The Denotational Semantics of slotted-Circus*

Paweł Gancarski[1] and Andrew Butterfield[1]

Trinity College Dublin, Andrew.Butterfield@cs.tcd.ie

Abstract. This paper describes a complete denotational semantics, in the UTP framework, of *slotted-Circus*, a generic framework for reasoning about discrete timed/synchronously clocked systems. The key result presented here is a comprehensive semantics of the entire language that addresses various semantics issues that have been uncovered, whilst laying foundations for future extensions, particularly towards prioritized choice.

1 Introduction

1.1 Circus and slotted-Circus

The formal notation *Circus* is a unification of Z and CSP, and has been given a UTP semantics [OCW09]. A *Circus* text describes behaviour as a collection of actions, which are a combination of processes with mutable state. However, apart from event sequencing, there is no notion of time in *Circus*. A timed version of *Circus* (*Circus* Time Action or CTA) has been explored [SH02, She06] that introduces the notion of discrete time-slots in which sequences of events occur. The semantics of CTA has been developed using UTP, and there we find a two-level notion of history: the top-level views history as a sequence of time-slots; whilst the bottom-level records a history of events within a given slot.

Our interest in hardware compilation languages such as Handel-C [Cel02] led to a development of semantic theories based on the notion of time-slots in CTA, but with much more structure ("microslots") to the events within the timeslots [BW05]. Looking for a way to formally link *Circus* as a specification language to Handel-C as an implementation language, and given that CTA was a step in this direction, we decided to explore a UTP semantics for Handel-C.

As the Handel-C semantics had three levels of complexity, each supporting a larger range of language features, it was decided to develop a generic theory (called *slotted-Circus*), with time-slots whose bottom-level contents could be parameterised, as simple traces, or multisets of events, or as one of the three successively more complex "micro-slot" structures [BSW07]. That paper discussed a number of fundamental issues that had to be addressed, most regarding healthiness conditions. More recent work [GBW09] looked at subtleties involving communication and state update.

* This research was supported by a grant from Science Foundation Ireland, as well as partial support from Lero, the Irish Software Engineering Research Centre

Another reason for using UTP was that it will allow us, in the future, to explore refinement links to other specification/programming languages also treated using the UTP framework.

This paper describes a complete denotational semantics, in the UTP framework, of *slotted-Circus*, finishing off earlier work. The key contribution here, apart from the completion, is an understanding of the key role played by refusals in the theory, particularly with respect to the semantics of hiding.

1.2 UTP: General Principles

Theories in UTP are expressed as second-order predicates[1] over a pre-defined collection of free observation variables, referred to as the *alphabet* of the theory. The predicates are generally used to describe a relation between a before-state and an after-state, the latter typically characterised by dashed versions of the observation variables. A predicate whose free variables are all undashed, referring only to the before-state, is called a *condition*. So for example, the program below on the left could be described by the predicate on the right:

$$\mathtt{f} := \mathtt{f} * \mathtt{x};\ \mathtt{x} := \mathtt{x} - 1 \qquad\qquad f' = f * x \land x' = x - 1$$

Here logical variables f and f' denote the before- and after-values of the program variable `f`. We note that UTP follows the key principle that "programs are predicates" [Hoa85b] and so does not distinguish between the syntax of some language and its semantics as alphabetised predicates. In practise, we also need auxiliary logical variables to capture other aspects of a programs behaviour. For example, in a theory of simple imperative programming, we might use ok and ok' to model respectively the successful start and termination of a program. Our above example would then have its full semantics as follows:

$$ok \Rightarrow (\ ok' \land f' = f * x \land x' = x - 1\ )$$

A given theory is characterised by its alphabet, and a series of *healthiness conditions* that constrain the valid assertions that predicates may make. A healthiness condition is a property of a predicate that distinguishes sensible predicates from nonsense. So, for example the following predicate is clearly nonsense under our intended interpretation:

$$\neg\ ok \land ok'$$

It asserts that a program has not been started, but yet has terminated! It can be ruled out by the following healthiness condition (which yields false for the above predicate):

$$P = (ok \Rightarrow P)$$

Note that healthiness conditions should not be confused with ordinary conditions (predicates with only before-variables).

[1] Most definitions are in fact 1st-order, but we need 2nd-order in order to handle the notion of "healthiness", and recursion.

$$
\begin{array}{rl}
\mathsf{Action} ::= & \mathit{Skip} \mid \mathit{Stop} \mid \mathit{Chaos} \mid \mathit{Wait}\ \mathsf{t} \\
\mid & \mathsf{Comm} \rightarrow \mathsf{Action} \mid \mathsf{Action} \sqcap \mathsf{Action} \mid \mathsf{Action} \,\Box\, \mathsf{Action} \\
\mid & \mathsf{Action} \,[\![\, \mathsf{VS} \mid \mathsf{CS} \mid \mathsf{VS} \,]\!]\, \mathsf{Action} \mid \mathsf{Action} \setminus \mathsf{CS} \mid \mu\, \mathsf{Name} \bullet F(\mathsf{Name}) \\
\mid & \mathsf{Name}^+ := \mathsf{Expr}^+ \mid \mathsf{Action}\,;\ \mathsf{Action} \mid \mathsf{Action} \lhd \mathsf{Expr} \rhd \mathsf{Action} \mid \mathsf{Expr} * \mathsf{Action} \\
\mathsf{Comm} ::= & \mathsf{Name.Expr} \mid \mathsf{Name!Expr} \mid \mathsf{Name?Name} \\
\mathsf{Expr} ::= & \text{expression} \\
\mathsf{t} ::= & \text{positive integer valued expression} \\
\mathsf{Name} ::= & \text{channel or variable names} \\
\mathsf{CS} ::= & \text{channel name sets} \\
\mathsf{VS} ::= & \text{variable sets}
\end{array}
$$

Fig. 1. Slotted-*Circus* Syntax

1.3 Structure and Focus

The main technical emphasis of this paper is on the details of the semantics definitions of the language constructs, to ensure that the desired laws can be verified. We first present the syntax §2, generic framework §3, and healthiness conditions §4. We then discuss semantics §5 in some detail, and present some laws with a sketch of the proof of one of interest §6. We finish by mentioning related §7 and future §8 work, and concluding §9.

2 Syntax

The syntax of Slotted-*Circus* is similar to that of *Circus*, and a subset, relevant to this paper, is shown in Figure 1. The notation X^+ denotes a sequence of one of more X. We assume an appropriate syntax for describing expressions and their types, subject only to the proviso that at least booleans and non-negative integers are included.

The basic actions *Skip*, *Stop*, *Chaos*, as well as event prefix ($e \rightarrow A$) and hiding ($A \setminus H$) are similar to the corresponding CSP behaviours [Hoa85a, Sch00], while we also introduce variable assignmement ($:=$). Actions can be combined with internal ($\sqcap$) or external ($\Box$) choice, sequential composition ($;$), parallel composition ($[\![\_ \mid \_ \mid \_]\!]$), or conditional choice ($\lhd c \rhd$). Iteration can be described explicitly ($*$), or defined recursively ($\mu \_ \bullet \_$). The key construct related to time-slots, and hence not part of *Circus*, is *Wait* t which denotes an action that simply waits for t time-slots to elapse, and then terminates.

As an example we present a simple one-shot"factorial server" (Fig. 2) that waits for a request (channel *freq*) containing a natural number n, and then computes its factorial, exploiting parallelism where possible, finally returning the result as a response message (channel *fresp*). The server has the timing of a Handel-C program, where each assignment and channel communication takes a full time-slot (a.k.a. "clock-cycle"). A run of the server showing communication events, state variable changes, and the passage of time-slots is shown in Figure

$$FS \mathrel{\widehat=} \mathit{freq}?n \rightarrow \mathit{Wait}\,1\,;\ \ \mathit{FCOMP}\,;\ \ \mathit{fresp}!f \rightarrow \mathit{Wait}\,1$$
$$FCOMP \mathrel{\widehat=} f := 1\,;\ \ \mathit{Wait}\,1\,;\ (n > 1) * ((f := n * f\,;\ \ \mathit{Wait}1) \,[\![\, \{f\} \mid \emptyset \mid \{n\} \,]\!]\, (n := n - 1\,;\ \ \mathit{Wait}1))$$

Fig. 2. Factorial Server

Slot	1	2	3	4	5	6	7	8	9
Event	–	*freq*.4	–	–	–	–	–	–	*fresp*.24
Var:n	–	4	4	3	2	1	1	1	1
Var:f	–	–	1	4	12	24	24	24	24

Fig. 3. Factorial Server Run

3. Here we wait one slot for a request to compute 4!, and the client looks for the result two slots after it becomes available.

3 Generic Slot-Theory

Both the semantics of Handel-C [BW05] and the timed extension to *Circus* called "Circus Timed Actions (CTA)" [SH02, She06] have in common the fact that the models involve a sequence of "slots" that capture the behaviour of the system between successive clock ticks. In [BSW07] a comprehensive account is given of a generic UTP framework that captures the common aspects of these semantic models. The reason for developing a generic slot theory is that the way that events are recorded within a slot in CTA and Handel-C differ, with the latter semantics itself having three distinct variants. Here we provide a summary of the key concepts involved.

Although we are modelling a "discrete-time" theory, it has to be stressed that we can allow events to be ordered within a time-slot, albeit without timestamps at a finer granularity. The key concept is of a system governed by a global clock, and each slot models all that happens in-between two consecutive clock ticks. A slot contains information about the events that occurred during one time slot ("history") as well as the events being refused at that point. In CTA, a history is just a sequence ("trace") of events in the order in which they occurred during a slot. So the following example shows a run of CTA where events a and b both occur at least once in some order in every second time-slot:

$$\langle \langle\rangle, \langle a, b\rangle, \langle\rangle, \langle b, a, a\rangle, \langle\rangle, \langle b, a\rangle, \langle\rangle, \ldots \rangle$$

In the multi-set action (MSA) variant, we ignore event ordering within slots, viewing history as a bag of events, so the above example appears as

$$\langle \{\}, \{a \mapsto 1, b \mapsto 1\}, \{\}, \{a \mapsto 2, b \mapsto 1\}, \{\}, \{a \mapsto 1, b \mapsto 1\}, \{\}, \ldots \rangle$$

In fact with each slot we not only record an event history of some form but also the events being refused during a time-slot. So if we have an event type E and a history type constructor $\mathcal{H}$, then the type of slots is defined as:

$$\mathcal{S} \mathrel{\widehat=} \mathcal{H}\, E \times \mathbb{P}\, E$$

Essentially we now have a semantic domain that is parametric in the choice of $\mathcal{H}$ (plus some supporting definitions). We then build up event observations as "slotted-sequences", which are non-empty sequences of slots. The presence of clock-ticks in the history is denoted by the adjacency of two slots, so a slot-sequence of length $n+1$ describes a situation in which the clock has ticked n times. The CTA example above can now be written in full, assuming that neither a nor b are refused during slots when they don't occur, but are refused at the end of the slot in which they do occur:

$$\langle (\langle\rangle, \emptyset), (\langle a, b\rangle, \{a, b\}), (\langle\rangle, \emptyset), (\langle b, a, a\rangle, \{a, b\}), (\langle\rangle, \emptyset), (\langle b, a\rangle, \{a, b\}), (\langle\rangle, \emptyset), \ldots \rangle$$

We can now describe the observational variables of our generic UTP theory:

$ok : \mathbb{B}$ —True if the process is stable, *i.e.*, not diverging.
$wait : \mathbb{B}$ —True if the process is waiting, *i.e.*, not terminated.
$state : Var \pfun Value$ —An environment giving the current values of slotted-*Circus* variables
$slots : \mathcal{S}^+$: —A non-empty sequence of slots recording the timed event behaviour of the system.

The variables *ok*, *wait* play the same role as the in the reactive systems theory in [HH98, Chp. 8], while *state* follows the trend in [SH02] of grouping all the program variables under one observational variable, to simplify the presentation of the theory. We need to be very clear about the distinction between events and program variables — events denote visible communication actions used for synchronisation and/or to transfer data, whilst program variables are considered global in this paper, and the *state* component tracks their values as the program executes. In particular, the action of assigning to a variable updates *state*, but is not an event, and so is not recorded in *slots*.

In order to give the generic semantics of the language, we need, in addition to $\mathcal{H}$, to have definitions supplied of operations on such histories, that satisfy key properties. For example, we need to know what an empty history looks like, and how to concatenate histories, so that concatenation is associative, with the empty history as the identity element. Other operations to be defined include a history prefix relation, history subtraction, event hiding in histories, and event synchronisation between histories running in parallel — all satisfying a key set of laws — see [BSW07] for details.

Given the definition of $\mathcal{H}$, and the associated functions and relations, we need to lift many of these to work with slots and slot-sequences (see Fig. 4). Relation $EqvTrace(tr, slots)$, asserts that tr, an event sequence, is compatible with the history in *slots*, ignoring time and refusals. Function *Refs* extracts refusals from

$$
\begin{aligned}
\mathit{EqvTrace} &: E^* \leftrightarrow \mathcal{S}^+ \\
\mathit{Refs} &: \mathcal{S}^+ \rightarrow (\mathbb{P}\, E)^+ \\
\preccurlyeq, \cong &: \mathcal{S}^+ \leftrightarrow \mathcal{S}^+ \\
\sharp, \backslash\backslash &: \mathcal{S}^+ \times \mathcal{S}^+ \nrightarrow \mathcal{S}^+ \\
\mathit{SSync} &: \mathbb{P}\, E \rightarrow \mathcal{S}^+ \times \mathcal{S}^+ \rightarrow \mathcal{S}^+ \\
\mathit{SHide} &: \mathit{SLOT} \times \mathbb{P}\, E \rightarrow \mathcal{S}
\end{aligned}
$$

Fig. 4. Slot-Sequence Functions/Relations

slot-sequences. The relations $\preccurlyeq$ and $\cong$ denote prefixing and equivalence of slot-sequences respectively — in this case equivalence is almost equality, except that the refusals in the last slot are ignored. Operations $\sharp$ and $\backslash\backslash$ denote slot concatenation and subtraction respectively — analogously to sequences, $\backslash\backslash$ is only defined if its second argument is a $\preccurlyeq$-prefix of its first. The key point to note here is that in the result of $s_1 \sharp s_2$, the last slot of s_1 is merged with the first slot of s_2. Function $\mathit{SSync}(c)(s_1, s_2)$ shows the effect of forcing the histories of s_1 and s_2 to synchronise on the events in set c, while $\mathit{SHide}(s, H)$ gives a slot were events in H are hidden (removed).

4 Healthiness Conditions

Healthiness conditions are characterised by idempotent predicate transformers, with a healthy predicate being a fixed point of such a transformer. The healthiness conditions we introduce here for slotted-*Circus* parallel some of those in [HH98, Chp. 8] for general reactive systems, namely **R1**, **R2**, **R3**, **CSP1** and **CSP2**. Here we shall only consider **R3**, **CSP1,2** in detail as they are explicitly invoked. **R1** and **R2** deal with the infeasibility of time travel and (direct) memory of past events, and are well covered elsewhere, and satisfied by all definitions we present in any case. The reactive conditions are aggregated as **R**, defined as the composition of **R1**–**3**.

The healthiness condition **R3** is one associated with all "reactive" systems in the UTP, covering process-algebras like ACP, CSP, and CCS.

$$
\begin{aligned}
\mathbf{R3}(P) &\mathrel{\widehat{=}} \mathit{I\!I} \lhd \mathit{wait} \rhd P \\
\mathit{I\!I} &\mathrel{\widehat{=}} \mathit{DIV} \lor \mathit{ok}' \land \mathit{wait}' = \mathit{wait} \land \mathit{slots}' = \mathit{slots} \\
\mathit{DIV} &\mathrel{\widehat{=}} \neg\ \mathit{ok} \land \mathit{slots} \preccurlyeq \mathit{slots}'
\end{aligned}
$$

R3 deals with the situation when a process has not actually started to run, because a prior process has yet to terminate, characterised by $\mathit{wait} = \textsc{True}$. In this case the action of a yet-to-be started process should simply be to do nothing, an action we call "reactive-skip" ($\mathit{I\!I}$). Reactive skip has two behavioural modes: if started in an unstable state (i.e the prior computation is diverging), then all it guarantees is that the slots may get extended somehow; otherwise it stays stable, and leaves most other observations unchanged.

Conditions **CSP1** and **CSP2** In [HH98, Chp. 8] there are five of these presented, but for our purposes it suffices to consider only the first two.

A process is **CSP1** healthy if *all* it asserts, when started in an unstable state (due to some serious earlier failure), is that the event history may be extended:

$$\mathbf{CSP1}(P) \mathrel{\widehat{=}} P \vee \neg\ ok \wedge slots \preccurlyeq slots'$$

Healthiness condition **R1** simply states that we can never undo past events, but **CSP1** deals with behaviour in a particular starting condition — it says that if ok is false, then the only thing we can assert is that events may happen in accordance with **R1**.

A process predicate is **CSP2** healthy if it does not mandate instability, so if true with $ok' = False$, it is also true with $ok' = True$, all other observation variables being unchanged.

$$\mathbf{CSP2}(P) \mathrel{\widehat{=}} P;\ (ok \Rightarrow ok') \wedge wait' = wait \wedge slots' = slots \wedge state' = state \tag{1}$$

The effect of post-composing P with $(ok \Rightarrow ok') \wedge \ldots$ is remove any assertion of $\neg\ ok'$, so for example calculation shows that $\mathbf{CSP2}(P \wedge \neg\ ok') = P$ whereas by contrast $\mathbf{CSP(}P \wedge ok') = P \wedge ok'$.

5 Slotted Semantics

The language constructs of sequential composition, internal and conditional choice, iteration all have the same semantics as in standard UTP:

$$\begin{aligned} P;\ Q &\mathrel{\widehat{=}} \exists\, obs_m \bullet P[obs_m/obs'] \wedge Q[obs_m/obs] \\ P \sqcap Q &\mathrel{\widehat{=}} P \vee Q \\ P \lhd c \rhd Q &\mathrel{\widehat{=}} c \wedge P \vee \neg\ c \wedge Q \\ c * P &\mathrel{\widehat{=}} \mu\, L \bullet (P;\ L) \lhd c \rhd Skip \end{aligned}$$

Recursion $(\mu\, X \bullet F(X))$ is defined as the least fixed-point of F w.r.t to the refinement ordering (reverse implication), and obs is shorthand for all the observational variables.

5.1 Semantic Building Blocks

We define the semantics of *slotted-Circus* in terms of a number of basic building-blocks, largely to do with events and communication, that we now describe. This building blocks are all **R1**-,**R2**-healthy, but in general will not satisfy **R3** or the CSP healthiness conditions in themselves— they are intended to be used in constructions that do.

First we provide a predicate $NOEVTS$ that describes a situation that allows time to pass ($\#slots' > \#slots$) but disallows the occurrence of any events:

$$NOEVTS \mathrel{\widehat{=}} EqvTrace(\langle\rangle, slots' \nwarrow slots)$$

It asserts that if we take the difference between before- and after-slots, then this will only be equivalent to the empty list $\langle\rangle$, which requires that every slot must contain an empty history component. A CTA example of this might be (r_i are arbitrary refusals):

$$slots' \backslash\backslash slots = \langle(\langle\rangle, r_1), (\langle\rangle, r_2), (\langle\rangle, r_3), (\langle\rangle, r_4)\rangle$$

Another very useful predicate asserts that a given set of events (E) have occurred, but that the clock has not yet ticked:

$$EVTSNOW(E) \mathrel{\widehat=} \exists\, tt \bullet elems(tt) = E \land EqvTrace(tt, slots' \backslash\backslash slots) \land \#slots = \#slots'$$

We are describing a situation where events occur in the first, and to date only time slot. We can find a trace tt equivalent to the observed slots, whose elements are the events in E, and where the before- and after-slots are of the same length, signifying that no clock tick has occurred. In CTA, this might be (r an arbitrary refusals, $E = \{a, b\}$):

$$slots' \backslash\backslash slots = \langle(\langle a, b, a\rangle, r)\rangle$$

In some situations, we want to describe events that occur immediately (in the first slot), as described by the predicate *IMMEVTS*:

$$IMMEVTS \mathrel{\widehat=} \exists\, E \bullet E \neq \emptyset \land EVTSNOW(E)\ ;\ \ slots \preccurlyeq slots'$$

We require the existence of a non-empty sets of events that occur "now" (i.e. in the first time-slot), followed by an arbitrary extension of slots.

5.2 Semantics of Basic Actions

We now give the semantics of the basic actions, construct by construct.

$$Chaos \mathrel{\widehat=} \mathbf{R}(\mathbf{true})$$

The worst possible action in *slotted-Circus* is *Chaos*. It is the most unpredictable healthy process, and bottom of the refinement lattice.

$$Stop \mathrel{\widehat=} \mathbf{CSP1}(\mathbf{R3}(ok' \land wait' \land NOEVTS))$$

Action *Stop* has deadlocked — is stable, never terminates and never performs any event.

$$Skip \mathrel{\widehat=} \mathbf{R3}(\mathbf{CSP1}(state = state' \land \neg wait' \land ok' \land slots \cong slots'))$$

Action *Skip* terminates immediately in a stable state, without performing any events. In keeping with the CSP definition, *Skip* ignores the refusals of any

preceding process, hence the use of slot-equivalence rather than slot-equality here.

$$Wait\ t \mathrel{\widehat=} \mathbf{CSP1}(\mathbf{R3}(ok' \land del(t) \land NOEVTS))$$
$$del(t) \mathrel{\widehat=} (\#slots' - \#slots < t) \lhd wait' \rhd (\#slots' - \#slots = t \land state' = state)$$

The action that introduces explicit timed behavior is $Wait\ t$. It never performs any events and has only two possible behaviors. The first one is to wait for t clock ticks, the second to terminate when the right time is reached.

$$x := e \mathrel{\widehat=} \mathbf{CSP1}\left(\mathbf{R3}\left(\begin{array}{l} ok' \land \neg wait' \land slots \cong slots' \\ \land\ state' = state \oplus \{x \mapsto val(e, state)\} \end{array}\right)\right)$$

Assignment is performed immediately, and for that reason is very similar to *Skip* — stable termination with no events or passing time observed. Valuation function *val* evaluates an expression given an environment. A key point to keep in mind here is that state-changes are recorded in the *state* variable and are not regarded as "events". The *state* here is globally visible — there are provisions in UTP and *Circus* for delimiting variable visibility but these are beyond the scope of this paper.

$$comm \rightarrow A \mathrel{\widehat=} (comm \rightarrow Skip);\ A$$

Unlike in CSP/CCS, an input communication binds an input value to a program variable, rather than the free occurrences of that name in the following process, so, for example, the input communication $c?x \rightarrow Skip$ ends by assigning the communicated value to the variable x. This allows us to treat the action $comm \rightarrow Skip$ as a basic building block and define more general prefixes in terms of it. We distinguish two basic behaviors of the prefix action: waiting for communication and performing it.

$$\begin{aligned} WTC(c) &\mathrel{\widehat=} POSS(c) \land NOEVTS \\ POSS(c) &\mathrel{\widehat=} c \notin \bigcup Refs(slots' \backslash\!\backslash slots) \\ TRMC(c) &\mathrel{\widehat=} EVTSNOW\{c\} \end{aligned}$$

While waiting for communication (WTC) we allow time to pass but we perform no events. We also inform the environment that we are ready to perform the specified event, by not refusing it (here *Refs* returns the refusals in each slot as a list). When we finally perform an event and terminate ($TRMC$) we have to make sure that the event is noted in the trace model. We also have to ensure that the specified event was not refused during the time-slots before the event occurred. For that reason we define the behavior of performing an event as $WTC(c);\ TRMC(c)$. We assemble all of this to get the following definition of prefix, noting in passing a key point that program variable state information is only propagated once the prefix action has terminated.

$$c \rightarrow Skip \mathrel{\widehat=} \mathbf{CSP1}\left(ok' \land \mathbf{R3}\left(WTC(c) \lhd wait' \rhd \left(\begin{array}{l} state' = state \land \\ WTC(c);\ TRMC(c) \end{array}\right)\right)\right)$$

The prefix action is also used to define channel-based communication. As per the usual CSP convention, sending a value is defined as performing an event - *channelName.value*, whilst receiving is defined as an external choice over all possible values allowable on a channel followed by assignment of the received value to the target variable. If we assume that channel c carries values of type $T = \{k_1, k_2, \ldots\}$, then

$$c!e \rightarrow Skip \mathrel{\widehat{=}} c.e \rightarrow Skip$$
$$c?x \rightarrow Skip \mathrel{\widehat{=}} \Box_{k:T} \bullet (c.k \rightarrow Skip;\ x := k)$$

Here $\Box_{k:T} \bullet P(x)$ is shorthand for $P(k_1) \Box P(k_2) \Box \ldots$.

5.3 Semantics of Composite Actions

External choice $(A \Box B)$ allows external events to determine which action runs, so for example if we have $(a \rightarrow A) \Box (b \rightarrow B)$, then, if the environment performs a, we see that event occur, followed by an execution of action A. Unfortunately the very simple definition[2] of external choice proposed in [HH98], no longer suffices, as we may have to wait for several clock-ticks before an external event arises that resolves the choice.

$$A \Box B \mathrel{\widehat{=}} \mathbf{CSP2}(\ Stop \wedge A \wedge B\ \vee\ Choice(A,B)\ \vee\ Choice(B,A)\ )$$
$$Choice(C,R) \mathrel{\widehat{=}} C \wedge \left(R \wedge NOEVTS;\ \left(\begin{array}{l} IMMEVTS \vee \\ slots \cong slots' \wedge (\neg wait' \vee \neg ok') \end{array} \right) \right)$$

Predicate $Choice(C,R)$ describes the circumstances where action C has been chosen, whilst R has been refused, which occurs in situation where R has performed no events. We capture these cases as follows: conjoin R with $NOEVTS$, and follow it sequentially with some "end"-condition E. All of this is conjoined with C to give

$$C \wedge (R \wedge NOEVTS;\ E)$$

i.e an execution of C consistent with R having done no events, and then ending in the situation described by E.

Now we can characterise three possible cases were C either: (i) performs an event after a delay: $E = IMMEVTS$; (ii) terminates without performing any events: $E = slots \cong slots' \wedge \neg wait'$ or (iii) diverges but performs no event: $E = slots \cong slots' \wedge \neg ok'$.

The parallel composition $A \,[\![\, s_A \mid \{\!|\ cs\ |\!\} \mid s_B \,]\!]\, B$ runs A and B in lock-step parallel (clock ticks at same time for both), with both actions required to synchronise on any channels mentioned in cs. Both actions run on local copies of the variables and are only allowed to modify those variables in their disjoint permission sets (s_A for A, s_B for B). The construct terminates when both actions

[2] $A \Box B \mathrel{\widehat{=}} A \wedge B \lhd Stop \rhd A \vee B$

have terminated — if one ends early then its behaviour is padded out with empty slots. If s_A and s_B overlap, or A (B) assigns or inputs into variables not in s_A (s_B), then the construct is ill-formed. At present, we do not consider shared-write access to variables as constituting a healthy or well-formed process. The reason for this restriction is that reasoning about parallel processes with global shared variables is a complex business [WH02]. There is of course scope for investigating more liberal forms of parallel composition, but that is left for future work.

The definition of parallel composition (for well-formed compositions) is large but conceptually straightforward:

$$\begin{aligned} A \,[\![\, s_A \mid \{\!|\, cs \,|\!\} \mid s_B \,]\!]\, B \mathrel{\widehat=} \exists\, obs_A, obs_B \bullet\ & A[obs_A/obs'] \land B[obs_B/obs'] \land \\ & ok' = ok_A \land ok_B \land \\ & wait' = (wait_A \lor wait_B) \land \\ & ValidMerge(cs)(slots, slots', slots_A, slots_B) \land \\ & (wait_A \Rightarrow \sharp slots_A \geq \sharp slots_B) \land \\ & (wait_B \Rightarrow \sharp slots_A \leq \sharp slots_B) \land \\ & (\neg wait' \Rightarrow state' = (state_A - s_B) \oplus (state_B - s_A)) \end{aligned}$$

Both actions are running on local copies of observation variables $A[obs_A/obs'] \land B[obs_B/obs']$ and the outcome is determined as an appropriate merge of these: The composition is stable if both A and B are, and is waiting if either action is. The resulting slots are a valid merge of compatible slot-sequences from each action. If an action is still waiting for events then it has seen at least as many clock ticks as the other process (which may have terminated). When the whole construct has terminated, the final $state'$ value is determined by merging the changes from each side.

$$ValidMerge \;:\; \mathbb{P}\, E \to ((\mathcal{S}\ E)^+)^4 \to \mathbb{B}$$
$$ValidMerge(cs)(s, s', s0, s_1) \mathrel{\widehat=} (s' \backslash\backslash s) \in TSync(cs)(s0 \backslash\backslash s), (s_1 \backslash\backslash s))$$

Merging the traces of parallel actions is captured by a predicate (*ValidMerge*) that asserts that the final slots execution ($slots' \backslash\backslash slots$) is a member of all the valid ways in which the two actions slots can be merged, taking the synchronisation sets cs into account (*TSync*). The *TSync* function returns all the possible fusings of two slot-sequences, slot-by-slot, with individual slots merged using *SSync*, the history-specific synchronisation parameter (see Fig 4). If one slot sequence is shorter than the other, then the shortest is padded out with null slots.

Our semantics, and that of CTA, differs here from that of timed-CSP [Sch00]. There *Skip* need not terminate immediately, but can delay, so facilitating the following law:

$$(a \to Skip \parallel Skip) = a \to Skip$$

The same law holds for *slotted-Circus*, even though *Skip* terminates immediately, because the singleton slots-sequence for the righthand *Skip* is padded out by the definition of parallel, to match that of the lefthand action as it waits for, and

$$
\begin{aligned}
Wait\ n \Box Wait\ n + m &= Wait\ n \\
(Skip \Box (Wait\ n;\ P)) &= Skip, \quad n > 0 \\
(c \rightarrow P) \Box (Wait\ n;\ (c \rightarrow P)) &= (c \rightarrow P) \\
Stop \Box A &= A \\
(c \rightarrow Skip) \setminus \{c\} &= Skip
\end{aligned}
$$

Fig. 5. (Some) Laws of slotted-*Circus*

eventually performs the event a.

$$
A \setminus H \mathrel{\widehat{=}} \mathbf{R3} \left(\begin{array}{l} \exists\, s' \bullet A[s'/slots'] \wedge \\ \quad slots' \lessdot slots = map(SHide(H))(s' \lessdot slots) \\ \quad \wedge\ H \subseteq \bigcap Refs(s' \lessdot slots) \end{array} \right) ;\ Skip
$$

The hiding operator $A \setminus H$ denotes an execution of action A, but with any events in event-set H hidden. The last assertion above about H and $Refs(\ldots)$ is implied by the definition of *SHide*, but is useful for proofs to have stated explicitly here. It has the effect of forcing a key properting of hiding, namely that of *maximal progress*, i.e. hidden events occur as soon as they are enabled. Without this semantic feature the following undesirable law would hold:

$$(a \rightarrow Skip) \setminus \{a\} = Wait\,0 \sqcap Wait\,1 \sqcap ... \sqcap Wait\ n \sqcap ...$$

This law is undesirable because it makes the performance of a single hidden event followed by termination equal to a wait for an arbitrary number of clock cycles — effectively a weak form of livelock. By forcing hidden events to be refused during every slot, we prevent them from waiting for a clock-tick, because the definition of prefix action requires events not to be refused when waiting. This results in the desired law, namely

$$(a \rightarrow Skip) \setminus \{a\} = Skip$$

At the end we add *Skip* to unconstrain the refusals set of the last slot.

6 Laws

The language constructs displayed here obey a wide range of laws, many of which have been described elsewhere [HH98, WC01, SH02, She06] for those constructs that slotted-*Circus* shares with other related languages like CSP or *Circus*(e.g. non-deterministic choice, sequential composition, conditional, guards, *STOP*, *SKIP*). Here we simply indicate (Fig.5) some of the laws that are peculiar to slotted-*Circus*, or whose proof was a challenge. The first law is a consequent of the fact that external choice treats termination as an "event" that can resolve an external choice. The proof of the latter two laws forced a lot of the design of the details of the semantic model described here. The definition of external choice and the its properties lead to the discovery of the state visibility issue

addressed in [GBW09]. The last law vindicates the semantic choice (used here, in CTA, and Timed-CSP) that entangles refusals up with the individual slots, rather than keeping them seperate from the event history, as in the *Failures* model of CSP [Ros97].

The proof of $(c \rightarrow \mathit{Skip}) \setminus \{c\} = \mathit{Skip}$ was long and difficult, based on a large range of properties from the very top level (healthiness conditions and circus specific actions), to a very low level, that of a single slot. The whole proof is roughly sixteen pages and for that reason we leave it to a technical report [BG09]. What makes this law special is that interaction between hiding and communication is the only place where refusals influence the behavior of the action and is actually responsible for a set of accepted traces. This can been seen by considering the following lemma which forms the core of the proof:

$$\begin{aligned}&(\mathit{WTC}(c);\ \mathit{TRMC}(c)) \wedge c \in \bigcap \mathit{Refs}(\mathit{slots}' \nwarrow \mathit{slots})\\ &= \mathit{TRMC}(c) \wedge c \in \bigcap \mathit{Refs}(\mathit{slots}' \nwarrow \mathit{slots})\end{aligned}$$

Predicate $\mathit{WTC}(c);\ \mathit{TRMC}(c)$ is a part of the prefix definition which describes a situation where the action waited (for zero or more clock ticks) to perform c, and then did so. During the waiting period, c was not being refused. By contrast, the predicate $c \in \bigcap \mathit{Refs}(\mathit{slots}' \nwarrow \mathit{slots})$ comes from the definition of hiding and requires that c be refused during any slots that have occurred. The only observations that satisfy both these requirements are ones where no clock ticks occur and communication is immediate, i.e $\mathit{TRMC}(C)$.

7 Related Work

In addition to the work done on state-rich reactive processes in UTP [OCW09] there has been attention paid to merging state and concurrency by others. The implementors of occam [SGS95] had to deal with the integration of state with its concurrency aspects, those being derived from CSP. An early integration of state and concurrency was the work on joining Object-Z and CSP [SD01], which was then followed up with real-time extensions [Smi02]. However these languages are very much at the specification level, with no explicit notion of assignment or global shared variables, as Object-Z schemas are interpreted as message-passing objects, so the concerns of this paper do not arise. The work on unifying CSP and B [But00] looks at linking the process of CSP with the actions of B. However while it converts CSP-like descriptions of behaviour into B state-machines, it has concept, at the CSP level of assignment to variables. Taking the denotational semantics of CSP and merging it with the algebraic semantics of CASL has resulting in a "data-rich' process algebra called CSP-CASL [Rog06]. Here the richness of CASL datatypes is made available for use as the types of values transmitted over communications channels. However there is no notion of state update through assignment in the theory.

A UTP semantics for Timed Communicating Object-Z (TCOZ [MD00]) is given in [QDC03]. The theory presented there has a communication component

which is a variant of He and Sherif's CTA [SH02], with a richer notion of event that differentiates between interprocess communication, and the interaction of the environment with sensors and actuators. Like the CTA semantics, it embeds **R3** into the definition of sequential composition, and defines communication to only assert the state is unchanged when communications has completed. Again "active objects" in TCOZ have their variable-state encapsulated. However TCOZ has an asynchronous interface mechanism of sensors and actuators, with the actuators linking a local variable to a global one. This mechanism can be used for internal communication as well as with the external environment.

8 Future Work

In [BSW07] we described a number of different ways to instantiate event histories within a time-slot, including:

- $\mathcal{CTA}$: histories are just events sequences (essentially the CTA theory [She06]).
- $\mathcal{MSA}$: histories are multisets or bags, so ordering within a slot is irrelevant
- $\mathcal{SCSP}$: histories are simple event sets — however these fails to satisfy the required laws on which the theory depends.

An important aspect that has yet to be covered is what distinguishes the various instantiations from one another, i.e. how do the laws of $\mathcal{CTA}$ differ from those of $\mathcal{MSA}$, for instance. We know for example that the following is not a law of $\mathcal{MSA}$, but does apply in $\mathcal{CTA}$:

$$(a \rightarrow b \rightarrow P) \parallel (b \rightarrow a \rightarrow P) = Stop$$

In $\mathcal{MSA}$ the deadlock can be avoided if both a and b occur in the same time-slot.

Another key concept, which has guided the precise form of the definition of external choice, is that of modelling priority among choices, which makes sense in a slotted-theory because we have a deadline (next clock-tick) against which any priority resolution scheme can operate. We plan to explore schemes to give semantic support to prioritised choice by appropriate modifications to the external choice definition. Interestingly, early indications are that a notion of priority will work in the $\mathcal{MSA}$ instantiation, but not the $\mathcal{CTA}$ incarnation !

Also worthy of exploration are the details of the behaviour of the Galois links [HH98, Chp 4] between different instances of slotted-*Circus*, and between those and standard *Circus*. These details will provide a framework for a comprehensive refinement calculus linking all these reactive theories together. The goal is a scheme whereby *Circus* is a specification language and *slotted-Circus* is a refinement stage, on the way to a hardware implementation.

9 Conclusions

A denotational semantics for *slotted-Circus* has been presented, backed up by a techreport giving fuller details[BG09]. The general layers of the theory have been

shown along with higher level building blocks used for defining the semantics. The key result presented here is a comprehensive semantics of the entire language that addresses various semantics issues that have been uncovered whilst laying foundations for future extensions, particularly towards prioritized choice.

A disadvantage of UTP is that some of the key proofs can be quite long and involved, as seen in the discussion regarding the hiding law. However, UTP also brings certain key advantages, which for our purposes outweigh this disadvantage:

- We are involved in a program of semantics unification — in this case bringing together *Circus*(itself a Z/CSP fusion), with related timed languages that combine both state and concurrency with message passing (CTA, Handel-C).
- Whilst standalone semantic models for each of the above are simpler, connecting them together formally is not.
- UTP is a common semantics foundation framework that supports both the merging of theories and the formal linking of them: given a predicate linking the observations of two different theories, the derivation of a galois connection putting them together in a refinement relationship is almost automatic [HH98, pp40–41]

The pain of developing formal models of languages, already well understood and formalised by other means, is, in our opinion, rewarded by the ease with which their formal interrelationships can then be explored.

Finally, the need to explicitly identify healthiness conditions, rather than have them emerge implicitly from the structure of a tailored semantic domain, seems to us to provide key comparative insights into the nature of languages under study.

The large amount of hand-proving involved has thrown the need for tool-support into sharp relief. This is exacerbated by the additional complexity that arises once time is added to the theory.

Acknowledgements We would like to thank Jim Woodcock and his colleagues for many fruitful discussions on various aspects of this work.

References

[BG09] Andrew Butterfield and Paweł Gancarski. slotted-Circus: A generic UTP framework for discretely-timed *circus*. Technical Report TCD-CS-09-32, School of Computer Science & Statistic Trinity College Dublin, Trinity College, Dublin 2, Ireland, July 2009. https://www.cs.tcd.ie/publications/tech-reports/reports.09/TCD-CS-2009-32.pdf.

[BSW07] Andrew Butterfield, Adnan Sherif, and Jim Woodcock. Slotted-*circus*: A UTP-family of reactive theories. In Jim Davies and Jeremy Gibbons, editors, *IFM 2007*, volume 4591 of *LNCS*, pages 75–97. Springer, 2007.

[But00] M. J. Butler. csp2b: A practical approach to combining csp and b. *Formal Aspects of Computing*, 12:182–196, 2000.

[BW05] Andrew Butterfield and Jim Woodcock. `prialt` in Handel-C: an operational semantics. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):248–267, June 2005.

[Cel02] Celoxica Ltd. *Handel-C Language Reference Manual, v3.0*, 2002. URL: `www.celoxica.com`.

[GBW09] Paweł Gancarski, Andrew Butterfield, and Jim Woodcock. State visibility and communication in unifying theories of programming. In Wei-Ngan Chin and Shengchao Qin, editors, *3rd IEEE International Symposium on Theoretical Aspects of Software Engineering*, pages 47–54. IEEE Computer Society, 2009.

[HH98] C. A. R. Hoare and Jifeng He. *Unifying Theories of Programming*. Series in Computer Science. Prentice Hall, 1998. http://www.unifyingtheories.org.

[Hoa85a] C. A. R. Hoare. *Communicating Sequential Processes*. Intl. Series in Computer Science. Prentice Hall, 1985.

[Hoa85b] C. A. R. Hoare. Programs are predicates. In *Proc. of a discussion meeting of the Royal Society of London on Mathematical logic and programming languages*, pages 141–155. Prentice-Hall, Inc., 1985.

[MD00] Brendan P. Mahony and Jin Song Dong. Timed communicating object Z. *IEEE Trans. Software Eng*, 26(2):150–177, 2000.

[OCW09] Marcel Oliveira, Ana Cavalcanti, and Jim Woodcock. A UTP semantics for circus. *Formal Asp. Comput*, 21(1-2):3–32, 2009.

[QDC03] Shengchao Qin, Jin Song Dong, and Wei-Ngan Chin. A semantic foundation for TCOZ in unifying theories of programming. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003*, volume 2805 of *LNCS*, pages 321–340. Springer, 2003.

[Rog06] Markus Roggenbach. CSP-CASL - A new integration of process algebra and algebraic specification. *Theor. Comput. Sci*, 354(1):42–71, 2006.

[Ros97] A. W. Roscoe. *The Theory and Practice of Concurrency*. international series in computer science. Prentice Hall, 1997.

[Sch00] Steve Schneider. *Concurrent and Real-time Systems — The CSP Approach*. Wiley, 2000.

[SD01] Graeme Smith and John Derrick. Specification, refinement and verification of concurrent systems-an integration of object-Z and CSP. *Formal Methods in System Design*, 18(3):249–284, 2001.

[SGS95] SGS-THOMSON Microelectronics Limited. *occam 2.1 reference manual*, May 12 1995.

[SH02] Adnan Sherif and Jifeng He. Towards a time model for circus. In Chris George and Huaikou Miao, editors, *ICFEM*, volume 2495 of *Lecture Notes in Computer Science*, pages 613–624. Springer, 2002.

[She06] Adnan Sherif. *A Framework for Specification and Validation of Real Time Systems using Circus Action*. Ph.d. thesis, Universidade Federale de Pernambuco, Recife, Brazil, Jan 2006.

[Smi02] Graeme Smith. An integration of real-time object-Z and CSP for specifying concurrent real-time systems. *LNCS*, 2335:267–285, 2002.

[WC01] Jim Woodcock and Ana Cavalcanti. *Circus*: a concurrent refinement language. Technical report, University of Kent at Canterbury, October 2001.

[WH02] Jim Woodcock and Arthur P. Hughes. Unifying theories of parallel programming. In *ICFEM'02*, pages 24–37. Springer-Verlag, 2002.