Mobile Barriers: Semantics, Implementation and Application

Peter Welch and Fred Barnes
(University of Kent)

IFIP WG 2.4, Jackson’s Mill
**Overview ...**

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

**Aim:** present some fine-grained *blood platelet* models in *occam*-π, motivating the above.

**Aim:** map these new *occam*-π mechanisms on to *CSP*, so that we can apply formal reasoning to the design and analysis of such systems.
Barriers (static)

The **occam-π** BARRIER type corresponds to a multiway CSP event, though some higher level design patterns (such as resignation) have been built in.

Basic CSP semantics apply. When a process synchronises on a barrier, it blocks until all other processes enrolled on the barrier have also synchronised. Once the barrier has completed (i.e. all enrolled processes have synchronised), all blocked processes are rescheduled for execution.
The \texttt{occam-\pi} \texttt{BARRIER} type corresponds to a multiway \texttt{CSP} \texttt{event}, though some higher level design patterns (such as \texttt{resignation}) have been built in.

A \texttt{PAR} construct must explicitly \texttt{ENROLL} its components on barriers.

The number of processes enrolled on an in-scope barrier is unchanged by a non-enrolling \texttt{PAR} – only one of its components may reference it.
Barriers (static)

Processes may synchronise on more than one barrier:

```
worker (0) worker (1) ... worker (n-1)
```

To synchronise on a barrier:

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

- `SYNC b`
  or
- `SYNC c`
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
  :
```
Of course, only one barrier is actually needed to synchronise the phases in this example:

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

occam-$\pi$ BARRIER synchronisation is safe in the sense that enrollment and resignation are automatically managed. A process may synchronise on a BARRIER if and only if it is enrolled.

Try to break this rule … your program won’t compile. There are zero memory and run-time costs to enforce it. 😊
Barriers – Auto-resignation

When an enrolled process **terminates**, it automatically **resigns** its enrollment on the barrier. This allows other enrolled processes to continue to synchronise on the barrier, **without being deadlocked** by the non-appearance of the terminated process.

This has the nice property that **SKIP** is a *unit* of all varieties of the **PAR** operator:

\[
\text{PAR } P \ (b) \ \text{SKIP} \quad = \quad \text{PAR ENROLL } b \ P \ (b) \ \text{SKIP} \quad = \quad P \ (b)
\]
Barriers – Auto-resignation

When an enrolled process terminates, it automatically resigns its enrollment on the barrier. This allows other enrolled processes to continue to synchronise on the barrier, without being deadlocked by the non-appearance of the terminated process.

It’s also what we want for our modelling …

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

worker processes may drop out of the system at any time – without provoking deadlock!
Barriers – Auto-resignation

In **CSP**, this is *not* the case! Parallel processes terminate together – so a process trying to terminate is still registered for all its *multi-way events* (barriers) and will block any siblings still trying to engage in them.

So in **CSP**, **SKIP** is a *unit* only of parallel *interleaving*. It is not a *unit* of any parallel operator bound to an event.

If we want that **CSP** semantics, simply declare and enroll an extra barrier and get each process to synchronise on it *once*, just before it terminates:

```
BARRIER alldone:
PAR i = 0 FOR n ENROLL b, alldone
SEQ
  worker (i, b)
SYNC alldone
```

All **worker** processes must terminate in their *same** **SYNC** **b** cycle – else deadlock!
An occam-π process may temporarily resign from a barrier on which it is currently enrolled:

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

For example:

```plaintext
SEQ
  RESIGN b
  SEQ
    x  -- on holiday (from b)
    c ! 0  -- request to come back
    d ! 0  -- acknowledge we are back
```

where the *end-of-resignation* control process (which must be enrolled on `b`) and in the agreed *SYNC phase*) executes:

```plaintext
ALT
  INT any:
    c ? any  -- accept request to come back
    d ? any  -- wait for acknowledge
  ...  other guards
```
So useful is this protocol that we are considering burning it into the language design – possibly:

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

---

Copyright P.H.Welch
**Barriers – Cost**

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

*occam-π* BARRIER synchronisation is fast: if cache prediction is defeated (e.g. by application behaviour causing irregular scheduling of processes), the BARRIER synchronisation cost is still very low (~150 ns per process for ~1M processes).

---

**Diagram:**
- **Y-axis:** Sync time per process (ns)
- **X-axis:** Number of processes (1 to 16M)
- Lines represent different numbers of processes:
  - 1
  - 4
  - 16
  - 1024
  - 16384
  - 65536
  - Random
Case Study: blood clotting

**Haemostasis:** we consider a greatly simplified model of the formation of blood clots in response to damage in blood vessels.

**Platelets** are passive quasi-cells carried in the bloodstream. They become *activated* when a balance between chemical suppressants and activators shift in favour of activation.

When activated, they become *sticky* …

We are just going to model the clumping together of such sticky activated platelets to form *clots*.

To learn and refine our modelling techniques, we shall start with a simple one-dimensional model of a bloodstream.
Platelet Model (‘busy’ CA)

Space is represented as a pipeline of cell processes. Activated (i.e. sticky) platelets are generated and injected into the pipeline at a user-determined randomised rate. They move through the cells at speeds inversely proportional to the size of the clot in which they become embedded – these speeds are randomised slightly. Clots that bump together stay together.

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

Platelets/clots are not directly modelled as processes.
Platelet Model ('busy' CA)

gen  →  cell  →  cell  →  cell  →  hole

keywatch  →  display  →  screen

keyboard  →  draw
Platelet Model (‘busy’ CA)

Key:

- Phase 0
- Phase 1

Shared state

Copyright P.H.Welch
Platelet Model (‘busy’ CA)

Key:

- Phase 0

Many readers, no writers

No readers, exclusive writer (per cell element)
Platelet Model (‘busy’ CA)

Key:

- Phase 1

- one reader, no writer

- no readers, exclusive writer

Draw:

- hole

- cell display array

- running

- cell

- cell

- cell

- hole

Keyboard:

- keywatch

Screen:

- display
Platelet Model ('busy' CA)

- empty
- full (platelet)
- don't care
- possibilities for middle cell
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
The (1-D) blood-stream zigzags (left-right, right-left, left-right, ...) down the screen: grey dots show empty cells, black dots show cells with a platelet. All platelets are sticky.
Platelet Model (‘busy’ CA)

**Performance:** each cell has to work harder if full (carrying a platelet). Also, clot 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>
Platelet Model (‘busy’ $\rightarrow$ ‘lazy’ CA)

**Scaling problem:** every cell is active every cycle – regardless of whether it contains a platelet. This works well for systems with up to ~100K cells.

For TUNA, we will need to be working in 3D (say, ~10M cells), modelling many different types of agent with much richer rules of engagement.

These automata must become ‘lazy’, whereby only processes with things to do remain in the computation.
Platelet Model (‘busy’ $\rightarrow$ ‘lazy’ CA)

*Logical problem:* the rules for the different stages in the life cycle of *platelets*, or *clots*, are coded into different cycles of the cells. Each *cell* sees lots of different *platelets* – sometimes bunched together as *clots* – and operates on them as they pass through.

No process directly models the development of a single *clot*.

The following system addresses this. The *cell* processes are pure *servers*, not enrolled on the time-synchronising *barrier*. Their *clients* are *clot* processes, *generated dynamically*, that are enrolled on the *barrier* and use that *barrier* to synchronise access to the *cell* servers with their generator and the display.

The *cell* processes are only worked as *clot boundaries* pass over them.
The following system addresses this. The cell processes are pure servers, not enrolled on the time, not synchronising barrier. Their clients are clot processes, generated dynamically, that are enrolled on the barrier and use that barrier to synchronise access to the cell servers with their generator and the display.

To manage this, we need to move barriers to FORKed processes. The general solution is given by making barriers MOBILE.
Barriers (mobile)

\textit{occam-\pi} includes \textit{mobile} barrier types:

\begin{center}
\begin{align*}
\text{MOBILE BARRIER } & b: \\
\text{SEQ} \\
& b := \text{MOBILE BARRIER} \\
\ldots & \text{logic involving } \text{SYNC } b
\end{align*}
\end{center}

Whenever a barrier is constructed, the process doing the construction becomes \textit{enrolled}.

Whenever a \textit{defined} barrier variable is overwritten or goes out of scope, the process holding it \textit{resigns}.

Channels may carry \textit{MOBILE BARRIER}s as components of their messages (\textit{occam-\pi PROTOCOL}s).

Whenever a barrier is communicated (e.g. to a \textit{forked} process), the receiving process dynamically and atomically \textit{enrolls} and the sending process \textit{resigns} (unless a \textit{CLONE} is sent).
Forking Processes with Barriers

\[ \text{FORKING} \quad X \quad \rightarrow \quad P \]

\text{occam-\(\pi\) view}

Copyright P.H.Welch
Forking Processes with Barriers

FORKING

P
P
P

occam-$\pi$ view

Copyright P.H.Welch
Platelet Model ('lazy' CA)

gén

cell cell cell cell cell cell
clot

draw

keywatch

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

cell → cell → cell → cell → cell → cell

gen → clot

draw

keywatch → display → screen

phase 0
Platelet Model (‘lazy’ CA)

gen

cell

cell

cell

cell

cell

cell

clot

keywatch

display

keyboard

draw

screen

phase 0
Platelet Model (‘lazy’ CA)

generation → cell → cell → cell → cell → cell → cell

clot

draw

keywatch

keyboard

display

screen

phase 1
Platelet Model (‘lazy’ CA)

cell cell cell cell cell cell

gen
clot

phase 0

draw

keywatch

display

keyboard

screen

Copyright P.H.Welch
Platelet Model (‘lazy’ CA)

cell → cell → cell → cell → cell → cell

keywatch

keyboard

display

screen

clot

phase 0

draw

gen
Platelet Model (‘lazy’ CA)
Platelet Model (‘lazy’ CA)
Platelet Model (‘lazy’ CA)

Diagram showing the process with nodes labeled 'cell', 'clot', 'gen', and 'draw'. Connections between these nodes indicate interactions such as generation, clot formation, and drawing.
Platelet Model ('lazy' CA)

gen → clot → cell → clot → cell → draw

keywatch → display → screen

keyboard
Platelet Model ('lazy' CA)

cell → cell → cell → cell → cell → cell

cloth

generate

draw

display

cell

keywatch

keyboard

screen

phase 1
Platelet Model (‘lazy’ CA)

cell cell cell cell cell cell
gen
clot

phase 1
draw

keywatch

keyboard

display

screen

Copyright P.H.Welch
**Platelet Model (‘lazy’ CA)**

**Performance:** A cell only works when a clot boundary moves through. Run-time depends only on the number of clots; the clot sizes are now irrelevant (2.4 GHz. P IV-M).

<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>
Clot Frequency by Position by Size

2000 clots, probability 1/256
Clot Frequency by Position by Size

2000 clots, probability 2/256

frequency

reporting position

clot size

Copyright P.H.Welch
Clot Frequency by Position by Size

2000 clots, probability 3/256

frequency

reporting position

clot size

Copyright P.H.Welch
Clot Frequency by Position by Size

2000 clots, probability 4/256
Clot Frequency by Position by Size

2000 clots, probability 5/256

frequency

reporting position

clot size
Clot Frequency by Position by Size

2000 clots, probability 6/256
Clot Frequency by Position by Size

2000 clots, probability 7/256
Clot Frequency by Position by Size

2000 clots, probability 8/256

frequency

reporting position

clot size
Clot Frequency by Position by Size

2000 clots, probability 9/256

Copyright P.H.Welch
We can’t directly model occam-\(\pi\) barriers as CSP multiway events because of the semantic dynamics (enrollment, resignation, mobility), which are unknown to CSP events.

Instead, we model a mobile barrier as a process, instrumented with channels to control dynamic enrollment/resignation and synchronisation. Each such process (i.e. mobile barrier) is constructed on demand and given a unique index number. Mobility derives from simply communicating that index.
enroll.b?p
resign.b
tenroll.b
tresign.b
sync.b
ack.b

BAR (b, refs, n, count)

index number
reference count
enrolled count
sync count
BAR \( (b, \text{refs}, n, \text{count}) \) =

- enroll.b?p \rightarrow BAR (b, \text{refs} + p, n + p, \text{count} + p) \[]
- resign.b \rightarrow BAR (b, \text{refs} - 1, n - 1, \text{count} - 1) \[]
- tresign.b \rightarrow BAR (b, \text{refs}, n - 1, \text{count} - 1) \[]
- tenroll.b \rightarrow BAR (b, \text{refs}, n + 1, \text{count} + 1) \[]
- sync.b \rightarrow BAR (b, \text{refs}, n, \text{count} - 1)

count > 0
BAR (b, refs, n, count)

BAR (b, refs, n, 0) = 
BAR_ACK (b, refs, n, 0)

BAR_ACK (b, refs, n, count) = 
ack.b -> BAR_ACK (b, refs, n, count + 1)

BAR_ACK (b, refs, n, n) = BAR (b, refs, n, n)
enroll.b?p
resign.b
tenroll.b
tresign.b
sync.b
ack.b

BAR (b, refs, n, count)

BAR (b, refs, 0, 0) =
tenrol.b -> BAR (b, refs, 1, 1)

BAR (b, 0, 0, 0) = SKIP

refs > 0
Kernel Processes

Mobile barrier processes are *forked* as needed by a generator:

\[
\text{MB} (b) = \begin{align*}
\text{getMB}!b & \rightarrow (\text{BAR} (b, 1, 1, 1) \ || \ || \ \text{MB} (b + 1)) \ [] \\
\text{noMoreBarriers} & \rightarrow \text{SKIP}
\end{align*}
\]

For convenience, we also define:

\[
\text{undefined} = 0 \\
\text{UNDEFINED\_BAR} = \begin{align*}
\text{resign.undefined} & \rightarrow \text{UNDEFINED\_BAR} \ [] \\
\text{noMoreBarriers} & \rightarrow \text{SKIP}
\end{align*}
\]
Kernel Processes

The mobile barrier kernel is:

```
MOBILE_BARRIERKERNEL =

   MB (1) | {noMoreBarriers} | UNDEFINEDBAR
```

Let’s define:

```
kernel_chans =

   {enrol.b.p, resign.b, tresign.b, tenrol.b, sync.b,
    ack.b, getMB, noMoreBarriers | b >= 0, p >= 1}
```
So, if \texttt{APPLICATION\_SYSTEM} is the occam-$\pi$ application and \texttt{APPLICATION\_SYSTEM'} is the CSP modelling of its mobile barrier primitives \textit{(in a minute)}, then the full model is:

\[
((\texttt{APPLICATION\_SYSTEM'}; \texttt{noMoreBarriers} \rightarrow \texttt{SKIP}) \\
|\texttt{kernel\_chans}| \\
\texttt{MOBILE\_BARRIER\_KERNEL}) \setminus \texttt{kernel\_chans}
\]

Here's a diagram ...
Application and Kernel Processes

**APPLICATION_SYSTEM’**

- resign.0
- noMoreBarriers
- getMB.b

**MOBILE_BARRIER KERNEL**

- BAR (1, ...)
- BAR (2, ...)
- BAR (3, ...)
- UNDEFINED_BAR
- MB (4)
We express this using *Circus*. Amongst other things, it adds variables and assignment into *CSP*, which we find convenient. The paper describes how these map down to pure *CSP*.

\[
\text{MOBILE BARRIER } b \colon \\
P\left( b \right)
\]

\[
\text{Var } b : \mathbb{N} \cdot b \coloneqq \text{undefined}; P'(b); \text{resign}.b \rightarrow \text{SKIP}
\]

where \( P'(b) \) is the *CSP* model of \( P(b) \).

\[
b \coloneqq \text{MOBILE BARRIER}
\]

\[
\text{getMP}?\text{tmp} \rightarrow (b := \text{tmp})
\]
Modelling occam-π Mobile Barriers

SYNC b ~ sync.b -> ack.b -> SKIP

c ! b ~ c!b -> (b := undefined)

c ! CLONE b ~ enroll.b!l -> c!b -> SKIP

c ? b ~ c?tmp -> resign.b -> (b := tmp)

a := b ~ resign.a -> (a := b);
(b := undefined)

a := CLONE b ~

((enroll.b!l -> SKIP) |||
(resign.a -> SKIP));
(a := b)
Modelling occam-π Mobile Barriers

RESIGN b
P

tresign.b -> P';
tenroll.b -> SKIP

RESIGN b
P
RESUME c! d!

tresign.b -> P';
c -> tenroll.b -> d -> SKIP

RESUME c? d?

c -> d -> SKIP
Modelling occam-$\pi$ Mobile Barriers

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

\[
\begin{align*}
(PAR\_COUNT(n) & ) \\
\{\text{down}.j \mid j = 0 \ldots (n-1)\} & \\
\text{enroll}.b!(n-1) & \to \\
\{P'(i, b); \\
\quad \{\text{down}.n \to (\text{SKIP} < | (n = 0) \mid > \text{resign}.b \to \text{SKIP}) \mid i = \text{start} \ldots \text{start} + (n-1)\} & \\
\} \setminus \{\text{down}.j \mid j = 0 \ldots (n-1)\}
\end{align*}
\]

where:

\[
\begin{align*}
\text{PAR\_COUNT(j)} & = \text{down}!(j-1) \to \text{PAR\_COUNT}(j-1) & j > 0 \\
\text{PAR\_COUNT}(0) & = \text{SKIP}
\end{align*}
\]
Modelling occam-$\pi$ Mobile Barriers

\[ \text{FORK } P \ (b) \quad \sim \quad \text{forkP!}b \rightarrow (b := \text{undefined}) \]

\[ \text{FORK } P \ (\text{CLONE } b) \quad \sim \quad \text{enroll.b!}1 \rightarrow \text{forkP!}b \rightarrow \text{SKIP} \]

\begin{align*}
\text{FORKING} \quad X \\
\quad \sim \quad (X'; \ done \rightarrow \text{SKIP}) \\
& \quad |\{\text{forkP.b, done | b > 0}\}| \quad \text{FORK_P} \\
& \quad \) \ \{\text{forkP.b, done | b > 0}\} \\
\end{align*}

where:

\[ \text{FORK_P} = \text{forkP?}b \rightarrow ((P' (b); \ resign.b \rightarrow \text{done} \rightarrow \text{SKIP}) \\
& \quad |\{\text{done}\}| \ \text{FORK_P}) \ [\] \\
& \quad \text{done} \rightarrow \text{SKIP} \]

Slides at end give an account of occam-$\pi$ forking ...
The semantics of static barriers follows from those for mobile ones. A static barrier is just a mobile we never move.

To transform static declarations into mobile ones:

\[
\text{BARRIER } b: \quad \text{P} \ (b) \quad \mapsto \quad \text{MOBILE BARRIER } b: \quad \text{SEQ} \\
\quad \quad b := \text{MOBILE BARRIER} \\
\quad \quad \text{P} \ (b)
\]

All BARRIER parameters / abbreviations become MOBILE BARRIER parameters / abbreviations.

No other transformations are needed.
Summary and the Future

We now have a complete model for mobile barriers (and channels) in CSP, so that we can apply formal reasoning to the design and analysis of occam-\(\pi\) systems. Model checking will require a little more work to constrain numbers to finite limits!

The occam-\(\pi\) barrier, forking and mobile mechanisms seem to be delivering their promises. Applications like these for TUNA will be a strong testing ground for the mixing of the dynamic mechanisms of the \(\pi\)-calculus into CSP.

Despite the very simple clotting model, unprogrammed behaviour (the phase change between the free-flow of clots and catastrophic jams) has emerged that is encouraging.
Summary and the Future

Our ambitions in the TUNA project call for scaling the size of these models through 3 orders of magnitude (~10M processes) and hard-to-quantify orders of complexity.

For the safe introduction of nanites implementing artificial blood platelets, getting the balance right between stimulation and inhibition of clotting reactions will be crucial to prevent a runaway chain reaction.

We will need to model (and visualise) in 3D, factor in a mass of environmental stimulators, inhibitors and necessary supporting materials (e.g. fibrinogen), and distribute the simulations efficiently over many machines – to obtain sufficient memory capacity and processor power.
Summary and the Future

- **We Aim to Have Fun ...**
  - Interesting applications everywhere ...
  - Beat the complexity / scalability rap ...
  - Would anyone like to join us ... ?

- **Google – I’m feeling Lucky ...**
  - KRoC + ofa -- **occam-π** (official)
  - occam + web server -- **occam-π** (latest)
  - JCSP -- CSP for Java
  - Quickstone -- JCSP Networking Edition (Java / J#)
  - Grand Challenges + UK -- In-vivo ↔ In-silico
  - WoTUG -- Lots of good people ...

- **Mailing lists ...**
  - occam-com@kent.ac.uk
  - java-threads@kent.ac.uk
Modelling Bio-Mechanisms

- **In-vivo ↔ In-silico**
  - One of the UK ‘Grand Challenge’ areas.
  - Move *life-sciences* from *description* to *modelling* and *prediction*.
  - Example: the Nematode worm.
  - Development: *from fertilised cell to adult* (with virtual experiments).
  - Sensors and movement: *reaction to stimuli*.
  - Interaction *between organisms and other pieces of environment*.

- **Implementation technologies**
  - Communicating process networks – fundamentally good fit.
  - Cope with growth / decay, combine / split (evolving topologies).
  - Mobility and location / neighbour awareness.
  - Simplicity, dynamics, performance and safety.

- **occam-π (and JCSP)**
  - Robust and lightweight – good theoretical support.
  - ~10,000,000 processes with useful behaviour in useful time.
  - Enough to make a start
Modelling Nanite-Assemblies

- **TUNA: Theory Underpinning Nanotech Assemblies**
  - Active *nano-devices* that manipulate the world at *nano-scale* *macroscopic* effects (e.g. through assembling artifacts).
  - Need vast numbers of them – but these can grow (exponentially).
  - Need capabilities to design, program and control complex and dynamic networks – build desired artifacts, not undesired ones.
  - Need a theory of dynamic networks and emergent properties.

- **Implementation technologies**
  - Communicating process networks – fundamentally good fit.
  - Cope with growth / decay, combine / split (evolving topologies).
  - Mobility and location / neighbour awareness.
  - Simplicity, dynamics, performance and safety.

- **occam-π (and JCSP)**
  - Robust and lightweight – good theoretical support.
  - ~10,000,000 processes with useful behaviour in useful time.
  - Enough to make a start

---

---
We now want to exercise this model to investigate how various factors affect the creation and distribution of clots along the bloodstream from the point of damage.

For this initial very simple model, the only factor we have is the generation probability. For instance, how great does this have to be to end free-flow and cause the clot to grow back to the source of the problem? For such a probability, how far beyond the damage does the clot extend?

We need a parameter sweep across the range of probabilities, repeating each experiment many times (since the basic model is stochastic). We need a lot of computation ... used Minimum intrusion Grid middleware.

We also need to instrument the model to log the necessary data ...
The Minimum intrusion Grid (MiG)

Grid middleware: security, scalability, privacy, strong scheduling and fault tolerance are included by design.

Non-intrusive: minimum initial effort, software and maintenance needed by either users or compute resources.

The MiG is a set of processes, running on a set of servers. It is not a special protocol that users and resources have to support.

Users and resources communicate with the MiG as clients, using standard protocols (https, scp and ssh).

Security: all users, resources and MiG servers are identified by a signed certificate and private key. No insecure transfers or storage (session only validity).
The Minimum intrusion Grid (MiG)

**User**
- submit job (https)
- get results (https)

**MiG**
- request job (https)
- job (scp)
- send results (https)
- cleanup (ssh)

**Resource**
MiG Execution Statistics ('busy' CA)
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, operating systems and modelling dynamic systems.

We do not operate a reference-anything heap in occam-π. Strict aliasing control is maintained even for dynamically allocated structures. We must also take care about non-mobile memory (statically allocated on process stacks) referenced by long-lived forked processes.
Forking Processes

PROC A (SHARED CHAN BYTE error!)
... local state
SEQ
...

**FORKING**
SEQ
...

**WHILE fork**ing
SEQ
...

**FORK** P (n, error!, svr, cli)
...
...
...

Often, we need to **FORK** processes within a **FORKING** block ...

Process P(...) starts running concurrently with this process.

All **FORK**ed processes must terminate before its **FORKING** block can terminate.
Forking Processes

PROC A (SHARED CHAN BYTE error!)
... local state
SEQ
...

FORKING
SEQ
...
WHILE forking
SEQ
...
...
FORK P (n, error!, svr, cli)
...
...
...

VAL data are copied into a FORKed process
MOBILE data and channel-ends are moved into a FORKed process
Forking Processes

PROC A (SHARED CHAN BYTE error!)
... local state
SEQ
...

FORKING
SEQ
...
WHILE forking
SEQ
...
...
FORK P (n, error!, svr, cli)
...
...

\begin{itemize}
  \item i.e. VAL data are \textit{communicated} to the FORKed process
  \item i.e. MOBILE data and channel-ends are \textit{communicated} to the FORKed process
  \item Non-mobile references must be \texttt{SHARED} and exist \texttt{global} to the FORKING block
  \item Otherwise, they may have ceased to exist before the FORKed process terminates
\end{itemize}