Communicating Complex Systems

Peter H. Welch, Frederick R.M. Barnes
Computing Laboratory
University of Kent, Canterbury
Kent, CT2 7NF, England
{p.h.welch, f.r.m.barnes}@kent.ac.uk

Fiona A.C. Polack
Department of Computer Science
University of York
Yorkshire, YO10 5DD, England
fiona@cs.york.ac.uk

Abstract

This paper presents efficient mechanisms for the direct *implementation of formal models of highly concurrent dynamic systems. The formalisms captured are CSP (for concurrency) and B (for state transformation). The technology is driving the development of* occam-π*, a multiprocessing language based on a careful combination of ideas from Hoare's CSP (giving compositional semantics, refinement and safety/liveness analysis) and Milner's π-calculus (giving dynamic network construction and mobility). We have been experimenting with systems developing as layered networks of self-organising neighbourhood-aware communicating processes, with no need for advanced planning or centralised control. The work reported is part of our TUNA ('Theory Underpinning Nanotech Assemblers') project, a partnership with colleagues from the Universities of York, Surrey and Kent, which is investigating formal approaches to the capture of safe emergent behaviour in highly complex systems. A particular study modelling artificial blood platelets is described. A novel contribution reported here is a fast resolution of (CSP external) choice between multiway process synchronisations from which any participant may withdraw its offer at any time. The software technology scales to millions of processes per processor and distributes over common multiprocessor clusters.*

1. Introduction and Motivation

We are interested in engineering for complex emergent systems, and specifically for deriving simulations. Such systems comprise many simple components (or unintelligent *agents*) following simple rules; emergent properties – the consequences of collective individual behaviour – are detected at a higher level. Often-cited examples of complex emergent systems include network navigation by ants (real or simulated), construction by termites, swarming and flocking, for example by birds or their simulated equivalent, boids. Our example case study operates at a finer level of granularity: a nanite system for haemostasis.

This is the third of three papers in these proceedings from the TUNA [22] project. Our overall approach to engineering for emergence is described in [23], with specific formal models presented and analysed in [21]. This paper looks in detail at one part of this, focussing on its CSP modelling and transformation into a faithfull and efficient executable simulation.

To model a complex emergent system, we need to describe the agents, the parts of the environment with which they interact, and the way in which emergence is detected [19]. In simulation terms, the detection is generally visual, so we must describe the basis of visualisation. In this paper, we focus on the development of the agents and their interaction with the environment.

An agent in a complex emergent system can move within its environment. It has:

- simple internal state;
- the ability to monitor some (not many) environmental parameters;
- a small number of outputs that may change the value of some environmental parameters (which may or may not be the same parameters that the agent monitors);
- basic operations that take the value of a monitored environmental parameter, change the internal state and (or) produce output.

Agents do not necessarily communicate directly with each other, but operate concurrently within their environment: influencing, and being influenced by, local factors.

In researching options for the modelling of systems of such agents, we seek well-defined modelling and implementation languages that can express all these properties. Concurrency is addressed by Hoare's CSP [10] and derived languages, and necessary concepts for mobility exist in Milner's π-calculus [17]. However, we also need state and operations, dictating the need for a model-based language element.

For our explorations, we are using two composite formal modelling languages and one implementation language. Modelling uses Circus [29] (a synthesis of CSP and Z, based on the unifying theory of programming [11]) and CSP||B [6] (B machines with CSP control). The rationale for Circus is that it is completely well-defined, and admits formal proof, refinement, and model-checking. The rationale for CSP||B is that it has been designed for use with existing tools, namely FDR2 (for CSP model-checking) and the B-Tool or Atelier-B. The inventors of each formalism are part of the TUNA research team.

For implementation, we are using and developing occam-π [26], an extension of classical occam [15] that includes mobile channels, barriers and processes. occam-π semantics are closely related to those of CSP, and with the evolution of Circus. Through its links with these formalisms, we will be able to map cleanly from formal models of agents in their environment to simulation programs.

An example formal model is given in section 4.1, with its translation to occam-π in sections 4.2 and 4.4. Section 4.3 presents novel and fast protocols for resolving CSP *external choice* between offers for multiway process synchronisation, allowing those offers to be withdrawn at any time.

occam-π enables the direct expression and efficient execution of self-organising dynamic systems about which we can formally reason (e.g. for safety analysis). Performance details (memory and run-time overheads) are reported (section 4.5) that will allow *massive* systems to be realised, giving the chance for interesting experiments and for complex and unplanned behaviour to emerge.

2. Overview of occam-pi

Systems programmed in occam-π are built as layered networks of communicating processes. occam-π processes are self-contained and self-executing components, interacting with their external environment (other processes) through parametrised communication channels, barriers (for multiway synchronisation) and a few other primitives, all built from CSP *events*. Compile-time alias analysis of state and event variables, combined with *Concurrent-Read-Exclusive-Write* parallel access rules, enables strong safety guarantees to be made. For example, *race hazards* on shared data simply cannot happen in occam-π systems. Deadlock/livelock dangers can be eliminated at design time, either through formal analysis of the specific CSP equations informing the design or (where possible) by sticking to synchronisation patterns for which there exist general safety theorems [28, 13, 14].

A *process* encapsulates state and logic for manipulating that state and synchronising with its environment. Process state is strictly private – neither observable nor changeable by other parties. A process runs its own life; its logic is not executed (invoked) by foreign threads of control. The events on which it synchronises, the patterns of synchronisation and its commitment to synchronise may be formally specified through CSP equations. We are looking to bind more elements of such specification into the occam-π language, so that the compiler can perform even more safety analysis (e.g. deadlock checks).

The *environment* of a process consists of other processes running in parallel and synchronising on (subsets of) common events. A set of processes and events form a *network*, which may itself be abstracted into a (super-)process with events having only local relevance hidden from the (super-)environment. Processes offer a natural and powerful structure for any computational system: *networks-within-networks*, explicit dependencies (through *visible plumbing*, i.e. shared events), and explicit independencies (through *air gaps*, i.e. no shared events).

In this paper, sections 4.2-4.4 assume modest familiarity with occam-π syntax and semantics [15, 3, 4, 26, 27, 2]. Key concurrency concepts not addressed by traditional programming languages are: channels (`CHAN`), shared channels (`SHARED` and `CLAIM`), message structures (`PROTOCOL`), synchronised unbuffered communications (`!` and `?`), multiway synchronisations (`BARRIER` and `SYNC`), abbreviations (`IS`), the parallel construtor (`PAR`), passive waiting for and choosing between events (`ALT`), and mobility (`MOBILE`). Data may be declared *mobile*, in which case communications and assignments *move* (rather than *copy*) them from source to destination, leaving the source variable *undefined*. Mobile items only have a single reference. Definedness analysis (with application mandated run-time checks, if not resolveable by the compiler) eliminates *null-pointer* errors.

This paper needs the choice constructor (`ALT`) to be extended to cover multiway synchronisation, describes how this can be done and provides (and uses) a mechanism for achieving this in the interim. This is a significant extension and the implementation is *fast* (linear with respect to the number of offers to synchronise – there is no backtracking).

Execution overheads for most concurrency mechanisms (e.g. process startup/shutdown, channel communication) are unit time and in the low tens of nanoseconds on current commodity processors. For an `ALT`, overheads are linear over the width of the choice. For a multiway `SYNC`, they are linear over the number of participants. Memory overheads for parallel processes are at most 32 bytes.

We have listed only the occam-π concurrency support relevant to this paper. The language encompasses much more: priorities, fairness, extended rendezvous, channel bundles, mobile channel-ends, mobile barriers, mobile processes, dynamic network construction, distribution across local clusters and the internet (with no change in semantics), etc. We leave them to be followed up by the interested reader – on-line resources are available from [2].

3. Simulating Complex Systems

Our work with occam-π allows increasingly elaborate models of an artificial platelet system to be modelled efficiently. The formal basis of occam-π allows formally-verifiable structures (e.g. the multiway synchronisation oracle presented later in section 4.3) to be added to the core occam-π functionality, permitting a direct implementation of the CSP model.

Distributed processing and the ultra-light occam-π concurrency primitives allow naturalistic real-time operation with low-cost support for visualisation (through graphical rendering of platelet movements and, potentially, environmental features such as chemical gradients). The required emergence can be observed, studied, and benchmarked across implementations [5].

The occam-π mobility mechanisms allow us to create a model of self-contained neighbourhood-aware autonomous agents that roam around a virtual world (so far, a one-dimensional world, but ultimately a realistic three-dimensional simulation) that we create for them. We can experiment with the simulation in interesting ways, such as changing the 'world' dynamically and *just-in-time* construction [20].

These occam-π extensions break the direct correspondence with CSP, whose primitives do not address the needed dynamics. However, work is in progress to re-establish that correspondence (through a further layer of CSP modelling) and this will be reported elsewhere.

In relation to nanotech assembler theory, the ability to experiment easily allows exploration of rule migration techniques and the layering of rule-spaces [19]. Whilst such simulated experimentation is not intended as a general technique for engineering real emergent systems, it is an important part of the development of a sound engineering paradigm for such systems.

Modelling for simulation is not completely equivalent to modelling with the aim of developing a nano-scale system (or, more accurately, system of systems). What direct role might be envisaged for simulation in the engineering of nanotech assemblers? We simulate emergent systems in the hope of gaining insight into the necessary and sufficient conditions for the required emergent properties to arise.

Simulation of agents builds on abstract models of agents, in the same way as development of physical nano-scale agents. However, the simulation only expresses the gross state and behaviour of the agent, making the assumption that the physics (and chemistry and biology, as appropriate) of the agent establishes these features. It is worth noting that the validity of this assumption in general, when working at the nano-scale, remains to be tested.

A simulation needs to capture those features of the environment that are sampled by the agents, and those features which the agents affect by their behaviour. In reality, of course, the environment is "given"; it cannot be controlled by the produced system. The simulation typically extracts what are believed to be pertinent characteristics of the environment, and models these as faithfully as is practical. A crucial feature for a simulation of emergent properties is that the environment is modelled independently of the agents. An agent is a particle in the environment; it owns the ability to monitor the environment, and can output to the environment, just like any other particle.

In the TUNA project, both the formal and the simulated modelling have spent much effort capturing relevant properties of the environment. In [19, 21], physical space is expressed independently from the platelet (agent). The simulations also model blood flow, the immediate environmental context for the platelets, although this is implicit. Platelets (or *clotted clumps* of platelets) move at a speed that is outside their control, simulating the platelet being carried along in the bloodstream. In addition, [21] starts to explore the environmental chemistry that is responsible for activating platelets. This exploits physical space and flow. Earlier CSP and occam-π models captured only the physical space.

There is much scope for elaborating the modelling of the environment. For instance, we need better models of chemical diffusion, more dynamic flow, and the introduction of many varieties of chemical. If our aim were to replicate real platelet behaviour in simulation, perhaps to complement biological studies, we would need to invest considerable resources in establishing a realistic environment.

The requirements of environmental simulation for engineering nano-scale systems are less clear. If we wish to reason about the precise behaviour of the artificial platelet in a real blood stream, we need a good environmental model. Ultimately we want to reason about the behaviour of the platelet in abnormal, and perhaps unforeseen, environmental situations. It is not yet clear what contribution a simulated environmenti, that is only ever an approximate model of reality, can play in assurance-driven engineering – but it is worth investigating.

4. Formal Design and Analysis

We consider the simplest of the CSP models being developed in the TUNA work programme. It describes the movement of *activated* (i.e. sticky) platelets along a one-dimensional bloodstream, capturing the rule that they must stick together should they ever bump.

The model heavily exploits *multiway synchronisation* combined with *external choice* – two of fundamental primitives of CSP. Unfortunately, in common with classical occam and all practical realisations of CSP to date [15, 1, 25, 24], this combination is specifically excluded from the current occam-π language. The reasons for this exclusion have always been cost (the only known implementation mechanisms requiring a *2-phase commit protocol* [12, 16]) and the difficulty in getting this logic right.

The traditional way to work around this problem has been to apply formally verified transformations from the CSP equations to equivalents (or refinements) that avoid the idiom. This introduces extra processes, channels, and the 2-phase commit logic (with its attendant overheads). Without automated tools, this transformation is error prone. The resulting system operates at a lower level that is hard to maintain. Maintenance, therefore, takes place at the higher level and the transformations have continually to be re-applied.

The example model (like many of the more complex TUNA models) relies on the *choice-over-multiway-synchronisation* idiom as its core structuring mechanism. We would like to remove the need for its removal (by enabling its direct expression in occam-π) and considerably reduce its cost (by finding an implementation faster than a 2-phase commit).

In the rest of this section, 4.1 presents a very simplified *bump-and-stick* model of blood clotting. 4.2 proposes an extension to the multiway synchronisation (`BARRIER`) primitive of occam-π, allowing it to be used as a guard in external choice (`ALT`) and enabling (for the first time) *direct* capture of the CSP equations in executable form. 4.3 presents the *fast* algorithm, using existing occam-π mechanisms, for resolving choice over multiway synchronisations. 4.4 applies this to translate the 4.1 code into actual executable code – of course, this will not be needed if the 4.2 proposal is accepted and implemented. 4.5 reports on observed behaviour and performance, noting and correcting a problem of under-specification in the original CSP specification that became apparent.

4.1. Sticky Platelet Model

This example has been distilled from richer models [21] developed by colleagues in the TUNA project.

`SYSTEM` is a parallel array of `SITE` processes representing the (one-dimensional) bloodstream. For each position in the stream, with index `i`, there is an event `pass.i` that represents the arrival of an activated platelet. The event `tock` represents the passing of one time unit:

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

SYSTEM =
  || i:{0..(N-1)} @ [EVENTS (i)] SITE (i)
```

where `N` is the number of positions in the bloodstream. The system diagram is shown in figure 1.

Each `SITE (i)` process monitors the arrival of a platelet at its own position (`pass.i`), its movement to the next position (`pass.i+1`), the movement of a platelet from the next

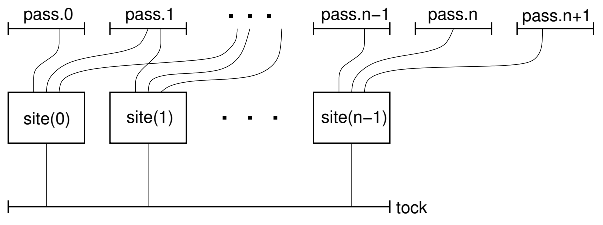

Figure 1. One-dimensional bloodstream model of sticky platelet flow

position to the next-but-one (`pass.i+2`), and time (`tock`). So, each event `pass.i` (other than those on the boundaries) requires *three* processes to engage before it can occur. The `tock` event requires *all* processes to engage. These are multiway synchronisations.

Note: the `pass.i+2` movement only plays a significant role when `SITE (i)` is in its *full* state (see below). However, that role brings that movement into the alphabet of `SITE (i)` for all its states. To ignore it in other states (*empty* and *almost-full*), it must be passively accepted.

The `SITE` processes exercise *choice* between overlapping sets of these multiway syncs. Each *site* starts off *empty*:

```
SITE (i) = EMPTY (i)
```

where:

```
EMPTY (i) =
  pass.i -> ALMOST (i) []
  pass.i+2 -> EMPTY (i) []
  tock -> EMPTY (i)
```

An *empty* site has no platelet to pass upstream – hence, it *refuses* the `pass.i+1` event. It allows, but is unaffected by, a platelet movement from position `i+1` to `i+2` (i.e. the event `pass.i+2`). Time may pass and it remains empty. An arriving platelet (`pass.i`) changes this site's state to *almost-full*:

```
ALMOST (i) =
  pass.i+2 -> ALMOST (i) []
  tock -> FULL (i)
```

This state converts to the *full* state by the passing of time. Until then, it does not allow the newly arrived platelet to move on (by refusing `pass.i+1`), thereby imposing a *speed limit* on the movement of platelets (to at most one site per `tock`). Like the *empty* state, *almost-full* allows the movement of any platelet next to it further upstream (by accepting `pass.i+2`). This is because that movement is in the same *clock cycle* as the movement of the platelet into this site – i.e. the two platelets do not touch and, therefore, stick together. Finally, it prevents the arrival of further platelets (by refusing `pass.i`), imposing a space limit of one platelet per position.

[It is a subtle aspect of CSP that lines of code *not written* (the events refused in each state) are as significant as the lines that *are written* (the events offered in each state).]

```
FULL (i) =
  pass.i+1 -> EMPTY (i) []
  pass.i+2 -> pass.i+1 -> EMPTY (i) []
  tock -> FULL (i)
```

This state also refuses a new platelet arrival (by refusing `pass.i`). However, at least one `tock` has happened since that arrival and, so, the platelet is allowed to move on (`pass.i+1`). Time (`tock`) may also pass. In this state, it is very interested in a platelet moving from position `i+1` to `i+2` (the event `pass.i+2`). That platelet must have been adjacent to the one in this position and, therefore, stuck to it. Hence, if that platelet moves, it *drags* the one here in its wake (in the *same* time unit – i.e. without a `tock`) and this position becomes empty.

The above system has been analysed by its designers using the FDR2 [9] model checker (for $N < 10$) and found to be deadlock and livelock free. Further analysis has verified that once-touching platelets remain stuck together. Such analysis uses ancillary equations that explicitly describe this *clumping* property, but this is not reported here. The considered point is that `SYSTEM` is safe and satisfies the key constraint. Our task is to turn this into something executable so that we can confirm its behaviour (with some additional visualisation) and start to perform experiments.

4.2. Language Binding

occam-π already has a `BARRIER` synchronisation type. Instances may be created dynamically, communicated and shared. Process enrolment and resignation is implicit and managed automatically by the occam-pi compiler and run-time kernel. Multiway synchronisation on such a barrier is *committed* and *individual*: a process cannot offer to synchronise on several barriers, select one that completes and back off the rest. Yet these extra properties are precisely what is required by the above CSP model.

We propose a new synchronisation type, `ALT.BARRIER`, that *can* be used as a guard in external choice (an occam-π `ALT`), giving us the property we need. We preserve the original `BARRIER` type since committed multiway synchronisation is also a common idiom – and we do not want to suffer the overheads of allowing withdrawal when that is not needed. Separate compilation of occam-π components means the distinction between these barrier types (which require very different code generation) cannot be resolved at compile time.

With the new type, occam-π code directly reflects the CSP equations. To avoid an unbounded demand for stack space, we replace the recursions (which are all *tail* recursions) with a simple state machine:

```
PROC site (ALT.BARRIER me, me.1, me.2, tock)
  VAL INT EMPTY IS 0:
  VAL INT ALMOST IS 1:
  VAL INT FULL IS 2:
  INITIAL INT state IS EMPTY:
  WHILE TRUE
    CASE state
      ...  EMPTY case
      ...  ALMOST case
      ...  FULL case
:
```

The parameters to `site` abstract the process away from the actual events on which it engages – these must be passed into each instance. We could have defined it with just an index number and accessed its events as globals (as we did in section 4.1 for the CSP `SITE`). However, the above abstraction makes the process self-contained, separately analysable and compilable, as well as faster executing. The system is put together with the following code:

```
[N+2]ALT.BARRIER pass:
ALT.BARRIER tock:

PAR i = 0 FOR N
  site (pass[i], pass[i+1], pass[i+2], tock)
```

In the body of `site`, the three state cases exercise *choice* over the multiway synchronisations (`ALT.BARRIERs`) the process offers. These choices directly reflect the CSP equations in section 4.1. Here is the most interesting of them:

```
{{{  FULL case
FULL
  ALT
    SYNC me.1
      state := EMPTY
    SYNC me.2
      SEQ
        SYNC me.1
        state := EMPTY
    SYNC tock
      SKIP
}}}
```

Of course, a useful system would include additional mechanisms (processes, channels, barriers, state and code) for visualisation, user interaction and performance monitoring. Such a system is outlined in section 4.5 below.

4.3. Implementation Oracle

The two-phase commit protocols employ *manager* processes for each multiway synchronisation event (`ALT.BARRIER`). Each manager receives offers to synchronise and counts down to zero as they arrive. Offers may be withdrawn at any time – including, sadly, after a count reaches zero! On reaching zero, the manager signals all offering processes to cancel whatever other offers they have made and commit to the one it is managing. Offers, withdrawals and commit requests fly around in parallel and a *2-phase* protocol is needed to secure the operation. This is moderately expensive, especially when managers that had reached a zero count find a cancel message arriving and have to *collapse* their current operation, undoing previous work.

Describing this to a colleague at Kent (Ian Marshall), he wondered why we did not integrate all managers into a single server that operated serially and dealt with offers one at a time. We wondered too! The parallel offers, countdowns, cancel messages and collapses all disappear, leaving decision costs that are linear with respect to the number of offered events.

Some things are lost though. For instance, some viable choices may never be made because of the particular sequence in which the offers are processed. However, the choices delivered will always be legal – i.e. we still have a valid *refinement* of the original CSP specification. If it matters to the system that all viable choices should *sometimes* be selected, the specification needs to be modified to say so (see section 4.5).

A distributed implementation requires remote communication between the offering processes and the centralised server; but this would be no worse than that needed for multiple managers. [There may be an elegant optimisation for a distributed implementation as a cascade of individual servers for each machine (dealing with local application processes only), occasionally forwarding locally completed synchronisations for resolution by a central super-server.]

We have a CSP description for this multiway synchronisation server with which we are working to prove correctness (i.e. that any system employing this server is a *refinement* of one with actual choice over multiway events). That work will be reported elsewhere. FDR scripts are available from the TUNA website [22].

Here, we present an occam-π realisation of this server, which is called `oracle`. It is very short and simple. Currently, this only supports a *fixed* number of multiway synchronisation events and a *fixed* number of processes making choices over them. However, occam-π is sufficiently dynamic to allow these numbers to change at run-time and support will be added later. This is the process header:

```
PROC oracle (MOBILE []ENROLLED enrolled,
             CHAN ORACLE.QUESTION in?,
             []CHAN ORACLE.ANSWER out!)
  ...  oracle code body
:
```

Each event has a unique index. Each applicant process has a unique index. These indices range from zero upwards and are consecutive.

The first parameter to `oracle` is an array indexed by the events. Each element is the list of indices of processes enrolled on the associated event. These lists have type:

```
DATA TYPE ENROLLED IS MOBILE []INT:
```

We assume that each of these lists has at least one member – i.e. that there are no redundant events (events on which no processes engage).

Applicant processes compete with each other to send their synchronisation offers through a `SHARED` channel to the `oracle`:

```
DATA TYPE OFFER IS MOBILE []INT:

PROTOCOL ORACLE.QUESTION IS INT; OFFER:
```

where an `OFFER` lists the indices of the offered events and the first component of the message protocol is the index of the offering process.

Applicant processes are connected to *separate* elements of the output channel array from the `oracle`. Having made an offer, an applicant process must wait on its *individual* answer channel for a reply:

```
PROTOCOL ORACLE.ANSWER IS INT; OFFER:
```

where the first component is the index of the chosen event and the second is the original offer. The former will, of course, be an element of the latter. The latter is returned because the `oracle` no longer needs it and the applicant probably does! [*Note:* occam-π `MOBILE` components *move* between sender and receiver processes, the sender losing it. So, having sent its offer, the applicant needs it back.]

Here is code for the applicant:

```
SEQ
  CLAIM to.oracle!
    to.oracle ! id; my.offer
  from.oracle ? answer; my.offer
```

The `CLAIM` blocks this process in a *FIFO* queue for exclusive access to the writing end of the channel to the `oracle`. That writing end is declared `SHARED`, so the writing line would not compile without its preceding `CLAIM`. The `answer` in the last line above is the index of the chosen synchronisation event.

Here is the code body for the `oracle`:

```
{{{  oracle code body
MOBILE []INT count:
MOBILE []OFFER offer:
SEQ
  count := MOBILE [SIZE enrolled]INT
  SEQ j = 0 FOR SIZE count
    count[j] := SIZE enrolled[j]
  offer := MOBILE [SIZE out]OFFER
  ...  oracle server loop
}}}
```

where:

```
{{{  oracle server loop
WHILE TRUE
  ...  loop invariant
  INT applicant:
  INITIAL INT chosen IS -1:  -- not chosen
  SEQ
    in ? applicant; offer[applicant]
    ...  decrement counts (may set chosen)
    IF
      chosen >= 0
        ...  process the decision
      TRUE
        SKIP
}}}
```

The `count` array is constructed and initialised to the number of processes enrolled on each event. The `offer` array is constructed to hold offers currently open from each applicant – initially, with all elements *undefined*.

The *loop invariant* states that, for each `j`, `count[j]` holds the number of offers still needed to complete multiway sync `j` – and that these counts are all currently greater than zero. It also states that, for each `i`, `offer[i]` is *defined* if and only if applicant `i` has made that offer and is awaiting an answer. Clearly, this is true on loop entry.

At the start of its loop body, `oracle` waits for a question, which contains the `applicant` index and offer. It saves that offer and, for each element of the offer, decrements the associated count and checks for zero:

```
{{{  decrement counts (may set chosen)
OFFER this.offer IS offer[applicant]:
SEQ j = 0 FOR SIZE this.offer
  VAL INT this.offer.j IS this.offer[j]:
  INT count.offer.j IS count[this.offer.j]:
  SEQ
    count.offer.j := count.offer.j - 1
    IF
      (count.offer.j = 0) AND (chosen < 0)
        chosen := this.offer.j
      TRUE
        SKIP
}}}
```

This sets `chosen` to the first event (index) found whose count decrements to zero – if any so do. *First* is an arbitrary choice – *any* zeroed event would work.

If no counts have reached zero, `oracle` loops back to await another offer. Otherwise a decision on a multiway synchronisation has been made; all processes enrolled on the chosen event must be informed and the counts incremented for all the events in their offers.

```
{{{  process the decision
ENROLLED release IS enrolled[chosen]:
SEQ i = 0 FOR SIZE release
  VAL INT release.i IS release[i]:
  OFFER next.offer IS offer[release.i]:
  SEQ
    SEQ j = 0 FOR SIZE next.offer
      INT count.offer.j IS count[next.offer[j]]:
      count.offer.j := count.offer.j + 1
    out[release.i] ! chosen; next.offer
}}}
```

The above returns all processed offers to the applicants, leaving the corresponding elements of the `offer` array *undefined*. The relevant counts have all been restored. The loop invariant is re-established.

Optimisations on the above algorithm are possible. For instance, the decrementing sequence could be stopped as soon as a zero count is found. This would yield a modest improvement in execution speed but it would complicate this presentation. We leave such things for a later day.

4.4. Applying the Oracle

The `oracle` process sets out logic for a fast resolution of choice between multiway synchronisations. We are binding this into a new occam-π compiler and kernel to support the `ALT.BARRIER` synchronisation type described in section 4.2.

An extension to JCSP (version 1.0-rc6 [25]) has been derived to provide the necessary (`AltingBarrier`) guard type. The extra controls needed for this are the same as those needed for the occam-π kernel. This version of JCSP and the extended occam-π will allow multiway synchronisation guards that can be freely mixed with existing guard types (channel inputs, timeouts and `SKIP`s) within an individual choice structure (`ALT`).

Direct translation to executable code will then become possible for almost all CSP equations. [The remaining problem area lies in supporting *arbitrary* patterns of event *interleaving*. occam-π provides only *structured* interleaving abstractions: shared channels, mobile channels, mobile barriers and mobile processes.]

Meanwhile, we can use the `oracle` at the application level to implement choice over multiway synchronisation. The system example from sections 4.1 and 4.2 becomes:

```
PROC site (VAL INT id,
    VAL INT me, me.1, me.2, tock,
    SHARED CHAN ORACLE.QUESTION to.oracle!,
    CHAN ORACLE.ANSWER from.oracle?)
  ...  site code body
:
```

where `id` is the index of *this* `site` process and `me`, `me.1`, `me.2` and `tock` are respective indices of the relevant multiway synchronisation events. The channel ends `to.oracle!` and `from.oracle?` will be connected to the `oracle` process. [*Note:* `id` and `me` are not necessarily the same value.]

The site implementation constructs the same state machine as before (section 4.2), but needs extra variables to hold all varieties of synchronisation offer it makes:

```
{{{  site code body
...  VAL INT EMPTY, ALMOST, FULL
INITIAL INT state IS EMPTY:
...  INITIAL OFFER empty, almost, full, drag
WHILE TRUE
  CASE state
    ...  EMPTY case
    ...  ALMOST case
    ...  FULL case
}}}
```

where:

```
{{{  INITIAL OFFER empty, almost, full, drag
INITIAL OFFER empty IS [me, me.2, tock]:
INITIAL OFFER almost IS [me.2, tock]:
INITIAL OFFER full IS [me.1, me,2, tock]:
INITIAL OFFER drag IS [me.1]:
}}}
```

As in section 4.2, we expand here only the most interesting case:

```
{{{  FULL case
FULL
  INT answer:
  SEQ
    CLAIM to.oracle!
      to.oracle ! id; full
    from.oracle ? answer; full
    CASE answer
      me.1
        state := EMPTY
      me.2
        SEQ
          CLAIM to.oracle!
            to.oracle ! id; drag
          from.oracle ? answer; drag
          state := EMPTY
      tock
        SKIP
}}}
```

The drag offer has, of course, only one possible answer (me.1) – but this still must be processed through the oracle.

This system is put together with the following code:

```
...  declare and set up 'enrolled' array

SHARED ! CHAN ORACLE.QUESTION to.oracle:
[N]CHAN ORACLE.ANSWER from.oracle:

PAR
  oracle (enrolled, to.oracle?, from.oracle!)
  PAR i = 0 FOR N
    site (i, i, i+1, i+2, N+2,
          to.oracle!, from.oracle[i]?)
```

This works – but it is hard work! Considerable care must be taken to enumerate all the events and processes in the system and pass the right indices to the right site processes. For this system, we have chosen to number the site processes from 0 through N-1 inclusive. The pass events have no explicit declaration but are also numbered from 0 through N+1 inclusive, leaving the tock event to be represented by N+2. Particular care must be taken in setting up the enrolled argument (which specifies which processes engage on which events – see section 4.3) for the oracle. None of this care will be necessary once we have the ALT.BARRIER type, which gives us explicit declaration of the events. – as shown in section 4.2.

4.5. Behaviour and Performance

The model in section 4.1 is a little under specified! For instance, one resolution of all choices made by all SITE processes would be simply to tock (an event always offered by all sites, whatever their states). In this case, no platelets would ever move – let alone stick together.

Another legal resolution would be for all platelets always to move forward one site after every tock. A FULL site in the middle of a clump sticks to the one on front anyway. A FULL site at the head of a clump also makes an offer to move on its platelet (pass.i+1); the EMPTY site ahead offers to accept that platelet (pass.i); the site behind (whether EMPTY, ALMOST full or FULL) is happy to let it go as well (pass.i+2). These are the three offers needed for the event. Hence, it may always be chosen.

If the above resolution always happens, platelet clumps remain isolated and flow forward in perfect synchrony – one movement per tock. They will never bump into each other and stick to form *larger* clumps.

For this SYSTEM, *always-move* is the resolution our oracle always delivers. The reason is that it makes decisions as soon as it has enough offers to complete one multi-way event. It does not wait to see if further offers may arise that would allow an alternative choice.

The movement forward of the head platelet of a clump requires only 3 offers. Staying still requires the passing of time – i.e. a tock offer from all N sites, where N will usually be considerably greater than 3. Hence, no contest – the platelets always move.

This is not very interesting. To allow the intended behaviour (bumping and clumping) to emerge, the FULL state of a site is therefore modified as follows:

```
FULL (i) = FULL_A (i) |~| FULL_B (i)
```

where FULL_A is what we had before:

```
FULL_A (i) =
  pass.i+1 -> EMPTY (i) []
  pass.i+2 -> pass.i+1 -> EMPTY (i) []
  tock -> FULL (i)
```

and FULL_B is the same except that it does not allow a forward move (unless dragged):

```
FULL_B (i) =
  pass.i+2 -> pass.i+1 -> EMPTY (i) []
  tock -> FULL (i)
```

A FULL site now has a non-deterministic (*internal*) choice between the behaviours FULL_A (which allows clump head platelets to move forward) and FULL_B (which does not). Formal analysis shows the same safety and liveness properties for this changed system.

The benefit is that implementations are free to resolve these internal choices any way they like – for example, *randomly*. If a 50-50 random policy is employed, average flow speed for the platelets is halved (i.e. one site every two tocks). However, no platelets maintain that speed uniformly. All experience *jitter* in their flow – sometimes not moving for several tocks and sometimes moving for several consecutive tocks. The result is bumping and, at last, clumping!

We have instrumented the occam-π implementation of this system to enable simple experiments. A generator process injects sticky platelets into one end of the pipeline (site (0)); they disappear automatically as they flow out the other end.

After each tock, all processes engage on another event called draw. This is a *fast* BARRIER – a committed multiway synchronisation, since no withdrawal is needed. Each site process maintains its own *pixel* from a display raster – though not between a tock and draw. A display process also cycles through these two events, rendering the raster between the tock and draw (when all site processes are blocked).

The display process also updates the screen with cycle rates for the system and various parameters. A keyboard process allows user control of these parameters: the rate at which the generator generates platelets, the raster sampling rate of the display, and the random decision ratio (between FULL_A and FULL_B) used by the site processes. It also controls system freezing and termination.

This system (documentation, source code and executable) is available from the TUNA website [22]. Table 1 lists *cycle* times per site, where a cycle is from one tock to the next (and includes computing a random number in the FULL state). Whole-system cycle times are these figures multiplied by the number of sites in the model (which happens to be 1800 in the reported experiments).

The figures were obtained from a PC powered by a 3.2 GHz Pentium 4, running under an otherwise unloaded Linux. Timings were averaged over one second runs (approximately one to two million site cycles) and rounded to the nearest 20 nanoseconds.

Performance depends *slightly* on the generation rate of sticky platelets (as shown in the table) and the random decision rates for FULL sites (set to 1/2 here). The raster sampling rate (for visualisation) was set low (1/256) so as not to interfere with gathering these results.

Table 1. Site cycle times

Generation Rate *(n/256)*	Cycle time *(nanosecs)*
0	480
1	480
2	500
4	520
8	540
16	600
32	700
64	880
128	1140
256	1160

With a zero generation rate, sites simply tock on empty. As that rate rises, platelets are introduced and start bumping and sticking as they flow through the system. Sites have more work to do as platelets pass through them and their cycle times increase. Clumps do not become arbitrarily big: the larger they grow, the larger the gaps between them and the less the likelihood of collision.

This 1-dimensional model has been a proving ground for our formal modelling and implementation techniques. Really useful experiments will need (two or) three dimensions, many more sites (perhaps hundreds of millions) and, of course, the layered models introducing realistic chemical interactions described in [21, 19, 27].

These experiments show simple emergent behaviour (clumping and limited growth) with performance sufficiently fast to encourage our ambition. They are the first directly to exploit the full expressive power of CSP (allowing choice over multiway synchronisation). We are developing *'lazy'* modelling techniques, whereby sites in which no actions are happening (e.g. EMPTY sites) do not need scheduling [20]. Such techniques may reduce processor loadings by one or two orders of magnitude. We will probably need them!

5. Related Work

Whilst formal modelling has been applied to agent systems [8, 7], there is little work on nano-scale complex adaptive systems. At the micro-scale, D'Inverno and others have modelled stem cell activity as agents; their work, which is aimed towards biologically-realistic simulation, uses the Z formal notation, and models only the state and operations of the agents. Their specification and implementation of the environment, which interacts with each stem cell to determine its division and state, is not formal, nor is their model of the operation of a system of stem-cell agents, as Z gives no mechanism for modelling the necessary concurrent and mobile interactions.

Calder and others have modelled biological systems such as protein signalling and molecular interaction using process algebras (PEPA, π-calculus). However, they do not aim to capture emergence or to establish a formal environment in which emergence is possible. For instance, the signalling is modelled in direct channels, and not through casual environmental interaction.

There is a wealth of biological simulation work, ranging from academic research models (Denis Noble's long-term cardiac modelling [18]) to pure entertainment. These simulations have a different goal to the TUNA work — they capture aspects of reality in ways that exhibit particular properties or illuminate understanding.

6. Conclusion and Future Work

Whilst the role of simulation in development of real nanoscale complex emergent systems remains to be determined, simulation can help us to understand emergence. Simulation is also necessary to demonstrate that a proven-correct model of an emergent systems can produce the required emergent properties.

This paper presents aspects of the occam-π simulation of the TUNA platelet case study, together with a novel and fast mechanism for resolving CSP *external choice* between *multiway synchronisation events* from which any participant may withdraw its offer at any time. This latter item provides the technique for efficient realisation of a proposed occam-π language extension directly capturing the pattern. It will considerably simplify the programming task and significantly broaden the range of CSP system that can be directly implemented (i.e. without further transformation and proof).

occam-π has existing and potential features that make it uniquely suited to modelling nano-scale systems of systems; it has natural support for mobility and concurrency, and a highly-efficient model of execution. TUNA is challenging and extending occam-π with coding patterns and kernel features needed to fully exploit its potential.

Several approaches are being considered. Just as the CSP models have been extended to CSP||B and to Circus models, allowing different properties to be analysed, so the occam-π systems include direct implementation of the CSP platelet models described in this paper, an independent CA model, and a very efficient lazy implementation based on the rule migration principles of [19]. At this stage, all approaches are potentially useful, and further analysis is needed to establish useful guidance in the implementation of nanotech simulations.

Our occam-π models faithfully implement the independent process layers (e.g. platelets, blood and chemical flow in physical space), and can elegantly capture the decomposition of the environment. Ultimately, this makes the implementation flexible; changes in the chemical model, for example, can easily be fed in to the simulation, without rewriting the rest of the environmental or platelet simulation.

Our (very) long-term goal is an engineering framework for dependable nanotech assemblers; our modelling and simulations will be ultimately useful only if they provide evidence for assurance arguments on real nano-scale systems of systems.

Acknowledgements

The TUNA project is funded by the UK EPSRC. It has given us the opportunity to interact with modellers working close to formal theory, system engineers making some sharp concurrency technology, and strategic thinkers in the context of emergent behaviour and novel computing paradigms. We thank our TUNA consortium colleagues (Ana Cavalcanti, Steve Schneider, Helen Treharne, Heather Turner, Susan Stepney, Jim Woodcock) for continuing discussions, criticism and encouragement. Special thanks go to Ian Marshall for striking the spark that led to the construction of the oracle for choice over multiway synchronisation. Finally, we record our gratitude to the anonymous referees, whose constructive comments were of considerable value both in the preparation of this paper and for our future work.

References

[1] M. Aubury, I. Page, G. Randall, J. Saul, and R. Watts. hcc: A Handel-C Compiler. Technical report, Oxford University Computing Laboratory, UK, Aug. 1996.

[2] F. Barnes, D. Dimmich, C. Jacobsen, M. Jadud, A. Sampson, and P. Welch. The occam-pi Home Page, 2006. Available at: `http://www.occam-pi.org/`.

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

[4] F. Barnes and P. Welch. Prioritised dynamic communicating and mobile processes. *IEE Proceedings – Software*, 150(2):121–136, Apr. 2003.

[5] F. Barnes, P. Welch, and A. Sampson. Barrier synchronisations for occam-pi. In H. R. Arabnia, editor, *Proceedings of the 2005 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2005)*, pages 173–179, Las Vegas, Nevada, USA, June 2005. CSREA press. ISBN: 1-932415-58-0.

[6] M. Butler. csp2B: A Practical Approach to Combining CSP and B. *Formal Aspects of Computing*, 12(3):182–198, Nov. 2000.

[7] M. Calder, S. Gilmore, and J. Hillston. Modelling the Influence of RKIP on the ERK Signalling Pathway using the Stochastic Process Algebra PEPA. In *Trabsactions on Computational Systems Biology*, LNCS. Springer, 2006. to appear.

[8] M. d'Inverno and R. Saunders. Agent-based Modelling of Stem-Cell Organisation in a Niche. In S. Bruekner, G. D. M. Serguendo, A. Karageorgos, and R. Nagpal, editors, *Engineering Self-Organising Systems – Methodologies and Applications*, volume 3464 of *LNAI*. Springer, 2005.

[9] Formal Systems (Europe) Ltd., 3, Alfred Street, Oxford. OX1 4EH, UK. *FDR2 User Manual*, May 2000.

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

[11] T. Hoare and H. Jifeng. *Unifying Theories of Programming*. Prentice Hall, Apr. 1998. ISBN: 0-134-58761-8.

[12] G. Jones. On Guards. In T. Muntean, editor, *7th occam Users Group & International Workshop on Parallel Programming of Transputer Based Machines*, pages 15–24, Grenoble, 1987. IOS Press, The Netherlands.

[13] J. Martin and P.H.Welch. A Design Strategy for Deadlock-free Concurrent Systems. *Transputer Communications*, 3(4):215–232, Oct. 1996.

[14] J. M. Martin and Y. Huddart. Parallel Algorithms for Deadlock and Livelock Analysis of Concurrent Systems. In P. Welch and A. Bakkers, editors, *Communicating Process Architectures, Proceedings of WoTUG 23*, volume 58 of *Concurrent Systems Engineering*, pages 1–14, Amsterdam, the Netherlands, Sept. 2000. WoTUG, IOS Press. ISBN: 1-58603-077-9.

[15] D. May. OCCAM. *ACM SIGPLAN Notices*, 18(4):69–79, Apr. 1983.

[16] A. A. McEwan. *Concurrent Program Development*. DPhil thesis, The University of Oxford, Submitted Trinity Term 2006.

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

[18] D. Noble. Oxford Cardiac Electrophysioogy Group – Home Page, 2006. Available at: `http://noble.physiol.ox.ac.uk/`.

[19] F. Polack, S. Stepney, H. Turner, P. H. Welch, and F. R. Barnes. An Architecture for Modelling Emergence in CA-Like Systems. In M. S. Capcarrère, A. A. Freitas, P. J. Bentley, C. G. Johnson, and J. Timmis, editors, *Advances in Artificial Life, 8th European Conference on Artificial Life (ECAL 2005)*, volume 3630 of *Lecture Notes in Computer Science*, pages 433–442, Canterbury, UK, Sept. 2005. Springer. ISBN: 3-540-28848-1.

[20] A. Sampson, P. Welch, and F. 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*, volume 63 of *Concurrent Systems Engineering Series*, pages 165–175, Amsterdam, The Netherlands, Sept. 2005. IOS Press. ISBN: 1-58603-561-4.

[21] S. Schneider, A. Cavalcanti, H. Treharne, and J. Woodcock. Bloody CSP: A Layered Behavioural Model of Platelets. In *Proceedings of ICECCS-2006*, Sept. 2006.

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

[23] S. Stepney, H. Turner, and F. Polack. Engineering Emergence. In *Proceedings of ICECCS-2006*, Sept. 2006.

[24] P. Welch. Process Oriented Design for Java – Concurrency for All. In *PDPTA 2000*, volume 1, pages 51–57. CSREA Press, June 2000. ISBN: 1-892512-52-1.

[25] P. Welch and P. Austin. The JCSP (CSP for Java) Home Page, 1999. Available at: `http://www.cs.kent.ac.uk/projects/ofa/jcsp/`.

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

[27] P. Welch and F. Barnes. Mobile Barriers for occam-pi: Semantics, Implementation and Application. In J. Broenink, H. Roebbers, J. Sunter, P. Welch, and D. Wood, editors, *Communicating Process Architectures 2005*, volume 63 of *Concurrent Systems Engineering Series*, pages 289–316, Amsterdam, The Netherlands, Sept. 2005. IOS Press. ISBN: 1-58603-561-4.

[28] P. Welch, G. Justo, and C. Willcock. Higher-Level Paradigms for Deadlock-Free High-Performance Systems. In R. Grebe, J. Hektor, S. Hilton, M. Jane, and P. 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`.

[29] J. Woodcock and A. 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.