Concurrency and State in UTP — Choice as Parallelism

Andrew Butterfield[1] Paweł Gancarski[1] Jim Woodcock[2]

Andrew.Butterfield@cs.tcd.ie

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

Dublin Concurrency Workshop, April 14th–15th 2011

$$ok \land \neg\, wait$$

A Simple Imperative Language

$$
\begin{array}{llll}
u, v \in \mathit{Var} & & & \text{variables} \\
e, c \in \mathit{Expr} & & & \text{expressions} \\
p, q, r \in \mathit{Prog} & ::= & & \\
 & & \mathit{skip} & \text{no-op} \\
 & | & v := e & \text{assignment} \\
 & | & p ; q & \text{seq. comp.} \\
 & | & p \lhd c \rhd q & \text{conditional} \\
 & | & c * p & \text{while-loop}
\end{array}
$$

- c denotes a *condition*: boolean valued expression over variables
- There are a number of ways to give this a formal semantics.

Prog Semantics

- Denotational Semantics:

$$\begin{array}{rcl} \rho \in \mathit{State} & = & \mathit{Var} \to \mathit{Val} \\ M_P & : & \mathit{Prog} \to \mathit{State} \to \mathit{State} \\ M_P[\![x := e]\!]\rho & \widehat{=} & \rho \oplus \{x \mapsto M_E[\![e]\!]\rho\} \end{array}$$

  (Programs as State-Transformers)

- Weakest Pre-Condition Semantics:

$$\begin{array}{rcl} C, D \in \mathit{Cond} & = & \mathit{State} \to \mathbb{B} \\ \mathit{WP} & : & \mathit{Prog} \to \mathit{Cond} \to \mathit{Cond} \\ \mathit{WP}(x := e)D & \widehat{=} & D[e/x] \end{array}$$

  (Programs as Predicate Transformers)

- we could go on …

A Simple Concurrent Language

$a, b \in Event$			Events
$p, q, r \in Conc$	$::=$		
		$stop$	deadlock
	$\mid$	$skip$	termination
	$\mid$	$a \rightarrow p$	event prefix
	$\mid$	$p; q$	sequence
	$\mid$	$p \sqcap q$	non-determinism
	$\mid$	$p \Box q$	event choice
	$\mid$	$p \parallel_A q$	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:

$$
\begin{array}{rcl}
tr \in Trace & = & Event^* \\
M_T & : & Conc \rightarrow \mathcal{P}\,Trace \\
M_T[\![stop]\!] & \widehat{=} & \{\langle\rangle\} \\
M_T[\![a \rightarrow p]\!] & \widehat{=} & \{\langle\rangle\} \cup \{tr \frown \langle a \rangle | tr \in M_T[\![p]\!]\}
\end{array}
$$

- Failures Semantics

$$
\begin{array}{rcl}
ref \in Refusal & = & \mathcal{P}\,Event \\
M_F & : & Conc \rightarrow \mathcal{P}(Trace \times Refusal) \\
M_F[\![stop]\!] & \widehat{=} & \{(\langle\rangle, ref) \mid ref \subseteq Event\} \\
M_F[\![a \rightarrow p]\!] & \widehat{=} & \{(\langle\rangle, ref) \mid a \notin ref\} \cup \\
 & & \{tr \frown \langle(a, ref)\rangle \mid tr \in M_F[\![p]\!], ref \subseteq Event\}
\end{array}
$$

- 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:

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

- 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.

Prog UTP Semantics (I)

- We define two key observations:
  - $state, state' : Var \rightarrow Val$
    program variable state
  - $ok, ok' : \mathbb{B}$
    the starting and finishing of the program.
- For total correctness, all our predicates have the form:
  $ok \land P \Rightarrow ok' \land Q$ — a.k.a. "Designs"
  If started when P is true, it finishes, ensuring that Q holds.
  We introduce a shorthand: $P \vdash Q$.

Prog UTP Semantics (II)

$$
\begin{array}{rcl}
\mathit{skip} & \widehat{=} & \mathit{True} \vdash \mathit{state}' = \mathit{state} \\
x := e & \widehat{=} & \mathit{True} \vdash \mathit{state}' = \mathit{state} \dagger \{x \mapsto e\} \\
p ; q & \widehat{=} & \exists \mathit{ok}_m, \mathit{state}_m \bullet \\
 & & \quad p[\mathit{ok}_m, \mathit{state}_m / \mathit{ok}', \mathit{state}'] \\
 & & \quad \wedge\ q[\mathit{ok}_m, \mathit{state}_m / \mathit{ok}, \mathit{state}] \\
p \lhd c \rhd q & \widehat{=} & c \wedge p \vee \neg c \wedge q \\
c * p & \widehat{=} & \mu W \bullet p ; W \lhd c \rhd \mathit{skip}
\end{array}
$$

(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.
- $$S \sqsubseteq P \quad \widehat{=} \quad [P \Rightarrow S]$$

  Here $[Q]$ denotes the universal closure of Q
- A consequence of this, given that $P \sqcap Q \sqsubseteq P$, is that we have the following definition of non-determinism: $P \sqcap Q \;\widehat{=}\; P \vee Q$

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:

$$\begin{array}{ll} \mathbf{H}1 & P = ok \Rightarrow P \\ \mathbf{H}2 & P = P ; (ok \Rightarrow ok') \wedge state' = state \end{array}$$

- 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.:

$$\mathbf{R1}(P) \mathrel{\widehat{=}} ok \Rightarrow P \qquad \mathbf{R1} \circ \mathbf{R1} = \mathbf{R1}$$

Designs, ordered by Refinement, form a Lattice

$$
\begin{tikzcd}
 & \mathit{True} \vdash \mathit{False} & & \text{Miracle} \\
\mathit{True} \vdash x'=3 \arrow[ur] & & \mathit{True} \vdash x'=6 \arrow[ul] & \\
 & \mathit{True} \vdash (x'=3 \vee x'=6) \arrow[ul] \arrow[ur] & & \\
 & \mathit{True} \vdash x' \in \{1 \ldots 10\} \arrow[u] & & \\
x'=\mathit{state}'(x) & \mathit{False} \vdash \mathit{True} \arrow[u] & & \text{Chaos} \arrow[uuuu, "\sqsubseteq"]
\end{tikzcd}
$$

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

$$\mathbf{H}4 \qquad P ; true = true$$

$$\begin{array}{ll} & \mathbf{H}4(\neg\ ok) \\ = & \neg\ ok ; true \\ = & \exists \ldots_m \bullet \neg\ ok \wedge true \\ = & \neg\ ok \\ \neq & true \end{array}$$

Conc UTP Semantics (I—Observations)

- We define four key observations:
  - $ok, ok' : \mathbb{B}$
    capture the absence of livelock.
  - $wait, wait' : Bool$
    captures that a process may be waiting for an event.
  - $tr, tr' : Event*$:
    Traces record the events observed to date
  - $ref, ref' : \mathcal{P} Event$
    contain the events being refused

Conc UTP Semantics (II—Healthiness)

$$
\begin{array}{rcl}
\mathbf{R1}(P) & \widehat{=} & P \land tr \leqslant tr' \\
\mathbf{R2}(P) & \widehat{=} & \exists s \bullet P[s, s \frown (tr' - tr)/tr, tr'] \\
\mathbf{R3}(P) & \widehat{=} & II \lhd wait \rhd P \\
II & \widehat{=} & \mathbf{R1}(\neg ok) \\
 & & \lor (ok' \land wait' = wait \land tr' = tr \land ref' = ref) \\
\mathbf{R} & \widehat{=} & \mathbf{R1} \circ \mathbf{R2} \circ \mathbf{R3} \\
\mathbf{CSP1}(P) & \widehat{=} & P \lor \mathbf{R1}(\neg ok) \\
\mathbf{CSP2}(P) & \widehat{=} & P ; J \\
J & \widehat{=} & (ok \Rightarrow ok') \land wait' = wait \land tr' = tr \land ref' = ref \\
\mathbf{CSP} & \widehat{=} & \mathbf{CSP1} \circ \mathbf{CSP2} \circ \mathbf{R}
\end{array}
$$

A key result

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

$$\mathbf{R}(\mathbf{CSP1}(\mathbf{CSP2}(P)))$$

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

$$\mathbf{R}(Q \vdash R)$$

- It turns out that $\mathcal{CSP} = \mathcal{RD}$
- In other words, CSP processes are Reactive Designs

Conc UTP Semantics (III—Definitions)

$$
\begin{array}{rcl}
stop & \widehat{=} & \mathbf{R}(True \vdash wait' \land tr' = tr) \\
skip & \widehat{=} & \mathbf{R}(True \vdash \neg\, wait' \land tr' = tr) \\
a \rightarrow skip & \widehat{=} & \mathbf{R}(True \vdash tr' = tr \land a \notin ref' \\
 & & \qquad \lhd\, wait' \rhd \\
 & & \qquad tr' = tr \frown \langle a \rangle\ ) \\
a \rightarrow p & \widehat{=} & (a \rightarrow skip; p) \\
p; q & \widehat{=} & \exists ok_m, wait_m, tr_m, ref_m \bullet \\
 & & \quad p[ok_m, state_m, tr_m, ref_m / ok', state', tr', ref'] \\
 & & \quad \land\, q[ok_m, state_m, tr_m, ref_m / ok, state, tr, ref] \\
p \sqcap q & \widehat{=} & p \lor q \\
p \,\Box\, q & \widehat{=} & (p \land q) \lhd stop \rhd (p \lor q)
\end{array}
$$

(Processes are (Relational) Predicates)

Conc UTP Semantics (IV—Parallel)

$$
\begin{array}{lll}
p \parallel_A q & \mathrel{\widehat{=}} & \exists ok_1, wait_1, tr_1, ref_1, ok_2, wait_2, tr_2, ref_2 \bullet \\
& & \quad p[ok_1, wait_1, tr_1, ref_1 / ok', wait', tr', ref'] \land \\
& & \quad q[ok_2, wait_2, tr_2, ref_2 / ok', wait', tr', ref'] \land \\
& & \quad ok' = ok_1 \land ok_2 \\
& & \quad wait' = wait_1 \lor wait_2 \\
& & \quad tr' - tr \in (tr_1 - tr) \between_A (tr_2 - tr) \\
& & \quad ref' \subseteq ((ref_1 \cup ref_2) \cap A) \cup ((ref_1 \cap ref_2) \setminus A)
\end{array}
$$

- 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 ($\between_A$ returns the way its trace arguments can be merged if required to synch on A).

Semantic Mashup

- We merged the syntax pretty easily, so lets 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:

$$
\begin{array}{lcll}
ok, ok' & : & \mathbb{B} & \text{from } Prog, Conc \\
wait, wait' & : & \mathbb{B} & \text{from } Conc \\
tr, tr' & : & Event* & \text{from } Conc \\
ref, ref' & : & \mathcal{P}\,Event & \text{from } Conc \\
state, state' & : & Var \to Val & \text{from } Prog
\end{array}
$$

Circus UTP Semantics (II—Healthiness)

We merge the state observations into *Conc* healthiness

$$
\begin{array}{rcl}
\mathbf{R1}(P) & \widehat{=} & P \wedge tr \leqslant tr' \\
\mathbf{R2}(P) & \widehat{=} & \exists s \bullet P[s, s \frown (tr' - tr)/tr, tr'] \\
\mathbf{R3}(P) & \widehat{=} & II \lhd wait \rhd P \\
II & \widehat{=} & \mathbf{R1}(\neg ok) \\
 & & \vee (ok' \wedge wait' = wait \wedge tr' = tr \wedge ref' = ref \\
 & & \quad \wedge state' = state) \\
\mathbf{R} & \widehat{=} & \mathbf{R1} \circ \mathbf{R2} \circ \mathbf{R3} \\
\mathbf{CSP1}(P) & \widehat{=} & P \vee \mathbf{R1}(\neg ok) \\
\mathbf{CSP2}(P) & \widehat{=} & P ; J \\
J & \widehat{=} & (ok \Rightarrow ok') \wedge wait' = wait \wedge tr' = tr \wedge ref' = ref \\
 & & \quad \wedge state' = state \\
\mathbf{CSP} & \widehat{=} & \mathbf{CSP1} \circ \mathbf{CSP2} \circ \mathbf{R}
\end{array}
$$

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

$$
\begin{array}{rcl}
skip & \mathrel{\widehat=} & \mathbf{R}(True \vdash \neg\, wait' \land tr' = tr \land state' = state) \\
a \rightarrow skip & \mathrel{\widehat=} & \mathbf{R}(True \vdash state' = state \land (tr' = tr \land a \notin ref' \\
 & & \qquad\qquad\qquad\qquad \lhd\, wait' \rhd \\
 & & \qquad\qquad\qquad\qquad tr' = tr \frown \langle a \rangle\ )) \\
p ; q & \mathrel{\widehat=} & \exists \ldots_m, state_m \bullet \\
 & & \quad p[\ldots_m, state_m / \ldots', state'] \\
 & & \quad \land\ q[\ldots_m, state_m / \ldots, state]
\end{array}
$$

Circus UTP Semantics (IV—Parallel)

$$
\begin{array}{l}
p |[U | A | V]| q \\
\quad \widehat{=} \quad \exists ok_1, wait_1, tr_1, ref_1, state_1\, ok_2, wait_2, tr_2, ref_2, state_2 \bullet \\
\qquad\qquad p[ok_1, wait_1, tr_1, ref_1, state_1 / ok', wait', tr', ref', state'] \land \\
\qquad\qquad q[ok_2, wait_2, tr_2, ref_2, state_2 / ok', wait', tr', ref', state'] \land \\
\qquad\qquad ok' = ok_1 \land ok_2 \\
\qquad\qquad wait' = wait_1 \lor wait_2 \\
\qquad\qquad tr' - tr \in (tr_1 - tr) \between_A (tr_2 - tr) \\
\qquad\qquad ref' \subseteq ((ref_1 \cup ref_2) \cap A) \cup ((ref_1 \cap ref_2) \setminus A) \\
\qquad\qquad state' = state \oplus state_1|_U \oplus state_2|_V
\end{array}
$$

- 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:

$$p \mathbin{\square} stop \quad = \quad p$$

Curious …

That's done, now let's play !

Consider the following *Circus* "program/process":

$$((x := 1; a \rightarrow skip) \, \Box \, (x := 2; b \rightarrow skip))$$
$$|[x|a, b, d|]| \qquad \text{lhs can modify } x\text{, synch. on all events}$$
$$(d \rightarrow skip)$$

- What is its behaviour according to our theory ?
- What is/should be the underlying *operational* intuition ?

Expanding $x := e; a \rightarrow skip$

The expansion:

$$
\begin{array}{ll}
\mathbf{R}(true \vdash & (\ (tr' = tr \wedge a \notin ref') \\
& \quad \lhd wait' \rhd \\
& \quad (tr' = tr \frown \langle a \rangle)) \\
& \wedge\ state' = state \oplus \{x \mapsto e\} \\
\quad) &
\end{array}
$$

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

Expanding the $\Box$

$$
\begin{array}{rl}
& (x := 1; a \rightarrow \mathit{skip}) \Box (x := 2; b \rightarrow \mathit{skip}) \\
= & \mathbf{R}((\mathit{true} \vdash \neg\, \mathit{wait}' \wedge \mathit{CHOOSE}) \vee \mathbf{R1}(\neg\, \mathit{ok}))
\end{array}
$$

- $\mathbf{R1}(\neg\, \mathit{ok})$ is “Miracle” — the top of the lattice resulting from the contradiction
- 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 $|[x|a, b, d|]d \rightarrow skip$

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 $\between_{\{a,b,d\}}$
- Calculation shows this reduces to $\mathbf{R1}(\neg\ ok)$

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 $x := 1; a \rightarrow skip$ 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:

$$
\begin{array}{ll}
 & ((x := 1; a \rightarrow skip) \Box (x := 2; b \rightarrow skip)) \\
 & |[x|a, b, d|]| \; (d \rightarrow skip) \\
= & \\
 & stop
\end{array}
$$

As $stop$ always $wait$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 *state′* is arbitrary (hidden):

$$\boxed{P} \;\widehat{=}\; \exists state' \bullet P$$

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

$$\begin{aligned}\mathbf{R3h}(P) &\;\widehat{=}\; \boxed{II} \lhd wait \rhd P \\ \mathbf{CSP4}(P) &\;\widehat{=}\; P ; skip\end{aligned}$$

- 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
  $P \Box \mathit{stop} = P$ 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-*Circus*: adding synchronous clocks to *Circus*
  original application: hardware compilation
  replace tr,ref by $slots : (Hist \times Ref)^+$
- Adding prioritised choice to slotted-*Circus* (Paweł Gancarski)
  also targeting hardware compilation
  now seen as a way to model wireless sensor networks
- Added probability to Designs, CSP, *Circus*, slotted-*Circus* (Riccardo Bresciani)
  replace $ok, state$ by $distr : State \to [0, 1]$
  early days yet …
- Linkages between *Circus* and CSP (Arshad Beg)
  linking variable-based and parametric-based state manipulation

$$ok' \wedge wait' \wedge questions \notin ref'$$

$$ok' \wedge \neg\, wait'$$