Slotted-Circus
A UTP-Family of Reactive Theories*

Andrew Butterfield[1], Adnan Sherif[2], and Jim Woodcock[3]

[1] Trinity College Dublin, Andrew.Butterfield@cs.tcd.ie
[2] Universidade Federal de Pernambuco, ams@cin.ufpe.br
[3] University of York, Jim.Woodcock@cs.york.ac.uk

Abstract. We 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 [WC01a]. This "slotted-*Circus*" framework is parameterised 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.
Two key results of this work are: the need to be very careful in formulating the healthiness conditions, particularly **R2**; and the demonstration that synchronous theories like SCSP [Bar93] do not fit well with the way reactive systems are currently formulated in UTP and *Circus*.

1 Introduction

1.1 *Circus* and slotted-*Circus*

The formal notation *Circus* [WC01a] is a unification of Z and CSP, and has been given a UTP semantics [WC02]. 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. In CTA, we have 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. The key notion in this paper is that we can instantiate the bottom-level history in a variety of ways: as simple traces, or multisets of events, or as the more complex "micro-slot" structures used in the operational semantics of Handel-C [BW05].

This paper describes a generalisation of CTA called "slotted-*Circus*", which is a collection of theories parameterised by different ways to instantiate the bottom-level event history within a time-slot. The motivation behind this work is the desire to re-cast existing semantics for Handel-C into the UTP framework so that *Circus* can be used as a specification language.

The Handel-C denotational [BW02] and operational semantics use this time-slot model, but with varying degrees of complexity in the slot structure, depending on which language constructs we wish to support. The slotted-*Circus* framework reported here is intended to be a foundation for formulating the common parts of these models, making it easier to explore the key differences.

1.2 UTP: General Principles

Theories in UTP are expressed as predicates 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

* research reported in this paper was partially supported by Qinetic

$$
\begin{aligned}
P;\ Q &\mathrel{\widehat{=}} \exists\, obs_0 \bullet P[obs_0/obs'] \wedge Q[obs_0/obs] \\
P \lhd c \rhd Q &\mathrel{\widehat{=}} c \wedge P \vee \neg\, c \wedge Q \\
P \sqcap Q &\mathrel{\widehat{=}} P \vee Q \\
\bigsqcap_{i \in I} P_i &\mathrel{\widehat{=}} \exists\, i : I \bullet P_i \\
S \sqsubseteq P &\mathrel{\widehat{=}} [P \Rightarrow S]
\end{aligned}
$$

Fig. 1. Basic UTP Operators

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 is called a *(pre-)condition*. A given theory is characterised by its alphabet, and a series of *healthiness conditions* that constrain the valid assertions that predicates may make. In almost all cases there are some basic operators common to every theory (Figure 1). Sequential composition ($P;\ Q$) corresponds to relational composition, *i.e.*, the existence of an intermediate state (obs_0), such that P relates obs to obs_0, whilst Q relates obs_0 to obs'. The conditional $P \lhd c \rhd Q$ is generally used when c is a condition and asserts that P holds if c is true, otherwise it asserts Q. Nondeterminism between two predicates $P \sqcap Q$ is simply logical disjunction, which extends to an existential quantifier for a nondeterministic choice over an indexed set ($\bigsqcap_i P_i$). We capture the notion of refinement $\sqsubseteq$ as logical entailment between the implementation and specification predicates, quantified over all free variables.

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 other words, we view the language constructs as being "syntactic sugar" for their predicate semantics, rather than defining some form of semantic function mapping some abstract syntax type to some domain capturing its meaning.

1.3 Structure and Focus

The main technical emphasis of this paper is on the construction of the generic framework and the required healthiness conditions, with the semantics of the language constructs and a case study provided to give a feel for its utility. We first present the syntax §2, generic framework §3, healthiness conditions §4, and semantics §5. We then discuss instantiation §6 and describe a case-study §7, before mentioning related §8 and future §9 work, and concluding §10. Two appendices give supporting material.

2 Syntax

The syntax of Slotted-*Circus* is similar to that of *Circus*, and a subset is shown in Figure 2. 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 booleans and non-negative integers are included.

The basic actions *Skip*, *Stop*, *Chaos* are similar to the corresponding CSP behaviours [Hoa85a, Sch00], respectively denoting actions that do nothing and terminate, do nothing and wait forever, or act unpredictably forever. We also introduce (multiple) assignment (:=) and event (communication) prefixes $\mathsf{Comm} \rightarrow \mathsf{Action}$ as basic actions. The communication prefixes range over communicating a value on a channel (Name.Expr), sending a value on a channel (Name!Expr), or receiving

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

Fig. 2. Slotted-*Circus* Syntax

a value on a channel (Name?Name). The composite action operator $\Box$ denotes external choice, whilst parallel composition of actions ($[\![\mathsf{VS} \mid \mathsf{CS} \mid \mathsf{VS}]\!]$) is parameterised by three sets, the first and third denoting the variables the corresponding action may modify, while the middle one specifies the synchronisation channels. We require that parallel processes modify disjoint parts of the state. We also have hiding ($\setminus \mathsf{CS}$) and recursively defined actions ($\mu\, \mathsf{Name} \bullet F(\mathsf{Name})$).

The key construct related to time-slots, and hence not part of *Circus*, is $\mathit{Wait}\ t$ which denotes an action that simply waits for t time-slots to elapse, and then terminates.

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. These slots contain information about the events that occurred during that time slot ("history") as well as the events being refused at that point. A key feature of all these semantic models is that the progress of events during a time-slot is observable, rather than just the overall outcome for an entire slot. While the initial goal was to develop a synchronous variant of *Circus*, it rapidly became clear that it was worth investing time in a generic slot-based theory, which could then be specialised to cover synchronicity, CTA, and the various slot-models that could be used to characterise Handel-C and similar synchronous hardware languages at various levels of detail.

We begin our description of the generic slotted theory by noting that it is parametric in three inter-related aspects:

- A given set of events, E.
- A type constructor $\mathcal{H}$ that builds a slot's history-type from an event type.
- A collection of basic functions that work with $\mathcal{H}\ E$, which must satisfy certain laws.

Given $\mathcal{H}$, we then define the notion of a slot ($\mathcal{S}$) as being a pair: a history and a set of events denoting a refusal:

$$\mathcal{S}\ E \mathrel{\widehat=} (\mathcal{H}\ E) \times (\mathbb{P}\ E) \tag{1}$$

In a sense a slot is similar to the notion of a failure in CSP [Ros97], except that it covers only the events within a single time-slot (*i.e.*, between two successive clock ticks). Given a notion of time-slot, we then introduce the top-level notion of event history as being a non-empty sequence

$$
\begin{aligned}
Acc_{\mathcal{H}} &: \mathcal{H}\ E \to \mathbb{P}\ E \\
EqvTrc_{\mathcal{H}} &: E^* \leftrightarrow \mathcal{H}\ E \\
HNull_{\mathcal{H}} &: \mathcal{H}\ E \\
\preceq_{\mathcal{H}} &: \mathcal{H}\ E \leftrightarrow \mathcal{H}\ E \\
Hadd_{\mathcal{H}} &: \mathcal{H}\ E \times \mathcal{H}\ E \to \mathcal{H}\ E \\
Hsub_{\mathcal{H}} &: \mathcal{H}\ E \times \mathcal{H}\ E \nrightarrow \mathcal{H}\ E \\
HHide_{\mathcal{H}} &: \mathbb{P}\ E \to \mathcal{H}\ E \to \mathcal{H}\ E \\
HSync_{\mathcal{H}} &: \mathbb{P}\ E \to \mathcal{H}\ E \times \mathcal{H}\ E \to \mathbb{P}(\mathcal{H}\ E)
\end{aligned}
$$

Fig. 3. Generic Functions over $\mathcal{H}\ E$

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.

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.
$state : Var \nrightarrow Value$ —An environment giving the current values of slotted-*Circus* variables
$slots : (\mathcal{S}\ E)^+$: —A non-empty sequence of slots recording the 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.

In order to give the generic semantics of the language, we need to provide six functions and two relations over $\mathcal{H}\ E$, listed in Figure 3. Function *Acc* returns the set of events mentioned (*Acc*epted) in its history argument. The relation *EqvTrc* relates a history to all event sequences (traces) compatible with it. *HNull* is a constant denoting an empty history. Infix symbol $\preceq$ captures the notion of one history being a prefix, of pre-history of another, and is required to be a pre-order. The functions *Hsub* and *Hadd* capture the notions of history subtraction and addition (extension). In particular we note that *Hsub* is partial and is only defined when the second argument is a pre-history of the first. Function *HHide* acts to remove a set of events from a history. Finally the *HSync* function generates all the possible histories that can result from the synchronisation of two histories over a given event set.

In order to produce a coherent theory, the functions have to obey a number of laws, listed in Appendix A. Most of the properties concerned capture reasonable behaviours that one would expect of histories, *e.g.*, that history addition is associative, or that the null history acts as a unit. Most of these laws where determined by the needs of the general theory, in particular the definitions and proofs needed to establish the required healthiness conditions.

As an example, a variation of the CTA theory of [She06] can be captured by defining an event history ($\mathcal{H}_{\mathcal{CTA}}\ E$) to be a sequence of events, and instantiating most of the functions and relations as the corresponding ones for sequences.

$$\mathcal{H}_{\mathcal{CTA}}\ E \mathrel{\widehat{=}} E^* \tag{2}$$

3.1 Derived Types and Operators

Given the definition of $\mathcal{H}$, and the associated functions and relations, we need to use these to define the corresponding aspects for slots, and the slot-sequences that comprise our observational

$$
\begin{aligned}
\mathit{EqvTrace} &: E^* \leftrightarrow (\mathcal{S}\ E)^* \\
\mathit{Refs} &: (\mathcal{S}\ E)^+ \rightarrow (\mathbb{P}\ E)^+ \\
\mathit{EqvRef} &: (\mathcal{S}\ E)^+ \rightarrow \mathbb{P}\ E \\
\preccurlyeq &: (\mathcal{S}\ E)^+ \leftrightarrow (\mathcal{S}\ E)^+ \\
\approx &: \mathcal{S}\ E \leftrightarrow \mathcal{S}\ E \\
\cong &: (\mathcal{S}\ E)^+ \leftrightarrow (\mathcal{S}\ E)^+ \\
\mathit{Sadd}_{\mathcal{S}} &: \mathcal{S}\ E \times \mathcal{S}\ E \rightarrow \mathcal{S}\ E \\
\mathit{Ssub}_{\mathcal{S}} &: \mathcal{S}\ E \times \mathcal{S}\ E \nrightarrow \mathcal{S}\ E \\
\sharp &: ((\mathcal{S}\ E)^+ \times (\mathcal{S}\ E)^+) \rightarrow (\mathcal{S}\ E)^+ \\
\backslash\backslash &: ((\mathcal{S}\ E)^+ \times (\mathcal{S}\ E)^+) \nrightarrow (\mathcal{S}\ E)^+
\end{aligned}
$$

Fig. 4. Derived Functions and Relations

variables (see Figure 4). *EqvTrace*, defined in terms of *EqvTrc*, relates traces to slot-sequences with which they are compatible. The functions *Refs* and *EqvRef* extract refusals from slot-sequences, with the former returning a refusal-set list, whilst the latter singles out the last refusal set. A slot-sequence s is a slot-prefix of a slot-sequence t, written $s \preccurlyeq t$ if the front of s is a prefix of t and the history component of the last slot of s is a history-prefix of the corresponding component of the first slot of $t - s$. The relation $\preccurlyeq$ is a pre-order. Slot equivalence $\approx$ and Slot-sequence equivalence ($\cong$) are the symmetric closure of $\preceq$ and $\preccurlyeq$ respectively, giving equivalence relations. An important point to note here is that if $s \cong t$, then s and t are identical, except for the refusal values in the last slot in each.

The notions of adding (extending) and subtracting histories are lifted to the slot level, but here an issue immediately arises as to how the refusal components are handled. If we consider history addition, then $\mathit{Hadd}(h_1, h_2)$ is intended to capture the history resulting from the events of history h_1, followed by those of h_2. We now note that in most CSP-like theories, a failure consisting of a trace/history of events (h) coupled with a refusal set (r), is to be interpreted as stating that the process under consideration is refusing the events in r, *after* having performed the events in h. Given this interpretation, we are then required to specify slot addition and subtraction as follows:

$$
\begin{aligned}
\mathit{Sadd}((h_1, \_), (h_2, r_2)) &\mathrel{\widehat{=}} (\mathit{Hadd}(h_1, h_2), r_2) \\
\mathit{Ssub}((h_1, r_1), (h_2, \_)) &\mathrel{\widehat{=}} (\mathit{Hsub}(h_1, h_2), r_1)
\end{aligned}
$$

For history subtraction, the value $\mathit{Hsub}(h_1, h_2)$ is defined only if $h_2 \preceq h_1$, and denotes those events in h_1 that occurred after those in h_2. The significance of this interpretation is important, as will be made clear when we consider an attempt to model Synchronous CSP (SCSP) [Bar93] later in this paper. A consequence of this interpretation is that one of the healthiness conditions discussed in the next section (**R2**) becomes more complex.

Given slot addition and subtraction, these can then be lifted to act on slot-sequences, as $\sharp$ and $\backslash\backslash$ respectively. The latter is only defined if its second argument is a $\preccurlyeq$-prefix of its first. Slot-sequence addition concatenates its two arguments, merging the last slot of the first with the first slot of the second:

$$
\mathit{slots}_1 \sharp \mathit{slots}_2 \mathrel{\widehat{=}} \mathit{front}(\mathit{slots}_1) \frown \langle \mathit{Sadd}(\mathit{last}(\mathit{slots}_1), \mathit{head}(\mathit{slots}_2)) \rangle \frown \mathit{tail}(\mathit{slots}_2) \tag{3}
$$

Slot-sequence subtraction $s \setminus\!\!\setminus t$ is defined when $t \preccurlyeq s$, in which case both s and t can be written as

$$s = pfx \frown \langle slot_s \rangle \frown sfx$$
$$t = pfx \frown \langle slot_t \rangle$$

In this case, the subtraction becomes:

$$s \setminus\!\!\setminus t \mathrel{\widehat{=}} \langle Ssub(slot_s, slot_t) \rangle \frown sfx \tag{4}$$

4 Healthiness Conditions

Given that we are defining semantics as predicates over before- and after-observations, we need to ensure that what we write is feasible, in that we do not describe behaviour that is computationally or physically infeasible (*e.g.*, undoing past events). In UTP, the approach to handling feasibility is to define a number of so-called healthiness conditions that characterise the sort of predicates which make sense in the intended interpretation of the theory.

While the notion of healthiness-conditions is well-understood in the UTP community, we are still going to take time for the presentation that follows, as we highlight a prevalent use of overloading that can have unexpected effects in inexperienced hands.

Given a healthiness condition called **H** we introduce two functions, **mkH** and **isH**. In order to denote a healthiness condition, we require that the former is an idempotent monotonic predicate transformer, w.rt. to the standard ordering used in UTP, namely that $S \sqsubseteq P$ iff $[P \Rightarrow S]$. The role of **mkH** is to convert an un-healthy predicate into a healthy one, in some fashion, but also to leave already healthy predicates unchanged (hence the need for idempotency, so that a healthy predicate is a fixed-point of **mkH**).

$$\mathbf{mkH} \ : \ \mathit{Predicate} \to \mathit{Predicate}$$
$$\mathbf{mkH} = \mathbf{mkH} \circ \mathbf{mkH}$$

Function **isH** asserts a healthiness condition, *i.e.*, is a higher order predicate that tests a given predicate to see if it is healthy:

$$\mathbf{isH} \ : \ \mathit{Predicate} \to \mathbb{B}$$
$$\mathbf{isH}(P) \mathrel{\widehat{=}} P \equiv \mathbf{mkH}(P)$$

We can summarise by saying that a healthy predicate is a fixed-point of the corresponding healthiness predicate transformer. In most material on UTP, it is conventional to overload the notation **H** to refer to both **mkH** and **isH**, with the use usually being clear from context. In either case it is also conventional to refer in general to **H** as a healthiness condition, even in a context were it would actually be a predicate transformer. We shall adopt this convention in the sequel.

However a hazard can arise when alternative formulations of **H** are available; note that different functions may have the same set of fixed-points. We illustrate this later when discussing **R2**.

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** and **CSP1**.

4.1 Reactive Healthiness

We shall discuss **R1** and **R3** first, as these are fairly straightforward, while **R2** deserves some discussion, as its adaption for slotted-*Circus* was decidedly non-trivial.

R1 simply states that a slotted-*Circus* process cannot undo the past, or in other words, that the $slots'$ observation must be an extension of *slots*, whilst **R3** deals with the situation when a process has not actually started to run, because a prior process has yet to terminate, characterised by $wait = \text{TRUE}$. In this case the action of a yet-to-be started process should simply be to do nothing, an action we call "reactive-skip" ($I\!I$). 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 all other observations unchanged.

$$\begin{aligned}
&\mathbf{R1}(P) \mathrel{\widehat{=}} P \wedge slots \preccurlyeq slots' \\
&\mathbf{R3}(P) \mathrel{\widehat{=}} I\!I \lhd wait \rhd P \\
&\quad I\!I \mathrel{\widehat{=}} \neg\, ok \wedge slots \preccurlyeq slots' \vee ok' \wedge wait' = wait \wedge state' = state \wedge slots' = slots
\end{aligned}$$

The purpose of the *slots* observation variable in slotted-*Circus*, and its trace analogue (tr) in UTP reactive-process theory, is to facilitate the definition of operators such as sequential composition. What is not permitted however, is for a process to be able to base its actions on the history of past events as recorded by this variable—any such "memory" of the past must be captured by the *state* observation. Healthiness condition **R2** is concerned with ensuring that a process can only specify how the history is extended, without reference to what has already happened. In [HH98, Chp. 8] this is captured by stating that P is **R2**-healthy if it is invariant under an arbitrary shift in the prehistory, or in other words, a non-deterministic choice over all possible values that tr might take:

$$\begin{aligned}
\mathbf{R2}\text{–}\mathbf{UTP}(P) &\mathrel{\widehat{=}} \sqcap_s P[s, s \frown (tr' - tr)/tr, tr'] \\
&\equiv \exists\, s \bullet P[s, s \frown (tr' - tr)/tr, tr']
\end{aligned}$$

It would seem reasonable to expect the slotted-*Circus* version to simply replace tr by *slots* and use the slot-sequence analogues of sequence concatenation and subtraction. This would result in the following definition (here the **a** indicates "almost"):

$$\mathbf{R2a}(P) \mathrel{\widehat{=}} \exists\, ss \bullet P[ss, ss \mathbin{\sharp} (slots' \mathbin{\backslash\!\backslash} slots)/slots, slots'] \tag{5}$$

Whilst this looks plausible, there is in fact a problem with it, which only becomes apparent when we attempt to apply the definition later on in the semantics and then prove certain key desirable properties. Consider the predicate $slots' = slots$ which asserts that no events occur. This predicate should be **R2**-healthy, as it describes a process that chooses to do nothing, regardless of the value of *slots*. However calculation shows that

$$\mathbf{R2a}(slots' = slots) \equiv slots' \cong slots\,.$$

The equality gets weakened to the slot-sequence equivalence introduced earlier. An immediate consequence of this is that $I\!I$ is not healthy by this definition, as calculation shows that the slot-equality is weakened to slot-equivalence (underlined below).

$$\mathbf{R2a}(I\!I) \equiv \neg\, ok \wedge slots \preccurlyeq slots' \vee ok' \wedge wait' = wait \wedge state' = state \wedge \underline{slots' \cong slots}$$

Original work explored keeping **R2a** as is, and redefining $I\!I$ to be that version shown above. However this then weakened a number of key properties of $I\!I$, most notably to do with its role as an identity for sequential composition under appropriate circumstances.

The underlying problem with **R2a** has to do with the fact that in slotted-*Circus*, unlike UTP, we have refusals interleaved with events in *slots*, and slot-sequence operators that treat refusals, particularly the last, in a non-uniform way. The problem is that **R2a** weakens the predicate a

little too much, so we need to find a way to strengthen its result appropriately. The appropriate way to handle this issue has turned out to be to modify the definition of **R2** to require that we only quantify over *ss* values that happen to agree with *slots* on the very last refusal. This has no impact on predicates like $\preccurlyeq$ and $\cong$ which are not concerned with the last refusals, but provides just enough extra information to allow slot-sequence equality be considered as **R2**-healthy. The slightly strengthened version now reads:

$$\mathbf{R2}(P) \mathrel{\widehat{=}} \exists\, ss \bullet P[ss, ss \,\sharp\, (slots' \mathbin{\backslash\!\backslash} slots)/slots, slots'] \wedge Ref(last(slots)) = Ref(last(ss))$$

The proof that **R2** is idempotent is somewhat more involved than those for **R1** and **R3**. Calculations show that predicates $slots \preccurlyeq slots'$, $slots' \cong slots$, $slots' = slots$ (se Appendix B) and $\mathbb{I}$, are all **R2**-healthy. It also distributes through disjunction, which is very important.

It is worth pointing out that two versions of **R2** are presented in [HH98]. The second, which we shall call **R2'** is shown in an appendix:

$$\mathbf{R2'}(P) \mathrel{\widehat{=}} P[\langle\rangle, tr' - tr/tr, tr']$$

Both **R2** and **R2'** have the same set of fixed points, so can be used interchangeably as a test for healthiness. However, if used to make a predicate healthy, then **R2** is more forgiving than **R2'**:

$$\mathbf{R2}(tr = \langle a\rangle \wedge tr' = \langle a, b\rangle) \equiv (tr' - tr) = \langle b\rangle$$
$$\mathbf{R2'}(tr = \langle a\rangle \wedge tr' = \langle a, b\rangle) \equiv \mathbf{false}$$

This is an example of where overloading the notation **H** to stand for both **mkH** and **isH** can be misleading. We note that the version of **R2** used in [She06] is the CTA equivalent of **R2'**.

Reactive Healthiness A reactive slotted-*Circus* process is one that satisfies all three of the above healthiness conditions, so we define an overall condition **R** as their composition:

$$\mathbf{R} \mathrel{\widehat{=}} \mathbf{R3} \circ \mathbf{R2} \circ \mathbf{R1} \tag{6}$$

In fact all three conditions commute with each other, so we re-order the above composition to suit.

4.2 CSP Healthiness

In addition to the reactive-healthiness just introduced, shared by a range of concurrent theories including ACP and CSP, there are a number of aspects of healthiness specific to CSP-like theories. In [HH98, Chp. 8] there are five of these presented, but for our purposes it suffices to consider only the first one.

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:

$$\mathbf{CSP1}(P) \mathrel{\widehat{=}} P \vee \neg\, ok \wedge slots \preccurlyeq slots' \tag{7}$$

5 Slotted Semantics

We are now in a position to give the semantics of the slotted-Circus language which is presented for completeness in Figures 5 & 6.

We shall not give a detailed commentary to all the definitions shown but instead will focus on some key points.

$$
\begin{aligned}
Chaos &\mathrel{\widehat=} \mathbf{R}(\mathbf{true}) \\
Stop &\mathrel{\widehat=} \mathbf{CSP1}(\mathbf{R3}(ok' \wedge wait' \wedge EqvTrace(\langle\rangle, slots' \nwarrow slots))) \\
b \& A &\mathrel{\widehat=} A \lhd b \rhd Stop \\
Skip &\mathrel{\widehat=} \mathbf{R}(\exists\, ref \bullet ref = EqvRef(slots) \wedge \mathbb{I}) \\
Wait\ t &\mathrel{\widehat=} \mathbf{CSP1}(\mathbf{R}(ok' \wedge delay(t) \wedge EqvTrace(\langle\rangle, slots' \nwarrow slots))) \\
&\quad delay(t) = (\#slots' - \#slots < t) \lhd wait' \rhd (\#slots' - \#slots = t \wedge state' = state) \\
x := e &\mathrel{\widehat=} \mathbf{CSP1}\left(\mathbf{R}\left(\begin{array}{l} ok = ok' \wedge wait = wait' \wedge slots = slots' \\ \quad \wedge\ state' = state \oplus \{x \mapsto val(e, state)\} \end{array}\right)\right) \\
&\quad val : \mathsf{Expr} \times (\mathsf{Name} \to Value) \nrightarrow Value \\
c.e \to Skip &\mathrel{\widehat=} \mathbf{CSP1}\left(ok' \wedge \mathbf{R}\left(wait\_com(c) \vee complete\_com(c.e)\right)\right) \\
&\quad wait\_com(c) = wait' \wedge possible(c)(slots, slots') \wedge EqvTrace(\langle\rangle, slots' \nwarrow slots) \\
&\quad possible(c)(slots, slots') = c \notin \bigcup Refs(slots' - front(slots)) \\
&\quad term\_com(c.e) = \neg\ wait' \wedge \#slots = \#slots' \wedge EqvTrace(\langle c \rangle, slots' \nwarrow slots) \\
&\quad complete\_com(c.e) = term\_com_(c.e) \vee wait\_com(c);\ term\_com(c.e) \\
c!e \to Skip &\mathrel{\widehat=} c.e \to Skip \\
c?x \to Skip &\mathrel{\widehat=} \exists\, e \bullet \left(c.e \to Skip[state_0/state] \wedge state' = state_0 \oplus \{x \mapsto e\}\right) \\
comm \to A &\mathrel{\widehat=} (comm \to Skip);\ A \\
A \,\Box\, B &\mathrel{\widehat=} \mathbf{CSP2}(ExtChoice1(A, B) \vee ExtChoice2(A, B)) \\
&\quad ExtChoice1(A, B) \mathrel{\widehat=} A \wedge B \wedge Stop \\
&\quad ExtChoice2(A, B) \mathrel{\widehat=} (A \vee B) \wedge DifDetected(A, B) \\
&\quad DifDetected(A, B) \mathrel{\widehat=} \neg\ ok' \vee \left(\begin{array}{l} \left(\begin{array}{l} (ok \wedge \neg\ wait) \wedge \\ \left(\begin{array}{l} \left(\begin{array}{l} A \wedge B \wedge ok' \wedge \\ wait' \wedge slots = slots' \end{array}\right) \vee \\ Skip \end{array}\right) \end{array}\right) ; \\ \left(\begin{array}{l} (ok' \wedge \neg\ wait' \wedge slots' = slots) \vee \\ (ok' \wedge ImmEvts(slots, slots')) \end{array}\right) \end{array}\right) \\
&\quad ImmEvts(slots, slots') \mathrel{\widehat=} \neg\ EqvTrc(\langle\rangle, head(slots' \nwarrow slots))
\end{aligned}
$$

Fig. 5. Slotted-*Circus* Semantics (part I)

$$
\begin{array}{l}
A \,[\![\, s_A \mid \{\!|\ cs\ |\!\} \mid s_B \,]\!]\, B \\
\quad \widehat{=}\ \exists\, obs_A, obs_B \bullet \\
\qquad A[obs_A/obs'] \wedge B[obs_B/obs'] \wedge \\
\qquad \left(\begin{array}{l}
\textbf{if } \left(\begin{array}{l} s_A \lhd state_A \neq s_A \lhd state \vee \\ s_B \lhd state_B \neq s_B \lhd state \vee \\ s_A \cap s_B \neq \emptyset \end{array} \right) \\
\textbf{then } \neg\, ok' \wedge slots \preccurlyeq slots' \\
\textbf{else } \left(\begin{array}{l} ok' = ok_A \wedge ok_B \wedge \\ wait' = (wait_A \vee 1.wait_B) \wedge \\ state' = (s_B \lhd state_A) \oplus (s_A \lhd state_B) \wedge \\ ValidMerge(cs)(slots, slots', slots_A, slots_B) \end{array} \right)
\end{array} \right) \\
\qquad ValidMerge : \mathbb{P}\, E \to ((\mathcal{S}\ E)^+)^4 \to \mathbb{B} \\
\qquad ValidMerge(cs)(, s, s', s0, s_1) = dif(s', s) \in TSync(cs)(dif(s0, s), dif(s_1, s)) \\
\qquad TSync : \mathbb{P}\, E \to (\mathcal{S}\ E)^* \times (\mathcal{S}\ E)^* \to \mathbb{P}((\mathcal{S}\ E)^+) \\
\qquad TSync(cs)(s_1, s_2) = TSync(cs)(s_2, s_1) \\
\qquad TSync(cs)(\langle\rangle, \langle\rangle) = \{\} \\
\qquad TSync(cs)(\langle s \rangle, \langle\rangle) = \{\, \langle s' \rangle \mid s' \in SSync(cs)(s, SNull(Ref(s))) \,\} \\
\qquad TSync(cs) \left(\begin{array}{l} s_1 \fatsemi S_1, \\ s_2 \fatsemi S_2 \end{array} \right) = \left\{ \begin{array}{l} s' \fatsemi S' \\ \mid s' \in SSync(cs)(s_1, s_2) \wedge \\ \quad S' \in TSync(cs)(S_1, S_2) \end{array} \right\} \\
A \setminus hidn \ \widehat{=}\ \mathbf{R} \left(\begin{array}{l} \exists\, s \bullet A[s/slots'] \wedge \\ slots' \lll slots = map(SHide(hidn))(dif(s, slot)) \end{array} \right) ;\ Skip \\
\mu\, X \bullet F(X) \ \widehat{=}\ \sqcap \{X \mid F(X) \sqsubseteq X\}
\end{array}
$$

Fig. 6. Slotted-*Circus* Semantics (II)

The *STOP* action refuses all events, but does allow the clock to keep ticking. Assignment and channel-communication take less than a clock-cycle, so we can sequence arbitrarily many in a time-slot. This does raise the possibility of Zeno processes (infinite events within a time-slot), so some care will be required here (disallowing infinite histories). This is more power than that required for synchronous hardware, where we expect these actions to synchronise with the clock, but we can model that by postfixing a *Wait* 1 statement, as used in the case study shown later. An important point to note is the definition of channel input ($c?x \rightarrow P$), not only involves an event $c.e$ for some e, but also updates the state. This is exploited later to allow shared variables.

The definition of external choice is quite complex —see [She06, p69] for a discussion.

We define slotted-parallel in a direct fashion, similar to that used for *Circus*, avoiding the complexities of the UTP/CTA approaches, and also handling error cases in passing. An error occurs in $P \,[\![\, s_A \mid C \mid s_B \,]\!]\, Q$ if P (Q) modifies any variable in s_B (s_A).

5.1 Laws

The language constructs displayed here obey a wide range of laws, many of which have been described elsewhere [HH98, WC01b, SH02, She06] for those constructs that slotted-*Circus* shares with other languages (e.g. non-deterministic choice, sequential composition, conditional, guards, *STOP*, *SKIP*). Here we simply indicate some of the laws regarding *Wait* that peculiar to slotted-*Circus* (Figure 7).

$$
\begin{aligned}
\mathit{Wait}\ n;\ \mathit{Wait}\ m &= \mathit{Wait}\ (m+n) \\
\mathit{Wait}\ n \,\Box\, \mathit{Wait}\ n+m &= \mathit{Wait}\ n \\
(\mathit{Wait}\ n;\ P) \,\Box\, (\mathit{Wait}\ n;\ Q) &= \mathit{Wait}\ n;\ (P \,\Box\, Q) \\
(\mathit{Skip} \,\Box\, (\mathit{Wait}\ n;\ P)) &= \mathit{Skip}, \quad n > 0 \\
(a \rightarrow P) \,\Box\, (\mathit{Wait}\ n;\ (a \rightarrow P)) &= (a \rightarrow P)
\end{aligned}
$$

Fig. 7. Laws of slotted-*Circus* *Wait*.

5.2 Links

In [HH98, §1.6,pp40–1], a general Galois connection between an abstract theory with observational variable a and a concrete theory over observation c is:

$$[(\exists\, c \bullet D(c) \wedge \ell(c,a)) \Rightarrow S(a)] \ \textbf{iff}\ [D(c) \Rightarrow (\forall\, a \bullet \ell(c,a) \Rightarrow S(a))]$$

Here D and S are corresponding design (concrete) and specification (abstract) predicates respectively, while $\ell(c,a)$ is the linking predicate connecting observations at the two worlds. Of interest to us in the main are links between *Circus* (playing the role of the abstract theory with observations a) and various instantiations of slotted-*Circus* (concrete, with obsevations c). The difference between *Circus* and slotted-*Circus* is that the former has observations *tr* and *ref*, whilst the latter subsumes both into *slots*. However we can immediately exploit the method just presented by using the following relationship to define ℓ, which here relates the *Circus* observational variables to those of slotted-*Circus*:

$$\mathit{EqvTrace}(tr, \mathit{slots}) \wedge \mathit{ref} = \mathit{EqvRef}(\mathit{slots}) \tag{8}$$

So we get a Galois-link between *Circus* and any instantiation of slotted-*Circus* for free. Similarly, a given relationship between different $\mathcal{H}$ types allows us to generate Galois-links between different slotted-*Circus* instantiations.

6 Instantiating Slotted-*Circus*

We now look at the issue of giving one or more concrete instantiations to the slotted-*Circus* framework just described. Originally, this work was aimed at producing a synchronous version of *Circus*, in which all events in a time-slot were to be considered as simultaneous. One motivation for this was to support the Handel-C language, which maps programs to synchronous hardware in which all variable updates are synchronised with a global clock edge marking the end of a computation cycle [Cel02]. However, there were two main difficulties with this approach.

The first was that the formal semantics developed for Handel-C outside of the UTP framework [BW02, BW05] actually modelled activity within a time-slot as a series of decision-making events spread out in time, all culminating in a set of simultaneous variable updates at the end of the slot. This approach, adopted in both the operational and denotational semantics, gives a very natural and intuitive description of what is taking place during Handel-C execution.

The second difficulty is more fundamental in nature, and exposed a key assumption underlying the UTP reactive theories, and those for CSP in general. Early work looked at the Ph.D thesis of Janet Barnes [Bar93] which introduced a synchronous version of CSP (SCSP). The key observation was a sequence of slots, each comprising two event sets, one denoting the events occurring in that slot (Acceptances) and the other describing the events refused (Refusals). A healthiness condition required that the acceptances and refusals in any slot be disjoint. However, implicit in this disjointedness condition is the notion that both the acceptances and refusals are truly simultaneous. However, in the failures of CSP, and the corresponding *tr* and *ref* observations of UTP, the key interpretation involved is that the refusals describe what is being refused given that the event history has just taken place. As a specific example, consider the process $a \rightarrow b \rightarrow P$. A possible (failure) observation of this process is $(\langle a \rangle, \{a\})$, *i.e.*, we have observed the occurrence of the a event and the fact that the process is now refusing to perform an a.

Consider trying to instantiate a slot where the history is simply an event-set, as per SCSP:

$$
\begin{aligned}
A \in \mathcal{SCSP}\ E &\mathrel{\widehat=} \mathbb{P}\, E \\
HNull_{\mathcal{SCSP}} &\mathrel{\widehat=} \emptyset \\
Hadd_{\mathcal{SCSP}}(A_1, A_2) &\mathrel{\widehat=} A_1 \cup A_2 \\
Hsub_{\mathcal{SCSP}}(A_1, A_2) &\mathrel{\widehat=} A_1 \setminus A_2 \\
&\ldots
\end{aligned}
$$

We find that we cannot guarantee law [Sadd:unit] (Appendix A), even if the SCSP invariant is not required. This property is required to demonstrate that $slots \cong slots'$ is **R2**-healthy. The underlying problem is that the definition of **R2** relies on being able to deduce that *slots* is empty if subtracting *slots* from $slots'$ leaves $slots'$ unchanged. However at the history-as-set level, we cannot deduce $H = \emptyset$, given that $H' \setminus H = H'$.

6.1 Multiset History Instantiation

We can define an instantiation where the event history is a multiset or bag of events ($\mathcal{H}_{\mathcal{MSA}}$), so event ordering is unimportant, but multiple event occurrences in a slot do matter (Figure 8). The bag notation used here is that of Z [Spi87]. The events accepted are simply the bag domain. A

$$
\begin{aligned}
\mathcal{H}_{\mathcal{MSA}}\ E &\mathrel{\widehat=} E \pfun \mathbb{N}_1 \\
Acc(bag) &\mathrel{\widehat=} dom(bag) \\
EqvTrc(tr, bag) &\mathrel{\widehat=} items(tr) = bag \\
HNull &\mathrel{\widehat=} [\![]\!] \\
bag_1 \preceq bag_2 &\mathrel{\widehat=} bag_1 \sqsubseteq bag_2 \\
Hadd(bag_1, bag_2) &\mathrel{\widehat=} bag_1 \oplus bag_2 \\
Hsub(bag_1, bag_2) &\mathrel{\widehat=} bag_1 \ominus bag_2 \\
HSync(cs)(bag_1, bag_2) &\mathrel{\widehat=} \{(cs \lhd (bag_1 \oplus bag_2)) \oplus (cs \lhd (bag_1 \cap bag_2))\} \\
\textbf{where} \quad & \cap \text{ is bag interesection} \\
HHide(hdn)bag &\mathrel{\widehat=} hdn \lhd bag
\end{aligned}
$$

Fig. 8. Multiset Action Instantiation ($\mathcal{MSA}$)

trace corresponds to a bag if it contains the same number of events as that bag. A null history is simply an empty bag. A bag is a prefix if smaller than another bag. History addition and subtract are the bag equivalents. History synchronisation merges the parts of the two bags disjoint from the synchronisation set, with the intersection of all three. Hiding is modelled by bag restriction.

The proofs that the above instantiation satisfy the properties in Appendix A are all straightforward. The proof of law [ET:pfx] for $\mathcal{MSA}$ is shown in Appendix B.

7 Example *Circus* process

We illustrate slotted *Circus* using an example originally due to Hoare [Hoa85a]. The problem is to compute the weighted sums of consecutive pairs of inputs. Suppose that the input stream contains the following values: $\mathtt{x}_0, \mathtt{x}_1, \mathtt{x}_2, \mathtt{x}_3, \mathtt{x}_4, \ldots$; then the output stream will be

$$(a * \mathtt{x}_0 + b * \mathtt{x}_1), (a * \mathtt{x}_1 + b * \mathtt{x}_2), (a * \mathtt{x}_2 + b * \mathtt{x}_3), (a * \mathtt{x}_3 + b * \mathtt{x}_4), \cdots$$

for weights a and b. We specify this problem with a synchronous process with two channels: *left*, used for input, and *right* used for output. Since each output requires two consecutive values from the input stream, the first output cannot occur before the third clock cycle.

clock	0	1	2	3	4	5	
left	$\mathtt{x}_0$	$\mathtt{x}_1$	$\mathtt{x}_2$	$\mathtt{x}_3$	$\mathtt{x}_4$	$\mathtt{x}_5$	$\cdots$
right			$a * \mathtt{x}_0 + b * \mathtt{x}_1$	$a * \mathtt{x}_1 + b * \mathtt{x}_2$	$a * \mathtt{x}_2 + b * \mathtt{x}_3$	$a * \mathtt{x}_3 + b * \mathtt{x}_4$	$\cdots$

Hoare's solution performs the two multiplications in parallel and then adds the results. Suppose the implementation technology is a single field-programmable gate array; the circuitry for the computation of the output would then be inherently parallel anyway. Let's assume instead that we want to implement the two multiplications on separate FPGAs. It's clear that the a-product is always ready one clock cycle before we need to perform the addition. Let's keep this intermediate result in the variable m:

clock	0	1	2	3	4	5	
left	$\mathtt{x}_0$	$\mathtt{x}_1$	$\mathtt{x}_2$	$\mathtt{x}_3$	$\mathtt{x}_4$	$\mathtt{x}_5$	$\cdots$
m		$a * \mathtt{x}_0$	$a * \mathtt{x}_1$	$a * \mathtt{x}_2$	$a * \mathtt{x}_3$	$a * \mathtt{x}_4$	$\cdots$
right			$m + b * \mathtt{x}_1$	$m + b * \mathtt{x}_2$	$m + b * \mathtt{x}_3$	$m + b * \mathtt{x}_4$	$\cdots$

First however, note we are going to target a Handel-C-like scenario where channel communication and assignment take one-clock cycle, and we have shared variables. We need to reason about interleavings of assignments, but rather than re-work the whole theory to have state-sequences, we simply convert assignments into channel communications. So for the following case study, we have the following shorthands:

shorthand	expansion
$c?_1 x$	$c?x \rightarrow Wait\,1.$
$c!_1 x$	$c!x \rightarrow Wait\,1.$
$x :=_1 e$	$(a!_1 e \,[\![\emptyset \mid a \mid x]\!]\, a?_1 x)$ where a is fresh.
δP	variables modified by P i.e used in $x := \ldots$ or $c?x$
$P \,\|\|\|\, Q$	$P \,[\![\delta P \mid \emptyset \mid \delta Q]\!]\, Q$

In effect the clock-cycle wait is built into the communication and assignment notations, effectively avoid any Zeno hazards. Now we're ready to specify the problem as a slotted *Circus* process.

$$\begin{aligned} WS \;\widehat{=}\; & \textsf{var}\; x, m : \mathbb{N} \bullet (\; left?_1 x \;;\; (\; left?_1 x \,|||\, m :=_1 a * x \;);\\ & \qquad (\; \mu\, X \bullet (\; left?_1 x \,|||\, m :=_1 a * x \,|||\, right!_1 (m + b * x)\;) \;;\; X\;)\;) \end{aligned}$$

The process WS is clearly deadlock and livelock free: it is a non-stopping process with no internal synchronisations; and it is hiding and chaos-free, with guarded recursion. Now we need to decompose WS into two parallel processes with encapsulated state. We can replace the use of m by a channel communication that passes the intermediate value. One process (WSL) will receive the input stream and compute the a-product; the other (WSR) will compute the b-product and the sum, and generate the output stream. But now we see a problem with WS. The value $\mathtt{x}_1$ is received by WSL in the first clock cycle, and so it can be communicated to WSR in the second cycle. So it can't be used by WSR until the third clock cycle. So we need to delay the output on the *right* by another clock cycle. Our timing diagram shows this more clearly.

clock	0	1	2	3	4	5	
left	$\mathtt{x}_0$	$\mathtt{x}_1$	$\mathtt{x}_2$	$\mathtt{x}_3$	$\mathtt{x}_4$	$\mathtt{x}_5$	$\cdots$
w		$\mathtt{x}_0$	$\mathtt{x}_1$	$\mathtt{x}_2$	$\mathtt{x}_3$	$\mathtt{x}_4$	$\cdots$
m			$a * \mathtt{x}_0$	$a * \mathtt{x}_1$	$a * \mathtt{x}_2$	$a * \mathtt{x}_3$	$\cdots$
right				$m + b * \mathtt{x}_1$	$m + b * \mathtt{x}_2$	$m + b * \mathtt{x}_3$	$\cdots$

Here's another version of WS that does this.

$$\begin{aligned} WS' \;\widehat{=}\; & \textsf{var}\; w, x, m : \mathbb{N} \bullet \\ & \quad left?_1 x \;;\; (\; left?_1 x \,|||\, w :=_1 x \;);\\ & \quad (\; left?_1 x \,|||\, w :=_1 x \,|||\, m :=_1 a * w \;);\\ & \quad (\; \mu\, X \bullet (\; left?_1 x \,|||\, w :=_1 x \,|||\, m :=_1 a * x \,|||\, right!_1 (m + b * w)\;) \;;\; X\;) \end{aligned}$$

Our refinement strategy is to split into two processes. The variable x belongs in WSL, since it is used to store the current input. The variable m can be placed in WSR, since it is used directly in producing outputs, but its value must be computed in WSL, and so the value will have to be communicated from left to right. The variable w records the previous input, and this is used in both left and right processes; so we duplicate its value using a ghost variable v. The ghost variable can then be used in the right-hand process in the calculation of the output on the right. Our refinement starts with organising the variables. (To reduce clutter, we abbreviate $left?_1 x$ by $?_1 x$

and $\mathit{right}!_1 e$ by $!_1 e$. We also separate the beginning and end of variable scopes.)

$$
\begin{array}{l}
\quad \mathsf{var}\ w, x, m\ ; \\
\qquad ?_1 x\ ;\ (?_1 x \ |||\ w :=_1 x)\ ;\ (?_1 x \ |||\ w, m :=_1 x, a * w)\ ; \\
\qquad (\mu\, X \bullet (?_1 x \ |||\ w, m :=_1 x, a * w \ |||\ !_1 (m + b * w))\ ;\ X)\ ; \\
\quad \mathsf{end}\ w, x, m \\
= \{\ v\ \mathit{ghosts}\ w\ \} \\
\quad \mathsf{var}\ w, x, m\ ; \\
\qquad ?_1 x\ ;\ (?_1 x \ |||\ w :=_1 x)\ ; \\
\qquad \mathsf{var}\ v\ ; \\
\qquad\quad (?_1 x \ |||\ v, w, m :=_1 x, x, a * w)\ ; \\
\qquad\quad (\mu\, X \bullet (?_1 x \ |||\ v, w, m :=_1 x, x, a * w \ |||\ !_1 (m + b * v))\ ;\ X)\ ; \\
\qquad \mathsf{end}\ v\ ; \\
\quad \mathsf{end}\ w, x, m \\
= \{\ \mathit{widen\ scope}\ \} \\
\quad \mathsf{var}\ v, w, x, m\ ; \\
\qquad ?_1 x\ ;\ (?_1 x \ |||\ w :=_1 x)\ ;\ (?_1 x \ |||\ v, w, m :=_1 x, x, a * w)\ ; \\
\qquad (\mu\, X \bullet (?_1 x \ |||\ v, w, m :=_1 x, x, a * w \ |||\ !_1 (m + b * v))\ ;\ X)\ ; \\
\quad \mathsf{end}\ v, w, x, m
\end{array}
$$

Our next step is to insert some hidden events to prepare for the communication of values between the two processes. We add two hidden channels: c communicates x's value; and mid communicates m's value. These events are not needed in the first two steps.

$$
\begin{array}{l}
= \{\ \mathit{hiding}\ \} \\
\quad (\ \mathsf{var}\ v, w, x, m\ ; \\
\qquad ?_1 x\ ;\ (?_1 x \ |||\ w :=_1 x)\ ;\ (?_1 x \ |||\ c.x \ |||\ \mathit{mid}.a * w \ |||\ v, w, m :=_1 x, x, a * w)\ ; \\
\qquad (\mu\, X \bullet (?_1 x \ |||\ c.x \ |||\ \mathit{mid}.a * w \ |||\ v, w, m :=_1 x, x, a * w \ |||\ !_1 (m + b * v))\ ;\ X)\ ; \\
\quad \mathsf{end}\ v, w, x, m\ ) \setminus \{\!|c, \mathit{mid}|\!\}
\end{array}
$$

Now we can prepare for the parallel split by organising each step into parallel parts, examining each atomic action and assigning it to the left or right component. The right-hand process doesn't need to do anything during the first two steps, so we make it wait. In the third step, the the input belongs on the left. The pair of actions $(c.x \ |||\ v :=_1 x)$ can be replaced by a communication: the left performs the output $c!_1 x$ and the right performs the input $c?_1 v$. Similarly, $(\mathit{mid}.a{*}w \ |||\ m :=_1 a{*}w)$ can be replaced by $\mathit{mid}!_1 a * w$ and $\mathit{mid}?_1 m$. Finally, the assignment to w belongs on the left. The body of the recursion is split in exactly the same way, with the addition of the output being assigned to the right-hand process.

$$
\begin{array}{l}
= \{\ \mathit{parallel}\ \} \\
\quad (\ \mathsf{var}\ v, w, x, m\ ; \\
\qquad (?_1 x \parallel \mathit{Wait}\, 1)\ ; \\
\qquad ((?_1 x \mid\mid\mid w :=_1 x) \parallel \mathit{Wait}\, 1)\ ; \\
\qquad ((?_1 x \mid\mid\mid c!_1 x \mid\mid\mid \mathit{mid}!_1 a * w \mid\mid\mid w :=_1 x) \parallel (c?_1 v \mid\mid\mid \mathit{mid}?_1 m))\ ; \\
\qquad (\mu\, X \bullet (\ ((?_1 x \mid\mid\mid c!_1 x \mid\mid\mid \mathit{mid}!_1 a * w \mid\mid\mid w :=_1 x) \\
\qquad\qquad \parallel (c?_1 v \mid\mid\mid \mathit{mid}?_1 m \mid\mid\mid !_1(m + b * v))\ )\ ;\ X))\ ; \\
\quad \mathsf{end}\ v, w, x, m\ ) \setminus \{\!|c, \mathit{mid}|\!\}
\end{array}
$$

We also need to split the recursion into two parallel parts. Since the body comprises two atomic steps in parallel, the fixed-point operator distributes cleanly through the parallel operator.

$$
\begin{array}{l}
= \{\ \mathit{parallel}\ \} \\
\quad (\ \mathsf{var}\ v, w, x, m\ ; \\
\qquad (?_1 x \parallel \mathit{Wait}\, 1)\ ; \\
\qquad ((?_1 x \mid\mid\mid w :=_1 x) \parallel \mathit{Wait}\, 1)\ ; \\
\qquad ((?_1 x \mid\mid\mid c!_1 x \mid\mid\mid \mathit{mid}!_1 a * w \mid\mid\mid w :=_1 x) \parallel (c?_1 v \mid\mid\mid \mathit{mid}?_1 m))\ ; \\
\qquad (\ (\mu\, X \bullet (?_1 x \mid\mid\mid c!_1 x \mid\mid\mid \mathit{mid}!_1 a * w \mid\mid\mid w :=_1 x)\ ;\ X)\ ; \\
\qquad \parallel (\mu\, X \bullet (c?_1 v \mid\mid\mid \mathit{mid}?_1 m \mid\mid\mid !_1(m + b * v))\ ;\ X)\ )\ ; \\
\quad \mathsf{end}\ v, w, x, m\ ) \setminus \{\!|c, \mathit{mid}|\!\}
\end{array}
$$

Now we can perform the parallel split, using an interchange law for sequence and parallel that is similar to the spreadsheet rules in UTP. We create the left-hand process by encapsulating w and x, retaining the left-hand parts, and discarding the right-hand parts. We create the right-hand process similarly.

$$
\begin{array}{l}
= \{\ \mathit{parallel\ split}\ \} \\
\quad (\ (\ \mathsf{var}\ w, x\ ; \\
\qquad ?_1 x\ ; \\
\qquad (?_1 x \mid\mid\mid w :=_1 x)\ ; \\
\qquad (?_1 x \mid\mid\mid c!_1 x \mid\mid\mid \mathit{mid}!_1 a * w \mid\mid\mid w :=_1 x)\ ; \\
\qquad (\mu\, X \bullet (?_1 x \mid\mid\mid c!_1 x \mid\mid\mid \mathit{mid}!_1 a * w \mid\mid\mid w :=_1 x)\ ;\ X)\ ; \\
\quad \mathsf{end}\ w, x\ ) \\
\quad \parallel \\
\quad (\ \mathsf{var}\ v, m\ ; \\
\qquad \mathit{Wait}\, 1\ ;\ \mathit{Wait}\, 1\ ; \\
\qquad (c?_1 v \mid\mid\mid \mathit{mid}?_1 m)\ ; \\
\qquad (\mu\, X \bullet (c?_1 v \mid\mid\mid \mathit{mid}?_1 m \mid\mid\mid !_1(m + b * v))\ ;\ X)\ ; \\
\quad \mathsf{end}\ v, m\ ) \\
\quad)\ \setminus \{\!|c, \mathit{mid}|\!\}
\end{array}
$$

Now we can tidy up the processes for our final result.

$$
\begin{array}{l}
(\ \mathsf{var}\ w, x : \mathbb{N} \bullet \\
\quad \mathit{left}?_1 x\ ;\ (\mathit{left}?_1 x \mid\mid\mid w :=_1 x)\ ;\ (\mathit{left}?_1 x \mid\mid\mid c!_1 x \mid\mid\mid \mathit{mid}!_1 a * w \mid\mid\mid w :=_1 x)\ ; \\
\quad (\mu\, X \bullet (\mathit{left}?_1 x \mid\mid\mid c!_1 x \mid\mid\mid \mathit{mid}!_1 a * w \mid\mid\mid w :=_1 x)\ ;\ X)\ ; \\
\parallel \\
\quad \mathsf{var}\ v, m : \mathbb{N} \bullet (\ \mathit{Wait}\, 2\ ;\ (c?_1 v \mid\mid\mid \mathit{mid}?_1 m)\ ; \\
\qquad\qquad (\mu\, X \bullet (c?_1 v \mid\mid\mid \mathit{mid}?_1 m \mid\mid\mid \mathit{right}!_1(m + b * v))\ ;\ X)\ ;) \\
\quad)\ \setminus \{\!|c, \mathit{mid}|\!\}
\end{array}
$$

Of course, since this is equal to WS, it is deadlock and livelock-free and computes the right results.

A key point of the above case-study is that it works in any of the instantiations mentioned so far for slotted-*Circus*, namely $\mathcal{CTA}$ or $\mathcal{MSA}$.

8 Related Work

During the development of Handel-C at Oxford, a lot of the principles and theory was developed and published [PL91, HIJ93]. Here the emphasis was very much on the verified compilation into hardware of an occam-like language. However with the commercialisation of this as the language Handel-C the formal aspects and hardware compilation parted company, and the Handel-C language acquired new constructs like "prialt" that were not treated in the literature.

Modern Handel-C [Cel02] also has the idea of connecting hardware with different clocks together using tightly controlled asynchronous interfaces. Modelling this kind of behaviour requires a theory that mixes time and asynchronicity, such as timed-CSP [Sch00].

There has been work done on hardware semantics, ranging from the "reFLect" language used by Intel for hardware verification [GJ06], to the language Esterel used mainly for the development of flight avionics [BG92]. The Intel approach is a suite of hardware description languages, model-checkers and theorem provers all written and/or integrated together using the reFLect language, aimed mainly at the verification of computer datapath hardware. The Esterel language is a hardware description language with a formal semantics, and so is quite low-level in character, and so in the context of this research could be considered a potential replacement of Handel-C as an implementation technology. However, it is unclear how well it would link to the kind of specification and refinement style of work that we are proposing to support.

9 Future Work

We have described a generic framework for instantiating a wide range of slotted-theories, capturing their common features. An important aspect that has yet to be covered is what distinguishes the 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 a law of $\mathcal{MSA}$, but not of $\mathcal{CTA}$, or slotted-*Circus* in general:

$$a \rightarrow b \rightarrow P = b \rightarrow a \rightarrow P$$

Also worthy of exploration are the details of the behaviour of the Galois links inbetween 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.

In order to deal with the asynchronously interfaced multiple-clock hardware now supported by Handel-C we will need to exploit the link from the slotted theories to the generally asynchronous *Circus* theory itself.

Also of interest will be to consider to what extent the work on "generic composition" [Che02, Che06] can contribute to a clear and or tractable presentation of this theory.

10 Conclusions

A framework for giving UTP semantics to a class of reactive systems whose execution is demarcated by regular clock ticks has been presented. The general nature of the observational variables and the key operations on same have been discussed, showing how they are used build to both the healthiness conditions and the language semantics. A key result of this work has been the care needed to get a satisfactory definition of **R2**, and exposing the fact that certain synchronous theories like SCSP do not fit this particular UTP pattern for describing reactive systems.

10.1 Acknowledgement

We would like to thank the Dean of Research at TCD and Qinetic for their support of this work, and the comments of the anonymous reviewers, which helped improve key material in this paper.

References

[Bar93] Janet E. Barnes. A Mathematical Theory of Synchronous Communication. Technical Monograph PRG-112, Oxford University Computing Laboratory Programming Research Group, Hilary Term 1993.

[BG92] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19:87–152, 1992.

[BW02] Andrew Butterfield and Jim Woodcock. Semantic domains for handel-C. *Electr. Notes Theor. Comput. Sci*, 74, 2002.

[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`.

[Che02] Yifeng Chen. Generic composition. *Formal Asp. Comput*, 14(2):108–122, 2002.

[Che06] Yifeng Chen. Hierarchical organisation of predicate-semantic models. In Steve Dunne and Bill Stoddart, editors, *UTP*, volume 4010 of *Lecture Notes in Computer Science*, pages 155–172. Springer, 2006.

[GJ06] Melham T. Grundy, J. and O'Leary J. A reflective functional language for hardware design and theorem proving. *Journal of Functional Programming*, 16(2):157–196, 2006.

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

[HIJ93] H. Jifeng, I. Page, and J. Bowen. Towards a provably correct hardware implementation of Occam. In G.J. Milne and L. Pierre, editors, *Correct Hardware Design and Verification Methods*, volume 683 of *Lecture Notes in Computer Science*, pages 214–225, Arles, France, May 1993. IFIP WG10.2, Springer-Verlag.

[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, Upper Saddle River, NJ, USA, 1985. Prentice-Hall, Inc.

[PL91] I. Page and W. Luk. Compiling Occam into field-programmable gate arrays. In W. Moore and W. Luk, editors, *FPGAs, Oxford Workshop on Field Programmable Logic and Applications*, pages 271–283, 15 Harcourt Way, Abingdon OX14 1NV, UK, 1991. Abingdon EE&CS Books.

[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.

[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.

[Spi87] Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 1987.

[WC01a] J. C. P. Woodcock and A. L. C. Cavalcanti. A Concurrent Language for Refinement. In A. Butterfield and C. Pahl, editors, *IWFM'01: 5th Irish Workshop in Formal Methods*, BCS Electronic Workshops in Computing, Dublin, Ireland, July 2001.

[WC01b] Jim Woodcock and Ana Cavalcanti. *Circus*: a concurrent refinement language. Technical report, University of Kent at Canterbury, October 2001.

[WC02] Jim Woodcock and Ana Cavalcanti. The semantics of circus. In *ZB*, pages 184–203, 2002.

A Generic Laws

The functions and relations over $\mathcal{H}\ E$ required to define a slotted-*Circus* theory, need to satisfy the following laws:

[ET:elems] $EqvTrc(tr, hist) \Rightarrow elems(tr) = Acc(hist)$

[HIST:eq] $(h_1 = h_2) \ \equiv \ \forall\, tr \bullet EqvTrc(tr, h_1) \equiv EqvTrc(tr, h_2)$

[HN:null] $Acc(HNull) = \{\}$

[pfx:refl] $hist \preceq hist = \textsc{True}$

[pfx:trans] $hist_1 \preceq hist_2 \land hist_2 \preceq hist_3 \Rightarrow hist_1 \preceq hist_3$

[pfx:anti-sym] $hist_1 \preceq hist_2 \land hist_2 \preceq hist_1 \Rightarrow hist_1 = hist_2$

[SN:pfx] $HNull \preceq hist$

[ET:pfx] $hist_1 \preceq hist_2 \Rightarrow \exists\, tr_1, tr_2 \bullet EqvTrc(tr_1, hist_1) \land EqvTrc(tr_2, hist_2) \land tr_1 \leq tr_2$

[Sadd:events] $Acc(Sadd(h_1, h_2)) = Acc(h_1) \cup Acc(h_2)$

[Sadd:unit] $Sadd(h_1, h_2) = h_1 \equiv h_2 = HNull$

[Sadd:assoc] $Sadd(h_1, Sadd(h_2, h_3)) = Sadd(Sadd(h_1, h_2), h_3)$

[Sadd:prefix] $h \preceq Sadd(h, h')$

[Ssub:pre] $\text{pre}\ Ssub(h_1, h_2) = h_2 \preceq h_1$

[Ssub:events] $h_2 \preceq h_1 \land h' = Ssub(h_1, h_2) \Rightarrow$
$Acc(h_1) \setminus Acc(h_2) \subseteq Acc(h') \subseteq Acc(h_1)$

[SSub:self] $Ssub(h, h) = HNull$

[SSub:nil] $Ssub(h, HNull) = h$

[SSub:same] $hist \preceq hist'_a \land hist \preceq hist'_b \Rightarrow$
$Ssub(hist'_a, hist) = Ssub(hist'_b, hist) \ \equiv \ hist'_a = hist'_b$

[SSub:subsub] $hist_c \preceq hist_a \land hist_c \preceq hist_b \land hist_b \preceq hist_a$
$\Rightarrow Ssub(Ssub(hist_a, hist_c), Ssub(hist_b, hist_c)) = Ssub(hist_a, hist_b)$

[Sadd:Ssub] $hist \preceq hist' \Rightarrow Sadd(hist, Ssub(hist', hist)) = hist'$

[Ssub:Sadd] $Ssub(Sadd(h_1, h_2), h_1) = h_2$

[SHid:evts] $Acc(SHide(hid)(h)) = Acc(h) \setminus hid$

[SNC:sym] $SSync(cs)(h_1, h_2) = SSync(cs)(h_2, h_1)$

[SNC:one] $\forall\, h' \in SSync(cs)(h_1, HNull) \bullet Acc(h') \subseteq Acc(h_1) \setminus cs$

[SNC:only] $h' \in Acc(SSync(cs)(h_1, h_2)) \Rightarrow Acc(h') \subseteq Acc(h_1) \cup Acc(h_2)$

[SNC:sync] $h' \in Acc(SSync(cs)(h_1, h_2)) \Rightarrow cs \cap Acc(h') \subseteq cs \cap (Acc(h_1) \cap Acc(h_2))$

[SNC:assoc] $SyncSet(cs)(h_1)(SSync(cs)(h_2, h_3)) = SyncSet(cs)(h_3)(SSync(cs)(h_1, h_2))$

B Proofs for R2-ness of $=$ and $\mathcal{MSA}$ prefix

$$
\begin{array}{ll}
 & \mathbf{R2}(slots' = slots) \\
\equiv & \text{“ defn. } \mathbf{R2}\text{, apply substitution, shorthand } RL(s) = Ref(last(s)) \text{ ”} \\
 & \exists\, ss \bullet ss \sharp (slots' \backslash\!\backslash slots) = ss \wedge RL(slots) = RL(ss) \\
\equiv & \text{“ Property 1 (below) ”} \\
 & \exists\, ss \bullet slots' \backslash\!\backslash slots = \langle SNull(RL(ss)) \rangle \wedge RL(slots) = RL(ss) \\
\equiv & \text{“ Property 2 (below) ”} \\
 & \exists\, ss \bullet front(slots') = front(slots) \wedge tail(slots').1 = tail(slots).1 \\
 & \quad \wedge RL(slots') = RL(ss) \wedge RL(slots) = RL(ss) \\
\equiv & \text{“ Liebniz, restrict quantification scope ”} \\
 & front(slots') = front(slots) \wedge tail(slots').1 = tail(slots).1 \\
 & \wedge RL(slots') = RL(slots) \wedge \exists\, ss \bullet RL(slots) = RL(ss) \\
\equiv & \text{“ defn. of equality, witness } ss = slots \text{ ”} \\
 & slots = slots'
\end{array}
$$

Property 1: $(\, ss \sharp tt = ss \,) \equiv tt = \langle SNull(RL(ss)) \rangle$

Property 2: $(tt' \backslash\!\backslash tt) = \langle SNull(r) \rangle \equiv front(tt) = front(tt') \wedge last(tt).1 = last(tt').1 \wedge RL(tt') = r$

$$
\begin{array}{ll}
 & bag_1 \preceq bag_2 \\
\equiv & \text{“ defn. of prefix ”} \\
 & bag_1 \sqsubseteq bag_2 \\
\equiv & \text{“ bag property ”} \\
 & \exists\, bag_\Delta \bullet bag_2 = bag_1 \oplus bag_\Delta \\
\equiv & \text{“ bag property: } \forall\, bag \bullet \exists\, tr \bullet items(tr) = bag \text{ ”} \\
 & \exists\, bag_\Delta, tr_\Delta, tr_1, \bullet bag_2 = bag_1 \oplus bag_\Delta \wedge items(tr_\Delta) = bag_\Delta \wedge items(tr_1) = bag_1 \\
\equiv & \text{“ One-point rule backwards } tr_2 = tr_1 \frown tr_\Delta \text{ ”} \\
 & \exists\, bag_\Delta, tr_\Delta, tr_1, tr_2 \bullet bag_2 = bag_1 \oplus bag_\Delta \wedge items(tr_\Delta) = bag_\Delta \wedge items(tr_1) = bag_1 \wedge tr_2 = tr_1 \frown tr_\Delta \\
\equiv & \text{“ One-point rule } bag_\Delta\text{, Liebniz } bag_1 \text{ ”} \\
 & \exists\, tr_\Delta, tr_1, tr_2 \bullet bag_2 = items(tr_1) \oplus items(tr_\Delta) \wedge items(tr_1) = bag_1 \wedge tr_2 = tr_1 \frown tr_\Delta \\
\equiv & \text{“ } items \text{ is a sequence homomorphism ”} \\
 & \exists\, tr_\Delta, tr_1, tr_2 \bullet bag_2 = items(tr_2) \wedge bag_1 = items(tr_1) \wedge tr_2 = tr_1 \frown tr_\Delta \\
\equiv & \text{“ sequence property ”} \\
 & \exists\, tr_\Delta, tr_1, tr_2 \bullet bag_2 = items(tr_2) \wedge bag_1 = items(tr_1) \wedge tr_\Delta = tr_2 - tr_1 \\
\equiv & \text{“ One point rule: } tr_\Delta\text{, requires definedness of } tr_2 - tr_1 \text{ ”} \\
 & \exists\, tr_1, tr_2 \bullet bag_2 = items(tr_2) \wedge bag_1 = items(tr_1) \wedge tr_1 \leq tr_2 \\
\equiv & \text{“ def. of } EqvTrc\text{, backwards ”} \\
 & \exists\, tr_1, tr_2 \bullet EqvTrc(tr_2, bag_2) \wedge EqvTrc(tr_1, bag_1) \wedge tr_1 \leq tr_2
\end{array}
$$