Freezing Mobile Processes: an Introduction to occam-π

Peter Welch and Fred Barnes
University of Kent at Canterbury
Computing Laboratory
p.h.welch@kent.ac.uk
f.r.m.barnes@kent.ac.uk

IFIP WG 2.4, Baden, Austria (7th. January, 2005)

Twenty Years Ago ...

"... improved understanding and architecture independence were the goals of the design by Inmos of the occam multiprocessing language and the Transputer. The goals were achieved by implementation of the abstract ideas of process algebra and with an efficiency that is today almost unimaginable and certainly unmatchable."


2003 ...

We have been extending the classical occam language with ideas of mobility and dynamic network reconfiguration which are taken from Milner's π-calculus.

We have found ways of implementing these extensions that still involve significantly less resource overhead than that imposed by the higher level – but less structured, informal and non-compositional – concurrency primitives of existing languages (such as Java) or libraries (such as Posix threads).

2003 ...

We have been extending the classical occam language with ideas of mobility and dynamic network reconfiguration which are taken from Milner's π-calculus.

As a result, we can run applications with the order of millions of concurrent processes on modestly powered PCs. We have plans to extend the system, without sacrifice of too much efficiency and none of logic, to simple clusters of workstation, wider networks such as the Grid and small embedded devices.

2003 ...

In the interests of provability, we have been careful to preserve the distinction between the original static point-to-point synchronised communication of occam and the dynamic asynchronous multiplexed communication of π-calculus; in this, we have been prepared to sacrifice the elegant sparsity of the π-calculus.

We conjecture that the extra complexity and discipline introduced will make the task of developing, proving and maintaining concurrent and distributed programs easier.

occam-π: Aspirations and Principles

- **Simplicity**
  - There must be a consistent (denotational) semantics that matches our intuitive understanding for Communicating Mobile Processes.
  - There must be direct a relationship as possible between the formal theory and the implementation technologies to be used.
  - Without the above link (e.g. using C++,Posix or Java/monitors), there will be too much uncertainty as to how well the systems we build correspond to the theoretical design.

- **Dynamics**
  - Theory and practice must be flexible enough to cope with process mobility, location awareness, network growth and decay, disconnect and re-connect and resource sharing.

- **Performance**
  - Computational overheads for managing (millions of) evolving processes must be sufficiently low so as not to be a show-stopper.

- **Safety**
  - Massive concurrency – but no race hazards, deadlock, livelock or process starvation.
Computing Laboratory

**Mobile Process Types**

The Santa-Cruz mobiles were like laptops that you had to boot down before they could be unplugged from their current environment (e.g. LAN), moved, plugged into their new environment and re-booted. Safe but tedious.

The Baden mobiles can be asked to suspend (freezing all current live sub-processes), disconnected, moved, re-connected and resumed (with all frozen processes carrying on from their suspension points).

The reason we did not propose this originally was that we did not see how to arrange for all the sub-processes to freeze safely, how the mover could be sure this had happened to allow safe movement ... nor how to find all the frozen sub-processes fast for re-activation. We do see how to do this now ... ☺ ☺ ☺

---

**Mobile Process Types**

At Santa-Cruz, a proposal for an (occam) language binding for process mobility was presented. This had some good properties ... but at least one bad one ... which was duly pointed out during questions.

One of the major powers of process-oriented design is that the state of a process is represented not only by the values of its variables but also by where it has reached in its execution of code. Its execution model does not have to depend (switch) on global state attributes, which can lead to poor engineering.

The Santa-Cruz mobiles lost this power. They had to terminate before movement, recording their state in global attributes that survived termination and re-activation ...

---

**Mobile Process Types**

An occam-π mobile process, embedded anywhere in a dynamically evolving network, may suspend itself mid-execution, be safely disconnected from its local environment, moved (by channel communication) to a new environment, reconnected to that new environment and reactivated.

---

**Mobile Process Types**

An occam-π mobile process, embedded anywhere in a dynamically evolving network, may suspend itself mid-execution, be safely disconnected from its local environment, moved (by channel communication) to a new environment, reconnected to that new environment and reactivated.
An occam-π mobile process, embedded anywhere in a dynamically evolving network, may suspend itself mid-execution, be safely disconnected from its local environment, moved (by channel communication) to a new environment, reconnected to that new environment and reactivated.

Upon reactivation, the process resumes from the same state (i.e. data values and code positions) it held when suspended. Its view of that environment is unchanged, since that is abstracted by its channel interface. The environment on the other side of that abstraction, however, will usually be different.

The mobile process may itself contain any number of levels of dynamically evolving parallel sub-network.

Mobile processes are entities encapsulating state and code. They may be active or passive. Initially, they are passive. When passive, they may be activated or moved. A moved process remains passive. An active process cannot be moved or activated in parallel.

When an active mobile process suspends, it becomes passive – retaining its state and code position. When it moves, its state moves with it. When re-activated, it sees its previous state and continues from where it left off.

Process type declarations give names to proc header templates. Mobile processes may implement types with synchronisation parameters only (i.e. channels, barriers, buckets, etc.) plus records and fixed-size arrays of the same. For example:

```
PROC TYPE IN.OUT.SUSPEND (CHAN INT in?, ?, suspendsuspend?):
```

The above declares a process type called IN.OUT.SUSPEND. Processes implementing this will be given three channels by the (re-)activating host process: two for input (in?, suspendsuspend?) and one for output (out!), all carrying INT traffic.

Process types are used in two ways: for the declaration of process variables and to define the connection interface to a mobile process.
A process type may be implemented by many mobile processes – each offering different behaviours.

The mobile process from the last slide, \texttt{integrate.suspend}, implements the process type, \texttt{IN.OUT.SUSPEND}, defined earlier. Other processes could implement the same type.

A process variable has a specific process type. Its value may be \texttt{undefined} or some mobile process implementing its type. A process variable may be bound to different mobile processes, offering different behaviours, at different times in its life. When \texttt{defined}, it can only be activated according to that type.
Mobile Networks

Thanks to Tony Hoare for the insight allowing for the safe suspension of mobiles that have gone parallel internally (bar conversation, UK “Grand Challenges in CS” conference, Newcastle (28/03/2004)).

Our earlier model handles this by requiring normal termination of a mobile before it can be moved – i.e. a multiway synchronisation on the termination event of all internal processes (standard CSP).

So, treat suspend as a special event bound to all internal processes of the mobile (and local to them – i.e. hidden from its environment). The suspend only completes when all internal processes engage. Then, the mobile “early terminates” its activation (extended CSP).

For implementation, we just need a CSP event (an \texttt{accom-x barrier}) reserved in the workspace of any mobile. To reactivate, all its suspended processes will be on the queue held by that event = easy!

Well, not quite that easy ... but it certainly sorted this problem.

Graceful Suspension

We must still arrange for ‘graceful’ suspension by all the processes within a mobile.

If one sub-process gets stuck on an internal communication while all its sibling processes have suspended, we have deadlock.

Fortunately, there is a standard protocol for safely arranging this parallel suspend – it’s the same as that for ‘graceful’ termination.

For now, this is left for the mobile application to implement. It’s a concern orthogonal to the (language) design and mechanics of mobile suspension – in the same way that the ‘graceful’ termination protocol is orthogonal to the mechanics of parallel termination.

Separately, we are considering language support for such distributed decisions ...
PROC plus.suspend (CHAN BOOL.INT in.0?, in.1?, out?)
WHILE TRUE
  BOOL b.0, b.1:
  INT x.0, x.1:
  SEQ
  PAR
    in.0 ? b.0; x.0 -- b.0 ≠ no suspend
    in.1 ? b.1; x.1 -- b.1 = TRUE
  IF
  b.0
  out ! TRUE; x.0 + x.1 -- new running sum
  TRUE
  SEQ
  out ! FALSE; x.1 -- suspend signal (with sum)
  SUSPEND
::

PROC delta.suspend (CHAN BOOL.INT in?, out.0!, CHAN INT out.1?)
WHILE TRUE
  BOOL b:
  INT x:
  SEQ
  in ? b; x -- b ≠ no suspend
  IF
    b
    SKIP
  TRUE
  SEQ
  out.0 ! TRUE; x -- feedback running sum
  out.1 ! x -- output running sum
  SUSPEND
::

PROC prefix.suspend (VAL INT n, CHAN BOOL.INT in?, out!)
SEQ
  out ! n
WHILE TRUE
  BOOL b:
  INT x:
  SEQ
  in ? b; x -- b ≠ no suspend
  IF
    b
    SKIP
    SUSPEND
  TRUE
  out ! TRUE; x -- feedback running sum (no suspend)
Graceful Suspension

This parallel version of the `integrate, suspend` mobile process promptly suspends when its environment offers its `suspend` signal. It does this without deadlocking, without accepting any further `in` data and with flushing to `out` any data owed to its environment — i.e. it honours the contract (we intend to associate with `IN.OUT.SUSPEND`).

Deadlock would occur if the sequence of `output communication` and `suspension` were reversed in any of its component processes.

In fact, the `output` and `suspend` operations could safely be run in parallel by all components, except for `profile, suspend` (where deadlock would result since the output would never be accepted).

This shows the care that must be taken in applying the `graceful suspension` protocol — responsibility for which we are leaving, for the moment, with the application engineer.

Mobile Contracts

- **Process Type**
  - Currently, the `PROC TYPE` defines only the `connections` that are required and offered by a mobile.
  - The activating process has complete charge over setting up those connections. They are the only way a mobile can interact with its hosting environment. Nothing can happen without the knowledge and active participation of the host.
- **Contract**
  - This describes how a mobile is prepared to `behave` with respect to the synchronisation offers it receives from its environment (as parameterised by the `PROC TYPE` of the mobile).
  - CSP provides a powerful algebra for specifying rich patterns of such behaviour.
- **Function**
  - This describes how `values generated` by the mobile relate to `values received`.
  - `Z` specifications of the mobile as a state machine work here (and are integrated with CSP in the `Circus` algebra of Woodcock et al.).

For example, an `IN.OUT.SUSPEND` process is a `server` on its `in` and `suspend` channels, responding to an `in` with an `out` and to a `suspend` with `suspension` ("early termination").

Or this could be strengthened to indicate `priorities` for service ...

Or weakened to specify just its traces ...

Or weakened further to allow the number of `in` events to exceed the `out` events by more than one … and, of course, that the `out`'s never exceed the `in`'s …

Finally, note that the request for a `suspend` need not come only from the environment of a mobile. It could be a unilateral decision by the mobile itself (subject, of course, to satisfying any behavioural contract declared by its underlying type). It could be initiated by the mobile and negotiated with its environment. It could be all of these in parallel!

The `graceful` protocol can deal with such concurrent decisions safely.

Graceful Suspension

A behaviour we may want to prohibit is that an `IN.OUT.SUSPEND` process will not accept a `suspend` with an answer outstanding — i.e. that a `suspend` may only occur when the number of `in` and `out` events are equal.

This may be important both for the hosting environment and the mobile. Without such a contract, an `IN.OUT.SUSPEND` mobile could arrive that always refuses its `suspend` channel (and could never be removed by its host!) or activates with an `out` (and deadlocks its host!).

Note that `integrate, suspend` satisfies all these discussed contracts …
**Micro- benchmarks (3.4 GHz, Pentium IV) show:**
- process (startup + shutdown): 50 ns (no priorities) → 50 ns (priorities);
- change priority (up / down): 140 ns;
- channel communication (INT): 50 ns (no priorities) → 50 ns (priorities);
- channel communication (fixed-sized MOBILE data): 150 ns (with priorities, independent of size of the MOBILE);
- all times independent of number of processes and priorities used – until cache misses kick in.

**Process Performance (occam–π)**

- Memory overheads per parallel process:
  - ≪ 32 bytes (depends on whether the process needs to wait on timeouts or perform choice (ALT) operations);
  - Micro-benchmarks (800 MHz, Pentium III) show:
    - process (startup + shutdown): 30 ns (no priorities) → 70 ns (priorities);
    - change priority (up / down): 160 ns;
    - channel communication (INT): 60 ns (no priorities) → 60 ns (priorities);
    - channel communication (fixed-sized MOBILE data): 120 ns (with priorities, independent of size of the MOBILE);
    - mobile process allocation: 450 ns; mobile process activate + terminate: 100 ns; mobile process suspend + re-activate: 630 ns;
    - all times independent of number of processes and priorities used – until cache misses kick in.

**Process Performance (occam–π)**

- Mobile Process Example

**Mobility via Mobile Channels (Tarzan)**

To swing down a chain of 1M servers, exchanging one INT during each visit: **770 nsecs/visit (P3)**, **280 nsecs/visit (P4)**.

To swing down a chain of 1M servers, but doing no business: **450 nsecs/visit (P3)**, **120 nsecs/visit (P4)**.
To tunnel through a chain of \textbf{1M} servers, exchanging one \textbf{INT} during each visit: \textbf{1580 nsecs} visit (P3), \textbf{620 nsecs} visit (P4)

To tunnel through a chain of \textbf{1M} servers, but doing no business: \textbf{1340 nsecs} visit (P3), \textbf{470 nsecs} visit (P4)
Mobility via Mobile Processes (Mole)

```
CHAN TYPE RAIL
    MOBILE RECORD
CHAN MOBILE VISITOR c? :
    CHAN INT dummy.in, dummy.out:
SEQ
    visitor (dummy.in?, dummy.out!, (dummy.in?, dummy.out!, client)    )
    client
CLAIM next

CHAN TYPE RAIL
    MOBILE RECORD
CHAN MOBILE VISITOR harry:
INT time:
SEQ
    harry := MOBILE VISITOR
    ... initialise harry (with number of visits to perform)
    ... set up, release, catch, and debrief harry
CLAIM rail.client[0]
    rail.client[0] ! harry
    rail.server[n.servers][c] ? harry
CLAIM rail.client[0] ! harry
CLAIM rail.client[0] ! harry
```

… for example …
**Modelling Bio-Mechanisms**

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

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

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

- **In-silico**
  - Programmability: how can they find each other and link up?
  - How can mobile processes know where they are, how can they find each other and link up?
  - Planned or emergent? at what level – individual processes or clusters?

**Mobility and Location Awareness**

- **Classical communicating process applications**
  - Static network structures.
  - Static memory / silicon requirements (pre-allocated).
  - Great for hardware design and software for embedded controllers.
  - Consistent and rich underlying theory – CSP.

- **Dynamic communicating processes – some questions**
  - Mutating topologies: how to keep them safe?
  - Mobile channel-ends and processes: dual notions?
  - Simple operational semantics: low overhead implementation? Yes.
  - Process algebra: combine the best of CSP and the π-calculus? Yes.
  - Refinement: for manageable system verification … can we keep?
  - Location awareness: how can mobile processes know where they are, how can they find each other and link up?
  - Programmability: at what level – individual processes or clusters?
  - Overall behaviour: planned or emergent?
Mobility and Location Awareness

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

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

A Thesis and Hypothesis

- Thesis
  - Natural systems are concurrent at all levels of scale. Control is devolved. Central command cannot manage the complexity.
  - Natural systems are robust, efficient, long-lived and continuously evolving. We should take the hint!
  - Natural mechanisms should map on to simple engineering principles with low cost and high benefit. Concurrency is a natural mechanism.
  - We should look on concurrency as a core design mechanism – not as something difficult, used only to boost performance.
  - Computer science took a wrong turn once. Concurrency should not introduce the algorithmic distortions and hazards evident in current practice. It should hasten the construction, commissioning and maintenance of systems.

- Hypothesis
  - The wrong turn can be corrected and this correction is needed now.

Summary – 1/4

- occam-π
  - Combines process and channel mobility (from the π-calculus) with the discipline and safety of occam and the composable semantics of CSP. Even with the new dynamics ... what you see is what you get.
  - Minor performance hits for the new dynamics. Overheads for mobiles are still comparable to those for static processes ... \sim 100 \text{ns}.
  - Potential security benefits for dynamic peer-to-peer networks and agent technologies ... to be explored.
  - Natural for multi-layer modelling of micro-organisms (or nanobots) and their environments ... to be explored.
  - Support for creating CLONEs of (passive) mobile processes ... done.
  - Semantics for mobile processes – OK (but need adapting for our new model). Mobile channels raise new problems ... to be explored.

Summary – 2/4

- occam-π
  - All dynamic extensions (including mobile processes) implemented in EUSL 1.3.3 (but 1.3.4-pre1 has more ))
  - Denotational semantics for mobile processes (UTop / Circus) in print (Jim Woodcock, Xinbei Tang) – supporting refinement.
  - Hierarchical networks, dynamic topologies, structural integrity, safe sharing (of data and channels).
  - Total alias control by compiler: zero aliasing accidents, zero race hazards, zero nil-pointer exceptions and zero garbage collection.
  - Zero buffer overruns.
  - Most concurrency management is unit time \sim 100 \text{ns} on modern architecture.
  - Only implemented for x86 Linux and JVMX – other targets straightforward (but no time to do them).
  - Full open source (GPL / L-GPL).
  - Formal methods: FDA model checker, refinement calculus (CSP and CSP-π ?), Circus (CSP + Z).

Summary – 3/4

- The right stuff
  - Nature builds robust, complex and successful systems by allowing independent organisms control of their own lives and letting them interact. Central points of control do not remain viable for long.
  - Computer (software) engineers should take the hint! Concurrency should be a natural way to design any computer system (or component) above a minimal level of complexity.
  - It should simplify and hasten the construction, commissioning and maintenance of systems. It should not introduce the hazards that are evident in current practice; and it should be employed as a matter of routine.
  - Natural mechanisms should map into simple engineering mechanisms with low cost and high benefit.
  - To do this requires a paradigm shift in the way we approach concurrency ... to something much simpler.
  - Failure to do this will result in failure to meet the ‘Grand Challenges’ that the 21st Century is stacking up for us.

Summary – 4/4

- We Aim to Have Fun ...
  - Beat the complexity / scalability rap ...
  - Necessary to start now ...

- Google – I’m Feeling Lucky ...
  - KRoC + ofa
  - KRoC + Linux
  - JCBP
  - Quickstone
  - Grand Challenges + UK
  - CPA 2004 + Conference

- Any Questions?

- Mailing lists ...
  - occam-com@kent.ac.uk
  - java-threads@kent.ac.uk