Communicating Process Architectures 2004
Ian East, Jeremy Martin, Peter Welch, David Duce, and Mark Green (Eds.)
IOS Press, 2004

Communicating Mobile Processes

Fred R.M. BARNES and Peter H. WELCH

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

{frmb, phw}@kent.ac.uk

Abstract. This paper presents a new model for 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. channels and barriers) interface and that remains constant. The environment behind that interface will (usually) be completely different. The mobile process itself may contain any number of levels of dynamic sub-network. This model is simpler and, in some ways, more powerful than our earlier proposal, which required a process to terminate before it could be moved. Its formal semantics and implementation, however, throw up extra challenges. We present details and performance of an initial implementation.

1 Introduction

occam-π is a sufficiently small language to allow experimental modification and extension, whilst being built on a language (classical occam) of proven industrial strength. It integrates the best features of CSP [1, 2] and the π-calculus [3], focussing them into a form whose semantics is intuitive and amenable to everyday engineering by people who are not specialised mathematicians — the mathematics being built into the language design, its compiler, runtime system and tools, so that users benefit automatically from that foundation. The new dynamics broadens its area of direct application to a wide field of industrial, commercial and scientific practice.

Our earlier model [4] for mobile processes requires them to terminate before they could be moved. This gives a simple and intuitive semantics for activation:

```
SEQ
  c ? x             -- mobile process arrives
  x (...)           -- process x (...) runs from start to finish
  d ! x             -- mobile process departs
```

and a relatively simple implementation. However, for there to be a purpose behind these mobiles, they have to be able to maintain some (passive) state that survives their termination, movement and reactivation. To achieve this necessitates a *class-like* syntax for those mobile processes, involving private fields (for persistent state), constructors (for initialising that state) and methods (for activation). Such mobiles do not suffer the problems associated with object-orientation — such as leaky encapsulation, aliasing and concurrency blindness — and may be interesting in their own right. A denotational semantics for them, based on Hoare and He's *Unified Theories of Programming* [5], has been constructed that naturally

supports system development and verification by refinement from formal specifications [6]. Accordingly, we are likely to combine this earlier model with the new one and we show in sections 2 and 3.1 below how this combination may be done cleanly. Readers unfamiliar with this earlier model may safely ignore these comparisons.

Our new proposal has a slightly less crisp semantics for mobile process activation:

```
SEQ
  c ? x             -- mobile process arrives
  x (...)           -- process x (...) runs from somewhere to somewhere
  d ! x             -- mobile process departs
```

where the *somewheres* are either the *start* of the process, a *suspension-point* or *termination*. The value of the process variable 'x', after it has been input, is (the CSP expression) P/t, where P is the original process and t is the trace it has executed so far (and elsewhere!).

Process types are as in [4] — just PROC header templates. A MOBILE process implementing such a type has an extra primitive it can invoke — 'SUSPEND'. When that happens, the activation '*early terminates*', retaining its state and program counters. This corresponds to a strong occam intuition: that it is very powerful to express process state as a combination of its data values *and* where it is in its code.

The process may now be moved by normal communication down a channel carrying its process type. It may then be reactivated by the receiving process, after plugging it into a local environment. The process resumes execution from where it suspended with its own state unchanged, but with its external synchronisations bound to the new environment[1].

1.1 Implementation Issues

When thinking how to support mobile processes a long time ago, we were concerned about the danger of race hazard between asking a process to suspend and its subsequent movement — how could we be sure it had really suspended (and was not in the middle of some crucial transaction with its current environment)? Note, however, the tenses in the abstract of this paper. Mobile processes have to SUSPEND themselves — they are not suspended by their environment, which would simply not be safe. External processes may ask a mobile to suspend, but the mobile must do it itself and may take its time. When it suspends, control automatically passes to the process that activated it, which may now safely move it.

We are particularly grateful to the insight Tony Hoare gave us for handling a mobile process that has gone parallel internally. Our earlier model handled this by waiting for full termination — i.e. a multi-way synchronisation on the termination event of all internal processes. So, treat SUSPEND also as a multi-way synchronisation bound to all the internal processes — they *all* have to suspend for the whole mobile to suspend. For implementation, we just need a CSP event (an occam-π 'BARRIER') reserved in the workspace of any mobile process. To reactivate the mobile, all its suspended processes will be on the queue held by that event — easy!

Well, not quite that easy. Processes — even mobile processes — are very lightweight mechanisms in occam-π and, currently, are not location independent. A complete occam-π system is, of course, location independent but individual processes have many things addressed relative to the base of the whole system, not the individual process. Moving a process to a new memory space (or 'CLONE'-ing it) means that its workspace is allocated elsewhere and pointers will have to be adjusted. Moving processes across soft channels to a process in the same memory space is no problem.

[1]The allowed parameters are restricted to synchronisation types only — e.g. channels and barriers (see section 2.2).

Of course, we have to arrange a '*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 arranging for parallel termination [7]. This is left for the mobile application to implement; it's not our concern as mobile process language designers. We will think about providing language support for such distributed decisions. But that is orthogonal to the issue of mobile processes.

1.2 Structure of this Paper

Section 2 describes how mobile processes are declared, initialised and activated. The 'mobile' aspect of these mobile processes — i.e. moving them around a process network — is described in section 3.

Section 4 discusses some of the potential applications of this technology, with respect to existing "mobile agent" ideas and practice.

An overview of the implementation of these mobile processes is given in section 5. Section 6 draws some preliminary conclusions and lays out intended future directions for this work.

2 Defining Mobile Processes

Mobile processes may be defined in one of two ways. The first, described in [4], allow a single mobile process to support multiple *implementations* of *process-types*. That method of defining a mobile process allows the various implementations to share the state that persists between activations. That state must be declared outside of the individual implementations, however. Process-types provide the type system for mobile processes, and describe the interface to that process. For example:

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

This declares a process-type called '`IO`', whose implementations must match the '`PROC`'-style signature given, i.e. one `INT` input channel and one `INT` output channel — the formal parameter names need not match. Like other types in occam, two similarly structured but differently named process-types are not considered compatible.

The second method of defining a mobile process, described here, only allows a single implementation. This can be viewed as a 'shorthand' syntax for single-implementation mobile processes declared using the first method — and where there is no shared state. Instead of writing, for example:

```
MOBILE PROC integrate                                    -- earlier model
  ...  persistent/shared state (in this case, empty)
  IMPLEMENTS IO (CHAN INT in?, out!)
    ...  process body
:
```

we would instead write:

```
MOBILE PROC integrate (CHAN INT in?, out!) IMPLEMENTS IO    -- new model
  ...  process body
:
```

Note that 'CONSTRUCT' blocks (section 2.1 and [4]) are not needed for these mobiles, since there is no shared or persistent state to initialise.

Furthermore, the "IMPLEMENTS IO", is not strictly required. The compiler can check this when a mobile process is allocated. It also permits a single mobile process to support multiple implementations, provided that the formal-parameters are compatible. For example:

```
MOBILE IO p:
SEQ
  p := MOBILE integrate
  ...  process using p
```

The type of 'p' is always well-known — occam-π does not support type polymorphism — thus the compiler can easily (and statically) check that 'IO' matches the formal parameters of 'integrate' at the point of its allocation (the 'MOBILE' assignment). The declaration of the mobile 'integrate' process might be simplified even further to an ordinary 'PROC' declaration. For example:

```
PROC integrate (CHAN INT in?, out!)
  ...  process body
:
```

This allows any PROC to be *mobilised*, providing its interface matches the corresponding process-type. However, there are arguments that it may be good programming practice always to include an 'IMPLEMENTS' when declaring a mobile process — as well as identifying explicitly that the process is 'MOBILE'. We may specifically require the latter to enable some optimisations planned for later stages of our implementation.

2.1 Allocating Mobile Processes

As shown above, mobile processes are dynamically allocated using a special form of assignment. This follows in a similar way to other 'MOBILE' allocations, e.g. for dynamic mobile arrays [8] and mobile channel-types [9, 10]. The 'fuller' version of mobile processes (described in [4]) requires the separately declared and persistent state to be initialised using a 'CONSTRUCT' block. This is called at the point of allocation, passing any parameters given. For example, consider the following mobile process definition:

```
MOBILE PROC integrate
  INT total:            -- persistent state

  CONSTRUCT (VAL INT i)
    total := i

  IMPLEMENTS IO (CHAN INT in?, out!)
    ...  process body
:
```

A variable of the 'IO' type could be allocated with:

```
MOBILE IO p:
SEQ
  p := MOBILE integrate (0)
  ...
```

The simplified version of mobile processes presented here does not need this type of allocation. Allocation is simply "`p := MOBILE integrate`", without any parameters. Initialisation of any internal state follows the normal (and natural) pattern of being the first thing the process does the first time it is activated

Following the allocation of a mobile process, the process variable ('`p`' in the above code fragment) is used purely in terms of its process-type (e.g. '`IO`'), not the process that created it (e.g. '`integrate`'). The only data relating to '`integrate`' stored inside '`p`' are the process entry-point, run-time memory requirements and a pointer to a 'workspace-map' for the process. These are, of course, not the concern of the programmer using these mobiles. Details are covered in section 5.

Figure 1 shows this process ('`p`') from its own point of view. The channels are not real channels as such, rather they are placeholders for channels that will be "plugged in" when the process is activated — discussed in the following section.

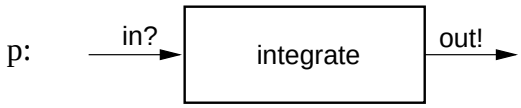

Figure 1: Value of mobile process variable '`p`' after allocation of '`integrate`'

2.2 Activating Mobile Processes

Mobile processes are *activated* by applying the process variable to a set of local arguments. This binds the mobile process to a *local* environment for the duration of the activation. For example:

```
CHAN INT to.p, from.p:
PAR
  local.environment (to.p!, from.p?, ...)

  MOBILE IO p:
  SEQ
    ...  p acquires some value
    p (to.p?, from.p!)
    ...
```

Figure 2 shows this mobile process active and connected to a local environment.

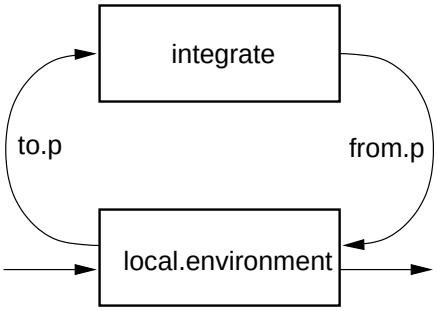

Figure 2: Active mobile 'integrate' process

3 Communicating Mobile Processes

Mobile processes in occam-π follow the semantics of existing mobiles — they are *moved* rather than *copied*, when assigned or communicated. They also share parts of the existing implementation for mobiles (section 5).

Figure 3 shows an example process network containing two processes connected by a channel. In this example, the channel carries mobile processes:

```
CHAN MOBILE IO c:
PAR
  A (c!)
  B (c?, ...)
```

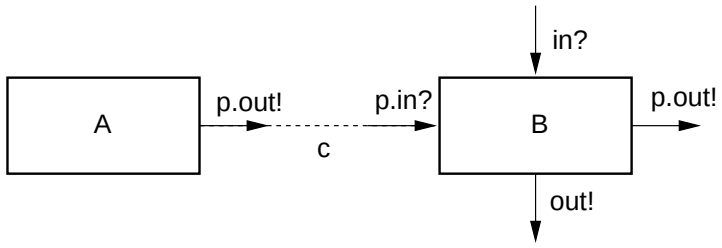

Figure 3: Process network for communicating mobile processes

The ‘A’ and ‘B’ processes are implemented such that ‘A’ creates a new mobile process and communicates it to ‘B’, that then activates it. For example:

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

```
PROC B (CHAN MOBILE IO p.in?, p.out!,
        CHAN INT in?, out!)
  MOBILE IO v:
  SEQ
    p.in ? v
    v (in?, out!)
    p.out ! v
    -- v is no longer defined
:
```

Note that the ‘B’ process is unaware of the actual implementation — it only knows how to connect with the process (given by the ‘IO’ process-type). This is one reason why it may turn out to be good programming practice to indicate explicitly what process-type a mobile process implements (e.g. “IMPLEMENTS IO”). That way, ‘B’ can at least be sure that the process it has in its ‘v’ variable is one that was intended to implement ‘IO’ — as opposed to a process whose interface is structurally the same, but whose behaviour is entirely different (as that could lead to deadlock when interfacing with the process). Section 3.2 examines this in more detail.

3.1 Suspending Mobile Processes

A serial implementation of ‘integrate’ involves a “WHILE TRUE” loop. For a mobile process, this would normally mean that once activated the process would never terminate. For most applications of such a mobile, this would be undesirable.

The mobile processes described in [4] support suspend/resume through explicitly persistent state that survives termination and re-activation. For example:

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

MOBILE PROC integrate.suspend.0
  INT total:                        -- persistent state

  CONSTRUCT ()                      -- simple constructor
    total := 0

  IMPLEMENTS IO.SUSPEND (CHAN INT in?, out!, CHAN BOOL suspend?)
    INITIAL BOOL running IS TRUE:
    WHILE running
      PRI ALT
        BOOL any:
        suspend ? any
          running := FALSE
        INT v:
        in ? v
          SEQ
            total := total + v
            out ! total
:
```

This is adequate for many purposes, even though the syntax is slightly cumbersome. One of the reasons for defining mobile processes this way is so that state may be shared between several implementations — this '`integrate.suspend.0`' has only one. There is a further, more subtle, problem however — if the mobile process goes parallel internally, those parallel processes must be shut-down before the mobile process can terminate.

The mobile processes described here, which need no special support for persistent state, cannot suspend through termination — all its state would go out of scope and be lost! Instead, a mechanism for explicitly suspending a process mid-execution is provided. The above process, for example, now becomes:

```
MOBILE PROC integrate.suspend.1 (CHAN INT in?, out!, CHAN BOOL suspend?)
IMPLEMENTS IO.SUSPEND
  INT total:
  SEQ
    total := 0
    WHILE TRUE
      PRI ALT
        BOOL any:
        suspend ? any
          SUSPEND               -- suspend process
          -- re-activates here
        INT v:
        in ? v
          SEQ
            total := total + v
            out ! total
:
```

If control reaches the `SUSPEND` line, the process suspends execution, retaining all local state and its resumption address (in much the same was as it does when de-scheduled) but returns control to its invoking process. The invoking process may communicate the mobile to a new location, where the receiving process may re-activate it by invocation on its own set of

arguments. The mobile then resumes execution from where it suspended with its local state unchanged. The channels bound to its parameters will (usually) be different — but that is of no concern to the mobile, whose semantics are defined with respect to its parameters and not the actual arguments supplied.

We believe that the mechanism for '`integrate.suspend.1`' is a little clearer and more natural than that for '`integrate.suspend.0`'.

3.2 Mobile Contracts

`PROC TYPE` interfaces define only the *connections* that are required and offered by the mobile. They do not define *how* those connections are used nor, indeed, how the values generated by the mobile relate to values received. We have just described three levels of specification that refine each other: the most general being the *Connection* (defined by the `PROC TYPE`), then the *Contract* (definable by a CSP specification of the behaviour of the mobile in terms of the events parameterised by its `PROC TYPE`), and finally the *Function* (definable by a Z specification of the mobile as a state machine). These all integrate nicely into the Circus algebra of Woodcock et al. [11].

For the safety of both the mobile process itself and its hosting environment, a *Connection* specification is insufficient. For flexibility, a *Function* specification is too constraining — we want to allow differently functioning mobiles (e.g. with successive bug-fixes) to be delivered to and activated in any host environment. Otherwise, mobile processes offer nothing new; we could have a static (conventional) process and just move around passive data.

A *Connection* interface is insufficient because the hosting environment needs to be sure that a mobile process will behave properly when invoked (connected) to its local environment — i.e. that it will not cause deadlock or livelock, will not starve any local processes of its attention and will suspend when asked. Of course, reciprocal promises by the hosting environment are equally important to the mobile. We call those promises a *Contract*.

CSP is sufficiently rich to enable the specification of such good behaviours. Model checkers (such as FDR [12]) are sufficiently powerful to check that *Contract* conforming hosts and mobiles will indeed be safe.

We are looking to boost the `PROC TYPE` of a mobile to include such a contract. For example, a contract on '`IO.SUSPEND`' might be that it is a *server* on its '`in?`' and '`suspend?`' channels, responding to an '`in?`' with an '`out!`' and to a '`suspend?`' with *suspension*. This could be strengthened to indicate its priorities for service. Or weakened to specify just its traces. Or weakened further to require only that the number of '`in?`' events in a trace can never be less than the number of '`out!`' events and that a '`suspend?`' may only occur when the number of '`in?`' events equals the number of '`out!`' events.

A behaviour we may want to prohibit in such a *Contract* is that of a '`suspend?`' (and, therefore, suspension) occurring in-between an '`in?`' and its corresponding '`out!`'. That way the host environment will know that the mobile will not suspend with an answer outstanding[2].

Without such a contract, an '`IO.SUSPEND`' mobile could arrive that always refuses its '`kill?`' channel (and could never be removed by its host!) or starts with an '`out!`' (and deadlocks with its host!).

We are considering extending the definition of `PROC TYPE`s to include some level of contract that the compiler can verify against implementing mobiles — but this is outside the scope of this paper. We note that these notions of behavioural contract would be valuable for *all* `PROC`s, mobile or not mobile.

[2]Note that this behaviour is honoured by '`integrate.suspend.1`' above.

3.3 Suspending Mobile Networks

The ‘IO.SUSPEND’ implementing processes given above are toy examples, but they illustrate mobiles that offer services and gather information. They have also been chosen because their base function (a running-sum integrator) can be implemented as a simple feedback network of *stateless* processes — and is a common teaching example.

We use it here to illustrate suspending a mobile that has gone parallel. The *graceful termination* [7] algorithm can be modified to provide secure distributed suspension. However, that algorithm was not concerned with saving state information — it was only concerned with termination for which the subsequent state of the processes was irrelevant! Here, we must take care to preserve state (in this case, the running total) *and* to honour a minimum level of the contract described in the previous section (i.e. that suspension must not come between an ‘in?’ and its matching ‘out!’).

There are many ways to to do this, figure 4 shows one.

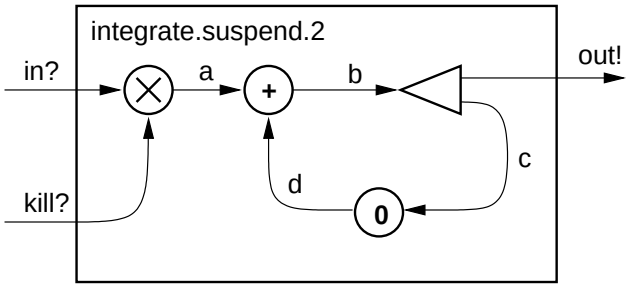

Figure 4: Suspendable parallel mobile integrator

The code for this parallel version of the integrator is:

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

MOBILE PROC integrate.suspend.2 (CHAN INT in?, out!, CHAN BOOL suspend?)
IMPLEMENTS IO.SUSPEND

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

The ‘OK.INT’ protocol tags INTs with a boolean, indicating whether the data carried is a *suspend* signal or *live* data. All sub-processes in the network remain stateless. The feedback loop holds the state of the whole mobile network — even during suspension (the propogating *suspend* signal carrying that state). Channel ‘d’ could have been ‘OK.INT’ as well. However, since ‘prefix.suspend’ only ever outputs *live* data, this has been optimised to just ‘INT’.

The ‘suspend’ process monitors its inputs and reacts in an obvious way:

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

Note that it prioritises its service in the same way as the serial (and stateful) ‘integrate.suspend.1’. The ‘plus.suspend’ is a simple modification to the standard adder process:

```
PROC plus.suspend (CHAN OK.INT in.0?, CHAN INT in.1?, CHAN OK.INT out!)
  WHILE TRUE
    BOOL b:
    INT x.0, x.1:
    SEQ
      PAR
        in.0 ? b; x.0
        in.1 ? x.1
      IF
        b                         -- live data received
          out ! TRUE; x.0 + x.1   -- send live data
        TRUE                      -- suspend signal received
          SEQ                     -- send suspend signal ...
            out ! FALSE; x.1      -- (carrying the running-sum)
            SUSPEND
:
```

Note: the graceful termination algorithm requires waiting for the ‘kill’ signal to return, discarding other data that arrives. For this application, we know that no other data exists, so that only the suspend signal would return. That return has been optimised away here. This also means that this component does not need to remember the running-sum state (‘x.1’) and remains stateless. The remaining processes now write themselves:

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

```
PROC prefix.suspend (VAL INT n, CHAN OK.INT in?, CHAN INT out!)
  SEQ
    out ! n
    WHILE TRUE
      BOOL b:
      INT x:
      SEQ
        in ? b; x
        IF
          b                       -- live data received
            SKIP
          TRUE                    -- suspend signal received
            SUSPEND
        out ! x                   -- send data
:
```

So, the running-sum state is actually held in '`prefix.suspend`' when the mobile network is moved. From the point of view of '`prefix.suspend`' it is *stateless* — i.e. it retains no data between its cycles. Within each cycle, it just reacts to the input received with an output — albeit with a suspension point in-between, depending on the type of input received.

The '`integrate.suspend.2`' mobile network gracefully suspends when its environment offers a '`suspend`' signal. It does this without deadlocking (which would certainly 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 `PAR` by all sub-processes *except* for '`prefix.suspend`' (where deadlock would result since the output would never be accepted).

This shows the care that needs to be taken in devising and implementing a safe suspension of all processes in a mobile network. However, this is a different responsibility from the actual mobile suspend mechanism. Responsibility for the former rests, for the moment, on the application engineer. We are investigating design (and, hopefully, language) rules to assist.

Finally, we note that initiation of a `SUSPEND` need not only come from the environment of the mobile. It could be a unilateral decision by the mobile itself (subject, of course, to satisfying any declared behavioural contract with its current environment) or initiated by the mobile and negotiated with its environment.

4 Applications

The most commonly understood meaning of the term "mobile agent" is primarily that of code and data mobility, as described by White in [13]. The main focus of which is on mobility of code and data between nodes in a distributed system, and where the infrastructure for handling mobile agents is provided largely by the application and libraries, not by the language itself. As noted by Jansen and Karygiannis in [14], there are many security issues relating to mobile agents, that should be addressed by any system wishing to support these agents. Generally, these can be divided into two categories — those that relate to the environment (e.g. admittance of an agent for execution) and those that relate to the agent (e.g. its interaction with the environment once 'connected').

The mobile processes of occam-π can provide an equivalent functionality[3]. Furthermore, mobile processes offer a comparatively secure implementation, as a direct consequence of using the occam-π language. For example, there is no way mobiles can access resources

[3]At the time of writing, there is no support for migration of mobile processes over networks. We hope to implement this in the near future, however (section 6).

to which their hosts do not explicitly provide connection. Connection and activation of the mobile is wholly under the control of the host.

4.1 Mobile Agents and Agent Platforms

Within the wider "mobile agent" community, an *agent* is the mobile (as expected), and an *agent platform* is the environment in which that mobile agent executes. This maps cleanly onto occam-π — agents are mobile processes and the agent platform is any process that activates a mobile. Mobile processes may also activate other mobiles, becoming "agent platforms" themselves.

The agent platform exists for two main purposes: firstly, to allow agents to interact with the system providing the agent platform; secondly, to allow agents to interact with each other. occam-π can support both types of interaction easily. When activated, mobile processes attach to the local environment (providing 'agent – platform' interaction), and that environment may allow agents to interact with each other (providing 'agent – agent' interaction). Within occam-π, agents may also form links directly between each other — the platform then serves as a mechanism to allow agents to find each other.

Figure 5 shows an example mobile agent system, where the 'platforms' could either be processes in a single system, or distributed over a network. Within the scope of other research (investigation and modelling of nanite assemblies on a grand scale) we are exploring systems containing millions of 'agent' processes and 'platforms'.

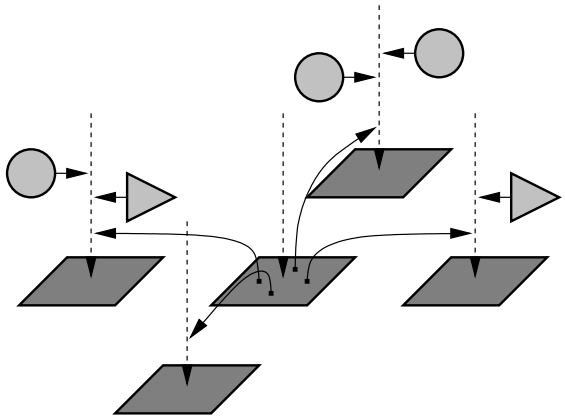

Figure 5: Example mobile agent system

In such systems, each 'platform' is a process constructed into a *matrix* of processes defining the topology of the space over which mobile agent processes roam. The matrix nodes are (mostly passive) servers, in touch with neighbouring nodes and on which arriving agents register. An agent attaches to one matrix node at a time, through which it can sense the presence of other agents and, hence, connect and interact as it chooses (using agent-specific protocols to avoid deadlock).

Matrix-agent protocols will be generic. Agents may enrol and resign from local (or global) barrier synchronisations to maintain a sense of time — as well as move and reproduce according to their own rules. Matrix nodes may also have their own agenda, allowing them to be pro-active in reshaping the space they define (e.g. through the creation of worm-holes) for more exotic environments.

We want to investigate the emergent properties of the system as a whole, rather than the behaviour of individual processes. This will require very large numbers of mobile processes — our current aspirations are for the order of 10^8 processes. The usefulness of networked

distribution here is in increasing the overall size of the system. However, we have to take care to minimise the effects of latency as processes migrate between nodes.

Another practical example of the use of mobile agents in distributed systems is in organising meetings, for example. When someone wishes to schedule a meeting, they send out an agent to a 'calendar' platform. This agent then waits for other user's agents to 'check-in' with the calendar, and negotiates suitable times. In practice, agents may need to remain connected to the calendar for some time — to wait for other agents and the finally decided (meeting) time. This leads to other issues, such as how a user's agent finds its way back to the user — the user may have moved. One possible solution would be a 'directory' platform, that can direct agents back to their user — users update a local directory whenever they connect or disconnect, and this information propagates between directories over time. As long as users migrate more slowly than their agents, such a system will work. Such issues are beyond the scope of this paper, however.

We are also exploring the use of mobile processes (as an agent mechanism) in RMoX [15], where having such mechanisms at the operating-system level may be useful — both for application and inter-RMoX use.

4.2 Security

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 [14, 16] and [17]. This section covers *some* of these issues. 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. Processes that activate mobile agents may safely assume that the agent is valid — because that agent was either created locally or came from another part of the system. Correspondingly, an agent may assume that whoever activates it was meant to do so. In a networked environment, possibly connected using public networks (e.g. the internet), the part of the system that manages network connections is responsible for ensuring the integrity of data communicated over networked channels (where the 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 minimised. Instead of communicating a serialised agent whole, the (source) code for the process could be sent, along with the saved state of the agent, and used to re-create the agent locally. The occam-π compiler can make certain guarantees about code it compiles, that compilers for many other languages cannot — e.g. that the code is generally safe and interacts with its environment in the intended way.

The use of a synchronisation-only interface to mobile processes limits many of the threats associated with existing agent systems. The widespread use of sequential programming languages (such as C) has led to a generally sequential interfacing for mobile agents. This would typically be realised using a "procedure-call" style activation of agents, that limits basic interactions to input on procedure call and output on procedure return. Concurrent interaction is still a possibility, but requires non-standard (or non-language controlled) code. One possible option would be to use RPC [18], as suggested in [16]. This does not (directly) permit the use of synchronisations between an agent and its environment, however.

The use of a synchronisation interface separates the activation of a mobile process from interaction with it, although the two are closely related — and there is (theoretically) no limit

to the amount of interaction that may occur during a single activation. To ensure correctness of those interactions, the 'TRACES' extension described in [19] could be used — although this is unsupported by the current occam-π compiler. This extension would enable to the compiler to check that both the agent and its environment conform to some pre-defined pattern of interaction (by specifying the CSP *traces* of those interactions). This mechanism, once implemented, will be able to define an important part of the *Contract*, described in section 3.2.

5 Implementation

Supporting the mobile processes as described here has required reasonably large modifications to the occam-π compiler used by KRoC [20, 21]. The most complex of these modifications is supporting the 'SUSPEND' functionality — which if not carefully managed could result in disaster (e.g. new activations contaminated with an old environment).

Mobile process variables (e.g. "MOBILE IO p") are implemented in a similar way to mobile channel-end variables — a single word in the process workspace that points to a block of dynamically allocated memory. This dynamically allocated block, ranging from 12 to 16 words in size, contains general information about the process, pointers to the process memories, and a *barrier* [22] that is used to hold suspended processes. Figure 6 shows the structure of this mobile-process descriptor.

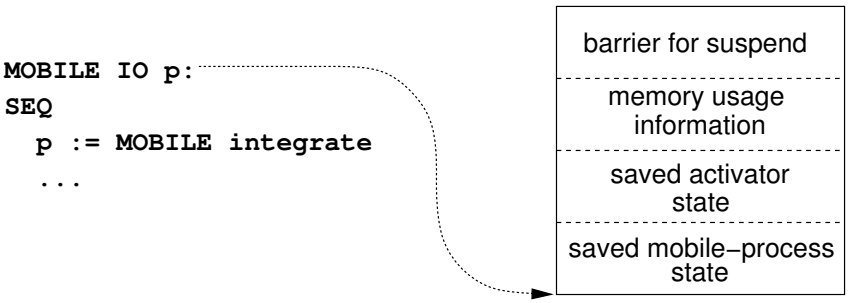

Figure 6: Structure of mobile-process descriptor

Mobile processes that do *not* suspend require very little special treatment — to the point where their activations are treated as ordinary PROC calls. However, once that mobile process has terminated, it may not be re-activated. This is implemented by setting the *reactivation-address* of the terminated process to code that raises a run-time error, or in 'stop' error-mode, deschedules the process attempting the activation.

Processes that *do* suspend require a certain amount of special treatment, largely in the code-generator. Because a mobile process may suspend mid-execution, there is a potential problem with parameters that have been abbreviated internally. Returning to the earlier parallel 'integrate', for example:

```
MOBILE PROC integrate.suspend.2 (CHAN INT in?, out!, CHAN BOOL suspend?)
IMPLEMENTS IO.SUSPEND
  CHAN OK.INT a, b, c:
  CHAN INT d:
  PAR
    suspend (in?, kill?, a!)
    plus.suspend (a?, d?, b!)
    delta.suspend (b?, c!, out!)
    prefix.suspend (0, c?, d!)
:
```

When this code is activated for the first time, it sets up a network of sub-processes, connecting channels in the interface to those sub-processes. The usual implementation of channel parameter-passing (and abbreviations) is to simply copy the channel-pointer. When the sub-processes 'SUSPEND', the mobile process is shut-down and control returns to the activating process. A subsequent reactivation may be to a different environment, that would render the interface originating channel-pointers inside 'plus.suspend', 'delta.suspend' and 'int.suspend' useless — as they contain channel addresses that came from the environment of the first activation.

This problem is solved by adding an extra layer of indirection in the implementation of channel-parameters. That is, instead of passing a channel-address as a parameter, a *pointer* to the channel-address is passed. This does not apply to the activation, however — that gets channel-pointers for channel parameters. When setting up the sub-processes, the compiler passes addresses that are inside the workspace of the mobile process, for both external and local channels. Local channels require an additional address temporary, since they are the channel themselves.

Inside the sub-processes, the indirect channel-pointers must be dereferenced before communication is attempted. Three new ETC [23] 'specials' have been added for this, that dereference the virtual-transputer A, B and C registers respectively. These instructions are generated immediately prior to communication. The compiler could produce the same result using the traditional "LDNL 0" (load non-local at word offset 0). This, however, would require a more substantial modification to the compiler, that is currently largely unaware of general pointers-to-pointers.

5.1 Suspending Mobile Processes

The idea to collect suspended parallel processes on a barrier — in order to support suspension of entire process networks, rather than just a single process — was suggested by Tony Hoare and documented in [24]. When a mobile process is initially created, the barrier's 'enrolled' count is set to 1. The compiler automatically generates code to enroll and resign parallel processes as they are created and destroyed. When the last parallel process 'SUSPEND's, or resigns, it will complete the barrier and return control to the activating process. When reactivated, the processes blocked in the barrier are put back on the run-queue.

The code that enrolls processes on a barrier is relatively trivial — the barrier count is incremented by $n-1$, where n is the number of parallel processes. The code that synchronises (on 'SUSPEND') and resigns processes from a barrier is not trivial, since both may complete the barrier synchronisation. Rather than being generated in-line, this code is built into the run-time library (written in '*virtual transputer*' assembly language). Compile-time constants required by the code (e.g. offset of the 'count' field within the barrier) are made available as pre-processor variables. Having this code external to the compiler reduces the complexity of the compiler and size of the generated code, at the expense of a slightly longer run-time. However, having the operation identified explicitly (by a procedure call) allows for future optimisation — the native-code translator (tranx86) *could* replace the procedure call with an optimised native-code version of the operation.

5.2 Implementing Mobile Process Communication

For communication and assignment of mobile processes, existing mobile-related code and instructions are used. Within a single system, communication and assignment are simply the moving of a pointer (to the mobile-process descriptor) between processes (or variables in the case of assignment). As with other dynamic mobiles, "old" processes are freed rather

than being moved into source of the communication or assignment — as happens for static mobiles.

Mobile processes can be duplicated using the 'CLONE' operator. This returns a copy of the mobile process operand, that is left defined. Supporting 'CLONE' requires knowledge about the workspace (and possibly vectorspace and mobilespace) layout of the process, so that any pointer values are correctly adjusted and other dynamic (mobile) state also 'CLONE'd (a *deep* copy). The one restriction is that any mobile process containing an *unshared* channel-bundle end may not be cloned — because that channel-bundle end cannot be duplicated using 'CLONE' (since that would break its *unshared* semantics!).

'Serialisation' of mobile processes is possible using the built-in 'ENCODE.CHANNEL' and 'DECODE.CHANNEL' processes, that were developed for KRoC.net [25]. 'DECODE.CHANNEL' would input a mobile process, then output a dynamic mobile BYTE array containing the position-independent state of the process, in addition to general information about the process, e.g. memory-requirements and a reference to the code that implements the process. The serialised state although position-independent, will still be architecture and layout dependant — e.g. the byte-ordering in words and the layout of individual variables in the process's workspace. Thus when communicating mobile processes between heterogeneous architectures, some (non-trivial) conversion may be required.

5.3 Performance

The implementation of mobile processes is, on the whole, very lightweight. At the time of writing, the parts of the implementation that are in place are still somewhat experimental, and so have not been optimised.

Measured on an 800 MHz Pentium-III, the time required to create and destroy a basic process (that does not take any visible parameters) is around 450ns — and when the dynamic memory required is immediately available from one of the free-memory lists. Including a complete activation raises the time to around 550ns, giving an approximate activation-deactivation time of 100ns.

Including a SUSPEND and activating the process twice takes approximately 920ns. Thus, the time required to suspend a process and then reactivate it is approximately 370ns.

These times are for a very simple process — that does not have either vectorspace or mobilespace. For more complex mobile processes (e.g. those that take parameters) the time required to activate the process will increase.

6 Conclusions and Future Work

This paper has described a new model for mobile processes in occam-π, that provides code and channel mobility (ideas from the π-calculus) with the discipline and rigour of occam and composable semantics of CSP. Mobile processes complement mobile channels [4, 9, 10] the occam-π programmer with powerful new tools for directly, safely and efficiently capturing the dynamic aspects of complex large-scale systems — e.g. multi-layer modelling of micro-organisms and their environments (the *In Vivo* ⇔ *In Silico* Grand Challenge [26, 27]) and process migration (agents) in distributed systems.

Currently, there is limited support for creating 'CLONE's of a mobile process. This, and 'serialisation' using 'DECODE.CHANNEL' or other means, requires the suspend process to be made independent of its location in memory. In order to do this, the run-time system (or compiler-generated code) needs a *memory-map* for each of the process's memories that contain pointers — i.e. workspace, vectorspace and optionally mobilespace. The compiler-side

of KRoC.net [28] will also require this support so that it may transport mobile processes between separate memory-spaces.

At the time of writing, the compiler produces this information for some, but not all, pointer-types in the process workspace — `CLONE` works for very basic processes. We hope to complete this support, and therefore support for `CLONE` and serialisation in the near future.

Acknowledgements

We are grateful to Tony Hoare for his insights and advice on suspending parallel process networks [24]. We also wish to thank colleagues and other members of the concurrency research group, in particular David Wood, Christian Jacobsen and Mario Schweigler for their input and advice.

The authors would additionally like to thank the anonymous reviewers for their feedback on an earlier revision of this paper.

References

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

[2] Steve Schneider. *Concurrent and Real-time Systems — The CSP Approach*. Wiley and Sons Ltd., UK, Baffins Lane, Chichester, UK, 2000. ISBN: 0-471-62373-3.

[3] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes – parts I and II. *Journal of Information and Computation*, 100:1–77, 1992. Available as technical report: ECS-LFCS-89-85/86, University of Edinburgh, UK.

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

[5] C.A.R. Hoare. Unified Theories of Programming. Technical report, Oxford University Computing Laboratory, July 1994. Available at: `http://users.comlab.ox.ac.uk/tony.hoare/publications.html`.

[6] Xinbei Tang and Jim Woodcock. Travelling processes. In Dexter Kozen, editor, *The 7the International Conference on Mathematics of Program Construction*, Lecture Notes in Computer Science, Stirling, Scotland, UK, July 2004. Springer-Verlag. To Appear.

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

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

[9] F.R.M. Barnes and P.H. Welch. Prioritised Dynamic Communicating Processes: Part I. In James Pascoe, Peter Welch, Roger Loader, and Vaidy Sunderam, editors, *Communicating Process Architectures 2002*, WoTUG-25, Concurrent Systems Engineering, pages 331–361, IOS Press, Amsterdam, The Netherlands, September 2002. ISBN: 1-58603-268-2.

[10] F.R.M. Barnes and P.H. Welch. Prioritised Dynamic Communicating Processes: Part II. In James Pascoe, Peter Welch, Roger Loader, and Vaidy Sunderam, editors, *Communicating Process Architectures 2002*, WoTUG-25, Concurrent Systems Engineering, pages 363–380, IOS Press, Amsterdam, The Netherlands, September 2002. ISBN: 1-58603-268-2.

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

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

[13] Jim White. Mobile agents white paper, 1996. General Magic. `http://citeseer.ist.psu.edu/white96mobile.html`.

[14] Wayne Jansen and Tom Karygiannis. 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`.

[15] F.R.M. Barnes, C.L. Jacobsen, and B. Vinter. RMoX: a Raw Metal occam Experiment. In J.F. Broenink and G.H. Hilderink, editors, *Communicating Process Architectures 2003*, WoTUG-26, Concurrent Systems Engineering, ISSN 1383-7575, pages 269–288, IOS Press, Amsterdam, The Netherlands, September 2003. ISBN: 1-58603-381-6.

[16] Wayne A. Jansen. Countermeasures for Mobile Agent Security. *Computer Communications, Special Issue on Advances in Research and Application of Network Security*, November 2000.

[17] David Chess, Colin Harrison, and Aaron Kershenbaum. Mobile agents: Are they a good idea? In Jan Vitek and Christian Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet*, volume 1222 of *Lecture Notes in Computer Science*, pages 25–45. Springer-Verlag, April 1997.

[18] A.D. Birrell and B.J. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

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

[20] P.H. Welch and D.C. Wood. The Kent Retargetable occam Compiler. In Brian O'Neill, editor, *Parallel Processing Developments, Proceedings of WoTUG 19*, volume 47 of *Concurrent Systems Engineering*, pages 143–166. World occam and Transputer User Group, IOS Press, Netherlands, March 1996. ISBN: 90-5199-261-0.

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

[22] Peter H. Welch and David C. Wood. Higher Levels of Process Synchronisation. In A. Bakkers, editor, *Parallel Programming and Java, Proceedings of WoTUG 20*, volume 50 of *Concurrent Systems Engineering*, pages 104–129, Amsterdam, The Netherlands, April 1997. World occam and Transputer User Group (WoTUG), IOS Press. ISBN: 90-5199-336-6.

[23] M.D. Poole. Extended Transputer Code - a Target-Independent Representation of Parallel Programs. In P.H. Welch and A.W.P. Bakkers, editors, *Architectures, Languages and Patterns for Parallel and Distributed Applications, Proceedings of WoTUG 21*, volume 52 of *Concurrent Systems Engineering*, pages 187–198, Amsterdam, The Netherlands, April 1998. WoTUG, IOS Press. ISBN: 90-5199-391-9.

[24] P.H. Welch. UKC-CRG-01-04-2004: Suspending Networks of Parallel Processes. Technical report, Computing Laboratory, University of Kent at Canterbury, UK, March 2004.

[25] M. Schweigler, F.R.M. Barnes, and P.H. Welch. Flexible, Transparent and Dynamic occam Networking with KRoC.net. In J.F. Broenink and G.H. Hilderink, editors, *Communicating Process Architectures 2003*, WoTUG-26, Concurrent Systems Engineering, ISSN 1383-7575, pages 199–224, IOS Press, Amsterdam, The Netherlands, September 2003. ISBN: 1-58603-381-6.

[26] R. Sleep. In Vivo ⇔ In Silico: High fidelity reactive modelling of development and behaviour in plants and animals, May 2003. Available from: `http://www.nesc.ac.uk/esi/events/Grand_Challenges/proposals/ViSoGCWebv2.pdf`.

[27] P.H. Welch. Infrastructure for Multi-Level Simulation of Organisms, March 2004. Available from: `http://www.nesc.ac.uk/esi/events/Grand_Challenges/gcconf04/submissions/42.pdf`.

[28] M. Schweigler. Adding Mobility to Networked Channel-Types. In I. East, J. Martin, P. Welch, D. Duce, and M. Green, editors, *Communicating Process Architectures 2004*, WoTUG-27, Concurrent Systems Engineering, ISSN 1383-7575, IOS Press, Amsterdam, The Netherlands, September 2004.