Communicating Process Architectures 2005
Jan Broenink, Herman Roebbers, Johan Sunter, Peter Welch, and David Wood (Eds.)
IOS Press, 2005

Mobile Barriers for occam-pi: Semantics, Implementation and Application

Peter WELCH and Fred BARNES

Computing Laboratory, University of Kent,
Canterbury, Kent, CT2 7NF, England.

{P.H.Welch , F.R.M.Barnes} @kent.ac.uk

Abstract. This paper introduces a safe language binding for CSP multiway events (*barriers* — both static and mobile) that has been built into occam-π (an extension of the classical occam language with dynamic parallelism, mobile processes and mobile channels). Barriers provide a simple way for synchronising multiple processes and are the fundamental control mechanism underlying both CSP (*Communicating Sequential Processes*) and BSP (*Bulk Synchronous Parallelism*). Formal semantics (through modelling in classical CSP), implementation details and early performance benchmarks (16 nanoseconds per process per barrier synchronisation on a 3.2 GHz Pentium IV) are presented, along with some likely directions for future research. Applications are outlined for the fine-grained modelling of dynamic systems, where barriers are used for maintaining *simulation time* and the *phased execution* of time steps, coordinating safe and desired patterns of communication between millions (and more) of processes. This work forms part of our TUNA project, investigating emergent properties in large dynamic systems (*nanite* assemblies).

Keywords. Barriers, events, processes, mobility, occam-pi, CSP, pi-calculus

Introduction

This paper describes the addition of multiway *barrier* synchronisation to the KRoC [1,2] occam-π system. occam-π [3,4,5] extends classical occam [6], including mechanisms for data, channel and process mobility (taken from Milner's π-calculus [7]), dynamic parallelism, extended rendezvous and process priority. *Static* barriers for occam-π were first reported in [8] — for completeness, some of that information is repeated here. The barriers presented in this paper may also be *mobile*, allowing them to be communicated to newly forked processes, as well as between processes. This lets us experiment with novel modelling techniques that closely follow real-world systems — such as the merging of biological organelles represented by clusters of parallel processes controlled through synchronisation on internal barriers.

Barriers are a synchronisation primitive on which parallel processes *enrol*, *synchronise* and *resign*. When a process synchronises on a barrier, it is blocked until all other processes enrolled on the barrier have also synchronised. Once the barrier is completed, all blocked processes are rescheduled. The semantics of barrier synchronisation are exactly those of an *event* in *Communicating Sequential Processes* (CSP) [9,10]. However, the dynamics of barrier *mobility*, *construction*, *enrolment* and *resignation* have no immediate counterparts in terms of CSP events. Nevertheless, we present a full CSP formalisation, modelling each barrier as a *process* rather than, directly, as an *event*. This occam-π language binding is *safe* in the sense that enrolment and resignation are automatically coordinated and that a process can synchronise on a barrier if, and only if, it is enrolled.

Barriers are used for a variety of purposes and with varying granularity in parallel programs. For example, the *Bulk Synchronous Parallelism* (BSP) [11] model describes parallel processes that run (mostly) independently on separate processors, but periodically synchronise on a single global barrier to exchange data. Such models will be supported by the networked version of occam-π (not yet released [12]). In this paper, we are concerned with much finer levels of control, with processes enrolling, synchronising and resigning dynamically on multiple barriers. We are particularly interested in applying these mechanisms to the design and implementation of highly dynamic massively parallel systems, such as those being investigated in our TUNA [13] project.

A previous implementation of barriers in KRoC [14] provided *user-defined* abstract data types [15]. 'BARRIER' variables could be declared, explicitly flagged as *shared* (through the use of compiler directives which overrode parallel usage checks) and operated via a number of procedure-calls ('initialise.barrier', 'synchronise.barrier', etc.) implemented in ETC (*Extended Transputer Code* [16]) assembler. This was functional and fast, but the programmer had to ensure that barriers were initialised correctly, that only enrolled processes could synchronise or resign and that barriers were not assigned or communicated (the semantics of which were undefined).

In the language binding presented here, barriers may be declared *static* or *mobile*, in line with occam-π data types and channels. Static barriers are *fixed* — they may not be communicated or assigned. Mobile barriers may be communicated, assigned and *cloned* (so that the source variable of the communication or assignment does not lose it). All barriers may be renamed through parameter passing and abbreviation.

Any process that declares a static barrier, or constructs a mobile one, is automatically enrolled on that barrier. Only processes enrolled on a barrier can synchronise on it. If an enrolled process itself goes parallel, there is a default constraint that at most one of its sub-processes inherits the enrolment — this is checked at compile time. However, an enrolled process may override this constraint by explicitly enrolling *all* parallel sub-processes on specific barrier(s) at the relevant PAR.

An enrolled process automatically resigns from its barrier if it loses it (through communication, exit from scope of the variable referencing it or, sometimes, on termination), so that other processes may continue to synchronize on it. A process automatically enrols on a barrier if it gains it (through communication).

An enrolled process may temporarily *resign* from a barrier — crucial for the *'lazy'* execution of simulation processes that are idle for long periods of *'time'* (see [17]). This is expressed through an explicit RESIGN block, with automatic re-enrolment at the end of the block. Often, such re-enrolment needs careful synchronisation and language support is proposed.

These occam-π barriers are more general than those of BSP (an occam-π system can contain any number of barriers, with some processes ignoring them and some registered with many). They are also more general than CSP events, incorporating ideas of *mobility* from the π-calculus and higher-level design patterns for process *resignation*. On the other hand, they are also, currently, less general than those of CSP (occam-π processes must *commit* to barrier synchronisation — which cannot, therefore, be used as a guard in a *choice* or ALT).

The language binding and informal semantics for occam-π barriers is covered in Section 1. A formal semantics is given in Section 2. An implementation outline is in Section 3, together with some early benchmarking results. Sample applications follow in Section 4. Finally, Section 5 summarises and discusses future work.

1. Language Binding and Informal Semantics

1.1. Static Barriers: Declaration

Barriers are declared in the same way as ordinary channels and variables, with the process following the declaration automatically enrolled. For example:

```
BARRIER b:        -- declaration of 'b'
...  process(es) synchronising on 'b'
```

1.2. Static Barriers: Parallel Enrolment

To enrol all sub-processes on a barrier, the parallel composition must explicitly declare this. For example:

```
PAR ENROLL b
  P (b)           -- all these
  Q (b)           -- sub-processes
  R (b)           -- are enrolled on 'b'
```

A replicated parallel may also enrol its sub-processes:

```
PAR i = 0 FOR n ENROLL b
  worker (i, b)   -- all enrolled on 'b'
```

In network diagrams, we represent a barrier as a 'bar', connected to all enrolled processes. Figure 1 shows the process network for the above '`worker`' fragment.

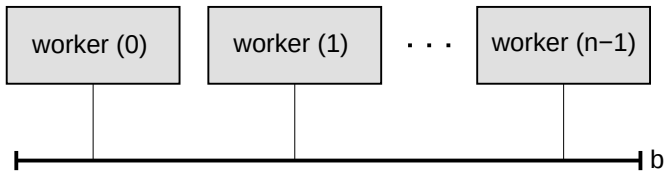

Figure 1. Barrier synchronised worker processes

1.3. Static Barriers: Synchronisation

Barrier synchronisation is expressed through a new `SYNC` primitive. For example:

```
PROC worker (VAL INT id, BARRIER x)
  SEQ
    ...  phase 0 computation
    SYNC x
    ...  phase 1 computation
:
```

The execution of the above `SYNC` line blocks until *all* other processes enrolled on the barrier similarly `SYNC`. It divides the global (parallel) computation of the system in Figure 1 into two time-separated phases (or *'supersteps'*, in BSP terminology).

Note that if a process has a barrier *parameter*, any invocation must have passed a barrier *argument* on which the invoking process was enrolled. Hence, we (and the compiler) may assume that a process with a barrier parameter *is* enrolled on whatever barrier is passed and that it is legal to synchronise.

1.4. Static Barriers: Parallel Non-Enrolment

An enrolled process that goes parallel in the normal way (i.e. without explicitly enrolling its barrier) passes its enrolment to *at most one* of its sub-processes. For example:

```
PROC worker (VAL INT id, BARRIER x)
  PAR
    A ()        -- not enrolled
    B (x)       -- enrolled on 'x'
    C ()        -- not enrolled
:
```

For a normal non-enrolling PAR such as this, exactly which (if any) of its sub-processes takes the enrolment does not matter. The compiler checks that no more than one enrols.

1.5. Static Barriers: Resign Blocks

An enrolled process may temporarily *resign* from a barrier through the use of a RESIGN-block. For example:

```
PROC worker (VAL INT id, BARRIER x)
  SEQ
    P (x)       -- enrolled on 'x'
    RESIGN x
      A ()      -- not enrolled on (and cannot reference) 'x'
    R (x)       -- enrolled on 'x'
:
```

Whilst executing process 'P(x)', this 'worker' must synchronise on the barrier (or it will block other enrolled processes that *are* synchronising). However, whilst executing the RESIGN-block 'A()', it plays no part in the barrier and other enrolled processes can synchronise amongst themselves freely. After the RESIGN-block, it is back in the barrier.

Note that some care must be taken to avoid non-determinism after exit from a RESIGN-block, since the precise time of that exit *and consequent re-enrolment in the barrier* is scheduling dependent. This is considered further in Section 1.9 below.

1.6. Static Barriers: Usage Rules

Process *enrolment* on a barrier is determined by the scope of its declaration, PAR ENROLL compositions and RESIGN blocks. The following usage rules for barriers are enforced by compiler checks:

- a process may only reference a barrier (i.e. SYNC, RESIGN or pass to a procedure) if, and only if, it is *enrolled* on that barrier;
- *at most one* component process of a non-enrolling PAR remains enrolled on any barrier for which the PAR, as a whole, is enrolled;
- an individual barrier may be passed to *only one* parameter of a PROC. Strict anti-aliasing laws apply: *different* barrier names always refer to *different* barriers.

1.7. Parallel Enrolment with Multiple Barriers

We may enrol multiple barriers in the same PAR construct. In the following example, the '*.timer' processes controls the timing of 'process.a' and 'process.b' by synchronising on their respective barriers regularly (at 'long' or 'short' time intervals). Processes

'process.a' and 'process.b' (which may resign from either or both time-slicing controls from time to time) also use a private barrier, 'b', to synchronise between themselves:

```
BARRIER long, short:
PAR ENROLL long, short
  PAR
    long.timer (long)
    short.timer (short)
  BARRIER b:
  PAR ENROLL b, long, short
    process.a (long, short, b)
    process.b (long, short, b)
```

1.8. Parallel Enrolment, Termination and Auto-Resignation

Each component process of a PAR ENROLL construct resigns from its so-enrolled barrier(s) just before it terminates, apart from the *last* one to finish.

This means that all components do not have to terminate in the *same* barrier cycle to avoid deadlock (as would be the case if occam-π barriers were direct reflections of CSP multiway *events*). Consider the example given in section 1.2:

```
PAR i = 0 FOR n ENROLL b
  worker (i, b)     -- all enrolled on 'b'
```

Any worker process may terminate early, leaving its companion processes running and synchronising with each other successfully on 'b' — the early-terminated process has resigned from the barrier.

In CSP, termination of components of a parallel composition happens simultaneously. If one component is ready to terminate, it commits exclusively to that and refuses all other events. So, if other components have not terminated and continue to try to synchronise on a multiway event bound to that parallel, there would be deadlock.

For occam-π, we want to be able to build collections of processes disciplined by common synchronisation on barrier(s); but which do not have to be kept running and synchronising when their job is done, just so that they may terminate together. The chosen semantics give us this directly.

If we really need the raw CSP semantics, we just declare and enrol an extra barrier and synchronise on it *once*:

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

Now, when one worker terminates, its driving process commits to engage in 'alldone' and refuses 'b', on which it is still enrolled. All other worker processes must also terminate without further engagement on 'b' — else deadlock.

Another nice property from these occam-π semantics is that SKIP is a *unit* of all versions of its PAR operator:

```
PAR          =    PAR ENROLL b    =    P (b)
  P (b)             P (b)
  SKIP              SKIP
```

In the first system, 'b' must be a global barrier and SKIP is not enrolled. Hence, SKIP's existence and termination have no impact on the continuing operation of P(b).

In the second system, SKIP *is* enrolled on 'b'. Unless P(b) finishes first, all this SKIP does is resign from 'b' and wait to terminate. Otherwise, it just terminates (together with P(b)). If P(b) synchronises on 'b', it cannot finish first and blocks until the SKIP has resigned (which will happen) and, then, continues as normal. If/when P(b) terminates, it does so with the waiting SKIP. Either way, the SKIP has no impact and we are left with P(b).

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

1.9. Controlled Exit from Resign Blocks

A subtle problem can arise through the careless exit from RESIGN blocks. Consider:

```
PROC always (BARRIER a, b)
  WHILE TRUE
    SEQ
      SYNC a
      ...  phase A compute (no SYNCs)
      SYNC b
      ...  phase B compute (no SYNCs)
:

PROC sometimes (BARRIER a, b)
  WHILE TRUE
    SEQ
      SYNC a
      ...  phase A compute (no SYNCs)
      SYNC b
      ...  phase B compute (no SYNCs)
      IF
        ...  decide on a holiday
          RESIGN a, b
            ...  enjoy holiday (e.g. sleep)
        TRUE
          SKIP
:

PAR ENROLL a, b
  always (a, b)
  sometimes (a, b)
```

So long as 'sometimes' stays enrolled in its barriers, all goes well — 'sometimes' and 'always' will continue their respective phased computations in parallel, keeping in step with each other as each phase ends.

If 'sometimes' decides to go on holiday, it resigns from its barriers and does other things (like sleep), leaving 'always' to continue on its own — all is still well.

The problem arises if 'sometimes' decides to come back. When it exits its RESIGN block, it re-enrols on its barriers and waits to SYNC on 'a'. If 'always' is in its phase B when this happens, we are lucky and the two processes resume in perfect synchronisation. But if 'always' is in phase A, its next SYNC is on 'b' and the system will deadlock.

To do this safely, 'sometimes' must coordinate its return with 'always'. One way to do this is for 'sometimes' to request permission from 'always' to return to their joint compu-

tations. The ‘always’ process only grants this permission in its phase B and, then, waits for confirmation that ‘sometimes’ has re-enrolled (i.e. has left its RESIGN block).

This behaviour is easy to manage by signalling and polling over standard channels:

```
PROC sometimes (BARRIER a, b, CHAN BOOL signal!)
  WHILE TRUE
    SEQ
      SYNC a
      ...  phase A compute (no SYNCs)
      SYNC b
      ...  phase B compute (no SYNCs)
      IF
        ...  decide on a holiday
          SEQ
            RESIGN a, b
              SEQ
                ...  enjoy holiday
                signal ! TRUE       -- request comeback
            signal ! TRUE           -- confirm comeback
        TRUE
          SKIP
:

PROC always (BARRIER a, b, CHAN BOOL signal?)
  WHILE TRUE
    SEQ
      SYNC a
      ...  phase A compute (no SYNCs)
      SYNC b
      ...  phase B compute (no SYNCs)
      PRI ALT
        BOOL any:
        signal ? any                -- grant comeback
          signal ? any              -- wait for confirm
        SKIP
          SKIP
:
```

and where the system is now:

```
CHAN BOOL signal:
PAR ENROLL a, b
  always (a, b, signal?)
  sometimes (a, b, signal!)
```

In a larger system, there may be many processes, like ‘sometimes’, that retire from the computation from time to time. Examples arise in large scale simulations of dynamic systems, where not all processes need to be continually active (because nothing is changing in their neighbourhood) but need to rejoin some barrier synchronisation (e.g. for managing simulation ‘time’) when something happens close to them — see [17].

In such cases, the above *comeback/confirm* protocol may be used between each resigning process and just *one* specialised process, like the above ‘always’, that is *always* cycling and

synchronising (and which need do nothing else). Separate *comeback* and *confirm* channels will be needed, SHARED at the resigning process ends. We are considering language support for such a protocol. For example, the resigning processes execute:

```
RESIGN b
  ...  resign block (may not reference 'b', 'c' or 'd')
  RESUME c! d!
```

where 'c' and 'd' are SHARED CHAN BOOLs. In the correct phase, the resuming process executes:

```
RESUME c? b?
```

where this may be used as an ALT (or PRI ALT) guard.

1.10. Mobile Barriers: Declaration

Mobile barriers follow the same general rules for declaration, construction, communication and assignment as mobile channels and processes. The declaration introduces the variable name but leaves it *undefined*. Barrier variables become *defined* either through construction, communication or assignment. The compiler tracks *defined-status* and prevents use of undefined variables. Explicit run-time checks (using the DEFINED prefix operator) are forced for cases where the compiler cannot deduce the *defined-status*.

```
MOBILE BARRIER b:
...  process (initially, 'b' is undefined)
```

At the end of scope of a mobile barrier declaration, if the variable ended up as *defined*, the process automatically resigns from the referenced barrier.

1.11. Mobile Barriers: Construction

Construction and assignment to a mobile variable are bound together:

```
b := MOBILE BARRIER
```

where 'b' must be a MOBILE BARRIER variable. If 'b' were currently defined, the executing process would first resign from the currently referenced mobile barrier. After this statement, 'b' is now defined and references a *new* mobile barrier and the executing process is enrolled.

Note that a mobile barrier may be declared and constructed in one line with the standard (though, in this case, rather unusual looking) initialising declaration:

```
INITIAL MOBILE BARRIER b IS MOBILE BARRIER:
...  process (enrolled on 'b')
```

1.12. Mobile Barriers: Communication

Communication follows the same semantics for all occam-π mobiles: the item *moves* to the new place, leaving the source variable undefined. For mobile barriers, there are additional rules about enrolment and resignation.

In the following, 'a' and 'b' are MOBILE BARRIER variables, 'b' must be *defined* and 'c' is a CHAN MOBILE BARRIER (i.e. a channel carrying mobile barriers).

```
c ? a
```

If ‘a’ is defined, the receiving process first resigns from the held barrier and, then, receives the new reference. Otherwise, it just receives the new reference. Either way, it is now enrolled on the received barrier.

```
c ! b
```

This moves the barrier to another process, leaving ‘b’ undefined. This sending process resigns from the barrier. If we didn’t want to lose it, we must send a clone:

```
c ! CLONE b
```

In this case, the sending process remains enrolled on the barrier. All relevant *enrols* and *resigns* of the processes happen automatically and atomically with the communication.

1.13. Mobile Barriers: Assignment

Assignment follows the same mobility and enrol/resign rules. Again, suppose ‘a’ is a `MOBILE BARRIER` and ‘b’ is a *defined* `MOBILE BARRIER`.

```
a := b
```

If ‘a’ were defined, the process first resigns from that barrier. The barrier reference *moves* from ‘b’ to ‘a’ and the process remains enrolled on it. Such assignments cannot introduce aliasing. However:

```
a := CLONE b          -- this may get banned!
```

always introduces aliasing. As before, if ‘a’ were defined, the process resigns from that barrier. The barrier reference is *copied* from ‘b’ to ‘a’ — variables ‘a’ and ‘b’ now reference the same barrier and the process is enrolled twice!

This aliasing may not be as bad as it seems. For example, the code on the left below is safe and may serve some purpose:

```
SEQ                       PAR ENROLL b
  a := CLONE b              P (b)
  PAR                       Q (b)
    P (a)
    Q (b)
```

It is *almost* the same as the code on the right, but omits the auto-resignation semantics (see Section 1.8).

To remain compatible with the rest of occam-π and to satisfy our intuition, assignment and communication should be related by laws that, in these cases, take the form:

```
a := b           =      CHAN MOBILE BARRIER c:
                        PAR
                          c ? a
                          c ! b
```

and:

```
a := CLONE b     =     CHAN MOBILE BARRIER c:
                       PAR
                         c ? a
                         c ! CLONE b
```

We definitely need to allow the cloned output mechanism — so simply banning cloned assignments is not enough to prevent aliasing.

1.14. Mobile Barriers: Forking

Passing arguments to *forked* processes in occam-π means communicating them — see [3]. Hence, forked processes may take mobile barrier parameters. If 'b' is a defined mobile barrier, then:

```
FORK P (b)
```

moves the barrier to the new process. The forking process resigns from the barrier and 'b' becomes undefined. More usually, of course, the forking process retains the barrier (for passing to processes it may fork in the future) by passing a clone and remaining enrolled:

```
FORK P (CLONE b)
```

Either way, the forked process is enrolled on the barrier. Just before the forked process terminates, it automatically resigns from whatever barrier (if any) its parameter is referencing. We need this for the same reason that auto-resignation was specified for parallel enrolled processes (1.8).

Note that the forking process must be enrolled on the barrier to be able to pass it to its forked processes. This enables the release of forked processes in the correct phase of barrier synchronisation with existing processes holding that barrier. Enrolment of the forked process happens atomically with its forking.

1.15. Mobile Barriers: Synchronisation, Parallel Enrolment and Resign Blocks

Synchronisation, parallel enrolment, parallel non-enrolment and resign blocks for mobile barriers have the same syntax and semantics as those for static barriers.

The usual parallel usage rules for read/write access to variables apply to mobile barrier variables. A process enrolled on a mobile barrier is considered to have *read* access on the variable — i.e. its value cannot be changed in parallel. Marrying this with the usage rules for static barriers (Section 1.6), we note one extra rule:

- component processes in a `PAR ENROLL` construct whose bound barrier(s) is mobile may not change the held reference (e.g. by assignment, input or non-cloned output).

That also means that such component processes may only pass a `PAR-ENROLL`-bound *mobile* barrier to a *static* barrier parameter/abbreviation.

2. A CSP Model for Mobile and Static Barriers

Our original approach was to model occam-π barriers directly as CSP multiway events. The dynamics of mobility, resignation and enrolment was to be handled with auxiliary *spinner* processes, interleaving with the application processes on the barriers and taking over synchronisation on them when the application process was not enrolled. This worked well for *static* barriers, but managing the infinite sets of *spinners* needed to explain *mobile* barriers was proving troublesome (and would be hard for model checkers to accommodate).

The approach presented here models each barrier as a *process*, rather than an event. It documents how they are supported by the occam-π kernel. It captures all the dynamic semantics of occam-π mobile barriers: run-time construction, communication and assignment, cloning, parallel enrolment and non-enrolment, termination resignation and resign blocks, and passing as arguments to forked processes.

Initially, we consider mobile barriers — the model for static barriers then follows trivially. A formal semantics for occam-π barriers then derives from the semantics of CSP.

2.1. Modelling an occam-π Mobile Barrier with a Process and Shared Channels

The insight is to give up trying to model these dynamic barriers directly with CSP events and spinner processes (maintaining synchronisation when their buddy application processes disengage). Instead, we model mobile barriers with processes and shared channels, but with added flexibility for the dynamic enrolment and resignation of processes.

So, occam-π mobile barrier variables become (mobile) integer variables, holding indices to the actual barriers. The latter are (kernel) processes, running in parallel to all application processes, and created dynamically as needed. This means that they are always accessible to all application processes, even though they are triggered within individual ones. So, we don't require the awkward *scope extrusion* concept of the π-calculus.

Table 1. Mobile barrier process fields

Field	Name	Purpose
b	index	identification — unique for each barrier
refs	reference count	the number of mobile barrier variables currently holding a reference to '*b*'
n	enrolled count	the number of processes currently enrolled on '*b*'
count	sync count	the number of processes still left to synchronise on '*b*' (to complete the barrier)

Table 2. Mobile barrier process events

Event	Purpose
enrol.b.p	enrol '*p*' processes on barrier '*b*'
resign.b	resign one process from barrier '*b*'
tresign.b	temporarily resign one process from barrier '*b*' ('`RESIGN`' block)'
tenrol.b	re-enrol one (temporarily resigned) process on barrier '*b*'
sync.b	offer (committed) to synchronise on barrier '*b*'
ack.b	complete synchronisation on barrier '*b*'

A mobile barrier process has four integer fields — shown in Table 1. System constraints will impose that $(b > 0)$ and $(refs \geqslant n \geqslant count \geqslant 0)$. Index zero is reserved for mobile barrier variables currently undefined — this is just for convenience in the following model (not strictly necessary). The mobile barrier process with index '*b*' engages on the events described in Table 2. Here is the process:

$$
\begin{aligned}
&BAR\,(b, refs, n, count) = \\
&\qquad \big(enrol.b.p \rightarrow BAR\,(b,\ refs + p,\ n + p,\ count + p)\big)\ \Box \\
&\qquad \big(resign.b \rightarrow BAR\,(b,\ refs - 1,\ n - 1,\ count - 1)\big)\ \Box \\
&\qquad \big(tresign.b \rightarrow BAR\,(b,\ refs,\ \ n - 1,\ count - 1)\big)\ \Box \\
&\qquad \big(tenrol.b \rightarrow BAR\,(b,\ refs,\ \ n + 1,\ count + 1)\big)\ \Box \\
&\qquad \big(sync.b \rightarrow BAR\,(b,\ refs,\ \ n,\ \ count - 1)\big), \qquad \text{if } (count > 0) \\
&BAR\,(b, refs, n, 0) = BAR\_ACK\,(b,\ refs,\ n,\ 0), \qquad \text{if } (n > 0) \\
&BAR\_ACK\,(b, refs, n, count) = \\
&\qquad ack.b \rightarrow BAR\_ACK\,(b,\ refs,\ n,\ count + 1), \qquad \text{if } (n > count)
\end{aligned}
$$

$$BAR\_ACK\,(b, refs, n, n) = BAR\,(b,\ refs,\ n,\ n)$$
$$BAR\,(b, refs, 0, 0) = tenrol.b \rightarrow BAR\,(b,\ refs,\ 1,\ 1), \qquad \text{if } (refs > 0)$$
$$BAR\,(b, 0, 0, 0) = SKIP$$

The difference between '*resign*.*b*' and '*tresign*.*b*' is that the latter does not decrement the reference count. There is a similar difference between '*enrol*.*b*.1' and '*tenrol*.*b*'. '*tresign*.*b*' and '*tenrol*.*b*' will be used to bracket `RESIGN` blocks, whose existence is the only reason that reference and enrolled counts may differ.

`SYNC` operations, in application processes, map to a sequence of a '*sync*.*b*' immediately followed by an '*ack*.*b*'. The former just decrements the synchronisation count. If that reaches zero, the barrier process locks into a sequence of '*ack*.*b*' events with length equal to the current enrolled count — these will all succeed, since there will be precisely that number of application processes blocked and waiting for them. *Note:* application processes *interleave* amongst themselves for engagement on all these barrier process control events.

Any '*resign*.*b*' event that reduces the reference count to zero will also, given the earlier constraint, have reduced the enrolled and synchronisation counts to zero — in which case, the barrier process simple terminates. Note that '*tresign*.*b*' does not change the reference count and, so, cannot reduce it to zero.

2.2. Kernel and Application Processes

The mobile barrier processes are *forked* off as needed by a generator process:

$$MB\,(b) = \big(getMB!b \rightarrow \big(BAR\,(b, 1, 1, 1) \,|||\, MB\,(b+1)\big)\big) \,\Box\, \big(noMoreBarriers \rightarrow SKIP\big)$$

For convenience, we also define:

$$UNDEFINED\_BAR = \big(resign.{}_0 \rightarrow UNDEFINED\_BAR\big) \,\Box\, \big(noMoreBarriers \rightarrow SKIP\big)$$

Now, if `SYSTEM` is the occam-π application and *SYSTEM*′ is the CSP modelling of its mobile barrier primitives (see below), the full model is:

$$\big(\,(SYSTEM' \,\fatsemi\, noMoreBarriers \rightarrow SKIP) \;\underset{\{kernelchans\}}{\big\|}\; MobileBarrierKernel\,\big) \setminus kernelchans$$

where:

$$MobileBarrierKernel = MB\,(1) \;\underset{\{noMoreBarriers\}}{\big\|}\; UNDEFINED\_BAR$$

and:

$$kernelchans = \big\{enrol.b.p, resign.b, tresign.b, tenrol.b, sync.b, ack.b,$$
$$getMB, noMoreBarriers \mid (b \geqslant 0), (p \geqslant 1)\big\}$$

2.3. Extending CSP with Variables and Assignment

For making precise the semantics of mobile barriers, we shall be using the syntax of Circus [18]. This introduces, amongst other things, variables and assignment into CSP. It allows us to work at a slightly higher, and clearer, level than pure CSP.

Such variables and assignments could be removed by introducing parallel terminatable state-processes for each variable, whose duration matches their scope; plus '*load*', '*store*' and '*kill*' channels for reading and writing their values and for termination. For example, the variable declaration and process:

$$\mathsf{Var}\ x : \mathbb{N} \bullet P$$

becomes:

$$\left((P' \fatsemi kill_X \rightarrow SKIP) \underset{\{load_X, store_X, kill_X\}}{\left\|\right\|} Var_X\right) \setminus \{load_X, store_X, kill_X\}$$

where:

$$Var_X(x) = \left(load_X!x \rightarrow Var_X(x)\right) \Box \left(store_X?tmp \rightarrow Var_X(tmp)\right) \Box \left(kill_X \rightarrow SKIP\right)$$

and P' is the result of removing similar variables from P. An assignment process:

$$x := y$$

becomes:

$$load_Y?tmp \rightarrow store_X!tmp \rightarrow SKIP$$

Any expression involving such variables requires prefixing with a sequence of *loads* into separate registers. For example:

$$c!(x + y)$$

becomes:

$$\left(load_X?tmp_0 \rightarrow load_Y?tmp_1 \rightarrow c!(tmp_0 + tmp_1) \rightarrow SKIP\right) \Box$$
$$\left(load_Y?tmp_1 \rightarrow load_X?tmp_0 \rightarrow c!(tmp_0 + tmp_1) \rightarrow SKIP\right)$$

All occam-π variables — including those for mobile barriers — map to such Circus variables. When reasoning formally about such CSP mappings, we should also take into account that occam-π processes are bound by its *parallel usage rules*. These need formalising.

2.4. Modelling the occam-π Primitives for Mobile Barriers

2.4.1. Mobile Barrier Declaration

Mobile barrier variables map into mobile integer (actually *natural number*) variables, holding *indices* to the referenced barrier processes:

```
MOBILE BARRIER b:
P
```
$$\rightsquigarrow \quad \mathsf{Var}\ b : \mathbb{N} \bullet b := \mathsf{undefined} \fatsemi P' \fatsemi resign.b \rightarrow SKIP$$

where undefined is zero and P' is the CSP model of P. Note that if 'b' is undefined when P' terminates, the '*resign*.b' is swallowed harmlessly by the *UNDEFINED_BAR kernel* process.

2.4.2. Mobile Barrier Construction

```
b := MOBILE BARRIER
```
$$\rightsquigarrow \quad getMB?tmp \rightarrow (b := tmp)$$

2.4.3. Mobile Barrier Synchronisation

`SYNC b` $\quad\rightsquigarrow\quad sync.b \rightarrow ack.b \rightarrow SKIP$

2.4.4. Mobile Barrier Send (Uncloned)

`c ! b` $\quad\rightsquigarrow\quad c!b \rightarrow (b := \mathsf{undefined})$

2.4.5. Mobile Barrier Send (Cloned)

`c ! CLONE b` $\quad\rightsquigarrow\quad enrol.b.1 \rightarrow c!b \rightarrow SKIP$

2.4.6. Mobile Barrier Receive

`c ? b` $\quad\rightsquigarrow\quad c?tmp \rightarrow resign.b \rightarrow (b := tmp)$

2.4.7. Mobile Barrier Assign (Uncloned)

`a := b` $\quad\rightsquigarrow\quad resign._a \rightarrow (a := b) \rightarrow (b := \mathsf{undefined})$

2.4.8. Mobile Barrier Assign (Cloned)

`a := CLONE b` $\quad\rightsquigarrow\quad \big((enrol.b.1 \rightarrow SKIP) \,|||\, (resign._a \rightarrow SKIP)\big) \fatsemi (a := b)$

2.4.9. Mobile Barrier Resign Block (Uncontrolled Resume)

```
RESIGN b
  P
```
$\rightsquigarrow\quad tresign.b \rightarrow P' \fatsemi tenrol.b \rightarrow SKIP$

2.4.10. Mobile Barrier Resign Block (Controlled Resume)

```
RESIGN b
  P
  RESUME c! d!
```
$\rightsquigarrow\quad tresign.b \rightarrow P' \fatsemi c \rightarrow tenrol.b \rightarrow d \rightarrow SKIP$

To coordinate resumption in the right *phase*, the resuming process should be enrolled on '`b`'. It executes:

`RESUME c? d?` $\quad\rightsquigarrow\quad c \rightarrow d \rightarrow SKIP$

Note: one resuming process can manage many resign-block processes. The latter interleave amongst themselves on the '`c`' and '`d`' channels, but synchronise on them with the former. We call them 'channels' since only two-way synchronisation is involved. No values are communicated over them.

2.4.11. Mobile Barrier Parallel Enrolment

```
PAR i = start FOR n ENROLL b
  P (i, b)
```

$$\rightsquigarrow\quad ParCount\,(n) \;\Big\|_{\{down\}}\; \Big(enrol.b.(n-1) \rightarrow \Big|||_{i\,=\,start}^{start\,+\,(n-1)} \big(P'\,(i,\,b) \fatsemi down?n \rightarrow \big(SKIP \blacktriangleleft (n{=}0) \blacktriangleright resign.b \rightarrow SKIP\big)\big)\Big)$$

where $P'\,(i,\,b)$ is the CSP model of `P (i, b)` and:

$$ParCount\,(n) = down!(n-1) \rightarrow ParCount\,(n-1), \qquad \text{if } (n > 0)$$

$$ParCount\,(0) = SKIP$$

The usual occam-π parallel usage rules apply for the barrier variable '`b`' here. So, the replicated process may use '`b`' but may not change it. All it may do is `SYNC` on it, `RESIGN` from it and release `CLONE`s.

Note that this captures the required semantics (Section 1.8) that each component process of the `PAR ENROLL` resigns from the barrier as it terminates, apart from the last one to finish.

2.4.12. Mobile Barrier Parallel Non-Enrolment

No special semantics are needed in this case: the parallel just maps to a CSP parallel construction. The occam-π parallel usage rules apply — i.e. only (at most) one of the component processes may change the barrier variable. However, occam-π imposes a stricter constraint: only (at most) one of the component processes may reference the barrier at all (i.e. `SYNC` on it, `RESIGN` from it, `CLONE` it, change it).

2.4.13. Mobile Barrier Passing to a Forked Process

$$\texttt{FORK P (b)} \quad \rightsquigarrow \quad \mathit{forkP}!b \rightarrow (b := \mathsf{undefined})$$

where '*forkP*' is a channel specific for forking instances of `P`.

More usually, of course, the forking process retains the barrier (for passing to processes it may fork in the future) by passing a clone and remaining enrolled:

$$\texttt{FORK P (CLONE b)} \quad \rightsquigarrow \quad \mathit{enrol}.b.1 \rightarrow \mathit{forkP}!b \rightarrow \mathit{SKIP}$$

Note that, either way, synchronisation on the barrier referenced by '`b`' cannot afterwards complete without participation by the forked process (e.g. by synchronisation or resignation).

To fork a process, we must be running in a `FORKING` block (which, by default, is the whole system). An explicit such block, that forks only instances of `P(b)` for some mobile barrier variable '`b`':

$$\texttt{FORKING X} \quad \rightsquigarrow \quad \Big((X' \fatsemi \mathit{done} \rightarrow \mathit{SKIP}) \underset{\{\mathit{forkP},\mathit{done}\}}{\Big\|} \mathit{ForkP}\Big) \setminus \{\mathit{forkP}, \mathit{done}\}$$

where X' is the CSP model of `X`, '*done*' is chosen so that it does not occur *free* in `X` or `P(b)`, and:

$$\mathit{ForkP} = \Big(\mathit{forkP}?b \rightarrow \big((P'(b) \fatsemi \mathit{resign}.b \rightarrow \mathit{done} \rightarrow \mathit{SKIP}) \underset{\{\mathit{done}\}}{\Big\|} \mathit{ForkP}\big)\Big)$$

$$\Box$$

$$\Big(\mathit{done} \rightarrow \mathit{SKIP}\Big)$$

and $P'(b)$ is the CSP model of `P(b)`.

Note that forked processes — like components of a `PAR ENROLL` construct — resign from whatever barriers (if any) are referenced by their parameters as they terminate. Note also that termination of the forking block waits for all forked processes to terminate.

2.5. Modelling the occam-π Primitives for Static Barriers

The semantics of *static* barriers did work out with the spinner mechanism previously considered. However, static barriers can always be replaced by mobile barriers that take no advantage of their mobility (i.e. communication and assignment). So, we may as well go with these new semantics!

To transform static barriers into mobiles, their declarations:

```
BARRIER b:
```

simply become the combined mobile declaration and initialisation:

```
INITIAL MOBILE BARRIER b IS MOBILE BARRIER:
```

All BARRIER parameters/abbreviations become MOBILE BARRIERs. No other transformations are needed, so we have their semantics.

Note: with static barriers, all we can do is synchronise, parallel enrol and resign block. If that is sufficient, use them rather than mobiles. There can be no aliasing problems with static barriers and their run-time overheads (memory and execution) are slightly lower.

3. Implementation and Benchmarking

Implementation follows all the mechanisms documented in the formal semantics given in Section 2. However, scheduling of the barrier processes is automatically serialised with in-line instructions generated by the occam-π compiler, supported by its kernel — no actual processes or channels are introduced.

Each barrier is managed though just five words of memory: three for the *reference*, *enrolled* and *synchronisation* counts (see Section 2.1) and two holding the front and back pointers to a *queue* holding processes blocked on the barrier. Barrier variables hold the start address (*index*) of this structure. For *mobile* barriers, the space is allocated dynamically in occam-π *mobile-space* (see [19]); for *static* barriers, the space lives on the stack of the declaring process.

A process synchronising on a barrier, unless the last to synchronise, is held on the barrier queue (rather than on an '*ack.b*' channel) and the next process is scheduled. A process completing a barrier (i.e. reducing the synchronisation count to zero) releases all the others — this is done in unit time by simply appending the barrier queue to the run queue, leaving the former empty. All adjustments to the barrier *counts* follow the rules defined in Sections 2.1 and 2.4 for modelling all the occam-π primitives in CSP.

Figure 2 shows the results of a benchmark that measures the time per barrier synchronisation for increasing numbers of concurrent processes, run on 3.2 GHz Pentium IV machines. Each process synchronises a fixed number of times, from which the average individual synchronisation time is calculated. A *stride* length is used to control the start-up (and subsequent scheduling) order of parallel sub-processes, demonstrating the effect of the processor's cache pre-fetching. Each curve in the figure reflects a different stride.

The memory foot-print for the 16 million process benchmark (actually 2^{24}) was just over 700 mega-bytes (approximately 44 bytes per process), so cache-misses will be heavy. The processes are allocated their workspaces contiguously according to their index. The *stride* forces their scheduling so that consecutively run process workspaces are (44**stride*) bytes apart. For small strides, the Pentium IV cache pre-fetching eliminates the problem of cache miss. For larger *stride*s, and especially for the *randomised* striding, the pre-fetching is defeated and cache miss penalties are felt.

Despite this, Figure 2 shows the implementation to be ultra-lightweight. The time for a *sixteen-million-wide* barrier synchronisation was only 16 ns per process in the best case (163 ms for the whole barrier) and 247 ns per process in the worst case. Typical application mixes will show *some* coherence in memory usage — the worst case above is really cruel! Also, applications running *real* processes (with real work to do) will not be able to afford more than the order of a million of them (because of memory limitations with current technology). The barrier mechanisms presented in this paper are useful and fast.

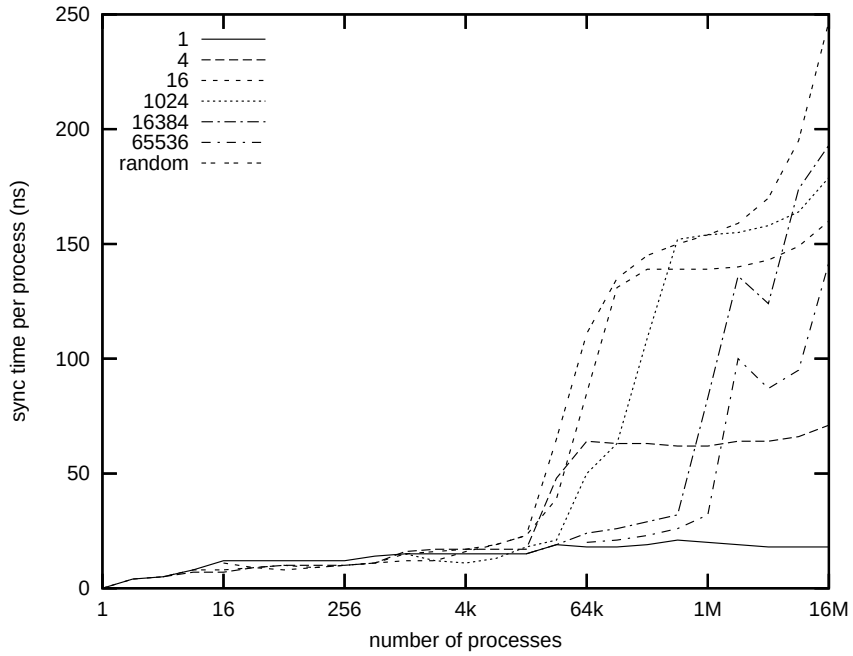

Figure 2. Synchronisation time for different strides

4. Sample Applications

4.1. The TUNA Project

This work binding barrier synchronisation safely and efficiently into the occam-π language was prompted by needs for TUNA (*Theory Underpinning Nanite Assemblers*) [13], a project involving researchers from the Universities of York, Surrey and Kent in the United Kingdom. This is investigating the emergent properties of systems containing millions of interacting agents — such as *nanites* or biological *organelles*. Here, goals are achieved by emergent behaviour from force of numbers, not by complicated programming or external direction. Such systems are complex, but not complicated. Medium term aims are the development of sufficient theory to enable the design of self-assembling nanite systems with controlled and predictable properties for application in human medicine.

A working case study looks at mechanisms of blood clotting. The model is loosely based on the medical process of *haemostasis*. *Platelets* are passive quasi-cells carried in the bloodstream. A platelet becomes active when a balance of chemical stimulators and suppressants changes in favour of activation, usually because of physical damage to the linings of blood vessels. Activated platelets become sticky, form clusters that restrict blood flow — a necessary first phase in limiting blood loss, healing of the wound and recovery.

Unlike systems developed for traditional embedded and parallel supercomputing applications, TUNA networks will be highly dynamic — with elements, such as channels and processes, growing and decaying in reaction to environmental pressures. Computational network topologies continually evolve as the organelles/nanites replicate, combine and decay.

To model more directly (and, hence, simply) the underlying biological/mechanical interactions, extremely fine-grained concurrency will be used. Complex behaviour will be obtained not by direct programming of individual process types, but by allowing maximum flexibility for self-organisation following encounters between mobile processes — randomised modulo physical constraints imposed by their modelled environments. We will need to develop location awareness for the lowest level processes, so they may be aware of other processes in their neighbourhood and what they have to offer. We will need to synchronise the development of organisms to maintain a common awareness of time.

Barrier mechanisms with user-defined and dynamic binding to processes are promising to be very helpful in this context.

4.2. Static Barrier Application: First Blood Clotting Model (Busy)

The clotting model and implementation described here are a gross simplification of what we will eventually require for TUNA. It is crucial, however, that we have a firm understanding and confidence in simple models, before attempting more elaborate models. We would not wish for any emergent behaviour of the system to be wholly determined by implementation-specific artifacts, such as programming errors arising from a lack of understanding.

Space is modelled as a one-dimensional pipeline of '`cell`' processes representing a section of a blood vessel. *Platelets* are in their activated (i.e. sticky) state. They flow through the cells at (average) 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, forming larger clots spanning many cells. Each cell maintains internal state indicating whether it contains a platelet. The model is time-stepped by having the cells synchronise on a barrier [8], which is also used to coordinate safe access to shared data.

4.2.1. System Network and Two-Phased Cycles

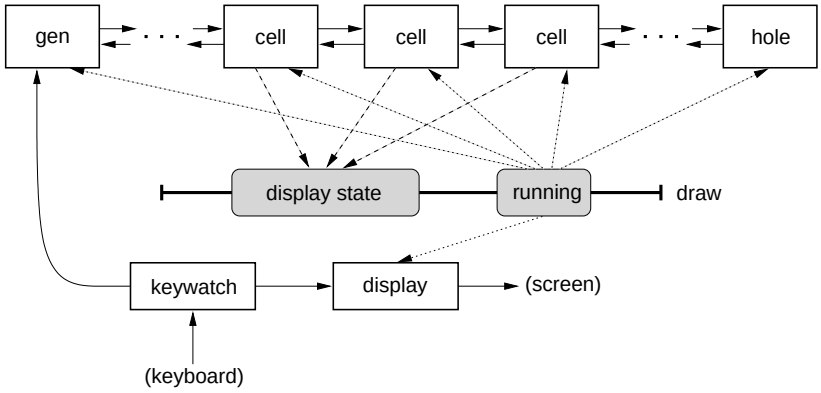

Figure 3. *'Busy'* clotting model process network (phase 0)

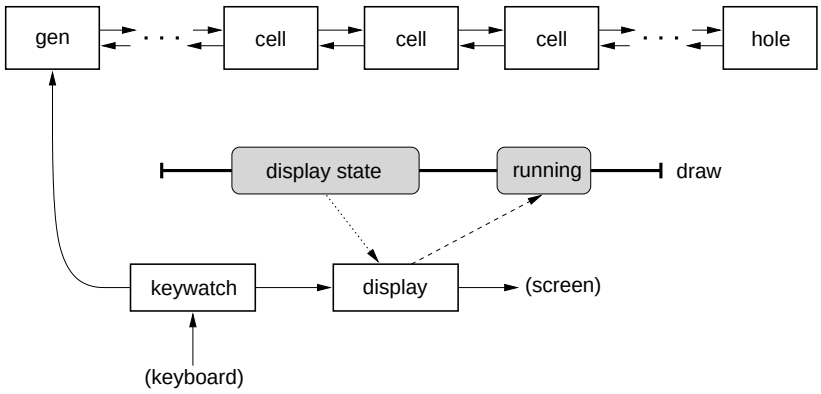

Figure 4. *'Busy'* clotting model process network (phase 1)

Figures 3 and 4 shows the two computational phases of the process network used in this clotting model. The '`generator`' process determines (stochastically) whether a new platelet is generated and, if so, injects it. The '`hole`' process just acts as a *sink* for platelets flowing out of the pipeline. The '`display`' process renders the (full or empty) state of the cells for visualisation and shows system parameters (such as platelet generation and display rates). The

'keywatch' process allows user-interaction for setting those parameters and for terminating the system.

The 'display state' and 'running' flag are not actually processes, but variables *shared* between the 'cell' and 'display' processes. (Such variables could, of course, be made into processes if we were worried about this — see Section 2.3).

Figures 3 and 4 extends the symbology of Figure 1. The shaded rounded boxes represent state variables. They are stuck on the barrier, 'draw', to indicate that access to them is controlled through the barrier. The dotted arrows between the processes and the shared variables indicate two things: reading/writing (depending on the arrow direction) and that the processes must synchronise on the underlying barrier to coordinate that reading or writing.

Race hazards to shared memory (and consequential loss of control) are avoided normally by occam-π's parallel usage rules, which enforce CREW (*Concurrent Read Exclusive Write*) principles. However, these apply between component processes of a PAR or between a FORKed process and the rest of the system. Here, we need a finer granularity of enforcement and this is managed through the 'draw' barrier.

All 'cell' processes together with 'generator', 'hole' and 'display' cycle through two phases, synchronised by the 'draw' barrier on which they are enrolled. To check CREW conformance, we just have to check that no read/write or write/write on shared state happens in the same phase. In this system, different components of the 'display state' are written by the cells in *phase 1*; they are read by the rendering 'display' process in *phase 0*. The 'running' flag is read by all enrolled processes in *phase 0* and written, by 'display', in *phase 1*.

4.2.2. The 'cell' Process

Here is outline code for the 'cell'. The first two *reference* data parameters give this process access to its component of the 'display state' (shared with the 'display' process) and the 'running' flag (shared with most other processes):

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

  ...  local declarations / initialisations  (phase 0)
  WHILE running
    SEQ

      SYNC draw    -- phase 1
      ...  PAR-I/O exchange of full/empty state with neighbour cells
      ...  if full
      ...    discover clot size (initiate or pass on count)
      ...    if head of clot
      ...      decide on move (non-deterministic choice)
      ...      if move, tell empty cell ahead (push decision)
      ...    else receive decision from cell ahead (pull decision)
      ...    if not tail of clot, pass movement decision back (pull)
      ...    if tail and movement, become empty
      ...  else if clot behind exists and moves (push), become full

      SYNC draw    -- phase 0
      ...  update my.visible.state

:
```

The 'CELL.CELL' protocol used for communication between cells is defined with:

```
PROTOCOL CELL.CELL
  CASE
    state; BOOL   -- full/empty
    push;  BOOL   -- move/no-move decision
    pull;  BOOL   -- move/no-move decision
    size;  INT    -- clot size
:
```

The barrier synchronisation forces all enrolled processes to start their *phase 1* computations together. The I/O-PAR communications of state between the 'cell's, which only use the above 'state' variant, cannot introduce deadlock [20].

After that, each cell knows the state of its immediate neighbours and works out what further communications, using the other variants of the 'CELL.CELL' protocol, are needed. All cells follow the same rules and reach matching decisions about those communications — so there can be no deadlock, despite this part of the logic not being I/O-PAR.

The 'generator' and 'hole' processes are cut-down versions of the 'cell'. Additionally, 'generator' polls its input channel from 'keywatch' for user-updates to the generation rate and makes decisions, based on that rate, for releasing new platelets (which it does by appearing *empty* or *full* to the first 'cell' process).

The 'keywatch' process is lazy and not enrolled on the barrier. It is triggered solely by user keystrokes.

It is worth noting that the movement decisions (by a 'cell' process at the head of a *clot*) and the new platelet release decisions (by the 'generator') are the *only* places in the system where non-determinism occurs (modelled in CSP as an *internal* choice). The 'cell' processes do not even contain a single ALT construct.

4.2.3. Scaling Up

In this system, every cell is always active, regardless of whether it contains a platelet — it is a classic *busy* Cellular Automaton (CA). It works well for systems with the order of hundreds of thousands of cells. For TUNA, we will need to be working in three dimensions, modelling many different types of agent all with much richer rules of engagement. To enable scaling up two (and more) orders of magnitude, these automata must become *lazy*, whereby only processes with things to do remain in the computation. One technique for achieving this are given in the next section; another is reported in [17].

4.3. Mobile Barrier Application: Second Blood Clotting Model (Lazy)

Something unsatisfactory about the CA approach described in the previous section is that the logic focusses on the *cell* processes. The rules for different stages in the life cycle of platelets or clots are coded into different cycles of the cells. From the point of view of the cell, which is what we design and program, we see lots of *different* platelets — sometimes bunched together forming clots — passing through. No process models the development of an individual clot.

4.3.1. Mobile Barriers, Mobile Channels and Forking

This model focusses on the life cycle of clots, each one directly represented by a 'clot' process. Initially, these are *forked* off by the 'generator' process as singleton platelets, straggling the first cell in the pipeline. Because these 'clot's need enrolment on the barrier,

the barrier must be passed to it by the '`generator`'. Because passing arguments to forked processes involves communication, the barrier must be a *mobile*.

As before, space is represented by the pipeline of '`cell`' processes — but this time they are not enrolled on the barrier. These cell processes are passive servers, responding to client requests on their *service channel bundles* — represented in Figures 5-10 by the vertical bi-directional channels on the top of the cells. Neighbourhood topology is determined by each cell's (shared) access to the next cell's service channels. Because we only support forward clot movements in this model, a cell only needs forward access — it would be easy to make connections in both directions should other models need this.

Cells hold state indicating whether they are being straddled by a passing clot; this state is shared with the '`display`' process. They are idle except when the front and rear boundaries of a clot passes through them.

Each '`clot`' process connects *feeler* channels to the cells immediately before and after the group of cells currently straddled — see the figures. It also connects to the *last* cell in its group, in which it deposits the *writing end* of its *tail-channel* — that deposition is not shown in the figures, but left free standing for clarity. All channels, apart from those connecting '`keywatch`' and the '`generator`' and '`display`' processes, are *mobile*.

The cell processes are shown underlain by the '`draw`' barrier. This means that processes connected to them (i.e. the clots and the display) must be enrolled on that barrier and coordinate their interaction with the cells through synchronisation on the barrier.

4.3.2. Computation Phase 0

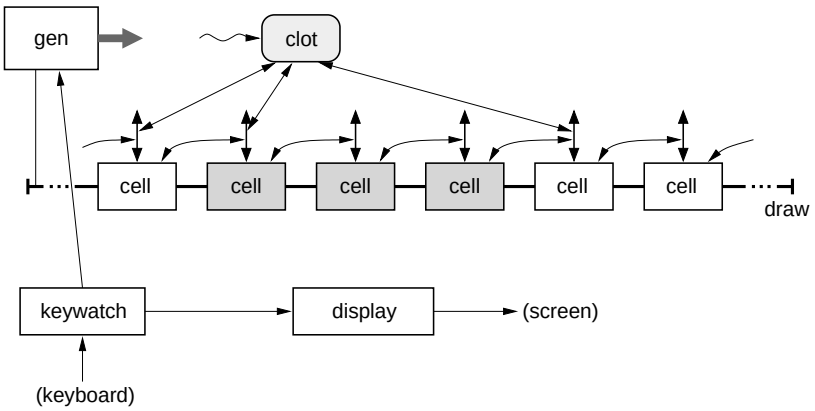

Figure 5. *'Lazy'* clotting model — before move (phase 0)

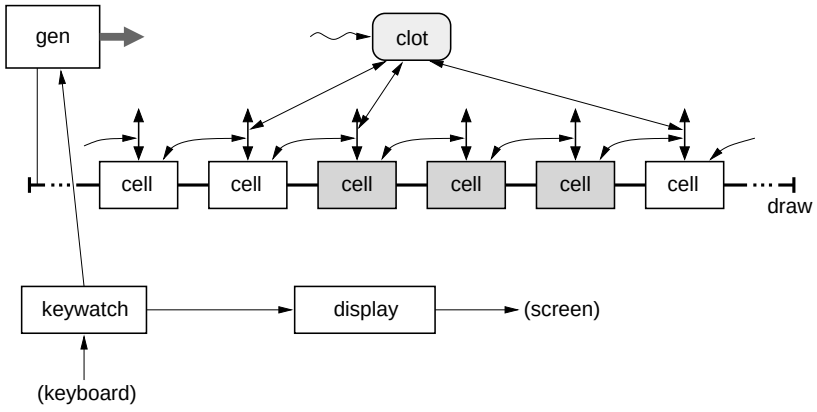

Figure 6. *'Lazy'* clotting model — after move (phase 0)

Through barrier synchronisation, we maintain the following invariant at the start of *phase 0* of each cycle: for each clot in the system, there are empty cells on either side of the (full) cells in the clot. This condition is shown in Figure 5. The computation proceeds by deciding and, if positive, moving the clot forwards by one cell — Figure 6. This requires communicating the *client-ends* of the cell service channel-bundles through the existing connections of the clot process, updating those connections accordingly, dragging the clot's tail forward one cell, marking the old rear cell empty and the new front one full. This all happens in *phase 0*, during which the '`display`' process is not reading the cell states (maintaining CREW rules).

4.3.3. Computation Phase 1

Following another barrier synchronisation, we are in *phase 1*. The invariant here is that no clots are moving. This allows them to inspect their environment — *location awareness* — by interrogating through their front and rear *feelers*. If other clots are detected, the bumping clots coalesce — Figures 7-10.

In Figure 7, two clots detect that they have touched. The left one, using its front *feeler*, acquires the writing end of the tail-channel of the one on the right (which was deposited in the cell probed by that *feeler*). The two clot processes have dynamically set up a connection between them — Figure 8.

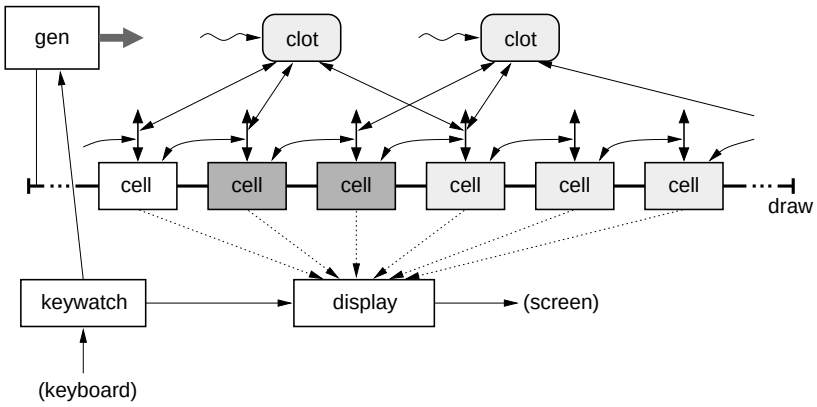

Figure 7. *'Lazy'* clotting model — bump detected (phase 1)

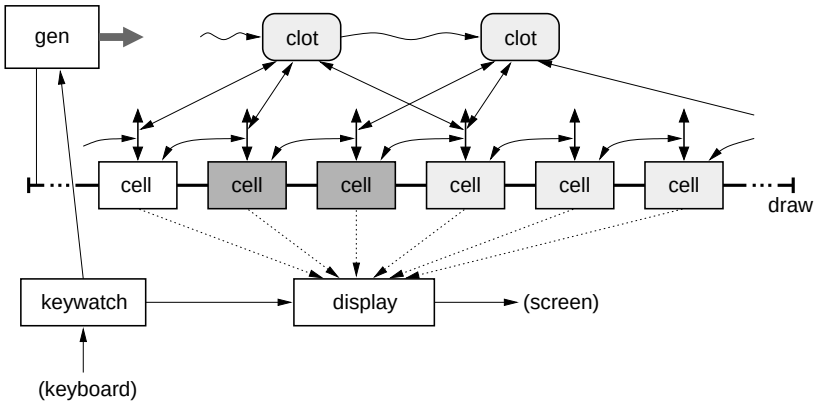

Figure 8. *'Lazy'* clotting model — communication established (phase 1)

The left clot communicates four items: its size, the reading end of its tail-channel and the client ends of its rear *feeler* and last clot cell services. The right clot increments its size accordingly and overwrites its corresponding connections with the three channel/bundle-ends received — Figure 9. Finally, the left clot terminates, the right clot having taken over the merger — Figure 10.

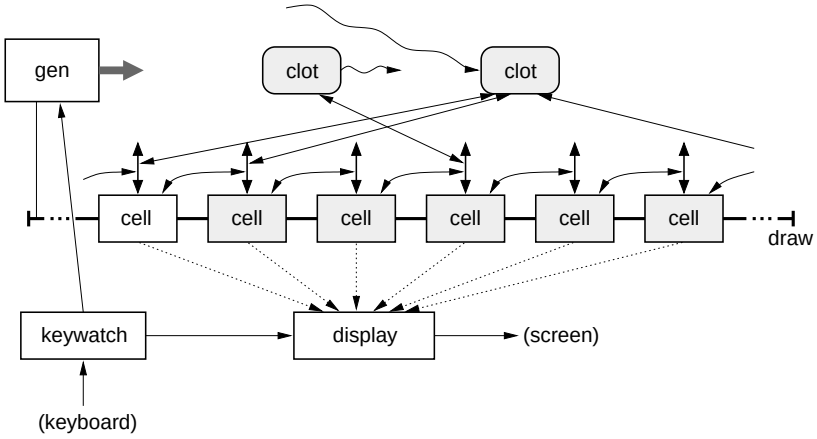

Figure 9. *'Lazy'* clotting model — tail and back legs passed (phase 1)

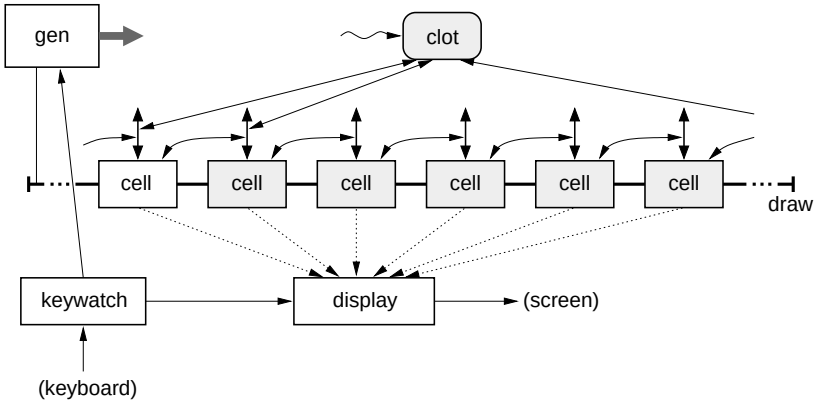

Figure 10. *'Lazy'* clotting model — clots merged, rear one terminated (phase 1)

During this phase, the (full or empty) state of the cells do not change and it is safe for the '`display`' process to read and render them.

Not shown in these figures is a shared '`running`' flag, operated across the phases in the same way as for the previous model — Section 4.2. Terminating the cell processes cannot be via this `running`' flag, since they are not enrolled on the barrier and have no way, safely, to read its value and ensure that all read it in the same cycle. Instead, termination has to be done in the classical way, using a *poison* message sent through the pipeline — see [21].

4.4. Performance of the Models

For the *'busy'* cellular automata of Section 4.2, performance is proportional to the number of cells since they are all active all the time. It also depends on the number of platelets in the system, since cells holding platelets have additional work to do. Further, clot sizes are recomputed every cycle — so large clumps also increase the cost.

For the *'lazy'* but dynamic system of Section 4.3, the number of cells only impacts on memory requirements — though that may cause cache-miss problems at run-time. Otherwise, its performance depends only on the number of clots in the system — their size (i.e. the number of platelets) is irrelevant.

Table 3 gives the cycle times per cell for systems of around 10K cells, running on a 2.4 GHz Pentium 4-m. The number of platelets in the system depends on the generation rate — these are given in the first column as fractions of 256 and represent the probability of release in each cycle. Each run, of course, has different properties but the overall performance does not change much. These results are averaged over 10 runs for each model and for each generation rate.

Table 3. Cell cycle times for the two models

Generation Rate (n / 256)	*'Busy'* (ns)	*'Lazy'* (ns)
0	650	0
1	660	8
2	670	12
4	680	14
8	700	16
16	740	18
32	1070	0

A generation rate of zero implies no work is done by the *'lazy'* model. A generation rate of 32/256 is too much for the bloodstream and causes a total jam, with the vessel containing one continuous clot. This causes extra work for the *'busy'* model, computing its length each cycle — as well as cycling all processes. For the *'lazy'* model, there is again nothing to do.

On balance, the *'lazy'* model is more than 40 times faster than the *'busy'* cellular automaton — in some circumstances, it is infinitely more efficient. Its logic is also simpler, more directly modelling the players in the system.

4.5. Emergent Behaviour

The clotting model presented here is particularly simple. It has been developed to try out techniques that need to be matured before the *real* modelling can be attempted. Nevertheless, unprogrammed behaviour has *emerged* that is encouraging and relevant to our TUNA investigations.

Considering the 1-dimensional pipeline as a capillary in the blood circulation system, these results reflect certain observed realities. Above a certain probability of platelet activation (resulting, initially, from tissue damage) and length, such a capillary always becomes blocked.

Figure 11 shows a screen-shot of a visualisation for a $100*50$ cell grid (arranged as a 1-dimensional *pipe*) using 16 pixels-per-cell and with a 4/256 probability of clot platelet generation at the start of the *pipe* (top-left in the picture).

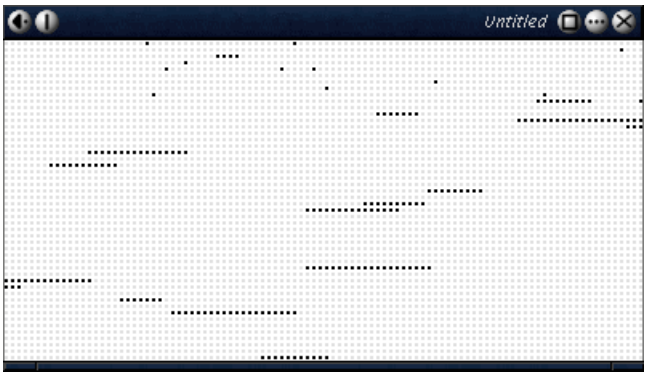

Figure 11. Clot model visualisation

The pipeline is displayed *snaking* down the image, with the first cell at the top-left, the next cells moving right along the first row, then left along the second row, etc.

In the early rows of Figure 11, only small (mainly single-celled) clots are seen. Further down the pipeline (*blood vessel*), small randomised variations in their speed have resulted in them bumping and coalescing into larger and slower moving clots. Even so, they manage to flow away fast enough that the faster moving singletons behind them coalesce into similarly large clots that cannot catch them and the stream continues to flow.

With higher probabilities of clot generation (not shown in the above figure), larger clots are formed that move slower still. Above a threshold (to be found by *in silico* experiment), these larger clots cannot escape being caught by smaller clots behind them — which leads to eventual catastrophic clotting of the whole system.

4.6. TUNA Perspective

For the introduction of *nanites* implementing artificial blood platelets, getting the balance right between the stimulation and inhibition of clotting reactions will be crucial to prevent a catastrophic runaway chain reaction. This model is a crude (as yet) platform for investigating the impact of many factors on that balance.

Our ambitions in the TUNA project call for scaling the size of these models through *three orders of magnitude* (i.e. tens of millions of processes) and *hard-to-quantify* orders of complexity. We will need to model (and visualise) two and three dimensional systems, factor in a mass of environmental stimulators, inhibitors and necessary supporting materials (such as *fibrinogen*) and distribute the simulation efficiently over many machines (to provide sufficient memory and processor power).

We suspect that simple cellular automata, as described in Section 4.2, will not be sufficient. We need to develop *lazy* versions, in which cells that are inactive make no demands on the processor. We also need to concentrate our modelling on processes that directly represent nanites/organelles, that are mobile and that attach themselves to particular locations in space (which can be modelled as *passive* server processes that do not need to be time-synchronised). Barrier *resignation* will be crucial to manage this laziness; but care will need to be applied to finding design patterns that overcome the *non-determinism* that arises from unconstrained use. Such an approach is taken in the model developed in Section 4.3. Another is presented in [17].

Achieving this will be a strong testing ground for the dynamic capabilities (e.g. mobile processes, channels and barriers) built into the new occam-π language, its compiler and run-time kernel. Currently, occam-π is the only candidate software infrastructure (of which we are aware) that offers support for our required scale of parallelism and relevant concurrency primitives. Further, it is backed up with compiler-checked rules against their misuse. We need the very high level of concurrency to give a chance for interesting complex behaviour to emerge that is not pre-programmed. We need to be able to capture rich emergent behaviour to investigate and develop the necessary theories to underpin the safe deployment of Nanite technology in medicine and elsewhere. How those theories will/may relate to the process algebra underlying occam-π semantics (i.e. Hoare's CSP and Milner's π-calculus) is a very interesting and very open question.

This work will contribute to the (UK) 'Grand Challenges for Computer Science' areas 1 (*In Vivo* ⇔ *In Silico*) and 7 (*Non-Standard Computation*).

5. Summary and Future Work

This paper has reported the introduction of *mobile* BARRIERs into the occam-π multiprocessing language. These provide an extra synchronisation mechanism, based upon the concept of *multiway events* from CSP and *mobility* from the π-calculus. The language binding, rules and semantics were presented first informally — followed by complete formal semantics through modelling in standard CSP. The current implementation mechanisms for occam-π were outlined, together with benchmark performance figures (from systems with up to 16 million processes). Finally, an application was described whose efficiency is transformed through the use of these barriers and their ability to be communicated.

The desired semantics for occam-π barrier synchronisation are precisely the same as those for CSP multiway events. Despite this, the former are not directly modelled by the latter, because of the need to capture the dynamics of run-time construction, enrolment, resignation and mobility (which are alien to CSP events). However, it turned out surprisingly easy to capture both the fundamental (CSP) synchronisation of barriers with their (π-calculus) dynamics — and we didn't have to step outside of standard CSP.

All that proved necessary was to model the support built into the occam-π kernel and the code generation sequences from the compiler (that interact with the kernel). Barriers become kernel processes operated through indexed control channels over which all application processes interleave. It would, perhaps, have been a better story to say that this CSP modelling came *first* (accompanied by some formal sanity check verifications and/or model checking) before the kernel and compiler were developed. Alas, we thought and did things the other way around.

This CSP modelling gives us both a *denotational* semantics (through the standard traces/failures/divergences semantics of CSP) and an *operational* semantics (describing the implementation). It enables formal verification and (finite) model checking for occam-π systems using mobile barriers. The denotational aspect further supports formal system specification and development through *refinement*. The operational aspect provides machine-independent formal documentation of the necessary compiler code generation and run-time kernel support.

This work has triggered a similar approach for the modelling of (occam-π) *mobile channels* in CSP. Again, kernel processes, rather than channels, are used to capture the synchronisation and dynamic semantics. This is a very recent result and will have to be reported elsewhere.

It may now be possible to provide a formal CSP model documenting the *entire* occam-π run-time kernel and supporting code generation. That would enable formal specification, development and analysis of all application systems, as well as provide a formal specification for the porting of occam-π to new target platforms (including the design of direct silicon support in future microprocessors).

Another development of this work could lead to a complete formal specification of a compiler from occam-π down to a simple register-based machine code — for example, see Section 2.3. Adding in formal constraints imposing the parallel and anti-aliasing *usage rules* of occam-π would further permit re-ordering of code sequences, necessary for the efficient operation of many modern microprocessors. Assistance for this is also given by avoiding unnecessary *serialisation* of code sequences in the formal definition — for example, Sections 2.3 and 2.4.8, where refinement into particular serialisations can be chosen at any stage (including their deferral till run-time). These re-orderings would be both *safe* (in terms of *sequential consistency* and multiprocessor execution) and *understandable* (by mortal systems designers and coders).

Such work is for the future, but should be relevant and within the timescale of the UK *'Grand Challenges in Computer Science'* [22] project on *Dependable Systems* [23]. The TUNA applications work, described in Section 4, are the beginings of contributions towards two of the other Grand Challenge areas: *In Vivo* ⇔ *In Silico* [24] and *Non-Classical Computation* [25].

Acknowledgements

We are grateful to our colleagues on the TUNA project for insights and much debate. Thanks especially to Jim Woodcock, Steve Schneider and Ana Cavalcanti for suggesting the blood clotting case study and for their own CSP models developing it — and for motivating us to

the importance of finding a formal semntics for the occam-π mobiles. We would also like to thank the anonymous reviewers for their helpful comments on an earlier version of this work.

References

[1] P.H. Welch and D.C. Wood. The Kent Retargetable occam Compiler. In *Proceedings of WoTUG 19*, pages 143–166. IOS Press, March 1996. ISBN: 90-5199-261-0.

[2] P.H. Welch, J. Moores, F.R.M. Barnes, and D.C. Wood. The KRoC Home Page, 2000. Available at: `http://www.cs.kent.ac.uk/projects/ofa/kroc/`.

[3] P.H. Welch and F.R.M. Barnes. Communicating mobile processes: introducing occam-pi. In A.E. Abdallah, C.B. Jones, and J.W. Sanders, editors, *25 Years of CSP*, volume 3525 of *Lecture Notes in Computer Science*, pages 175–210. Springer Verlag, April 2005.

[4] Frederick R.M. Barnes. *Dynamics and Pragmatics for High Performance Concurrency*. PhD thesis, University of Kent, June 2003.

[5] F.R.M. Barnes and P.H. Welch. Prioritised dynamic communicating and mobile processes. *IEE Proceedings – Software*, 150(2):121–136, April 2003.

[6] Inmos Limited. occam 2.1 Reference Manual. Technical report, Inmos Limited, May 1995. Available at: `http://wotug.org/occam/`.

[7] R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999. ISBN-10: 0521658691, ISBN-13: 9780521658690.

[8] F.R.M. Barnes, P.H. Welch, and A.T. Sampson. Barrier synchronisations for occam-pi. In *Proceedings of the 2005 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2005)*. CSREA press, June 2005.

[9] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, London, 1985. ISBN: 0-13-153271-5.

[10] A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1997. ISBN: 0-13-674409-5.

[11] L.G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[12] M. Schweigler. Adding Mobility to Networked Channel-Types. In *Proceedings of Communicating Process Architectures 2004*, pages 107–126, September 2004. ISBN: 1-58603-458-8.

[13] S. Stepney, P.H. Welch, F.A.C. Pollack, J.C.P. Woodcock, S. Schneider, H.E. Treharne, and A.L.C. Cavalcanti. TUNA: Theory underpinning nanotech assemblers (feasibility study), January 2005. EPSRC grant EP/C516966/1. Available from: `http://www.cs.york.ac.uk/nature/tuna/index.htm`.

[14] Peter H. Welch and David C. Wood. Higher Levels of Process Synchronisation. In *Proceedings of WoTUG 20*, pages 104–129. IOS Press, April 1997. ISBN: 90-5199-336-6.

[15] D.C. Wood and J. Moores. User-Defined Data Types and Operators in occam. In *Proceedings of WoTUG 22*, pages 121–146. IOS Press, April 1999. ISBN: 90-5199-480-X.

[16] M.D. Poole. Extended Transputer Code - a Target-Independent Representation of Parallel Programs. In *Proceedings of WoTUG 21*, pages 187–198. IOS Press, April 1998. ISBN: 90-5199-391-9.

[17] A.T. Sampson, P.H. Welch, and F.R.M. Barnes. Lazy Simulation of Cellular Automata with Communicating Processes. In J. Broenink, H. Roebbers, J. Sunter, P. Welch, and D. Wood, editors, *Communicating Process Architectures 2005*. IOS Press, September 2005.

[18] J.C.P. Woodcock and A.L.C. Cavalcanti. The Semantics of Circus. In *ZB 2002: Formal Specification and Development in Z and B*, volume 2272 of *Lecture Notes in Computer Science*, pages 184–203. Springer-Verlag, 2002.

[19] F.R.M. Barnes and P.H. Welch. Mobile Data, Dynamic Allocation and Zero Aliasing: an occam Experiment. In Alan Chalmers, Majid Mirmehdi, and Henk Muller, editors, *Communicating Process Architectures 2001*, volume 59 of *Concurrent Systems Engineering*, pages 243–264, Amsterdam, The Netherlands, September 2001. WoTUG, IOS Press. ISBN: 1-58603-202-X.

[20] P.H. Welch, G.R.R. Justo, and C.J. Willcock. Higher-Level Paradigms for Deadlock-Free High-Performance Systems. In R. Grebe, J. Hektor, S.C. Hilton, M.R. Jane, and P.H. Welch, editors, *Transputer Applications and Systems '93, Proceedings of the 1993 World Transputer Congress*, volume 2, pages 981–1004, Aachen, Germany, September 1993. IOS Press, Netherlands. ISBN 90-5199-140-1. See also: `http://www.cs.kent.ac.uk/pubs/1993/279`.

[21] P.H. Welch. Graceful Termination – Graceful Resetting. In *Applying Transputer-Based Parallel Machines, Proceedings of OUG 10*, pages 310–317, Enschede, Netherlands, April 1989. Occam User Group, IOS Press, Netherlands. ISBN 90 5199 007 3.

[22] UKCRC. Grand Challenges for Computing Research, 2004.
http://www.nesc.ac.uk/esi/events/Grand_Challenges/.
[23] J.C.P. Woodcock. Dependable Systems Evolution, May 2004. Available from:
http://www.nesc.ac.uk/esi/events/Grand_Challenges/proposals/.
[24] R. Sleep. In Vivo ⇔ In Silico: High fidelity reactive modelling of development and behaviour in plants and animals, May 2004. Available from:
http://www.nesc.ac.uk/esi/events/Grand_Challenges/proposals/.
[25] S. Stepney. Journeys in Non-Classical Computation, May 2004. Available from:
http://www.nesc.ac.uk/esi/events/Grand_Challenges/proposals/.