comprehensive pretty printing. Accompanying the core operators we have built a library of proof tactics for solving different classes of problem expressed as UTP predicates, and a large library of algebraic laws to aid in proof. This core then provides the foundation for building more complex UTP theories that enable the characterisation of different programming and modelling paradigms.

Isabelle/UTP provides the following types:

- *'m uapred* – alphabetised predicates
- *('a, 'm) pexpr* – polymorphic expressions of type $'a$
- *('a, 'm) pvar* – polymorphic variables of type $'a$

Predicates can be constructed using the built-in UTP predicate parser, where a predicate can be written using near identical syntax to mathematical UTP surrounding by backticks. There are some exceptions to this, for example the predicate $x = 1 \land y = 2$ contains expressions with UTP variables x and y. In order to distinguish these from Isabelle variables, they must be prepended with a dollar, hence the predicate in *Isabelle/UTP* becomes ‘$\$x = 1 \land \$y = 2$‘.

3 UTP Formal Semantics of VDM-RT

In this section we describe our overall approach to the creation of a new UTP-based semantics for VDM-RT. As we have said, UTP semantic framework allows us to consider the theoretical aspects of a language in isolation, such as object orientation or real-time, and later to compose these with other aspects to produce a complete denotational semantics for a language. Our UTP approach to creation of formal semantics for a language is based around the following three step process:

1. creation of *UTP theories* that isolate and formalise the paradigmatic aspects of a language, such as concurrency, real time, object orientation, or hybrid programming;
2. fusion and linking of UTP theories to generate a suitable model for the target language, and provision of denotational semantics for the language operators;
3. derivation of applicable semantic models, such as operational or axiomatic semantics, from the denotational semantics.

This process is iterative and can be executed in different orders depending on the initial artefacts available. For example, we already have the operational semantics that we mechanised in appendix A (as an aid to understanding) and so that will be used as an input to derivation of the denotational semantics, (though it should be stressed that our semantic model will be new,

independent, and reusable). In addition, we may have a set of algebraic laws for our language operators and these should likewise be respected by the UTP denotational and operational semantics. In this work we will focus principally on step (1), and consider the aspects of the VDM-RT language as UTP theories. Ultimately, our aim is the creation of a *lingua franca* for the various notations in INTO-CPS, that we dub *INTO-CSP*, that will be built in steps (2) and (3) as a fusion of the various different theories we develop. Thus in this deliverable we consider a theory of object orientation, that can later be augmented with theories of real-time and concurrency and cover the other aspects of VDM-RT. We will briefly consider the latter aspects before going on to object-orientation in more depth.

3.1 VDM-RT in the UTP

Real-time, concurrency, and rich-state modelling have already been given substantial study in the UTP, specifically within the context of the *Circus* language family [35]. *Circus* is a formal modelling language that combines the constructs of the CSP language [20] for modelling of concurrent systems with rich-state modelling as provided by the Z specification language [45]. *Circus* thus provides semantically equivalent constructs to those in VDM-SL for modelling operations and state, together with the ability to represent concurrent stateful systems. Process state variables in *Circus* are not shared by concurrent processes, and so information can be conveyed from process to process only through CSP-style communication channels. As we also mentioned in Section 2.3 the basis of *Circus* is the reactive design, that allows the expression of reactive specifications with assumptions and commitments.

The *CircusTime* language [40] is an extension of *Circus* that introduces discrete real-time modelling constructs such as timeouts and deadlines. Additionally, the COMPASS modelling language [44] (*CML*) is a more recent development that provides similar modelling constructs, but in the context of a new semantic model that has an improved treatment of time in the presence of operators like external choice [6]. All of these languages are based on an extension of reactive designs called *timed reactive designs* [18, 42]. A timed reactive design, is defined in terms of a healthiness condition $\mathbf{RT}$ that subsumes $\mathbf{R}$ from reactive designs, and also ensures that measurement of time is well-behaved. A timed reactive design is then written as $\mathbf{RT}(P \vdash Q)$, for timed assumption P and commitment Q. The addition of timing information here allows the expression of constraints like timing budgets, and could therefore we used to give an account to implicit operations in VDM-RT.

Also of interest here are languages like Timed CSP [36] and Hybrid CSP [19] that provide support for continuous time modelling (and in the case of the latter, system dynamics). Although VDM-RT is discrete time, other notations such as Modelica [32] are continuous time based and so the links between discrete and continuous CSP variants needs to be considered to allow the use of these languages in a co-simulation framework. Again, this is a big motivation for the creation of INTO-CSP.

The semantic model of *CircusTime* or *CML* can thus be applied to give a semantics to the real-time CPUs and threads of VDM-RT. We will model each thread, object, CPU, and bus from VDM-RT as different types of *Circus* processes, as outlined in Figure 3. A thread would, for example, have a collection of local state variables that would be synchronised with the corresponding object process variable store, by the CPU process, when sufficient time has passed as defined by the enclosing `duration` statement. The `duration` statement also has deadline capabilities which can also be modelled in *CML* and *CircusTime* using the deadline

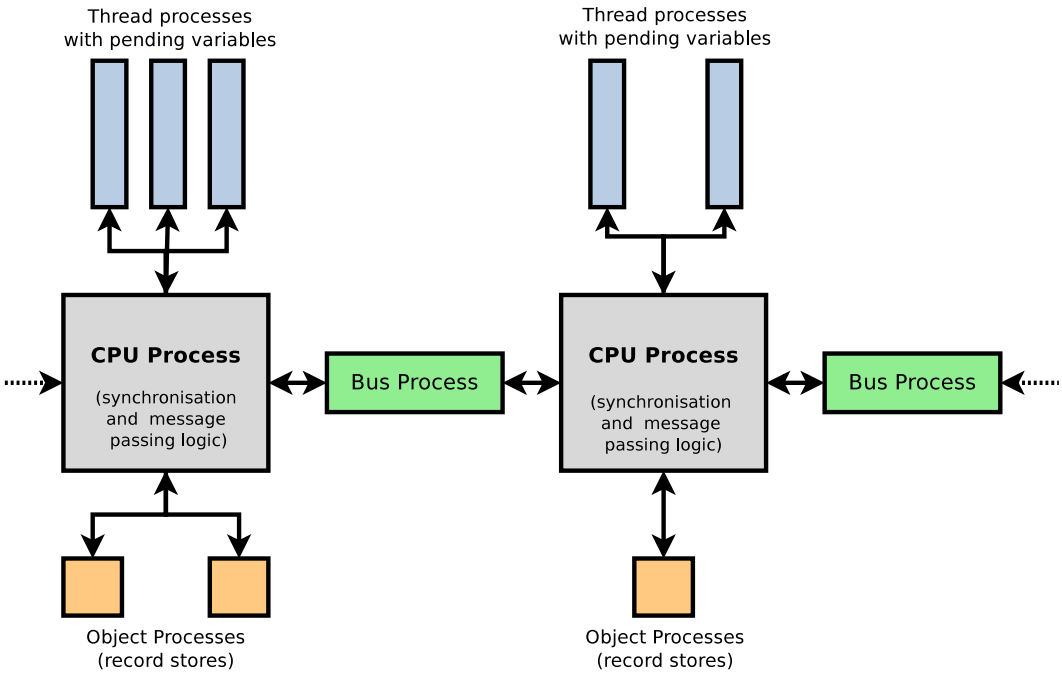


Figure 3: Overview of VDM-RT entities modelled as *Circus* processes

operator $P \blacktriangleright t$, where P must terminate before t time units expire. Busses are processes which deliver messages between CPUs whilst exhibiting time penalties that can be modelled by the wait statement, *Wait* n that pauses execution for n time units. CPUs then also need to provide routing for operation call messages using appropriate *Circus* channels and events.

Time in such a setup can be measured by a global clock, and suitable synchronisation measures provided for the various threads. In this way the system can be ordered to advance a certain number of units, or until a condition on an object state variable is satisfied. This will then allow us to link the VDM-RT semantics to FMI [13], which requires the specification of a *doStep(s, n)* function that advances a model in state s by up to n time units, in the presence of certain inputs [5]. For more information on the semantics of FMI, please see the sister Deliverable D2.1d [1]. For VDM-RT model, the *doStep* function will be given a semantics that executes all possible behaviour across the CPUs, until either the given time bound n has elapsed, or else one of externally visible discrete variables changes state.

We have thus provided the context for the work in the remaining sections of this particular deliverable. Here we focus, not on the real-time behaviour, but on the state space of the individual objects as typed by VDM-RT classes.

3.2 Object-oriented semantics in the UTP

Object-orientation was first given a UTP semantics in the work of Santos [37, 10]. This work provides a general theory for the definition of class structures containing attributes and methods. Classes can be inherited to add further attributes and methods, and method implementations in parent classes can be overridden as is usual in OO languages. Unlike VDM-RT, Santos' theory considers only classes which have one parent, that is single-inheritance, where VDM-RT has multiple-inheritance.

The class theory uses the UTP theory of designs to define these commands in a purely relational setting. Objects in this setting are simply described by records that enumerate the attributes of the corresponding class. Methods are represented as UTP procedures that are defined with the context of Higher Order UTP (see [21] chapter 9 and also [46]) that enables predicates to contain variables that themselves have predicate types. Thus a method definition corresponds to a UTP variable with such a predicate type defining the method's implementation.

This work was later extended by Zeyda [47] who added a modular approach to method definitions in the presence of recursion. The original work [10] requires that a collection of (mutually) recursive methods be declared simultaneously so that a common fixed-point can be declared that encompasses them all. Zeyda [47] overcomes this problem by introducing higher-order method variables that can be invoked in the body of a method and only later bound to an actual implementation. This means the fixed point need only be created at the point of method call, not the point of definition. We will not explicitly consider recursion in our theory, although we note that Zeyda's work is fully compatible.

3.3 UTP semantics of VDM-RT classes

Although VDM-RT already has an operational semantics [28] (see also appendix A), this does not cover the semantics of the declarative class mechanism. Specifically, although it supports the creation of objects from classes, the assumption is that class definitions have already been flattened to deal with things like inheritance and overriding, so that an unambiguous collection of definitions is available in the operational context. Morever, as highlighted in Section 2.1 a number of unresolved issues exist in the tool implementation of VDM-RT, such as diamond inheritance.

The aim of the new UTP semantics is to provide a semantic framework for VDM-RT classes, in which the relationships between operators can be precisely studied. This will also allow us to begin to address some of semantic issues outlines above. Though we cannot give solutions to all of these issues, many of which require further discussion on the intended semantics of VDM-RT, nevertheless we hope that our semantics provides a framework in which these issues can be explored. Indeed the UTP has been designed specifically for this purpose.

Our approach to object-orientation in the VDM-RT semantics is two-fold. Firstly we define an extended version of Santos' UTP theory of classes [10] that adds support for multiple inheritance in Section 4. This will act as the low-level declarative language for classes, and will have a relational semantics. This theory is highly abstract, and does not directly consider concepts which can be defined through syntactic transformation such as method overloading, or static classes. This general theory then provides us with a framework in which different kinds of object-oriented programming language can be constrasted. Secondly, in Section 5 we define a syntactic mapping from VDM-RT classes into this extended theory that provides the denotational semantics. In this work we do not focus on recursive methods, instead we refer the reader to [47].

For multiple inheritance, the main problem to be solved is how a method implementation should be selected when two or more disjoint ancestor classes implement the same method. Different languages take different strategies here. Java, for instance, allows only multiple inheritance of class interfaces, that can only specify method signatures and not implementations thus removing the possibility of conflict. C++ allows multiple inheritance, but insists that methods be fully

qualified by the originating class name so that no ambiguity is present. Scala [34] implements *traits*, which are like Java interfaces except that they can provide method implementations, but unlike classes cannot have parametric constructors. These restrictions allow an algorithm that can unambiguously pick an appropriate method implementation based on the order in which a trait was added to the class. Eiffel [29] takes a different approach again wherein unrestricted multiple inheritance is allowed, but the programming must explicitly state which features are inherited into the new class. Moreover different features with the same name are disallowed, so there can be no ambiguity present. For VDM-RT, therefore, a number of different options are available, and so for this work we will simply provide non-deterministic selection of inherited methods, with possible refinement of this as future work.

4 Theory of Classes and Objects

In this section we present a UTP-based theory of classes and objects that is generally applicable to semantics of OO languages and in particular can be applied to VDM-RT (and hence VDM++). We will use the mathematical Z notation [45] to give the definitions of our operators[4]. Our theory is a conservative extension of the theory developed by Santos [37, 10] and then later extended by Zeyda and others [47]. Specifically, we augment class declaration with the ability to declare classes with multiple superclasses, and appropriately update the method call semantics taking this into account. Method resolution is supported by representing methods as explicit tables of possible implementations, rather than as a cascade of tests on the specified object as in the original works.

The theory we define will then in turn be used to give a semantics to VDM++ classes in Section 5. Defining a general theory rather than giving a direct semantics to VDM++ will allow us to draw out the common elements with those of other language which also exhibit object-oriented aspects, such as Modelica, though this is left as future work.

Our theory relies heavily on the existing UTP theories of designs and procedures. We also make use of UTP procedures that we will use to give a semantics to VDM-SL operation calls. A UTP procedure of the form $pds \bullet P$ consists of a set of parameters pds and a body P using the parameters. Parameters can take three forms:

- Value parameters ($\mathbf{val}\, x : A$), where the caller supplies an input value of type A that populates variable x;
- Result parameters ($\mathbf{res}\, x : A$), where the caller supplies a variable of type A that the procedure writes to. Result parameters thus generalise the concept of a method "return" statement;
- Value-result parameters ($\mathbf{vres}\, x : A$), where caller supplies a variable and the procedure both reads from and writes to it.

For example, we can specify a simple operation for adding two numbers together:

$$add \triangleq \mathbf{val}\, x : \mathbb{N};\ \mathbf{val}\, y : \mathbb{N};\ \mathbf{res}\, r : \mathbb{N} \bullet r := x + y$$

[4] A helpful reference manual detailing all the operators we use can be found at http://spivey.oriel.ox.ac.uk/~mike/zrm/zrm.pdf

This has two value parameters, x and y of type $\mathbb{N}$, and a result parameter r of type $\mathbb{N}$. The body assigns $x + y$ to r that will cause it's value to be propagated to the caller. Internally, a UTP operation is simply a λ-abstracted predicated, and therefore can be called like a normal function: $add(5, 7, x)$. The return statement in languages like VDM-RT can be handled through creation of a distinguished result parameter, such as `RESULT`, to which the return statement assigns the given expression.

4.1 Extended Theory of Classes

As we stated in Section 2.3 a UTP theory consists of its *alphabet* of observational variables, the *signature* of operators in the language, and the *healthiness conditions* as idempotent functions that constrain the theory's domain. We will now proceed to define each of these for our extended theory of classes. We assume the following types for our theory of classes:

- *Type*, the set of all type names;
- $\mathcal{T}(\subset \mathit{Type})$, the set of basic type names;
- $\mathit{CName}(\subset \mathit{Type})$, the set of class names;
- *AName*, the set of attribute[5] names.

For VDM-RT, $\mathcal{T}$ consists of the usual constructions such as strings, integers, sequences, maps, and sets. We also assume a function $\mathit{carrier} : \mathcal{T} \to \mathbb{P}\,\mathbb{U}$ that gives the set of values for each type (in a suitable universe). Moreover, we assume that every type yields a non-empty carrier, that is $\forall\, t \in \mathcal{T} \,.\, \exists\, v.v \in \mathit{carrier}(t)$, which ensures that we can always pick an arbitrary value for each type by using the indefinite description operator ϵ (as in Isabelle/HOL). Next we give the observational variables of our UTP theory of classes.

$$
\begin{aligned}
\mathit{cls} &: \mathbb{P}\,\mathit{CName} \\
\mathit{atts} &: \mathit{CName} \pfun (\mathit{AName} \pfun \mathit{Type}) \\
\mathit{sc} &: \mathit{CName} \rel \mathit{CName} \\
\mathit{ivr} &: \mathit{CName} \pfun \textbf{\textit{Object}} \pfun \mathit{Pred}
\end{aligned}
$$

$$
\begin{aligned}
C \prec D &\mathrel{\widehat{=}} (C, D) \in \mathit{sc}^+ \\
C \preceq D &\mathrel{\widehat{=}} (C, D) \in \mathit{sc}^*
\end{aligned}
$$

The observational variable *cls* records the set of defined classes, *atts* is a partial function assigning a set of attributes to each class, and *sc* is the subclass relationship. We deviate slightly from [10] and [47] in that *sc* is now a relation ($\mathit{CName} \rel \mathit{CName}$) rather than a partial function. This allows us to support multiple inheritance, where the original work only supported single inheritance. We also introduce syntax for the strict superclass relation $\prec$, and the non-strict version $\preceq$.

As a side note, we could have alternatively defined *sc* to have type $\mathit{CName} \pfun \mathrm{iseq}\,\mathit{CName}$, which would also impose an order on the inherited classes. This can potentially be useful to help resolve conflicts between overridden methods. However, for now we opt for the simpler version, and leave potential refinement to future work. We also define a new observational

[5] Attributes in this language correspond to instance variables in VDM-RT. We will sometimes use these terms interchangeably.

	Invariant ψ for **SIH**(ψ)
OO1	***Object*** $\in cls$
OO2	$\mathrm{dom}\, sc = cls \setminus \{$***Object***$\}$
OO2a	$\mathrm{ran}\, sc \subseteq cls$
OO3	$\forall C \in \mathrm{dom}\, sc \bullet (C,$ ***Object***$) \in sc^+$
OO3a	$\forall C \in \mathrm{dom}\, sc \bullet \forall D_1, D_2 \in sc(\!\vert \{C\} \vert\!) \mid D_1 \neq D_2 \bullet (D_1, D_2) \notin sc^+$
OO4	$\mathrm{dom}\, atts = cls$
OO4a	$\mathrm{dom}\, ivr = cls$
OO5	$\forall C_1, C_2 \mid C_1 \neq C_2 \bullet \mathrm{dom}(atts\, C_1) \cap \mathrm{dom}(atts\, C_2) = \emptyset$
OO6	$\mathrm{ran}(\bigcup \mathrm{ran}\ atts) \subseteq \mathcal{T} \cup cls$

Table 6: Healthiness conditions of class theory

variable *ivr* that associates an invariant predicate with each class. As usual, we also define a distinguished class ***Object*** that acts as the base of the class hierarchy.

With our observational variables defined, we can proceed to define our healthiness conditions which define the invariants on the class structure. They are given in Table 6. As with [47] we define these relative to the theory of invariants [9]. The theory of invariants effectively converts a predicate over the state to a relation that specifies preservation of that predicate by lifting it into a design. It defines a parametric healthiness condition ***SIH***(ϕ), where ϕ is the invariant, such that ***SIH***$(\phi)(P \vdash Q) = (P \wedge \phi) \vdash (Q \wedge \phi')$. Thus our class invariants are both assumed and preserved by elements of the UTP theory.

The healthiness conditions are mainly adapted from [47], though we have added three additional healthiness conditions. ***OO1*** states that ***Object*** must always be a member of the set of classes and thus this set is never vacuous. ***OO2*** states that the domain of the subclass relationship includes every class other than ***Object***, or alternatively every class other than ***Object*** has a superclass. ***OO3*** states that every class must be related to the ***Object*** class via the subclass relationship.

In the original work ***OO2*** and ***OO3*** together were sufficient to ensure that the subclass graph is acyclic, effectively by ensuring that the underlying relation forms a tree rooted by ***Object***. However, since we now permit multiple inheritance, we do not necessarily have a tree, but rather a directed acyclic graph (DAG). We therefore require two additional healthiness conditions, ***OO2a*** and ***OO3a*** (named so as to remain consistent with existing work).

OO2a requires that the range of *sc* is within the set of classes. This was automatically satisfied in the case of single inheritance through ***OO2*** and ***OO3*** since every class other than the root class ***Object*** is in the domain, and so each class must eventually reach the root (the inheritance tree is rooted). However, since we have multiple inheritance, apart from ***OO2a*** it would be possible to have classes with two roots. ***OO3a*** disallows redundant inheritance, that is inheriting classes D_1 and D_2 where D_1 is superseded by D_2. Specifically, it states that if D_1 and D_2 are in the image of *sc* under $\{C\}$[6], and it follows that D_1 and D_2 are different, then D_1 and D_2 cannot be related in the subclass hierarchy sc^+. This also helps ensure that there are no cycles in sc^+. ***OO3a*** is trivially satisfied in the case of single inheritance.

[6]The $(\!|\ |\!)$ brackets denote relational image

Healthiness condition ***OO4*** states that attributes are defined for each class, and ***OO4a*** ensures that an invariant is defined for each class. ***OO5*** states that no two different classes can have the same attributes, that is attribute names are unique. Finally, ***OO6*** states that the type of each attribute must either be a basic type, or else a defined class in cls. Thus we can define the overall healthiness condition for class declaration:

$$\mathbf{OO} \mathrel{\widehat{=}} \mathbf{SIH}\left(\begin{array}{l}\mathbf{OO1} \wedge \mathbf{OO2} \wedge \mathbf{OO2a} \wedge \mathbf{OO3} \wedge \mathbf{OO3a} \wedge \\ \mathbf{OO4} \wedge \mathbf{OO4a} \wedge \mathbf{OO5} \wedge \mathbf{OO6}\end{array}\right)$$

This healthiness condition is idempotent, since any ***SIH*** lifted healthiness condition is idempotent through the idempotency of conjunction. From these healthiness conditions, we can prove the following important property.

Theorem 4.1 ($\prec$ is a strict partial order under *OO*)

Proof 4.1 Assume ***OO***. It suffices to show that $\prec$ is transitive and irreflexive:

- **Transitivity**. By definition of $\prec$ as a transitive closure.
- **Irreflexivity**. By contradiction. Assume that there is a class A where $A \prec A$. Then there is a chain $B_1 \cdots B_n$ where $\forall\, 1 \leq i \leq n-1 \bullet (B_i, B_{i+1}) \in sc$, such that $B_1 = B_n = A$. We know that for any i such that $1 \leq i \leq n$, by ***OO2*** that $B_i \neq \mathbf{Object}$, since ***Object*** can have no parent, and by ***OO3*** that $B_i \prec \mathbf{Object}$. Then it must be the case that there exists k and C such that $1 \leq k \leq n$ where $(B_k, C) \in sc$ and $\nexists\, i.B_i = C$, i.e. C is not in the chain. Moreover, it follows that $(B_k, B_{k+1}) \in sc$. But then also $B_{k+1} \prec B_k$ and by transitivity, $B_{k+1} \prec C$. By ***OO3a*** this is not allowed and so we have a contradiction. □

As a corollary, we can assert that $\prec$ is acyclic as required.

4.2 Class Declaration

The class theory defines commands for the creation of classes, attributes, invariants, and methods. It contains the following operators, the first three of which are adapted directly from [10]:

- ***class*** A ***extends*** $B_1 \cdots B_n$ – create a new class called A, with superclasses $B_1 \cdots B_n$;
- ***att*** $A\ x : T$ – create a new attribute in class A called x with type T;
- ***meth*** $A\ m\ pds \bullet p$ – create a new method in class A called m with parameters pds and body p;
- ***invar*** $A\ p$ – add an invariant predicate p to class A or augment the existing invariant.

The definition of a class then consists of a sequence of commands that create such definitions. We give a semantics to this calculus using the theory of designs, where each construct will specify as the precondition the declarations that must already exist. The postcondition will then add new declarations. Moreover, each command must preserve the invariants specified in ***OO***. Thus each class definition command has the form $\mathbf{OO}(P \vdash Q)$, i.e. it is a design that satisfies the theory invariants.

For our semantic definitions we assume the presence of a fixed contextual alphabet Σ that contains all the theory observational variables and method variables declared. We will now

proceed to give definitions for each of the commands in our class calculus. These definitions are heavily mathematical, and so a casual reader may wish to skip to Section 5 to see how these commands are applied in the VDM-RT semantics.

4.2.1 Classes and Attributes

Definition 4.1 (Class introduction) The declaration of a class is defined as shown below.

$$\textbf{\textit{class}}\ A\ \textbf{\textit{extends}}\ B_1 \cdots B_n\ \widehat{=}\ \textbf{\textit{OO}} \left(\begin{array}{l} A \neq \textbf{\textit{Object}} \land A \notin \mathcal{T} \cup cls \land \\ (\forall i \in \{1 \cdots n\} \bullet B_i \in cls) \land \\ (\forall i,j \in \{1 \cdots n\} \bullet B_i \not\prec B_j) \\ \vdash \\ (cls' = cls \cup \{A\}) \\ \land\ (sc' = sc \cup \{A \mapsto B_i \mid 1 \leq i \leq n\}) \\ \land\ (atts' = atts \cup \{A \mapsto \emptyset\}) \\ \land\ (ivr' = ivr \cup \{A \mapsto \textbf{\textit{val self}} : \textbf{\textit{Object}} \bullet \textbf{\textit{true}}\}) \\ \land\ \mathit{II}_{\Sigma \setminus \{cls,sc,atts,ivr\}} \end{array} \right)$$

The design updates variable cls with A and maps it in the relation sc to $B_1 \cdots B_n$ as its immediate superclasses. Only new names are allowed ($A \notin \mathcal{T} \cup cls$), and classes $B_1 \cdots B_n$ need to have been previously declared ($B_i \in cls$). Moreover we require that no two superclasses are ancestors of each other; this ensures that ***OO3a*** is satisfied. An entry for A in $atts$ is mapped to the empty set, and the initial class invariant is simply ***true*** for any object. No other observational variable is modified as specified through conjoining the predicate with a skip operator (II) parametrised by the unchanged variables.

Definition 4.2 (Attribute introduction) To introduce an attribute x of type T in class A we can use the construct defined below.

$$\textbf{\textit{att}}\ A\ x : T\ \widehat{=}\ \textbf{\textit{OO}} \left(\begin{array}{l} A \in cls \\ \land\ x \notin \mathrm{dom} \bigcup \{ N : cls \bullet atts(N) \} \\ \land\ T \in \mathcal{T} \cup cls \\ \vdash \\ (atts' = atts \oplus \{A \mapsto (atts(A) \cup \{x \mapsto T\})\}) \\ \land\ \mathit{II}_{\Sigma \setminus \{atts\}} \end{array} \right)$$

The class A must exist, the attribute must not already be declared in any existing class, and the type must be a valid type. The second of these may seem like a substantial limitation in the creation of class hierarchies, but it should be noted that each attribute name of a VDM-RT class is qualified by the containing class. This allows us to overcome the problem of diamond inheritance for attributes (see Section 5.1). Assuming all the these preconditions are satisfied, the set of attributes is updated.

4.2.2 Method Definition

Each class can potentially provide its own implementation of a method m. Thus, a method is represented by a variable $m : CName \nrightarrow (\textbf{\textit{Object}} \times T_1 \times \cdots \times T_n \rightarrow Pred)$ that represents the

method table associated with m. As usual the method takes the object as a parameter, together with value and result parameters. As new method implementations are defined, this table is updated to include implementations for particular classes. This is in contrast to [10] where each method is simply represented by the body of that method with a top-level conditional that picks the appropriate implementation based on the given object. We chose to explicitly represent the table as it enables non-deterministic method selection for multiple inheritance.

Definition 4.3 (New method introduction) For new methods, the declaration is defined as follows.

$$\begin{array}{l} \textbf{\textit{meth}}\ A\ m\ pds \bullet p\ \widehat{=} \\ \quad \textbf{\textit{OO}} \left(\begin{array}{l} \textbf{\textit{var}}\ m\ ; \\ \left(\begin{array}{l} A \in cls \\ \land\ \forall t \in types(pds) \bullet t \in \mathcal{T} \cup cls \\ \vdash \\ (m' = \{A \mapsto \textbf{\textit{vres self}} : \textbf{\textit{Object}}\ ; pds \bullet P\}) \\ \land\ \Pi_{\Sigma \setminus \{m\}} \end{array} \right) \end{array} \right) \\ \textbf{\textit{provided}}\ m \notin \Sigma \end{array}$$

We introduce a new variable m into scope, the identifier for the new method. The class design requires that the class A exists, and that each type in the list of method parameters is an existing type. That being the case, m is assigned to a singleton mapping from the class name A to the method body, which is extended with the object value-result parameter ***self***. This operator has a well defined behaviour only when m is not already in Σ, i.e. it is not already a declared method. Otherwise the operator below should be used instead. It should be noted that this construct is built on top of higher-order UTP (see chapter 9 of [21]), since it assigns a predicate (the body of the method) to a variable.

Definition 4.4 (Method redefinition) If the method name declared is not new, the corresponding definition is the following.

$$\begin{array}{l} \textbf{\textit{meth}}\ A\ m\ pds \bullet p\ \widehat{=} \\ \quad \textbf{\textit{OO}} \left(\begin{array}{l} A \in cls \\ \land\ \forall t \in types(pds) \bullet t \in \mathcal{T} \cup cls \\ \vdash \\ (m' = m \oplus \{A \mapsto \textbf{\textit{vres self}} : \textbf{\textit{Object}}\ ; pds \bullet P\}) \\ \land\ \Pi_{\Sigma \setminus \{m\}} \end{array} \right) \\ \textbf{\textit{provided}}\ m \in \Sigma \end{array}$$

If the method name has been declared already, then rather than creating a new variable we need to update the existing one. The preconditions of the design are the same, but the postcondition updates the method table with a new implementation for class A.

Definition 4.5 (Invariant definition) To introduce an invariant, or augment an existing one, we can use the construct defined below.

$$\begin{array}{l} \textbf{\textit{invar}}\ A\ p\ \widehat{=} \\ \quad \textbf{\textit{OO}} \left(\begin{array}{l} A \in cls \\ \vdash \\ (ivr' = ivr \oplus \{A \mapsto \textbf{\textit{val self}} : \textbf{\textit{Object}} \bullet ivr(A)(\textbf{\textit{self}}) \land p\}) \\ \land\ \Pi_{\Sigma \setminus \{ivr\}} \end{array} \right) \end{array}$$

Definition of an invariant requires that the specified class exists, and, if it does, updates the invariant map. If an invariant already exists for the class, then the new formula is conjoined with the old invariant.

4.3 Object Manipulation

With the operators defined for declaring classes, attributes, and methods we proceed to define the core language elements for creating and manipulating objects of these classes. The UTP gives us a language of imperative programs in the theory of alphabetised relations, and as illustrated in Figure 4 on page 13 an operational semantics can be derived from this. Since objects in this setting are simply instances of record types, we can use the standard assignment and other operators to handle them as ordinary values. However, we need to define some additional operators to deal with method call and object creation, and this is the purpose of this section.

4.3.1 Object Expressions

UTP expressions do not, by default, contain explicit type information. For object oriented programming it is necessary that we carry type data about the class to which an object belongs. We thus define a derived form of expression and functions for querying this type data. An object expression e is represented by a pair: (e_τ, e_v), where the first element $e_\tau \in \mathcal{T} \cup cls$ is the type of e and the second element e_v is its value. We can then define the construct ***null*** that stands for a family of values, one for each class. The type held by e_τ in this case is inferred from the context. For instance, in an assignment $x := \textbf{\textit{null}}$, we have that $e_\tau = x_\tau$; this means that the runtime type of ***null*** is the declared type of variable x. For our implementation of the UTP in Isabelle all variables and expressions carry such data from the type system, thus this is a valid assumption (*Isabelle/UTP* is inherently strongly typed in nature).

4.3.2 Method Call

Method call in our calculus differs from [10] in that we need to pick the appropriate method implementation from the table. The presence of multiple inheritance means that several valid implementations can be present, and so we make a non-deterministic choice between them. If only one implementation is present this one will be selected.

Definition 4.6 (Method call (unqualified)) A call to an unqualified method is defined as follows.

$$obj.f(v_1 \cdots v_n) \mathrel{\widehat{=}} \bigsqcap \{m(obj, v_1 \cdots v_n) \mid m \in f(\!| \mathit{Min}_{\prec}\{C \in \mathrm{dom} f \mid obj_\tau \preceq C\} |\!)\}$$
$$\textbf{\textit{where}}\ \mathit{Min}_{<}(A) \mathrel{\widehat{=}} \{x \in A \mid \forall y \in A.\, y < x \Rightarrow y = x\}$$

We first take the set of superclasses of the object's class obj_τ that implements the given method f (i.e. $C \in \mathrm{dom} f$). We then calculate from this the set of minimal elements $\mathit{Min}_{\prec}$, that is, those classes whose implementation of f has not been overridden. Since by theorem 4.1 we know that $\prec$ is a partial order, we also know that at least one such minimal element must exist. Once

we have obtained all such candidate methods, we apply them to the object and parameters and take a non-deterministic choice of the results.

VDM-RT also allows explicitly qualified method calls, with which an implementing class can be explicitly chosen.

Definition 4.7 (Method call (qualified))

$$obj.A`f(v_1 \cdots v_n) \mathrel{\widehat{=}} \bigsqcap \{m(obj, v_1 \cdots v_n) \mid m \in f(\!| \{A\} |\!)\}$$

The semantics here is much simpler as we can directly pick the class method.

4.3.3 Object creation

An object value is a pair $(type, value)$: the *type* is a class name and the *value* is a mapping from names to state component values. Using *sc* and *atts* to recover state components and inheritance information, we provide a definition for ***new*** as follows.

$$\begin{array}{l} \textbf{\textit{new}}\ N \mathrel{\widehat{=}} \\ \quad \left(N, \left\{ \begin{array}{l} x : \mathrm{dom}\, map;\ t : Type;\ v : \mathbb{U} \mid \\ ((map(x) \in \mathcal{T}) \land (t = map(x)) \land (v = \epsilon x \bullet x \in carrier(t))) \\ \lor\ (\exists T : cls \bullet (map(x) = T) \land (t = T) \land (v = \textbf{\textit{null}})) \bullet \\ x \mapsto (t, v) \end{array} \right\} \right) \\ \textbf{\textit{where}}\ map = \bigcup(atts(\!| sc^*(\!| \{N\} |\!) |\!)) \end{array}$$

An object is represented as a record mapping each attribute name to the respective type and value. Creation of a new object involves creating an instance of such a record with default values for each attribute. First, though, we must establish a collection of attributes associated with the class, which includes the attribute from all inherited classes. This is role of the constant *map*; we use the reflexive transitive closure of *sc* to calculate all superclasses of N and then extracts the attributes from each class. Each attribute type in the class can either be a member of $\mathcal{T}$ (a basic type) or *cls* (a class). If it is a basic type attribute, then we simply pick an arbitrary member of the type's carrier. If it is a class we insert the value ***null***.

5 Semantics of VDM-RT Classes

In this section we show how VDM-RT classes can be given a semantics in terms of the calculus presented in section 4. We first describe some preprocessing steps, and then describe the semantic transformations for the VDM-RT language.

5.1 Preprocessing

Our class theory does not explicitly handle resolution of name clashes through shadowing, multiple inheritance, or overloading. There is a general assumption that attributes and differently typed methods possess unique names across the scope of all classes. In real programs this is often not be the case, and so prior to application of our semantic transformations we require some

```
                    class A
                      instance variables
                        x : real
                    end

class B
  instance variables
    y : seq of nat1                    class C
                                         instance variables
  operations                               y : set of char
    bop : nat1 ==> ()                  end
    bop(n) == y := [n] ^ y
end

            class D is subclass of B, C end
```

Table 7: Diamond inheritance problem

initial static type checking and α-renaming. This static renaming should be performed either during parsing or type checking in an implementing tool. This is possible since, unlike method overriding, which is subject to dynamic dispatch, attribute lookup and overloading resolution can always be statically determined.

Attribute uniqueness is enforced through explicitly qualifying each attribute with the name of the class in which is was declared. This serves to ensure that each object, when created, contains each attribute of each inherited class qualified by the name of that class. This also partially solves the diamond inheritance problem, illustrated in Table 7. Two instances of the problem manifest in class `D`: firstly, attribute `x` is inherited twice (from both `B` and `C`), and two differently typed versions of `y` are given. Qualifying the attribute names means that class `D` has three attributes, namely `A`‘`x`, `B`‘`y`, and `C`‘`y`.

Methods that access these variables access their local version – for example, in the body of operation `bop`, `y` is is renamed so that `y` becomes `B`‘`y`. Moreover, since only one copy of `x` is present, all methods access the same variable; there is no need to distinguish different copies. An attempt to access an attribute called `y` in `D` can either result in a type error (if this situation is undesirable), or else a standard resolution strategy can be adopted, such as right-to-left ordering of superclass attributes. We leave this as an open question for discussion.

In general an attribute access expression can be rewritten thusly:

$$\texttt{x.a} \rightsquigarrow \texttt{x.A`a}$$
$$\textbf{\textit{where}}\ \texttt{x} : \texttt{B},\ \texttt{A} \in \mathsf{Min}_{\prec}\{\texttt{C} \mid \texttt{B} \preceq \texttt{C} \wedge \texttt{a} \in \mathit{attr}(\texttt{C})\}$$

$\mathsf{Min}_{<}$ yields the (non-empty) set of minimal elements under the strict partial order $<$, $\prec$ and $\preceq$ are the strict and non-strict subclass relations, and *attr* yields the set of attribute names defined in a class. Thus `A` is picked from one of the most direct ancestors of `B` that define an attribute named `a`. If `B` itself defines `a`, then a singleton set results, and thus `B` is picked. This semantics requires that we can query the subclass relationship and the direct attributes of a class, so in an implementing tool this information must be computed as an initial step.

Overloading is resolved through renaming methods to include the input and output types. For example, if we can have two identically named operations:

```
myop : nat ==> ()
```

```
myop : char ==> ()
```

Then we would create two methods, `myop_nat` and `myop_char`, corresponding to the two types. Calls to such operations are likewise rewritten through a static type check to determine the correct input parameters. Alternatively if being implemented within a logical tool like Isabelle that supports overloading natively these facilities can also be utilised (as we do in Section 6).

5.2 Semantic Transformations

In this section we define a collection of semantic functions that map elements of the α-renamed VDM++ abstract syntax tree to UTP class predicates. We present these semantic functions using the Oxford-style semantic brackets. This a departure from the UTP approach to simply having elements of the language distinguished with a teletype font face. We use this approach as we sometimes also want to carry contextual information in the transformation, and we want to emphasise that this stage is technically the second stage of rewriting.

Aside from the functions here explicitly defined, we also assume the following functions:

- $[\![e]\!]_{\mathbf{exp}}$ – maps a VDM expression to a UTP expression. The translation is relatively straightforward as both have their foundations in ZF set theory;
- $[\![P]\!]_{\mathbf{prg}}$ – maps a VDM operation body (a program) to a UTP alphabetised relation. Again this is more or less direct, and uses the operators defined for object manipulation in Section 4.3;
- $[\![tydefs]\!]^A_{\mathbf{tyd}}$ – defines constants for each of the given type definitions. Types in VDM are simply represented as sets in UTP (or types in HOL);
- $[\![valdefs]\!]^A_{\mathbf{vld}}$ – defines constants for each of the value definitions;
- $[\![valdefs]\!]^A_{\mathbf{fnd}}$ – defines constants for each of the function definitions.

In the following transformations, we focus only on the object-oriented aspects.

5.2.1 Classes

The semantics of a VDM++ (or VDM-RT) class is given as a mapping into the class calculus. Declaration of a class A actually yields two class declarations in the calculus: one named A that corresponds to the actual declared class, and one named *A-stat* that will group the static elements of A, such as attributes. This treatment of static constructs is novel contribution beyond the previous work in [10, 47].

Declarations of types, values, functions, invariants, and operations are all given a semantics through corresponding semantic functions. A default constructor for the static class is created. It is called *A-stat* and assigns default values to each of the static instance variables. Finally, a single object of type *A-stat* is created that acts as the companion object for A with all static content.

$$\left[\!\!\left[\begin{array}{l}\textbf{class}\ A\ \textbf{is subclass of}\ P_1 \cdots P_n \\ \quad \textbf{types}\ tydefs \\ \quad \textbf{values}\ valdefs \\ \quad \textbf{functions}\ fundefs \\ \quad \textbf{instance variables}\ ivdefs \\ \quad \textbf{operations}\ opdefs \\ \textbf{end}\ A\end{array}\right]\!\!\right]_{\textbf{vdm-pp}} \mathrel{\widehat{=}} \left(\begin{array}{l}\textbf{class}\ A\ \textbf{extends}\ P_1 \cdots P_n\ ; \\ \textbf{class}\ A\text{-}stat\ ; \\ [\![tydefs]\!]^A_{\textbf{tyd}}\ ;\ [\![valdefs]\!]^A_{\textbf{vld}}\ ; \\ [\![fundefs]\!]^A_{\textbf{fnd}}\ ;\ [\![ivdefs]\!]^A_{\textbf{ivd}}\ ; \\ [\![opdefs]\!]^{A,[\![ivdefs]\!]_{\textbf{asn}}}_{\textbf{opd}}\ ; \\ \textbf{meth}\ A\text{-}stat\ A\text{-}stat = \\ \quad \textbf{res}\ r : () \bullet [\![ivdecls]\!]_{\textbf{sasn}}\ ; \\ A := \textbf{new}A\text{-}stat()\end{array}\right)$$

5.2.2 Instance Variables

Instance variables are represented as attributes (**att**) in the theory. Class variables are assigned to the corresponding static class *A-stat*. Invariants are mapped to the invariant command. A sequence of attribute definitions is simply sequentially composed.

$$\begin{aligned}[\![x : T[:= e]]\!]^A_{\textbf{ivd}} &\mathrel{\widehat{=}} \textbf{att}\ A\ x : T \\ [\![\textbf{static}\ x : T[:= e]]\!]_{\textbf{ivd}} &\mathrel{\widehat{=}} \textbf{att}\ A\text{-}stat\ x : T \\ [\![\textbf{inv}\ P]\!]^A_{\textbf{ivd}} &\mathrel{\widehat{=}} \textbf{invar}\ A\ [\![P]\!]_{\textbf{exp}} \\ [\![V_1\ ;\ V_2]\!]^A_{\textbf{ivd}} &\mathrel{\widehat{=}} [\![V_1]\!]^A_{\textbf{ivd}}\ ;\ [\![V_2]\!]^A_{\textbf{ivd}}\end{aligned}$$

Default assignments for instance variables are accumulated into an assignment predicate, that is used to augment the default constructor. That is to say, in this semantics default assignments are really considered as the creation of a default constructor. Only instance variables are considered here (they map to skip), with default assignments to class variables handled separately. Instance variables with no default values also yield a skip in the constructor.

$$\begin{aligned}[\![x : T]\!]^A_{\textbf{asn}} &\mathrel{\widehat{=}} \mathit{II} \\ [\![x : T := e]\!]^A_{\textbf{asn}} &\mathrel{\widehat{=}} x := e \\ [\![\textbf{static}\ x : T := e]\!]_{\textbf{asn}} &\mathrel{\widehat{=}} \mathit{II} \\ [\![\textbf{inv}\ P]\!]^A_{\textbf{asn}} &\mathrel{\widehat{=}} \mathit{II} \\ [\![V_1\ ;\ V_2]\!]^A_{\textbf{asn}} &\mathrel{\widehat{=}} [\![V_1]\!]^A_{\textbf{asn}}\ ;\ [\![V_2]\!]^A_{\textbf{asn}}\end{aligned}$$

5.2.3 Implicit and Explicit Operations

Both kinds of operation are interpreted as methods whose bodies are UTP designs. The designs assume the precondition and invariant holds, and commits to establishing the post condition and preserving the invariant. An explicit operation additionally encapsulates the semantics of the operation body in the post condition. Therefore, an explicit operation whose body does not satisfy the postcondition or violates the invariant is equivalent to the miraculous design $\top_D$. This is a strong interpretation of the use of invariants and postconditions in the semantics, and

in practical terms it may be unnecessarily restrictive. For example, we could alternatively prove that the operation satisfies its obligations, rather than asserting that it does. Nevertheless, since this is a safe use of the assertional formulas we retain the strong semantics.

$$\left[\!\!\left[\begin{array}{l} opn(x_1 : T_1 \ldots x_n : T_n)\ r : U \\ \textbf{pre}\ Pre\ \textbf{post}\ Post \end{array} \right]\!\!\right]^{C,asn}_{\textbf{opd}} \ \widehat{=}\ \textbf{meth}\ C\ opn \left(\begin{array}{l} \textbf{res}\ r : U\ ;\ \textbf{val}\ x_1 : T_1 \cdots x_n : T_n \\ \quad \bullet \left(\begin{array}{l} [\![Pre]\!]_{\textbf{exp}} \\ \wedge\, \mathcal{I}(C) \end{array} \right) \vdash \left(\begin{array}{l} [\![Post]\!]_{\textbf{exp}} \\ \wedge\, \mathcal{I}(C)' \end{array} \right) \end{array} \right)$$

provided $opn \neq C$

$$\left[\!\!\left[\begin{array}{l} opn : (T_1 \times T_n) \Longrightarrow U \\ opn(x_1 \ldots x_n) \ == \ Body \\ \textbf{pre}\ Pre\ \textbf{post}\ Post \end{array} \right]\!\!\right]^{C,asn}_{\textbf{opd}} \ \widehat{=}\ \textbf{meth}\ C\ opn \left(\begin{array}{l} \textbf{res}\ RESULT : U\ ;\ \textbf{val}\ x_1 : T_1 \cdots x_n : T_n \\ \quad \bullet \left(\begin{array}{l} [\![Pre]\!]_{\textbf{exp}} \\ \wedge\, \mathcal{I}(C) \end{array} \right) \vdash \left(\begin{array}{l} [\![Body]\!]_{\textbf{prg}} \wedge \\ [\![Post]\!]_{\textbf{exp}} \\ \wedge\, \mathcal{I}(C)' \end{array} \right) \end{array} \right)$$

provided $opn \neq C$

5.2.4 Constructors

A constructor is a special class of method whose name matches that of the encapsulating class. Constructors, like regular operations, are given semantics as designs. Constructors in VDM-RT are always explicit operations, and do not have pre or postconditions; thus the constructor design has assumption ***true***. The commitment assigns default values to variables using the accumulated *asn*, and then executes the constructor body. Additionally, we assume that each constructor must establish the invariant (though it need not hold initially); a constructor which does not establish the invariant is miraculous ($\top_D$). The definition of a sequence of operations yields a sequential composition.

$$\left[\!\!\left[\begin{array}{l} C : (T_1 \times T_n) \Longrightarrow C \\ C(x_1 \ldots x_n) \ == \ Body \end{array} \right]\!\!\right]^{C,asn}_{\textbf{opd}} \ \widehat{=}\ \textbf{meth}\ C\ C \left(\begin{array}{l} \textbf{res}\ RESULT : C\ ;\ \textbf{val}\ x_1 : T_1 \cdots x_n : T_n \\ \quad \bullet\ \textbf{\textit{true}} \vdash (asn\ ;\ [\![Body]\!]_{\textbf{prg}}) \wedge \mathcal{I}(C)' \end{array} \right)$$

$$[\![Op_1\ ;\ Op_2]\!]^{A,asn}_{\textbf{opd}} \ \widehat{=}\ [\![Op_1]\!]^{A,asn}_{\textbf{opd}}\ ;\ [\![Op_2]\!]^{A,asn}_{\textbf{opd}}$$

6 Mechanisation of Objects and Classes

In this section we consider an experimental mechanisation of the theory of objects in our Isabelle-based theorem prover, *Isabelle/UTP*. The section has been type set and processed using Isabelle's automatic document preparation system. Thus all definitions herein have been verified by Isabelle.

A major question to be answered by a semantic embedding of the theory of objects is how should the definitional constructs of classes, attributes, and methods be handled in the theorem prover. Our theory in Section 4 represents a collection of declarations as a UTP predicate that assigns the constructs to variables. In *Isabelle/UTP* we have a semantic embedding of the UTP predicate model and therefore we could use this to provide a target for class declarations. Indeed, if we wish to prove that our class constructs satisfy the healthiness conditions this is

what we must do. This requires though that we can give an account to higher order predicates in our mechanisation to allow method definitions.

Giving a full account to higher order predicates in *Isabelle/UTP* is a non-trivial problem because it requires a self-injective universe of predicates. Our model of predicates in *Isabelle/UTP* is binding sets, that is $Pred = \mathbb{P}(\mathcal{V} \to \mathbb{U})$ sets of functions from variables ($\mathcal{V}$) to values in some universe $\mathbb{U}$. In order to have predicates assigned to variables, we require that $Pred \subseteq \mathbb{U}$, but such a construction is generally impossible in HOL as it would invoke Russell's paradox. A potential solution to this problem lies in axiomatic extension to Isabelle that carefully constructs such as universe that supports only limited self-injection. An initial attempt at such an extension in Isabelle has been developed by Zeyda[7] and appears promising. However, if we wish to remaining purely definitional, we need another solution.

For our implementation, we follow an alternative shallower approach of using Isabelle's own definitional mechanisms for the declaration of classes and methods. Thus the observational variables like *cls*, *atts*, and method definitions *m* rather than being UTP variables, instead become variables of the Isabelle theory state that can be manipulated by ML code. We therefore avoid some of the problems of Higher Order UTP, although limit ourselves to constructions possible directly in Isabelle. The implementation considers a fragment of the language we defined in Section 4. Specifically we implement the procedure and object manipulation constructs, and not the declarative constructs, which can be implemented through ML procedures.

We also implement a form of dynamic dispatch through the use of Isabelle's own type system. Specfically, dynamic dispatch is achieved through the creation of HOL's polymorphic constants that enable the selection of an appropriate method implementation at runtime. Moreover we provide direct support for parameter overloading through the type system – this is strictly speaking unnecessary and could also be implemented through method renaming (though using the type system achieves a more direct translation). The implementation here is thus a relatively shallow embedding of UTP classes, and means we cannot prove much about the dispatch mechanism since it is core to HOL. However, we can most likely use this for theorem proving, whilst retaining the advantages of a deep predicate model. Nevertheless, we recognise that achieving an embedding that provides both the fidelity of a deep embedding and the verification utility of a shallow embedding remains future work.

Objects are modelled using HOL records, and classes as record types. Class methods are simply UTP procedures. We have already implemented the syntax of UTP procedues in *Isabelle/UTP*, thus we can write definitions like this:

definition *prc* :: $(nat \times (nat, 'a)\ pvar, 'a)\ uproc$ **where**
$prc = ``val\ x : nat, res\ y : nat \bullet y := \$x + \ll 5 \gg ``$

It represents a simple procedure taking a value parameter x of type *nat*, and a result parameter y of type *nat* that is assigned the value $x + 5$ in the body. Procedures are characterised by the type $('par, 'm)\ uproc$, for some parameter type $'par$ model type $'m$. For *prc* the parameter type pairs a natural number with a variable of the natural number type, the latter being the result variable. Additionally we provide syntax for the `return` statement in the form of *ReturnA* which assigns to a distinguished result parameter called `RESULT` the given value. For example we could rewrite the above procedure with a return as

definition *prc'* :: $(nat \times (nat, 'a)\ pvar, 'a)\ uproc$ **where**

[7] https://github.com/isabelle-utp/utp-main/blob/axiomatic/theories/theories/thiago.thy

$prc' = ``val\ x : nat, res\ RESULT : nat \bullet return\ (\$x + \ll 5 \gg)``$

Due to the shallow nature of this model, we cannot directly implement the healthiness conditions given in Section 4, as these would need to operate at the level of the Isabelle definitional mechanisms rather than UTP predicates. Specifically, we here rather than creating a UTP predicate consisting of the definitions, we apply the definitions directly to the HOL proof state – it is this which makes our implementation essentially shallow. For similar reasons, we also cannot handle recursive method definitions. Theoretical support for rescursive methods has already been accomplished in a new UTP theory of methods in [47]. However, this approach uses Higher-Order UTP predicates rather than Isabelle constants to formalise method definitions. Initial investigation seems to conclude that this kind of machinery is beyond a purely definitional implementation and may require axiomatic extensions to Isabelle/HOL, which we are exploring separately.

type-synonym $('cl, 'inp, 'out, 'm)\ umeth$
$= (('cl, 'm)\ pvar * ('out, 'm)\ pvar * 'inp, 'm)\ uproc$

We introduce our type of methods *umeth* as an Isabelle type synonym. It is a parametric type with four parameters:

- *cl* represents the class type, and will usually be the Isabelle record type containing the classes (and objects) state space;
- *inp* represents the input type of the method, which will usually be a tuple of types;
- *out* represents the output type of the method, which may be the unit type *unit* if no value is returned;
- *m* is the value model of Isabelle/UTP.

The representation of a method is then simply a procedure (*uproc*) where the parameters consist of a tuple with three elements:

- a variable with the class type, which records the target of a method call;
- a variable with the output type, which records the result;
- the input type.

Next we introduce a special class of methods that represent class constructors.

consts $UCONSTRUCT :: ('cl, 'inp, unit, 'm)\ umeth$

Class constructors are methods that are called upon object creation. We represent the constructor implementation table as a polymorphic constant *UCONSTRUCT* with a method type returning no value. This can then be overridden for each new class created, and each new input combination.

With the core types defined, we can next proceed to define the core constructs of object oriented programs, namely the `new` and method call commands. For this mechanisation we do not handle object expressions as given in 4.3, but rather represent the constructs directly as commands.

definition *NewA* ::
$'m :: TYPED\text{-}MODEL\ alpha \Rightarrow 'class\ itself \Rightarrow ('class, 'm)\ pvar \Rightarrow 'm\ uapred$
where $NewA\ A\ c\ x = AssignA\ x\ A\ (LitPE\ (undefined :: 'class))$

The basic new construct, *NewA*, takes an alphabet, class type, and variable to assign the new class to. There is no constructor to call as such, so the meaning of this statement is simply the assignment of an arbitrary literal value of type *class* selected by the *undefined* construct.

definition *MethCallA* ::
$'m :: TYPED\text{-}MODEL\ alpha \Rightarrow ('cl, 'm)\ pvar \Rightarrow ('cl, 'inp, 'out, 'm)\ umeth$
$\Rightarrow ('inp, 'm)\ pexpr \Rightarrow ('out, 'm)\ pvar \Rightarrow 'm\ uapred$ **where**
$MethCallA\ A\ obj\ f\ v\ x = ProcCallA\ f\ (ProdPE\ (LitPE\ obj)\ (ProdPE\ (LitPE\ x)\ v))$

The method call command, *MethCallA*, takes an alphabet, object variable, method, an expression of the input type, and a variable of the output type. It constructs a product (using *ProdPE*) of the input object, input value, and output variable which to which the method is applied using the procedure call operator *ProcCallA*. The latter simply evaluates the passed parameter expression and feeds it through the lambda term representing the method.

definition *NewConsA* ::
$'m :: TYPED\text{-}MODEL\ alpha \Rightarrow 'class\ itself \Rightarrow ('class, 'm)\ pvar$
$\Rightarrow ('inp, 'm)\ pexpr \Rightarrow 'm\ uapred$
where $NewConsA\ A\ c\ obj\ inp = (NewA\ A\ c\ obj\ ;_{\alpha}\ MethCallA\ A\ obj\ UCONSTRUCT\ inp\ undefined)$

Construction of a new class via a constructor is achieved with the *NewConsA* command. It takes an alphabet, the class type to be constructed, a variable where the new object should be stored, and an expression representing the input to the constructor. Internally, this command constructs a new (arbitrary) object instance using *NewA* and then calls the *UCONSTRUCT* method on the new object which executes an appropriate constructor (if one exists).

We now give a short example to illustrate how classes and object can be constructed in this framework, with the help of two classes: *Date* and *Person* that we create. The *date* class consists of three fields representing the day, month, and year, associated accessor methods, and constructors as indicated below.

```
class Date
  instance variables
    day   : nat
    month : nat
    year  : nat
  operations
    getDay : () ==> nat
    getDay() == return day

    getMonth : () ==> nat
    getMonth() == return month

    getYear : () ==> nat
    getYear() == return year

    Date : () ==> Date
    Date() == day := 1; month := 1; year := 1970

    Date : (nat * nat * nat) ==> Date
    Date(d, m, y) == day := d; month := m; year := y
end
```

The Person class consists of three fields for the forename, surname, and date of birth.

```
class Person
  instance variables
    forename    : seq of char
    surname     : seq of char
    dateOfBirth : Date
  operations
    Person : () ==> Person
    Person = surname := ""; forename := ""; dateOfBirth := new Date()
```

In Isabelle we represent the Date class as the following record:

record *date* =
day :: *nat*
month :: *nat*
year :: *nat*

This record introduces a new unique Isabelle type that will represent the class, and all objects in the class will have this type. Next we define three accessor methods for these attributes.

consts
getDay :: ('*cl*, '*inp*, '*out*, '*m*) *umeth*
getMonth :: ('*cl*, '*inp*, '*out*, '*m*) *umeth*
getYear :: ('*cl*, '*inp*, '*out*, '*m*) *umeth*

As for *UCONSTRUCT* we define them all as polymorphic constants. At this level the types of these constants are completely unconstrained, as they can be implemented for any class, and simply act as method names rather than implementations. We next define two implementations of constructors for the *date* class.

definition
date-constructor-1 =
‘‘*vres self* : *date*, *res RESULT* : *unit*, *val dummy* : *unit*
· *self*•*day* := *1*
; *self*•*month* := *1*
; *self*•*year* := *1970*‘‘

The first implementation, *date-constructor-1*, is the default constructor which takes no inputs and produces no outputs. It takes a date object parameter *self*, a result parameter (which is nullary), and an input parameter called *dummy* (which is also nullarly as the method has no input). The body assigns the values 1, 1, and 1970 to the three object attributes.

definition
date-constructor-2 =
‘‘*vres self* : *date*, *res RESULT* : *unit*, *val day* : *nat*,
val month : *nat*, *val year* : *nat*
· *self*•*day* := $*day*; *self*•*month* := $*month*; *self*•*year* := $*year*‘‘

The second implementation, *date-constructor-2*, takes three explicit parameters to populate the three attributes.

defs (**overloaded**)
date-constructor-1-meth [*simp*]: *UCONSTRUCT* ≡ *date-constructor-1*

date-constructor-2-meth [*simp*]: *UCONSTRUCT* ≡ *date-constructor-2*

With two constructors defined we use Isabelle's overloading mechanism to provide constructor implementations for the *date* constructor in *UCONSTRUCT*. Thus when a new object of type *date* is constructed with one of the two parameter types, the appropriate implementation will be selected by the type system. We similarly define implementations of the three class methods. Unlike the constructors, these also have return types.

definition *date-getDay* = ‘‘*vres self* : *date*, *res RESULT* : *nat*, *val dummy* : *unit*
· *return*(\$*self*•*day*)‘‘

definition *date-getMonth* = ‘‘*vres self* : *date*, *res RESULT* : *nat*, *val dummy* : *unit*
· *return*(\$*self*•*month*)‘‘

definition *date-getYear* = ‘‘*vres self* : *date*, *res RESULT* : *nat*, *val dummy* : *unit*
· *return*(\$*self*•*year*)‘‘

defs (**overloaded**)
date-getDay-meth [*simp*]: *getDay* ≡ *date-getDay*
date-getMonth-meth [*simp*]: *getMonth* ≡ *date-getMonth*
date-getYear-meth [*simp*]: *getYear* ≡ *date-getYear*

We show below how this class can then be used in a UTP program. We first use the first constructor to build a new date type, and apply the `getDay` method to it (which in this case will yield 1), assigning its return value to a local variable `y`.

term ‘‘*dcl y* : *nat* · *new*{*x*:*TYPE*(*date*)}() ; *y* := *x*•*getDay*()‘‘

In the second example we provide three explicit inputs to the constructor, which will therefore cause the second constructor to be selected.

term ‘‘*dcl y* : *nat* · *new*{*x*:*TYPE*(*date*)}(≪*25* :: *nat*≫, ≪*08* :: *nat*≫, ≪*1983* :: *nat*≫) ; *y* := *x*•*getDay*()‘‘

Finally, we show how the *date* class can be used in the context of another class call *person*. We first create a new record type, which has a `dateOfBirth` attribute.

record *person* =
surname :: *string*
forename :: *string*
dateOfBirth :: *date*

We then produce a default constructor for the person class, which sets both the surname and forename of the person to an empty string, constructs a new date object, and finally assigns this to `dateOfBirth`.

definition
person-constructor-1 =
‘‘*res self* : *person*, *res RESULT* : *unit*, *val dummy* : *unit* ·
self•*surname* := ‘‘‘‘; *self*•*forename* := ‘‘‘‘;
(*dcl dob* : *date* · *new*{*dob*:*TYPE*(*date*)}() ; *self*•*dateOfBirth* := \$*dob*)‘‘

defs (**overloaded**)
person-constructor-1-meth [*simp*]: *UCONSTRUCT* ≡ *person-constructor-1*

When the date constructor is invoked the Isabelle type system will detect that we give it the input type *unit* and will therefore select implementation 1 for the constructor. We also likewise overload *UCONSTRUCT* with new definition for the person constructor.

Thus we have shown how classes can be represented in Isabelle. Such class definitions for Date and Person could in the future we generated by suitable ML code in Isabelle. We would then have the ability to start to prove theorems about VDM-RT classes. For now though, this is also left as future work.

7 Conclusion

In this deliverable we have summarised the initial work towards creation of a novel semantics for VDM-RT. We described how object-orientation can be handled through a UTP theory of classes and methods, including novel healthiness conditions that allow handling of multiple inheritance. We then gave a translation from VDM-RT classes into this calculus that includes a new approach to handling of static attributes, methods, and constructors. We also provided an Isabelle proof-of-concept for some of key object-oriented constructs in the UTP. In addition we also mechanised the existing operational semantics in appendix A.1 that has allowed us to gain greater insight into the language. Moreover, our own UTP theory effectively elaborates the flat class state-space of the former semantics with inheritance.

The next deliverable in this series, Deliverable D2.2b, will accompany the work described herein by providing a semantics for real-time and interruption. We outlined in this deliverable a proposed approach using *CircusTime* or *CML* in section 3 that we will investigate further. In particular we will show how VDM-RT threads and their concurrent behaviour can be modelled using a variant of *CircusTime*/*CML* processes. This will ensure that VDM-RT has a well-founded semantics, and also allow us to begin to formal link VDM-RT to the other notations in this project, using our *lingua franca* language, *INTO-CSP*. This semantics will allow us to ensure the INTO-CPS notations can be soundly integrated, enabling a well-founded tool-chain where evidence can be gathered from a variety of different components. We previously demonstrated this approach in our implementation of the *Symphony* tool[8] for modelling systems of systems, that allowed us to build a suite of tools with the common foundation of the UTP [14]. For example, our mechanisation in Isabelle could provide a way forward for building a program verifier for VDM-RT controllers.

Appendices

A Mechanised Operational Semantics

In this appendix we provide a partial mechanisation of the operational semantics of VDM-RT from [28] in Isabelle. This is not a UTP semantics, which is our eventual aim, but rather a direct implementation using Isabelle datatypes and inductive sets. This then gives us a baseline

[8] http://symphonytool.org/

mechanised semantics with which to guide future progress. Additionally the mechanisation highlighted a few small inconsistencies in [28] that we have now fixed. Though we have not verified the operational semantics, it has passed Isabelle's type checker and is therefore well-formed.

Transition rules in the mechanised operational semantics are written using Isabelle's notation for inductive predicate clauses. A typical law will have the form $[\![A;\ B;\ C]\!] \Longrightarrow D$ where A, B, and C are assumptions of the rule, and D is the consequent, in sequent calculus style.

A.1 VDM-RT State Types

We describe the types needed to describe the state of a VDM-RT model, in terms of CPUs, threads, busses, statements, objects, message and so on. The majority of the types are adapted from [28] but modified for Isabelle/HOL, for example by the replacement of union types with disjoint unions and algebraic datatypes. Moreover the names of primitives and types differ slightly and we have made a number of simplifications. Datatypes not directly relevant for the semantics (such as patterns) are omitted from the presentation, though present in the Isabelle theories. We also specify some basic types like *VDMValue* and *VDMExpr*, and functions like the VDM-SL typing relation $x :_v t$ that are omitted. A number of identifier types are also defined, including:

- *cla-id* – class identifier
- *obj-id* – object identifier
- *op-id* – operation identifier
- *fun-id* – function identifier
- *thr-id* – thread identifier
- *bus-id* – bus identifier
- *var-id* – variable identifier

We first describe bindings Σ, that are partial functions mapping variables identifiers to VDM values. They are used widely in the semantics to represent state spaces, let definitions, and the result of executing pattern matches.

type-synonym $\Sigma = \textit{var-id} \rightharpoonup \textit{VDMValue}$

Next we defined pending updates to a state, which are partial mappings from object identifiers to bindings. Each write within an object does not update the global CPU state until specific synchronisation times. Until then, they are stored in a pending buffer.

type-synonym $\textit{Pending} = \textit{obj-id} \rightharpoonup \Sigma$

An assignment associated variable identifier, or object and variable identifier, with an expression.

record *Assignment* =
 asntarget :: *var-id* + (*obj-id* × *var-id*)
 asnexp :: *VDMExpr*

record *CallContext* =

ccpending :: *Pending*
ccstate :: Σ
ccpost :: *VDMExpr option*

type-synonym *Definition* = *var-id* $*$ *VDMExpr*

The statement type specifies all the different kind of statements that can be constructed in VDM-RT.

datatype *Stm*
= *Skip* A null statement
| *Assignment Assignment* Assign an expression to a variable
| *Atomic Assignment list* A list of atomic assignments
| *AsyncCall* (*name*: (*obj-id* $\times$ *op-id*) + (*cpu-id* $\times$ *obj-id* $\times$ *op-id*))
(*args*: *VDMExpr list*)
Asychronous call, consisting of a method identifier and arguments
| *SyncCall* (*ctarget*: (*var-id* + (*cpu-id* $\times$ *thr-id*)) *option*)
(*name*: (*obj-id* $\times$ *op-id*) + (*cpu-id* $\times$ *obj-id* $\times$ *op-id*))
(*args*: *VDMExpr list*)
Sychronous call
| *Cases* (*exp*: *VDMExpr*) (*cases*: (*Pattern* $\times$ *Stm*) *list*) *Stm option*
Case statement
| *Cycles* (*cycles*: *VDMExpr*) (*body*: *Stm*)
Cycles statement
| *DurOrPDur* (*durOf*: *DurOrPDur*)
Either a duration statement or a (semantic) partial duration statement
| *ForIndex* (*var*: *var-id*) (*fromind*: *VDMExpr*) (*toind*: *VDMExpr*)
(*byind*: *VDMExpr*) (*body*: *Stm*)
For loop over an incremented index interval
| *ForSet* (*pattern*: *Pattern*) (*setExp*: *VDMExpr*) (*body*: *Stm*)
For loop over a set
| *ForSeq* (*pattern*: *Pattern*) (*setExp*: *VDMExpr*) (*body*: *Stm*)
For loop over a sequence
| *If* (*exp*: *VDMExpr*) (*thenb*: *Stm*) (*elseb*: *Stm*)
If-then-else statement
| *LetBe* (*bind*: *Bind*) (*suchThat*: *VDMExpr*) (*body*: *Stm*)
Let-be-such-that statement
| *LetDef* (*localDefs*: *Definition list*) (*body*: *Stm*)
Local let definition of a list of expressions to variables
| *New* (*nclass*: *cla-id*) (*target*: *var-id*)
New statement: create a new object from the given class
| *Return* (*rexp*: *VDMExpr option*)
Return statement
| *PartialLetDef* (*ctx*: Σ) (*plocalDefs*: *Definition list*) (*pbody*: *Stm*)
Partial let statement (semantics only). Gives the evaluated variable valuations so far together with the remaining unevaluated definitions.
| *ObjectContext* (*object*: *obj-id*) (*objbody*: *Stm*) (*callctx*: *CallContext*)
Object context for a statement (semantic only)
| *Wait* (*wtarget*: (*var-id* + (*cpu-id* $\times$ *thr-id*)) *option*)
| *PartialAtomic Assignment list obj-id set*
A collection of atomic assignments
| *SimpleBlock* (*sbody*: *Stm list*)

A block consisting of a sequence of statements
| *Start* (*obj*: *obj-id*)
Instruction to start an object thread running
| *While* (*exp*: *VDMExpr*) (*body*: *Stm*)
While looop
and *DurOrPDur* = *Dur* (*duration*: *DurTime*) (*dbody*: *Stm list*)
| *PDur* (*duration*: *DurTime*) (*elapsed*: *Time*) (*dbody*: *Stm list*)

The duration statement body consists either of a complete duration statement with a specified time and list of statements, or else a partially executed duration statement that is used only in the operational semantics. The duration can either be *ExecTime*, meaning the statement takes the duration is the calculated statement duration, or else *ExpTime e* meaning the duration is overriden by the specified time expression *e*.

Function definitions consist of a list of argument declarations, a list of returns types, a body expression, a precondition expression and a postcondition expression.

record *Fun* =
args :: (*var-id* × *VDMType*) *list*
ret :: *VDMType list*
body :: *VDMExpr*
pre :: *VDMExpr*
post :: *VDMExpr*

Operation definitions consist of an asynchronous flag, a list of argument declarations, a list of return types, a body consist of a list of statements wrapped by duration statements, a precondition, a postcondition, and a measure function (to aid in showing termination of a recursive operation).

record *Op* =
async :: *bool*
args :: (*var-id* × *VDMType*) *list*
ret :: *VDMType list*
body :: *DurOrPDur list*
pre :: *VDMExpr option*
post :: *VDMExpr option*
measure :: *Fun*

A periodic thread specifies the operation that should be executed, the period of time between each call, the clock jitter, clock delay, and offset.

record *Periodic* =
op :: *op-id*
period :: *VDMExpr*
jitter :: *VDMExpr*
delay :: *VDMExpr*
offset :: *VDMExpr*

A VDM-RT class consists of a set of superclasses, a set of constant values specified by the class, a set of variables specifying types and initial values, a set of named operations, a set of invariants over the instance variables, and initial behaviour which can either be a list of duration statements (a main operation), or a periodic thread.

record *Class* =

parents :: *cla-id set*
clvalues :: Σ
vars :: *var-id* $\rightharpoonup$ *VDMType* $\times$ *VDMExpr*
ops :: *op-id* $\rightharpoonup$ *Op*
funs :: *fun-id* $\rightharpoonup$ *Fun*
invs :: *Fun set*
initial :: (*DurOrPDur list*) + *Periodic*

An object consists of the class (to which the object belongs), a state binding giving values to each of the instance variables, and a periodic countdown that determines when the next execution of the associated periodic thread should take place.

record *Object* =
objclass :: *cla-id*
objstate :: Σ
periodicCountdown :: *Time option*

A call message consists of the object identifier, the operation identifier, the list of arguments, an optional thread to which control should be returned after the completion of the operation (in the case of a synchronous call) and the time stamp of when the message was sent.

record *CMessage* =
obj :: *obj-id*
oper :: *op-id*
args :: *VDMValue list*
replyto :: (*cpu-id* $\times$ *thr-id*) *option*
sendTime :: *Time*

A return message consists of the list of values being returned, the thread to which control will be returned to, and a time-stamp.

record *RMessage* =
rvalue :: *VDMValue list*
replyto :: *cpu-id* $\times$ *thr-id*
sendTime :: *Time*

A thread consists of its status, a collection of pending writes associated with the thread, an object context within which the thread runs, and a body giving the statements to be executed by the thread. A thread can be in one of five states: running, runnable, waiting, pending, or completed.

datatype *ThreadStatus* = *Running* | *Runnable* | *Waiting* | *ThrPending* | *Completed*

record *Thread* =
status :: *ThreadStatus*
pending :: *Pending*
tcontext :: *obj-id*
tbody :: *Stm list*

A bus consists of a set of CPUs that the bus connects, a speed indicating how fast the bus forwards messages with respect to the global clock, and a queue of messages awaiting delivery to one of the connected CPUs.

record *Bus* =

cpus :: *cpu-id set*
speed :: *nat*
queue :: (*cpu-id* × (*CMessage* + *RMessage*)) *list*

A CPU consists of the set of objects that is running on the CPU, the set of associated threads, and the speed of the CPU relative to the global clock.

record *CPU* =
objects :: *obj-id* ⇀ *Object*
threads :: *thr-id* ⇀ *Thread*
speed :: *nat*

With all the main components of different parts of the VDM-RT state specified, we now complete the definition by giving the overall state in terms of the CPUs, Busses, Classes, and present time.

type-synonym *CPUs* = *cpu-id* ⇀ *CPU*
type-synonym *Busses* = *bus-id* ⇀ *Bus*
type-synonym *Classes* = *cla-id* ⇀ *Class*

record *VDMRT* =
cpus :: *CPUs*
busses :: *Busses*
time :: *Time*
classes :: *Classes*

A.2 VDM Timed Expressions

A VDM-RT expression has a time penalty associated with it, defining how much time evaluation of the expression takes. The following Isabelle locale gives a context for evaluation of timed expressions.

locale *VDMRT-Context* =
fixes
SkipTime IfTime WhileTime CasesTime NewTime ForIndexTime ForSeqTime
ForSetTime LetDefTime LetBeTime LocalAssignmentTime RemoteAssignmentTime
AtomicTime StartTime LocalSyncCallTime LocalAsyncCallTime RemoteSyncCallTime
RemoteAsyncCallTime :: *Time*
begin

The locale specifies a fixed constant for each of the statements which specifies its default time penalty. We also specify a (partial) evaluation function for timed expressions. It takes a context consisting of the current time, the set of classes, set of CPUs, set of local pending variables values, and the object context. Given a context and a VDM expression, the evaluation function yields a (optional) value and new time.

fun *evalVDMRTExpr* ::
Time × *Classes* × *CPUs* × *Pending* × *obj-id* ⇒
VDMExpr ⇒ (*VDMValue* × *Time*) *option* (- ⊢ ⟦-⟧ [*100*,*0*] *100*) **where**
(τ, *C*, *N*, *P*, *ob*) ⊢ ⟦*VarE x*⟧ = *do* { Γ <− *P ob*; *v* <− Γ(*x*); *Some* (*v*, *0*) } |
(τ, *C*, *N*, *P*, *ob*) ⊢ ⟦*LitE v*⟧ = *Some* (*v*, *0*) |

$$
\begin{array}{l}
(\tau, C, N, P, ob) \vdash [\![BinE\ b\ e\ f]\!] = \\
\ do\ \{\ (u, t1) <- (\tau, C, N, P, ob) \vdash [\![e]\!] \\
\quad ;\ (v, t2) <- (\tau, C, N, P, ob) \vdash [\![f]\!] \\
\quad ;\ if\ (b \in ArithOp) \\
\quad\ then\ do\ \{\ x <- numOf(u);\ y <- numOf(v);\ n <- numBin\ b\ (x, y) \\
\qquad\quad ;\ Some\ (NumV\ n, t1 + t2)\ \} \\
\quad\ else \\
\quad\ if\ (b \in NumCmpOp) \\
\quad\ then\ do\ \{\ x <- numOf(u);\ y <- numOf(v);\ n <- numCmp\ b\ (x, y) \\
\qquad\quad ;\ Some\ (BoolV\ n, t1 + t2)\ \} \\
\quad\ else \\
\quad\ if\ (b \in LogOp) \\
\quad\ then\ do\ \{\ x <- boolOf(u);\ y <- boolOf(v);\ n <- logOp\ b\ (x, y) \\
\qquad\quad ;\ Some\ (BoolV\ n, t1 + t2)\ \} \\
\quad\ else\ None\ \} \mid \\
(\tau, C, N, P, ob) \vdash [\![ThisE]\!] = Some\ (ObjRefV\ ob, 0) \mid \\
(\tau, C, N, P, ob) \vdash [\![TimeE]\!] = Some\ (NumV\ \tau, 0)
\end{array}
$$

A.3 VDM-RT Operational Semantics

The VDM-RT operational semantics is specified as a collection of inductively defined sets that give the transition relation at different levels of abstraction. For example the set *vdmrt-stmt-rel* describes the operational semantics of individual statements in the context of an object, within a thread running on a particular CPU. We begin by entering the *VDMRT-Context* locale that gives the context for timed evaluation of statements and expressions. Each inductive set takes a similar form of $\Gamma \vdash p \rightarrow q$ where Γ is a set of contextual elements (such as the set of CPUs of the system, the set of objects running, the current time and so on), p is the element which is being transitioned, and q is the result of applying the transition rule.

context *VDMRT-Context*
begin

The operational semantic rules are defined in a different order to [28]; this is simply because in Isabelle we must declare elements in order of their dependencies. We first describe the transition relation for evaluation of bindings, such as set bindings or type bindings. The transition relation has a context, and shows how a bind, together with a set of pending updates and a CPU context, should map to a pattern and binding associated with that pattern. Additionally, we also produce an updated time since evaluation of a set bind contains and expression and can thus exert a time penalty.

inductive *vdmrt-bind-rel* ::
$(Time \times Classes \times CPUs \times cpu\text{-}id \times thr\text{-}id \times obj\text{-}id) \Rightarrow$
$(Bind \times Pending \times CPU) \Rightarrow$
$(Pattern\ list \times \Sigma \times Time) \Rightarrow$
$bool\ (\text{-} \vdash \text{-} -bind\rightarrow \text{-}\ [60,0,60]\ 60)$ **where**

Select a possible x of type ty and match the pattern against it using the *match* function.

Type-Bind:
$[\![\ length(p) = 1$

$$
\begin{array}{l}
\;;\; x :_t ty \\
\;;\; match(hd(p), x) = Some(\sigma') \\
[\![\Longrightarrow \\
(\tau, css, cs, c, t, ob) \vdash (Inr\ (TypeBind\ p\ ty), pdg, cpu)\ -bind\rightarrow\ (p, \sigma', 0)\ |
\end{array}
$$

Match over a list of patterns and merge the resulting binding.

Multi-Type-Bind:

$$
\begin{array}{l}
[\![length(ps) > 1 \\
\;;\; \sigma' = merge\ \{\sigma \mid \sigma\ p\ x.\ p \in set(ps) \wedge x :_t ty \wedge match(p,x) = Some\ \sigma\} \\
[\![\Longrightarrow \\
(\tau, css, cs, c, t, ob) \vdash (Inr\ (TypeBind\ ps\ ty), pdg, cpu)\ -bind\rightarrow\ (ps, \sigma', 0)\ |
\end{array}
$$

Evaluate the given set expression, select and element, and pattern match against it.

Set-Bind:

$$
\begin{array}{l}
[\![length(p) = 1 \\
\;;\; (\tau, css, cs, pdg, ob) \vdash [\![e]\!] = Some\ (SetV\ valueSet, \delta_e) \\
\;;\; x \in rcset\ valueSet \\
\;;\; match(hd(p), x) = Some(\sigma') \\
[\![\Longrightarrow \\
(\tau, css, cs, c, t, ob) \vdash (Inl\ (SetBind\ p\ e), pdg, cpu)\ -bind\rightarrow\ (p, \sigma', \delta_e)\ |
\end{array}
$$

Match over a list of patterns and merge the resulting binding.

Multi-Set-Bind:

$$
\begin{array}{l}
[\![length(ps) > 1 \\
\;;\; (\tau, css, cs, pdg, ob) \vdash [\![e]\!] = Some\ (SetV\ valueSet, \delta_e) \\
\;;\; \sigma' = merge\ \{\sigma \mid \sigma\ p\ x.\ p \in set(ps) \wedge x \in rcset(valueSet)\} \\
[\![\Longrightarrow \\
(\tau, css, cs, c, t, ob) \vdash (Inl\ (SetBind\ ps\ e), pdg, cpu)\ -bind\rightarrow\ (ps, \sigma', \delta_e)
\end{array}
$$

Next we describe the transition relation for statements, which is the largest of all the semantic definitions. It describes how a list of statements, together with pending rewrites, a CPU and collection of busses, is transformed through time. A statement is executed in the context of a particular CPU, thread, and object, that are identified in the context. The transition relation is inductive over the list of statements and provides a ”big-step” operational semantics. Each rule includes a line (usually second to last) that calculates the overall time penalty for the transition by summing up the constituent statement penalties.

inductive *vdmrt-stmt-rel* ::

$$
\begin{array}{l}
Time \times Classes \times CPUs \times cpu\text{-}id \times thr\text{-}id \times obj\text{-}id \Rightarrow \\
Stm\ list \times Pending \times CPU \times Busses \Rightarrow \\
Stm\ list \times Pending \times CPU \times Busses \times Time \Rightarrow \\
bool\ (\text{-} \vdash \text{-} -stmt\rightarrow \text{-}\ [60,0,60]\ 60)\ \textbf{where}
\end{array}
$$

The base case, an empty list of statements has no effect and zero time. In this rule τ represents the present time, *clss* the collection of classes, *cpss* the collection of CPUs, c the present CPU id, t the present thread id, and *ob* the present object id.

Stm-Base: $(\tau, clss, cpss, c, t, ob) \vdash ([], pdg, cpu, bss)\ -stmt\rightarrow\ ([], pdg, cpu, bss, 0)\ |$

If skip is at the head of the list of statements, calculate the transition for the remainder of the statements, and add to the resulting time penalty the skip penalty.

Stm-Skip:
$[\![(\tau, clss, cs, c, t, ob) \vdash (rest, pdg, cpu, bss) -stmt\rightarrow (rest', pdg, cpu, bss, \delta)$
$; \ \delta' = SkipTime + \delta \ ]\!] \Longrightarrow$
$(\tau, css, cs, c, t, ob) \vdash$
$(Skip \ \# \ rest, pdg, cpu, bss) -stmt\rightarrow (rest', pdg, cpu, bss, \delta') \ |$

The semantics of simple block simple unwraps the contents of the block and prepends it to the beginning of the statement list before calculating the latter's semantics.

Stm-SimpleBlock:
$[\![(\tau, css, cs, c, t, ob) \vdash$
$(stms \bullet rest, pdg', cpu, bss) -stmt\rightarrow (rest', pdg', cpu', bss', \delta) \ ]\!] \Longrightarrow$
$(\tau, css, cs, c, t, ob) \vdash$
$(SimpleBlock(stms) \ \# \ stms, pdg, cpu, bss) -stmt\rightarrow (rest', pdg', cpu', bss', \delta) \ |$

The semantics of if-then-else first evaluates the conditional expression. If it evaluates to true then the *then* branch is prepended to the statement list and the semantics of this is calculated. Otherwise the *else* branch is used. The overall time penalty for the statement is calculated by summing the conditional evaluation time, the remaining statement execution time, and the default time penalty in *IfTime* (that is provided by the enclosing locale).

Stmt-IfTrue:
$[\![(\tau, css, cs, pdg, ob) \vdash [\![e]\!] = Some \ (TrueV, \delta_e)$
$; \ (\tau, css, cs, c, t, ob)$
$\vdash (th \ \# \ rest, pdg, cpu, bss) -stmt\rightarrow (rest', pdg', cpu', bss', \delta)$
$; \ \delta' = \delta_e + \delta + IfTime \ ]\!] \Longrightarrow$
$(\tau, css, cs, c, t, ob) \vdash$
$(If \ e \ th \ el \ \# \ rest, pdg, cpu, bss) -stmt\rightarrow (rest', pdg', cpu', bss', \delta') \ |$
Stmt-IfFalse:
$[\![(\tau, css, cs, pdg, ob) \vdash [\![e]\!] = Some \ (FalseV, \delta_e)$
$; \ (\tau, css, cs, c, t, ob)$
$\vdash (el \ \# \ rest, pdg, cpu, bss) -stmt\rightarrow (rest', pdg', cpu', bss', \delta)$
$; \ \delta' = \delta_e + \delta + IfTime \ ]\!] \Longrightarrow$
$(\tau, css, cs, c, t, ob) \vdash$
$(If \ e \ th \ el \ \# \ rest, pdg, cpu, bss) -stmt\rightarrow (rest', pdg', cpu', bss', \delta') \ |$

The while loop functions much the same as if-then-else, except that the while body and a copy of the while statement is prepended to the statement list.

Stmt-WhileTrue:
$[\![(\tau, css, cs, pdg, ob) \vdash [\![e]\!] = Some \ (TrueV, \delta_e)$
$; \ stms = [bdy, While \ e \ bdy] \bullet rest$
$; \ (\tau, css, cs, c, t, ob)$
$\vdash (stms, pdg, cpu, bss) -stmt\rightarrow (rest', pdg', cpu', bss', \delta)$
$; \ \delta' = \delta_e + \delta + WhileTime \ ]\!] \Longrightarrow$
$(\tau, css, cs, c, t, ob) \vdash$
$(While \ e \ bdy \ \# \ rest, pdg, cpu, bss) -stmt\rightarrow (rest', pdg', cpu', busses', \delta') \ |$
Stmt-WhileFalse:

$$
\begin{array}{l}
[\![\ (\tau, css, cs, pdg, ob) \vdash [\![e]\!] = Some\ (FalseV, \delta_e) \\
\ ;\ (\tau, css, cs, c, t, ob) \vdash (rest, pdg, cpu, bss)\ {-stmt\rightarrow}\ (rest', pdg', cpu', bss', \delta) \\
\ ;\ \delta' = \delta_e + \delta + WhileTime\ ]\!] \Longrightarrow \\
(\tau, css, cs, c, t, ob) \vdash \\
(While\ e\ bdy\ \#\ rest, pdg, cpu, bss)\ {-stmt\rightarrow}\ (rest', pdg', cpu', busses', \delta')\ |
\end{array}
$$

The semantics of the case statement first evaluates the selection expresssion e. This is then used to select matching case patterns through a list comprehension. This (potentially empty) list is prepended to the list of other options, and a default skip option. The head of the resulting list is selected as the case branch that will be taken, resulting in a binding σ giving values to any case variables, and a statement *stm* to be executed. Finally, a partial let def is constructed for the new pattern variable definitions, and the semantics of the resulting expression is calculated. The overall time penalty is also calculated from the expression time and statement times.

Stmt-Cases:

$$
\begin{array}{l}
[\![\ (\tau, css, cs, pdg, ob) \vdash [\![e]\!] = Some\ (value, \delta_e) \\
\ ;\ alts = [(\sigma, stm).\ i \leftarrow [0..{<}length\ cass] \\
\qquad\qquad , (\exists\ p.\ cass!i = (p, stm) \\
\qquad\qquad\quad \wedge\ Some\ \sigma = match(p, value) \\
\qquad\qquad\quad \wedge\ \sigma \neq Map.empty)] \\
\qquad \bullet\ [(Map.empty, the(others)).\ others \neq None] \bullet [(Map.empty, Skip)] \\
\ ;\ (\sigma, stm) = hd(alts) \\
\ ;\ lt = PartialLetDef\ \sigma\ []\ stm \\
\ ;\ (\tau, css, cs, c, t, ob) \\
\qquad \vdash (lt\ \#\ rest, pdg, cpu, bss)\ {-stmt\rightarrow}\ (rest', pdg', cpu', bss', \delta) \\
\ ;\ \delta' = \delta_e + \delta + CasesTime \\
\ ]\!] \Longrightarrow \\
(\tau, css, cs, c, t, ob) \vdash \\
(Cases\ e\ cass\ others\ \#\ rest, pdg, cpu, bss)\ {-stmt\rightarrow}\ (rest', pdg', cpu', bss', \delta')\ |
\end{array}
$$

The semantics for creating a new object from class *cl* is somewhat complicated. The class *cl* must exist, and should not have periodic initial behaviour. A new object identifier *oid* is selected which must not already exist on the given CPU. Initial constant values and initial variable expressions can be calculated for the new object from the class *cl*. Each initial variable expression is then evaluated, resulting in a list of values and time penalties. The initial constant and variable values are then composed to produce σ, the initial state space for the object. The list of associated time penalties is then summed to produce the overall time penalty for variable initialisation. The new object *obje* is then constructed and the corresponding CPU is updated, adding this new object to the store associated with new object identifier. The map of pending variables is then also updated, with the object target variable *tgt* pointing to the new object reference. Finally the remainder semantics is calculated and overall time penalty produced. This includes δ_e that is calculated by summing up all the time penalties associated with evaluating the initial variable value expressions. The summing is performed by the builtin in HOL function *listsum* which adds together the elements of a list.

Stmt-New:

$$
\begin{array}{l}
[\![\ cl \in dom(css) \\
\ ;\ \neg\ isPeriodic(initial(the(css(cl)))) \\
\ ;\ oid \notin \bigcup\ \{dom\ (objects\ acpu)\ |\ acpu.\ acpu \in ran(cs) \cup \{cpu\}\} \\
\ ;\ initVals = clvalues\ (the(css(cl))) \\
\ ;\ initVars = vars\ (the(css(cl))) \\
\ ;\ inits = [i \mapsto (v, \delta_e)\ |\ i\ v\ \delta_e.\ i \in dom(initVars)
\end{array}
$$

$$\wedge\ (\tau, css, cs, pdg +\!\!+ [oid \mapsto initVals], oid)$$
$$\vdash [\![snd\ (the\ (initVars\ i))]\!] = Some\ (v, \delta_e)]$$
$$;\ \sigma = initVals +\!\!+ [i \mapsto v \mid i\ v.\ i \in dom(inits) \wedge Some\ (v, \text{-}) = inits(i)]$$
$$;\ \delta_e = listsum\ (map\ snd\ (sorted\text{-}list\text{-}of\text{-}map\ ((Some \circ snd) \circ_m inits)))$$
$$;\ obje = Object.make\ cl\ \sigma\ None$$
$$;\ cpu' = cpu(\!|objects := objects\ cpu +\!\!+ [oid \mapsto obje]|\!)$$
$$;\ pdg' = pdg +\!\!+ [ob \mapsto (the(pdg(ob)) +\!\!+ [tgt \mapsto ObjRefV\ oid])]$$
$$;\ (\tau, css, cs, c, t, ob) \vdash (rest, pdg', cpu', bss) \text{ -}stmt\rightarrow (rest', pdg'', cpu'', bss', \delta)$$
$$;\ \delta' = \delta + \delta_e + NewTime$$
$$]\!] \Longrightarrow$$
$$(\tau, css, cs, c, t, ob) \vdash$$
$$(New\ cl\ tgt\ \#\ rest, pdg, cpu, bss) \text{ -}stmt\rightarrow (rest', pending'', cpu'', busses', \delta')\ |$$

The next five rules deal with evaluation and execution of duration statements. The first rule takes care of evaluating a duration statement that contains an expression. If the expression is not a constant, then it is evaluated and a new duration statement is constructed containing the resulting constant value. This statement, plus the rest of the program, is then executed.

Stmt-Duration-Eval:

$$[\![expr \notin TimeConst$$
$$;\ (\tau, css, cs, pdg, ob) \vdash [\![e]\!] = Some\ (value, \tau')$$
$$;\ rest' = DurOrPDur\ (Dur\ (ExpTime\ (LitE\ value))\ stms)\ \#\ rest$$
$$;\ (\tau, css, cs, c, t, ob) \vdash (rest', pdg, cpu, bss) \text{ -}stmt\rightarrow (rest'', pdg', cpu', bss', \delta')$$
$$]\!] \Longrightarrow$$
$$(\tau, css, cs, c, t, ob) \vdash$$
$$(DurOrPDur\ (Dur\ (ExpTime\ expr)\ stms)\ \#\ rest, pdg, cpu, bss)$$
$$\text{ -}stmt\rightarrow (rest'', pdg', cpu', bss', \delta')\ |$$

The next rule deals with the case when the duration statement body completes, that is it state with no further statements or a singleton return statement. If the duration statement has already been evaluated (i.e. it is a constant literal) or if the duration is *ExecTime* meaning the duration should be calculated, and such a state is reached then the remainder of the program transition is calculuated, and the overall time penalty calculated.

Stmt-Duration-Complete:

$$[\![dur = DurOrPDur\ (Dur\ value\ stms)$$
$$;\ value \in TimeConstDur \cup \{ExecTime\}$$
$$;\ (\tau, css, cs, c, t, ob) \vdash (stms, pdg, cpu, bss) \text{ -}stmt\rightarrow (rest', pdg', cpu', bss', \delta')$$
$$;\ rest' = [] \vee (rest' = [Return\ (Some\ v)] \wedge v \in VDMValue)$$
$$;\ rest'' = rest' \bullet rest$$
$$;\ value \neq ExecTime \longrightarrow \delta \leq (value :: Time)$$
$$;\ (\tau, css, cs, c, t, ob) \vdash (rest'', pdg', c', bss') \text{ -}stmt\rightarrow (rest''', pdg'', cpu'', bss'', \delta')$$
$$;\ value \neq ExecTime \longrightarrow \delta'' = (value :: Time) + \delta'$$
$$;\ value = ExecTime \longrightarrow \delta'' = \delta + \delta'$$
$$]\!] \Longrightarrow$$
$$(\tau, css, cs, c, t, ob) \vdash$$
$$(dur\ \#\ rest, pdg, cpu, bss) \text{ -}stmt\rightarrow (rest''', pdg'', cpu'', bss'', \delta'')\ |$$

The next rule deals with the situation when the duration statement can executed only partially, that is the time elapsed is less than the time penalty for the whole body. This being the case the duration statement becomes a partial duration statement specifying the time elapsed so far, the remain time, and

the remaining body of the statement.

Stmt-Duration-to-PartialDuration:
$[\![value \in TimeConstDur \cup \{ExecTime\}$
$; (\tau, css, cs, c, t, ob) \vdash (stms, pdg, cpu, bss) -stmt\rightarrow (rest', pdg', cpu', bss', \delta)$
$; hd(rest) \notin range(Return)$
$; value \neq ExecTime \longrightarrow \delta \leq (value :: Time)$
$; rest'' = [PDur\ value\ \delta\ rest']$
$]\!] \Longrightarrow$
$(\tau, css, cs, c, t, ob) \vdash$
$(DurOrPDur\ (Dur\ value\ stms) \# rest, pdg, cpu, bss) -stmt\rightarrow (rest'', pdg', cpu', bss', \delta)\ |$

The next rule deals with further transitions of a partial duration statement and is very similar to the previous rule.

Stmt-Duration-Step-PartialDuration:
$[\![value \in TimeConstDur \cup \{ExecTime\}$
$; (\tau, css, cs, c, t, ob) \vdash (stms, pdg, cpu, bss) -stmt\rightarrow (rest', pdg', cpu', bss', \delta)$
$; hd(rest :: Stm\ list) \notin range(Return)$
$; value \neq ExecTime \longrightarrow \delta \leq ((value :: Time) - \delta_e)$
$; rest'' = [PDur\ value\ (\delta_e + \delta)\ rest']$
$]\!] \Longrightarrow$
$(\tau, css, cs, c, t, ob) \vdash$
$(DurOrPDur\ (PDur\ value\ \delta_e\ stms) \# rest, pdg, cpu, bss) -stmt\rightarrow (rest'', pdg', cpu', bss', \delta)\ |$

The final duration rule deals with the completion of a partial dureation statement when the body transitions to a state where no further statements remain, or else only a return statement remains.

Stmt-Duration-Complete-PartialDuration:
$[\![partialduration = DurOrPDur\ (PDur\ value\ \delta_e\ stms)$
$; (\tau, css, cs, c, t, ob) \vdash (stms, pdg, cpu, bss) -stmt\rightarrow (rest', pdg', cpu', bss', \delta)$
$; rest' = [] \vee (rest' = [Return\ (Some\ v)] \wedge v \in VDMValue)$
$; rest'' = rest' \bullet rest$
$; value \neq ExecTime \longrightarrow \delta \leq ((value :: Time) - \delta_e)$
$; (\tau, css, cs, c, t, ob) \vdash (rest'', pdg', cpu', bss') -stmt\rightarrow (rest''', pdg'', cpu'', bss'', \delta')$
$; value \neq ExecTime \longrightarrow \delta'' = (value :: Time) + \delta'$
$; value = ExecTime \longrightarrow \delta'' = \delta + \delta'$
$]\!] \Longrightarrow$
$(\tau, css, cs, c, t, ob) \vdash$
$(partialduration \# rest, pdg, cpu, bss) -stmt\rightarrow (rest''', pdg'', cpu'', bss'', \delta'')\ |$

The next three rules deal with iteration of a statement over an index range, over a sequence data structure, and over a finite set, respectively. They all follow a similar structure: the bounds of the iteration are calculated, a list of partial let defs is created – one for each instance of the iteration – giving values to the internal variables, and the overall semantics for the resulting sequence of statements is calculated.

Stmt-ForIndex:
$[\![forindex = ForIndex\ id_v\ e_f\ e_t\ e_b\ stm$
$; (\tau, css, cs, pdg, ob) \vdash [\![e_f]\!] = Some\ (v_f, \delta_f)$
$; (\tau, css, cs, pdg, ob) \vdash [\![e_t]\!] = Some\ (v_t, \delta_t)$

$$
\begin{array}{l}
\;;\; (\tau, css, cs, pdg, ob) \vdash [\![e_b]\!] = Some\ (v_b, \delta_b) \\
\;;\; stms = [PartialLetDef\ [id_v \mapsto NumV\ v]\ []\ stm \\
\qquad\quad . n \leftarrow [1..((v_f :: int) - v_t) + 1] \\
\qquad\quad , n \in Nats \\
\qquad\quad , v = (v_f :: int) + (n * (v_b :: int))] \\
\;;\; (\tau, css, cs, c, t, ob) \vdash (stms \bullet rest, pdg, cpu, bss) \; -stmt\rightarrow \; (rest', pdg', cpu', bss', \delta) \\
\;;\; \delta' = \delta_f + \delta_t + \delta + ForIndexTime \\
[\!] \Longrightarrow \\
(\tau, css, cs, c,t\ ,ob) \vdash \\
(forindex\ \#\ rest, pdg, cpu, bss) \; -stmt\rightarrow \; (rest', pdg', cpu', bss', \delta') \mid
\end{array}
$$

Stmt-ForSeq:

$$
\begin{array}{l}
[\![\ (\tau, css, cs, pdg, ob) \vdash [\![seqExpr]\!] = Some\ (SeqV\ seq, \delta_e) \\
\;;\; stms = [PartialLetDef\ \sigma\ []\ stm.\ i \leftarrow [0..<length\ seq], match(p, seq!i) = Some\ \sigma] \\
\;;\; (\tau, css, cs, c, t, ob) \vdash (stms \bullet rest, pdg, cpu, bss) \; -stmt\rightarrow \; (rest', pdg', cpu', bss', \delta) \\
\;;\; \delta' = \delta_e + \delta + ForSeqTime \\
[\!] \Longrightarrow \\
(\tau, css, cs, c,t\ ,ob) \vdash \\
(ForSeq\ p\ seqExp\ stm\ \#\ rest, pdg, cpu, bss) \; -stmt\rightarrow \; (rest', pdg', cpu', bss', \delta') \mid
\end{array}
$$

Stmt-ForSet:

$$
\begin{array}{l}
[\![\ (\tau, css, cs, pdg, ob) \vdash [\![setExpr]\!] = Some\ (SetV\ setv, \delta_e) \\
\;;\; stms = [PartialLetDef\ \sigma\ []\ stm \\
\qquad\quad . i \leftarrow [0..<length(set2seq(setv))] \\
\qquad\quad , match(p, (set2seq(setv))!i) = Some\ \sigma] \\
\;;\; (\tau, css, cs, c, t, ob) \vdash (stms \bullet rest, pdg, cpu, bss) \; -stmt\rightarrow \; (rest', pdg', cpu', bss', \delta) \\
\;;\; \delta' = \delta_e + \delta + ForSetTime \\
[\!] \Longrightarrow \\
(\tau, css, cs, c,t\ ,ob) \vdash \\
(ForSet\ p\ setExpr\ stm\ \#\ rest, pdg, cpu, bss) \; -stmt\rightarrow \; (rest', pdg', cpu', bss', \delta') \mid
\end{array}
$$

The cycles statement functions is giving a semantics in terms of the duration statement, with the appropriate time scale coefficient given by the executing CPU context.

Stmt-Cycle:

$$
\begin{array}{l}
[\![\ (\tau, css, cs, pdg, ob) \vdash [\![e]\!] = Some\ (val, \tau_v) \\
\;;\; (val :: int) \geq 0 \\
\;;\; ti = convertCyclesToTime(val, cpu) \\
\;;\; dur = Dur\ (ExpTime\ (LitE\ (NumV\ ti)))\ [stm] \\
\;;\; (\tau, css, cs, c, t, ob) \vdash (dur\ \#\ rest, pdg, cpu, bss) \; -stmt\rightarrow \; (rest', pdg', cpu', bss', \delta) \\
[\!] \Longrightarrow \\
(\tau, css, cs, c,t\ ,ob) \vdash \\
(Cycles\ e\ stm\ \#\ rest, pdg, cpu, bss) \; -stmt\rightarrow \; (rest', pdg', cpu', bss', \delta) \mid
\end{array}
$$

The next rule deals with definition of local variables in a let statement. The rule creates a partial let definition from the given variable binding and the semantics of the latter is calculated.

Stmt-LetDef:

$$
\begin{array}{l}
[\![\ stms = PartialLetDef\ Map.empty\ defs\ bdy\ \#\ rest \\
\;;\; (\tau, css, cs, c, t, ob) \vdash (stms, pdg, cpu, bss) \; -stmt\rightarrow \; (rest', pdg', cpu', bss', \delta) \\
\;;\; \delta' = \delta + LetDefTime
\end{array}
$$

$$
\begin{array}{l}
[\![\Longrightarrow \\
(\tau, css, cs, c, t, ob) \vdash \\
(LetDef\ defs\ bdy\ \#\ rest, pdg, cpu, bss)\ -stmt\rightarrow\ (rest', pdg', cpu', bss', \delta')\ |
\end{array}
$$

Let-be-such-that functions much the same as a let, except that the variable value being declared is selected non-deterministically through the use of the bind transition.

Stmt-LetBe:

$$
\begin{array}{l}
[\![\ (\tau, css, cs, c, t, ob) \vdash (bnd, pdg, cpu)\ -bind\rightarrow\ (ps, \sigma', \delta_b) \\
;\ pdg' = pdg(ob \mapsto the(pdg(ob)) +\!\!+ \sigma') \\
;\ (\tau, css, cs, pdg', ob) \vdash [\![e]\!] = Some\ (TrueV, \delta_e) \\
;\ stms = PartialLetDef\ \sigma'\ []\ bdy\ \#\ rest \\
;\ (\tau, css, cs, c, t, ob) \vdash (stms, pdg, cpu, bss)\ -stmt\rightarrow\ (rest', pdg', cpu', bss', \delta) \\
;\ \delta' = \delta_e + \delta_b + \delta + LetBeTime \\
[\![\Longrightarrow \\
(\tau, css, cs, c, t, ob) \vdash \\
(LetBe\ bnd\ e\ bdy\ \#\ rest, pdg, cpu, bss)\ -stmt\rightarrow\ (rest', pdg', cpu', bss', \delta')\ |
\end{array}
$$

The next three rules deal with partial let definitions. When variables are declared locally their values must be evaluated in the given context. Partial let defs are a semantic artefact that enable this. The context is augmented step-by-step with the variable assignmens being made. The first rule shows how a single local assignment is evaluated. Given a local variable id_v, assigned expression e, and partially evaluated let context σ, e is evaluated within the context of σ and the partial let context is updated with the new evaluation.

Stmt-PartialLetDef-Step:

$$
\begin{array}{l}
[\![\ partialletdef = PartialLetDef\ \sigma\ defs\ bdy \\
;\ (id_v, e) = hd\ defs \\
;\ pdg' = pdg(ob \mapsto the(pdg(ob)) +\!\!+ \sigma) \\
;\ (\tau, css, cs, pdg', ob) \vdash [\![e]\!] = Some\ (v, \delta_e) \\
;\ \sigma' = \sigma(id_v \mapsto v) \\
;\ stms = PartialLetDef\ \sigma'\ (tl\ defs)\ bdy\ \#\ rest \\
;\ (\tau, css, cs, c, t, ob) \vdash (stms, pdg, cpu, bss)\ -stmt\rightarrow\ (rest', pdg'', cpu', bss', \delta) \\
;\ \delta' = \delta_e + \delta \\
[\![\Longrightarrow \\
(\tau, css, cs, c, t, ob) \vdash \\
(partialletdef\ \#\ rest, pdg, cpu, bss)\ -stmt\rightarrow\ (rest', pdg'', cpu', bss', \delta')\ |
\end{array}
$$

The next rule gives the case when a partial let def is fully evaluated – no further variable declarations remain in the list. This being the case the inner body can be executed in the context of the evaluated variable declarations. This rule is applicable only when the resulting state is not an empty list of statements or a singleton return, which is dealt with by the next rule.

Stmt-PartialLetDef-Eval-Waiting:

$$
\begin{array}{l}
[\![\ partialletdef = PartialLetDef\ \sigma\ []\ bdy \\
;\ pdg' = pdg(ob \mapsto \sigma) \\
;\ (\tau, css, cs, c, t, ob) \vdash ([bdy], pdg', cpu, bss)\ -stmt\rightarrow\ (rest', pdg'', cpu', bss', \delta) \\
;\ rest' \neq [] \wedge (rest' \neq [Return\ -]) \\
;\ pdg''' = (pdg''\ |`\ (-\ \{ob\}))\ +\!\!+\ (pdg\ |`\ \{ob\}) \\
;\ \sigma' = the\ (pdg''\ ob) \\
;\ rest'' = PartialLetDef\ \sigma'\ []\ (SimpleBlock\ rest')\ \#\ rest
\end{array}
$$

$$
\begin{array}{l}
[\![\Longrightarrow \\
(\tau, css, cs, c, t, ob) \vdash \\
(partialletdef \ \# \ rest, pdg, cpu, bss) \ -stmt\rightarrow \ (rest'', pdg''', cpu'', bss'', \delta') \ |
\end{array}
$$

The final partial let definition rule deals with when the inner body has completed; i.e. the resulting statement list is empty or a return statement. This being the case the statements following the partial let definition are executed.

Stmt-PartialLetDef-Eval-Complete:

$$
\begin{array}{l}
[\![\ partialletdef = PartialLetDef \ \sigma \ [] \ bdy \\
; \ pdg' = pdg(ob \mapsto \sigma) \\
; \ (\tau, css, cs, c, t, ob) \vdash ([bdy], pdg', cpu, bss) \ -stmt\rightarrow \ (rest', pdg'', cpu', bss', \delta_b) \\
; \ rest' = [] \ \vee \ (rest' = [Return \ (Some \ v)] \ \wedge \ v \in VDMValue) \\
; \ rest'' = rest' \bullet rest \\
; \ pdg''' = (pdg'' \mid` (- \{ob\})) \ ++ \ (pdg \mid` \{ob\}) \\
; \ (\tau, css, cs, c, t, ob) \vdash (rest'', pdg''', cpu', bss') \ -stmt\rightarrow \ (rest''', pdg'''', cpu'', bss'', \delta) \\
; \ \delta' = \delta_b + \delta \\
]\!] \Longrightarrow \\
(\tau, css, cs, c, t, ob) \vdash \\
(partialletdef \ \# \ rest, pdg, cpu, bss) \ -stmt\rightarrow \ (rest''', pdg'''', cpu'', bss'', \delta') \ |
\end{array}
$$

The next two rules deal with variable assignments to local and remote variables. The first rule evaluates the expression to be assigned, and updates the pending variable state associated with the object context. The invariant of the associated class is also checked at this point using the *checkInvs* function on the updated state.

Stmt-Assign-Local:

$$
\begin{array}{l}
[\![\ (\tau, css, cs, pdg, ob) \vdash [\![e]\!] = Some \ (value, \delta_e) \\
; \ \sigma' = the(pdg(ob))(tgt \mapsto value) \\
; \ pdg' = pdg(ob \mapsto \sigma') \\
; \ objects \ cpu \ ob = Some \ obje \\
; \ checkInvs(invs(the(css(class \ obje)))), objstate \ obje \ ++ \ \sigma') \\
; \ (\tau, css, cs, c, t, ob) \vdash (rest, pdg', cpu, bss) \ -stmt\rightarrow \ (rest', pdg'', cpu', bss', \delta) \\
; \ \delta' = \delta_e + \delta + LocalAssignmentTime \\
]\!] \Longrightarrow \\
(\tau, css, cs, c, t, ob) \vdash \\
(Assignment \ (\!| \ asntarget = Inl \ tgt, asnexp = e \ |\!) \ \# \ rest, pdg, cpu, bss) \\
\quad -stmt\rightarrow \ (rest', pdg'', cpu', bss', \delta') \ |
\end{array}
$$

The second assignment rule deals with the case when the assignment is made to a variable in a different object to the one where the assignment statement is located. In [28] this is a called ”remote assignment”, however here we reserve the term “remote” for inter-CPU communication. The result of this rule is an update to the pending state of the associated object.

Stmt-Assign-Nonlocal:

$$
\begin{array}{l}
[\![\ (\tau, css, cs, pdg, ob) \vdash [\![e]\!] = Some \ (value, \delta_e) \\
; \ \sigma' = the(pdg(oid))(v \mapsto value) \\
; \ pdg' = pdg(oid \mapsto \sigma') \\
; \ objects \ cpu \ ob = Some \ obje \\
; \ checkInvs(invs(the(css(class \ obje)))), objstate \ obje \ ++ \ \sigma') \\
; \ (\tau, css, cs, c, t, ob) \vdash (rest, pdg', cpu, bss) \ -stmt\rightarrow \ (rest', pdg'', cpu', bss', \delta)
\end{array}
$$

$$
\begin{array}{l}
\;;\; \delta' = \delta_e + \delta + \mathit{NonlocalAssignmentTime} \\
[\![\Longrightarrow \\
(\tau, \mathit{css}, \mathit{cs}, c, t, \mathit{ob}) \vdash \\
(\mathit{Assignment}\ (\!|\ \mathit{asntarget} = \mathit{Inr}\ (\mathit{oid}, v), \mathit{asnexp} = e\ |\!)\ \#\ \mathit{rest}, \mathit{pdg}, \mathit{cpu}, \mathit{bss}) \\
\quad -\mathit{stmt}\rightarrow (\mathit{rest}', \mathit{pdg}'', \mathit{cpu}', \mathit{bss}', \delta')\ |
\end{array}
$$

The next four rules deal with atomic assignments, where several variables are assigned simultaneously. The invariant need only hold when the whole assignment is completed. The first rule converts an atomic assignment into a partial assignment.

Stmt-Atomic-Start:

$$
\begin{array}{l}
[\![\ \mathit{stmts} = \mathit{PartialAtomic}\ \mathit{assigns}\ \#\ \mathit{rest} \\
\;;\; (\tau, \mathit{css}, \mathit{cs}, c, t, \mathit{ob}) \vdash (\mathit{stmts}, \mathit{pdg}, \mathit{cpu}, \mathit{bss})\ -\mathit{stmt}\rightarrow (\mathit{rest}', \mathit{pdg}'', \mathit{cpu}', \mathit{bss}', \delta') \\
[\![\Longrightarrow \\
(\tau, \mathit{css}, \mathit{cs}, c, t, \mathit{ob}) \vdash \\
(\mathit{Atomic}\ \mathit{assigns}\ \#\ \mathit{rest}, \mathit{pdg}, \mathit{cpu}, \mathit{bss})\ -\mathit{stmt}\rightarrow (\mathit{rest}', \mathit{pdg}'', \mathit{cpu}', \mathit{bss}', \delta')\ |
\end{array}
$$

The next rule is the base case for atomic assignments. Once all assignments have been made the invariant is checked.

Stmt-PartialAtomic-Base:

$$
\begin{array}{l}
[\![\ \mathit{states} = [\mathit{oid} \mapsto \mathit{objstate}(\mathit{the}(\mathit{objects}\ \mathit{cpu}\ \mathit{oid})) +\!\!+ \mathit{the}(\mathit{pdg}(\mathit{oid})) \\
\qquad\quad |\ \mathit{oid}\ \sigma.\ (\mathit{oid}, \sigma) \in_m \mathit{objects}\ \mathit{cpu}] \\
\;;\; \mathit{invars} = [\mathit{oid} \mapsto \mathit{invs}(\mathit{the}(\mathit{css}(\mathit{objclass}(\mathit{the}(\mathit{objects}\ \mathit{cpu}\ \mathit{oid})))))) \\
\qquad\quad |\ \mathit{oid}.\ \mathit{oid} \in \mathit{dom}(\mathit{objects}\ \mathit{cpu})] \\
\;;\; \forall\ \mathit{oid} \in \mathit{dom}(\mathit{objects}\ \mathit{cpu}).\ \mathit{checkInvs}(\mathit{the}(\mathit{invars}(\mathit{oid})), \mathit{the}(\mathit{states}(\mathit{oid}))) \\
\;;\; \delta = \mathit{AtomicTime} \\
]\!] \Longrightarrow \\
(\tau, \mathit{css}, \mathit{cs}, c, t, \mathit{ob}) \vdash \\
(\mathit{PartialAtomic}\ []\ \#\ \mathit{rest}, \mathit{pdg}, \mathit{cpu}, \mathit{bss})\ -\mathit{stmt}\rightarrow ([], \mathit{pdg}, \mathit{cpu}, \mathit{bss}, \delta)\ |
\end{array}
$$

The next two rules apply local and non-local assignments. They differ from the normal assignment rules in that the invariant is not checked, as this is only done once all assignments in the atomic assignment have been made.

Stmt-PartialAtomic-Local:

$$
\begin{array}{l}
[\![\ \mathit{length}(\mathit{assigns}) \geq 1 \\
\;;\; \mathit{hd}(\mathit{assigns}) = \mathit{Assignment.make}\ (\mathit{Inl}\ \mathit{tgt})\ \mathit{expr} \\
\;;\; (\tau, \mathit{css}, \mathit{cs}, \mathit{pdg}, \mathit{ob}) \vdash [\![\mathit{expr}]\!] = \mathit{Some}\ (\mathit{value}, \delta_e) \\
\;;\; \sigma' = \mathit{the}(\mathit{pdg}(\mathit{ob})) +\!\!+ [\mathit{tgt} \mapsto \mathit{value}] \\
\;;\; \mathit{pdg}' = \mathit{pdg} +\!\!+ [\mathit{ob} \mapsto \sigma'] \\
\;;\; \mathit{rest}' = \mathit{PartialAtomic}\ (\mathit{tl}\ \mathit{assigns})\ \#\ \mathit{rest} \\
\;;\; (\tau, \mathit{css}, \mathit{cs}, c, t, \mathit{ob}) \vdash (\mathit{rest}', \mathit{pdg}', \mathit{cpu}, \mathit{bss})\ -\mathit{stmt}\rightarrow (\mathit{rest}'', \mathit{pdg}'', \mathit{cpu}', \mathit{bss}', \delta) \\
\;;\; \delta' = \delta_e + \delta + \mathit{LocalAssignmentTime} \\
]\!] \Longrightarrow \\
(\tau, \mathit{css}, \mathit{cs}, c, t, \mathit{ob}) \vdash \\
(\mathit{PartialAtomic}\ \mathit{assigns}\ \#\ \mathit{rest}, \mathit{pdg}, \mathit{cpu}, \mathit{bss})\ -\mathit{stmt}\rightarrow (\mathit{rest}'', \mathit{pdg}'', \mathit{cpu}', \mathit{bss}', \delta')\ |
\end{array}
$$

Stmt-PartialAtomic-Nonlocal:

$$
\begin{array}{l}
[\![\ \mathit{length}(\mathit{assigns}) \geq 1 \\
\;;\; \mathit{hd}(\mathit{assigns}) = \mathit{Assignment.make}\ (\mathit{Inr}\ (\mathit{id}_o, \mathit{id}_v))\ \mathit{expr} \\
\;;\; (\tau, \mathit{css}, \mathit{cs}, \mathit{pdg}, \mathit{ob}) \vdash [\![\mathit{expr}]\!] = \mathit{Some}\ (\mathit{value}, \delta_e)
\end{array}
$$

$$
\begin{array}{l}
\;;\; \sigma' = the(pdg(id_o)) +\!\!+ [id_v \mapsto value] \\
\;;\; pdg' = pdg +\!\!+ [id_o \mapsto \sigma'] \\
\;;\; rest' = PartialAtomic\ (tl\ assigns)\ \#\ rest \\
\;;\; (\tau, css, cs, c, t, ob) \vdash (rest', pdg', cpu, bss) -stmt\rightarrow (rest'', pdg'', cpu', bss', \delta) \\
\;;\; \delta' = \delta_e + \delta + NonlocalAssignmentTime \\
[\![\Longrightarrow \\
(\tau, css, cs, c, t, ob) \vdash \\
(PartialAtomic\ assigns\ \#\ rest, pdg, cpu, bss) -stmt\rightarrow (rest'', pdg'', cpu', bss', \delta') \mid
\end{array}
$$

The next rule deal with starting an object's thread. The object must not already have a running thread, and the object must have a non-periodic thread body. This being the case a new thread is created on the current CPU.

Stmt-Start:

$$
\begin{array}{l}
[\![\forall\ thr \in ran(threads\ cpu).\ tcontext\ thr \neq obje \\
\;;\; Inl\ bdy = initial(the(css(class(the(objects\ cpu\ obje))))) \\
\;;\; cpu' = createThread(cpu, obje, bdy) \\
\;;\; (\tau, css, cs, c, t, ob) \vdash (rest, pdg, cpu', bss) -stmt\rightarrow (rest', pdg', cpu'', bss', \delta) \\
\;;\; \delta' = \delta + StartTime \\
[\![\Longrightarrow \\
(\tau, css, cs, c, t, ob) \vdash \\
(Start\ obje\ \#\ rest, pdg, cpu, bss) -stmt\rightarrow (rest', pdg', cpu'', bss', \delta') \mid
\end{array}
$$

The next two rules deal with the execution of a context for an object operation call. Object contexts exist to record the valuation of the state at the point that the execution of the operation began. They are used to evaluate the postcondition at the conclusion of the operation. The first rule simply steps an object context forward by stepping the encapsulated body, provided that the resulting program is not empty or a singleton return.

Stmt-ObjectContext-Step:

$$
\begin{array}{l}
[\![(\tau, css, cs, c, t, oid) \vdash ([bdy], pdg, cpu, bss) -stmt\rightarrow (rest', pdg', cpu', bss', \delta) \\
\;;\; rest' \neq [] \wedge hd(rest') \notin range\ Return \\
\;;\; rest'' = ObjectContext\ oid\ (SimpleBlock\ rest')\ cctx\ \#\ rest \\
[\![\Longrightarrow \\
(\tau, css, cs, c, t, ob) \vdash \\
(ObjectContext\ oid\ bdy\ cctx\ \#\ rest, pdg, cpu, bss) -stmt\rightarrow (rest'', pdg', cpu', bss', \delta') \mid
\end{array}
$$

The second rule deals with the completion of the operation behaviour. When the body's execution yields an empty sequence or return the call context is extracted. This is then used to ensure that the operation postcondition is satisfied. Assuming that is the case, the remaining behaviour is calculated.

Stmt-ObjectContext-Complete:

$$
\begin{array}{l}
[\![(\tau, css, cs, c, t, oid) \vdash ([bdy], pdg, cpu, bss) -stmt\rightarrow (rest', pdg', cpu', bss', \delta) \\
\;;\; rest' = [] \vee (hd\ rest' = Return\ (Some\ v) \wedge v \in range\ LitE) \\
\;;\; cctx = CallContext.make\ prepending\ argues\ (Some\ postc) \\
\;;\; checkCallPost(css, cs, oid, prepending, pending', argues, [v], postc) \\
\;;\; rest'' = rest' \bullet rest \\
\;;\; (\tau, css, cs, c, t, ob) \vdash (rest'', pdg', cpu', bss') -stmt\rightarrow (rest''', pdg'', cpu'', bss'', \delta') \\
\;;\; \delta'' = \delta + \delta' \\
[\![\Longrightarrow
\end{array}
$$

$(\tau, css, cs, c, t, ob) \vdash$
$(ObjectContext\ oid\ bdy\ cctx \# rest, pdg, cpu, bss) -stmt\rightarrow (rest''', pdg'', cpu'', bss'', \delta'')\ |$

The next four rules deal with operation calls, in their various varieties – asynchronous vs. synchronous and local vs. remote. The first rule deals with the case of a synchronous local operation call to operation *opr* on object *oid* with arguments *argues*. The result of the operation (if any) is assigned to *tgt*. The list of arguments is first evaluated, and the specified operation is extracted from the object's class definition. The local state σ in which the operation will be executed is created by assigning the argument values to their corresponding variables. The precondition of the operation is also checked against the parameter values. An object context is then created for the operation by construction of a partial let definition for local parameters and a call context to store the state at the beginning of the operation. A call block is then constructed which adds a synchronous wait following the execution of the operation body. The synchronous wait will be matched with a return statement later.

Stmt-Call-Op-Local-Sync:
$[\![\ argsTimed = [(val, \delta_e).\ a \leftarrow argues, (\tau, css, cs, pdg, ob) \vdash [\![a]\!] = Some\ (val, \delta_e)]$
$;\ argues' = [val.\ (val, \text{-}) \leftarrow argsTimed]$
$;\ ops(the(css(objclass\ (the\ (objects\ cpu\ oid))))))\ opr$
$\quad = Some\ (Op.make\ \text{-}\ params\ retr\ bdy\ (Some\ prec)\ postc\ msr)$
$;\ \sigma = [p \mapsto a \mid a\ i\ p.\ i \in \{0..<length(argues')\} \wedge a = argues'!i \wedge params!i = (p, \text{-})]$
$;\ checkCallPre(css, cs, pdg, argues', params, oid, prec)$
$;\ callContext = CallContext.make\ pdg\ \sigma\ postc$
$;\ partialLetDef = PartialLetDef\ \sigma\ []\ (SimpleBlock\ bdy)$
$;\ objContext = ObjectContext\ oid\ partialLetDef\ callContext$
$;\ callBlock = [objContext, Wait\ tgt]$
$;\ stms = callBlock \bullet rest$
$;\ (\tau, css, cs, c, t, ob) \vdash (stms, pdg, cpu, bss) -stmt\rightarrow (rest', pdg', cpu', bss', \delta_r)$
$;\ \delta' = listsum\ [\delta_e.\ (\text{-}, \delta_e) \leftarrow argsTimed] + \delta_r + LocalSyncCallTime$
$]\!] \Longrightarrow$
$(\tau, css, cs, c, t, ob) \vdash$
$(SyncCall\ tgt\ (Inl\ (oid, opr))\ argues \# rest, pdg, cpu, bss)$
$\quad -stmt\rightarrow (rest'', pdg'', cpu'', bss'', \delta')\ |$

For the case of a local asynchronous call a new thread is created containing a synchronous call to the operation (which will in turn use the previous rule).

Stmt-Call-Op-Local-Async:
$[\![\ opTgt = Inl\ (oid, opr)$
$;\ argsTimed = [(val, \delta_e).\ a \leftarrow argues, (\tau, css, cs, pdg, ob) \vdash [\![a]\!] = Some\ (val, \delta_e)]$
$;\ argues' = [val.\ (val, \text{-}) \leftarrow argsTimed]$
$;\ ops(the(css(objclass\ (the\ (objects\ cpu\ oid))))))\ opr$
$\quad = Some\ (Op.make\ True\ params\ retr\ bdy\ (Some\ prec)\ postc\ msr)$
$;\ cpu' = createThread(cpu, oid, [DurOrPDur\ (Dur\ ExecTime\ [SyncCall\ None\ opTgt\ argues])])$
$;\ (\tau, css, cs, c, t, ob) \vdash (stms, pdg, cpu, bss) -stmt\rightarrow (rest', pdg', cpu'', bss', \delta)$
$;\ \delta' = listsum\ [\delta_e.\ (\text{-}, \delta_e) \leftarrow argsTimed] + \delta + LocalAsyncCallTime$
$]\!] \Longrightarrow$
$(\tau, css, cs, c, t, ob) \vdash$
$(AsyncCall\ opTgt\ argues \# rest, pdg, cpu, bss) -stmt\rightarrow (rest', pdg', cpu'', bss', \delta')\ |$

A remote asynchronous operation call requires the use of the message bus to convey the message to the remote CPU (*ccpu*) where the object (*oid*) resides. This call information is first extrated from the

call target. As before, the arguments are then evaluated, and the operation definition of *opr* is extracted. The bus which links the local CPU c to the remote CPU *ccpu* is then calculated from the bus map *bss*. A message is then constructed to convey the operation call, and this is appended to the queue for the message bus. Finally the remaining behaviour is calculated, along with the overall time penalty.

Stmt-Call-Op-Remote-Async:
$[\![opTgt = Inr\ (ccpu, oid, opr)$
$;\ argsTimed = [(val, \delta_e).\ a \leftarrow argues, (\tau, css, cs, pdg, ob) \vdash [\![a]\!] = Some\ (val, \delta_e)]$
$;\ argues' = [val.\ (val, \text{-}) \leftarrow argsTimed]$
$;\ ops(the(css(objclass\ (the\ (objects\ cpu\ oid))))))\ opr$
$\quad = Some\ (Op.make\ True\ params\ retr\ bdy\ (Some\ prec)\ postc\ msr)$
$;\ bss(bs) = Some\ (Bus.make\ (\{ccpu, c\} \cup connectd)\ spd\ que)$
$;\ cmsg = CMessage.make\ oid\ opr\ argues'\ None\ \tau$
$;\ bss' = bss(bus \mapsto Bus.make\ (\{ccpu, c\} \cup connectd)\ spd\ (que \bullet [(ccpu, cmesg)]))$
$;\ (\tau, css, cs, c, t, ob) \vdash (rest, pdg, cpu, bss') -stmt\rightarrow (rest', pdg', cpu', bss'', \delta)$
$;\ \delta' = listsum\ [\delta_e.\ (\text{-}, \delta_e) \leftarrow argsTimed] + \delta + RemoteAsyncCallTime$
$]\!] \Longrightarrow$
$(\tau, css, cs, c, t, ob) \vdash$
$(AsyncCall\ opTgt\ argues\ \#\ rest, pdg, cpu, bss) -stmt\rightarrow (rest', pdg', cpu', bss'', \delta')\ |$

A remote synchronous follows a similar form to the asynchronous call, but additionally has to handle return messages. The remaining statement list for the thread is prepended with a semantic wait statement indicating that the thread will resume when the target sends a return value. The call message is then inserted into the correct bus, and the status of the current thread is set to waiting. The bus rules will take care of resuming the threads execution when the operation returns.

Stmt-Call-Op-Remote-Sync:
$[\![opTgt = Inr\ (ccpu, oid, opr)$
$;\ argsTimed = [(val, \delta_e).\ a \leftarrow argues, (\tau, css, cs, pdg, ob) \vdash [\![a]\!] = Some\ (val, \delta_e)]$
$;\ argues' = [val.\ (val, \text{-}) \leftarrow argsTimed]$
$;\ ops(the(css(objclass\ (the\ (objects\ cpu\ oid))))))\ opr$
$\quad = Some\ (Op.make\ True\ params\ retr\ bdy\ (Some\ prec)\ postc\ msr)$
$;\ rest' = Wait\ (Some\ tgt)\ \#\ rest$
$;\ bss(bs) = Some\ (Bus.make\ (\{ccpu, c\} \cup connectd)\ spd\ que)$
$;\ cmsg = CMessage.make\ oid\ opr\ argues'\ (Some\ (c, t))\ \tau$
$;\ bss' = bss(bus \mapsto Bus.make\ (\{ccpu, c\} \cup connectd)\ spd\ (que \bullet [(ccpu, cmesg)]))$
$;\ cpu' = changeThreadStatus(cpu, t, Waiting)$
$;\ \delta' = listsum\ [\delta_e.\ (\text{-}, \delta_e) \leftarrow argsTimed] + RemoteSyncCallTime$
$]\!] \Longrightarrow$
$(\tau, css, cs, c, t, ob) \vdash$
$(SyncCall\ (Some\ tgt)\ opTgt\ argues\ \#\ rest, pdg, cpu, bss) -stmt\rightarrow (rest', pdg', cpu', bss', \delta')\ |$

The final six statement rules deal with operations returning a value. The first deals with the case when a return statement contains an expression which must be evaluated.

Stmt-Return-Eval:
$[\![e \notin VDMValue$
$;\ (\tau, css, cs, pdg, ob) \vdash [\![e]\!] = Some\ (retValue, \delta_e)$
$;\ rest' = Return\ (Some\ (LitE\ retValue))\ \#\ rest$
$;\ (\tau, css, cs, c, t, ob) \vdash (rest', pdg, cpu, bss) -stmt\rightarrow (rest'', pdg', cpu', bss', \delta)$
$;\ \delta' = \delta_e + ReturnTime$

$$
\begin{array}{l}
[\![\Longrightarrow \\
(\tau, css, cs, c, t, ob) \vdash \\
(Return\ (Some\ e) \# rest, pdg, cpu, bss) -stmt\rightarrow (rest'', pdg', cpu', bss', \delta') \mid
\end{array}
$$

The next rule consumes a proceeding statement if it is not a wait, and thus cannot be executed since it comes after the return statement.

Stmt-Return-Eat:

$$
\begin{array}{l}
[\![rest \neq [] \\
;\ hd(rest) \notin range\ Wait \\
;\ rest' = Return\ v \# tl(rest) \\
;\ (\tau, css, cs, c, t, ob) \vdash (rest', pdg, cpu, bss) -stmt\rightarrow (rest'', pdg', cpu', bss', \delta) \\
[\![\Longrightarrow \\
(\tau, css, cs, c, t, ob) \vdash \\
(Return\ (Some\ e) \# rest, pdg, cpu, bss) -stmt\rightarrow (rest'', pdg', cpu', bss', \delta) \mid
\end{array}
$$

The next rule is the base case, which applies when no further statements exist following the return.

Stmt-Return-Base:

$$
\begin{array}{l}
[\![v \in VDMValue \\
;\ stms = [Return\ (Some\ v)] \\
[\![\Longrightarrow \\
(\tau, css, cs, c, t, ob) \vdash (stms, pdg, cpu, bss) -stmt\rightarrow (stms, pdg, cpu, bss, 0) \mid
\end{array}
$$

The next rule deals with the case when a return is paired with a nil wait statement, that is one that does not expect a return value. This being the case, execution simply resumes from after the wait.

Stmt-Wait-Nil:

$$
\begin{array}{l}
[\![(\tau, css, cs, c, t, ob) \vdash (rest, pdg, cpu, bss) -stmt\rightarrow (rest', pdg', cpu', bss', \delta) \\
[\![\Longrightarrow \\
(\tau, css, cs, c, t, ob) \vdash \\
([Return\ None, Wait\ None] \bullet rest, pdg, cpu, bss) -stmt\rightarrow \\
\quad (rest', pdg', cpu', bss', \delta) \mid
\end{array}
$$

The next rule deals with the case when a return is paired with a wait holding a target, that is a variable into which the return value should be placed. This being the case the thread pending state is updated so that the variable points to the returned value.

Stmt-Return-Wait:

$$
\begin{array}{l}
[\![\sigma' = (the(pdg\ ob))(tgt \mapsto litExprOf(v)) \\
;\ pdg' = pdg(ob \mapsto \sigma') \\
;\ (\tau, css, cs, c, t, ob) \vdash (rest, pdg', cpu, bss) -stmt\rightarrow (rest', pdg'', cpu', bss', \delta) \\
[\![\Longrightarrow \\
(\tau, css, cs, c, t, ob) \vdash \\
([Return\ (Some\ v), Wait\ (Some\ (Inl\ tgt))] \bullet rest, pdg, cpu, bss) \\
\quad -stmt\rightarrow (rest', pdg'', cpu', bss', \delta) \mid
\end{array}
$$

Stmt-Wait-Message-Return:

$$
\begin{array}{l}
[\![tgt = (r_c, r_t) \\
;\ rmsg = RMessage.make\ [litExprOf(v)]\ tgt\ \tau
\end{array}
$$

$$
\begin{array}{l}
\;;\; bss(bus) = Some\ (Bus.make\ (\{r_c, c\} \cup connectd)\ spd\ que) \\
\;;\; bss' = bss(bus \mapsto Bus.make\ (\{r_c, c\} \cup connectd)\ spd\ (que \bullet [(r_c, Inr\ rmsg)])) \\
\;;\; (\tau, css, cs, c, t, ob) \vdash (rest, pdg, cpu, bss') -stmt\rightarrow (rest', pdg', cpu', bss'', \delta) \\
[\![\Longrightarrow \\
(\tau, css, cs, c, t, ob) \vdash \\
([Return\ (Some\ v), Wait\ (Some\ (Inr\ tgt))] \bullet rest, pdg, cpu, bss) \\
\quad -stmt\rightarrow (rest', pdg', cpu', bss'', \delta')
\end{array}
$$

inductive *vdmrt-dur-rel* ::
$Time \times Classes \times CPUs \times cpu\text{-}id \times thr\text{-}id \times obj\text{-}id \Rightarrow$
$Stm\ list \times Pending \times CPU \times Busses \Rightarrow$
$Stm\ list \times Pending \times CPU \times Busses \times Time \Rightarrow bool$ (- ⊢ - −*dur*→ - [60,0,60] 60) **where**

Duration-Eval:

$$
\begin{array}{l}
[\![expr \notin TimeConst \\
\;;\; (\tau, css, cs, pdg, ob) \vdash [\![e]\!] = Some\ (value, \tau') \\
\;;\; bdy\ = DurOrPDur\ (Dur\ (ExpTime\ expr)\ stmts) \# rest \\
\;;\; bdy' = DurOrPDur\ (Dur\ (ExpTime\ (LitE\ value))\ stmts) \# rest \\
\;;\; (\tau, css, cs, c, t, ob) \vdash (bdy, pdg, cpu, bss) -stmt\rightarrow (bdy', pdg', cpu', bss', \delta) \\
[\![\Longrightarrow \\
(\tau, css, cs, c, t, ob) \vdash (bdy, pdg, cpu, bss) -dur\rightarrow (bdy', pdg', cpu', bss', \delta)\ |
\end{array}
$$

Duration-Step-to-PartialDuration:

$$
\begin{array}{l}
[\![n \in TimeConstDur \cup \{ExecTime\} \\
\;;\; n \neq ExecTime \longrightarrow \delta \leq n \\
\;;\; bdy = DurOrPDur\ (Dur\ n\ stms) \# tail \\
\;;\; (\tau, css, cs, c, t, ob) \vdash (stms, pdg, cpu, bss) -stmt\rightarrow (rest, pdf', cpu', bss', \delta) \\
\;;\; bdy' = DurOrPDur\ (PDur\ n\ \delta\ rest) \# tail \\
[\![\Longrightarrow \\
(\tau, css, cs, c, t, ob) \vdash (bdy, pdg, cpu, bss) -dur\rightarrow (bdy', pdg', cpu', bss', \delta)\ |
\end{array}
$$

Duration-Step-PartialDuration:

$$
\begin{array}{l}
[\![n \in TimeConstDur \cup \{ExecTime\} \\
\;;\; n \neq ExecTime \longrightarrow \delta \leq n - \delta_e \\
\;;\; bdy = DurOrPDur\ (PDur\ n\ \delta_e\ stms) \# tail \\
\;;\; (\tau, css, cs, c, t, ob) \vdash (stms, pdg, cpu, bss) -stmt\rightarrow (rest, pdf', cpu', bss', \delta) \\
\;;\; bdy' = DurOrPDur\ (PDur\ n\ (\delta_e + \delta)\ rest) \# tail \\
[\![\Longrightarrow \\
(\tau, css, cs, c, t, ob) \vdash (bdy, pdg, cpu, bss) -dur\rightarrow (bdy', pdg', cpu', bss', \delta)
\end{array}
$$

inductive *vdmrt-cpu-rel* ::
$Time \times Classes \times CPUs \times cpu\text{-}id \Rightarrow$
$CPU \times Busses \Rightarrow$
$CPU \times Busses \times Time \Rightarrow bool$ (- ⊢ - −*cpu*→ - [60,0,60] 60) **where**

CPU-Pending:

$$
\begin{array}{l}
[\![cpu = CPU.make\ objs\ thrs\ spd \\
\;;\; t \in dom(thrs) \\
\;;\; the(thrs(t)) = Thread.make\ Running\ pdg\ ob\ bdy \\
\;;\; (\tau, css, cs, c, t, ob) \vdash (bdy, pdg, cpu, bss) -dur\rightarrow (bdy', pdg', cpu', bss', \delta) \\
\;;\; dbody\ (durOf\ (hd\ bdy')) = [] \\
\;;\; thrs' = thrs(t \mapsto Thread.make\ ThrPending\ pdg'\ ob\ bdy') \\
\;;\; cpu'' = CPU.make\ objs\ thrs'\ spd \\
[\![\Longrightarrow
\end{array}
$$

$(\tau, css, cs, c) \vdash (cpu, bss) -cpu\rightarrow (cpu'', bss', \delta)$ |
CPU-Running:
$[\![cpu = CPU.make\ objs\ thrs\ spd$
; $t \in dom(thrs)$
; $the(thrs(t)) = Thread.make\ Running\ pdg\ ob\ bdy$
; $(\tau, css, cs, c, t, ob) \vdash (bdy, pdg, cpu, bss) -dur\rightarrow (bdy', pdg', cpu', bss', \delta)$
; $dbody\ (durOf\ (hd\ bdy')) \neq []$
; $thrs' = thrs(t \mapsto Thread.make\ Running\ pdg'\ ob\ bdy')$
; $cpu'' = CPU.make\ objs\ thrs'\ spd$
$]\!] \Longrightarrow$
$(\tau, css, cs, c) \vdash (cpu, bss) -cpu\rightarrow (cpu'', bss', \delta)$

inductive *vdmrt-cpus-rel* ::
$Time \times Classes \Rightarrow$
$CPUs \times Busses \Rightarrow$
$CPUs \times Busses \times enat \Rightarrow bool$ (- $\vdash$ - $-cpus\rightarrow$ - [60,0,60] 60) **where**
CPUs-Base:
$(\tau, css) \vdash (Map.empty, bss) -cpus\rightarrow (Map.empty, bss, \infty)$ |
CPUs-Step:
$[\![c \in dom(cs)$
; $the(cs(c)) = cpu$
; $(\tau, css, cs, c) \vdash (cpu, bss) -cpu\rightarrow (cpu', bss', elapsd)$
; $(\tau, css) \vdash (cs\ |`\ \{c\}, bss') -cpus\rightarrow (css'', bss'', \tau_b)$
; $cpus''' = cpus''(c \mapsto cpu')$
; $\tau_b' = min\ elapsd\ \tau_b$
$]\!] \Longrightarrow$
$(\tau, css) \vdash (cs, bss) -cpus\rightarrow (css''', bss'', \tau_b')$

inductive *vdmrt-bus-rel* ::
$Time \times Classes \Rightarrow$
$Bus \times CPUs \Rightarrow$
$Bus \times CPUs \Rightarrow bool$ (- $\vdash$ - $-bus\rightarrow$ - [60,0,60] 60) **where**
Bus-Base:
$[\![queue\ bus = (c, msg) \# queue'$
; $\tau < arrivalTime\ (Bus.speed\ bus)\ msg$
$]\!] \Longrightarrow$
$(\tau, css) \vdash (bus, cs) -bus\rightarrow (bus, cs)$ |
Bus-Call:
$[\![queue\ bus = (c, msg) \# queue'$
; $\tau \geq arrivalTime\ (Bus.speed\ bus)\ msg$
; $msg = Inl\ (CMessage.make\ id_o\ opr\ argus\ replto\ sendTim)$
; $id_o \in dom(objects\ (the(cs(c))))$
; $cpu' = createThread\ (cpu, id_o, [DurOrPDur$
$(Dur$
$(ExpTime\ (LitE\ (NumV\ 0)))$
$[SyncCall\ (replto >>= Some \circ Inr)$
$(Inl\ (id_o, opr))$
$(map\ LitE\ argus)])])$
; $cs' = cs(c \mapsto cpu')$
; $bus' = Bus.make\ (Bus.cpus\ bus)\ (Bus.speed\ bus)\ queue'$
; $(\tau, css) \vdash (bus', cpus') -bus\rightarrow (bus'', cpus'')$

$$
\begin{array}{l}
\rrbracket \Longrightarrow \\
(\tau, css) \vdash (bus, cs)\ -bus\rightarrow\ (bus'', cs'')\ |
\end{array}
$$

Bus-Return:

$$
\begin{array}{l}
[\![\ queue\ bus = (c, msg)\ \#\ queue' \\
;\ \tau \geq arrivalTime\ (Bus.speed\ bus)\ msg \\
;\ msg = Inr\ (RMessage.make\ values\ replto\ sendTim) \\
;\ replto = (c, t) \\
;\ c \in dom(cs) \\
;\ the(cs(c)) = CPU.make\ objs\ thrs\ spd \\
;\ t \in dom(thrs) \\
;\ the(thrs(t)) = Thread.make\ Waiting\ pdg\ ctxt\ bdy \\
;\ bdy = DurOrPDur\ (Dur\ \tau_d\ stms)\ \#\ remainder \\
;\ stms' = insertReturn(stms, Return\ vals) \\
;\ bdy' = DurOrPDur\ (Dur\ \tau_d\ stms')\ \#\ remainder \\
;\ thr' = Thread.make\ Runnable\ pdg\ ctxt\ bdy' \\
;\ thrs' = thrs(t \mapsto thr') \\
;\ cs' = cs(c \mapsto CPU.make\ os\ thrs'\ spd) \\
;\ bus' = Bus.make\ (Bus.cpus\ bus)\ (Bus.speed\ bus)\ queue' \\
;\ (\tau, css) \vdash (bus', cs')\ -bus\rightarrow\ (bus'', cs'') \\
\rrbracket \Longrightarrow \\
(\tau, css) \vdash (bus, cs)\ -bus\rightarrow\ (bus'', cs'')
\end{array}
$$

inductive *vdmrt-busses-rel* :: $VDMRT \Rightarrow VDMRT \Rightarrow bool$ (**infix** $-busses\rightarrow 60$) **where**

Busses:

$$
\begin{array}{l}
[\![\ b \in dom(bss);\ \ the(bss(b)) = bus \\
;\ (\tau, css) \vdash (bus, cs)\ -bus\rightarrow\ (bus', cs') \\
;\ bss' = bss\ |`\ \{b\} \\
;\ VDMRT.make\ cs'\ bss'\ \tau\ cls\ -busses\rightarrow\ VDMRT.make\ cs''\ bss''\ \tau\ cls \\
;\ bss''' = bss(b \mapsto bus') \\
\rrbracket \Longrightarrow \\
VDMRT.make\ cs\ bss\ \tau\ cls\ -busses\rightarrow\ VDMRT.make\ cs''\ bss'''\ \tau\ cls\ |
\end{array}
$$

Busses-Base:

$$VDMRT.make\ cs\ Map.empty\ \tau\ cls\ -busses\rightarrow\ VDMRT.make\ cs\ Map.empty\ \tau\ cls$$

inductive *vdmrt-exec-rel* :: $VDMRT \Rightarrow (VDMRT \times Time) \Rightarrow bool$ (**infix** $-exec\rightarrow 60$) **where**

Exec:

$$
\begin{array}{l}
(\tau, css) \vdash (cs, bss)\ -cpus\rightarrow\ (cs', bss', \tau_b) \Longrightarrow \\
\quad VDMRT.make\ cs\ bss\ \tau\ css\ -exec\rightarrow\ (VDMRT.make\ cs'\ bss'\ \tau\ css, \tau_b)
\end{array}
$$

inductive *vdmrt-rel* :: $(VDMRT \times Time) \Rightarrow (VDMRT \times Time) \Rightarrow bool$ (**infix** $-vdmrt\rightarrow 60$) **where**

Big-Step:

$$
\begin{array}{l}
[\![\ vdmrt_1 = commitPendingValuesAndUpdateTime(vdmrt, \tau) \\
;\ vdmrt_1\ -busses\rightarrow\ vdmrt_2 \\
;\ vdmrt_3 = createPeriodicThreads(vdmrt_2) \\
;\ vdmrt_4 = doContextSwitches(vdmrt_3) \\
;\ vdmrt_4\ -exec\rightarrow\ (vdmrt_5, \tau_b) \\
;\ \tau_b' = min\ \tau_b\ (minPendingCommitTime(vdmrt_5)) \\
\rrbracket \Longrightarrow (vdmrt, \tau)\ -vdmrt\rightarrow\ (vdmrt_5, \tau_b')
\end{array}
$$

References

[1] Nuno Amalio, Ana Cavalcanti, Stefan Gulan, Christian König, and Jim Woodcock. Foundations for FMI Co-Modelling. Technical report, INTO-CPS Deliverable, D2.1d, December 2015.

[2] Nick Battle. Object oriented issues in VDM++. In Ken Pierce, Nico Plat, and Sune Wolff, editors, *Proc. 8th Overture Workshop*, volume CS-TR-1224, pages 9–18. Newcastle University, 2010.

[3] 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.

[4] 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.

[5] David Broman, Christopher Brooks, Lev Greenberg, Edward A. Lee, Michael Masin, Stavros Tripakis, and Michael Wetter. Determinate composition of FMUs for co-simulation. In *EMSOFT 2013*. IEEE, 2013.

[6] 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.

[7] A. Cavalcanti. The safety-critical java memory model formalised. *Formal Aspects of Computing*, 25(1):37–57, 2013.

[8] 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.

[9] Ana Cavalcanti, Andy Wellings, and Jim Woodcock. The safety-critical java memory model formalised. *Formal Aspects of Computing*, 25(1):37–57, 2013.

[10] T. L. V. de Lima Santos. *A Unifying Theory of Object-Orientation*. Tese de doutorado, Centro de Informática, Universidade Federal de Pernambuco, 2010.

[11] John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat, and Marcel Verhoef. *Validated Designs for Object–oriented Systems*. Springer, New York, 2005.

[12] John Fitzgerald, Peter Gorm Larsen, and Marcel Verhoef, editors. *Collaborative Design for Embedded Systems – Co-modelling and Co-simulation*. Springer, 2014.

[13] FMI development group. Functional Mock-up Interface for Model Exchange and Co-Simulation v2.0. Modelica Association Project "FMI", October 2014. Standard Specification.

[14] 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.

[15] S. Foster and J. Woodcock. Unifying theories of programming in Isabelle. In *ICTAC 2013 School on Software Engineering*, volume 8050 of *LNCS*, pages 109–155. Springer, 2013.

[16] S. Foster and J. Woodcock. Mechanised theory engineering in Isabelle. In M. Irlbeck, D. Peled, and A. Pretschner, editors, *Dependable Software Systems Engineering*. IOS Press, 2015.

[17] 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.

[18] I. J. Hayes, S. E. Dunne, and L. Meinicke. Unifying theories of programming that distinguish nontermination and abort. In *Mathematics of Program Construction (MPC)*, volume 6120 of *LNCS*, pages 178–194. Springer, 2010.

[19] Jifeng He. From CSP to hybrid systems. In A. W. Roscoe, editor, *A classical mind: essays in honour of C. A. R. Hoare*, pages 171–189. Prentice Hall, 1994.

[20] Tony Hoare. *Communicating Sequential Processes*. Prentice-Hall International, Englewood Cliffs, New Jersey 07632, 1985.

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

[22] 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.

[23] M. Iancu and F. Rabe. Formalising foundations of mathematics. *Mathematical Structures in Computer Science*, 21:883–911, 8 2011.

[24] Information technology – Programming languages, their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language, December 1996.

[25] C. B. Jones. Tentative steps towards a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, 1983.

[26] C. B. Jones. The early search for tractable ways of reasonning about programs. *IEEE, Annals of the History of Computing*, 25(2):26–49, 2003.

[27] Peter Gorm Larsen and Wiesław Pawłowski. The Formal Semantics of ISO VDM-SL. *Computer Standards and Interfaces*, 17(5–6):585–602, September 1995.

[28] 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.

[29] Betrand Meyer et al. Eiffel: Analysis, design and programming language. Technical Report ECMA-567, Ecma International, 2006.

[30] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[31] 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.

[32] Modelica Association. Modelica - A Unified Object-Oriented Language for Systems Modeling - Version 3.3 Revision 1. Standard Specification, July 2014.

[33] T. Nipkow, M. Wenzel, and L. C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[34] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima Inc, 2010.

[35] 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.

[36] G. M. Reed and A. W. Roscoe. A timed model for communicating sequential processes. *Theoretical Computer Science*, 58(1–3):249–261, 1988.

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

[38] Marcel Verhoef. *Modeling and Validating Distributed Embedded Real-Time Control Systems*. PhD thesis, Radboud University Nijmegen, 2009.

[39] 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.

[40] 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.

[41] M. Wenzel. The Isabelle/Isar reference manual. Technical report, http://isabelle.in.tum.de/doc/isar-ref.pdf, May 2015.

[42] J. Woodcock, J. Bryams, S. Canham, and S. Foster. The COMPASS modelling language: Timed semantics in UTP. In P. H. Welch, editor, *Communicating Process Architectures*. Open Channel Publishing, 2014.

[43] 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.

[44] 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.

[45] Jim Woodcock and Jim Davies. *Using Z – Specification, Refinement, and Proof*. Prentice Hall International Series in Computer Science, 1996.

[46] F. Zeyda and A. Cavalcanti. Higher-order UTP for a theory of methods. In *4th Intl. Symp. on Unifying Theories of Programming (UTP)*, volume 7681 of *LNCS*, pages 204–223. Springer, 2012.

[47] Frank Zeyda, Thiago Santos, Ana Cavalcanti, and Augusto Sampaio. A modular theory of object orientation in higher-order UTP. In Cliff Jones, Pekka Pihlajasaari, and Jun Sun, editors, *FM 2014: Formal Methods*, volume 8442 of *Lecture Notes in Computer Science*, pages 627–642. Springer, 2014.