A Fast Resolution of Choice between Multiway Synchronisations

Peter Welch (University of Kent)

IFIP WG 2.4, Glasgow (21\textsuperscript{st}. July, 2006)
**Background …**

**Communicating processes** offer a natural and scaleable architecture for computational systems. We have **structure** ("networks-within-networks"), **explicit dependencies** ("visible plumbing" – shared events) and **explicit independencies** ("air-gaps" – no shared events).

**CSP** is a process algebra enabling the formal **specification** of such systems and their formal **refinement** to implementation. Its semantics are sufficiently powerful to capture notions of **deadlock**, **divergence**, **non-determinism**, **multiway synchronisation** and **channel communication**.
To date, programming languages (such as *occam*-π) and libraries (*JCSP*, *CTJ*, *C++CSP*, etc.) offer *CSP* primitives and operators ... *but have always restricted the use of certain combinations*. This has been solely for pragmatic reasons ... *overheads*!
The major constraint concerns the set of events a process may offer (i.e. wait for) and non-deterministically choose between if more than one becomes available.

**Note:** only when all processes registered on an event make an offer to synchronise on that event can that event be chosen by the processes making the offer and all must make the same choice.  

**Resolving this choice is tricky!** Offers may be withdrawn at any time just when you think you’ve got a full set … 😞 😞 😞
Hence, constraint:

If one process is offering \((i.e. \textit{choosing between})\) event \(e\) together with some other events, other processes offering \(e\) must do so only in a \textit{committed} way \((i.e. \textit{not as part of a choice})\).

The choice can then be resolved with a simple handshake.

For \textit{channels}, only the process \textit{on the input side} may offer it as part of a choice. An \textit{outputting} process must commit. 😞

Choice involving \textit{multiway synchronisations} is banned – since symmetry permits no single process the privilege. 😞 😞 😞
This is unfortunate. Free-wheeling choices between any kinds of event are routinely specified when designing in CSP. Those systems must be transformed to meet the constraints.

The (current) transformations resolve these decisions with a manager process for each event, extra channels between those managers and the application processes … and a 2-phase commit protocol … which introduces serious overheads!

With no automated tools, those transformations are error prone and the resulting system is expressed at a lower level that is hard to maintain. Maintenance has to be at the higher level and the transformations always re-applied.
We present a fast resolution of choice between multiway synchronisation (the most general form of CSP event).

It does not involve a 2-phase protocol and its cost is linear in the number of choice events offered by all participants.

A formal proof of correctness (i.e. that the resolution is a traces-failures-divergences refinement of the specified CSP) has not been finished … but there is confidence.

Preliminary bindings have been built into the JCSP library and an experimental (complete re-write) occam-π compiler.
Solution ...

We present a fast resolution of choice between multiway synchronisation (the most general form of CSP event).

This will remove almost all constraints in the direct and efficient realisation of CSP designs as executable code ...

... if our resolution is correct !!!

☺☺☺☺☺ ☺☺ ☺☺ ☺☺ ☺☺ ☺☺
Talk roadmap ...

Simple example (sticky platelets / blood clotting) ...

Resolution oracle ...

occam-$\pi$ language binding ...

Applying the oracle ...

Sticky platelet behaviour and performance ...

End note ...
Sticky Platelet Model

This gives low-level rules for the movement of **activated (sticky) blood platelets** that give rise to a simple “bump-and-stick” model of **clot formation**.

Only a 1-D bloodstream is modelled. The model relies entirely on multiway synchronisation, choice and concurrency.

The example is greatly distilled from richer (3-D) models developed by colleagues **(Steve Schneider, Jim Woodcock, Ana Cavalcanti, Helen Treharne)** in our **TUNA** project, which is developing theories for emergent behaviour with special emphasis on nanite assemblies:

[http://www.cs.york.ac.uk/nature/tuna/](http://www.cs.york.ac.uk/nature/tuna/)
**SYSTEM** is a parallel array of **SITE** processes representing a 1-D bloodstream.

For each **SITE** in the stream with index \(i\), there is an event **pass.\ i** that represents the arrival of an activated platelet. The event **tock** represents the passing of one time unit.

```plaintext
event pass : \{0..n+1\}
event tock
```

Each **SITE (i)** monitors the arrival of a platelet at its own position (**pass.\ i**), its movement to the next one (**pass.\ i+1**), the movement of a platelet from the next position to the next-but-one (**pass.\ i+2**), and time (**tock**).

```plaintext
EVENTS (i) = \{pass.\ i, pass.\ i+1, pass.\ i+2, tock\}
SYSTEM = || i:{0..n-1} @ [EVENTS (i)] SITE (i)
```
event pass : \{0 .. n+1\}

event tock

EVENTS (i) = \{pass.i, pass.i+1, pass.i+2, tock\}

SYSTEM = || i:{0..n-1} @ [EVENTS (i)] SITE (i)
For a platelet to move to (say) position $i+2$, three processes – $\text{SITE } (i+2)$, $\text{SITE } (i+1)$ and $\text{SITE } (i)$ – must approve by offering the event $\text{pass.}i+2$.

$\text{EVENTS } (i) = \{\text{pass.}i, \text{pass.}i+1, \text{pass.}i+2, \text{tock}\}$

$\text{SYSTEM } = \{i : 0..n-1\} @ [\text{EVENTS } (i)] \text{ SITE } (i)$
A site exercises *choice* between subsets of its events (which intersect with sets between which other sites are choosing). Sites have three states. They start off *empty*.

EVENTS (i) = {pass.i, pass.i+1, pass.i+2, tock}

SITE (i) = EMPTY (i)
EVENTS (i) = \{\text{pass}.i, \text{pass}.i+1, \text{pass}.i+2, \text{tock}\}

EMPTY (i) = 
- \text{pass}.i \rightarrow \text{ALMOST} (i) [\]
- \text{pass}.i+2 \rightarrow \text{EMPTY} (i) [\]
- \text{tock} \rightarrow \text{EMPTY} (i)

\text{pass}.i+1 is not allowed from the ‘empty’ state.

An ‘empty’ site has no platelet to move on!
An upstream platelet is allowed to move in. The site becomes ‘almost’ full …
EVENTS (i) = {pass.i, pass.i+1, pass.i+2, tock}

EMPTY (i) =
- pass.i -> ALMOST (i) []
- pass.i+2 -> EMPTY (i) []
- tock -> EMPTY (i)

When 'empty', this does not concern us.

A downstream platelet is allowed to move on.
EVENTS (i) = \{pass.i, pass.i+1, pass.i+2, tock\}

EMPTY (i) =
- pass.i -> ALMOST (i) []
- pass.i+2 -> EMPTY (i) []
- tock -> EMPTY (i)

Time is allowed to move on.
EVENTS (i) = \{pass.i, pass.i+1, pass.i+2, tock\}

ALMOST (i) =
\begin{align*}
\text{pass.i+2} & \rightarrow \text{ALMOST (i)} \[
\text{tock} & \rightarrow \text{FULL (i)}
\end{align*}

A platelet has arrived – this imposes a space limit of one platelet per site.
pass.i-1

pass.i

pass.i+1

pass.i+2

SITE (i-1)

SITE (i)

SITE (i+1)

SITE (i+2)

tock

EVENTS (i) = \{pass.i, pass.i+1, pass.i+2, tock\}

ALMOST (i) =

\[ \text{pass.i+2} \rightarrow \text{ALMOST (i)} \]

\[ \text{tock} \rightarrow \text{FULL (i)} \]

pass.i+1 is not allowed from the ‘almost’ state.

Time has not moved on since the platelet arrived – this imposes a speed limit of one site per tock.
EVENTS (i) = \{\text{pass}.i, \text{pass}.i+1, \text{pass}.i+2, \text{tock}\}

\text{ALMOST} (i) =
\begin{align*}
\text{pass}.i+2 & \rightarrow \text{ALMOST} (i) [] \\
\text{tock} & \rightarrow \text{FULL} (i)
\end{align*}

A downstream platelet is allowed to move on.

It’s moving in the same clock cycle as the one just arrived – so they don’t bump (and stick).
EVENTS (i) = \{\text{pass}.i, \text{pass}.i+1, \text{pass}.i+2, \text{tock}\}

ALMOST (i) = 
\begin{align*}
\text{pass}.i+2 &\rightarrow \text{ALMOST} (i) [\] \\
\text{tock} &\rightarrow \text{FULL} (i)
\end{align*}

Time is allowed to move on.

\text{The site becomes ‘full’}.
EVENTS (i) = \{\text{pass} . i, \text{pass} . i+1, \text{pass} . i+2, \text{tock}\}

\text{FULL} (i) =
\begin{align*}
\text{pass} . i+1 \rightarrow & \text{EMPTY} (i) \, [], \\
\text{pass} . i+2 \rightarrow & \text{pass} . i+1 \rightarrow \text{EMPTY} (i) \\
\text{tock} \rightarrow & \text{FULL} (i)
\end{align*}

The site is full – this maintains the space limit of one platelet per site.
\[ \text{EVENTS} (i) = \{ \text{pass}.i, \text{pass}.i+1, \text{pass}.i+2, \text{tock} \} \]

\[ \text{FULL} (i) = \]
- \[ \text{pass}.i+1 \rightarrow \text{EMPTY} (i) [ ] \]
- \[ \text{pass}.i+2 \rightarrow \text{pass}.i+1 \rightarrow \text{EMPTY} (i) \]
- \[ \text{tock} \rightarrow \text{FULL} (i) \]

Time has passed - our platelet may move on.
EVENTS (i) = \{\text{pass}.i, \text{pass}.i+1, \text{pass}.i+2, \text{tock}\}

\begin{align*}
\text{FULL} (i) = \\
\text{pass}.i+1 & \rightarrow \text{EMPTY} (i) [] \\
\text{pass}.i+2 & \rightarrow \text{pass}.i+1 \rightarrow \text{EMPTY} (i) \\
\text{tock} & \rightarrow \text{FULL} (i)
\end{align*}
EVENTS (i) = {pass.i, pass.i+1, pass.i+2, tock}

FULL (i) =
- pass.i+1 -> EMPTY (i) []
- pass.i+2 -> pass.i+1 -> EMPTY (i)
- tock -> FULL (i)

If that downstream platelet moves, it drags our platelet with it – they’re stuck together.
| Simple example (sticky platelets / blood clotting) ... |
| Resolution oracle ... |
| **occam-π** language binding ... |
| Applying the oracle ... |
| Sticky platelet behaviour and performance ... |
| End note ... |
Let $P(0) \ldots P(n-1)$ be application processes that, from time to time, need to make a choice between multiway syncs.

Abandon distributed decision making!

Concentrate decisions in one sequential process – the oracle.
The \textbf{oracle} waits for offers on its \textbf{ask} channel. Application processes, \textbf{P(0)} \ldots \textbf{P(n-1)}, queue up to make offers to the (shared writing end) of the \textbf{ask} channel. An offer consists of the \textbf{id} of the offering process and the set of events offered. After making an offer, a supplicant waits for the decision on its (exclusive) \textbf{ans} channel. The answer is which of the set of events offered has been chosen.
Application processes are indexed 0…n-1. Index the events on which they engage 0…m-1.

Each event has a set of application processes that may make offers on it. The oracle has this information.

An event can fire only when a complete set of offers has been made. For each event, the oracle counts down have many have still to be made.

For each application process, the oracle records any offer made – until it reports its decision.
Inside the Oracle

<table>
<thead>
<tr>
<th>events</th>
<th>processes</th>
<th>offers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0, 1</td>
<td>3, m-1</td>
</tr>
<tr>
<td>2</td>
<td>0, 1, 2</td>
<td>3, 4, m-1</td>
</tr>
<tr>
<td>3</td>
<td>1, 2, 3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2, 3, 4</td>
<td></td>
</tr>
<tr>
<td>m-1</td>
<td>0, 1, ..., n-1</td>
<td>n-1</td>
</tr>
</tbody>
</table>

counts

ask

4, [4, 6, m-1]

ans.0
ans.1
...
ans.n-1
Inside the Oracle

<table>
<thead>
<tr>
<th>events</th>
<th>counts</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0, 1</td>
</tr>
<tr>
<td>2</td>
<td>0, 1, 2</td>
</tr>
<tr>
<td>3</td>
<td>1, 2, 3</td>
</tr>
<tr>
<td>4</td>
<td>2, 3, 4</td>
</tr>
<tr>
<td>m-1</td>
<td>0, 1, ..., n-1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>processes</th>
<th>offers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3, m-1</td>
</tr>
<tr>
<td>1</td>
<td>3, 4, m-1</td>
</tr>
<tr>
<td>2</td>
<td>3, 4, m-1</td>
</tr>
<tr>
<td>3</td>
<td>4, 6, m-1</td>
</tr>
<tr>
<td>4</td>
<td>n-1</td>
</tr>
</tbody>
</table>

3, [3, 5, m-1]
Inside the Oracle

<table>
<thead>
<tr>
<th>events</th>
<th>counts</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0, 1</td>
</tr>
<tr>
<td>2</td>
<td>0, 1, 2</td>
</tr>
<tr>
<td>3</td>
<td>1, 2, 3</td>
</tr>
<tr>
<td>4</td>
<td>2, 3, 4</td>
</tr>
<tr>
<td>m-1</td>
<td>0, 1, ..., n-1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>processes</th>
<th>offers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3, m-1</td>
</tr>
<tr>
<td>1</td>
<td>3, 4, m-1</td>
</tr>
<tr>
<td>2</td>
<td>3, 5, m-1</td>
</tr>
<tr>
<td>3</td>
<td>4, 6, m-1</td>
</tr>
<tr>
<td>n-1</td>
<td></td>
</tr>
</tbody>
</table>
## Inside the Oracle

### Chart

#### Events

<table>
<thead>
<tr>
<th>Event</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>m-1</td>
<td></td>
</tr>
<tr>
<td>0, 1, ..., n-1</td>
<td>40</td>
</tr>
</tbody>
</table>

#### Processes

<table>
<thead>
<tr>
<th>Process</th>
<th>Offers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3, m-1</td>
</tr>
<tr>
<td>1</td>
<td>3, 4, m-1</td>
</tr>
<tr>
<td>2</td>
<td>3, 5, m-1</td>
</tr>
<tr>
<td>3</td>
<td>4, 6, m-1</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>n-1</td>
<td></td>
</tr>
</tbody>
</table>

#### Counting

- `ask` to `processes`
- `processes` to `counts`
- `counts` to `events`
- `events` to `processes`
- `processes` to `offers`

### Notes

- The chart illustrates how events are processed and counted, leading to offers being generated.
- The `ask` mechanism is used to query the `processes` table and retrieve relevant data.
- The `events` and `processes` tables are interconnected, allowing for a dynamic flow of data.
- The `counts` table keeps track of the progress and is updated accordingly.
- The `offers` table outputs the final results based on the processed data.
Inside the Oracle

<table>
<thead>
<tr>
<th>events</th>
<th>processes</th>
<th>offers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0, m-1</td>
</tr>
<tr>
<td>1</td>
<td>0, 1</td>
<td>3, m-1</td>
</tr>
<tr>
<td>2</td>
<td>0, 1, 2</td>
<td>3, 4, m-1</td>
</tr>
<tr>
<td>3</td>
<td>1, 2, 3</td>
<td>3, 5, m-1</td>
</tr>
<tr>
<td>4</td>
<td>2, 3, 4</td>
<td>4, 6, m-1</td>
</tr>
<tr>
<td>m-1</td>
<td>0, 1, ..., n-1</td>
<td>n-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>processes</th>
<th>counts</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>offers</th>
<th>counts</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>40</td>
</tr>
</tbody>
</table>

ask

ans.0

ans.1

... ans.n-1
### Inside the Oracle

#### Processes

<table>
<thead>
<tr>
<th>Events</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>m-1</td>
<td>n-1</td>
</tr>
</tbody>
</table>

#### Offers

<table>
<thead>
<tr>
<th>Processes</th>
<th>Offers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3, 4, m-1</td>
</tr>
<tr>
<td>3</td>
<td>3, 5, m-1</td>
</tr>
<tr>
<td>4</td>
<td>4, 6, m-1</td>
</tr>
<tr>
<td>n-1</td>
<td></td>
</tr>
</tbody>
</table>

#### Events

<table>
<thead>
<tr>
<th>Events</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>ans.0</td>
<td>3</td>
</tr>
<tr>
<td>ans.1</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>ans.n-1</td>
<td></td>
</tr>
</tbody>
</table>
Inside the Oracle

<table>
<thead>
<tr>
<th>events</th>
<th>processes</th>
<th>offers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0, 1</td>
<td>0, 1, 2</td>
<td>4, 6, m-1</td>
</tr>
<tr>
<td>1, 2, 3</td>
<td>2, 3, 4</td>
<td></td>
</tr>
<tr>
<td>0, 1, ..., n-1</td>
<td>0, 1, ..., n-1</td>
<td>n-1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>counts</th>
<th>offers</th>
<th>oracle</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>offers</th>
<th>oracle</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Inside the Oracle

<table>
<thead>
<tr>
<th>events</th>
<th>processes</th>
<th>offers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0, 1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0, 1, 2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>1, 2, 3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>2, 3, 4</td>
<td>2</td>
</tr>
<tr>
<td>m-1</td>
<td>0, 1, ..., n-1</td>
<td>43</td>
</tr>
</tbody>
</table>

ask

<table>
<thead>
<tr>
<th>processes</th>
<th>offers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4, 6, m-1</td>
</tr>
</tbody>
</table>

oracle

ans.0

ans.1

ans..n-1

3
Inside the Oracle

processes = < 0 .. (n-1) >
events = < 0 .. (m-1) >

enrolled (i, j) = ...

"process i is enrolled on event j"

customers (j) = < i <- processes, enrolled (i, j) >
alphabet (i) = < j <- events, enrolled (i, j) >
Inside the Oracle

\[ \text{processes} = \langle 0 .. (n-1) \rangle \]
\[ \text{events} = \langle 0 .. (m-1) \rangle \]

\[ \text{event ask} : \{0..n-1\}.Set\{0..m-1\} \]

Who is asking
(process id)

Offer
(set of event ids)
Inside the Oracle

processes = < 0 .. (n-1) >
events = < 0 .. (m-1) >

event ans : {0..n-1}.{0..m-1}

who was asking (process id)
chosen event
Inside the Oracle

\[
\text{oracle} = \\
\text{let} \\
\quad \text{initial_counts} = \langle \#\text{customers} (j) \mid j \leftarrow \text{events} \rangle \\
\quad \text{initial_offers} = \langle \{} \mid i \leftarrow \text{processes} \rangle \\
\text{within} \\
\quad \text{ORACLE} (\text{initial_counts}, \text{initial_offers})
\]
Inside the Oracle

ORACLE \((\text{counts, offers}) = \)

\(\text{ask?app?offer} \rightarrow\)

let

\(\text{new_counts} = \text{decrement } (\text{counts, offer})\)
\(\text{new_offers} = \text{update } (\text{offers, app, offer})\)
\(\text{zero_counts} = \{j \leftarrow \text{offer}, \text{new_counts}[j] = 0\}\)

within

if \(\#\text{zero_counts} = 0\) then

\(\text{ORACLE } (\text{new_counts, new_offers})\)

else

\(\sim\) chosen:zero_counts @

\(\text{ORA } (\text{chosen, new_counts, new_offers})\)
Inside the Oracle

```
ORA (chosen, counts, offers) =
  let
    releases = customers (chosen)
    (restored_counts, restored_offers) =
      restore (releases, counts, offers)
  within
    ( ||| i:releases @ ans.i!chosen ~> SKIP );
  ORACLE (restored_counts, restored_offers)
```

counts[chosen] = 0

one of these was the triggering ‘app’

```
skip
ans.0
ans.1
ans.n-1
```

ask
Inside the Oracle

```
estore (<i>^rest, counts, offers) =
let
  up_counts = increment (counts, offers[i])
  down_offers = update (offers, i, { })
within
  restore (rest, up_counts, down_offers)
estore (<>, counts, offers) = (counts, offers)
```

- only those currently unanswered
- all greater than zero
- one of these was the triggering ‘app’
- one of these was the triggering ‘offer’
Talk roadmap ...

Simple example (sticky platelets / blood clotting) ...
Resolution oracle ...

cam-π language binding ...
Applying the oracle ...
Sticky platelet behaviour and performance ...
End note ...
occam-π Language Binding

occam-π already supports multiway synchronisation (BARRIER).

Processes must commit to them though. They cannot be offered alongside other events (and, maybe, backed off if another is chosen).

Language design makes process enrollment / resignation automatic and unavoidable. No run-time action is needed to check correct usage. They are fast.

Therefore, we propose a separate synchronisation type (ALT.BARRIER), with the same semantics and safety as BARRIER – except it can be offered alongside other events.

occam-π can then directly code the CSP equations. 😊😊😊
event pass : \{0 .. n+1\}

EVENTS (i) = \{pass.i, pass.i+1, pass.i+2, tock\}

SYSTEM = || i:{0..n-1} @ [EVENTS (i)] SITE (i)
\[\text{[n+2]} \text{ALT.BARRIER pass:}\]
\[\text{ALT.BARRIER tock:}\]

\[\text{PAR } i = 0 \text{ FOR } n\]
\[\text{site } (\text{pass}[i], \text{pass}[i+1], \text{pass}[i+2], \text{tock})\]
PROC site (ALT.BARRIER me, me.1, me.2, tock)

SITE (i) = EMPTY (i)

EMPTY (i) =
  pass.i -> ALMOST (i) []
  pass.i+2 -> EMPTY (i) []
  tock -> EMPTY (i)

ALMOST (i) =
  pass.i+2 -> ALMOST (i) []
  tock -> FULL (i)

FULL (i) =
  pass.i+1 -> EMPTY (i) []
  pass.i+2 -> pass.i+1 -> EMPTY (i)
  tock -> FULL (i)
PROC site (ALT.BARRIER me, me.1, me.2, tock)

VAL INT EMPTY IS 0:
VAL INT ALMOST IS 1:
VAL INT FULL IS 2:

INITIAL INT state IS EMPTY:

SITE (i) = EMPTY (i)

EMPTY (i) =
    pass.i -> ALMOST (i) []
    pass.i+2 -> EMPTY (i) []
    tock -> EMPTY (i)

ALMOST (i) =
    pass.i+2 -> ALMOST (i) []
    tock -> FULL (i)

FULL (i) =
    pass.i+1 -> EMPTY (i) []
    pass.i+2 -> pass.i+1 -> EMPTY (i)
    tock -> FULL (i)
PROC site (ALT.BARRIER me, me.1, me.2, tock)

VAL INT EMPTY IS 0:
VAL INT ALMOST IS 1:
VAL INT FULL IS 2:

INITIAL INT state IS EMPTY:

WHILE TRUE
    CASE state
        ... EMPTY case
        ... ALMOST case
        ... FULL case

SITE (i) = EMPTY (i)
EMPTY (i) =
    pass.i -> ALMOST (i) []
    pass.i+2 -> EMPTY (i) []
    tock -> EMPTY (i)

ALMOST (i) =
    pass.i+2 -> ALMOST (i) []
    tock -> FULL (i)

FULL (i) =
    pass.i+1 -> EMPTY (i) []
    pass.i+2 -> pass.i+1 -> EMPTY (i)
    tock -> FULL (i)
{{{ EMPTY case

EMPTY

ALT

SYNC me
  state := ALMOST

SYNC me.2
  SKIP

SYNC tock
  SKIP

}}}

EMPTY (i) =
  pass.i -> ALMOST (i) []
  pass.i+2 -> EMPTY (i) []
  tock -> EMPTY (i)
almost case

almost

alt

sync me.2

skip

sync tock

state := full

almost (i) =

pass.i+2 -> almost (i) []

tock -> full (i)
{{{ FULL case

FULL

ALT

SYNC me.1

state := EMPTY

SYNC me.2

SEQ

SYNC me.1

state := EMPTY

SYNC tock

SKIP

}}}

FULL (i) =

pass.i+1 -> EMPTY (i) []

pass.i+2 -> pass.i+1 -> EMPTY (i)

tock -> FULL (i)
Talk roadmap ...

Simple example (sticky platelets / blood clotting) ...
Resolution oracle ...
\texttt{occam-\pi} language binding ...
Applying the oracle ...
Sticky platelet behaviour and performance ...
End note ...
Resolution Oracle: \textit{occam-\(\pi\)}

\begin{verbatim}
PROC oracle (MOBILE []ENROLLED enrolled, 
CHAN ORACLE.ASK ask?,
[]CHAN ORACLE.ANS ans!)

: 

PROTOCOL ORACLE.ASK IS INT; OFFER:

PROTOCOL ORACLE.ANS IS INT; OFFER:

who is asking

the chosen event

returned offer

setup care needed
\end{verbatim}
PROC site (ALT.BARRIER me, me.1, me.2, tock)

VAL INT EMPTY IS 0:
VAL INT ALMOST IS 1:
VAL INT FULL IS 2:

INITIAL INT state IS EMPTY:

WHILE TRUE
    CASE state
        ... EMPTY case
        ... ALMOST case
        ... FULL case
        :

SITE (i) = EMPTY (i)
EMPTY (i) =
    pass.i -> ALMOST (i) []
    pass.i+2 -> EMPTY (i) []
    tock -> EMPTY (i)

ALMOST (i) =
    pass.i+2 -> ALMOST (i) []
    tock -> FULL (i)

FULL (i) =
    pass.i+1 -> EMPTY (i) []
    pass.i+2 -> pass.i+1 -> EMPTY (i)
    tock -> FULL (i)
PROC site (VAL INT id, VAL INT me, me.1, me.2, tock, 
SHARED CHAN ORACLE.ASK to.oracle!,
CHAN ORACLE.ANS from.oracle?)

... VAL INT EMPTY, ALMOST, FULL
... INITIAL INT state
... VAL OFFER empty, almost, full, drag

WHILE TRUE
    CASE state
        ... EMPTY case
        ... ALMOST case
        ... FULL case

SITE (i) = EMPTY (i)

EMPTY (i) =
    pass.i -> ALMOST (i) []
    pass.i+2 -> EMPTY (i) []
    tock -> EMPTY (i)

ALMOST (i) =
    pass.i+2 -> ALMOST (i) []
    tock -> FULL (i)

FULL (i) =
    pass.i+1 -> EMPTY (i) []
    pass.i+2 -> pass.i+1 -> EMPTY (i)
    tock -> FULL (i)
{{{ VAL OFFER empty, almost, full, full.a.b
VAL OFFER empty IS [me, me.2, tock]:
VAL OFFER almost IS [me.2, tock]:
VAL OFFER full IS [me.1, me.2, tock]:
VAL OFFER drag IS [me.1]:
}}}

SITE (i) = EMPTY (i)

EMPTY (i) =
  pass.i -> ALMOST (i) []
  pass.i+2 -> EMPTY (i) []
  tock -> EMPTY (i)

ALMOST (i) =
  pass.i+2 -> ALMOST (i) []
  tock -> FULL (i)

FULL (i) =
  pass.i+1 -> EMPTY (i) []
  pass.i+2 -> pass.i+1 -> EMPTY (i)
  tock -> FULL (i)
PROC site (VAL INT id, VAL INT me, me.1, me.2, tock,
              SHARED CHAN ORACLE.ASK to.oracle!,
              CHAN ORACLE.ANS from.oracle?)

... VAL INT EMPTY, ALMOST, FULL
... INITIAL INT state
... VAL OFFER emtpy, almost, full, drag

WHILE TRUE
    CASE state
        ... EMPTY case
        ... ALMOST case
        ... FULL case

SITE (i) = EMPTY (i)
EMPTY (i) =
    pass.i -> ALMOST (i) []
    pass.i+2 -> EMPTY (i) []
    tock -> EMPTY (i)

ALMOST (i) =
    pass.i+2 -> ALMOST (i) []
    tock -> FULL (i)

FULL (i) =
    pass.i+1 -> EMPTY (i) []
    pass.i+2 -> pass.i+1 -> EMPTY (i)
    tock -> FULL (i)
{{{ FULL case
FULL
  ALT
    SYNC me.1
    state := EMPTY
    SYNC me.2
    SEQ
      SYNC me.1
      state := EMPTY
    SYNC tock
    SKIP
}}}

FULL (i) =
  pass.i+1 -> EMPTY (i) []
  pass.i+2 -> pass.i+1 -> EMPTY (i)
  tock -> FULL (i)
{{ FULL case

FULL

INT answer:
SEQ

CLAIM to.oracle!
  to.oracle ! id; full
from.oracle ? answer; full

CASE answer
  ... me.1
  ... me.2
  ... tock

}}}

FULL (i) =
  pass.i+1 -> EMPTY (i) []
  pass.i+2 -> pass.i+1 -> EMPTY (i)
  tock -> FULL (i)

[me.1, me.2, tock]
\begin{equation}
\text{FULL (i) =}
\begin{align*}
\text{pass.i+1 -> EMPTY (i) []} \\
\text{pass.i+2 -> pass.i+1 -> EMPTY (i)} \\
\text{tack -> FULL (i)}
\end{align*}
\end{equation}

\begin{verbatim}
{\{ me.1
me.1
  state := EMPTY
\}}
\end{verbatim}

\begin{verbatim}
{\{ tack
tack
  SKIP
\}}
\end{verbatim}
FULL (i) =
pass.i+1 -> EMPTY (i) []
pass.i+2 -> pass.i+1 -> EMPTY (i)
tock -> FULL (i)

{{{{
me.2
me.2
SEQ
CLAIM to.oracle!
to.oracle ! id; drag
from.oracle ? answer; drag
state := EMPTY
}}}}}
Simple example (sticky platelets / blood clotting) …
Resolution oracle …
\texttt{occam-\pi} language binding …
Applying the oracle …
Sticky platelet behaviour and performance …
End note …
Talk roadmap ...

Simple example (sticky platelets / blood clotting) ...
Resolution oracle …
\textit{occam-\pi} language binding …
Applying the oracle …
Sticky platelet behaviour and performance …
End note …
Status and Future

We have shown a fast resolution of choice between multiway synchronisation (the most general form of CSP event).

It does not involve a 2-phase protocol and its cost is linear in the number of choice events offered by all participants.

A formal proof of correctness (i.e. that the resolution is a traces-failures-divergences refinement of the specified CSP) has not been finished … but there is confidence.

Preliminary bindings have been built into the JCSP (1.0-rc6) library and an experimental (complete re-write) occam-π compiler.

If correct, this removes almost all constraints in the direct and efficient realisation of CSP designs as executable code …
CSP and occam-π are getting closer together

Unstructured interleaving, ...

Priorities, timeouts, ...

All CSP primitives with fast implementation – SEQ, PAR, channels, multiway synchronisation, ALT (on anything), SHARED channels, recursion, mobile channels, mobile processes, mobile barriers, ...
**CSP** and **occam-π** are getting closer together

- Events (multiway synchronisation) and event guards (including output guards), unstructured interleaving, …
- Mobiles (data, channel-ends, processes), priorities, timeouts, …
- All **CSP** primitives with fast implementation – **SEQ**, **PAR**, channels, **ALT** (input guards only), **SHARED** channels, recursion, …