Kent TUNA Group
Computing Laboratory
University of Kent at Canterbury

TUNA meeting, Surrey (11th April, 2005)
[Revised: 13th June, 2005]

Barriers, Mobiles, Semantics and Platelets
Caution: These slides represent ideas that are very much work-in-progress.

**Aim:** present *occam-\(\pi\)* barrier *synchronisation, forking* and *mobile* barriers and channels.

**Aim:** map all these *occam-\(\pi\)* mechanisms on to CSP, so that we can apply formal reasoning to the design and analysis of *occam-\(\pi\)* systems.

**Aim:** present some *occam-\(\pi\)* *platelet* models, motivating the above.
occam-π (semantics, application)
- Barriers, mobiles, forking + CSP models
- Simple platelet models

occam-π (demonstrations)
- Platelet Model #6 (‘busy’ CA)
- Platelet Model #7 (‘lazy’ CA)
- Graphics (2D above, 3D ‘Game-of-Life’)

13-Jun-05
Copyright P.H.Welch
occam-π (semantics, application)

- Barriers (CSP multiway events)
- Platelet Model #6 (‘busy’ CA)
- Barrier resignation (spinning, (priorities))
- Forking (communication, parallel recursion)
- Platelet model #7 (‘lazy’ CA)
- Mobile barriers (more spinning)
- Channel-ends (design pattern or for real?)
- Mobile channel-ends (tokens, interleaving)
- Shared mobile channel-ends (ditto)
The \texttt{occam-\pi BARRIER} type corresponds to a multiway CSP \texttt{event}, though some higher level design patterns (such as \texttt{resignation}) have been built in.

The \texttt{BARRIER} type declaration \textit{enrolls} the process in its scope.

Basic CSP semantics apply. When a process \textit{synchronises} on a barrier, it blocks until all other processes \textit{enrolled} on the barrier have also \textit{synchronised}. Once the barrier has completed (i.e. all \textit{enrolled} processes have \textit{synchronised}), all blocked processes are rescheduled for execution.
Barriers (static)

A parallel composition may enroll all its processes on a barrier:

\[
\text{PAR ENROLL } b \\
P \\
Q \\
R
\]

\[
|\{b\}| \{P, Q, R\}
\]

The number of the processes enrolled on the barrier increases by one less than the number of components in the enrolling PAR construct (for the duration of the PAR).

\[
\text{PAR } i = 0 \text{ FOR } n \text{ ENROLL } b, c \\
\text{worker (i)}
\]

\[
|\{b, c\}| \{\text{worker (i)} | i = 0..(n-1)\}
\]
Barriers (static)

Normal (non-enrolling) parallel composition interleaves the processes with respect to synchronisation on barriers:

\[
\text{PAR} \quad \parallel \quad \{P, Q, R\}
\]

The number of processes enrolled on barriers does not change for normal parallel composition. Of course, the processes may synchronise with each other using channels (not considered here).

Actually, occam-π enforces a design constraint on the above. At most, only one of the processes in a normal parallel composition is allowed to synchronise on any in-scope barrier.
Barriers (static)

Note: in occam-π, if those processes are PROC instances, we should pass in the barriers as arguments:

```
PAR ENROLL b
P (b)
Q (b)
R (b)
```

```
PAR i = 0 FOR n ENROLL b, c
worker (i, b, c)
```

Though barrier arguments (like channel arguments) are implicit in occam-π network diagrams:
Barriers (static)

Barriers are commonly used to synchronise multiple *phases* of computation between a set of processes. Within each phase, other synchronisations (channel/barrier) may take place:

```
PROC worker (VAL INT id, BARRIER b, c)
  ... local declarations / initialisation
  WHILE running
    SEQ
      SYNC b
      ... phase b computation
      SYNC c
      ... phase c computation
```


Barriers (static)

Often, only one barrier is needed to synchronise these phases:

```proc
PROC worker (VAL INT id, BARRIER b)
  ... local declarations / initialisation
  WHILE running
    SEQ
      SYNC b
      ... phase 0 computation
      SYNC b
      ... phase 1 computation
: 
```
Barriers (static)

\textbf{occam-π BARRIER synchronisation} is fast: around \textbf{15 ns} per sync per process (measured on a \textbf{10,000,000} process benchmark on a \textbf{3.2 GHz. Pentium IV}) – though cache prediction strategies by the Pentium take some of the credit.

\textbf{occam-π BARRIER synchronisation} is safe in the sense that \textbf{enrollment} (and \textbf{resignation}) are automatically managed. A process may \textbf{synchronise} on a \textbf{BARRIER} if and only if it is \textbf{enrolled}.

Try to break this rule … your program won’t compile. 😞

\textbf{BARRIER resignation} comes later. We don’t need it for the first \textbf{platelet} model.
occam-π (semantics, application)

- Barriers (CSP multiway events)
- Platelet Model #6 ('busy' CA)
- Barrier resignation (spinning, priorities)
- Forking (communication, parallel recursion)
- Platelet model #7 ('lazy' CA)
- Mobile barriers (more spinning)
- Channel-ends (design pattern or for real?)
- Mobile channel-ends (tokens, interleaving)
- Shared mobile channel-ends (ditto)
Platelet Model (‘busy’ CA)

**Space** is represented as a pipeline of **cell** processes. **Platelets** move through the **cell**s as data at speeds inversely proportional to the size of the **clump** in which they may become embedded – these speeds are randomised slightly. **Clumps** that bump together stay together.

The **cell**s do all the work and work all the time, even when empty. **Platelets/clumps** pass through them – at which times, the **cell**s compute part of their life-cycle.

**Platelets/clumps** are not directly modelled as processes.
Platelet Model (‘busy’ CA)
Platelet Model (‘busy’ CA)

Key:
- Phase 0
- Phase 1
Platelet Model (‘busy’ CA)

Key:

Phase 0
Platelet Model (‘busy’ CA)

Key:

Phase 1

13-Jun-05 Copyright P.H.Welch
Platelet Model (‘busy’ CA)

PROC cell (BYTE my.visible.state, BOOL running, BARRIER draw, 
CHAN CELL.CELL l.in?, l.out!, r.in?, r.out!)

... local declarations / initialisation (phase 0)
WHILE running
SEQ
SYNC draw -- phase 1
... PAR-I/O exchange of full/empty state
... if full,
... discover clump size (pass count through)
... if head,
... decide on move (non-deterministic choice)
... if move, tell empty cell ahead
... else receive decision on move from cell ahead
... if not tail, pass decision back
... if tail and move, become empty
... else if clump behind exists and moves, become full
SYNC draw -- phase 0
... update my.visible.state

:
Platelet Model (‘busy’ CA)
Platelet Model (‘busy’ CA)

**Performance:** each cell has to work harder if full (carrying a platelet). Also, clump sizes are recomputed every cycle – so large clumps increase the cost. (2.4 GHz. P IV ‘mobile’).

<table>
<thead>
<tr>
<th>Generate probability (n / 256)</th>
<th>Cell cycle time (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>650</td>
</tr>
<tr>
<td>1</td>
<td>660</td>
</tr>
<tr>
<td>2</td>
<td>670</td>
</tr>
<tr>
<td>4</td>
<td>680</td>
</tr>
<tr>
<td>8</td>
<td>700</td>
</tr>
<tr>
<td>16</td>
<td>740</td>
</tr>
<tr>
<td>32</td>
<td>1070 (total jam)</td>
</tr>
</tbody>
</table>
occam-$\pi$ (semantics, application)

- Barriers (CSP multiway events)
- Platelet Model #6 ('busy' CA)
- Barrier resignation (spinning, (priorities))
- Forking (communication, parallel recursion)
- Platelet model #7 ('lazy' CA)
- Mobile barriers (more spinning)
- Channel-ends (design pattern or for real?)
- Mobile channel-ends (tokens, interleaving)
- Shared mobile channel-ends (ditto)
Barrier Resignation

In the previous *platelet* model, all processes engaged in the barrier terminated in the same cycle.

Future models will need the flexibility for engaged processes to drop out of the computation – either permanently (through termination) or temporarily. In which case, we won’t want:

\[
\text{PAR } i = 0 \text{ FOR } n \text{ ENROLL } b \\
\text{P}(i, b)
\]

\[
|\{b\}| \{P(i) \mid i = 0..(n-1)\}
\]

With the above semantics, if any parallel component process terminated early, the others would deadlock in their next cycle as they tried to synchronise on the barrier.
Barrier Resignation

So, let’s define the semantics of a barrier-enrolling parallel to fire off a barrier-synchronising spinner when each process terminates:

\[
\text{ENROLL} \ b \ \left( \text{PAR } i = 0 \text{ FOR } n \ \Rightarrow \ P(i, b) \right)
\]

\[
\sim (\{b, \text{done}\} \downarrow \{P(i); \ Sb \mid i = 0..(n-1)\} \setminus \{\text{done}\})
\]

where:

\[
Sb = (b \rightarrow Sb) [] (\text{done} \rightarrow \text{SKIP})
\]

The trick is to stop them spinning when all are spinning (i.e. all processes have terminated). The above doesn’t quite work.
For example, in the degenerate case when $n = 1$, $P(0, b) = \text{skip}$, and the barrier is local, then:

$$BARRIER \ b:$$
$$\text{PAR} \ i = 0 \ \text{FOR} \ \ n \ \ \text{ENROLL} \ b$$
$$P(i, b)$$

$$(\{b, \text{done}\} \ \{P(i); Sb \mid i = 0..(n-1)\}) \ \{\text{done}, b\}$$

= $$(\{b, \text{done}\} \ \{Sb\}) \ \{\text{done}, b\}$$

= $$Sb \ \{\text{done}, b\}$$

= $$((b \rightarrow Sb) [] (\text{done} \rightarrow \text{skip})) \ \{\text{done}, b\}$$

= $$\text{SPIN} \mid \mid \text{skip}$$

where: $$\text{SPIN} = (e \rightarrow \text{SPIN}) \ \{e\}$$

But we want this to be: $$\text{skip}$$
Barrier Resignation

If CSP extended to *prioritised* choice, then we could define:

\[
\text{PAR}\ i = 0\ \text{FOR}\ n\ \text{ENROLL}\ b
\]

\[
P (i, b)
\]

\[
(|\{b, done}\| {P(i); Sb'} | i = 0..(n-1)\}) \setminus \{done\}
\]

where:

\[
Sb' = (b \rightarrow Sb') [\rightarrow] (\text{done} \rightarrow \text{SKIP})
\]

Now, when all the \( P(i) \) components terminate, we are left with:

\[
(|\{b, done}\| \{Sb' | i = 0..(n-1)\}) \setminus \{done\}
\]

which, with a good semantics for [\rightarrow], should collapse to:

\[
\text{SKIP}
\]
Barrier Resignation

Without *prioritised* choice, we can still do it. If we knew which was the *last* process to terminate, then that could be followed just by the *done* signal rather than the spinner, \( S_b \). We don’t know … but we can add an extra one and make sure that it is the *last* to terminate:

\[
\text{Count} \ (n) = (b \rightarrow \text{Count} \ (n)) \ \[] \\
(\text{down} \rightarrow \text{Count} \ (n - 1)), \quad \text{if } n > 0
\]

\[
\text{Count} \ (0) = (\text{done} \rightarrow \text{SKIP})
\]

So, *done* is refused until *n* counts down to zero – but barrier events, *b*, are accepted. At zero, the barrier is refused and only the *done* signal, terminating all the **spinners**, is engaged.
Barrier Resignation

Then:

\[
\text{PAR } i = 0 \text{ FOR } n \text{ ENROLL } b \\
\quad P(i, b)
\]

\[
( (\{b, \text{done}\} | \{P(i); \text{down} \rightarrow Sb | i = 0..(n-1)\}) \\
\quad |\{b, \text{down, done}\}| \\
\quad \text{Count}(n) \\
) \setminus \{\text{down, done}\}
\]

Now, when all the \(P(i)\) components terminate, the counter has reached zero ... blocking the barrier and issuing \textbf{done}. The \textbf{spinners} have no option but to terminate.
occam-π (semantics, application)

- Barriers (CSP multiway events)
- Platelet Model #6 (‘busy’ CA)
- Barrier resignation (spinning, (priorities))
- Forking (communication, parallel recursion)
- Platelet model #7 (‘lazy’ CA)
- Mobile barriers (more spinning)
- Channel-ends (design pattern or for real?)
- Mobile channel-ends (tokens, interleaving)
- Shared mobile channel-ends (ditto)
Forking Processes

The **PAR** construct creates processes dynamically, but the creating process has to wait for them all to terminate before it can do anything else.

This is not always what we want! Many processes need to be able to **fork** off new processes (whose memory will need to be allocated at run-time) and carry on concurrently with them. Examples include web servers and operating systems … *and nanite assemblies …*

But we are not operating a *free-for-all* heap in our new **occam** – strict aliasing control is maintained even for dynamically allocated structures. So, we must take care about memory referenced by long-lived **forked** processes.
Sometimes, we must fork processes within a fork block.

Process $\texttt{P(...)}$ starts running concurrently with this process.

All forked processes must terminate before a fork block can terminate.

```
INT answer:
SEQ
....
FORKING
SEQ
....
WHILE forking
SEQ
....
....
FORK P (n, answer, svr, cli)
....
....
....
```
INT \texttt{answer:}
SEQ
...
\begin{itemize}
\item \textbf{FORKING}
\item \textbf{SEQ}
\item \textbf{WHILE forking}
\item \textbf{SEQ}
\item \textbf{FORK P} (n, \texttt{answer}, \texttt{svr}, \texttt{cli})
\item ...
\item ...
\item ...
\item ...
\end{itemize}

Otherwise, it may have ceased to exist before the \texttt{FORKed} process terminates

\begin{itemize}
\item VAL data are \textit{copied} into a \texttt{FORKed} process
\item MOBILE data and channel-ends are \textit{moved} into a \texttt{FORKed} process
\item Reference data must be \textit{SHARED} and declared \textit{global} to the \texttt{FORKING} block
\end{itemize}
Forking Processes

INT answer:
SEQ
....
FORKING
SEQ
....
WHILE forking
SEQ
....
....
FORK P (n, answer, svr, cli)
....
....
....

Otherwise, it may have ceased to exist before the FORKed process terminates

i.e. VAL data are communicated to the FORKed process

i.e. MOBILE data / channel-ends are communicated to the FORKed process

Reference data must be SHARED and declared global to the FORKING block

13-Jun-05 Copyright P.H.Welch
Forking Processes

Semantically, *forking* a new process is just a special pattern of parallel recursion:

\[
\text{FORK P (arg)} \quad \rightarrow \\
\text{forkP!arg} \rightarrow \text{SKIP}
\]

where:

\[
\text{ForkP} = \text{forkP?x} \rightarrow (\text{P (x)} | | | \text{ForkP})
\]

*ForkP* runs in parallel with the *Forking* block (i.e. all processes that may choose to *fork* a process *P*):

\[
\text{FORKING X} \quad \rightarrow \\
(X | \{\text{forkP}\} | \text{ForkP}) \setminus \{\text{forkP}\}
\]

and where processes within *Forking* block, *x*, *interleave* over the *forkP* channel.
Forking Processes

We should arrange for the \texttt{ForkP} process to terminate when no longer needed:

\begin{equation}
\text{FORK} \ P \ (\text{arg}) \ \sim \ \text{forkP!arg} \rightarrow \text{SKIP}
\end{equation}

where:

\begin{equation}
\text{ForkP} = \text{forkP?x} \rightarrow (P \ (x) \ || \ || \ \text{ForkP})
\end{equation}

\begin{equation}
[] \ \text{done} \rightarrow \text{SKIP}
\end{equation}

and:

\begin{equation}
\text{ForkING} \ X
\end{equation}

\begin{equation}
((X; \ \text{done} \rightarrow \text{SKIP}) \ | \ \{\text{forkP, done}\} \ | \ \text{ForkP})
\end{equation}

\begin{equation}
\backslash \ \{\text{forkP, done}\}
\end{equation}

where \text{done} is chosen free in \text{P(x)} and \text{x}. 
Forking Processes with Barriers

Our occam-π platelet models will need to fork processes, enrolling them to engage on existing barriers. In general, to pass a barrier to a forked process, we need to be able to communicate it ... i.e. we will need mobile barriers.

For the moment, we restrict attention to the forking of instances of the same process and where all instances are given the same barrier. A static barrier is sufficient for this:

```
BARRIER b:
FORKING
SEQ
  ...
  WHILE forking
    SEQ
      ...
      FORK P (b, arg)
  ...
```

Termination needs care when CSP modelling this – we need to keep count of the number of forked processes still active.
Forking Processes with Barriers

So, run an extended counter in parallel with the Forking block whose forked processes take the barrier:

\[
\text{Count} (n) = (\begin{cases} 
    b \rightarrow \text{Count} (n) & \text{[]} \\
    \text{down} \rightarrow \text{Count} (n - 1) & \\
    \text{up} \rightarrow \text{Count} (n + 1), & \text{if } n > 0 
\end{cases})
\]

\[
\text{Count} (0) = (\text{done} \rightarrow \text{SKIP})
\]

Then:

\[
\text{BARRIER } b: \text{FORKING } X
\]

This terminates ForkPb and all the Sb spinners installed as each forked process and X terminates.

\[
(\text{Count} (1) | \{b, \text{up}, \text{down}, \text{done}\}| \\
(\text{X; down} \rightarrow \text{Sb}) | \{b, \text{forkPb}, \text{done}\}| \text{ForkPb}) \\
\) \{b, \text{forkPb}, \text{up}, \text{down}, \text{done}\}
\]
Forking Processes with Barriers

where:

\[
\text{ForkPb} = (\text{forkPb} ? x \rightarrow \\
( (Pb(x); \text{down} \rightarrow Sb) \mid \{b, \text{done}\} \mid \text{ForkPb} ) \\
) [] \\
(b \rightarrow \text{ForkPb}) [] \\
(done \rightarrow \text{SKIP})
\]

and:

\[
\text{Sb} = (b \rightarrow Sb) [] (\text{done} \rightarrow \text{SKIP})
\]

and where \(P_b(x)\) models \(P(b, x)\).

Inside the **Forking** block \(x\), each **fork** is modelled:

\[
\text{FORK } P(b, \text{arg}) \xrightarrow{up} \text{forkPb}!\text{arg} \rightarrow \text{SKIP}
\]
The above CSP model ensures that, following the *forking* of a process with a barrier:

\[
\text{FORK } P \ (b, \ arg) \quad \text{up} \rightarrow \text{forkPb!arg} \rightarrow \text{SKIP}
\]

the following properties hold:

- subsequent *synchronisation* on the barrier cannot complete without the participation of the *forked* process;
- subsequent *termination* of the *Forking* block cannot complete without the termination of the forked process.
Forking Processes with Barriers

FORKING\(\pi\) view

Copyright P.H.Welch

13-Jun-05
Forking Processes with Barriers

occam-$\pi$ view
Forking Processes with Barriers

CSP model
On the previous slide, the dashed lines represent interleaving by the attached processes. Thus, \( X, \text{ForkPb} \), and the forked \( \text{Pb}' \) processes interleave with each other on events up and down, but synchronise on those events with Count(1).

On the other hand, Count(1), \( X, \text{ForkPb} \), and the forked \( \text{Pb}' \) processes multiway synchronise on the done event (for termination) – and, of course, on the barrier \( b \).

\( \text{Pb}' \) represents a forked process, followed by the down signal introducing the spinner – i.e. \( \text{Pb}(x); \text{down} \rightarrow \text{Sb} \), where \( X \) represents some (non-barrier) argument.
Forking Processes with Barriers

**CSP model**

- **X**
- **ForkPb**
- **Pb'**
- **Pb'**
- **Pb'**
- **Count(1)**
Forking Processes with Barriers

Each Pb', running in parallel with ForkPb, is the remains of an earlier instance of ForkPb.

In this model, the forked Pb' processes cannot directly fork an instance of themselves (although they could ask x to do it on their behalf). Directly self-replicating processes would be nice and are no problem for occam-π.

To allow this, the Pb' processes need to interleave with each other (and with x) to write on the forkPb channel to ForkPb. Since CSP does not distinguish between the writing-end forkPb of a channel (e.g. forkPb!) and the reading-end (e.g. forkPb?), there is a problem. Fortunately, there is a way to model a channel with two CSP-distinct ends – later.
Forking Processes with Barriers

CSP model
Forking Processes with Barriers

Note that the *Forking* block $x$ may safely go parallel and *fork* instances of $P$ (i.e. $P_b'$) in parallel. Such parallel processes would *interleave* with each other to write to *forkPb* and synchronise with *Count(1)* on *up*.

Simple extensions of this model describe passing more than one barrier to a *forked* process (e.g. processes *ForkPbc*, *Sbc* etc.). Also, the *forking* of more than one type of process may be handled (e.g. processes *ForkPc*, *Sc* etc.). For both these extensions, only *one* *Count(1)* is needed (spinning on all managed barriers).

The next *occam-π platelet* model needs to *fork* just *one* process type and pass it a *single* barrier – i.e. we have the CSP model.
A general (and cleaner) solution requires the modelling of mobile barriers: barriers on which processes can be dynamically enrolled through channel communication.

Whenever a barrier is communicated (e.g. to a forked process), the receiving process enrolls and the sending process resigns (in that order). Whenever a barrier variable is overwritten or goes out of scope, the process holding it resigns.
occam-π (semantics, application)

- Barriers (CSP multiway events)
- Platelet Model #6 ('bus{y} CA)
- Barrier resignation (spinning, (priorities))
- Forking (communication, parallel recursion)
- Platelet model #7 ('lazy' CA)
- Mobile barriers (more spinning)
- Channel-ends (design pattern or for real?)
- Mobile channel-ends (tokens, interleaving)
- Shared mobile channel-ends (ditto)
Platelet Model (‘lazy’ CA)
Platelet Model (‘lazy’ CA)
Platelet Model ('lazy' CA)
Platelet Model (‘lazy’ CA)

gen → clot → cell → cell → cell → cell → cell → draw

keywatch → display → screen

keyboard
Platelet Model (‘lazy’ CA)
Platelet Model ('lazy' CA)
Platelet Model (‘lazy’ CA)

gen

clot

clot

cell

cell

cell

cell

cell

cell

cell

cell

cell

cell

keywatch

draw

display

cell

phase 1

screen

keyboard
Platelet Model (‘lazy’ CA)
**Platelet Model (‘lazy’ CA)**

**Performance:** each cell only works when a clump moves through. Run-time depends only on the number of clumps; the clump sizes are now irrelevant (2.4 GHz. P IV ‘mobile’).

<table>
<thead>
<tr>
<th>Generate probability (n / 256)</th>
<th>‘Busy’ (ns)</th>
<th>‘Lazy’ (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>650</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>660</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>670</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>680</td>
<td>14</td>
</tr>
<tr>
<td>8</td>
<td>700</td>
<td>16</td>
</tr>
<tr>
<td>16</td>
<td>740</td>
<td>18</td>
</tr>
<tr>
<td>32</td>
<td>1070 (total jam)</td>
<td>0 (total jam)</td>
</tr>
</tbody>
</table>
occam-π (semantics, application)

- Barriers (CSP multiway events)
- Platelet Model #6 (‘busy’ CA)
- Barrier resignation (spinning, (priorities))
- Forking (communication, parallel recursion)
- Platelet model #7 (‘lazy’ CA)

- Mobile barriers (more spinning)
- Channel-ends (design pattern or for real?)
- Mobile channel-ends (tokens, interleaving)
- Shared mobile channel-ends (ditto)
Barriers (mobile)

`occam-\(\pi\)` includes `mobile` barrier types:

```plaintext
MOBILE BARRIER b:
SEQ
  b := MOBILE BARRIER
  P (b)
```

As shown above, `mobile` barriers are constructed `dynamically` – their variables being initially `undefined`.

Channels may carry `mobile` barriers as components of their messages (`occam-\(\pi\)` `PROTOCOL`s).

Whenever a barrier is communicated (e.g. to a `forked` process), the receiving process dynamically `enrolls` and the sending process `resigns` (in that order). Whenever a barrier variable is overwritten or goes out of scope, the process holding it it `resigns`. 
Barriers (mobile)

For example:

\[
\text{FORK } P (b) \rightarrow \text{up } \rightarrow \text{forkP}!b \rightarrow \text{SKIP}
\]

where:

\[
\text{ForkP} = \text{forkP}?b \rightarrow (P' (b) \parallel \parallel \text{ForkP})
\]

We have to model barrier communication … and the passing of barrier arguments …
The idea is Jim’s. For every barrier, \( b \), that a process may receive, we introduce a proxy spinner process, \( P_{\text{Sb}} \), that synchronises on the barrier whenever its buddy process does not have it. This spinner is pauseable and terminatable:

\[
\text{PSb} = \text{b} \rightarrow \text{PSb} \]

\[
\text{pauseB} \rightarrow \text{spinB} \rightarrow \text{PSb} \]

\[
\text{done} \rightarrow \text{SKIP}
\]

The process, \( P \), runs in parallel with its spinner, \( P_{\text{Sb}} \). They interleave in their synchronisation on \( b \):

\[
(P; \text{done} \rightarrow \text{SKIP}) \ |\{\text{pauseB, spinB, done}\} | P_{\text{Sb}}
\]

\[
\backslash \{\text{pauseB, spinB, done}\}
\]

Of course, the above needs extending to include spinners for as many barriers as may be received (possibly unbounded).
The plan is to \texttt{pause} the spinner \texttt{PSb} whenever \texttt{P} receives the barrier \texttt{b} – and \texttt{spin} it again whenever \texttt{P} loses it.

We say that \texttt{P} is ‘\textit{enrolled}’ on a mobile barrier \texttt{b} \texttt{iff} its \texttt{spinner} is paused.

The process sending \texttt{P} the barrier must already be \textit{enrolled} on that barrier. So, make it wait for an acknowledgment that \texttt{P} has paused its \texttt{spinner}. That prevents synchronisation on \texttt{b}, ensuring that \texttt{P} \textit{can} pause its \texttt{spinner} and that \texttt{P} becomes \textit{enrolled}.

Although \texttt{P} and \texttt{PSb} technically \textit{interleave} on \texttt{b}, only one of them will ever try to synchronise on \texttt{b} at any particular time.
Barriers (mobile)

A mobile barrier $b$ is (represented by) a (mobile) integer index to an infinite set of events:

\[
\{mb.b \mid b = 0, 1, 2, \ldots\}
\]

In parallel with the whole system is a process, $MB(0)$, handing out these indices (i.e. ‘constructing’ mobile barriers):

\[
MB(b) = (getMB!b \rightarrow MB(b+1)) \ [\]
(noMoreMB \rightarrow SKIP)
\]

occam-$\pi$ mobile barriers are just mobile integers indexing $mb$:

\[
b := MOBILE\ BARRIER \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \ quasi-fluid
Barriers (mobile)

Synchronising on a mobile barrier is obvious:

\[
\text{SYNC } b \quad \sim \quad mb.b \rightarrow \text{SKIP}
\]

Any channel carrying mobile barriers must be paired with an acknowledge channel. Sending a barrier is modelled:

\[
c ! b \quad \sim \quad c!b \rightarrow \text{cAck} \rightarrow \text{spin.b} \rightarrow \text{SKIP}
\]

Receiving a barrier is modelled:

\[
c ? b \quad \sim \quad c?\text{tmp} \rightarrow (b := \text{tmp}); \quad \text{pause.b} \rightarrow \text{cAck} \rightarrow \text{SKIP}
\]

where the mobile barrier variable, \(b\), was previously undefined.
Barriers (mobile)

In \texttt{occam-\pi}, it is always known whether a \texttt{mobile} variable is \texttt{defined}. If the receiving variable, \texttt{b}, was previously \texttt{defined}, we must also restart the relevant spinner (of the barrier we are about to lose):

\[
\begin{align*}
\text{c} \ ? \ \text{b} & \sim \\
\text{c?tmp} & \rightarrow \ \text{spin.b} & \rightarrow & \ (b := \text{tmp}) ; \\
\text{pause.b} & \rightarrow \ \text{cAck} & \rightarrow & \ \text{SKIP}
\end{align*}
\]

where the \texttt{spinner} processes have become:

\[
\begin{align*}
\text{PS}(b) &= \text{mb.b} \rightarrow \ \text{PS}(b) \ [ ] \\
\text{pause.b} & \rightarrow \ \text{spin.b} & \rightarrow & \ \text{PS}(b) \ [ ] \\
\text{done} & \rightarrow & \ \text{SKIP}
\end{align*}
\]

Note that we only need one signal to terminate all \texttt{spinners}. 

‘disengage’ from \texttt{b}

‘engage’ with \texttt{b}
Barriers (mobile)

So a process, \( P \), that may receive (or lose) a mobile barrier must be prepared to receive (or lose) any of them – i.e. it needs a full set of spinners:

\[
(P; \text{done} \rightarrow \text{SKIP}) \mid \{\text{pause, spin, done}\} \mid \text{PS} \\
\backslash \{\text{pause, spin, done}\}
\]

where:

\[
\text{PS} = \mid \{\text{done}\} \mid \{\text{PS}(b) \mid b = 0, 1, 2, \ldots\}
\]

Is that allowed? FDR will want that PS set limited a bit!!

Note: these spinners are different from the ones introduced earlier for the \textit{occam-\pi PAR ENROLL} and \textit{FORK} constructs. Perhaps they could be merged …? Simpler not to …?
Barriers (mobile)

So a process, $P$, that may receive (or lose) a mobile barrier must be prepared to receive (or lose) any of them – i.e. it needs a full set of spinners:

$$(P; \text{done} \rightarrow \text{SKIP}) \mid \{\text{pause}, \text{spin}, \text{done}\} \mid \text{PS} \\ \backslash \{\text{pause}, \text{spin}, \text{done}\}$$

where:

$$\text{PS} = |\{\text{done}\}| \{\text{PS}(b) | b = 0, 1, 2, \ldots\}$$

Note that an extra set of spinners will be needed for each component process of a PAR (or PAR ENROLL) or FORK that constructs, receives or loses mobile barriers.
Barriers (mobile)

So a process, $P$, that may receive (or lose) a mobile barrier must be prepared to receive (or lose) any of them – i.e. it needs a full set of spinners:

\[
(P; \text{done} \rightarrow \text{SKIP}) | \{\text{pause, spin, done}\} | \text{PS} \\
\backslash \{\text{pause, spin, done}\}
\]

where:

\[
\text{PS} = |\{\text{done}\}| \{\text{PS}(b) \mid b = 0, 1, 2, \ldots\}
\]

If a process starts off enrolled on a mobile barrier, $b$, (e.g. from a PAR ENROLL or FORK), then its corresponding spinner must start in its paused state,

\[
\text{spin.b} \rightarrow \text{PS}(b)
\]
Barriers (RESIGN blocks)

An occam-\(\pi\) process may temporarily RESIGN from a barrier on which it is currently enrolled:

\[
\text{RESIGN} \quad b \\
\text{x}
\]

\[
\text{spin.b} \rightarrow \text{x}; \quad \text{pause.b} \rightarrow \text{SKIP}
\]

However, its use has to be more structured than this. To control the phase (see the earlier platelet models) in which a resigned process rejoins the barrier, end-of-resignation has to be approved by (and acknowledged to) another process that is also enrolled on the barrier.

We also need this protocol to force our spinner to take immediate notice of the pause.b signal.
Barriers (RESIGN blocks)

For example:

SEQ
  RESIGN b
SEQ
  X
  c ! 0
  c ! 0

-- on holiday (from b)
-- request to come back
-- acknowledge we are back

spin.b -> X; c -> pause.b -> c -> SKIP

where the end-of-resignation control process executes:

INT any:
SEQ
  c ? any
  c ? any

  c -> c -> SKIP
Barriers (**RESIGN** blocks)

So necessary is this protocol that we are considering burning it into the language design – possibly:

```
RESIGN b  
X -- may not use b
RESUME c!
```

```
spin.b -> X; c -> pause.b -> c -> SKIP
```

where the *end-of-resignation* control process executes:

```
RESUME c?
```

```
c -> c -> SKIP
```

may be used as an ALT guard
Barriers (**RESIGN** blocks)

Note that an **occam-π** process may *temporarilly* **RESIGN** from a **static** (as well as **mobile**) barrier.

This means that the process following a **static** barrier declaration (if it includes a **RESIGN** block) might also be modelled with a **pauseable terminatable** **spinner** process. Just one is needed for each barrier on which a **RESIGN** block occurs.

*Alternatively*, it’s simpler not to do the above! Just introduce a new **terminatable** **spinner** process for the duration of the **RESIGN** block. *Details on the next slide …*

We have not used this **RESIGN** mechanism in our **occam-π** **platelet** models (so far). They *are* used in the **lazy-cellular-automaton** version of our ‘*Game-of-Life*’ model (with a single and **static** barrier).
Barriers (RESIGN blocks)

RESIGN b
X  -- may not use b
RESUME c!

(static) BARRIER

CHAN INT

\((X; c \rightarrow \text{done} \rightarrow c \rightarrow \text{SKIP}) | \{\text{done}\} | Sb \) \\
\{\text{done}\}

where:

\[ Sb = (b \rightarrow Sb) [] (\text{done} \rightarrow \text{SKIP}) \]

where the end-of-resignation control process executes:

RESUME c?

may be an ALT guard

c \rightarrow c \rightarrow \text{SKIP}
**Barriers (mobile aliasing)**

*Aliasing* always leads to tears. In *occam-π*, two variables both in scope and of the same type should never refer to the *same* entity (e.g. data, channel, barrier). This assumption is crucial for the enforcement of parallel usage rules, as well as for the general transparency of code – *WYSIWYG*.

Language design prevents this happening for everything except *mobile barriers* and *SHARED mobile channel-ends* (looked at later).

*Aliasing* is difficult to prevent for *mobile barriers*. All that needs to happen is for one process to send the *same* barrier twice and for the receiving process to input it into distinct variables. This must be stopped!
Barriers (mobile aliasing)

Fortunately, any attempt to set up aliasing of mobile barrier variables will deadlock the sending and receiving processes (with our CSP model).

If a process receives the index of a mobile barrier on which it is already enrolled (via another mobile barrier variable), it will try to pause an already-paused spinner. That puts the receiving process into deadlock with that spinner. The sending process is also left deadlocked, waiting for the acknowledgement.

So, at least we cannot have running code in which mobile barrier variables are aliased to the same mobile barrier.

But it would be nice if we could devise compile-time checkable language rules that would prevent such attempts (and the resulting deadlocks).
Barriers (mobile aliasing)

Note that if we wanted to allow aliasing of mobile barrier variables, our CSP model could easily accommodate that:

\[
PS(b) = mb.b \rightarrow PS(b) \quad \Box \\
pause.b \rightarrow PausePS(b,1) \quad \Box \\
done.b \rightarrow SKIP
\]

\[
PausePS(b,n) = (spin.b \rightarrow PausePS(b,n-1)) \quad \Box \\
(pause.b \rightarrow PausePS(b,n+1)), \quad \text{if } n > 0
\]

\[
PausePS(b,0) = PS(b)
\]

But aliasing causes so many problems, we do not want this!
occam-π (semantics, application)

- Barriers (CSP multiway events)
- Platelet Model #6 (‘busy’ CA)
- Barrier resignation (spinning, (priorities))
- Forking (communication, parallel recursion)
- Platelet model #7 (‘lazy’ CA)
- Mobile barriers (more spinning)

Channel-ends (design pattern or for real?)
Mobile channel-ends (tokens, interleaving)
Shared mobile channel-ends (ditto)
Channel-ends (design pattern or for real?)

Mobile channel-ends (tokens, interleaving)

Shared mobile channel-ends (ditto)

Sorry – still to be written up. However, it looks like these are simpler to model than the mobile barriers – no spinning!

The model for mobile channel-ends (shared or otherwise) will not be derived from that for mobile barriers. We are modelling patterns of event use – and occam-π patterns of channel use are different to those for its barriers (even though CSP sees them the same and allows the same freedom to both).

The key simplifying constraint is that only two processes (sender and receiver) ever engage to complete a channel synchronisation – even when the channel-ends are shared.
for example

This is the matrix-agents example described in earlier presentations. Our Platelet model #7 is the first realisation of the techniques introduced here.
Location (Neighbourhood) Awareness

The Matrix

Mobile Agents
Location (Neighbourhood) Awareness
Location (Neighbourhood) Awareness
Location (Neighbourhood) Awareness
Mobility and Location Awareness

- **The Matrix**
  - A network of (mostly passive) server processes.
  - Responds to client requests from the mobile agents and, occasionally, from *neighbouring* server nodes.
  - Deadlock avoided (in the matrix) *either* by one-place buffered server channels *or* by pure-client slave processes (one per matrix node) that ask their server node for elements (e.g. mobile agents) and forward them to neighbouring nodes.
  - Server nodes only see neighbours, maintain registry of currently located agents (and, maybe, agents on the neighbouring nodes) and answer queries from local agents (including moving them).

- **The Agents**
  - Attached to one node of the Matrix at a time.
  - Sense presence of other agents – on local or neighbouring nodes.
  - Interact with other local agents – must use agent-specific protocol to avoid deadlock. May decide to reproduce, split or move.
  - Local (or global) *sync barriers* to maintain sense of time.
Barriers, Mobiles, Semantics and Platelets

Kent TUNA Group
Computing Laboratory
University of Kent at Canterbury

TUNA meeting, Surrey (11\textsuperscript{th} April, 2005)
[Revised: 13th June, 2005]
Barriers (mobile)

Following these modelling rules for passing a mobile barrier to a forked process:

\[
\text{FORK P (b)}
\]

\[
\text{up} \rightarrow \text{forkP!b} \rightarrow \text{forkPack} \rightarrow \text{spin.b} \rightarrow \text{SKIP}
\]

where:

\[
\text{ForkP} = \text{forkP?b} \rightarrow (P' (b) \ ||\ | \ | \ | \ ForkP)
\]

\[
\text{c?tmp} \rightarrow (b := \text{tmp});
\]

\[
\text{pause.b} \rightarrow \text{cAck} \rightarrow \text{SKIP}
\]

‘disengage’ from b

‘engage’ with b