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 ocaml-π) and libraries (JCSP, CTJ, C++CSP, etc.) offer CSP primitives and operators … but have always restricted the use of certain combinations.

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 … 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 Offers may be withdrawn at any time … just when you think you’ve got a full set …

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.

The example is greatly distilled from richer (3-D) models which is developing theories for emergent behaviour with special emphasis on nanite assemblies: Ana Cavalcanti, Helen Treharne (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/

The example is greatly distilled from richer (3-D) models developed by colleagues (Steve Schneider, Jim Woodcock) 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/

The event pass represents the arrival of an activated platelet. The event tock represents the passing of one time unit.

Each SITE (i) monitors the arrival of a platelet at its own position (pass), 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).

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

SYSTEM = || i:{0..n} & EVENTS (i) @ SITE(i)

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.

Resolution oracle ... occam-π language binding ... Applying the oracle ... Sticky platelet behaviour and performance ... End note ...
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 $\text{pass}.i+2$.

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.

An 'empty' site has no platelet to move on!

A downstream platelet is allowed to move in.

An upstream platelet is allowed to move in.

When 'empty', this does not concern us.
A platelet has arrived – this imposes a space limit of one platelet per site.

It’s moving in the same clock cycle as the one just arrived – so they don’t bump (and stick).

The site becomes ‘full’.

The site is full – this maintains the space limit of one platelet per site.

Time has not moved on since the platelet arrived – this imposes a speed limit of one site per tock.

Time is allowed to move on.

Our platelet may move on.
If that downstream platelet moves, it drags our platelet with it—they're stuck together.

Talk roadmap ...

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

Resolution oracle ...

occam-language binding ...

Applying the oracle ...

Sticky platelet behaviour and performance ...

End note ...

Resolution Oracle

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.

Resolution Oracle

Application processes are indexed \( 0 \ldots n-1 \). Index the events on which they engage \( 0 \ldots n-1 \).

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 how many have still to be made.

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

In the Oracle:

- **processes**: \(< 0 \ldots (n-1) >
- **events**: \(< 0 \ldots (m-1) >

**enrolled (i, j) = ...**

- **customers (j)** = \(< i \leftarrow \text{processes}, \text{enrolled (i, j)} >
- **alphabet (i)** = \(< j \leftarrow \text{events}, \text{enrolled (i, j)} >

**who is asking**

- **process ask**: \(< 0 \ldots (n-1) >
- **set of event ids**: \(< 0 \ldots (m-1) >

**event ask**

- **set (0..m-1)**
- **offer**

Title goes here
Inside the Oracle

processes = < 0 .. (n-1) >
events = < 0 .. (n-1) >
event ans : {0..n-1}.{0..m-1}

who was asking (process id)
chosen event

Inside the Oracle

oracles

ask

Inside the Oracle

oracle

Inside the Oracle

ask

inside

ask

Inside the Oracle

ask

inside

ask
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.

\[
\begin{align*}
\text{PROC site (ALT.BARRIER me, me.1, me.2, tock)} \\
\text{\hspace{1cm} SITE (i-1) = EMPTY (i)} \\
\text{\hspace{1cm} EMPTY (i) =} \\
\text{\hspace{1cm} pass.i \to ALMOST (i) \[]} \\
\text{\hspace{1cm} pass.i \to EMPTY (i) \[]} \\
\text{\hspace{1cm} tock \to EMPTY (i)} \\
\text{\hspace{1cm} ALMOST (i) =} \\
\text{\hspace{1cm} pass.i+2 \to ALMOST (i) \[]} \\
\text{\hspace{1cm} tock \to FULL (i)} \\
\text{\hspace{1cm} FULL (i) =} \\
\text{\hspace{1cm} pass.i+1 \to EMPTY (i) \[]} \\
\text{\hspace{1cm} pass.i+2 \to pass.i+1 \to EMPTY (i)} \\
\text{\hspace{1cm} tock \to FULL (i)} \\
\end{align*}
\]
### Resolution Oracle: occam-π

**Talk roadmap ...**

Simple example (sticky platelets / blood clotting) ...
Resolution oracle ...
occam-π language binding ...
Applying the oracle ...
Sticky platelet behaviour and performance ...
End note ...

### Resolution Oracle:

```
PROC oracle (MOBILE []ENROLLED enrolled, CHAN ORACLE.ASK ask?, [ ]CHAN ORACLE.ANS ans!) ...
...
PROC oracle (MOBILE [])ENROLLED enrolled, CHAN ORACLE.ASK ask?, [ ]CHAN ORACLE.ANS ans!)
...
```

**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
```

**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)
```

**site (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)
```

```
SYNCH ma
state := ALMOST
SYNCH ma.2
SYNCH tock
SKIP
```

```
SYNCH ma.2
SYNCH tock
state := FULL
```

```
SYNCH ma.1
state := EMPTY
SEQ
SYNCH ma.1
state := EMPTY
SYNCH tock
SKIP
```

```
SYNCH ma.1
state := EMPTY
SEQ
SYNCH ma.1
state := EMPTY
SYNCH tock
SKIP
```

```
SYNCH ma
state := ALMOST
SYNCH ma.2
SYNCH tock
SKIP
```

```
SYNCH ma
state := ALMOST
SYNCH ma.2
SYNCH tock
SKIP
```

```
SYNCH ma
state := ALMOST
SYNCH ma.2
SYNCH tock
SKIP
```

```
SYNCH ma
state := ALMOST
SYNCH ma.2
SYNCH tock
SKIP
```

```
SYNCH ma
state := ALMOST
SYNCH ma.2
SYNCH tock
SKIP
```

```
SYNCH ma
state := ALMOST
SYNCH ma.2
SYNCH tock
SKIP
```

```
SYNCH ma
state := ALMOST
SYNCH ma.2
SYNCH tock
SKIP
```

```
SYNCH ma
state := ALMOST
SYNCH ma.2
SYNCH tock
SKIP
```
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

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, …

2003