Communicating Process Architectures 2014
P.H. Welch et al. (Eds.)
Open Channel Publishing Ltd., 2014

The COMPASS Modelling Language: Timed Semantics in UTP

Jim WOODCOCK [a,1], Jeremy BRYANS [b], Samuel CANHAM [a] and Simon FOSTER [a]
[a] *Department of Computer Science, University of York, UK*
[b] *School of Computing Science, Newcastle University, UK*

Abstract. We describe the denotational semantics of a subset of the COMPASS Modelling Language (CML), using Hoare & He's Unifying Theories of Programming. The subset consists of rich state and operations based on VDM, concurrency and communication, based on CSP, and discrete time, based on Timed CSP. Other features of CML not treated here include object orientation, pointers and object references, and mobile processes and channels; extensions planned for the future include priority, probabilistic choice, and continuous time. A rich collection of language features such as this presents significant challenges when building a formal semantics, so the approach taken in CML is compositional: each feature is given a separate semantics and a domain-specific language is then composed from whichever features are required for the job in hand. Composition is achieved from the use of Galois connections. In this paper, we describe the semantics for the timed, imperative process algebra subset of CML. We adopt a semantic domain suggested by Lowe & Ouaknine—timed testing traces—as the basis for our UTP semantics. We include an example CML specification taken from industry: a leadership election protocol for a system of systems.

Keywords. COMPASS Modelling Language, compositional semantics, formal semantics, CSP, VDM, Unifying Theories of Programming

Introduction

The COMPASS Modelling Language (CML) is a new language, developed for the modelling and analysis of systems of systems (SoS), which are typically large-scale systems composed of independent constituent systems [1]. The COMPASS project is described in detail at `www.compass-research.eu`. CML is based on a combination of VDM [2], CSP [3], and *Circus* [4,5,6]. Broadly speaking, a CML model consists of a collection of types, functions, channels and processes. Each process encapsulates a state and operations in the style of VDM and interacts with the environment via synchronous communications in the style of CSP. The main elements of the basic CML language with state, concurrency, and timing are described in Table 1. Additionally, CML is object oriented.

The main contribution of this paper is to present a semantics of CML. Our style provides a natural contract language for all language constructs, including nonterminating reactive processes. The result is a compositional formal definition of a complex language, with individual parts being available for reuse. Our work shows that the use of UTP scales up to industrial-strength languages.

Several aspects of CML are noteworthy:

- The model that we propose extends Lowe & Ouaknine's timed testing traces semantic model of [7], adding termination, divergence and additional program operators. The

[1]Corresponding Author: *Jim Woodcock, Department of Computer Science, University of York, Deramore Lane, York YO1 7BZ, UK* . E-mail: `jim.woodcock@york.ac.uk`.

deadlock	$STOP$	*termination*	$SKIP$
divergence	$CHAOS$	*assignment*	$(v := e)$
specification statement	$w : [pre, post]$	*simple prefix*	$a \rightarrow SKIP$
prefixed action	$a \rightarrow P$	*guarded action*	$[g] \,\&\, P$
sequential composition	$P \,;\, Q$	*internal choice*	$P \sqcap Q$
external choice	$P \,\Box\, Q$	*parallel composition*	$P \parallel_{cs} Q$
interleaving	$P \,\|\|\|\, Q$	*abstraction*	$P \setminus A$
recursion	$\mu X \bullet P(X)$	*wait*	$Wait(n)$
while	$b * P$	*timeout*	$P \overset{n}{\rhd} Q$
untimed timeout	$P \rhd Q$	*interrupt*	$P \,\triangle\, Q$
timed interrupt	$P \overset{n}{\triangle} Q$	*starts by*	$P \, startsby(n)$
ends by	$P \, endsby(n)$		

Table 1. The CML language.

timed testing trace model includes five axioms: all processes have a feasible initial state, are prefix closed, can refuse events they can't engage in, allow time to pass and do not allow an unbounded number of events to occur in finite time—this last property is called Zeno freedom. CML processes satisfy modified forms of these axioms, which describe non-terminated program behaviours. For example, in CML, every process must have a non-terminated feasible initial state, every prefix of an observable trace must be a feasible non-terminated trace of the process and non-terminated processes must allow time to pass. Where possible, these properties are enforced by the healthiness conditions of CML, but in the cases of prefix closure and Zeno freedom it is easier to prove the property inductively over operators in the CML signature.

- CSP contains unstable states from which a process must either terminate or engage in an event within a very short amount of time [3]. CML identifies this amount of time with the smallest distinguishable time duration in the theory: one tock step. This means that all unstable states, including the state of a process like *SKIP* before it terminates, are observable for up to one time unit; this in turn means that unstable observations can be made of CML processes, although the processes must reach stable states whenever time passes. In a CSP unstable state, the process can refuse arbitrary events, and thus the value of the refusals is interesting only when it occurs in the trace. In particular, we are not interested in the current value of the refusals, which is recorded in similar models such as [8]: process behaviour cannot depend on this value until it is recorded in the trace. The CML healthiness conditions enforce that processes are urgency free: every process starts and terminates in an unstable state which can last for up to one time unit.
- Constituent subsystems of an SoS have local private states that must be concealed from observers. State information can be communicated only over event channels. We use the model of concealed states described in [9]: non-terminated processes operate on an existentially quantified concealed state and reveal an arbitrary state to the environment. CML takes all of the advantages of this model, which prevents specification of infeasible constructs using external choice and state; however, this does restrict the ability of the interrupt operator to access the interrupted state, which is completely concealed from it: the CML interrupt operator checkpoints the initial state and rolls back if an interrupt occurs.

The paper is organised as follows. Section 1 contains a brief overview of the UTP philosophy and approach and Section 2 describes the semantic domain for CML. In Section 3, we present our case study: a CML model of an a industrial leadership election protocol used in

a commercial AudioVisual entertainment SoS. Finally, Section 4 describes related work and Section 5 concludes the paper.

1. Unifying Theories of Programming

UTP [10] sets out a long-term research agenda summarised as follows: researchers propose programming theories and practitioners use pragmatic programming paradigms; what is the relationship between them? UTP, based on predicative programming [11], gives three principal ways to study such relationships: (i) by computational paradigm, identifying common concepts; (ii) by level of abstraction, from requirements, through architectures and components, to platform-specific implementation technology; and (iii) by method of presentation—denotational, algebraic, and operational semantics and their mutual embeddings.

UTP presents a theoretical foundation for understanding software and systems engineering. It has been already been exploited in areas such as component-based systems [12], hardware [13,14], and hardware/software co-design [15], but UTP can also be used in a more active way as a domain-specific language for constructing domain-specific languages, especially ones with heterogeneous semantics. An example is the semantics for Safety-Critical Java [16,17]. The analogy is of a theory supermarket, where you shop for exactly those features you need, while being confident that the theories plug-and-play together.

The semantic model is an alphabetised version of Tarski's relational calculus, presented in a predicative style that is reminiscent of the schema calculus in the Z notation [18]. Each programming construct is formalised as a relation between an initial and an intermediate or final observation. The collection of these relations forms a *theory* of a paradigm that contains three essential parts: an alphabet, a signature, and healthiness conditions.

The *alphabet* is a set of variable names that gives the vocabulary for the theory being studied. Names are chosen for any relevant external observations of behaviour. For instance, programming variables x, y, and z would be part of the alphabet. Theories for particular programming paradigms require the observation of extra information; some examples are: a flag that says whether the program has started (*ok*); the current time (*clock*); the number of available resources (*res*); a trace of the events in the life of the program (*tr*); a set of refused events (*ref*), or a flag that says whether the program is waiting for interaction with its environment (*wait*). The *signature* gives syntactic rules for denoting objects of the theory. *Healthiness conditions* identify properties that characterise the theory, which can often be expressed in terms of a function ϕ that makes a program healthy. There is no point in applying ϕ twice, since we cannot make a healthy program even healthier, so ϕ must be idempotent: $\phi \circ \phi(P) = \phi(P)$. The fixed-points of the equation $P = \phi(P)$ are then the healthy predicates of the theory.

An *alphabetised predicate* $(P, Q, \ldots, \textbf{\textit{true}})$ is an alphabet-predicate pair, such that the predicate's free variables are all members of the alphabet. Relations are predicates in which the alphabet comprises plain variables (x, y, z, ...) and dashed variables (x', a', ...); the former represent initial observations, and the latter, intermediate or final observations. The alphabet of P is denoted αP, and may be divided into its before-variables ($in\alpha P$) and its after-variables ($out\alpha P$). A *homogeneous relation* has $out\alpha P = in\alpha P'$, where $in\alpha P'$ is the set of variables obtained by dashing all variables in the alphabet $in\alpha P$. A *condition* $(b, c, d, \ldots, true)$ has an empty output alphabet. Standard predicate calculus operators are used to combine alphabetised predicates, but their definitions must specify the alphabet of the combined predicate. For instance, the alphabet of a conjunction is the union of the alphabets of its components.

A distinguishing feature of UTP is its concern with program correctness, which is the same in every paradigm in [10]: in each state, the behaviour of an implementation implies

its specification. Suppose $\alpha P = \{a, b, a', b'\}$, then the *universal closure* of P is simply $\forall a, b, a', b' \bullet P$, denoted $[P]$. Program correctness for P with respect to specification S is denoted $S \sqsubseteq P$ (S is refined by P), and is defined as: $S \sqsubseteq P$ **iff** $[P \Rightarrow S]$.

UTP has an infix syntax for the conditional, written $P \lhd b \rhd Q$ and defined as the proposition $(b \wedge P) \vee (\neg b \wedge Q)$, provided $\alpha b \subseteq \alpha P = \alpha Q$. Sequence is modelled as relational composition: two relations may be composed, provided the alphabets match: $P(v') \,;\, Q(v) \;\widehat{=}\; \exists v_0 \bullet P(v_0) \wedge Q(v_0)$, if $out\alpha P = in\alpha Q' = \{v'\}$. If $A = \{x, y, \ldots, z\}$ and $\alpha e \subseteq A$, the assignment $x :=_A e$ of expression e to variable x changes only x's value: $x :=_A e \;\widehat{=}\; (x' = e \wedge y' = y \wedge \cdots \wedge z' = z)$. There is a degenerate form of assignment that changes no variable, called "skip": $\mathbb{I}_A \;\widehat{=}\; (v' = v)$, if $A = \{v\}$. Nondeterminism can arise in one of two ways: either as the result of run-time factors, such as distributed processing; or as the under-specification of implementation choices. Either way, nondeterminism is modelled by choice; the semantics is simply disjunction: $P \sqcap Q \;\widehat{=}\; P \vee Q$.

Variable blocks are split into the commands ***var*** x, which declares and introduces x in scope, and ***end*** x, which removes x from scope. In the definitions, A is an alphabet containing x and x'.

$$\textbf{\textit{var}}\, x \;\widehat{=}\; (\exists x \bullet \mathbb{I}_A) \qquad \textbf{\textit{end}}\, x \;\widehat{=}\; (\exists x' \bullet \mathbb{I}_A)$$

The relation ***var*** x is not homogeneous, since it does not include x in its alphabet, but it does include x'; similarly, ***end*** x includes x, but not x'.

The set of alphabetised predicates with a particular alphabet A forms a complete lattice under the refinement ordering (which is a partial order). The bottom element is denoted $\bot_A$, and is the weakest predicate ***true***; this is the program that aborts, and behaves quite arbitrarily. The top element is denoted $\top_A$, and is the strongest predicate ***false***; this is the program that performs miracles and implements every specification. Since alphabetised relations form a complete lattice, every construction defined solely using monotonic operators has a complete lattice of fixed points. The weakest fixed-point of the function F is denoted by μF, and is simply the greatest lower bound (the *weakest*) of all the fixed-points of F. This is defined: $\mu F \;\widehat{=}\; \bigsqcap \{ X \mid F(X) \sqsubseteq X \}$. The strongest fixed-point νF is the dual of the weakest fixed-point.

2. Timed Testing Traces

The semantic domain consists of traces with embedded refusal sets. We refer to these augmented traces as *timed testing traces*, because of their similarity with Lowe & Ouaknine's Timed Testing Traces [7], which is in turn related to the standard semantics for CSP.

The timed testing traces model of Lowe & Ouaknine records the passing of time with an explicit *tock* event and allows refusal experiments to be made only before *tock*s. In our model we do not observe a *tock* event directly. Instead, we observe the passage of time through the refusal experiments. At the end of each time interval either a refusal experiment is made or the empty refusal set is recorded. If we let Σ be the universe of events, then observations within this model are drawn from the following set:

Definition 2.1

$$timedTrace \;\widehat{=}\; (\Sigma + \mathbb{P}\,\Sigma)^*$$

In our semantics we restrict ourselves to finite traces, so $+$ is to be understood as a choice and x^* as all finite sequences containing only x. All members of this set are potential timed testing traces of CML processes. We refer to members of the set *timedTraces* as *traces* throughout the paper. We now give some useful definitions over traces.

Definition 2.2 *Let $A \subseteq \Sigma$, $a \in \Sigma$ and $t \in timedTrace$. Then*

$$events(t) = t \upharpoonright \Sigma$$
$$refsduring(t) = \mathrm{ran}(t \rhd \mathbb{P}(\Sigma))$$
$$refusals(t) = \bigcup refsduring(t)$$

The idle prefix of a trace t is denoted $idleprefix(t)$ and describes the longest prefix of t containing no observable events. The idle suffix of t is the remainder of the trace after the first visible event has been removed.

Definition 2.3 (Idleprefix and Idlesuffix) *Let $A \subseteq \Sigma$, $a \in \Sigma$ and $t \in timedTrace$. Then*

$$idleprefix(\langle\rangle) = \langle\rangle$$
$$idleprefix(\langle A\rangle \frown t) = \langle A\rangle \frown idleprefix(t)$$
$$idleprefix(\langle a\rangle \frown t) = \langle\rangle$$
$$idlesuffix(\langle\rangle) = \langle\rangle$$
$$idlesuffix(\langle A\rangle \frown t) = idlesuffix(t)$$
$$idlesuffix(\langle a\rangle \frown t) = t$$

We also define a function $tocks(t)$ that replaces all the refusal sets in a timed trace t with a new event, $tock \notin \Sigma$. This proves useful for placing conditions on duration of a trace.

Definition 2.4 *Let $A \subseteq \Sigma$, $a \in \Sigma$ and $t \in timedTrace$. Then*

$$tocks(\langle\rangle) = \langle\rangle$$
$$tocks(\langle a\rangle \frown t) = \langle a\rangle \frown tocks(t)$$
$$tocks(\langle A\rangle \frown t) = \langle tock\rangle \frown tocks(t)$$

We introduce other operators on timed traces as we need them. We present now the observation variables and healthiness conditions that characterise our semantic domain.

2.1. Observation Variables and Healthiness Conditions

Observations of CML processes contain four pairs of variables.

- ok, ok': These are the observation variables from designs [10, Chapter 3]. The observation ok describes the situation in which a process has been started in a stable state, whilst ok' describes the situation in which a process has reached a stable state.
- $wait, wait'$: These are the observation variables from reactive processes [10, Chapter 8]. The observation $wait$ describes the situation in which a process occupies a waiting state of its sequential predecessor, whilst $wait'$ describes the situation in which the process has reached a waiting state. The combination of ok and $wait$ and their dashed counterparts allow sequential combination to be defined as relational composition.
- rt, rt': These are the observations of the trace of the previous process (rt) and the current process (rt'). Traces encode all observations we wish to make about particular executions of CML processes: the trace of events marked out by the passage of time and the refusal experiments that can be made during execution.
- v, v': These are the variables that record our observations of the initial and final state of the current process.

We also introduce a derived variable: tt' is equal to $rt'-rt$ whenever that expression is defined, and undefined otherwise. Intuitively, tt' represents the portion of the trace carried out by the currently active process; however, it is not an observation variable and is therefore not quantified by $[\,]$ (universal closure over alphabets) or by sequential composition—it can always be replaced by $rt'-rt$ in any expression.

We make use of the notational shorthand introduced in [19]:

Definition 2.5

$$P^b_c = P[b, c/ok', wait]$$

which denotes the substitution of the boolean variables b and c for the variables ok' and $wait$.

There are seven healthiness conditions. We note in passing that we do not need to restrict the structure of the trace variables with a healthiness condition, since all elements of the set *timedTrace* are structurally valid observations of timed reactive processes.

The first requirement is that tt' is well-defined. This requires that the observation of rt prefixes the observation of rt'. ***RT1*** is similar to ***R1*** in that it ensures that a process cannot alter the part of the trace that has already been observed; all it may do is append to rt.

Definition 2.6 (*RT1*)

$$\mathbf{RT1}(P) = P \wedge rt \leq rt'$$

Our next healthiness condition is similar to ***R2*** in Hoare & He's theory of reactive processes (see [10, p.195]). It controls the use of the trace variable to make sure that P is not sensitive to the behaviour of its predecessors. For example, it cannot depend on certain events already having taken place, or on a particular amount of time having elapsed under its predecessor's control.

Definition 2.7 (*RT2*)

$$\mathbf{RT2}(P) = P[\langle\rangle, tt'/rt, rt']$$

The healthiness condition ***RT3*** is a modified form of ***R3*** in the theory of reactive processes (see [10, p.196]). It is similar to the condition ***R3h*** proposed in [9], and describes the behaviour of a process that has not been started: it may not extend the trace ($tt' = \langle\rangle$), and it may not observe the internal state of its predecessor. ***R3h*** in [9] differs from ***R3*** by removing the insistence that the state does not change while the process is waiting for external interaction. Changes to the internal state of a process are permitted by ***RT3***, but should remain unobservable until some interaction takes place. This inability to observe internal interaction has the consequence that a choice between two processes cannot be resolved by internal state changes, but only external events or the termination of one of the processes.

Definition 2.8 (*RT3*)

$$\mathbf{RT3}(P) = (\exists\, v' \bullet I\!I) \lhd wait \rhd P$$

Our fourth healthiness condition corresponds to ***CSP1*** in Hoare & He's theory of CSP (see [10, p.208]). If P's predecessor is in an unstable state, then P will not be started and we have $\neg\ ok$. What contribution will P now make to the divergent behaviour of its predecessor? It cannot alter the behaviour that has already been observed (***RT1***), but otherwise it can behave arbitrarily.

Definition 2.9 (*RT4*)

$$\mathbf{RT4}(P) = \mathbf{RT1}(\neg\ ok) \vee P$$

Our fifth healthiness condition is analogous to **CSP2** in [10], and states that P must be monotonic in the value of the ok' variable, just like a design: P cannot demand instability and nontermination. In other words, it is always possible to terminate: if P is capable of reaching a state with $\neg\, ok'$, then it must be capable of reaching the same state with ok'. In [19] in the context of designs, the authors show that this healthiness property is equivalent to $[P^f \Rightarrow P^t]$.

Definition 2.10 (RT5)

$$\mathbf{RT5}(P) = P \,;\, (\mathit{II}_{\{rt,wait,v\}} \wedge (ok \Rightarrow ok'))$$

We subscript the relational identity II with the set of observation variables that are required to be constant. Notice that **RT4** and **RT5** are the timed reactive versions of **H1** and **H2**, respectively (in the same way that **CSP1** and **CSP2** are the reactive versions). The remaining healthiness conditions involve the process *SKIP*. *SKIP* has precondition true, and it must terminate before any time passes. It is impossible for *SKIP* to engage in any events before it terminates. When it terminates, it does not change the state, although it can conceal its intermediate state before terminating.

Definition 2.11 (*SKIP*)

$$\mathit{SKIP} = \mathbf{RT3} \circ \mathbf{RT4}(ok' \wedge tt' = \langle\rangle \wedge (\neg\, wait' \Rightarrow v' = v))$$

The sixth healthiness condition corresponds to **CSP3**, insisting that *SKIP* is a left unit of sequential composition. In *CSP*, the effect of this was to make processes independent of the refusals of their predecessor, but since refusal information is captured in the trace in CML, this is already guaranteed by **RT2**. Instead, **RT6** ensures that processes never engage in urgent initial actions: it is always possible to make a stable initial observation of any process.

Definition 2.12 (RT6)

$$\mathbf{RT6}(P) = \mathit{SKIP} \,;\, P$$

The seventh healthiness condition corresponds to **CSP4**, insisting that *SKIP* is a right unit of sequential composition. Any process with an **RT3** process as a right unit, such as *SKIP*, will conceal the value of its intermediate states when it hasn't terminated. Any process with an **RT4** process as a right unit can engage in arbitrary behaviour after it has diverged, and so its precondition must be prefix closed. Additionally, any process which has *SKIP* as a right unit cannot terminate urgently: if it is possible to observe the process terminated after any behaviour, it was also possible to observe the process with the same trace before it had terminated.

Definition 2.13 (RT7)

$$\mathbf{RT7}(P) = P \,;\, \mathit{SKIP}$$

Lemma 2.1 (RT functions are commuting monotonic idempotents)

1. **RT1**–**RT7** *are all monotonic idempotents.*
2. **RT1**–**RT7** *all commute.*

Definition 2.14 (RT)

$$\mathbf{RT} \;\widehat{=}\; \mathbf{RT1} \circ \mathbf{RT2} \circ \mathbf{RT3} \circ \mathbf{RT4} \circ \mathbf{RT5} \circ \mathbf{RT6} \circ \mathbf{RT7}$$

We can now proceed to define the rest of our language. We define processes as timed reactive designs in the style of *Circus* (for an introduction to this style, see [19]).

In the definitions that follow, we make the assumption that any constituent processes in a process definition are themselves **RT**-healthy.

2.2. Deadlock

The inactive language construct is the deadlocked process: *STOP*. This process is an **RT**-healthy design with precondition true that never engages in any events and is perpetually waiting ($wait'$). In the postcondition, we also have that $events(tt') = \langle\rangle$: no events are ever observed. *STOP* deadlocks events but it cannot deadlock the clock, so refusal experiments can happen freely and no further trace restriction is required. *STOP* also says nothing about the final value of the program variables v', which are left unconstrained.

Definition 2.15 (Deadlock)

$$STOP = \mathbf{RT}(\mathbf{true} \vdash events(tt') = \langle\rangle \wedge wait')$$

2.3. Assignment

For the assignment $v := e$, we make the simplifying assumption that the expression e is well defined. The assignment takes place immediately and the process then terminates. This process has precondition *true* and a postcondition (which guarantees stability) that it has terminated ($\neg\, wait'$) without any events ($tt' = \langle\rangle$), but having completed the assignment ($v' = e$). Elements of state other than v are unaffected. This design is then made healthy with the application of the healthiness conditions.

Notice that since it is **RT7**, it will be possible to observe an assignment before it has terminated or for it to be preempted by other processes. Assignment can be used to represent *SKIP* as a reactive design, since $(v := v) = SKIP$.

Definition 2.16 (Assignment)

$$(v := e) = \mathbf{RT}(\mathbf{true} \vdash tt' = \langle\rangle \wedge \neg\, wait' \wedge v' = e)$$

2.4. Prefixed Termination

Prefixed termination is the process that is willing to perform a single, given event, and having done it terminates immediately. It has precondition *true*, and a postcondition that has two parts. Either the process is still waiting to engage in its event (a, say), in which case no events will occur and a will not be refused. Alternatively, the a has occurred, in which case it was the only event, and the process terminated immediately with the state unchanged. Prefixed termination is used together with relational composition in Section 2.8 to define the general prefix process $a \rightarrow P$.

Definition 2.17 ($a \rightarrow SKIP$)

$$a \rightarrow SKIP = \mathbf{RT}\left(\mathbf{true} \vdash \begin{pmatrix} a \notin refusals(tt') \wedge \\ \begin{pmatrix} events(tt') = \langle\rangle \\ \quad \triangleleft\, wait' \,\triangleright \\ \quad\quad tt' = idleprefix(tt') \frown \langle a\rangle \wedge v' = v \end{pmatrix} \end{pmatrix}\right)$$

2.5. Divergence

CHAOS is the least predictable process that satisfies the healthiness conditions. The precondition of *CHAOS* never holds, so its behaviour is always divergent.

Definition 2.18 (*CHAOS*)

$$CHAOS = \mathbf{RT}(\mathbf{false} \vdash \mathbf{true})$$

2.6. Specification Statement

CML inherits a specification statement from VDM as a way of describing operations and functions. If feasible, this may be refined into a CML action. If the post condition holds, the specification statement will terminate immediately and successfully. w is the framed variables: only these may be changed by the postcondition.

Definition 2.19 (Specification statement)

$$w : [\texttt{pre}\ P \quad \texttt{post}\ Q] = \mathbf{RT}(P \vdash Q \wedge \neg\, wait' \wedge tt' = \langle\rangle \wedge (v' \setminus w) = (v \setminus w))$$

2.7. Sequential Composition

Sequential composition of two reactive processes (written P $;_{\mathbf{RT}}$ Q) is simply relational composition, given our healthiness conditions. It is closed under the healthiness conditions. We will dispense with the subscript on this operator after the definition.

Definition 2.20 (Sequential composition)

$$P ;_{\mathbf{RT}} Q = P ; Q$$

The sequential composition of two designs has two parts to its precondition: there must be no possible values for v' and $wait'$ that would have allowed process P to diverge at any point along its trace ($\neg\ (P^f_f ;\ \mathbf{RT1}(\mathbf{true}))$) and the successful termination of P must not lead to a point where Q diverges ($\neg\ (P^t_f[false/wait'] ;\ \mathbf{RT1}(Q^f_f))$). Given the conjunction of these two conditions, P can be expected to terminate and lead to a stable observation of Q.

Lemma 2.2 (Precondition/Postcondition form)

$$P ; Q = \mathbf{RT}(\neg\ (P^f_f ;\ \mathbf{RT1}(\mathbf{true})) \wedge \neg\ (P^t_f[false/wait'] ;\ \mathbf{RT1}(Q^f_f)) \vdash P^t_f ;\ Q^{tt})$$

The abbreviation Q^{cd} is used to stand for $Q[c, d/ok, ok']$.

2.8. Prefix

The prefixed processes $a \rightarrow P$ is determined to engage in the event a and nothing else; after engaging in a it behaves like P. It is defined as a derived operator using prefixed termination (Section 2.4) and sequential composition (Section 2.7).

Definition 2.21 (Prefixing)

$$a \rightarrow P = a \rightarrow SKIP ; P$$

As a design, this operator is defined as follows: The only possibility of divergence of the process $a \rightarrow P$ is if the process P diverges, and this is possible only if the event a is the initial visible event of the trace: $\langle a \rangle \leq events(tt')$. In the period before this observation, a cannot be refused: $a \notin refusals(idleprefix(tt'))$. Provided that the trace subsequent to this does *not* cause P to diverge ($\neg\ P^f_f[idlesuffix(tt')/tt']$) then the postcondition will hold.

The postcondition describes the two possible states of $a \rightarrow P$. Either no events have been observed ($events(tt') = \langle\rangle$), in which case the process is waiting for input. The event a must not have been refused: $a \notin refusals(idleprefix(tt'))$, which is equivalent to $a \notin refusals(tt')$ because no event has been observed. The process cannot diverge in this case. Alternatively, an event has been observed and it must have been the a-event: the first event must be a. It is still the case that the event a must not be refused before it occurred, and after it occurs the process will continue as P. The future behaviour of the process is given by $P^t_f[idlesuffix(tt')/tt']$.

Lemma 2.3 (Precondition/Postcondition form)

$$a \rightarrow P = \mathbf{RT}\left(\begin{array}{l}((\langle a\rangle \leq events(tt') \wedge a \notin refusals(idleprefix(tt'))) \Rightarrow \\ \quad \neg\, P_f^f[idlesuffix(tt')/tt'] \\ \vdash \\ (a \notin refusals(idleprefix(tt')) \wedge \\ \quad \left(\begin{array}{l} events(tt') = \langle\rangle \\ \quad \triangleleft\, wait' \,\triangleright \\ \quad\quad \langle a\rangle \leq events(tt') \wedge P_f^t[idlesuffix(tt')/tt'] \end{array}\right)\end{array}\right)$$

2.9. Internal Choice

Internal choice is simply disjunction, as usual.

Definition 2.22 (Internal choice)

$$P \sqcap Q = P \vee Q$$

An internal choice can diverge if either of its component processes diverges, and guarantees termination only if both guarantee termination.

Lemma 2.4 (Precondition/Postcondition form)

$$P \sqcap Q = \mathbf{RT}(\neg\, P_f^f \wedge \neg\, Q_f^f \vdash P_f^t \vee Q_f^t)$$

2.10. External Choice

An external choice can be resolved if the process diverges, engages in a visible event or terminates. Either the observed behaviour is unresolved and does not diverge, in which case processes must agree on the observed behaviour, or something observable happens, the choice is resolved and the process which was responsible for observable event is now responsible for the subsequent observation.

However, in order to guarantee prefix closure of the observed behaviour, it must still be the case after the choice has resolved that the initial behaviours that didn't resolve the choice were compatible with both branches. The relevant behaviours are the prefixes of the trace which contain no events and on which the precondition of both branches still holds.

Definition 2.23 (External choice)

$$P \,\Box\, Q \;\widehat{=}\; \left(\begin{array}{l}(P \wedge Q \wedge events(tt') = \langle\rangle \wedge wait' \wedge ok' \\ \vee \\ (P \vee Q) \wedge \forall\, tt_0 \leq idleprefix(tt') \bullet \\ \quad\quad (\neg\, (P \wedge Q)^f \Rightarrow (P \wedge Q)^t)[tt_0, true/tt', wait']\end{array}\right)$$

The design form is, of course, more involved than internal choice.

The process $P \,\Box\, Q$ can diverge in any way that either of its branches could diverge, provided that initial prefixes of that behaviour do not violate the other branch of the choice. To enforce this condition, we require that on any initial behaviour which does not resolve the choice and satisfies the preconditions of both processes, it must be possible to satisfy the postconditions of both processes:

$$\forall\, tt_0 \leq idleprefix(tt') \bullet ((\neg\, P_f^f \wedge \neg\, Q_f^f) \Rightarrow (P_f^t \wedge Q_f^t))[tt_0, true/tt', wait']$$

If this condition holds, the behaviour is feasible and the preconditions of both branches must be satisfied: $\neg\, P_f^f \wedge \neg\, Q_f^f$.

Provided the process does not diverge, its behaviour must be compatible with at least one of its branches: $P_f^t \vee Q_f^t$. Additionally, it must satisfy both branches on any initial behaviour that does not resolve the choice: $(P_f^t \wedge Q_f^t)[idleprefix(tt'), true/tt', wait']$. The choice has not yet resolved when $events(tt') = \langle\rangle \wedge wait'$ holds, in which case this condition will insist that both branches still hold: $P_f^t \wedge Q_f^t$.

Lemma 2.5 (Precondition/postcondition form)

$$P \,\Box\, Q = \mathbf{RT}\left(\begin{array}{l}\left(\begin{array}{l}\forall\, tt_0 \leq idleprefix(tt') \bullet \\ \quad ((\neg\, P_f^f \wedge \neg\, Q_f^f) \Rightarrow (P_f^t \wedge Q_f^t))[tt_0, true/tt', wait'] \\ \quad \Rightarrow (\neg\, P_f^f \wedge \neg\, Q_f^f)\end{array}\right) \\ \vdash \\ (P_f^t \vee Q_f^t) \wedge (P_f^t \wedge Q_f^t)[idleprefix(tt'), true/tt', wait']\end{array}\right)$$

2.11. Parallel Composition

A parallel composition specifies the set of events that require synchronisation between two processes; outside this set events happen independently, without needing the participation of the other process. Parallel composition is then a form of restricted conjunction, where the behaviour of each process is seen as a projection of the overall trace.

We call two timed reactive designs *disjoint* if they share no programming variables; they are allowed, of course, to share the observational variables rt, $wait$, and ok. Parallel composition is restricted to disjoint processes. This rules out shared-variable parallelism.

The precondition of the parallel composition of P and Q is the conjunction of the preconditions of P and Q. The postcondition merges the intermediate or final states of the two processes. Since the program variables are partitioned, the equation $(v' = v)$ takes care of the appropriate merging of these programming variables, and we need worry only about merging the observational variables. The composition is in a waiting state if either of the processes end up in a waiting state. This is taken care of by taking the disjunction of their waiting states.

To take care of tt', we define a semantic operator on traces that *merges* a pair of traces together to give the set of traces that can result if the pair of traces are observed in parallel. To define this, we start by defining an intersection operator for refusal sets that will tell us what the refusal set will be for the parallel composition. Suppose that P has a refusal set X, Q has a refusal set Y, and A is the synchronisation alphabet. Our intersection operator (written $X \cap_A Y$) has three cases:

1. $X \cap A$: the set of synchronisation events refused by P.
2. $Y \cap A$: the set of synchronisation events refused by Q.
3. $X \cap Y$: the set of independent events refused by both P and Q.

Any subset of the union of these three sets is a refusal of the parallel composition of P and Q.

Definition 2.24 (Refusal set intersection)

$$X \cap_A Y \;\widehat{=}\; (X \cap A) \cup (Y \cap A) \cup (X \cap Y)$$

Now we are ready to define our semantic operator on timed testing traces. This is similar to the one defined in [3].

Definition 2.25 (Trace interleaving) *Let* $t, u \in timedTrace;\ a, b \in A;\ c, d \notin A;\ S, T \in \mathbb{P}\,\Sigma$

$$
\begin{array}{rl}
t \parallel_A u & = u \parallel_A t \\
\langle\rangle \parallel_A \langle\rangle & = \{\langle\rangle\} \\
\langle\rangle \parallel_A \langle b\rangle \frown u & = \{\} \\
\langle\rangle \parallel_A \langle d\rangle \frown u & = \{\, \langle d\rangle \frown v \mid v \in \langle\rangle \parallel_A u \,\} \\
\langle\rangle \parallel_A \langle T\rangle \frown u & = \{\, \langle T \cup U\rangle \frown v \mid U \subseteq A \wedge v \in t \parallel_A u \,\} \\
\langle a\rangle \frown t \parallel_A \langle a\rangle \frown u & = \{\, \langle a\rangle \frown v \mid v \in t \parallel_A u \,\} \\
\langle a\rangle \frown t \parallel_A \langle b\rangle \frown u & = \{\} \\
\langle a\rangle \frown t \parallel_A \langle d\rangle \frown u & = \{\, \langle d\rangle \frown v \mid v \in \langle a\rangle \frown t \parallel_A u \,\} \\
\langle a\rangle \frown t \parallel_A \langle T\rangle \frown u & = \{\} \\
\langle c\rangle \frown t \parallel_A \langle d\rangle \frown u & = \{\, \langle c\rangle \frown v \mid v \in t \parallel_A \langle d\rangle \frown u \,\} \cup \\
 & \qquad \{\, \langle d\rangle \frown v \mid v \in \langle c\rangle \frown t \parallel_A u \,\} \\
\langle c\rangle \frown t \parallel_A \langle T\rangle \frown u & = \{\, \langle c\rangle \frown v \mid v \in t \parallel_A \langle T\rangle \frown u \,\} \\
\langle S\rangle \frown t \parallel_A \langle T\rangle \frown u & = \{\, \langle U\rangle \frown v \mid U = S \cap_A T \wedge v \in t \parallel_A u \,\}
\end{array}
$$

The process *Skip* in parallel with an active process P will suspend termination until P finishes. They will both then synchronise on termination. The definition of alphabetised parallel operator, where process P can write to state variables ns_1, and process Q can write to state variables ns_2, and both processes synchronise on the channels cs, is given by

Definition 2.26

$$P\ [|\, ns_1 | cs | ns_2 \,|]\ Q \ \widehat{=}\ (P;\ U0(out\alpha(P))\ ||\ Q;\ U1(out\alpha(Q)))_{+\{v,tt\}};\ M_{\mathsf{CML}}$$

Here Ui are relabelling functions. They map each observational variable *obs* of their arguments to *i.obs*. This ensures that the two sides of the parallel composition do not share variables, and hence do not interfere with each other. Subscripting an action with a set of events adds that set of events to the alphabet: $R_{\{+n\}} = R \wedge n = n'$. Here we add the original values of v and tt back. M_{CML} is the CML merge function, and

Definition 2.27

$$M_{\mathsf{CML}} \ \widehat{=}\ \begin{pmatrix} (ok' = P.ok \wedge Q.ok) \wedge \\ (wait' = P.wait \vee Q.wait) \wedge \\ (tt' \in (P.rt - rt)\ ||_A\ (Q.rt - rt)) \wedge \\ (v' = merge(v, P.v, Q.v)) \end{pmatrix};\ Skip$$

The function *merge* merges the copies of state that the operands have taken. If process P is restricted to the name space ns_1, and process Q to ns_2, then *merge* is defined as

Definition 2.28 (merging state)

$$merge(v, v1, v2) = (ns_1 \lhd v_1) \cup (ns_2 \lhd v_2) \cup ((ns_1 \cup ns_2) \lhd\!\!\!\!- v)$$

where the domain restriction operator $ns \lhd v$ restricts the domain of the mapping v to the nameset ns, and the domain subtraction operator $ns \lhd\!\!\!\!- v$ removes the nameset ns from the domain of the mapping v. Using these new operators, we are in a position to define parallel composition.

A parallel composition will diverge on a trace tt' if it can be constructed as the trace composition of two traces tt_1 and tt_2, and one of the operands diverges at some point along either tt_1 or tt_2.

Lemma 2.6

$$
\begin{array}{l}
(P\ [|\, ns_1|cs|ns_2 \,|]\ Q)^f_f \\
= \\
\exists\, tt_1, tt_2 \bullet \begin{pmatrix} (P^f_f[tt_1/tt'] \wedge Q^t_f[tt_2/tt']);\ \textbf{\textit{RT1}}(true) \\ \vee \\ (P^t_f[tt_1/tt'] \wedge Q^f_f[tt_2/tt']);\ \textbf{\textit{RT1}}(true) \end{pmatrix} \wedge tt' \in tt_1 \|_A tt_2
\end{array}
$$

Definition 2.29 (Parallel composition)

$$
P\ [|\, ns_1|cs|ns_2 \,|]\ Q = \textbf{\textit{RT}} \left(\begin{array}{l}
\forall\, tt_1, tt_2 \bullet \\
(tt' \in tt_1\ ||_A\ tt_2) \Rightarrow \\
\qquad \neg \begin{pmatrix} ((P^f_f[tt_1/tt'] \wedge Q^t_f[tt_2/tt']);\ \textbf{\textit{RT1}}(true)) \\ \vee \\ ((P^t_f[tt_1/tt'] \wedge Q^f_f[tt_2/tt']);\ \textbf{\textit{RT1}}(true)) \end{pmatrix} \\
\vdash \\
\exists\, wait_1, wait_2, tt_1, tt_2 \bullet \\
\qquad tt' \in tt_1\ \|_A\ tt_2 \wedge \\
\qquad wait' = wait_1 \vee wait_2 \wedge \\
\qquad P^t_f[wait_1, tt_1, v_1/wait', tt', v'] \wedge \\
\qquad Q^t_f[wait_2, tt_2, v_2/wait', tt', v'] \wedge \\
\qquad v' = merge(v, v_1, v_2)
\end{array} \right)
$$

2.12. Interleaving Parallel

Interleaving of two processes is a straightforward derived operator: it is formed as the parallel composition of two processes, communicating on an empty set of events.

Definition 2.30 (Interleaving) *If P and Q may modify variables in ns_1 ns_2 respectively, and $ns_1 \cap ns_1 = \emptyset$, then*

$$P\ |||\ Q = P\ [|\, ns_1|\emptyset|ns_2 \,|]\ Q$$

2.13. Abstraction

The abstraction or hiding operator provides a way to abstract processes by internalising some events, thus making them unobservable by the environment. An assumption of maximal progress requires that no time may elapse whilst hidden events are on offer; they must happen as soon as they become available. Once more, the definition is given using semantic functions.

The assumption of maximal progress is modelled by considering only the A-urgent traces of P: the traces where all possible occurrences of every event in A happen as soon as they become available. In an A-urgent trace all events in the set A will appear in all refusal sets: no further events from A can be performed at any point.

Definition 2.31 (Urgency) *If $A \subseteq \Sigma$ and $t \in timedTrace$, then*

$$A\ \mathsf{urgent}\ t\ \widehat{=}\ \forall\, s, X \bullet s \frown \langle X \rangle \leq t \Rightarrow A \subseteq X$$

The semantic trace hiding operator is defined inductively:

Definition 2.32 (Trace hiding) *If $A, S \subseteq \Sigma$; $a \in A$; $b \notin A$, then*

$$
\begin{array}{l}
\langle\rangle \setminus A = \langle\rangle \\
(\langle S \rangle \frown tt) \setminus A = \langle S \setminus A \rangle \frown (tt \setminus A) \\
(\langle a \rangle \frown tt) \setminus A = tt \setminus A \\
(\langle b \rangle \frown tt) \setminus A = \langle b \rangle \frown (tt \setminus A)
\end{array}
$$

The behaviour of a process with an internalised set of events A is derived only from the A-urgent traces of the process:

Definition 2.33 (Hiding)

$$P \setminus A \;\widehat{=}\; \exists t \bullet P[t/tt'] \wedge A \text{ urgent } t \wedge tt' = t \setminus A$$

An abstracted process will diverge on an observed trace in the case where we are able to identify a prefix of the observed trace for which there is a corresponding observation (t) that (1) is an observation of the unabstracted process (A urgent $t \wedge tt' = t \setminus A$), and (2) represents a divergence of the unabstracted process ($P^f_f[t/tt']$). $\mathbf{RT1}(true)$ indicates that we do not care what happens in the latter potion of the trace. Obviously this is a case to avoid.

$$\neg\ ((\exists t \bullet P^f_f[t/tt'] \wedge A \text{ urgent } t \wedge tt' = t \setminus A)\ ;\ \mathbf{RT1}(true))$$

Only A-urgent observations of the unabstracted process contribute to the postcondition of the abstracted process.

$$(\exists t \bullet P^t_f[t/tt'] \wedge A \text{ urgent } t \wedge tt' = t \setminus A)$$

We form the design by ensuring that these observations are $\mathbf{RT}$ healthy.

$$P \setminus A = \mathbf{RT}\begin{pmatrix} \neg\ \Big((\exists t \bullet P^f_f[t/tt'] \wedge A \text{ urgent } t \wedge tt' = t \setminus A)\ ;\ \mathbf{RT1}(true)\Big) \\ \vdash \\ (\exists t \bullet P^t_f[t/tt'] \wedge A \text{ urgent } t \wedge tt' = t \setminus A) \end{pmatrix}$$

2.14. Recursion

Recursion is defined as the $\mathbf{RT}$-healthy least fixed point.

Definition 2.34 (Recursion)

$$\mu X \bullet F(X) = \mathbf{RT}(\sqcap\{\, P \mid F(P) \sqsubseteq P \,\})$$

2.15. Timeout

The timeout $P \overset{n}{\rhd} Q$ is the process which offers to behave as P for the first n time units. If P fails to begin communication by then, process Q silently takes over.

The precondition for the timeout process $P \overset{n}{\rhd} Q$ comes in two parts, to exclude two possible behaviours. The first part deals with the case where the process has waited up to n time units without any visible event. The behaviours that we wish to exclude here are those in which the precondition of P has failed. P's precondition will fail to hold on any trace that we can divide up into two $\mathbf{RT}$-healthy portions, the first of which is of duration less than or equal to n time units, at the end of which P^f_f holds. $\mathbf{RT1}(true)$ ensures that no constraints are imposed on the second part of the trace. Obviously, we do not want this situation.

$$\neg\ ((events(tt') = \langle\rangle \wedge \#tt' \leq n) \wedge P^f_f\ ;\ \mathbf{RT1}(\mathbf{true}))$$

The second part of the precondition excludes the case where P's precondition held successfully over an interval of n time units, but at that point P's postcondition fails to establish the precondition for Q:

$$\neg\ ((P^t_f \wedge events(tt') = \langle\rangle \wedge \#tt' = n)\ ;\ (wait \wedge Q^f_f))$$

This can only occur if Q is initiated when P's value for $wait'$ is *true*; in which case this value will be transferred (by the sequential composition) to *wait* for Q. Of course, Q is now initiated, so Q_f holds.

The postcondition for the timeout process $P \overset{n}{\rhd} Q$ also has two parts. It initially offers to behave like P. If this offer is taken up before n time units have passed, then the entire observable behaviour is due to P:

$$(P^t_f \wedge \#(idleprefix(tt')) \leq n)$$

If no events occur and P hasn't terminated in the first n time units, the combined process proceeds to behave like Q, initialised with an arbitrary choice of state. Allowing Q to begin with the state of the previous process would mean that the timeout operator allowed the state of deadlocked processes to be exposed, and so $(v := 1)$; *STOP* $\overset{2}{\rhd}$ *SKIP* would be distinguishable from $(v := 2)$; *STOP* $\overset{2}{\rhd}$ *SKIP*, even though $(v := 1)$; *STOP* is not distinguishable from $(v := 2)$; *STOP*.

$$\exists\, v_0 \bullet \Big((P^t_f \wedge events(tt') = \langle\rangle \wedge \#tt' = n \wedge v = v_0)\ ;\ (wait \wedge Q^t_f[v_0/v])\Big)$$

Note that the timeout operator is *non-strict* in the sense of [20,21]: events of P can be performed (unstably) between the nth and $n+1$th tock.

Definition 2.35 (Timeout)

$$P \overset{n}{\rhd} Q = \mathbf{RT}\begin{pmatrix} \neg\ \Big((events(tt') = \langle\rangle \wedge \#tt' \leq n) \wedge P^f_f\ ;\ \mathbf{RT1}(\mathbf{true})\Big) \\ \wedge \\ \neg\ \Big((P^t_f \wedge events(tt') = \langle\rangle \wedge \#tt' = n)\ ;\ (wait \wedge Q^f_f)\Big) \\ \vdash \\ (P^t_f \wedge \#(idleprefix(tt')) \leq n) \\ \vee \\ \exists\, v_0 \bullet \Big((P^t_f \wedge events(tt') = \langle\rangle \wedge \#tt' = n \wedge v = v_0)\ ;\ (wait \wedge Q^t_f[v_0/v])\Big) \end{pmatrix}$$

2.16. Untimed Timeout

The untimed variant of the timeout operator $P \rhd Q$ allows the first process to be timed out at any time. It is defined directly in the same way as the timeout operator (Section 2.15), but with no restriction on the point at which the timeout may occur.

Definition 2.36 (Untimed timeout)

$$P \rhd Q = \mathbf{RT}\begin{pmatrix} \neg\ ((events(tt') = \langle\rangle \wedge P^f_f)\ ;\ \mathbf{RT1}(\mathbf{true})) \\ \wedge \\ \neg\ ((P^t_f \wedge events(tt') = \langle\rangle)\ ;\ (wait \wedge Q^f_f)) \\ \vdash \\ P^t_f \\ \vee \\ \exists\, v_0 \bullet \Big((P^t_f \wedge events(tt') = \langle\rangle \wedge v = v_0)\ ;\ (wait \wedge Q^t_f[v_0/v])\Big) \end{pmatrix}$$

2.17. Wait

The delay operator is defined as a timeout process. It behaves like *STOP* for a specified number of time units, then terminates successfully.

Definition 2.37 (Delay)

$$Wait(n) = STOP \overset{n}{\rhd} SKIP$$

2.18. Interrupt

Like the timeout operator, the interrupt operator also has timed and untimed versions. Unlike the timeout operator, however, the basic form of the interrupt operator is the untimed version, written $P \triangle Q$. The untimed interrupt initially behaves like P, except that it cannot refuse to engage in the initial events of Q. If any of the initial events of Q occurs, the combined process starts to behave like Q.

We begin by defining the set of initial events of a process. An event a is an initial event of R if it is the initial event of a trace of R.

Definition 2.38

$$initials(R) = \{ a \mid \langle a \rangle \leq events(tt_0) \bullet R[tt_0/tt'] \}$$

The condition $notoffered(A, t)$ holds when no events in A have been offered during the observation of t. In this case, no events in A will have been observed or refused in the trace t.

Definition 2.39 *If $A \subseteq \Sigma$ and $t \in timedTrace$, then*

$$notoffered(A, t) = t \upharpoonright A = \langle\rangle \wedge A \cap refusals(t) = \emptyset$$

Like the timeout operator, the precondition comes in two parts to exclude two possible cases. The first is the case where we can divide the trace up into two parts, and identify an initial segment over which the initial events of Q have not been offered, and on which the precondition of P fails to hold. This is a case to exclude.

$$\neg\ (notoffered(initials(Q), tt') \Rightarrow P_f^f\ ;\ \mathbf{RT1}(true))$$

The second case to avoid is when the interrupt is triggered at a point at which P is failing to establish the precondition of Q. P must have held up to the point of interruption. As in the corresponding case for timeout, P must not have terminated after the first segment, so $wait'$ holds and this value is transferred via the relational composition to $wait$.

$$\neg\ ((P_f^t \wedge notoffered(initials(Q), tt'))\ ;\ (wait \wedge Q_f^f))$$

The postcondition describes two cases. In the first, P has terminated successfully (which is only possible if none of the initial events of Q were offered) and the relational composition transfers control to $SKIP$. In the second control is transferred via an initial event of Q. Q begins with a non-deterministic choice of state, again to avoid the possibility of exposing the state of a deadlocked process.

$$\exists\, v_0 \bullet ((P_f^t \wedge notoffered(initials(Q), tt'))\ ;\ (SKIP \vee (wait \wedge Q_f^t[v_0/v])))$$

These parts combine in the definition of the interrupt operator.

Definition 2.40 (Interrupt)

$$P \triangle Q = \mathbf{RT}\left(\begin{array}{l} \neg\ \Big(notoffered(initials(Q), tt') \Rightarrow P_f^f\ ;\ \mathbf{RT1}(true)\Big) \wedge \\ \neg\ \Big((P_f^t \wedge notoffered(initials(Q), tt'))\ ;\ (wait \wedge Q_f^f)\Big) \\ \vdash \\ \exists\, v_0 \bullet \Big((P_f^t \wedge notoffered(initials(Q), tt'))\ ;\ (SKIP \vee (wait \wedge Q_f^t[v_0/v]))\Big) \end{array}\right)$$

2.19. Timed Interrupt

The timed version of the interrupt operator allows the first process to continue for a specified number of time units, after which it will be interrupted by Q. Note that the timed interrupt operator is not invoked if the first process *terminates* before the time value, whereas a timeout operator is not invoked if the first process *begins* before the time value.

We begin with the definition of the *duration* of a trace, written $dur(t)$, as being the number of time units that elapse during a trace. This definition ignores the events in the trace, and so differs from the way that we calculate the passage of time in Section 2.15, where we know that no events have occurred.

Definition 2.41 *Let $A \subseteq \Sigma$, $a \in \Sigma$ and $t \in timedTrace$. Then*

$$dur(t) = \#(tocks(t) \upharpoonright \{tock\})$$

The precondition of timed interrupt has two parts, again disallowing two ways in which divergence can arise. The first is the case where an initial segment of the trace has a duration of less than or equal to the interrupt parameter, and leads to the case where P diverges.

$$\neg\ (dur(tt') \leq n \Rightarrow P_f^f\ ;\ \mathbf{RT1}(true))$$

The second case is the one where P fails to establish the precondition for P at the point of interruption.

$$\neg\ (dur(tt') = n \wedge P_f^t)\ ;\ (wait \wedge Q_f^f)$$

The postcondition also has two parts. Either the duration of the trace is less than the interrupt parameter, or P is interrupted and the subsequent behaviour is from Q.

Definition 2.42

$$P \overset{n}{\triangle} Q = \mathbf{RT}\begin{pmatrix} \neg\ \Big(dur(tt') \leq n \Rightarrow P_f^f\ ;\ \mathbf{RT1}(true)\Big) \\ \wedge \\ \neg\ (dur(tt') = n \wedge P_f^t)\ ;\ (wait \wedge Q_f^f) \\ \vdash \\ P_f^t \wedge dur(tt') \leq n \\ \vee \\ (P_f^t \wedge dur(tt') = n)\ ;\ (wait \wedge Q_f^t) \end{pmatrix}$$

During the first n time units control cannot transfer to Q, and P is unhindered. Either it terminates before the n time units, or at n time units it will be interrupted by Q.

2.20. Startsby

The *startsby* operator insists that a process begins communication by a deadline. The process $P\,startsby(n)$ behaves like P, and exhibits entirely unpredictable behaviour if P hasn't engaged in an event in the first n time units.

Definition 2.43 (startsby)

$$P\,startsby(n) = P \overset{n}{\triangleright} CHAOS$$

2.21. Endsby

The *endsby* operator insists that a process terminates by a deadline, otherwise it is entirely unpredictable.

Definition 2.44

$$P\,endsby(n) = P \overset{n}{\triangle} CHAOS$$

2.22. While

The while loop recursively behaves like P so long as the condition b holds. If b fails, it terminates. It is defined as a derived operator, using recursion and the conditional operator (defined in Section 1.)

Definition 2.45 (While loop)

$$b * P = \mu X \bullet (P \,;\, X) \lhd b \rhd SKIP$$

2.23. Guarded Actions

The guarded action $[g]\,\&P$ behaves as P if the guard g holds, otherwise it stops.

Definition 2.46

$$[g]\,\&\,P = P \lhd g \rhd STOP$$

3. A Leader Election Case Study

The leader election problem is a familiar one, met in many contexts. The example we describe comes from the audo-visual domain, in which multiple audio and video devices are synchronised. Maintaining and distributing the system configuration is the responsibility of a single device (the `leader`), that provides relevant information to other devices (`followers`) via a publish-subscribe architecture. If the chosen leader is switched off, or moves out of range, one of the followers must become the leader. The algorithm is biased to favour devices that have been live for the longest period of time. We present the algorithm in CML.

We first declare the types and values. `CLAIM` is the record-type of all claims that can be associated with a node: a node can be `leader`, `follower`, `undecided` or `off`. The strength of a node's claim to be a leader is a natural number. It increases as the node remains in leadership and is reset when the node finishes leading. The maximum value of this strength is given by `uls` (the upper limit of strength).

The record `MSG` is the type of message, containing source (`src`) and destination (`dest`) fields, and a payload of type `CLAIM`. Messages in transit are stored in the `MessageStore`.

```
values
  allNodes : set of nat1 = {1,...,3}
  uls :  nat  = 10
  n_timeout : nat = 1

types
  NODE =  nat1
   inv n == n in set allNodes

  CLAIM = <leader>|<follower>|<undecided>|<off>
  STRENGTH = nat
```

```
  CS :: c : CLAIM
        s : STRENGTH

  MSG :: src  : NODE
         dest : NODE
         cs   : CS
```

The channels `inn` and `out` are named from the point of view of the store, and correspond to messages going into and coming out of the store respectively. `init`, `on` and `off` allow the user to interact with the nodes in the system by initialising them, and turning them on and off.

```
channels
   inn, out : NODE * NODE * MSG
   on,off,init : nat1
```

The behaviour of the message store is given as the independent process `MessageStore`, which maintains an, initially empty, sequence of messages for each node. An operation that removes the head of one of the sequences is available, with the precondition that the sequence must not be empty.

```
process MessageStore =
begin
  state
   store : map NODE to seq of MSG := {i|->[]|i in set allNodes}

  operations
    RmvHdFrmStore:NODE ==> ()
    RmvHdFrmStore(i) ==
     store := store ++ {i |-> tl(store(i))}
    pre store(i) <> []
```

The behaviour of the message store is given by the single action `Loop`. Messages may be received through the channel `inn` in which case they are appended to the appropriate sequence. The fields `src` and `dest` are assumed to be consistent with the channel on which the message was received. Notice that here the message is added to the store directly within the action `Loop`, although this could have been encapsulated as another operation.

The message store also constantly offers each node the next message, if there are any available, on a first-in-first-out basis.

```
  actions
     Loop =
      inn?src?des?m -> store(m.dest) := store(m.dest)^[m]; Skip
      []
      ([] i in set allNodes @
        [store(i) <> []] &  -- there are message(s) for node i
           let m = hd(store(i)) in
            out!(m.dest)!(m.src)!m ->
                RmvHdFrmStore(i);
                  Skip)
    @ while true do Loop
end
```

Each `node` is given an identity parameter `id` on initialisation. The nodes communicate by sharing their current claim. Each node maintains (in `mem`) a record of the last claim made by each of the other nodes, if it is aware of any. The `mem` variable is periodically reset. The square brackets around the `CS` indicate that this type is optional. If the node is not aware of a claim since `mem` was reset, this takes the value `nil`. The invariant ensures that each node has a place in memory to record the claims of for all the other nodes and that there is at least one other node in the network.

The other variables summarise the information in mem. The node will make use of all of these when it is time to update the leadership claim. highest_strength is the strength of strongest leadership claim that the node is aware of from among the neighbouring nodes. highest_strength_id is the identity of the neighbouring node making the leadership claim with the highest strength. If more than one neighbouring node has the strongest claim, this will be the one with the highest id. leaders records the number of neighbouring nodes that are claiming leadership (with any strength of claim.) myCS is the current claim and strength of the node, and isleader is true if the node believes it should be a leader.

```
process Node = i : nat1 @ begin
state
  mem: map NODE to [CS] := { cid |-> nil | cid in set allNodes \ {i} }
   inv dom mem = allNodes \ {i} and dom mem <> {}

  highest_strength : [STRENGTH] := nil
  highest_strength_id : [NODE] := nil
    inv highest_strength_id <> nil => highest_strength_id in set (dom mem union {i})
  leaders : nat:= 0
    inv leaders <= card dom mem

  myCS : CS := mk_CS(<off>, 0)
  isleader : bool := false
```

The node has a set of operations for manipulating state from the within the actions.

The operation flushMemory resets the variable mem, and is used when the node is turned off. flushSummary resets the summary variables to their initialisation values. flushState applies both operations. The operation write receives a node and a claim, and updates mem(n) with that claim, overwriting any older claim. The precondition ensures that the node is in the domain of the memory, and the postcondition ensures that the update is correct.

```
operations
  flushState: () ==> ()
  flushState() ==
  (
    flushMemory();flushSummary()
  )

  flushMemory: () ==> ()
  flushMemory() ==
  (
    mem := { cid |-> nil | cid in set allNodes \ {i} }
  )

  flushSummary: () ==> ()
  flushSummary() ==
  (
    highest_strength := nil;
    highest_strength_id := nil;
    leaders := 0;
    myCS := mk_CS(<off>, 0);
    isleader := false
  )

  write: NODE * MSG ==> ()
  write(j,m) ==
  (
   mem := mem ++ {j |-> m.cs}
  )
  pre j in set dom mem
  post mem(j) = m.cs
```

The update operation updates the number of leaders, the highest strength observed, and the identity of the node with the highest strength, using the operations maxStrength and maxStrengthID. If more than one node has the strongest claim to be leader, maxStrengthID will return the one with the highest id.

```
update:()==>()
update() ==
(
  leaders := card{n|n in set dom mem @ mem(n)<>nil and mem(n).c = <leader>};
  highest_strength := maxStrength();
  highest_strength_id := maxStrengthID();
  isleader := amILeader()
)

maxStrength:() ==> [nat]
maxStrength() ==
(
 let
   leaderNodes = {n|n in set dom mem @ mem(n) <> nil and mem(n).c = <leader>} in
 (
  if leaderNodes <> {} then
  let strs = {mem(l).s|l in set leaderNodes} in
    if strs <> {} then return maxSet(strs)
    else return nil
  else return nil
 )
)

maxStrengthID : () ==> [NODE]
maxStrengthID() ==
(
 let
   leaderNodes = {n|n in set dom mem @ mem(n) <> nil and mem(n).c = <leader>} in
  (
   if leaderNodes = {} then return nil
   else
    let maxStrIds = {n | n in set leaderNodes @ mem(n).s = highest_strength} in
     return maxSet(maxStrIds)
  )
)
```

changeClaim updates the claim of a node. The precondition enforces the fact that nodes cannot transition directly from leading to following, they must go through a period of indecision; incStrength increases the strength of the node's leadership claim, up to the limit uls. The operation amILeader returns true if the summary information indicates that the node should be the leader. This is the case if: (i) there are no other nodes claiming to be leaders; (ii) the highest strength of any other claim is nil (in fact, this holds exactly when there are no other leaders); (iii) the strength of the highest claim is less than the strength of my claim; or (iv) another node has an equally strong claim, but it's identity is lower.

maxSet returns the maximum of a (non-empty) set of natural numbers. It uses the implicit function select to select an element from a set.

```
changeClaim: CLAIM ==> ()
changeClaim(newc) ==
(
  myCS.c := newc
)
pre myCS.c = <off> => newc = <undecided> and
    myCS.c = <undecided> => newc = <leader> or newc = <follower> and
    myCS.c = <leader> => newc = <undecided> and
```

```
     myCS.c = <follower> => newc = <undecided>

  changeStrength: nat ==> ()
  changeStrength(n) ==
    myCS := mk_CS(myCS.c,n)

  incStrength:()==>()
  incStrength() ==
   (
    if myCS.s < uls
    then myCS := mk_CS(myCS.c, myCS.s+1)
    else Skip
   )

  amILeader: () ==> bool
  amILeader() ==
  (return (leaders = 0)  or
          (highest_strength = nil) or
          (highest_strength <> nil and highest_strength < myCS.s) or
          (highest_strength <> nil and highest_strength = myCS.s
           and highest_strength_id < i)
  )

  maxSet: set of nat ==> nat
  maxSet(sn) ==
   (
    dcl s: set of nat @
      s:= sn;
    (
     dcl c: nat @
      c := select(s);
      s := s \ {c};
       while (s <> {}) do
         (let n = select(s) in
           (if (n>c) then c:=n else c:=c;
            s := s\{n})
         );
   return c
   )
  )
  pre sn <> {}

  select(sn : set of nat) r: nat
  pre sn <> {}
  post r in set sn
```

The four main states of a node are `Off`, `Undecided`, `Follower` and `Leader`. A node is initialised to `Off`, from which state it may receive an `on` signal and turn on. At this point, it enters the `Undecided` state, and the subsequent behaviour can be interrupted at any point by an `off` event.

```
Off =  on!i -> (Undecided /_\  off!i -> flushState();Off)
```

When the node is in state `Undecided`, it begins by changing its claim to match its state. It then flushes the volatile memory `mem` and listens to the network (via the action `Listener`.)

`Listener` begins by invoking the action `ReceiveData`, which receives messages from the other nodes and updates memory for `n_timeout` time units, then the operation `update` is called to update the summary variables. The node then uses the variable `isleader` in guard, moving to the `leader` state if this is set, and `follower` otherwise.

The behaviour is similar in states `Leader` and `Follower`. In each case the claim of the node is reported to the network before the memory is flushed. In state `Leader` the strength of the claim is increased for every successful round of leadership. In state `Follower` the strength of the leadership claim is set of zero (`changeStrength(0)`), and decision on whether or not to continue following is based on the number of leaders – when there is more than one the node becomes undecided.

```
  actions
  Undecided = changeClaim(<undecided>);undecided!i ->  flushMemory();Listener;
               (
                [isleader] & Leader
                []
                [not isleader] & Follower
               )

  Leader =  changeClaim(<leader>);leading.i -> SendCS; flushMemory();Listener;
             (
              [not isleader] & Undecided
              []
              [isleader] & incStrength();Leader
             )

  Follower = changeClaim(<follower>);changeStrength(0);
               following.i -> SendCS; flushMemory();Listener;
                 (
                  [leaders <> 1] & Undecided
                  []
                  [leaders = 1] & Follower
                 )
end
  Listener = ReceiveData;update();Skip

  ReceiveData =
    (
     out!i?j?msg -> (mem := mem ++ {j |-> msg.cs}); ReceiveData
    ) [_ n_timeout _> Skip

  SendCS = (||| t in set dom mem @ [{}] inn!i!t!(mk_MSG(i,t,myCS)) -> Skip)
  @ init.i -> flushState();Off
```

The nodes are combined in the process `AllNodes`. They communicate via the `MessageStore` on the channels `inn` and `out`, which are then hidden.

```
process AllNodes = ||| i in set allNodes @  (Node(i))
process Election = AllNodes [|{|inn,out|}|] MessageStore \\ {|inn,out|}
```

The leader election protocol that we describe is inspired by work with one of our industrial partners. We have deliberately attempted to exercise a substantial proportion of CML in the development of the model.

4. Related Work

4.1. Languages with UTP semantics

The work reported in this paper is part of a larger programme of research in defining the semantics of heterogeneous languages that started with UTP [10], which we describe in Section 1. Since the publication of Hoare & He's book, a number of heterogeneous languages have been successfully defined using UTP. These include:

1. *Circus*, which was first proposed as a concurrent language for refinement in 2001 [22]. The first semantics was given in UTP [4], and this was mechanised in the ProofPower interactive theorem prover [23,5].
2. Handel-C, which is a high-level programming language that targets low-level hardware and is most commonly used in the programming of FPGAs (field-programmable gate arrays). Handel-C is a rich subset of C with non-standard extensions to control hardware instantiation with an emphasis on parallelism. Reminiscent of occam, Handel-C is to hardware design what the first high-level programming languages were to programming computers. Unlike many other design languages that target a specific architecture, Handel-C can be compiled to a number of design languages and then synthesised for the desired hardware. This raises the level of abstraction for a developer, who can then largely ignore the idiosyncrasies of particular design notations and oddities of hardware architectures. UTP semantics are given in [15,24,25].
3. SCJ, Safety-Critical Java, a new language based on RTSJ (the Real-time Specification for Java), which is designed to support the creation of applications, infrastructures, and libraries that are amenable to certification under safety-critical standards such as DO-178B, Level A [26]. SCJ gives developers a simplified programming model based on missions (independent application tasks) that access restricted scoped memory. Threading, reflection, and class loading are restrained to reduce the code base and simplify certification. The SCJ specification is based around three levels of compliance to offer richer language facilities for more sophisticated applications [27]. The SCJ memory model has been formalised in UTP [28,17] and a development process created based on *Circus* [16,29].
4. CML, the COMPASS Modelling Language, which is the topic of this paper. Earlier descriptions of the language include [1,30]. The Symphony IDE is an open-source tool for constructing and reasoning about CML models and their refinements (see `symphonytool.org`).

4.2. Semantic heterogeneity

SoS are intrinsically heterogeneous, and this is particularly true for cyber-physical systems. Semantic heterogeneity is a significant challenge to integration in SoS Engineering (SoSE) due the large variety of languages, domains, and tools that are used in their construction. A strategy for managing this heterogeneity is to decompose domain-specific languages into their building-block theories that can be independently analysed and used as a basis for linking to similar notations. This provides a systematic approach to building tool-chains that integrate different theories, methods, and tools used in SoSE. An approach based on UTP has been piloted on the development of theories enabling machine-supported analysis of SysML models of SoSs [31].

A key objective of the semantically heterogeneous CML language is to be semantically open, allowing further paradigms to be added, such as process mobility, continuous physical models, and stochastic processes. The CML semantics deals separately with each paradigm, composing them with Galois connections, leading to a natural contract language for all constructs in all paradigms. The result is a compositional formal definition of a complex language, with the individual parts being available for reuse in other language definitions [32].

4.3. Semantic paradigms

In building the languages mentioned in Section 4.1, a number of different semantic paradigms have already been described in UTP.

4.3.1. State and Imperative Programming

A theory of total correctness for imperative programming, known as the theory of designs, appeared in the original book [10]. The consequences of the interaction between state visibility and communication have been explored in [9].

4.3.2. Definedness

CML includes VDM [2], where the treatment of undefined terms and predicates uses the Logic of Partial Functions [33]. For various practical reasons, CML uses McCarthy's left-to-right evaluation logic, so the question becomes: what is the relation between these two logics. The answer is given by unifying theories of undefinedness in UTP [34] and unifying theories of logic and specification [35].

4.3.3. Object Orientation

A unification of object classes and processes was carried out in *OhCircus* [36] and a theory of object orientation presented in [37]. rCOS, a refinement calculus of object systems, is described in [38,39,40]. Theories of pointers and Records in UTP have been explored in [41, 42,43,44]. A basic theory of separation logic in UTP can be found in [32]. A theory of higher-order UTP, which is required for a theory of methods, is worked out in [45].

4.3.4. Reactive Process

The theory of reactive processes is worked out in detail in [10], where it is used to describe the relationship between the process algebras ACP, CCS, and CSP. Further work on the semantics, its algebraic laws, and the mechanisation of the theory was carried out by Oliveira in his PhD thesis [46]. The notion of a reactive design first appeared here and in [19,47].

4.3.5. Time

A basic theory for a time model for *Circus* was first proposed by Sherif [8], together with a process algebraic framework for specification and validation of real-time systems using *Circus* Actions[48,49,50]. The semantic domain for CML is closely based on the work of Lowe & Ouaknine [7]. The consequences of using a complete lattice with a miracle as its top element was explored in [47,51].

4.3.6. Interrupt-driven Programming

Hoare has shown that an interrupt operator can be given a pleasing process algebraic semantics [52]. McEwan has produced a model in UTP that supports Hoare's algebraic laws in a unifying theories of interrupts [53].

4.3.7. Mobility

Tang has explored the idea of mobile CSP processes in unifying theories in [54,55]. The key idea is to model processes as first-class citizens by using higher-order predicates that specify behaviour, and that can be assigned to variables and transmitted over communication channels.

4.3.8. Angelic choice

Angelic nondeterminism was first described in UTP in [56,57]. It has subsequently been used in a development of Simulink timed models for program verification [58]. Ribeiro has gone on to build a theory of designs with angelic nondeterminism [59].

4.3.9. Probability

He & Sanders introduce a unification of probability with standard computation in which a non-zero chance of disaster is treated as disaster [60]. Laws and a Galois connection with a more traditional probabilistic model are provided and the formalism is applied to unify quantum computation and cryptography within the probabilistic method. Stoddart & Zeyda present a unification of probabilistic choice within a design-based model of reversible computation [61]. Zhu and his colleagues present a denotational semantics for a probabilistic timed shared-variable language [62] and Bresciani & Butterfield describe a probabilistic theory of designs based on distributions [63].

4.3.10. Security

Banks & Jacob have developed a theory of confidentiality in UTP. In [64], they present a framework for reasoning about the security of confidential data within software systems and show how information flow between users can be modelled. They devise conditions for verifying that system designs may not leak secret information to untrusted users. They also investigate how these conditions can be combined with existing notions of refinement to produce refinement relations suitable for deriving secure implementations of systems. In [65], they outline generic techniques for identifying the information that a user can deduce about the systems behaviour from observations. In [66], they introduce a notation for specifying which aspects of *Circus* processes are confidential and should not be revealed to low-level users. They also describe a novel procedure for verifying that a process satisfies its confidentiality properties.

4.3.11. Synchrony

Butterfield and his colleagues present a generic framework of UTP theories for describing systems whose behaviour is characterised by regular time-slots, compatible with the general structure of the *Circus* language [67]. This slotted *Circus* framework is parametrised by the particular way in which event histories are observable within a time-slot, and specifies what laws a desired parameterisation must obey in order for a satisfactory theory to emerge.

4.4. Mechanisation

Nuka & Woodcock present the first attempt at a formalisation of a subset of UTP as a deep embedding in Z with a corresponding mechanisation in ProofPowerZ[68]. Oliveira extends this approach using ProofPowerZ with a comprehensive treatment of the theories of alphabetised relations, designs, reactive processes, and CSP [69,70]. This work is complemented by Zeyda & Cavalcanti who show how a theory of UTP can be used to encode particular *Circus* specifications and their refinements. Feliachi and his colleagues develop a machine-checked, formal semantics based on a shallow embedding of *Circus* [71] in Isabelle/HOL [72]. They derive proof rules from and implement tactic support that allows for refinement proofs for *Circus* processes involving both data and behavioural aspects. Foster & Woodcock describe Isabelle/UTP, a mechanised theory engineering framework for UTP [73,74].

4.5. Refinement

Hoare & He define the notion of refinement as follows: program P is a refinement of specification S, providing every observed behaviour of P is also a behaviour of S [10]; they suggest that this should be a standard definition of refinement across all semantics paradigms. *Circus* was originally conceived as a language for describing concurrent system centred around refinement [22]. Sampaio et al. propose a sound data refinement technique for process refinement in *Circus* [75], propose laws for actions refinement [76], and present a general refine-

ment strategy for *Circus* [77]. Oliveira's PhD thesis focuses on formal derivation of state-rich reactive programs using *Circus* [46].

5. Conclusions and Future Work

We present in this paper the formal semantics of the COMPASS Modelling Language based on UTP. The work shows that a complicated language with features taken from many different paradigms can indeed be given a formal semantics and this approach scales up to industrial-strength modelling languages because of the compositional nature of UTP; see [32] for more details on how Galois connections are used to compose individual paradigms. Our technique is just one way of approaching the problem of semantic heterogeneity, an important challenge for engineering cyber-physical systems, but it does seem a promising one.

In the future, we propose to carry out the following work programme in heterogenous language development:

1. Complete the design of CML with both continuous and stochastic processes.
2. Complete the mechanisation of the UTP semantics for CML.
3. Construct a public repository of UTP theories and associated links that can be then be used to develop domain-specific languages and tool chains for SoSE, including CPS.
4. Construct an ontology for the concepts and vocabulary of SoSE by formalising them as UTP theories mechanised in Isabelle/UTP. The ontology will take the form of a web-based theory library, similar to the existing Archive of Formal Proofs (see `afp.sourceforge.net`), but using UTP theories and with a greater emphasis on theory composition and reuse.

The ultimate and rather ambitious objective for this work is to increase automation for SoSE based on heterogeneous languages for heterogeneous SoS. This will expose universal modelling concepts and allow the exploitation of large sets of libraries in several different physical domains. With rapid prototyping of multidisciplinary applications, the larger SoS community will benefit from these capabilities.

Acknowledgements

This work is supported by EU Framework 7 Integrated Project *Comprehensive Modelling for Advanced Systems of Systems* (COMPASS, Grant Agreement 287829); for more information see `www.compass-research.eu`. Ana Cavalcanti, Alvaro Miyazawa, Zoe Andrews, Uwe Assmann, Victor Bandur, Joey Coleman, John Fitzgerald, Klaus Kristensen, Peter Gorm Larsen, and Richard Payne all made important contributions to this work as it was being developed.

References

[1] Jim Woodcock, Ana Cavalcanti, John S. Fitzgerald, Peter Gorm Larsen, Alvaro Miyazawa, and S. Perry. Features of CML: a formal modelling language for systems of systems. In *7th International Conference on System of Systems Engineering, SoSE 2012, Genova, 16–19 July 2012*, pages 445–450. IEEE, 2012.

[2] C. B. Jones. *Systematic Software Development Using VDM*. Prentice Hall, 1986.

[3] A. W. Roscoe. *Understanding Concurrent Systems*. Springer, 2010.

[4] Jim Woodcock and Ana Cavalcanti. The semantics of *Circus*. In Didier Bert, Jonathan Bowen, Martin Henson, and Ken Robinson, editors, *ZB 2002:Formal Specification and Development in Z and B*, volume 2272 of *Lecture Notes in Computer Science*, pages 184–203. Springer, 2002.

[5] 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, 2009.

[6] *Circus* homepage. `www.cs.york.ac.uk/circus/`, accessed August 19, 2014.
[7] Gavin Lowe and Joël Ouaknine. On timed models and full abstraction. *Electr. Notes Theor. Comput. Sci.*, 155:497–519, 2006.
[8] Adnan Sherif and Jifeng He. Towards a time model for *Circus*. In Chris George and Huaikou Miao, editors, *Formal Methods and Software Engineering, 4th International Conference on Formal Engineering Methods, ICFEM 2002 Shanghai, 21–25 October 2002*, volume 2495 of *Lecture Notes in Computer Science*, pages 613–624. Springer, 2002.
[9] Andrew Butterfield, Pawel Gancarski, and Jim Woodcock. State visibility and communication in unifying theories of programming. In Wei-Ngan Chin and Shengchao Qin, editors, *TASE 2009, Third IEEE International Symposium on Theoretical Aspects of Software Engineering, 29–31 July 2009, Tianjin*, pages 47–54. IEEE Computer Society, 2009.
[10] C. A. R. Hoare and Jifeng He. *Unifying Theories of Programming*. Prentice Hall, 1998.
[11] Eric C. R. Hehner. Retrospective and prospective for Unifying Theories of Programming. In Steve Dunne and Bill Stoddart, editors, *Unifying Theories of Programming, First International Symposium, UTP 2006, Walworth Castle, 5–7 February 2006*, volume 4010 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2006.
[12] Naijun Zhan, Eun-Young Kang, and Zhiming Liu. Component publications and compositions. In Andrew Butterfield, editor, *Unifying Theories of Programming, Second International Symposium, UTP 2008, Dublin, 8–10 September 2008*, volume 5713 of *Lecture Notes in Computer Science*, pages 238–257. Springer, 2010.
[13] Juan Ignacio Perna, Jim Woodcock, Augusto Sampaio, and Juliano Iyoda. Correct hardware synthesis—an algebraic approach. *Acta Inf.*, 48(7–8):363–396, 2011.
[14] Huibiao Zhu, Peng Liu, Jifeng He, and Shengchao Qin. Mechanical approach to linking operational semantics and algebraic semantics for Verilog using Maude. In Burkhart Wolff, Marie-Claude Gaudel, and Abderrahmane Feliachi, editors, *Unifying Theories of Programming, 4th International Symposium, UTP 2012, Paris, 27–28 August 2012*, volume 7681 of *Lecture Notes in Computer Science*, pages 164–185. Springer, 2013.
[15] Andrew Butterfield. A denotational semantics for Handel-C. In Cliff B. Jones, Zhiming Liu, and Jim Woodcock, editors, *Formal Methods and Hybrid Real-Time Systems, Essays in Honor of Dines Bjørner and Chaochen Zhou on the Occasion of Their 70th Birthdays, Papers presented at a Symposium held in Macao, 24–25 September 2007*, volume 4700 of *Lecture Notes in Computer Science*, pages 45–66. Springer, 2007.
[16] Ana Cavalcanti, Andy J. Wellings, Jim Woodcock, Kun Wei, and Frank Zeyda. Safety-critical Java in *Circus*. In Andy J. Wellings and Anders P. Ravn, editors, *The 9th International Workshop on Java Technologies for Real-time and Embedded Systems, JTRES'11, York, 26–28 September 2011*, pages 20–29. ACM, 2011.
[17] Ana Cavalcanti, Andy J. Wellings, and Jim Woodcock. The Safety-critical Java memory model formalised. *Formal Asp. Comput.*, 25(1):37–57, 2013.
[18] Jim Woodcock. *Using Z—Specification, Refinement, and Proof*. Prentice Hall, 1996.
[19] Ana Cavalcanti and Jim Woodcock. A tutorial introduction to CSP in Unifying Theories of Programming. In Ana Cavalcanti, Augusto Sampaio, and Jim Woodcock, editors, *Refinement Techniques in Software Engineering, First Pernambuco Summer School on Software Engineering, PSSE 2004, Recife, 23 November–5 December 2004*, volume 3167 of *Lecture Notes in Computer Science*, pages 220–268. Springer, 2006.
[20] Joël Ouaknine. Discrete analysis of continuous behaviour in real-time concurrent systems. D Phil thesis, Oxford University, 2000.
[21] Joël Ouaknine. Digitisation and full abstraction for dense-time model checking. In *TACAS*, pages 37–51, 2002.
[22] Jim Woodcock and Ana Cavalcanti. A concurrent language for refinement. In Andrew Butterfield, Glenn Strong, and Claus Pahl, editors, *5th Irish Workshop on Formal Methods, IWFM 2001, Dublin, 16–17 July 2001*, Workshops in Computing. BCS, 2001.
[23] Marcel Oliveira, Ana Cavalcanti, and Jim Woodcock. A denotational semantics for *Circus*. *Electr. Notes Theor. Comput. Sci.*, 187:107–123, 2007.
[24] Juan Ignacio Perna and Jim Woodcock. A denotational semantics for Handel-C hardware compilation. In Michael Butler, Michael G. Hinchey, and María M. Larrondo-Petrie, editors, *Formal Methods and Software Engineering, 9th International Conference on Formal Engineering Methods, ICFEM 2007, Boca Raton, 14–15 September 2007*, volume 4789 of *Lecture Notes in Computer Science*, pages 266–285. Springer, 2007.
[25] Juan Ignacio Perna and Jim Woodcock. UTP semantics for Handel-C. In Andrew Butterfield, editor, *Unifying Theories of Programming, Second International Symposium, UTP 2008, Dublin, 8–10 Septem-*

ber 2008, volume 5713 of *Lecture Notes in Computer Science*, pages 142–160. Springer, 2010.

[26] D. Locke, B. S. Andersen, B. Brosgol, M. Fulton, T. Henties, J. J. Hunt, J. O. Nielsen, K. Nilsen, M. Schoeberl, J. Tokar, J. Vitek, and A. Wellings. *Safety Critical Java Specification, First Release 0.76*. The Open Group, 2010.

[27] Ales Plsek, Lei Zhao, Veysel H. Sahin, Daniel Tang, Tomas Kalibera, and Jan Vitek. Developing safety critical Java applications with oSCJ/L0. In Tomas Kalibera and Jan Vitek, editors, *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems, JTRES 2010, Prague, 19–21 August 2010*, ACM International Conference Proceeding Series, pages 95–101, 2010.

[28] Ana Cavalcanti, Andy J. Wellings, and Jim Woodcock. The Safety-critical Java memory model: a formal account. In Michael Butler and Wolfram Schulte, editors, *FM 2011: Formal Methods—17th International Symposium on Formal Methods, Limerick, 20–24 June 2011*, volume 6664 of *Lecture Notes in Computer Science*, pages 246–261. Springer, 2011.

[29] Ana Cavalcanti, Frank Zeyda, Andy J. Wellings, Jim Woodcock, and Kun Wei. Safety-critical Java programs from *Circus* models. *Real-Time Systems*, 49(5):614–667, 2013.

[30] Jeremy Bryans, Samuel Canham, and Jim Woodcock. CML Definition. Public Document Deliverable Number: D23.4, Version: 1.0, COMPASS Project, University of York, March 2013.

[31] 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.

[32] Jim Woodcock. Engineering UToPiA—formal semantics for CML. In Cliff B. Jones, Pekka Pihlajasaari, and Jun Sun, editors, *FM 2014: Formal Methods—19th International Symposium, Singapore, 12–16 May 2014*, volume 8442 of *Lecture Notes in Computer Science*, pages 22–41. Springer, 2014.

[33] Howard Barringer, J. H. Cheng, and Cliff B. Jones. A logic covering undefinedness in program proofs. *Acta Inf.*, 21:251–269, 1984.

[34] Jim Woodcock and Victor Bandur. Unifying theories of undefinedness in UTP. In Burkhart Wolff, Marie-Claude Gaudel, and Abderrahmane Feliachi, editors, *Unifying Theories of Programming, 4th International Symposium, UTP 2012, Paris, 27–28 August 2012*, volume 7681 of *Lecture Notes in Computer Science*, pages 1–22. Springer, 2013.

[35] Victor Bandur and Jim Woodcock. Unifying theories of logic and specification. In Juliano Iyoda and Leonardo Mendonca de Moura, editors, *Formal Methods: Foundations and Applications—16th Brazilian Symposium, SBMF 2013, Brasilia, 29 September–4 October 2013*, volume 8195 of *Lecture Notes in Computer Science*, pages 18–33. Springer, 2013.

[36] Ana Cavalcanti, Augusto Sampaio, and Jim Woodcock. Unifying classes and processes. *Software and System Modeling*, 4(3):277–296, 2005.

[37] Thiago L. V. L. Santos, Ana Cavalcanti, and Augusto Sampaio. Object-orientation in the UTP. In *[78]*, pages 18–37, 2006.

[38] Zhiming Liu, Jifeng He, and Xiaoshan Li. rCOS: Refinement of component and object systems. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, Third International Symposium, FMCO 2004, Leiden, The Netherlands, 2–5 November 2004*, volume 3657 of *Lecture Notes in Computer Science*, pages 183–221. Springer, 2005.

[39] Jifeng He, Zhiming Liu, Xiaoshan Li, and Shengchao Qin. A relational model for object-oriented designs. In Wei-Ngan Chin, editor, *Programming Languages and Systems: Second Asian Symposium, APLAS 2004, Taipei, Taiwan, 4–6 November 2004*, volume 3302 of *Lecture Notes in Computer Science*, pages 415–436. Springer, 2004.

[40] Jifeng He, Xiaoshan Li, and Zhiming Liu. rCOS: A refinement calculus of object systems. *Theor. Comput. Sci.*, 365(1–2):109–142, 2006.

[41] C. A. R. Hoare and Jifeng He. A trace model for pointers and objects. In Rachid Guerraoui, editor, *ECOOP'99—Object-Oriented Programming, 13th European Conference, Lisbon, Portugal, 14–18 June 1999*, volume 1628 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 1999.

[42] Ana Cavalcanti, Will Harwood, and Jim Woodcock. Pointers and records in the Unifying Theories of Programming. In *[78]*, pages 200–216, 2006.

[43] Michael Anthony Smith and Jeremy Gibbons. Unifying theories of locations. In *[79]*, pages 161–180, 2008.

[44] Will Harwood, Ana Cavalcanti, and Jim Woodcock. A theory of pointers for the UTP. In John S. Fitzgerald, Anne Elisabeth Haxthausen, and Hüsnü Yenigün, editors, *Theoretical Aspects of Computing—ICTAC 2008, 5th International Colloquium, Istanbul, 1–3 September 2008*, volume 5160 of *Lecture Notes in Computer Science*, pages 141–155. Springer, 2008.

[45] Frank Zeyda and Ana Cavalcanti. Higher-order UTP for a theory of methods. In *[80]*, pages 204–223, 2012.

[46] M. V. M. Oliveira. *Formal derivation of state-rich reactive programs using Circus*. Phd thesis, Department of Computer Science, University of York, Report YCST-2006/02 2005.
[47] Jim Woodcock. The miracle of reactive programming. In *[79]*, pages 202–217, 2008.
[48] Adnan Sherif, Jifeng He, Ana Cavalcanti, and Augusto Sampaio. A framework for specification and validation of real-time systems using *Circus* actions. In Zhiming Liu and Keijiro Araki, editors, *Theoretical Aspects of Computing—ICTAC 2004, First International Colloquium, Guiyang, 20–24 September 2004*, volume 3407 of *Lecture Notes in Computer Science*, pages 478–493. Springer, 2005.
[49] Adnan Sherif. *A Framework for Specification and Validation of Real-Time Systems using Circus Actions*. PhD thesis, Center for Informatics, Federal University of Pernambuco, Brazil, 2006.
[50] Adnan Sherif, Ana Cavalcanti, Jifeng He, and Augusto Sampaio. A process algebraic framework for specification and validation of real-time systems. *Formal Asp. Comput.*, 22(2):153–191, 2010.
[51] Kun Wei, Jim Woodcock, and Ana Cavalcanti. *CircusTime* with reactive designs. In *[80]*, pages 68–87, 2012.
[52] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
[53] Alistair A. McEwan and Jim Woodcock. Unifying theories of interrupts. In *[79]*, pages 122–141, 2008.
[54] Xinbei Tang and Jim Woodcock. Towards mobile processes in Unifying Theories. In *2nd International Conference on Software Engineering and Formal Methods (SEFM 2004), 28–30 September 2004, Beijing*, pages 44–53. IEEE Computer Society, 2004.
[55] Xinbei Tang and Jim Woodcock. Travelling processes. In Dexter Kozen and Carron Shankland, editors, *Mathematics of Program Construction, 7th International Conference, MPC 2004, Stirling, Scotland, 12–14 July 2004*, volume 3125 of *Lecture Notes in Computer Science*, pages 381–399. Springer, 2004.
[56] Ana Cavalcanti and Jim Woodcock. Angelic nondeterminism and Unifying Theories of Programming. *Electr. Notes Theor. Comput. Sci.*, 137(2):45–66, 2005.
[57] Ana Cavalcanti, Jim Woodcock, and Steve Dunne. Angelic nondeterminism in the Unifying Theories of Programming. *Formal Asp. Comput.*, 18(3):288–307, 2006.
[58] Ana Cavalcanti, Alexandre Mota, and Jim Woodcock. Simulink timed models for program verification. In Zhiming Liu, Jim Woodcock, and Huibiao Zhu, editors, *Theories of Programming and Formal Methods—Essays Dedicated to Jifeng He on the Occasion of His 70th Birthday*, volume 8051 of *Lecture Notes in Computer Science*, pages 82–99. Springer, 2013.
[59] Pedro Ribeiro and Ana Cavalcanti. Designs with angelic nondeterminism. In *Seventh International Symposium on Theoretical Aspects of Software Engineering, TASE 2013, 1–3 July 2013, Birmingham*, pages 71–78. IEEE, 2013.
[60] Jifeng He and Jeff W. Sanders. Unifying probability. In *[78]*, pages 173–199, 2006.
[61] Bill Stoddart and Frank Zeyda. A unification of probabilistic choice within a design-based model of reversible computation. *Formal Asp. Comput.*, 25(1):107–131, 2013.
[62] Huibiao Zhu, Jeff W. Sanders, Jifeng He, and Shengchao Qin. Denotational semantics for a probabilistic timed shared-variable language. In *[80]*, pages 224–247, 2012.
[63] Riccardo Bresciani and Andrew Butterfield. A probabilistic theory of designs based on distributions. In *[80]*, pages 105–123, 2012.
[64] Michael J. Banks and Jeremy L. Jacob. Unifying theories of confidentiality. In *[81]*, pages 120–136, 2010.
[65] Michael J. Banks and Jeremy L. Jacob. On modelling user observations in the UTP. In *[81]*, pages 101–119, 2010.
[66] Michael J. Banks and Jeremy L. Jacob. Specifying confidentiality in *Circus*. In Michael Butler and Wolfram Schulte, editors, *FM 2011: Formal Methods—17th International Symposium on Formal Methods, Limerick, 20–24 June 2011*, volume 6664 of *Lecture Notes in Computer Science*, pages 215–230. Springer, 2011.
[67] Andrew Butterfield, Adnan Sherif, and Jim Woodcock. Slotted *Circus*. In Jim Davies and Jeremy Gibbons, editors, *Integrated Formal Methods, 6th International Conference, IFM 2007, Oxford, 2–5 July 2007*, volume 4591 of *Lecture Notes in Computer Science*, pages 75–97. Springer, 2007.
[68] Gift Nuka and Jim Woodcock. Mechanising a unifying theory. In *[78]*, pages 217–235, 2006.
[69] Marcel Oliveira, Ana Cavalcanti, and Jim Woodcock. Unifying Theories in ProofPower-Z. In *[78]*, pages 123–140, 2006.
[70] Marcel Oliveira, Ana Cavalcanti, and Jim Woodcock. Unifying theories in ProofPower-Z. *Formal Asp. Comput.*, 25(1):133–158, 2013.
[71] Abderrahmane Feliachi, Marie-Claude Gaudel, and Burkhart Wolff. Unifying Theories in Isabelle/HOL. In *[81]*, pages 188–206, 2010.
[72] T. Nipkow, M. Wenzel, and L. C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
[73] Simon Foster and Jim Woodcock. Unifying Theories of Programming in Isabelle. In Zhiming Liu,

Jim Woodcock, and Huibiao Zhu, editors, *Unifying Theories of Programming and Formal Engineering Methods—International Training School on Software Engineering, Held at ICTAC 2013, Shanghai, 26–30 August 2013*, volume 8050 of *Lecture Notes in Computer Science*, pages 109–155. Springer, 2013.

[74] S. Foster, F. Zeyda, and J. Woodcock. Isabelle/UTP: a mechanised theory engineering framework. In *5th International Symposium on Unifying Theories of Programming (To appear)*, 2014.

[75] Augusto Sampaio, Jim Woodcock, and Ana Cavalcanti. Refinement in *Circus*. In Lars-Henrik Eriksson and Peter A. Lindsay, editors, *FME 2002: Formal Methods—Getting IT Right, International Symposium of Formal Methods Europe, Copenhagen, 22–24 July 2002*, volume 2391 of *Lecture Notes in Computer Science*, pages 451–470. Springer, 2002.

[76] Ana Cavalcanti, Augusto Sampaio, and Jim Woodcock. Refinement of actions in *Circus*. *Electr. Notes Theor. Comput. Sci.*, 70(3):132–162, 2002.

[77] Ana Cavalcanti, Augusto Sampaio, and Jim Woodcock. A refinement strategy for *Circus*. *Formal Asp. Comput.*, 15(2–3):146–181, 2003.

[78] Steve Dunne and Bill Stoddart, editors. *Unifying Theories of Programming, First International Symposium, UTP 2006, Walworth Castle, County Durham, 5–7 February 2006*, volume 4010 of *Lecture Notes in Computer Science*. Springer, 2006.

[79] Andrew Butterfield, editor. *Unifying Theories of Programming, Second International Symposium, UTP 2008, Dublin, 8–10 September 2008*, volume 5713 of *Lecture Notes in Computer Science*. Springer, 2010.

[80] Burkhart Wolff, Marie-Claude Gaudel, and Abderrahmane Feliachi, editors. *Unifying Theories of Programming, 4th International Symposium, UTP 2012, Paris, 27–28 August 2012*, volume 7681 of *Lecture Notes in Computer Science*. Springer, 2013.

[81] Shengchao Qin, editor. *Unifying Theories of Programming, Third International Symposium, UTP 2010, Shanghai, 15–16 November 2010*, volume 6445 of *Lecture Notes in Computer Science*. Springer, 2010.