State Visibility and Communication in Unifying Theories of Programming

Andrew Butterfield, Paweł Gancarski
Dept. of Computer Science
Trinity College Dublin
Ireland
{butrfeld,gawcarsp}@cs.tcd.ie

Jim Woodcock
Dept. of Computer Science
University of York
UK
jim@cs.york.ac.uk

Abstract

We explore the interactions between program-variable state visibility and communication behaviour in state-rich CSP-like processes, using the Unifying Theories of Programming (UTP) framework. The key results of this work are: having variable state visible while a process is waiting to communicate, results in an operationally complex theory of behaviour; by contrast, considering state as unobservable during communication wait-periods results in an elegant theory, with much cleaner operational intuitions. The language constructs most affected by this observability choice are those of external choice and parallel composition. We also discuss situations where this state hiding can prevent the adoption of interesting operators that seize control from waiting processes.

1. Introduction

The formal notation Circus [1] is a unification of Z and CSP, to give a “state-rich” process algebra, with (restricted) global shared variables. Circus has been given a semantics in UTP [2], and there has been work in developing extensions to it in UTP: adding discrete time (Circus Timed Actions [3]); and developing a generic notion of time-slots (slotted-Circus [4]).

This paper addresses a number of foundational issues that arise when considering the interaction between the reactive systems theory and the need to manage explicit state that arises in both Circus and slotted-Circus. In particular, we find that external choice introduces miracles if process program-variable state is visible when waiting on external events. The contribution of this paper is a clear exposition of both the problem, and the range of possible solutions available, in a context where we might want to link these theories to others, also defined in the UTP framework. It also demonstrates the ease with which this framework can be used to explore choices during theory development.

1.1. Structure and Focus

In §2 we give a comprehensive introduction to the UTP theory of reactive systems with state, while in §3 we discuss the problematic interaction between state visibility and waiting. In §4 we look at a number of approaches that can be used to resolve the problem, with an emphasis on the benefits and disadvantages of each. Finally, in §5 we mention related work and then we conclude in §6.

2. Background

We present the relevant background to UTP in three parts: the first discusses general principles, and introduces the notion of a design; the second summarises a theory of “state-rich” CSP as reactive designs; whilst the third discusses the issue of state-visibility in processes waiting on events. A far more comprehensive treatment of the first two of these issues, can be found in [5].

2.1. UTP: General Principles

Theories in UTP are expressed as (second-order) predicates over a pre-defined collection
of free observation variables, referred to as the alphabet of the theory, and are generally used to describe a relation between a before-state and an after-state, the latter typically indicated by dashed versions of the observation variables. A condition is a predicate whose free variables are all undashed. In Fig. 1 we show operators common to most UTP theories: Sequential composition \((P; Q)\) corresponds to relational composition, \(i.e.,\) the existence of an intermediate state \((\nu_m)\), such that \(P\) relates \(\nu\) to \(\nu_m\), whilst \(Q\) relates \(\nu_m\) to \(\nu'\), where \(\nu\) here denotes all the observational variables. The notation \([e_1, \ldots, e_n/x_1, \ldots, x_n]\) denotes the simultaneous substitution of the \(e_i, i \in 1 \ldots n\) for any free occurrence of the corresponding \(x_i\). The conditional \(P <c> Q\) 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 \(P \cap Q\) is simply logical disjunction. We capture the notion of refinement \(\subseteq\) as logical entailment between the implementation and specification predicates, universally quantified over all free variables (denoted by [\ldots\ldots\ldots]).

A given theory is characterised by a series of healthiness conditions that constrain the valid assertions that predicates may make. Healthiness conditions are characterised by a monotonic idempotent predicate transformer, with healthy predicates being fixpoints of these.

As an example of healthiness, one that will be used in the sequel, we introduce the notion of a UTP “design”, [6, Chp. 3], [5, §4], characterised by an observation of stability \((ok : \mathbb{B})\), that captures a program that has not failed (non-divergent). Note that in the sequential world, divergence covers non-termination. So, at present we interpret \(ok\) being true denotes a program that has started, whilst \(ok'\) being true captures successful termination. The observation variables correspond to program variables (e.g. \(x, y, z\)) and we use \(\nu\) to denote the vector of all such variables not otherwise mentioned.

The first condition \(H1\) states that a predicate \(P\) makes no assertions until it is started:

\[
H1(P) =_{def} ok \Rightarrow P
\]

The second states that a predicate should not mandate non-termination (if non-termination is possible, then so is termination):

\[
H2(P) \equiv P; (ok \Rightarrow ok') \land \nu' = \nu
\]

The effect of \(H2(P)\) is to suppress any assertion of \(\neg ok'\) by \(P\). All other observations, of program variables, here represented by \(\nu\), are unchanged. We now use the term design to denote a behaviour described by a predicate that is a fixed point of both \(H1\) and \(H2\).

Interestingly, we can define designs directly in UTP, by factoring out the \(ok\) and \(ok'\) observations. We define the following shorthand, where \(Q\) and \(R\) do not mention \(ok\) or \(ok'\):

\[
Q 
\vdash R =_{def} (ok \land Q) \Rightarrow (R \land ok')
\]

and note that predicates of this form are designs, and any design \(D\) can be written in this form.

\[
D \equiv \neg D[false/ok'] \vdash D[true/ok']
\]

2.2. CSP in UTP

1. In [6, Chp. 8] a theory of reactive systems is introduced, to cover process algebras like CSP[7], and, to a lesser extent, CCS[8]. To this end, observations of event traces \((tr : Event^*)\) and refusals \((ref : \mathcal{P}Event)\) have to be added, to support semantic models such as CSP’s failures. In addition, the boolean values of \(ok\) are not enough to characterise the key modes of operation of a process. As well as observing termination/stability, we also need to distinguish cases where the process has not terminated, but is stable and waiting for some event to occur. We introduce another boolean observation \(wait\), whose value is

1. We assume that binary predicate connectives have the following precedence ordering (tightest first) \(\land, \lor, \Rightarrow, \equiv\).
\( R = \text{def} \ R_3 \circ R_2 \circ R_1 \)

\( R_1(P) = \text{def} \ P \land tr \leq tr' \)

\( R_2(P) = \text{def} \ \exists s \cdot P[s, s \cdot (tr' - tr)/tr, tr'] \)

\( R_3(P) = \text{def} \ \Pi_{\text{nea}} <\text{wait}> P \)

\( \Pi_{\text{nea}} = \text{def} \ (\neg ok \land tr \leq tr') \lor (ok' \land wait' = wait \land tr' = tr \land ref' = ref) \)

\( \text{CSP1}(P) = \text{def} \ P \lor \neg ok \land tr \leq tr' \)

\( \text{CSP2}(P) = \text{def} \ P ; (ok \rightarrow ok') \land wait' = wait \land tr' = tr' \land ref' = ref \land v' = v \)

\[ H_{\text{rd}} = \text{def} \ (\neg ok \land tr \leq tr') \lor (ok' \land wait' = wait \land tr' = tr \land ref' = ref \land v' = v) \]

\[ \text{CSP2}_{\text{rd}}(P) = \text{def} \ P ; (ok \rightarrow ok') \land wait' = wait \land tr' = tr' \land ref' = ref \land v' = v \]

Figure 2. Reactive and CSP Healthiness

Figure 3. Reactive Skip and CSP2

only relevant when \( ok \) is true, and which denotes a process waiting for an event:

\( \neg ok \quad \text{— unstable, diverging} \)

\( ok \land wait \quad \text{— stable, waiting} \)

\( ok \land \neg wait \quad \text{— stable, terminated} \)

Note that in effect we have teased out the observation of non-termination from that of general divergence, and use \( \text{wait} \) to capture this distinction. The introduction of \( \text{wait} \) allows us to view process behaviour at intermediate execution points, rather than just beginning and end, and is what distinguishes reactive systems from designs.

Reactive systems in general are characterised by three healthiness conditions (Fig. 2) (all of which commute). The first, \( R_1 \), forbids alteration of the past (no time travel), whilst the second, \( R_2 \), asserts that if we replace \( tr \) by an arbitrary \( s \), and shift \( tr' \) to \( s \cdot (tr' - tr) \), then the predicate is unchanged (no event memory). Condition \( R_3 \) simply adds to the stipulation that if not started (\( \text{wait} = \text{true} \)), then a process agrees with whatever is running before it, propagating any divergence that may have emerged up to that point.

For CSP processes, there are five extra conditions, \( \text{CSP1}–\text{CSP5} \), of which the first two characterise the basic behaviour common to all CSP processes (Fig. 2). The first states that a CSP process, if started in an unstable state, can only guarantee the extension of the trace, and no more. The second, just like \( H_2 \), disallows processes that mandate instability as an outcome, in a similar fashion to \( H_2 \). In [5], the reactive theory is reformulated by treating CSP processes as “reactive designs”, by showing that a CSP process \((R, \text{CSP1, CSP2}-\text{healthy})\) can be expressed as an \( R \)-healthy design:

\[ P \equiv R(\neg \text{Pf}_f \vdash \text{Pf}_j) \]

Here we introduce the following shorthand notation: \( \text{Pf}_{\text{c}} = \text{def} \ P[b, c/ok', \text{wait}] \).

2.3. “state-rich” CSP in UTP

There is little mention of program variable state in [6, Chp. 8], and none in [5], but its integration with the reactive theory is given a comprehensive treatment in [9] treating CSP processes as “reactive designs”. Program variables are introduced as observational variables \( (v) \), and the question that arises is how we integrate these variables into the theory — in particular how are \( H \) and \( \text{CSP2} \) modified? For now, we follow [9] in adopting the definitions in Fig. 3, which simply add \( v' = v \) in the appropriate place. We assume \( R_3 \) to be redefined using \( H_{\text{rd}} \), and will denote it as \( R_{3, \text{rd}} \) when we need to emphasise this fact.

Reactive designs form a boolean lattice under the refinement ordering with \( R(\text{true} \vdash \text{false}) \) as top and \( R(\text{false} \vdash \text{true}) \) as bottom. The top process simplifies to \( R(\neg \text{ok}) \), and is the “miraculous”
process that refines any other, whereas the bottom process simplifies to \( R(\text{true}) \), and is refined by any other.

\[
R(\text{true}) \sqsubseteq (P \parallel Q) \sqsubseteq R(\lnot \text{ok})
\]

We now consider a simple subset of Circus[10], a state-rich process algebra, whose syntax is similar to that of CSP, with the addition of assignment (Figure 4). We now briefly present the semantics of the language constructs as reactive designs (see Figure 5), following the presentation of [9]. Chaos is the most non-deterministic healthy behaviour possible. Stop is deadlock, it is stable, performs no events, but never terminates. Skip terminates immediately, without events or state-change occurring. Assignment \( x := e \) updates variable \( x \) with the value of expression \( e \). Prefix \( a \rightarrow \text{Skip} \) waits for event \( a \) to happen, not refusing it, and then performs it and terminates, without changing the state. We then define \( a \rightarrow P \) for general \( P \) as \( a \rightarrow \text{Skip}; P \).

External choice \( P \parallel Q \) allows the environment to choose between \( P \) and \( Q \) based on the first events in which each are willing to participate. The precondition is the conjunction of those of \( P \) and \( Q \), whilst the post-condition has two cases: if \( tr' = tr \land \text{wait}' \) then no events have occurred, and we are waiting for one, so both \( P \) and \( Q \) must agree on refusals; otherwise some events have occurred or we have terminated, so the behaviour is that of \( P \) or \( Q \) as appropriate. \( P \parallel Q \) runs \( P \) and \( Q \) in parallel, synchronising on events in both their alphabets and only allowed to modify disjoint variable sets. The full definition of parallel composition is quite complex [11], and we only give a partial description that suffices for our purposes. For the postcondition, we run \( P \) and \( Q \) simultaneously with output variables renamed, and then sequentially compose their outcomes with a parallel merge predicate [6, Chp. 7]. The merge predicate \( M_{||} \) states, among other things, that the final observed trace is any that is consistent with traces from \( P \) and \( Q \) that agree on synchronisation events in \( A \), and that the whole construct is waiting when any component is. Termination occurs when both \( P \) and \( Q \) have terminated, and then \( \text{MS}t \) describes how the final program variable state is determined from that of \( P \) and \( Q \).

2.4. Deadlock ignores state

The observant reader might have noticed that, while \( \Pi_{\text{Id}}, \text{Skip}, \) and \( a \rightarrow \text{Skip} \) all keep the variable-state unchanged, \( \text{Stop} \) does not mention it, and parallel does only when \( \text{wait}' \) is false. In [9], the reason given for \( \text{Stop} \) ignoring state has to do with its interaction with external choice (\( \square \)), because choices are made based on events, not state. It is shown that definition is required to get the following important CSP law:

\[
P \parallel \text{Stop} \equiv P \quad \text{for healthy } P;
\]

i.e., that \( \text{Stop} \) is the unit for external choice. For parallel composition, the state is only merged when both processes terminate, as the operational intuition here is that each side takes its own copy of the state, modifies permitted variables as it goes along, and we only see the overall effect when everything has terminated.

3. The Problem

First, consider the semantics of the process \( US(x, e, a) \) that updates variable \( x \) with value \( e \), and then synchronises on event \( a \):

\[
US(x, e, a) =_{\text{def}} x := e ; a \rightarrow \text{Stop}
\]

Calculation (using laws found variously in [5], [11], [9]), gives the result shown in Fig. 6. We see that \( US(x, e, a) \) does not refuse \( a \) until it performs it, and that \( x \) is updated with the value of \( e \), as expected. Now let us consider the following external choice, where \( e \) and \( f \) are not equal:

\[
US(x, e, a) \square US(x, f, b)
\]

However the consequences in our current theory of this (Fig. 6), is when waiting for either \( a \) or \( b \) to occur, i.e., after having had both assignments.
diverging state (occur, is to behave as if we have been started in a
the definition reduces to:
the denotation of this process, thus violating the
does not appear, so the empty trace is not in
wait
leads to an assertion that
of the two processes when waiting for the first
waited, and whose details are not important
the final outcome of the choice, once it has
where
merge, agreeing on
set of traces when \( tr_1 \) and \( tr_2 \) merge, agreeing on A (see [11])
MSt \( =_{\text{def}} \forall x \bullet (x \neq 1.x \Rightarrow x' = 1.x) \land (x \neq 2.x \Rightarrow x' = 2.x) \land (x = 1.x = 2.x \Rightarrow x' = x) \)

Figure 5. Circus semantics

\[
\begin{align*}
US(x, e, a) &= \mathcal{R}(\text{true} \vdash (tr' = tr \land a \notin \ref') \triangleright tr' = tr \land \langle a \rangle) \land u' = u \land x' = e) \\
US(x, e, a) \perp US(x, f, b) &= \mathcal{R}(\neg \mathcal{R}_1(\neg ok) \vdash \mathcal{R}_1(\neg ok) \triangleright tr' = tr \land \text{wait}' \triangleright \mathcal{R}_1(\neg ok) \Rightarrow \ldots))
\end{align*}
\]

Figure 6. Key Calculation outcomes

occur, is to behave as if we have been started in a
diverging state (\( \mathcal{R}_1(\neg ok) \)), directly contradicting the pre-condition. The problem arises because external choice effectively takes the conjunction of the two processes when waiting for the first (deciding event). In this case this conjunction leads to an assertion that \( x' = e \land x' = f \), which reduces it to false. Further calculation shows that the definition reduces to:

\[
\mathcal{R}(\text{true} \vdash \neg \text{wait}' \land \text{CHOOSE}) \lor \mathcal{R}_1(\neg ok)
\]

where \( \text{CHOOSE} \) is the disjunction that captures the final outcome of the choice, once it has terminated, and whose details are not important here, other than one disjunct asserts \( tr' = tr \land \langle a \rangle \) while the other asserts \( tr' = tr \land \langle b \rangle \). What is important is that the term \( \text{wait}' \) does not appear, so we have a process that never waits for an event, but insists that the choice, and event occur immediately. Also, the option to have \( tr' = tr \) does not appear, so the empty trace is not in the denotation of this process, thus violating the
prefix-closure principle of CSP.

We can observe the consequences of not having the empty trace as a possible behaviour by putting the result in parallel with a prefix doing a different event, synchronising on all events, where we would expect deadlock:

\[
(US(x, e, a) \perp US(x, f, b)) \parallel (c \rightarrow \text{Skip})
\]

Calculation shows this reduces to \( \mathcal{R}(\neg ok) \), i.e., the reactive miracle. The difficulty we have here is that a composition of processes built up from assignments and simple prefixing, using external choice and parallel composition, has lead to a process that refines all specifications. It is clear that the theory as presented has a difficulty, which we must address. As we shall see, there are a number of choices we can make.

4. Possible Solutions

The problem is related to an issue that arose when defining \( \text{Stop} \), to ensure that it was a unit for
external choice. There, we ensured that \textit{Stop} made no assertions regarding the state, whilst always asserting \textit{wait}'. Inspection of our definitions show that some of our other definitions do assert \textit{wait}' and make an assertion regarding state, notably \( a \rightarrow \text{Skip} \). We now explore some alternatives approaches to handling this issue.

4.1. Fixing Semantic Definitions

The problem case just presented was caused by the definition of \( a \rightarrow \text{Skip} \) asserting \( v' = v \) when \( tr' = tr \land wait' \) was true, so we could avoid the problem by re-defining it (Fig. 7). Another alternative might be to modify the definition of external choice, so that it ignores the state in the waiting case (Fig. 7). Here the construct \( \exists v' \bullet P \) hides all mention of the final state. This tackles the problem at source, but at the cost of changing a general definition that works across a number of theories.

4.2. A new healthiness condition

There is a conceptual problem with fixing up individual language construct semantics as just described — it obscures a key, fundamental principle of our notion of healthy processes, namely that, for a healthy process:

\begin{center}
program variable state is not visible while waiting for external events
\end{center}

A key part of the philosophy of UTP is the ability to give a precise and formal answer to the question: what is a healthy process? In particular, should we extend the language with some new construct that involves communication, what formal guidance, if any exists to ensure that we avoid the pitfall we have seen with external choice?

So, another approach is to define a healthiness condition that captures this principle, which we now proceed to do. First we define a shorthand of a “boxed” process as one where the final variable values are considered as arbitrary:

\[
\boxed{P} =_{\text{def}} \left( \exists v' \bullet P \right)
\]

We can then directly define our new healthiness condition (which we call \textbf{R4}) as requiring state to be boxed when waiting:

\[
\textbf{R4}(P) =_{\text{def}} \boxed{P} \langle \text{wait'}\rangle P
\]

How do we merge \textbf{R4} with \textbf{R1-3}? The question is moot if \textbf{R4} commutes with the three healthiness conditions. For \textbf{R1} and \textbf{R2}, simple calculations show this to be the case, but for \textbf{R3} we have a complication. The current definition of \( II \) is not \textbf{R4}-healthy, because one of its disjuncts asserts both \( v' = v \) and \( \textit{wait}' = \textit{wait} \). This is unfortunate as \( II \) occurs throughout the theory in various guises, resulting for instance in \textit{Skip} not being \textbf{R4}. We could “fix” \( II \) to make it \textbf{R4}-healthy, but this then compromises its ability to act as a unit for sequential composition when combined with other healthy processes. The condition \textbf{R4} suggested here is appealing, but it soon becomes clear that introducing it involves re-working the theory to some degree. Also, its introduction into \textbf{R} requires a large range of proofs using \textbf{R} to be re-visited.

4.3. Re-working existing healthiness conditions

It turns out there is a simpler way to tackle this issue, which introduces a minimal change to the theory as a whole.

1) We replace the original \textbf{R3} by \textbf{R3h} (‘h’ for hide) defined as follows:

\[
\textbf{R3h}(P) =_{\text{def}} II \langle wait'\rangle P
\]

So a \textbf{R3h}-healthy process will not assert anything about its predecessor’s state if not started.

2) We also want to stop \( P \) exposing its state changes when it is started but waiting. We do this by considering a healthiness condition called \textbf{CSP4}:

\[
\textbf{CSP4}(P) =_{\text{def}} P; \text{Skip}
\]

As \textit{Skip} is defined using \textbf{R}, which will now incorporate \textbf{R3h}, it will, through the use of \textbf{CSP4}, enforce the hiding of \( P \)'s state while it is waiting.

So, we replace the original \textbf{R3} by \textbf{R3h} uniformly throughout the theory, and add \textbf{CSP4} to our “standard” notion of a healthy process. All other definitions, including \( II \), remain unchanged.

Finally, we would like to point out that our solution, whilst solving one problem, introduces potential difficulties for possible extensions to the
theory. Work by He Jifeng on hybrid CSP [12], and Alistair McEwan’s PhD thesis [13] discuss the notion of one process being able to interrupt another. This requires introducing a new form of “healthiness” condition (I3) that is the converse of R3, in that it allows a process to proceed only if the prior one is waiting: I3(P) = def P □ wait ⊥.

This leads to a form of state scope extrusion, quoting from [13, Chp.3, p45]:

If P contains some state that is in scope until P terminates, then that scope also extends to the interrupting process. In other words, the interrupting process has access to all of the state in P until it terminates.

The problem we face is that the need of the interrupting process to see the state of its predecessor while it is waiting contradicts our requirement to have it hidden. Possible approaches to reconciling this could be based on the fact that I3 processes, in combination with others, can be wrapped up in R3 to give an overall healthy outcome.

5. Related Work

In addition to the work done on state-rich reactive processes in UTP [14], [2] there has been attention paid to this issue by others. The implementors of occam [15] 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 [16], which was then followed up with real-time extensions [17]. 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.

A UTP semantics for Timed Communicating Object-Z (TCOZ [18]) is given in [19]. The theory presented there has a communication component which is a variant of He and Sherif’s CTA [20], 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 [21] 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.

6. Future Work and Conclusions

We have shown a potential problem arising from an interaction between state visibility and waiting states, and illustrated a number of approaches by which it can be resolved, at least within our existing UTP framework. The key observation is that many issues can be resolved either by choosing to modify language constructs, or the introduction of suitable healthiness conditions.

This work is feeding into research programs that are exploring state-rich concurrent processes via the Circus language [9] and the addition of “time-slots” [4], and will be used to inform the healthiness conditions of those frameworks, as well as the semantics of the various language constructs.

However, it is also the case that this work, as well as that done on the “slotted”-theory [4], is showing up the need for a generic framework of meta-theory surrounding the notion of the interlinking of theories. Whilst the concept and basic meta-theory of linking is discussed in depth in [6, Chp. 4] and [22], [23] there is still a need for further results in this area.
Acknowledgement. We would like to thank Science Foundation Ireland (SFI), for their support of this work, and Ana Cavalcanti and Leo Freitas for many fruitful and important discussions.

References


