Concurrencey and State in UTP—Choice as Parallelism

Andrew Butterfield¹  Paweł Gancarski¹  Jim Woodcock²

Andrew.Butterfield@cs.tcd.ie

¹Lero@TCD, Trinity College Dublin, Dublin 2, Ireland
²University of York, UK

Dublin Concurrency Workshop, April 14th–15th 2011
ok \land \neg wait
A Simple Imperative Language

\[ u, v \in Var \]
\[ e, c \in Expr \]
\[ p, q, r \in Prog \]

::= 

\[ \text{skip} \] no-op
\[ v := e \] assignment
\[ p; q \] seq. comp.
\[ p \triangleleft c \triangleright q \] conditional
\[ c * p \] while-loop

- \( c \) denotes a \textit{condition}: boolean valued expression over variables
- There are a number of ways to give this a formal semantics.
**Prog Semantics**

- **Denotational Semantics:**

  \[ ρ ∈ \text{State} = \text{Var} → \text{Val} \]
  \[ M_P : \text{Prog} → \text{State} → \text{State} \]
  \[ M_P[x := e]_ρ ≡ ρ ⊕ \{x ↦ M_E[e]_ρ\} \]

  (Programs as State-Transformers)

- **Weakest Pre-Condition Semantics:**

  \[ C, D ∈ \text{Cond} = \text{State} → \mathbb{B} \]
  \[ WP : \text{Prog} → \text{Cond} → \text{Cond} \]
  \[ WP(x := e)D ≡ D[e/x] \]

  (Programs as Predicate Transformers)

- we could go on …
A Simple Concurrent Language

\[ a, b \in \text{Event} \]
\[ p, q, r \in \text{Conc} \]

Events:

- \( \text{stop} \) — deadlock
- \( \text{skip} \) — termination
- \( a \rightarrow p \) — event prefix
- \( p ; q \) — sequence
- \( p \sqcap q \) — non-determinism
- \( p \sqcup q \) — event choice
- \( p \parallel_A q \) — parallel (synch on \( A \))

- Events are atomic
- There are also number of ways to give this a formal semantics.
**Conc** Denotational Semantics (I)

- **Trace Semantics:**

  
  
  \[
  \begin{align*}
  tr \in \text{Trace} & = \text{Event}^* \\
  M_T : \text{Conc} & \rightarrow \mathcal{P}(\text{Trace}) \\
  M_T[\text{stop}] & = \{\langle \rangle \} \\
  M_T[\text{a} \rightarrow \text{p}] & = \{\langle \rangle \} \cup \{tr \searrow \langle \text{a} \rangle | tr \in M_T[\text{p}]\}
  \end{align*}
  \]

- **Failures Semantics**

  
  \[
  \begin{align*}
  \text{ref} \in \text{Refusal} & = \mathcal{P}(\text{Event}) \\
  M_F : \text{Conc} & \rightarrow \mathcal{P}(\text{Trace} \times \text{Refusal}) \\
  M_F[\text{stop}] & = \{\langle \rangle, \text{ref} \} | \text{ref} \subseteq \text{Event} \\
  M_F[\text{a} \rightarrow \text{p}] & = \{\langle \rangle, \text{ref} \} | \text{a} \notin \text{ref}\} \cup \\
  \{tr \searrow \langle (\text{a}, \text{ref}) \rangle | tr \in M_F[\text{p}], \text{ref} \subseteq \text{Event}\}
  \end{align*}
  \]

- Failures-Divergences, Labelled Transition Systems, …
Introducing *Circus*

- *Circus* is a language that combines Z and CSP (a mashup of *Prog* and *Conc*)
- The syntax (of a simple version) is easy:

\[
p, q, r \in \text{Circus} ::= \begin{align*}
\text{skip} & \quad \text{termination} \\
\nu := e & \quad \text{assignment} \\
p; q & \quad \text{sequence} \\
p \triangleleft c \triangleright q & \quad \text{conditional} \\
c \ast p & \quad \text{while-loop} \\
\text{stop} & \quad \text{deadlock} \\
\text{a } \rightarrow p & \quad \text{event prefix} \\
p \sqcap q & \quad \text{non-determinism} \\
p \sqcup q & \quad \text{event choice} \\
p|\{U|A|V\}|q & \quad \text{parallel, } U,V \text{ var-sets}
\end{align*}
\]

- What about the semantics?
Unifying Theories of Programming (UTP)

- UTP is a semantic framework that tries to merge semantic models.
- The approach is to encode them using predicates that characterise relations between before- and after-states.

\[ P(o_1, \ldots, o_n, o'_1, \ldots, o'_n) \]

- \( o_i \) before-value of observation \( o_i \)
- \( o'_i \) after-value of observation \( o_i \)

“Programs (and Processes) as Relational Predicates”.

- Observations consist of program variable values, along with other (auxilliary) variables that capture relevant aspects of behaviour.
We define two key observations:

- **state, state’ : Var → Val**
  program variable state
- **ok, ok’ : □**
  the starting and finishing of the program.

For total correctness, all our predicates have the form:

\[ ok \land P \implies ok' \land Q \] — a.k.a. “Designs”

If started when \( P \) is true, it finishes, ensuring that \( Q \) holds.

We introduce a shorthand: \( P \models Q \).
**Prog** UTP Semantics (II)

\[
\begin{align*}
\text{skip} & \; \equiv \; True \vdash state' = state \\
x := e & \; \equiv \; True \vdash state' = state \upharpoonright \{x \mapsto e\} \\
p; q & \; \equiv \; \exists ok_m, state_m \bullet \\
& \quad \quad \quad \quad p[ok_m, state_m/ok', state'] \\
& \quad \quad \quad \quad \quad \quad \wedge q[ok_m, state_m/ok, state] \\
p < c > q & \; \equiv \; c \wedge p \lor \neg c \wedge q \\
c * p & \; \equiv \; \mu W \bullet p; W < c > skip
\end{align*}
\]

(Programs are (Relational) Predicates)
Refinement

- UTP has been formulated to support refinement
- If \( S \) is a specification, and \( P \) is a program then \( P \) satisfies \( S \) (\( S \sqsubseteq P \)) if every behaviour of \( P \) implies one of \( S \)
- A behaviour of predicate \( Q \) is any assignment of values to both dashed and un-dashed variables that satisfies \( Q \).

\[
S \sqsubseteq P \triangleq [P \Rightarrow S]
\]

Here \([Q]\) denotes the universal closure of \( Q \)

- A consequence of this, given that \( P \cap Q \sqsubseteq P \), is that we have the following definition of non-determinism:

\[
P \cap Q \triangleq P \lor Q
\]
Healthiness Conditions

- The predicate subspace of designs, and other interesting subspaces are characterised by Healthiness Conditions.
- For example, all design predicates satisfy the following laws:

\[ H_1 \quad P = ok \implies P \]
\[ H_2 \quad P = P; (ok \implies ok') \land state' = state \]

- Both of these, and many others, can be captured as stating that a healthy predicate is a fixpoint of an idempotent predicate-transformer, e.g.:

\[ R1(P) \triangleq ok \implies P \quad R1 \circ R1 = R1 \]
Designs, ordered by Refinement, form a Lattice

- **True ⊢ False**
- **True ⊢ x′ = 3**
- **True ⊢ x′ = 6**
- **True ⊢ (x′ = 3 ∨ x′ = 6)**
- **True ⊢ x′ ∈ {1...10}**
- **x′ = state′(x)**
- **False ⊢ True**
What is *Miracle*?

- *Miracle* \(\neg ok\) is the lattice top.
- It refines everything else, hence its name.
- It is clearly infeasible (it can never be started).
- Why do we include it?
  - It simplifies the math (we keep the lattice).
  - We can trap it and similar pathologies with another healthiness condition that it fails.

\[
\begin{align*}
H_4 & \quad P; true = true \\
H_4(\neg ok) & = \neg ok; true \\
& = \exists \ldots m \bullet \neg ok \land true \\
& = \neg ok \\
& \neq true
\end{align*}
\]
Conc UTP Semantics (I—Observations)

- We define four key observations:
  - ok, ok′ : \( \mathbb{B} \)
    capture the absence of livelock.
  - wait, wait′ : \( \text{Bool} \)
    captures that a process may be waiting for an event.
  - tr, tr′ : \( \text{Event}^* \):
    Traces record the events observed to date
  - ref, ref′ : \( \mathcal{P} \text{Event} \)
    contain the events being refused
**Conc** UTP Semantics (II—Healthiness)

\[
\begin{align*}
R1(P) & \equiv P \land tr \leq tr' \\
R2(P) & \equiv \exists s \cdot P[s, s \leftarrow (tr' - tr)/tr, tr'] \\
R3(P) & \equiv II \leftarrow wait \rightarrow P \\
II & \equiv R1(\neg ok) \\
& \lor (ok' \land wait' = wait \land tr' = tr \land ref' = ref) \\
R & \equiv R1 \circ R2 \circ R3 \\
CSP1(P) & \equiv P \lor R1(\neg ok) \\
CSP2(P) & \equiv P; J \\
J & \equiv (ok \Rightarrow ok') \land wait' = wait \land tr' = tr \land ref' = ref \\
CSP & \equiv CSP1 \circ CSP2 \circ R
\end{align*}
\]
A key result

- Assume that \( P \) mentions \( ok, tr, ref, wait, ok', tr', ref', wait' \)
- Consider the predicate space \( CSP \) formed by taking all such \( P \) and forming

\[
R(CSP1(CSP2(P)))
\]

- Assume that \( Q \) and \( R \) only mention \( tr, ref, wait, tr', ref', wait' \)
- Consider the predicate space \( RD \) formed by taking all such \( Q \) and \( R \) and forming

\[
R(Q \vdash R)
\]

- It turns out that \( CSP = RD \)
- In other words, CSP processes are Reactive Designs
**Conc** UTP Semantics (III—Definitions)

\[
\begin{align*}
\text{stop} & \equiv R(\text{True} \vdash \text{wait}' \land \text{tr}' = \text{tr}) \\
\text{skip} & \equiv R(\text{True} \vdash \neg \text{wait}' \land \text{tr}' = \text{tr}) \\
a \rightarrow \text{skip} & \equiv R(\text{True} \vdash \text{tr}' = \text{tr} \land a \not\in \text{ref}') \\
& \hspace{1cm} \triangledown \text{wait}' \triangleright \\
& \hspace{1cm} \text{tr}' = \text{tr} \leftarrow \langle a \rangle \\
a \rightarrow p & \equiv (a \rightarrow \text{skip}; p) \\
p; q & \equiv \exists \text{ok}_m, \text{wait}_m, \text{tr}_m, \text{ref}_m \bullet \\
& \hspace{1cm} p[\text{ok}_m, \text{state}_m, \text{tr}_m, \text{ref}_m/\text{ok}', \text{state}', \text{tr}', \text{ref}'] \\
& \hspace{1cm} \land q[\text{ok}_m, \text{state}_m, \text{tr}_m, \text{ref}_m/\text{ok}, \text{state}, \text{tr}, \text{ref}] \\
p \sqcap q & \equiv p \lor q \\
p \Box q & \equiv (p \land q) \triangledown \text{stop} \triangleright (p \lor q)
\end{align*}
\]

(Processes are (Relational) Predicates)
\textbf{Conc} UTP Semantics (IV—Parallel)

\[ p \parallel_A q \equiv \exists ok_1, wait_1, tr_1, ref_1, ok_2, wait_2, tr_2, ref_2 \bullet \\
\quad p[ok_1, wait_1, tr_1, ref_1 / ok', wait', tr', ref'] \land \\
\quad q[ok_2, wait_2, tr_2, ref_2 / ok', wait', tr', ref'] \land \\
\quad ok' = ok_1 \land ok_2 \\
\quad wait' = wait_1 \lor wait_2 \\
\quad tr' - tr \in (tr_1 - tr) \cap_A (tr_2 - tr) \\
\quad ref' \subseteq ((ref_1 \cup ref_2) \cap A) \cup ((ref_1 \cap ref_2) \setminus A) \]

- We “run” \( p \) and \( q \) together, relabelling their final state. Effectively each runs on its own local copy of the state.
- We merge the outcomes appropriately \((\cap_A \text{ returns the way its trace arguments can be merged if required to synch on } A)\).
We merged the syntax pretty easily, so let's mash the semantics together.

UTP also supports methods to link different theories via a Galois Connection, typically capturing a notion of refinement.

...beyond the scope of this talk
Circus UTP Semantics (I—Observations)

We simply mash the observations together:

- \( \text{ok, ok'} : B \) from \( \text{Prog, Conc} \)
- \( \text{wait, wait'} : B \) from \( \text{Conc} \)
- \( \text{tr, tr'} : \text{Event}^* \) from \( \text{Conc} \)
- \( \text{ref, ref'} : \mathcal{P}\text{Event} \) from \( \text{Conc} \)
- \( \text{state, state'} : \text{Var} \rightarrow \text{Val} \) from \( \text{Prog} \)
**Circus UTP Semantics (II—Healthiness)**

We merge the state observations into *Conc* healthiness

\[
R_1(P) \equiv P \land tr \leq tr' \\
R_2(P) \equiv \exists s \bullet P[s, s \prec (tr' - tr)/tr, tr'] \\
R_3(P) \equiv II \prec wait \triangleright P \\
II \equiv R_1(\neg ok) \\
\lor (ok' \land wait' = wait \land tr' = tr \land ref' = ref \\
\land state' = state) \\
\]

\[
R \equiv R_1 \circ R_2 \circ R_3 \\
CSP_1(P) \equiv P \lor R_1(\neg ok) \\
CSP_2(P) \equiv P; J \\
J \equiv (ok \Rightarrow ok') \land wait' = wait \land tr' = tr \land ref' = ref \\
\land state' = state \\
CSP \equiv CSP_1 \circ CSP_2 \circ R \\
\]

Both *II* and *J* now assert that *state* does not change.
Circus UTP Semantics (III—Definitions)

We just show those definitions that explicitly mention state

\[
\begin{align*}
\text{skip} & \equiv R(\text{True} \vdash \neg \text{wait}' \land tr' = tr \land state' = state) \\
a \rightarrow \text{skip} & \equiv R(\text{True} \vdash state' = state \land (tr' = tr \land a \notin \text{ref}') \\
& \quad \langle \text{wait}' \rangle \\
& \quad tr' = tr \leftarrow \langle a \rangle ) \\
p; q & \equiv \exists \ldots m, state_m \bullet \\
& \quad p[\ldots m, state_m/\ldots', state'] \\
& \quad \land q[\ldots m, state_m/\ldots, state]
\end{align*}
\]
**Circus UTP Semantics (IV—Parallel)**

\[ p[U|A|V]|q \]
\[ \equiv \exists ok_1, wait_1, tr_1, ref_1, state_1 ok_2, wait_2, tr_2, ref_2, state_2 \]
\[ p[ok_1, wait_1, tr_1, ref_1, state_1/ok', wait', tr', ref', state'] \land \\
q[ok_2, wait_2, tr_2, ref_2, state_2/ok', wait', tr', ref', state'] \land \\
ok' = ok_1 \land ok_2 \\
wait' = wait_1 \lor wait_2 \\
tr' - tr \in (tr_1 - tr) \setminus_A (tr_2 - tr) \\
ref' \subseteq ((ref_1 \cup ref_2) \cap A) \cup ((ref_1 \cap ref_2) \setminus A) \\
state' = state \oplus state_1|_U \oplus state_2|_V \\
\]

- We now have to duplicate variable state
- We have to merge variable state changes, but we assume \( U \) and \( V \) are disjoint
stop says nothing about state

The definition of stop is unchanged. It cannot assert that state' = state, or we would lose the following (very useful) CSP law:

\[ p \Box stop = p \]

Curious …
That’s done, now let’s play!

Consider the following *Circus* “program/process”:

$\begin{align*}
&((x := 1; a \rightarrow \text{skip}) \quad \square (x := 2; b \rightarrow \text{skip})) \\
&[x|a, b, d]\quad \text{lhs can modify } x, \text{ synch. on all events}
\end{align*}$

$(d \rightarrow \text{skip})$

- What is its behaviour according to our theory?
- What is/should be the underlying *operational* intuition?
Expanding $x := e; a \to \text{skip}$

The expansion:

$$R(\text{true} \vdash ( (tr' = tr \land a \notin \text{ref'}) \triangleleft \text{wait'} \triangleright (tr' = tr \triangleleft \langle a \rangle)))$$

$$\land state' = state \oplus \{x \mapsto e\}$$

We see what is in effect the conjunction of the assignment and prefix action, suggesting that it might be the same behaviour as $a \to x := e$
Expanding the □

\((x := 1; a \rightarrow \text{skip}) □ (x := 2; b \rightarrow \text{skip})\)

\[= \ R(((\text{true} \vdash \neg \text{wait}') \land \text{CHOOSE}) \lor R1(\neg \text{ok}))\]

- \(R1(\neg \text{ok})\) is “Miracle” — the top of the lattice resulting from the contradiction
- \text{CHOOSE} is final outcome of the choice (a disjunction)
- This process never waits for an event, but insists that the event and choice occur immediately
- There is no empty trace possibility, violating prefix closure.
Adding in $|[x|a, b, d]|d \rightarrow \text{skip}$

The parallel construct requires synchronisation on all events
- Lhs process has traces: $\langle a \rangle, \langle b \rangle$
- Rhs process has traces $\langle \rangle, \langle d \rangle$
- None of these can be merged using $\{a, b, d\}$
- Calculation shows this reduces to $R1(\neg ok)$

We have a theory in which simple pieces put together with standard language operators results in *Miracle*, the (totally infeasible) process that refines any specification.
What should happen?

- Process $x := 1; a \rightarrow \text{skip}$ will assign 1 to $x$, wait for and participate in event $a$ and then terminate.

- The behaviour of the external choice should be to run both arms in parallel on local copies of the state, until an external event resolves the choice. Then the losing arm and its state changes are discarded.
  - In other words a multiple-event waiting point, needs a thread with local state copied, per event, and once an awaited event occurs, it kills the un-satisfied threads (occam actually did this!)

- Our problem arises because we treat these local state copies as visible.
What should happen? (cont.)

- The parallel composition puts a process that does \( c \) with one that does either \( a \) or \( b \), with full synchronisation, so it should deadlock.

- Our theory should predict:

\[
(((x := 1; a \rightarrow \text{skip}) \square (x := 2; b \rightarrow \text{skip}))
\left[ [x | \{a, b, d\}] \right] (d \rightarrow \text{skip})
\]

\[=
\text{stop}
\]

As \textit{stop} always \textit{w}ait\textit{s}, the value of \( x \) is not visible.
Fixing the theory

- Key idea:
  Program variable state is not visible while waiting for external events.

- We say a predicate is “boxed” if $\text{state}'$ is arbitrary (hidden):

  $$P \equiv \exists \text{state}' \cdot P$$

- We modify an existing healthiness condition and add a new one:

  $$\text{R3h}(P) \equiv \lll < \text{wait} > P$$
  $$\text{CSP4}(P) \equiv P; \text{skip}$$

- All other definitions remain unchanged.
Where do we put the hard stuff?

- Mixing variables and concurrency is tricky, as this example shows.
- We could expose the “user” to it (make leading assignments illegal in external choice).
- We could have laws with lots of side-conditions: $P \Box \text{stop} = P$ provided “mumble state mumble …”
- Or we can adopt our preferred approach — try to hide it (bury?) in the foundations.
  - Providing an reasoning algebra that works at the programming language level.
The really hard stuff (UTP@TCD)

- slotted-\textit{Circus}: adding synchronous clocks to \textit{Circus}
  original application: hardware compilation
  replace $tr,ref$ by $slots : (Hist \times Ref)^+$

- Adding prioritised choice to slotted-\textit{Circus} (Paweł Gancarski)
  also targeting hardware compilation
  now seen as a way to model wireless sensor networks

- Added probability to Designs, CSP, \textit{Circus}, slotted-\textit{Circus}
  (Riccardo Bresciani)
  replace $ok, state$ by $distr : State \rightarrow [0, 1]$
  early days yet …

- Linkages between \textit{Circus} and CSP (Arshad Beg)
  linking variable-based and parametric-based state manipulation
ok' \land wait' \land questions \notin ref'
\( \text{ok}' \land \neg \text{wait}' \)