Communicating Mobile Processes

Introducing occam-pi

Peter H. Welch and Frederick R.M. 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 occam-π, an efficient and safe binding of key elements from Hoare's CSP and Milner's π-calculus into a programming language of industrial strength. A brief overview of classical occam is presented, before focussing on the extensions providing data, channel and process mobility. Some implementation details are given, along with current benchmark results. Application techniques exploiting mobile processes for the direct modelling of large-scale natural systems are outlined, including the modelling of locality (so that free-ranging processes can locate each other). Run-time overheads are sufficiently low so that systems comprising millions of dynamically assembling and communicating processes are practical on modest processor resources. The ideas and technology will scale further to address larger systems of arbitrary complexity, distributed over multiple processors with no semantic discontinuity. Semantic design, comprehension and analysis are made possible through a natural structuring of systems into multiple levels of network and the compositionality of the underlying algebra.

1 Introduction

1.1 Mobile Processes in occam-π

A process, embedded anywhere in a dynamically evolving network, may suspend itself mid-execution, be safely disconnected from its local environment, moved (by communication along a channel), reconnected to a new environment and reactivated. Upon reactivation, the process resumes execution from the same state (i.e. data values and code positions) it held when it suspended. Its view of its environment is unchanged, since that is abstracted by its synchronisation (e.g. channel and barrier) interface and that remains constant. The actual environment bound to that interface will usually be different at each activation. The mobile process itself may contain any number of levels of dynamically evolving sub-network.

1.2 Structure of this Paper

The rest of this section describes the background to this work, along with some of the forces motivating it. Section 2 provides an overview of process and network construction in the occam-π language, with specific details on mobile data,

This article appeared in *Communicating Sequential Processes: The First 25 Years. Symposium on the Occasion of 25 Years of CSP*; July 7-8, 2004; London, UK; LNCS 3525, pp. 175-210; DOI: 10.1007/11423348_10. Copyright © 2005 Springer-Verlag, Berlin/Heidelberg. The original publication is available at www.springerlink.com: http://dx.doi.org/10.1007/11423348_10

mobile channels and dynamic process creation. The main work presented in this paper concerns mobile processes, covered in section 3. Performance benchmarks and figures for the various occam-π mechanisms are given in section 4. A notion of *duality* between mobile channel and mobile process mechanisms, arising from two of the benchmarks, is considered in section 4.6. Some application areas are explored in section 5. Finally, section 6 draws some conclusions and discusses the scope for future work.

1.3 Background

Twenty years ago, improved understanding and architecture independence were the goals of the design by Inmos of the occam [1, 2] multiprocessing language and the Transputer. The goals were achieved by implementation of the abstract ideas of process algebra (primarily CSP) and with an efficiency that is today almost unimaginable and certainly unmatchable.

We have been extending the classical occam language with ideas of mobility and dynamic network reconfiguration [3–7] which are taken from the π-calculus [8]. We have found ways of implementing these extensions that involve significantly less resource overhead than that imposed by the rather less structured concurrency primitives of existing languages (such as Java) or libraries (such as Posix threads). As a result, we can run applications with the order of millions of 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 workstations, wider networks such as the Grid and small embedded devices.

We are calling this new language, for the time being at least, occam-π. Classical occam built CSP primitives and operators into the language as first-class entities with a semantics that directly reflected those of CSP. occam-π extends this by judicious inclusion of the mobility features of the π-calculus. 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 the 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 and proving concurrent and distributed programs easier.

A further, minor, difference between occam-π and the underlying process algebra is its focussing on *channel-ends* in some places, rather than *channels*; this is to constrain the direction of data-flow over any particular channel to *one-way* only. More significant differences are apparent because of the direct language support for state information and transformation (such as variables, block structure and assignment). These are orthogonal to concurrency considerations — thanks largely to the strict control of aliasing inherited from the classical occam— and greatly simplify its application to industrial scale problems.

We view occam-π as an *experiment* in language design and implementation. It is sufficiently small to allow modification and extension, whilst being sufficiently powerful to build significant applications. The abstractions and semantics captured are not settled and may change in the light of future experience and theory (for example, into its formal semantics). However, it is sufficiently stable and efficient to invite others to play. The semantics will be denotational, retaining properties of compositionality derived from CSP and a calculus of refinement. This mathematics is built into the language design, its compiler, run-time system and tools, so that users benefit automatically from that foundation — without themselves needing to be experts in the theory. The new dynamics broadens its area of direct application to a wide field of industrial, commercial and scientific practice. The key safety properties of classical occam are retained by occam-π, giving strong guarantees against a range of common programming errors (such as aliasing accidents and race hazards). The language also provides high visibility of other classic problems of concurrency (such as deadlock, livelock and process starvation) and is supported by a range of formally verified design guidelines for combating them. Its close relationship with the process algebra allows, of course, these problems to be eliminated formally before implementation coding.

1.4 Natural Process Metaphors for Computing

The natural world exhibits concurrency at all levels of scale — from atomic, through human, to astronomic. This concurrency is endemic: a central point of control never remains stable for long, ultimately working against the logic and efficiency of whatever is supposed to be under that control. Natural systems are very resilient, efficient, long-lived and evolving.

Natural mechanisms should map on to simple engineering principles that offer high benefits with low costs, but the mapping has first to be accurate. In this case, the underlying mechanisms seem to be processes, communication and networks — precisely those addressed by our process algebra. Our belief, therefore, is that the basis for a good mapping exists, so that concurrency can and should be viewed as a *core design mechanism* for computer systems — not as something that is advanced and difficult, and only to be used as a last resort to boost performance. Concurrency should *simplify* the design, construction, commissioning and maintenance of systems.

This is not the current state of practice. Standard concurrency technologies are based on multiple threads of execution plus various kinds of locks to control the sharing of resources. Too little locking and systems mysteriously corrupt themselves — too much and they deadlock. Received wisdom from decades of practice is that concurrency is very hard, that we are faced with a barrage of new hazards, that our intuition derived from experience in serial computing is not valid in this context and that our solutions will be fragile. We are advised to steer well clear if at all possible [9].

On top of these logical problems, there are also problems for performance. Standard thread management imposes significant overheads in the form of additional memory demands (to maintain thread state) and run time (to allocate and garbage-collect thread state, switch processor context between states, recover from cache misses resulting from switched contexts and execute the protocols necessary for the correct and safe operation of locks). Even when using *'lightweight'* threads, applications need to limit implementations to only a few hundred threads per processor — beyond which performance catastrophically collapses (usually because of memory thrashing).

Threads are an engineering artifact derived from our successes in serial computing. Threads do not correspond well with nature. They support only a transient concept of ownership (as they lock resources for temporary use), an indirect form of communication (through their side-effects on shared data) and no notion of structure (to reflect natural layering of process networks).

Processes, however, have strong ownership of their internal resources (other processes cannot see or change them), communication (synchronous or asynchronous) as fundamental primitives and structure (a network of processes is itself a process, available for use as a component of a higher-level network).

We do claim performance wins from this *process-oriented* model of computing, but they are not the primary concern. The primary concern is a model of concurrency that is mathematically clean, yields no engineering surprises and scales well with complexity. We must be careful not to damage any of this as we extend the classical occam/CSP model with the dynamics of mobility from the π-calculus (and learn to exploit a few more tricks from nature).

2 An Overview of occam-π

The occam-π language is an extension of classical occam, incorporating: mobile data, channels and processes; dynamic process creation; recursion; extended rendezvous; process priority; protocol inheritance; and numerous other less language-centric enhancements. For instance, a (generally) faster `ALT` implementation, a fix to a long-standing bug with tagged-protocol communication, and greatly enhanced support for interacting with the system environment outside of occam-π. A more concise list of new features can be found on the KRoC web-page [3].

An example of an 'integrator' component is used throughout this and the following section. This particular component is a well-used teaching example, due to its simplicity and range of implementations. The basic interface to the process is two channels, one input and one output. Given the input sequence x, y, z, the integrator will output running sums: $x, (x + y), (x + y + z)$ and so on.

2.1 Defining Processes

Figure 1 shows the design and implementation of a *serial* integrator. The code is largely classical occam, with the exceptions of the removal of the '`OF`' keyword in channel declarations, the introduction of *channel direction specifiers* ('`?`', '`!`') on channel variables, and the use of an '`INITIAL`' declaration [10, 11] (with the obvious behaviour).

Channel direction specifiers declare channels as either being for input or output, as shown by the arrows in the diagrams. In fact, the classical occam compiler always deduced this information. This extension just makes that information explicit, bringing design and representation closer and enabling more accurate compiler error messages if the programmer contradicts herself.

in?
integrate
out!

```
PROC integrate (CHAN INT in?, out!)
  INITIAL INT total IS 0:
  WHILE TRUE
    INT x:
    SEQ
      in ? x
      total := total + x
      out ! total
:
```

Fig. 1. Serial integrate design and implementation

Note that this process never terminates — evident from its '`WHILE`' loop condition. Neither occam nor occam-π provide mechanisms for forcefully, and externally, terminating a process — this is dangerous. If we wish the process to be 'killable', that behaviour must be engineered into it. Adding such support to this serial integrator is trivial, as shown in figure 2.

in?
integrate.kill
out!
kill?

```
PROC integrate.kill (CHAN INT in?, out!,
                     CHAN INT kill?)
  INITIAL INT total IS 0:
  INITIAL BOOL ok IS TRUE:
  WHILE ok
    INT x:
    PRI ALT
      kill ? x
        ok := FALSE
      in ? x
        SEQ
          total := total + x
          out ! total
:
```

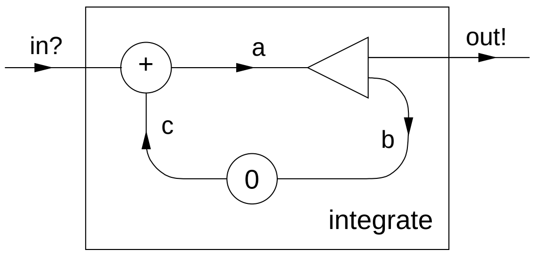
Fig. 2. A killable serial integrator

The process alternates between its two input channels, giving priority to the '`kill?`' channel. Ordinary input data values are added to the running total and output as before. A communication on the '`kill?`' channel causes the process to stop looping and terminate normally.

It should be noted that certain behaviours by the environment can cause deadlock with these processes. It would help to declare a "contract" [12] that formally specifies how a process is prepared to interact with its environment. For `integrate.kill`, the contract might specify that each communication on '`in?`' will only be followed by a communication on '`out!`', before any other communication (either on '`in?`' or '`kill?`') is accepted. Further, that a communication on the '`kill?`' channel will only be followed by termination. Such a contract guides both the implementation of the process and its safe positioning in an environment. This becomes even more of an issue for mobile processes, whose position with respect to its environment may change! Contracts are discussed further in section 3.5.

2.2 Process Networks

Static process networks in occam-π are no different from occam. Figure 3 shows a parallel version of the integrator process. It is a network of *stateless* components: an adder (that waits, in parallel, for a number on each input channel and then outputs their sum), a stream splitter (that outputs each input number, in parallel, on each output channel) and a prefixer (that initially generates a zero and, then copies input to output). State (the running-sum) emerges from the feedback loop in the network.

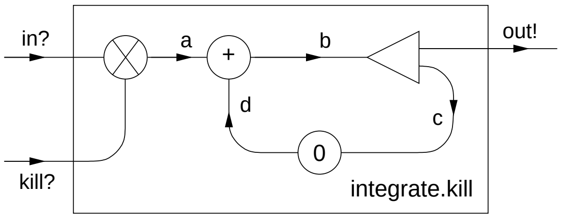


```
PROC integrate (CHAN INT in?, out!)
  CHAN INT a, b, c:
  PAR
    plus (in?, c?, a!)
    delta (a?, out!, b!)
    prefix (0, b?, c!)
:
```

Fig. 3. Parallel integrator design and implementation

Figure 3 implements a slightly relaxed version of the contract honoured by the process in figure 1. Internal buffering allows two '`in?`' events to occur before there must be an '`out!`'. Formally, figure 1 is a *refinement* of figure 3.

A killable parallel version requires some careful engineering to avoid internal deadlock. The "*graceful termination*" protocol described in [13] can be used to this effect. Figure 4 shows the modified process network.



```
PROC integrate.kill (CHAN INT in?,
                     CHAN INT out!, kill?)
  CHAN TAGGED.INT a, b, c, d:
  PAR
    kill (in?, kill?, a!)
    plus (a?, d?, b!)
    delta (b?, out!, c!)
    prefix (0, c?, d!)
:
```

Fig. 4. A killable parallel integrator

In order for the ‘`integrate.kill`’ process to terminate, all its parallel sub-components must terminate. This requires some changes to those components. The internal channels now carry a ‘`TAGGED.INT`’ protocol consisting of a boolean and an integer, where the boolean indicates whether the integer data is ‘good’ or this is a ‘kill’ signal. The implementation of each component must forward a ‘kill’ and then terminate. Care must be taken to do this in the right order or deadlock (not termination) will result! Further discussion of this protocol is postponed to section 3.4, where it is considered in the context of (mobile) process *suspension* (which is a little more delicate than termination, since network state must also be preserved).

2.3 Mobile Data

occam-π adds the concept of *mobility* to classical occam, incorporating mobile data, mobile channels and mobile processes. Mobile processes are discussed in section 3.

Communication and assignment in classical occam have a *copying* semantics. That is, the ‘source’ in output or assignment remains safely usable after the operation — the ‘target’ has received a copy. Clearly this precludes the creation of aliases, but has implications for performance if the data size is large (on shared-memory systems).

Mobile data types on the other hand have a *movement* semantics. That is, the ‘source’ in output or assignment is not available after the operation — it has moved to the target. This also precludes the creation of aliases. On shared-memory systems, this is a constant-time operation (effectively a pointer copy). If the communication is between memory spaces, copying has to happen — but the semantics remain that of movement (i.e. the ‘source’ always loses the data).

Mobile data types are declared simply by adding the ‘`MOBILE`’ keyword. For example:

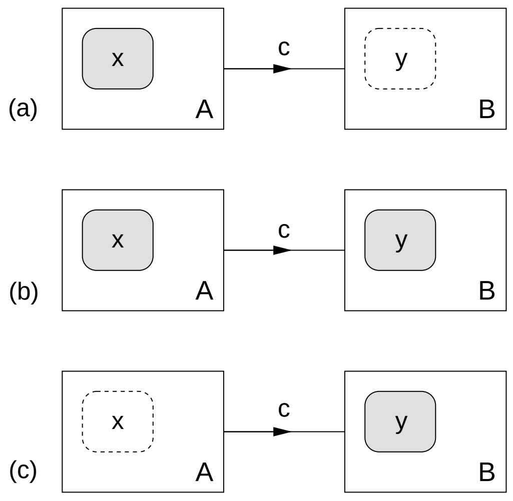
```
DATA TYPE FOO
  RECORD
    ...  data fields
:
```

declares a classical occam data type; whereas:

```
DATA TYPE FOO
  MOBILE RECORD
    ...  data fields
:
```

declares the mobile version. No changes are required to process codes operating on the type, but the semantics of communication and assignment on its variables become those of movement.

Figure 5 illustrates the difference between copying and movement semantics. Picture (a) shows the state of the system just before its communication — with the ‘x’ variable in process ‘A’ initialised and the ‘y’ variable in ‘B’ undefined. If ‘FOO’ were a classical (non-mobile) type, picture (b) shows system state just after communication — where ‘x’ still has its data and ‘y’ has a copy. If ‘FOO’ were a mobile type, picture (c) shows a different state following communication — where the data has moved to ‘y’ and ‘x’ has no data (i.e. is undefined).



```
PROC A (CHAN FOO out!)
  FOO x:
  SEQ
    ...  initialise ‘x’
    out ! x
    ...  continue
:

PROC B (CHAN FOO in?)
  FOO y:
  SEQ
    in ? y
    ...  use ‘y’
:

CHAN FOO c:
PAR
  A (c!)
  B (c?)
```

Fig. 5. Copying and movement semantics

The movement semantics leaves the ‘x’ variable undefined after the output — picture (c). Any subsequent attempt by process ‘A’ to use the value of ‘x’, before ‘x’ is reset, will result in a compile-time ‘undefined’ error. This *undefinedness-check* is an addition to the occam-π compiler, that now (pessimistically) tracks the *defined* status for all variables and channels — not just the mobile ones. There is also a ‘DEFINED’ prefix operator, applicable to any mobile variable, that may be used to resolve ambiguity in the defined status at run-time. It is impossible to write code that causes a *null-pointer* to be followed.

A copying semantics can be enforced on mobile data by use of the ‘CLONE’ operator. This creates a temporary mobile containing a copy of the data and it is this copy that is moved. For example:

```
PROC A.copy (CHAN FOO out!)
  FOO x:
  SEQ
    ...  initialise ‘x’
    out ! CLONE x
    ...  ‘x’ still defined
:
```

Dynamic Mobile Arrays. The mobile data described above has fixed-size memory requirements, allowing the compiler to pre-allocate space statically — despite their dynamic semantics.

occam-π has run-time sized arrays, whose allocation and deallocation must be performed dynamically. Such arrays are always mobile. Non-mobile dynamic arrays are currently not permitted — they are not strictly necessary, since 'CLONE' can be used to enforce copying semantics where necessary.

Dynamic mobile arrays are declared in a similar way to fixed-size mobile arrays. For example:

```
MOBILE []REAL64 data:
SEQ
  ...  process using 'data'
```

Unlike a fixed-size array, this 'data' initially has no elements. Any attempt to assign one of its elements would result in a run-time (array-bound) error. Before the elements can be accessed, the array must be sized and allocated. This is done using a special form of assignment:

```
data := MOBILE [n]REAL64
```

where 'n' is an integer expression, computable at run-time. Once allocated, the elements may be accessed, but they must be written (defined) before they can be read. The current occam-π compiler does not fully track this nested 'definedness' state, treating all elements as a single block — they are either all defined or all undefined.

The semantics for assignment and communication of these dynamic mobiles arrays are the same as for the static sized mobiles. Note that, because of the *single-reference* rule maintained by the semantics of mobility, no garbage-collection is needed to manage these dynamic types. The compiler always knows when that single reference is lost and automatically generates deallocation code.

The memory-allocation mechanism for these dynamic mobile arrays is based on Brinch-Hansen's allocator for parallel recursion [14], which is also used to provide memory for the other occam-π dynamic mechanisms that require it.

2.4 Mobile Channel Types

Mobile channels types in occam-π provide a mechanism for moving *channel-ends* — either by assignment or communication. This behaviour is not described in standard CSP, where processes (or parallel operators) are bound to *fixed* event alphabets. Moving channel-ends around means those alphabets are changing as the system evolves. The π-calculus [8] however is centered on this concept of channel mobility, allowing only channels to be communicated over channels in its purest form. We have an operational semantics for mobile channel-end communication, but do not yet have a denotational semantics.[1]

[1] It is important for this to be addressed in the future — see section 6.

The mobile channels of occam-π are defined by means of a structured channel-type (an idea partly taken from occam3 [10]). These define a group of one or more channels, accessed individually using a record subscript syntax. For example:

```
CHAN TYPE IO.KILL
  MOBILE RECORD
    CHAN INT in?:
    CHAN INT out!:
    CHAN INT kill?:
:
```

Variables of the channel-type hold its *ends* and must indicate which end explicitly. The terms 'server' and 'client' are used informally to refer to the two ends, with '`?`' and '`!`' as respective formal symbols. The *server-end* uses the component channels in the directions indicated by the channel-type declaration; the *client-end* uses them in the opposite directions. The usage pattern need not be 'client-server', however. For the above example, the channel-end types are written '`IO.KILL?`' and '`IO.KILL!`', for 'server' and 'client' ends respectively.

Mobile channels are created dynamically, by means of an assignment similar to that for mobile data, but where the right-hand side of the assignment produces the two ends of newly created channel 'bundle'. For example:

```
IO.KILL? io.svr:
IO.KILL! io.cli:
SEQ
  io.svr, io.cli := MOBILE IO.KILL
  ...  continue
```

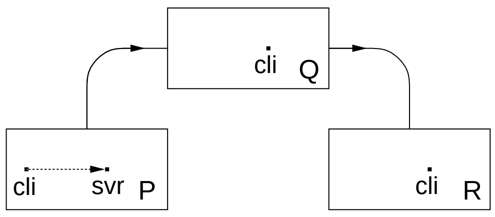
Once allocated, the channel-ends '`io.svr`' and '`io.cli`' may be used for communication or be themselves communicated (or assigned) to other processes (or variables). The semantics of the latter operations are the same as those for mobile data — the channel-end *moves* and the source variable becomes undefined.

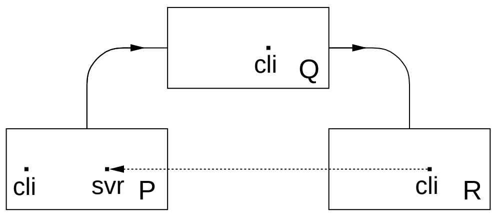
Figure 6 shows a simple network consisting of three processes '`P`', '`Q`' and '`R`', that communicate an '`IO.KILL`' client channel-end (which is, of course, a bundle of three scaler channel-ends). The server-end of the mobile channel-bundle is marked with an arrow pointing from the client-end — even though communication over the bundle will probably be in both directions.

Initially, processes '`P`' and '`R`' have no direct means of communication. '`P`' creates a channel-bundle and passes its client-end, via '`Q`', to '`R`'. '`P`' and '`R`' may now communicate directly over the channel bundle, observing some agreed usage pattern. For example:

```
INT x:                         INT v:
SEQ                            SEQ
  svr[in] ? x                    cli[in] ! 42
  svr[out] ! f(x)                cli[out] ? v
```

where the code on the left is in process '`P`' and the right is in '`R`'.





```
PROC P (CHAN IO.KILL! out!)
  IO.KILL! cli:
  IO.KILL? svr:
  SEQ
    cli, svr := MOBILE IO.KILL
    out ! cli
    ...  use 'svr' ('cli' undefined)
:

PROC Q (CHAN IO.KILL! in?, out!)
  WHILE TRUE
    IO.KILL! cli:
    SEQ
      in ? cli
      out ! cli
:

PROC R (CHAN IO.KILL! in?)
  IO.KILL! cli:
  SEQ
    in ? cli
    ...  use 'cli'
:
```

Fig. 6. Mobile channel-end communication

Currently, there are no restrictions on the communication of mobile channel-ends, enabling process networks to re-wire themselves arbitrarily. Some discipline will need to be enforced to render deadlock analysis, for example, manageable.

They also break another principle of occam that we hold dear, which is that that there should be no hidden ties between processes — all the plumbing should be visible (*WYSIWYG*) or their reusability as system components is compromised. We have plans to restore this principle through the explicit declaration of (typed) '`HOLE`'s in process interfaces, through which dynamically acquired channel-ends must be wired before they can be used for communication [15]. This will assist the behavioural specification of processes using mobile channels and maintain the compositionality of their semantics.

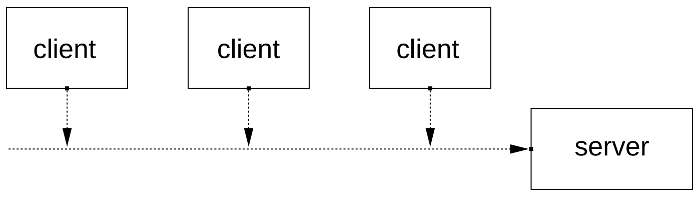
2.5 Shared Mobile Channel Types.

In addition to the *point-to-point* mobile channels described above, occam-π supports 'shared' channel-ends. These allow channel-ends (server or client) to be connected to *any* number of processes, although only one may be conducting business over it at a time.

A shared channel-end is communicated and assigned in the same way as a non-shared one, except that output and assignment automatically '`CLONE`' that end — leaving it defined locally. Before a process may use any of the component channels within a shared end, it must '`CLAIM`' exclusive access. Whilst so

'CLAIM'ed, the channel-end loses its mobility, preventing its communication or assignment.

Figure 7 shows a network of client and server processes connected using a shared channel-bundle.



```
PROC client (SHARED IO.KILL! cli)          PROC server (IO.KILL? svr)
  ...  local state                            ...  local state
  WHILE TRUE                                  WHILE TRUE
    INT v:                                      INT x:
    SEQ                                         SEQ
      CLAIM cli                                   svr[in] ? x
        SEQ                                       svr[out] ! f(x)
          cli[in] ! 42                            ...  update local state
          cli[out] ? v                        :
      ...  update local state
:
```

Fig. 7. Shared mobile channel bundles

The code to create this network is:

```
SHARED IO.KILL! cli:
IO.KILL? svr:
SEQ
  cli, svr := MOBILE IO.KILL
  PAR
    server (svr)
    PAR i = 0 FOR n.clients
      client (cli)
```

In this example the mobile channel-ends are "hard-wired" into the processes as they are created, but they could be communicated dynamically, if desired. An earlier paper describing mobile channels [4] shows this in detail.

Simple *request-answer* patterns of use across a channel-bundle correspond to simple CSP *interleaving* of the clients with respect to the shared channel-end. Richer patterns require semaphore processes to manage the locking. Locking of a resource, of course, opens new opportunities for deadlock. To reduce this risk, the occam-π compiler disallows any 'CLAIM' inside the 'CLAIM' of a client-end, but allows 'CLAIM's inside the 'CLAIM' of a server-end. This prevents the deadlock of "partially acquired resource", if multiple clients try to acquire the same set of channel-ends.

2.6 Dynamic Process Creation

Shared channel-ends are useful in their own right, but particularly so when combined with dynamic process creation.

In classical occam, networks are statically organised, with all potential configurations of all processes known in advance. occam-π enables dynamic network creation, in response to run-time decisions. Four mechanisms are provided for this: mobile processes (covered in section 3); (self-)recursive processes; run-time specified replicated '`PAR`' counts (as in the network code from the previous section); and the run-time "forking" of a parallel process. The last of these is examined here.

Forking a process is expressed in a similar way to an ordinary procedure call, but with an additional '`FORK`' keyword. Classical occam (and occam-π) use a *renaming* semantics for normal parameter-passing. Forked processes use a communication semantics for their parameters, since the forked process may out-live its given arguments — and that would break renaming. The use of communication semantics places restrictions on the parameter types that may be used: specifically, the parameters must be communicable — e.g. no reference parameters. Mobile parameters (data, channel-ends and processes) are allowed, since they have a well-defined communication semantics.

A common use of dynamic process creation is for setting up process 'farms'[4]. The network creation code for figure 7, for example, could also be written as:

```
SHARED IO.KILL! cli:
IO.KILL? svr:
SEQ
  cli, svr := MOBILE IO.KILL
  FORK server (svr)
  SEQ i = 0 FOR n.clients
    FORK client (cli)
  ...  do other things
```

The "other things", in the above code, may include waiting for events that trigger the forking of more clients — or, maybe, shutting some down. The code uses just forking to create its parallel process network. The parallelism is derived from the semantic model of the '`FORK`', described in [16]. This involves an external parallel process that receives, from the forking process, arguments for the forked one and constructs an instance of the requested process, with those arguments, in parallel with a recursive instance of itself. Forking offers no semantic power over that available from parallel recursion, but for many applications it is more convenient to program and has important implementation benefits (such as no memory leakage and faster setup).

3 Mobile Processes

The main subject of this paper, mobile processes, combines aspects of both mobility and dynamic process creation. The model for mobile processes, used by occam-π, is summarised at the start of this paper (section 1.1).

Note that mobile processes, encapsulating data and code, exist in one of two meta-states: *active* and *passive* — see figure 8. The internal (computational) state of a mobile process is only relevant when the process is active and interacting with the rest of the system. Initially, a mobile process is passive. In its passive state, a mobile process may be *activated* or *moved*. Once active, a mobile process only becomes passive either by *suspending* or *terminating* — these are voluntary internal events, not imposed (though may be requested) by its environment. The internal computational state (of data values and code positions) is retained between suspension and reactivation, and moves with the process. When reactivated, a mobile process sees exactly the same computational state that it did when it suspended. Once terminated, the mobile process may not be reactivated. Any attempt to do so behaves as *Stop*.

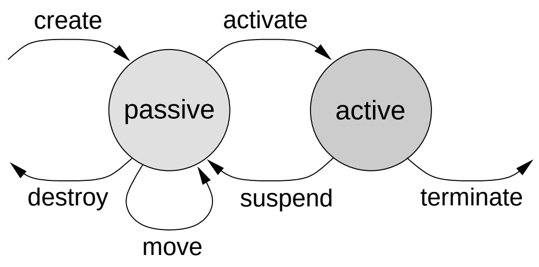


Fig. 8. Mobile process meta-state transitions

3.1 Process Types

The interface to a mobile process is defined through *process types*. For example, the `integrator.kill` processes (sections 2.1 and 2.2) match the type:

```
PROC TYPE IO.SUSPEND IS (CHAN INT in?, out!, suspend?):
```

where we have renamed the 'kill' property to 'suspend' for this context.

Activation arguments must conform to the parameter template defined by the mobile's process type — the activator process does not usually know, or care about, the actual process lying beneath that type. The activator sleeps while its activated process runs. The environment of the activator becomes the environment of the active mobile, interfaced through, and only through, the arguments supplied to the mobile.

Process types serve two purposes: the definition of the connection interface to a mobile process (section 3.2) and the declaration of mobile process variables (section 3.3).

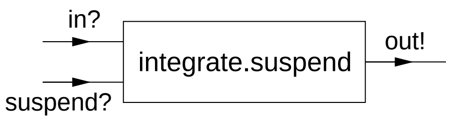
Note that the process type is not itself explicitly mobile. This allows process types to be used for non-mobile mechanisms in the future (such as making classical, as well as mobile, processes first-class types so they may be passed through parameter lists — similar to 'function pointers' in C).

3.2 Defining Mobile Processes

Mobile processes are defined in a similar way to ordinary occam-π procedures, except that they must be explicitly declared ‘MOBILE’ and must indicate which process-type is implemented.

Different mobile processes may implement the same process-type, assuming that the code conforms to any *contract* (section 3.5) that may, in future, be specified for the process type. For this example, a contract may be that an ‘in?’ event triggers an ‘out!’, and that a ‘suspend?’ signal triggers suspension of the mobile. However, suspension must not occur until the number of ‘in?’ and ‘out!’ events are equal.

Figure 9 shows the design and implementation of a ‘suspendable’ serial integrator that honours such a contract. To suspend itself, a mobile process invokes the new ‘SUSPEND’ primitive process. This suspends the mobile process and returns control to the activator. When next activated, the ‘SUSPEND’ terminates and control resumes (on the line indicated) with its local state (in this case, total and s) unchanged. The environment on the other side of its interface will probably be different. Activation of a mobile is covered in the next section.



```
MOBILE PROC integrate.suspend (CHAN INT in?, out!, suspend?)
 IMPLEMENTS IO.SUSPEND
  INITIAL INT total IS 0:           -- local state
  WHILE TRUE
    PRI ALT
      INT s:
      suspend ? s
        SUSPEND      -- return control to activator
        -- control returns here when next activated
      INT x:
      in ? x
        SEQ
          total := total + x
          out ! total
:
```

Fig. 9. A suspendable serial mobile integrator

The above mobile has a purely serial implementation. Suspending a mobile with a parallel implementation is presented in section 3.4.

3.3 Declaring, Allocating, Moving and Activating Mobile Processes

Mobile process variables are declared with reference to a process type. They hold instances of mobile processes, possibly many different ones during their lifetime.

Allocation of a mobile process is similar to the allocation of other mobiles — via a special assignment. For example, an instance of the '`integrate.suspend`' mobile process (defined in the previous section) is allocated by:

```
MOBILE IO.SUSPEND x:
SEQ
  x := MOBILE integrate.suspend
  ...  use 'x'
```

After allocation, the process in '`x`' may be communicated, assigned or activated. Communication and assignment follow the semantics of other mobiles — which is that the mobile process *moves*, leaving the source undefined.

The '`CLONE`' operator may be used to copy a mobile process, with a restriction that the mobile must not contain any state that cannot itself be cloned. For example, a mobile process containing an *unshared* mobile channel-end cannot be cloned. Any attempt to do so results in a compiler (or run-time) error.

Activation of a mobile process connects its interface to a local environment and transfers control to it. Control is returned when the mobile process either terminates or suspends.

Figure 10 shows a network of two processes, '`A`' and '`B`'. The '`A`' process simply creates a new mobile process then outputs it. '`B`' inputs a mobile process, activates it using channels from its own environment, waits for the activation to suspend (or terminate), before passing on the mobile.

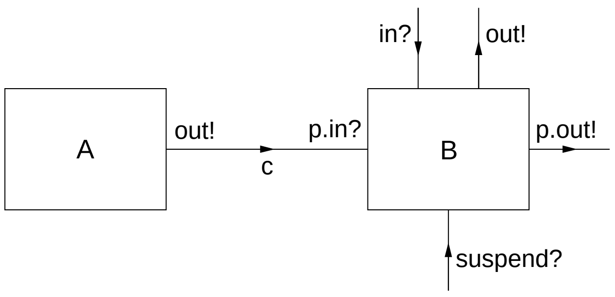


Fig. 10. A communicating mobile process network

The implementation of these examples are trivial:

```
PROC A (CHAN MOBILE IO.SUSPEND out!)
  MOBILE IO.SUSPEND x:
  SEQ
    x := MOBILE integrate.suspend
    out ! x
    -- 'x' is no longer defined
:
```

```
PROC B (CHAN MOBILE IO.SUSPEND p.in?, p.out!,
        CHAN INT in?, out!, suspend?)
  MOBILE IO.SUSPEND v:
  SEQ
    p.in ? v
    v (in?, out!, suspend?)
    -- control returns here when 'v' terminates or suspends
    p.out ! v
:
```

Note that the 'B' process is unaware what mobile process it is activating — only that it carries the 'IO.SUSPEND' interface. Note also the strong synchronisation between an activated mobile and its host. There is no way the host can operate on the mobile while it is active — it has to wait for the mobile to suspend or terminate. The parallel usage checker (implemented by the occam-π compiler) views an activated process variable as *writable* — i.e. it may change state. This means that that variable may not be *observed* in parallel with that activation — i.e. it may not be activated, moved, cloned or overwritten. Any attempt to do so is a language violation and will not be compiled.

The code implementing the portion of the network shown in figure 10 is:

```
CHAN MOBILE IO.SUSPEND c:
PAR
  A (c!)
  B (c?, p.out!, in?, out!, suspend?)
```

3.4 Suspending Mobile Networks

So far we have shown how a serial mobile process may be activated, suspended and moved. We are grateful to Tony Hoare for providing insight into how a mobile process, that has gone parallel internally, may be safely suspended and efficiently re-activated. An earlier proposal for mobile processes in occam-π [4] required the mobile to *terminate* before it could be moved. For parallel mobiles, such termination is just the multi-way synchronisation of all sub-processes on the termination event. So for each mobile process, introduce a hidden '*suspension*' event for all its sub-processes to synchronise upon — this, then, is the meaning of the new 'SUSPEND' primitive.

The suspension event barrier on which processes synchronise when executing 'SUSPEND' is internal to the mobile process and follows a similar implementation to that described in [17] for multiway events. The main difference being that whichever process completes the synchronisation must then arrange for control to be returned to the activator. Barrier completion may also be triggered when processes internally *resign* from the event (e.g. when terminating). The use of this barrier synchronisation enables very efficient re-activation — since all suspended sub-processes are on the queue (implemented by the barrier), they can be instantly located and rescheduled together in a constant-time operation (by appending the barrier queue to the kernel run-queue).

Parallel Suspension. As an example we consider a suspendable version of the parallel integrator. The design of this integrator is similar to the earlier 'killable' parallel integrator and is shown in figure 11. As with the suspendable serial integrator, the process is declared as implementing the '`IO.SUSPEND`' interface.

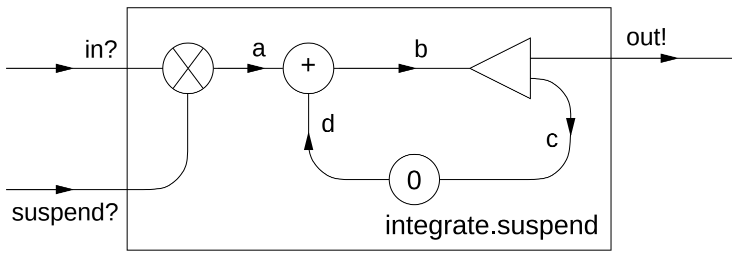


Fig. 11. A suspendable mobile parallel integrator

The top-level implementation of this mobile network is:

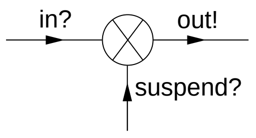
```
MOBILE PROC integrate.suspend (CHAN INT in?, out!, suspend?)
 IMPLEMENTS IO.SUSPEND
  CHAN TAGGED.INT a, b, c, d:
  PAR
    freeze (in?, suspend?, a!)
    plus.suspend (a?, d?, b!)
    delta.suspend (b?, c!, out!)
    prefix.suspend (0, c?, d!)
:
```

Note that the internal channels carry a boolean tag:

```
PROTOCOL TAGGED.INT IS BOOL; INT:
```

where a '`TRUE`' tag means that the `INT` part carried 'live' data (compute as normal) and a '`FALSE`' tag indicates 'suspended' data (forward and suspend). The '`freeze`' process is implemented:

```
PROC freeze (CHAN INT in?, suspend?, CHAN TAGGED.INT out!)
  WHILE TRUE
    INT x:
    PRI ALT
      suspend ? x
        SEQ
          out ! FALSE; 0   -- suspend signal
          SUSPEND
      in ? x
        out ! TRUE; x      -- live data
:
```



For structuring reasons and general reusability, we allow mobile processes to invoke 'ordinary' `PROC`s (which is what is happening between `integrate.suspend` and `freeze`). There is, therefore, the possibility some other application may invoke `freeze` by a chain of calls from a top-level process that is not itself mobile! If that happens, '`SUSPEND`' behaves as '`STOP`'.

The 'graceful' protocol safely distributes the suspend signal to all processes that need it. The implementation of '`plus.suspend`' and '`delta.suspend`', therefor, become:

```
PROC plus.suspend
     (CHAN TAGGED.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
        in.1 ? b.1; x.1
      IF
        b.0     -- live data
          out ! TRUE; x.0 + x.1
        TRUE    -- suspend signal
          SEQ
            out ! FALSE; x.1
            SUSPEND
:
```

```
PROC delta.suspend
     (CHAN TAGGED.INT in?,
      out.0!, CHAN INT out.1!)
  WHILE TRUE
    BOOL b:
    INT x:
    SEQ
      in ? b; x
      IF
        b       -- live data
          PAR
            out.0 ! TRUE; x
            out.1 ! x
        TRUE    -- suspend signal
          SEQ
            out.0 ! FALSE; x
            SUSPEND
:
```

Unlike the other two '`.suspend`' components, '`prefix.suspend`' executes its '`SUSPEND`' *between* input and output. It is the last process in the network that receives the suspend signal and someone has to hold the suspended data. The implementation is:

```
PROC prefix.suspend (VAL INT n, CHAN TAGGED.INT in?, out!)
  SEQ
    out ! FALSE; n
    WHILE TRUE
      BOOL b:
      INT x:
      SEQ
        in ? b; x
        IF
          b               -- input was live data
            SKIP
          TRUE            -- input was a suspend signal
            SUSPEND
        out ! TRUE; x     -- output is always live data
:
```

The way in which the '`integrate.suspend`' network suspends is as follows. A communication made on the external '`suspend?`' channel is intercepted by the '`freeze`' process, which reacts by outputting a suspend signal before suspending itself. The '`plus.suspend`' component inputs this, in parallel with the current running-sum, and outputs a suspend carrying the current running-sum before suspending itself. '`delta.suspend`' reacts to the suspend by forwarding the suspend (and associated running-sum) on the feedback channel only, and then suspending itself — no output is made to the external (integer) channel. '`prefix.suspend`' is the final process to receive the suspend signal and reacts by immediately suspending. At this point, all sub-processes have suspended and the network, therefore, suspends, returning control to its activator.

When the network is reactivated (elsewhere), the sub-processes resume execution from their respective '`SUSPEND`s. The '`prefix.suspend`' component returns the saved running-sum to '`plus.suspend`' and the network state is restored (as though the suspend never happened).

So, this parallel mobile '`integrate.suspend`' promptly suspends when its environment offers the '`suspend?`' signal. It does this without deadlocking, without accepting any further data from '`in?`' and flushing on '`out!`' any data owed to its environment — i.e. it honours the contract that we intend to associate with the '`IO.SUSPEND`' process-type (section 3.5).

Care must be taken to implement this "graceful suspension" protocol correctly to avoid deadlock. If the sequence of output and suspension were reversed in any of the internal components, deadlock would occur. In fact, the output and suspension could be run in parallel by all components *except* for '`prefix.suspend`' (where deadlock would result, since its output would never be accepted). For the moment, responsibility for getting this right lies with the application engineer.

Note that the request for a suspend need not come from the environment — it could be a unilateral decision taken by the mobile process itself, provided that it conforms to any specified behavioural contract for the process (e.g. that the number of '`in?`'s equals the number of '`out!`'s. In general, the decision to trigger suspension in a mobile process network may happen in several places independently. The protocol for managing safely the deadlock-free distribution of the multiple suspend signals so generated is described in [13].

Finally, although the '`integrate.suspend`' mobile behaves as a 'server', responding only to ('`in?`' and '`suspend?`') communications from its environment, this need not be the case. A mobile could behave as a 'client', gathering data from its various environments (which behave as 'servers'). Indeed, the relationship between mobile and its environment could follow any pattern — but it would help to formalise that into a *contract*.

3.5 Mobile Contracts

A "`PROC TYPE`" only defines a connection interface — a set of abstract events that are bound to actual events each time its implementing mobile is activated.

Such an interface is necessary. It prevents arriving mobiles from accessing resources the host is unwilling to provide. Activation is entirely under the control of the accepting host, who must set up all connections to the mobile (as well as actually activate it). An occam-π process cannot simply make "system calls" (e.g. to access a file), unless it has been given the means to make them (e.g. the file server channels). So, the host is in charge. If suspicious, the host may still provide resource access channels, but route them via a monitoring "fire-wall" process with whatever level of security it chooses. This is in marked contrast to conventional mobile platforms (e.g. web browsers and common office tools), which execute arriving code with the authority and permissions granted to the platform. Various "sand-boxing" techniques are available to counter the worst behaviour the mobile might throw, but these are foreign to the normal execution model. For the *process-oriented* model around which occam-π is centered, such "sand-boxing" is the way things are arranged anyway — and the security is automatic[2].

However, process type interfaces are not sufficient to guarantee safety. The host environment needs further assurance of good behaviour from an arriving mobile that it will use its given channels properly — e.g. that it will not cause deadlock or livelock, and will not starve processes in the host environment of attention (including a request to suspend). Conversely, a mobile process requires similar guarantees of good behaviour from whatever environment activates it.

We are currently investigating ways to augment process-types with a contract that makes some level of CSP specification about process behaviour. Initially we are considering methods of specifying traces for a mobile process, that the compiler can verify against an implementing mobile and any (potential) host environment. Such contracts would be burnt into an extended definition of the process type. We have not yet made proposals for a syntax for these contracts.

For the '`IO.SUSPEND`' process type, a contract might specify that implementing mobiles are a 'server' on the '`in?`' and '`suspend?`' channels, responding to an '`in?`' with an '`out!`', and to '`suspend?`' with suspension. This could be strengthened to indicate priorities for service, or weakened to allow some level of internal buffering.

A particular behaviour that a contract may wish to prohibit (for the example considered here) is that of suspension with an output outstanding on '`out!`' — i.e. that suspension may only occur when the number of '`in?`' and '`out!`' events are equal. Without such a contract, a mobile could arrive that activates with an '`out!`' to an environment that offers only an '`in?`'.

4 Performance

4.1 Basics

The implementation of the various concurrency mechanisms in occam-π are very lightweight compared with other software technologies (e.g. threads in Java or

[2] Like any software system, it is ultimately possible to circumvent guarantees such as this — but not if all codes are compiled from source by a certified occam-π compiler.

C), while providing substantial guarantees about the integrity of concurrent systems (an attribute preserved from classical occam and CSP).

The memory overhead for a parallel process is less-than or equal to 32 bytes, depending on what kinds of synchronisation it may choose to perform (e.g. ALTing and/or timeouts). The memory overhead for setting up a network of parallel process is approximately 16 bytes.

Table 1 shows the times for a number of "micro-benchmarks", measured on an 800 MHz Pentium-3 and a 3.4 GHz Pentium-4. These measure the minimum time to perform an operation, where the code and data required by a process is in the processor cache. Both machines have 512 Kbytes of fast cache. All times derive from multiple runs on an otherwise quiet Linux machine and are rounded to the nearest 10 nanoseconds.

Table 1. occam-π micro-benchmarks

Benchmark	Time (nanoseconds)	
	P3 (0.8)	P4 (3.4)
process startup + shutdown (no priorities)	30	0
process startup + shutdown (priorities)	70	50
priority change (up and down)	160	140
channel communication (INTs, no priorities)	60	50
channel communication (INTs, priorities)	60	40
channel communication (mobile fixed-size data, priorities)	120	150
channel communication (mobile runtime-sized data, priorities)	120	110
channel communication (mobile channel-ends, priorities)	120	110

The time for starting up and shutting down a process on the P4 running the occam-π kernel, with no support for priorities, was too small to measure accurately. The P4 (integer) channel communication costs were lower using the kernel *with* priorities than *without*. This shows the problems of relying too much on micro-benchmarks and we present them only as a guide.

4.2 Missing the Cache

A separate benchmark measures the penalty resulting from cache misses. This communicates integer messages between pairs of processes, with the number of pairs ranging from 1 to 1 million, increasing in factors of 10. The results from this benchmark are shown in figure 12. Graphs are drawn showing the effect of setting (and not setting) relevant optimisation flags to the compiler that in-line certain kernel operations.

Up to 1000 pairs of processes, the total memory footprint for the benchmark fits into cache. For 10,000 pairs and above, it does not. (In the case of a million pairs, the footprint is around 100 Mbytes.) Each cycle of the benchmark exercises all the data. Between each communication by any one process, all other

(20,000+) processes will have been scheduled once and cached state will have been lost. There are ways of managing scheduling that attempt to minimise cache displacement that might work for this benchmark. However, the KRoC runtime for occam-π simply uses *round-robin* scheduling on each priority queue of runnable processes. This benchmark uses no priorities, but it was run on the standard KRoC system build supporting them.

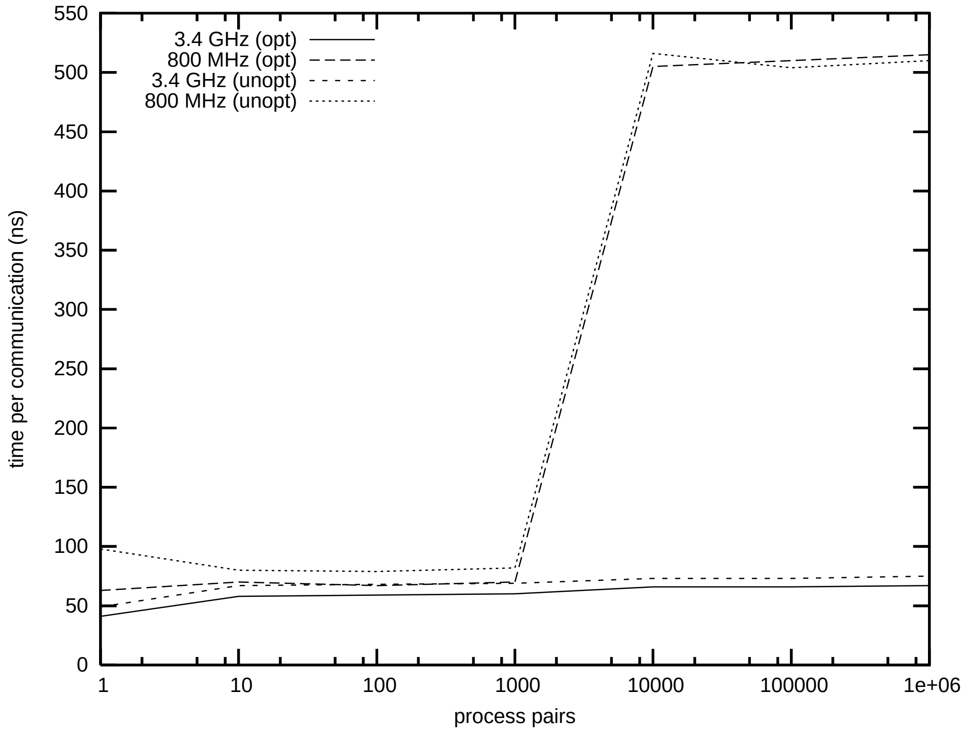


Fig. 12. Results for the communicating process pairs benchmark

As can be seen, the difference between optimised and unoptimised compiled codes is minor and consistent (except where very small numbers of processes are concerned). For the 800 MHz Pentium-3, the channel communication costs ceiling out at (a still respectable) 520ns for 10,000 pairs (20,000 processes) and above — measured up to 2M processes. The extra cost (over the minimum 80ns, when all process state is permanently cached) results from the relatively slow memory bus on that machine. The 3.4 GHz Pentium-4 machine has a more modern and much faster memory — even so, the results are remarkable! The costs start around 40ns and ceiling out at 70ns. Cache behaviour is not always what we expect; but whatever it is that the P4 is doing (and it may involve parallel operations from its *Hyperthreading* mechanism [18]), it is very well-suited to the operation of our occam-π kernel. The figures for large numbers of processes do reflect the worst-case memory behaviour that a large application might exhibit.

4.3 Mobile Process Basics

Table 2 shows micro-benchmark results for mobile process operations. All are well under 1 micro-second. Even so, they are still slightly higher than we eventually hope to achieve, due to the relative immaturity of the implementation.

The figures given for suspension and re-activation only apply to a serial mobile process (i.e. just *one* process synchronising on the hidden implementation barrier). Note that mobile process activation and termination costs are similar to those for ordinary procedure call and return.

Table 2. Micro-benchmarks for mobile process operations

Benchmark	Time per visit (nanoseconds) P3 (0.8 GHz)	P4 (3.4 GHz)
Mobile process allocation and deallocation	450	210
Mobile process activation and termination	100	20
Mobile process suspend and re-activate	630	260

For a more application-oriented scenario, two further benchmarks have been created that stress the memory cache and exercise mechanisms for mobility that are relevant for large-scale modelling. The first, "tarzan", provides mobility using mobile channels; the second, "mole", provides mobility using mobile processes. Both do similar work and show a sense of *duality* between mobile channels and mobile processes. This duality is considered further in section 4.6.

4.4 The Tarzan Benchmark

This benchmark measures the time taken to "swing" a process down a chain of a million 'server' processes, using mobile channels. The process network is shown in figure 13.

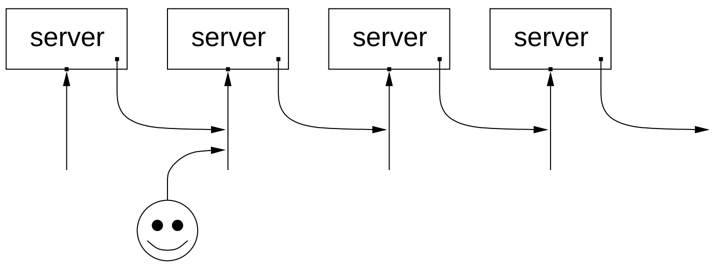


Fig. 13. Process network for the 'tarzan' benchmark

Starting with a connection to the first process in the pipeline, the `tarzan` process does some 'business' with the server and, then, receives from it the *shared* mobile channel-end of the next server. `tarzan` overwrites its connection to the current server (a shared mobile channel-end variable) with the connection to the next server, and loops. In this way, `tarzan` 'moves' (swings) down the chain. In fact, `tarzan` is actually fixed in memory and continuously running — only its connection to the individual servers changes as it swings down the line.

Note that if each server had connections to both its neighbours, it would be trivial for `tarzan` to move in both directions along the chain — in response to run-time decisions based on his communications with the chain nodes. Step this up one or two dimensions, add millions of other `tarzans` (and, maybe, some `janes`) and we are into serious application modelling — see section 5.

The channel type that defines the service channels in this benchmark is:

```
RECURSIVE CHAN TYPE SERVE
  MOBILE RECORD
    ...  business channels
    CHAN SHARED SERVE! next!:
:
```

The '`RECURSIVE`' keyword causes the name '`SERVE`' to be brought into scope early, instead of at the end of the declaration. This allows a channel-type to contain channels that communicate ends of its own channel-type (they may be 'client' or 'server' ends, shared or unshared). This is useful for many situations — e.g. having some client give up its (typically unshared) connection to a server, by communicating the client-end back to the server (for distribution to, and reuse by, some other client not known to the original one). For this benchmark, the feature enables a server to communicate (to its visiting `tarzan`) a 'client'-end connection to the next server in the pipeline.

The main loop of the `tarzan` process, for example, is implemented:

```
SEQ i = 0 FOR 1000000
  SHARED SERVE! next.server:
  SEQ
    CLAIM current.server
      SEQ
        ...  do business using 'current.server' channels
        current.server[next] ? next.server
    current.server := next.server
```

The `tarzan` client measures the time it takes to swing through 1 million server processes, and then reports. Table 3 shows the results for a client that just swings through the servers, doing no business (other than getting the link to the next server); and a client that asks each server a question (represented by an integer) and receives a reply (another integer), which it uses on the next server. Each visit by the `tarzan` client causes a cache miss as the service channel is accessed and the corresponding server is scheduled. `tarzan`'s own state will remain in cache (since it is repeatedly scheduled for each visit).

Table 3. Results for the 'tarzan' benchmark

Benchmark	Time per visit (nanoseconds) P3 (0.8 GHz)	P4 (3.4 GHz)
'just visiting' client	450	120
'question and answer' client	770	280

These results show over 3.5 million interacting visits per second are possible with this mechanism.

4.5 The Mole Benchmark

This benchmark is similar to the above in its basic operation (a visitor process interacting with and moving down a chain of servers), but is implemented using mobile processes rather than mobile channel-ends. Instead of moving a server connection to the visitor, the visitor suspends itself and is moved by its environment to the next server.

Figure 14 shows the process network for this 'mole' benchmark, with an activated visitor, our `mole`, connected to one of the servers. When a visitor arrives at the `butler` process, the latter forks a `host` platform and passes to it the visitor, the local server connection and the connection to the next butler. This host activates the `mole`, giving it the server connection. When the `mole` suspends, the host sends it on its way to the next butler and terminates.

This protocol is complicated by the fact that we wish to allow multiple visitors to connect to any single server at the same time. Our benchmark runs only one such visitor, so the butler could have done the work of the host platform itself without any extra concurrency (the forked host) — but then it could only service one visitor at a time. This would reduce the overheads measured by the benchmark, but also the realism of the scenario.

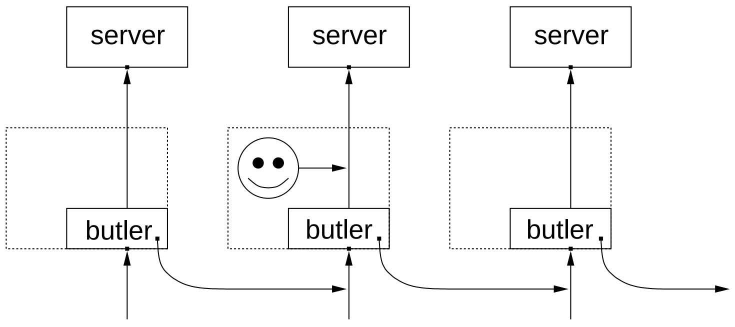


Fig. 14. Process network for the 'mole' benchmark

The channel-types servicing, respectively, the server and butler processes are:

```
CHAN TYPE SERVE.2                         CHAN TYPE BUTLER.2
  MOBILE RECORD                             MOBILE RECORD
    ...  business channels                    CHAN MOBILE VISITOR c?:
:                                         :
```

where the process type of the mobile visitors is:

```
PROC TYPE VISITOR (SHARED SERVE.2! client, CHAN INT in?, out!):
```

The extra '`in?`' and '`out!`' channels in the '`VISITOR`' type allow initial state to be loaded into the mobile and results to be downloaded upon completion of the benchmark. This is not an happy situation since those channels are not used during server visits (and, therefore, dummies must be supplied by the activating host platform). Our previous model for mobile processes, [4], allowed them to implement many process types. That would let us activate our visitor with one interface for initialisation, another for server visits and a third for debriefing. We are considering ways to combine the two models robustly.

The host and butler processes are rather trivial, apart from the current awkwardness with the dummy channels:

```
PROC host (MOBILE VISITOR mole,
           SHARED SERVE.2! my.server,
           SHARED BUTLER.2! next.butler)
  CHAN INT dummy.in, dummy.out:
  SEQ
    mole (my.server, dummy.in, dummy.out!)  -- dummy chans not used
    CLAIM next.butler
      next.butler[c] ! mole
:

PROC butler (CHAN MOBILE VISITOR in?,
             SHARED SERVE.2! my.server,
             SHARED BUTLER.2! next.butler)
  WHILE TRUE
    MOBILE VISITOR mole:
    SEQ
      in ? mole
      FORK host (mole, my.server, next.butler)
:
```

The main loop of the `mole` process is very similar to that for `tarzan`, except that it *suspends* and lets its environment move it to the next server:

```
SEQ i = 0 FOR 1000000
  SEQ
    CLAIM current.server
      ...  do business using ‘current.server’ channels
    SUSPEND
```

Table 4 shows the results for a `mole` that does no business with servers (other than claim their service channels) and one that does the same ‘question and answer’ interaction described for `tarzan`.

Table 4. Results for the ‘mole’ benchmark

	Time per visit (nanoseconds)	
Benchmark	P3 (0.8 GHz)	P4 (3.4 GHz)
‘just visiting’ client	1340	470
‘question and answer’ client	1590	620

The results show that the time per visit for this ‘mole’ benchmark is more than double the time per visit for the ‘tarzan’ benchmark. Some of the extra overhead comes from the mobile process suspension and re-activation in between visits — `tarzan` never stopped running! The rest comes from the forking of a new host platform to activate the mobile process. Nevertheless, more than 1.5 interacting visits per second are achieved with this mechanism.

4.6 Mobile Channels and Mobile Processes — a Duality

The two benchmark programs show how similar functionality can be implemented either using mobile channels or mobile processes. In both cases, a 'client' process moves down a line of 'server' processes, interacting with each in turn.

The main difference between the benchmarks involves the *locality* of processes. In the 'tarzan' benchmark, the visitor remains 'alive' throughout: channels are moved, 'stretching' across the network to provide mobility to the visitor (that sees itself serially connected to different servers). In the 'mole' benchmark, the visitor suspends its execution and is moved to the locality of the server — before being plugged in, re-activated and interacting over local channels. Putting aside the mechanism-specific code (for communicating a mobile channel-end in one and suspending in the other), the visitors and servers have identical logic.

On individual shared-memory systems (e.g. a typical workstation), the cost of communicating a mobile channel-end and the cost of communicating a passive mobile process are approximately the same — in the order of tens of nanoseconds. As we have seen, however, the mobile process cost has to be supplemented with the cost of suspension, forking and re-activation. Once connected, however, the costs of doing business in the new environment are the same, regardless of the mechanism used to achieve mobility.

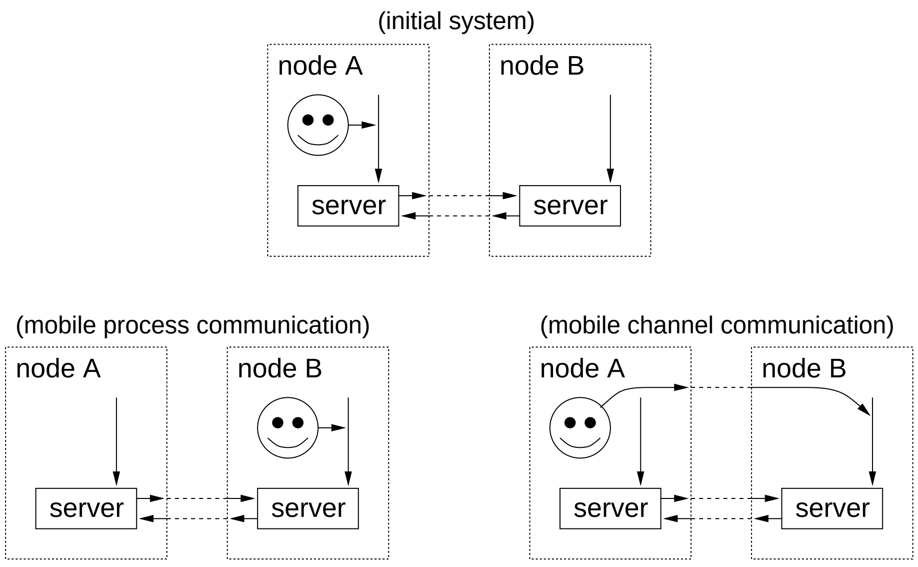


Fig. 15. Communicating mobile channels and processes between nodes in a network

If the system is distributed over a network of processors operating in separate memory spaces, the costs of doing business if a network link is involved differ significantly. Communicating a mobile process between nodes in a network has a relatively constant cost. Communicating mobile channel-ends between nodes in a network has a similar constant cost, but the 'stretching' of that channel between the nodes incurs a network overhead for each subsequent communication on the channel. Figure 15 illustrates this difference.

For optimal performance on distributed systems, the two techniques can be combined. Mobile processes are *moved* only when they need to connect to a new environment across the network. Otherwise, only channel-ends are moved. This reduces the level of transparency, however, since processes will need to be aware of where they are currently placed in the *physical distribution* of the system.

5 Application Outlines

5.1 Grand Challenges

"in Vivo ⇔ in Silico" (iViS) is one of the UK *'Grand Challenges in Computer Science'* project areas [19–21]. Its aims are to move the application of computing in the life sciences beyond cataloguing and pattern discovery and into modelling and prediction. An exemplar challenge is to model the development of a Nematode worm, one of the simplest multicellular forms of animal life, from fertilised cell to adult — allowing *virtual* experiments to be performed on its reactions to various stimuli, physical or chemical, and on interactions between organisms. It is hoped that success will lead to better understanding of the basic science and the processes involved, followed by improved treatment of disease and environmental dangers. One particular dream is the conduction of drug trials within the computer (*in silico*) that are trustable in real life (*in vivo*).

For the necessary modelling technologies, dynamic communicating process networks are a good fit. The fundamental ideas of *process*, *communication*, *concurrency* and *mobility* are uniformly applicable at any level of granularity and those levels build on each other seamlessly. They enable the expression of controlled, but not specifically planned, self-evolving topologies reflecting natural growth and decay. This uniformity of concept could contribute to simplicity of structure and understanding of multi-level simulation programs applied in biology. Furthermore, the semantics are independent of the actual distribution of systems on to different computer architectures and network configurations, allowing them to take quick advantage of all technological improvements to the hardware.

The mechanisms and implementation of occam-π, described in this paper, offer one way to make a start in these experiments. They are lightweight and robust and have good theoretical foundations — though we are aware that there is a lot more work to be done. To investigate emergent properties of such networks, self-constructed from low level processes with explicitly programmed behaviour, will require very large numbers of mobiles. Fortunately, current low cost architectures (e.g. PC networks) let us build systems with millions of processes per processor, yielding useful work in useful run-times.

5.2 Locality, Environment and the Matrix

Our models need to capture a sense of *location*, so that free-ranging processes become aware of who else is in their neighbourhood and do business with them (or, maybe, run away!). Processes may also be influenced by pervasive forces in their *environment* — these may be widely dispersed (e.g. gravitational) or highly localised (e.g. chemical).

Figure 16 illustrates some ideas for meeting these requirements. Space is modelled by the *'Matrix'* — a network of (usually passive and non-mobile) server processes representing locations. The figure shows a portion of a regular 2-dimensional grid. Other spaces may have higher dimensions, or distortions (e.g. wormholes), or the ability to change shape (reflecting dramatic changes in the modelled world, such as physical damage).

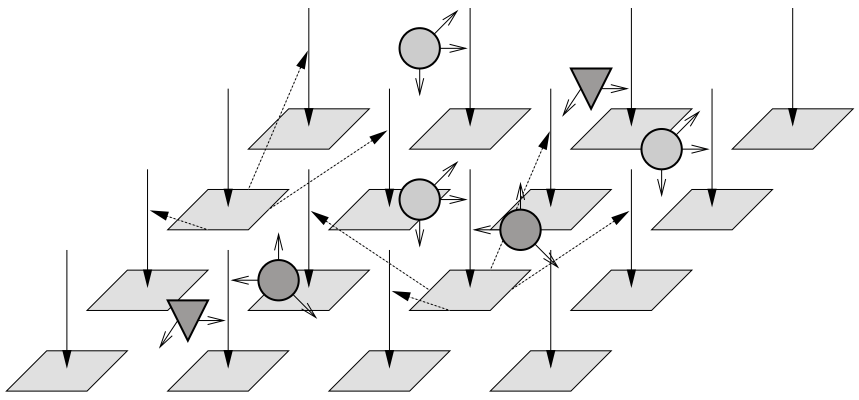


Fig. 16. Process matrix with mobile agents

Each matrix node services a channel bundle, shown in the figure as a vertical downward pointing arrow ending at the node. The 'server-end' of each bundle is exclusive to the node. The 'client-end', however, is `SHARED` and `MOBILE` (i.e. freely useable by, and communicable to, any number of clients).

Locality is realised by each node having access to the 'client-ends' of each of its neighbours' service channels, where we have free choice in deciding who are those neighbours. (In figure 16, only two sets of these connections are shown, but all nodes have them.) Once the connections are established, there are no run-time costs associated with locating neighbourhoods — even in the most twisted of topologies.

Organisms, or parts of organisms, living in this space are modelled by mobile processes — *'Agents'*. (These are shown by the shaded circles and triangles in figure 16.) An agent attaches to one matrix node (location) at a time, by acquiring the 'client-end' of its service channel bundle. It interacts with the server node, first to register its arrival and any connections to itself it cares to share with the locality. Then, it enquires about the local *environment* (e.g. electrostatic or chemical forces) and connections to other agents currently present. It may pick up compatible connections and transact business directly with those other agents. This may include combining with them to form larger agent structures or to reproduce. It may also pick up connections to neighbouring locations and decide to move.

Agent-matrix interactions must follow matrix-defined protocols (‘contracts’, section 3.5) for the avoidance of deadlock. Agent-agent communication protocols will be specific to the types of agent involved. The extent of these interactions will vary, along with the computations provoked by them. Model simulation time may need to be maintained by global (or, maybe, local) event barriers.

occam-π provides all the mechanisms needed to express such designs directly and execute them. Its overheads are sufficiently low so that the very large numbers of processes required for modelling realism will not be a show stopper. Formal verification of the systems, at least for the absence of deadlock and race hazards, also becomes possible.

Serial implementations of these designs, that iterate through collections of passive objects representing the locations and agents, may run (a little) faster. Unfortunately, the logic expressing object behaviour has to be inverted to the point of view of the (single) thread executing them all — there can be no *direct* expression. This introduces complexity, making formal and informal reasoning much harder. It will be necessary to experiment with many rules of behaviour, changing them quickly and often. The direct reflection of behaviour in the programming of active processes, together with the compositional semantics of the underlying algebra, simplifies this.

Finally, we note that the ‘tarzan’ and ‘mole’ benchmarks (sections 4.4 and 4.5) are stripped down versions of this scheme — where the matrix has one dimension, neighbourhoods are connected one-way only and there is just one agent. The discussion of *duality* between the use of mobile channels and mobile processes in these benchmarks (section 4.6) is directly relevant to this grander vision, especially for large scale models that need to be distributed over many machines.

5.3 Agents in Distributed Systems — and Security

The most commonly understood meaning of the term “mobile agent” is that of code and data mobility, as described by White in [22]. The main focus is on mobility between nodes in a distributed system. *Agents* are stateful mobile units of execution and *agent platforms* are the environments in which those agents operate. Supporting infrastructure is provided by the applications and by libraries, not by the programming model or language.

occam-π provides a simple model and language for agents: agents are mobile processes and agent platforms are processes that *activate* a mobile. Mobile processes may also activate other mobiles, becoming agent platforms themselves — i.e. nested hierarchies of agent are naturally expressed.

Agent platforms exist for two purposes: to allow agents to interact with the host system providing the platform; and to allow agents to interact with each other. occam-π supports both types of interaction, as outlined in the previous section.

Within the wider mobile-agent community, there is a good deal of concern for the security of mobile agents and agent based systems, as discussed in [23–25]. Broadly, these security considerations fall into two categories: those affecting the integrity of the overall system; and those affecting the integrity of individual agents and agent-platforms.

Integrity of the overall system is outside the scope of this paper. Here, we assume that arriving mobile agents are *valid* — because that agent was either created locally or came from another part of the system that we trust over secure links. Correspondingly, an agent may assume that whatever activates it was meant to do so.

In an insecure networked environment (such as the Internet), the part of the system that manages network connections would need to be responsible for ensuring the integrity of data communicated over networked channels (where that data may be 'serialised' mobile processes). This may involve proper (public/private key) authentication and encryption.

Of course, we could create a system that freely admits mobile processes from open network connections. Such a system would be open to many of the potential abuses that afflict mobile-agent systems in general. The use of occam-π in the construction of agents allows some of this threat to be eliminated. Instead of communicating serialised agent object code, source (or byte) code could be sent, along with the saved state of the agent, and used to re-create the agent locally. The occam-π compiler makes certain guarantees about the systems it compiles. For example, agents (processes) cannot access resources without being given specific connections (channels) to those resources and that giving is entirely at the discretion of the activating host — see section 3.5.

The mandated use of a synchronisation-only interface to mobile processes further limits the threats associated with existing agent systems. It separates the activation of an agent from its interaction with the local resources granted to it, by safely modelling the *concurrency* between the agent and those resources (which existed before the agent arrived and will continue to exist after it departs). There can be no unsynchronised actions between the agent and its host environment that can lead to race hazards.

Further, the concept of 'contract' (also described in section 3.5) would enforce safe patterns of synchronisation, eliminating the dangers of the agent deadlocking its host — or vice-versa. Such contracts are not yet defined for occam-π, although some preliminary investigations have been completed (see the '`TRACES`' extension described in [16]).

6 Conclusions and Future Research

This paper has given an introduction to the occam-π language, concentrating on mobile processes and channels. occam-π combines process and channel mobility (from the π-calculus) with the disciplines of classical occam (whose semantics follow CSP). Mobile processes complement mobile channels to provide the occam-π programmer with powerful new tools for directly, safely and efficiently capturing the dynamic aspects of complex large-scale systems. Applications for the multi-layer modelling of micro-organisms and their environments (the *'in Vivo ⇔ in Silico'* Grand Challenge [20, 21]) and process migration (agents) in distributed systems have been outlined. Performance benchmarks have been reported.

The occam-π language is implemented by recent releases of KRoC (the Kent Retargetable occam Compiler) [3]. Current versions of the system support all aspects of mobility described here, with the exception of support for 'serialisation' (and de-serialisation) of mobile processes — needed for their movement between distinct memory spaces.

At this time of writing, no *distributed* version of occam-π has been released (although library processes providing non-blocking low-level support for socket communication have long been included in the release). The *distributed* version, KRoC.net, will provide for the stretching of channels across network fabric (with no change in semantics), automatic multiplexing and de-multiplexing of channels over limited network resource (with no change in semantics), brokers for the discovery and run-time connection of processes between network nodes and full support for the networked communication of mobile data, channel-ends and processes [15].

Also under investigation are ways of formally specifying behaviours for process types ('contracts'), in ways that allow the compiler to verify that a mobile process conforms. In cases where this is too complex, the compiler may generate information suitable for use with a separate model checker (e.g. FDR [26]).

We emphasise that this work is still an *experiment* in language design and implementation. The abstractions and semantics captured are not settled and may change — especially in the light of new theory and experience with (large) applications. Certain elements of the language are incomplete. For example, we need *static* channel-bundle types as well as the *mobile* ones implemented so far; we need *arrays* of shared classical channels as well as the scalar ones currently available. However, such developments are largely routine and are a matter of (finding the) time.

occam-π is built upon classical occam and very little has been discarded. Classical occam was very compact, powerful and elegant. A key principle underlying the extensions is that the original semantics are not disturbed, so that the ultra-low overheads for process management and all the safety guarantees are preserved — despite the introduction of the new dynamics. For example, although there is now plenty of dynamic memory allocation (for run-time sized arrays, parallel process replication, recursion, forking, mobile channels and mobile processes), there is no need for any garbage collection — the system deallocates immediately when final references are lost (thanks to the strong policing of aliases, carried over from classical occam). Such properties are crucial for its continued relevance to real-time applications.

Nevertheless, perhaps *Ockham's razor* needs to be wielded a little more aggressively — the removal of the '`OF`' keyword is not very radical! For example, the syntax for declaring channel-bundle variables is not aligned with that for classical channels — maybe one of these versions should go? Could the compiler decide whether elements should be implemented as *mobile* or *shared* so that the programmer does not have to make this explicit — or would that require extra run-time cost and reduce system clarity? The duality noted between mobile processes and (some ways of working with) mobile channels may indicate that there is some simpler abstraction out there, from which these are special projections.

A formal denotational semantics, supporting refinement, needs completing. This is necessary both as a sanity check on the new ideas and to enable formal design and development. Such a semantics, based on Hoare and Jifeng's *Unified Theories of Programming* [27] has been built by Woodcock and Tang [28] for our earlier proposal for mobile processes [4]. That model allowed *multiple* interfaces for mobile processes but did not support *suspension* — they had to *terminate* before they could be moved and that required extra syntax to define persistent state (that moved with them). However, suspension should not be a major problem for that semantics to capture. In any case, it seems possible (and may be necessary) to merge that proposal with the one reported in this paper — the awkwardness of only having a single interface for mobile processes, discussed in section 4.5, needs addressing. It is also important for the semantics to address the issues raised by mobile channels, since the events bound to a process (mobile or static) will change as channel-ends are moved — section 2.4.

We welcome all feedback on this work. We shall be working towards the applications outlined in section 5, plus a few others — including RMoX [29], which is experimenting with occam-π for the design and implementation of real-time operating/embedded systems with low memory footprint, very fast reaction times and high-level (occam-π) programmability. The latest occam-π release, supported by the KRoC system, may be downloaded from [3].

7 Acknowledgements

We are grateful to Tony Hoare for his insights and advice, especially on how to suspend parallel process networks safely and efficiently [30]. We also wish to thank colleagues and other members of the concurrency research group at Kent — in particular David Wood, Christian Jacobsen and Mario Schweigler for their advice and associated work; and Jim Woodcock and Xinbei Tang, who produced the denotational semantics for our earlier model of mobile processes [4, 28].

References

1. May, D.: OCCAM. ACM SIGPLAN Notices **18** (1983) 69–79
2. Inmos Limited: occam2 Reference Manual. Prentice Hall (1988) ISBN: 0-13-629312-3.
3. Welch, P., Moores, J., Barnes, F., Wood, D.: The KRoC Home Page (2000) Available at: `http://www.cs.kent.ac.uk/projects/ofa/kroc/`.
4. Barnes, F., Welch, P.: Prioritised dynamic communicating and mobile processes. IEE Proceedings – Software **150** (2003) 121–136

5. Barnes, F., Welch, P.: Mobile Data Types for Communicating Processes. In: Proceedings of the 2001 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2001). Volume 1., CSREA press (2001) 20–26 ISBN: 1-892512-66-1.
6. Schweigler, M., Barnes, F., Welch, P.: Flexible, Transparent and Dynamic occam Networking with KRoC.net. In Broenink, J., Hilderink, G., eds.: Communicating Process Architectures 2003. WoTUG-26, Concurrent Systems Engineering, ISSN 1383-7575, Amsterdam, The Netherlands, IOS Press (2003) 199–224 ISBN: 1-58603-381-6.
7. Barnes, F., Welch, P.: Communicating Mobile Processes. In East, I., Martin, J., Welch, P., Duce, D., Green, M., eds.: Communicating Process Architectures 2004. Volume 62 of WoTUG-27, Concurrent Systems Engineering, ISSN 1383-7575., Amsterdam, The Netherlands, IOS Press (2004) 201–218 ISBN: 1-58603-458-8.
8. Milner, R., Parrow, J., Walker, D.: A Calculus of Mobile Processes – parts I and II. Journal of Information and Computation **100** (1992) 1–77 Available as technical report: ECS-LFCS-89-85/86, University of Edinburgh, UK.
9. Muller, H., Walrath, K.: Threads and Swing (2000) Available from: `http://java.sun.com/products/jfc/tsc/articles/threads/threads1.html`.
10. Barrett, G.: occam 3 Reference Manual. Technical report, Inmos Limited (1992) Available at: `http://wotug.org/parallel/occam/documentation/`.
11. Moores, J.: The Design and Implementation of occam/CSP Support for a Range of Languages and Platforms. PhD thesis, The University of Kent at Canterbury, Canterbury, Kent. CT2 7NF (2000)
12. Boosten, M.: Formal Contracts: Enabling Component Composition. In Broenink, J., Hilderink, G., eds.: Communicating Process Architectures 2003. WoTUG-26, Concurrent Systems Engineering, ISSN 1383-7575, Amsterdam, The Netherlands, IOS Press (2003) 185–197 ISBN: 1-58603-381-6.
13. Welch, P.: Graceful Termination – Graceful Resetting. In: Applying Transputer-Based Parallel Machines, Proceedings of OUG 10, Enschede, Netherlands, Occam User Group, IOS Press, Netherlands (1989) 310–317 ISBN 90 5199 007 3.
14. Brinch Hansen, P.: Efficient Parallel Recursion. ACM SIGPLAN Notices **30** (1995) 9–16 Reprinted in: *The Origin of Concurrent Programming*, edited by Per Brinch Hansen, pp. 525-534, Springer, ISBN 0-387-95401-5. 2002.
15. Schweigler, M.: Adding Mobility to Networked Channel-Types. In East, I., Martin, J., Welch, P., Duce, D., Green, M., eds.: Communicating Process Architectures 2004. Volume 62 of WoTUG-27, Concurrent Systems Engineering, ISSN 1383-7575., Amsterdam, The Netherlands, IOS Press (2004) 107–126 ISBN: 1-58603-458-8.
16. Barnes, F.R.: Dynamics and Pragmatics for High Performance Concurrency. PhD thesis, University of Kent (2003)
17. Welch, P.H., Wood, D.C.: Higher Levels of Process Synchronisation. In Bakkers, A., ed.: Parallel Programming and Java, Proceedings of WoTUG 20. Volume 50 of Concurrent Systems Engineering., Amsterdam, The Netherlands, World occam and Transputer User Group (WoTUG), IOS Press (1997) 104–129 ISBN: 90-5199-336-6.
18. Lin Chao et al.: Hyper-Threading Technology. Intel Technology Journal **6** (2002) ISSN: 1535-766X.
19. UKCRC: Grand Challenges for Computing Research (2004) `http://www.nesc.ac.uk/esi/events/Grand_Challenges/`.

20. Sleep, R.: In Vivo ⇔ In Silico: High fidelity reactive modelling of development and behaviour in plants and animals (2003) Available from: `http://www.nesc.ac.uk/esi/events/Grand_Challenges/proposals/ViSoGCWebv2.pdf`.
21. Welch, P.: Infrastructure for Multi-Level Simulation of Organisms (2004) Available from: `http://www.nesc.ac.uk/esi/events/Grand_Challenges/gcconf04/submissions/42.pdf`.
22. White, J.: Mobile agents white paper (1996) General Magic. `http://citeseer.ist.psu.edu/white96mobile.html`.
23. Jansen, W., Karygiannis, T.: NIST special publication 800-19 – mobile agent security. Technical report, National Institute of Standards and Technology, Computer Security Division, Gaithersburg, MD 20899. U.S. (2000) `http://citeseer.ist.psu.edu/jansen00nist.html`.
24. Jansen, W.A.: Countermeasures for Mobile Agent Security. Computer Communications, Special Issue on Advances in Research and Application of Network Security (2000)
25. Chess, D., Harrison, C., Kershenbaum, A.: Mobile agents: Are they a good idea? In Vitek, J., Tschudin, C., eds.: Mobile Object Systems: Towards the Programmable Internet. Volume 1222 of Lecture Notes in Computer Science., Springer-Verlag (1997) 25–45
26. Formal Systems (Europe) Ltd. 3, Alfred Street, Oxford. OX1 4EH, UK.: FDR2 User Manual. (2000)
27. Hoare, T., Jifeng, H.: Unified Theories of Programming. Prentice Hall (1998) ISBN: 0-134-58761-8.
28. Tang, X., Woodcock, J.: Travelling processes. In Kozen, D., ed.: The 7the International Conference on Mathematics of Program Construction. Lecture Notes in Computer Science, Stirling, Scotland, UK, Springer-Verlag (2004) To Appear.
29. Barnes, F., Jacobsen, C., Vinter, B.: RMoX: a Raw Metal occam Experiment. In Broenink, J., Hilderink, G., eds.: Communicating Process Architectures 2003. WoTUG-26, Concurrent Systems Engineering, ISSN 1383-7575, Amsterdam, The Netherlands, IOS Press (2003) 269–288 ISBN: 1-58603-381-6.
30. Welch, P.: UKC-CRG-01-04-2004: Suspending Networks of Parallel Processes. Technical report, Computing Laboratory, University of Kent at Canterbury, UK (2004)