Mobile Barriers: Semantics, Implementation and Application
Peter Welch and Fred Barnes (University of Kent)
IFIP WG 2.4, Jackson's Mill (4th, Oct, 2005)

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.

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.

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

A PAR construct must explicitly ENROLL its components on barriers.

Barriers (static)
Processes may synchronise on more than one barrier:

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 / initialization
WHILE running
SEQ
SYNC b
... phase b computation
SYNC c
... phase c computation

To synchronise on a barrier:

Sync b or Sync c

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 number of processes enrolled on an in-scope barrier is unchanged by a non-enrolling PAR – only one of its components may reference it.

To synchronise on a barrier:

Sync b or Sync c

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.
**Barriers (static)**

Of course, only one barrier is actually needed to synchronise the phases in this example:

```plaintext
PROC worker (VAL INT id, BARRIER b) {
  while running
    SEQ
      SYNC b
    phase 0 computation
    SYNC b
    phase 1 computation
  
  ... local declarations / initialisation...
}
```

**Barriers – Safety**

**occam**'s **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:

```plaintext
PAR b SKIP = PAR ENROLL b b SKIP = b
```

**Barriers – Auto-resignation**

It's also what we want for our modelling …

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:

**Barriers – RESIGN blocks**

An **occam** process may temporarily **RESIGN** from a barrier on which it is currently enrolled:

- logic involving **SYNC** b
- **RESIGN** b
- logic involving **SYNC** b

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:

```
SEQ
  RESIGN b
  x    -- on holiday (from b)
  a ! 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:

```
ALT
  INF any:
    c ? any -- accept request to come back
    d ? any -- wait for acknowledgement
```

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

```
RESIGN b
  x    -- may set msg b
  RESUME c! d!
```

where the end-of-resignation control process executes:

```
RESUME c? d?
```

Barriers – Cost

**occam**\(\pi\) 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.

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)

**Key:**
- Phase 0
- Phase 1
- Exclusive writer
- Many readers, no writers
- One reader, no writer
- No readers, exclusive writer
- No readers, no writers
- Full (platelet)
- Empty
- Possibilities for middle cell

```
PROCP cell (BYTE my.visible.state, BOOL running, BARRIER draw
    CHAN int, out, in)
... local declarations / initialisation (phase 0)
WHILE running
    SEQ
        SYNCS 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
                ... if not tail, pass decision back
                ... if tail and move, become empty
            ... else receive decision on move from cell ahead
        ... else if clump behind exists and moves, become full
        SYNCS draw -- phase 0
        ... update my.visible.state
```
Platelet Model (Visualisation)

The (1-D) blood-stream zigzags (left-right, right-left, ... 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’ → ‘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’ → ‘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.

Platelet Model (‘busy’ → ‘lazy’ CA)

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

Their clients, that are enrolled on the barrier synchronise access to the cell servers with their generator and the display.

Barriers (mobile)

occam-π includes mobile barrier types:

\[ \text{MOBILE BARRIER } b : \text{ seq } \]
\[ b = \text{MOBILE BARRIER } \ldots \text{ logic involving SYNCH } b \]

Whenever a barrier is constructed, the process doing the construction becomes enrolled.

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

Channels may carry MOBILE BARRIERS as components of their messages (occam-π PROTOCOL).

Whenever a barrier is communicated (e.g. to a FORKed process), the receiving process dynamically and atomically enrolls and the sending process resigns (unless a CLONE is sent).
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</td>
<td>0 (total jam)</td>
</tr>
</tbody>
</table>

Clot Frequency by Position by Size
Title goes here
**Formal Semantics (CSP) and Implementation**

We cannot directly model occam-π 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.

\[
\text{BAR} (b, \text{refs}, n, \text{count})
\]

\[
\text{enroll.b} \rightarrow \text{BAR} (b, \text{refs} + p, n + p, \text{count} + p)
\]

\[
\text{resign.b} \rightarrow \text{BAR} (b, \text{refs} - 1, n - 1, \text{count} - 1)
\]

\[
\text{tresign.b} \rightarrow \text{BAR} (b, \text{refs} , n - 1, \text{count} - 1)
\]

\[
\text{tenroll.b} \rightarrow \text{BAR} (b, \text{refs} , n + 1, \text{count} + 1)
\]

\[
\text{sync.b} \rightarrow \text{BAR} (b, \text{refs} , n , \text{count} - 1)
\]

\[
\text{ack.b} \rightarrow \text{BARACK} (b, \text{refs} , n)
\]

\[
\text{BARACK} (b, \text{refs} , n , \text{count}) = \begin{cases} 
\text{BAR} (b, \text{refs} , n , \text{count} + 1) & \text{if } \text{count} < n \\
\text{BAR} (b, \text{refs} , n , \text{count}) & \text{otherwise} 
\end{cases}
\]
Kernel Processes

The mobile barrier kernel is:

\[
\text{MOBILE\_BARRIER\_KERNEL} = \text{MB} (1) \ ||\ |\noMoreBarriers|\ |\text{UNDEFINED\_BAR}
\]

Let's define:

\[
\text{kernel\_chans} = \\
\{\text{enroll.b.p, resign.b, tresign.b, tenroll.b, sync.b, ack.b, getMB, noMoreBarriers | b} \geq 0, p \geq 1\}
\]

Application and Kernel Processes

So, if \(\text{APPLICATION\_SYSTEM}\) is the occam-\(\pi\) application and \(\text{APPLICATION\_SYSTEM}'\) is the CSP modelling of its mobile barrier primitives (in a minute), then the full model is:

\[
\{\text{APPLICATION\_SYSTEM'}, \noMoreBarriers \rightarrow \text{SKIP}\} \ \\text{\{kernel\_chans\}} \ \\text{\{MOBILE\_BARRIER\_KERNEL\}} \ \\text{\setminus kernel\_chans}
\]

Here's a diagram ...

Modelling occam-\(\pi\) Mobile Barriers

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.

\[
\text{MOBILE\_BARRIER\_KERNEL} \ \ P (b) \\
\text{Var b : MOBILE\_BARRIER} \ \ P' (b) \ \text{resign.b} \ \text{SKIP}
\]

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

- \(\textsc{BARRIER}\) becomes \(\textsc{MOBILE\_BARRIER}\)

All \(\textsc{BARRIER}\) parameters/abbreviations become \(\textsc{MOBILE\_BARRIER}\) 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 \(\pi\)-systems. Model checking will require a little more work to constrain numbers to finite limits!

The \(\pi\)-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.
Modelling Bio-Mechanisms

* In-vivo vs In-silico
  - One of the UK 'Grand Challenge' areas.
  - Move life-sciences from description to modeling 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.
  - *occams* (and JCP)
    - 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 to have macroscopic effects (e.g. through assembling artifacts).
- Need number 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.
  - *occams* (and JCP)
    - Robust and lightweight – good theoretical support.
    - ~10,000,000 processes with useful behaviour in useful time.
    - Enough to make a start.

Platelet Model

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 or resources have to support.

Users and resources communicate with the *MiG* as clients, using standard protocols (http, qnp 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)

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-w. 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.

MiG

user
submit job
get results
send results
cleanup
request job
job (seq)

resource

MiG

PROC A (SHARED CHAN BYTE error!)
... local state
SEQ
... FORKING
SEQ
... WHILE forking
SEQ
... FORK P (n, error!, avr, all)
... 
... 

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

Otherwise, they may have ceased to exist before the FORKed process terminates
Non-mobile references must be SHARED and exist global to the FORKING block

MiG

PROC A (SHARED CHAN BYTE error!)
... local state
SEQ
... FORKING
SEQ
... WHILE forking
SEQ
... FORK P (n, error!, avr, all)
... 
... 

VAL data are communicated to the FORKed process
MOBILE data and channel-ends are communicated to the FORKed process

Cont'd 2q. q may terminate concurrently with the parent.
All forked processes must terminate before its FORKING block can terminate