Prioritised dynamic communicating and mobile processes

F.R.M. Barnes and P.H. Welch

Abstract: Continuing research on language design, compilation and kernel support for highly dynamic concurrent reactive systems is reported. The work extends the occam multiprocessing language, which is both sufficiently small to allow for easy experimentation and sufficiently powerful to yield results that are directly applicable to a wide range of industrial and commercial practice. Classical occam was designed for embedded systems and enforced a number of constraints, such as statically predetermined memory allocation and concurrency limits, that were relevant to that generation of application and hardware technology. This work removes most of these constraints and introduces a number of new facilities: explicit channel ends, channel bundles, mobile ends of channels and bundles, dynamic process creation, the extended rendezvous and process priorities. These significantly broaden occam's field of application and raise the level of concurrent system design directly supported. Concurrency overheads have been driven ever downwards, for example synchronising channel communication is now around 100 nanoseconds on an 800 MHz P3, and most operations have unit time cost. Finally, a proposal for secure mobile processes is made.

1 Background

Classical occam™ [1] is a concurrent programming language, based on the CSP process algebra [2–4]. It was originally designed for programming and reasoning about embedded systems, in particular, those using the Inmos transputer [5–7]. Safety aspects of those applications required occam programs to have statically determinable memory requirements. The transputer directly supported two levels of priority scheduling. If further levels were needed, they had to be programmed at the user level. Its process management and communication capabilities were so fast that the overheads this introduced were negligible – but having to worry about them at all was a distraction to the application designer. Nowadays, real-time programmers expect many more than two levels of priority and these are usually offered, although with somewhat imprecise semantics and levels of guarantee.

On modern commodity computing platforms such as specialised servers or general purpose workstations, programmers expect to have some mechanism for dynamic memory management, so that systems can allocate those resources in response to events happening at run-time. For languages such as Java [8], this happens fairly automatically; the programmer only needs to worry about creating new data or code structures within a program. Concurrent with user code execution, the garbage collector [9] clears away unused structures periodically to free-up memory. A negative side to this, however, is a loss of guaranteed timing behaviour when the garbage collector strikes, which is serious for real-time applications. At a slightly lower level, languages such as C [10, 11] and C++ [12] provide dynamic allocation, but leave its de-allocation to the mercy of the programmer, whose commonly incorrect usage leads to subtle but eventually catastrophic run-time errors. In all cases, dynamic memory management is ultimately performed by the operating system (the '`brk`' system-call on UNIX for example).

In light of this, classical occam appears restricted in terms of dynamic and real-time capability. Yet occam remains highly desirable for its simplicity (through concurrency), safety (enforced zero-aliasing and race-hazard-free code), and run-time performance, with context-switch overheads in the order of tens of nanoseconds. The KRoC [13] project, alongside others, is largely responsible for the continued interest and development in the occam language, since the transputer production line stopped. KRoC (the Kent retargettable occam compiler) comprises a number separate programs which generate native-platform binaries from occam sources. KRoC ports exist for a wide range of architectures, developed under the EPSRC-funded 'occam For All' project [14] which finished in May 1997. The work described here concentrates on the KRoC/Linux [15] port, primarily for the Intel i386 architecture, which has been developed since that time.

2 Overview and motivation

This paper describes four main extensions to the occam/CSP programming language: channel bundles (with mobile ends), dynamic process creation, an extended rendezvous and process priorities. The dynamic memory mechanisms, needed for mobile channels and dynamic process creation, also allow the introduction of recursive processes and run-time sized `PAR` process replication. The last-mentioned requires no significant language change and is described in a related paper [16] along with the tidying-

© IEE, 2003

IEE Proceedings online no. 20030182

DOI: 10.1049/ip-sen:20030182

Paper received 10 June 2002

The authors are with the Computing Laboratory, University of Kent, Canterbury, Kent, CT2 7NF, UK

up of a number of additional items from the original occam language. Note that these dynamic extensions are introduced with full maintenance of alias checking, zero race-hazard guarantees and the absence of both garbage collection and memory leaks. Finally, we also make a proposal for secure mobile processes (Section 9).

2.1 Fixed process networks

The traditional method of constructing occam programs follows a simple *process oriented* design pattern. This means (layered) networks of independent active processes that communicate and/or synchronise through the primitives available. In the KRoC [13] implementation of occam, those primitives are (CSP) channel communications and implicit barriers (at the ends of `PAR` constructs) plus some non-compiler based synchronisations (semaphores, barriers, buckets and resources), as described in [17]. These capabilities let us construct networks of fixed size and topology, which is indeed sufficient for the design and implementation of a wide range of system. However, we now want to be able to create and network processes dynamically.

2.2 Fixed process workspace and mobiles

Classical occam also lacks language constructs for pointer assignment and communication. A major reason for this was to control aliasing, which is easily introduced by the copying of pointers, and without whose rigorous policing, concurrent systems would die from race hazards. However, that omission has memory and run-time implications, since data must be copied rather than just references.

The introduction of *mobile* data-types to occam [18, 19] provides a safe non-aliasing *movement* semantics implemented using pointer manipulation, as well as the ability to create run-time sized mobile arrays. Parallel race hazards are not introduced since the sending process loses the reference pointer, i.e. no alias is made to the mobile data. Further, because the compiler always knows when data becomes inaccessible, code is automatically generated to recover its memory. This takes a very short and unit time, so real-time response remains calculable despite the dynamic allocation of all mobile structures.

2.3 Dynamic bundles of channels

Channel types (first proposed for the unimplemented occam3 [20] language) provide a method of grouping together a 'bundle' of related channels within a single structure – for example, a pair of opposing direction channels supporting a client/server interaction. Our channel types modify this idea: their variables reference only one of the ends of the bundle and those ends are *mobile*. To distinguish between the two different ends, a *channel direction specifier* ('`?`' or '`!`', described in Section 3) is added to the type when declaring the channel end.

These channel ends are similar in idea to Icarus' ports [21, 22], except that we allow an arbitrary number of channels within a single type. The channel types added to occam are treated as first-class citizens in the type system (like any mobile type), allowing channel bundle ends to be declared and subsequently communicated to other processes. Assignment and parameter passing are handled in a similar (*mobile*) way. Channel ends (either side) may also be `SHARE`d (another occam3 idea) by many processes, with safe access enforced by the language design.

Section 4 describes the syntax and semantics of channel types in occam, within the framework of KRoC/Linux [15].

2.4 Dynamic process creation

Standard occam only allows the creation of new processes inside a `PAR` construct, and forces them to synchronise at the end of the `PAR` on a barrier. Restrictions also apply to the replicated `PAR`, whose replicator count must be a compiler-known constant expression. An earlier paper [16] describes an extension for handling run-time-count replicated `PAR`s. Further information can be found in [23]. This greatly enhances the expressiveness of a replicated `PAR`, although it makes more difficult the compiler's task of parallel usage checking within the replicated process.

We introduce a new mechanism for process creation, the `FORK`, which dynamically creates a `PROC` instance and runs it parallel to the invoking process. Locally, `FORK` behaves as *Skip* (a do-nothing process), with some minor exceptions. Termination of a `FORK`ed process is controlled by a `FORKING` block. Before a `FORKING` block can finish, it must wait for any unfinished `FORK`ed processes to terminate. A `FORK`ed process may finish early however, allowing dynamically allocated resources to be reused immediately.

Section 6 describes the syntax and semantics of the `FORK`. A typical application for `FORK`s is in internet servers, where it is highly desirable to be able to spawn a new process for handling an incoming connection. An experimental version of the occam web-server [24] is under construction, using `FORK`s and channel-types.

With dynamic process creation, dynamic channel (or bundle) creation and the mobility of channel ends, we now have the technology for the dynamic construction of process networks of arbitrary scale and topology. Of course, care will still be needed.

2.5 Extended rendezvous

Simple channel communication in occam is described, following CSP, by processes participating in an event (the channel). The outputting process readies the data then synchronises on the channel. The inputting process synchronises on the channel and then takes the data. The point at which both processes synchronise is viewed as an instantaneous action where the data (or rights to the data) are copied. Both inputting and outputting processes then resume execution.

Sometimes it is desirable to have the inputting process perform some action on the received data, before the outputting process resumes execution. Currently, the only way to do this is to alter the inputting and outputting processes so that they perform an additional synchronisation at the required point. However, doing this is intrusive to the outputting process (which doesn't need to know such details about the inputting process), and there is every possibility that a misplacement of the second synchronisation on either side will cause deadlock.

The extended rendezvous offers a nice solution to this problem, since it requires no modification of any outputting process and provides a safe (compiler checked) syntax and semantics for the inputting process. Section 5 describes this in detail.

2.6 Process priorities

Process priority has always been necessary for the efficient scheduling of processes sharing the same processor and for computing worst-case response times in real-time applica-

tions. Strictly speaking, only two priorities are ever needed, but only if the application engineer is prepared to do additional scheduling operations on top of them. We have built in (as an option) 32 levels of priority to simplify this task, more levels could easily be added if necessary. Currently, priorities have not been burnt into the *language* but are available through a *library* of compiler *built-in* PROCs, plus significant changes in the run-time system (which have raised synchronisation overheads only marginally – still only tens of nanoseconds). This form of priority was first investigated in [25]. Section 8 describes our implementation of priority.

3 Channel direction specifiers

The occam programming language encourages programmers to build applications as layered networks of active/reactive processes (components), synchronising and communicating through channels. Classical occam channels provide point-to-point, synchronised, unbuffered communication between processes. Conceptually, a channel has two ends: the input end and the output end. Fig. 1 shows an example component, a *running-sum integrator*, as seen by processes outside of it. The implementation is hidden and is not the concern of the surrounding network, providing all implementations behave in a consistent manner (in their pattern of use of external channels and the functions they compute).

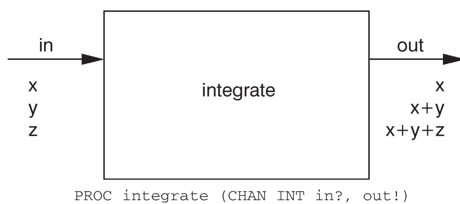

Fig. 1 occam *running-sum integrator*

An occam process sees only one end of any channel parameter it is given (either for input or output). That direction is specified using a *channel direction specifier* ('?' for input or '!' for output), along with the structure of the messages carried, in the case of Fig. 1, simple INTs. The directions are specified from the point of view of the process.

Classical occam did not have these direction specifiers. Their introduction enables much more accurate error messages to be reported by the compiler on silly mistakes.

Any implementation of 'integrate' must use its parameterised channels in the direction specified. For example, here is a serial implementation:

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

The compiler checks that the usage of the channel parameters ('in' and 'out') conforms.

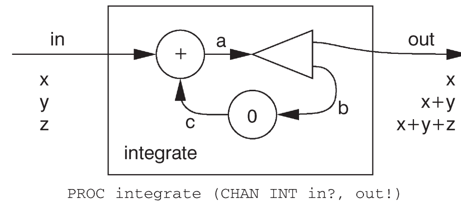

Fig. 2 occam *running-sum integrator showing internal processes*

A *parallel* implementation, as shown in Fig. 2 [Note 1], simply plugs its given channel ends into the relevant internal subcomponents. The subcomponents in Fig. 2 are very simple processes that cycle indefinitely, retaining no state between cycles. The adder component ('plus') waits for a number on each input and outputs their sum. The triangular process ('delta') waits for a number and outputs, in parallel, on its two output channels. The third process ('prefix(0)') initially outputs a zero; then it simply copies input numbers to its output channel. The code for this parallel implementation is:

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

The correct ends of any locally declared channels must be used when wiring things up.

The above code contains all the information necessary to draw the diagram in Fig. 2. The compiler will check that, for example, the delta process expects a channel *input end* for its first parameter and channel *output* ends for its second and third.

4 Mobile channel types

Here is an example of a channel bundle type:

```
CHAN TYPE BUF.MGR
  MOBILE RECORD
    CHAN INT req?:            -- integer
    CHAN MOBILE []BYTE buf!:  -- dynamic array
    CHAN MOBILE []BYTE ret?:  -- dynamic array
:
```

This declares a *mobile* channel type called 'BUF.MGR'. Being a channel type, the fields inside the RECORD structure are only permitted to be channels (or arrays of channels). *Data* fields are not allowed.

'BUF.MGR' contains three channels. 'req' is used by a *client* to request a buffer of some size. The buffer is acquired from the 'buf' channel. Once the client is done with the buffer, it sends it back to the server using the 'ret' channel. The channel direction specifiers specify *server*-relative directions. The *server* side can only use 'req' and 'ret' for input and 'buf' for output. In contrast, the client side can only use 'req' and 'ret' for output

Note 1: Strictly this parallel implementation of 'integrate' is not equivalent to the serial version. The difference concerns the buffering capacity of each implementation: 1 for the serial version and 2 for the parallel version. The serial version pipelined with a trivial one-place buffer process is, however, semantically the same as the parallel version.

and ‘buf’ for input. This behaviour is enforced by the compiler, as is the placement of direction specifiers on the fields within the channel-structure.

We use the words ‘*client*’ and ‘*server*’ here to mean a particular *end* of the channel bundle. Whether the application chooses to use them as such is entirely up to itself. We would prefer, however, that they were used according to a well-understood usage pattern, such as client–server, IO-SEQ or IO-PAR [26].

4.1 Variables and allocation

Channel type variables come in two forms, a *server-end* and a *client-end*. For example:

```
BUF.MGR? buf.svr:     -- server-end variable
BUF.MGR! buf.cli:     -- client-end variable
```

A direction specifier is used on the *type* to indicate either a server (‘buf.svr’) or a client variable (‘buf.cli’). Once declared, they are *undefined* until they are either allocated or used as a target of assignment or input. These are *mobile* variables and have the same underlying semantics as the mobile *data-types* described in [19]. They are allocated in pairs at run-time:

```
buf.cli, buf.svr := MOBILE BUF.MGR
```

This operation dynamically allocates the channel-structure record and assigns it to both target variables. Their use in assignment and communication is strictly controlled by the rules for ordinary MOBILE variables, i.e. a *movement* semantics: the source variable loses it (becoming undefined). The compiler is not fussy about the order in which the client and server variables appear on the left-hand side, but does check to ensure that one is a client and the other is a server. This allocation syntax is similar to general dynamic mobile allocation [19], except that in this case there is no array dimension to be specified and there are two results from the allocation instead of one.

The variables ‘buf.svr’ and ‘buf.cli’ can be used in two ways: either as themselves in communication, assignment, parameter passing and abbreviations, or with subscripts to access individual channel fields.

4.2 Using channel types

For the most part, channel type variables can be treated like ordinary mobile variables. Once allocated, this means that moving one of the channel-bundle ends around the network *stretches* the channels within it. The behaviour is conceptually like the similar mechanism in Icarus [21, 27], but our implementation differs significantly.

Fig. 3 shows a simple occam program which allocates a mobile channel-bundle (in ‘generator’), then communicates its ends to other (the ‘server’ and ‘client’) processes. Those processes then use the channels contained within the bundle to communicate directly with each other. The same BUF.MGR type described at the start of Section 4 is used.

Fig. 4 shows the state of things after the ‘generator’ process has allocated the channels, but before it has communicated them.

The ‘generator’ process then moves the server-end to the ‘server’ process. This has the effect of pulling the channels ‘over’ the network, as shown in Fig. 5. The client-end is then moved to the ‘client’ process and the ‘generator’ process terminates. The two processes (‘server’ and ‘client’) then proceed to communicate

```
PROC server (CHAN BUF.MGR? in?)
  BUF.MGR? sv:
  SEQ
    in ? sv                   -- get server-end

    INT s:
    MOBILE [ ]BYTE b:
    SEQ
      sv[req] ? s             -- get size
      b := MOBILE [s] BYTE    -- allocate buffer
      sv[buf] ! b             -- move buffer out
      sv[ret] ? b             -- take buffer back
:

PROC client (CHAN BUF.MGR! in?)
  BUF.MGR! cv:
  SEQ
    in ? cv                   -- get client-end
    MOBILE [ ]BYTE b:
    SEQ
      cv[req] ! 1518          -- send desired size
      cv[buf] ? b             -- get buffer
      ...  use ‘b’
      cv[ret] ! b             -- move buffer back
:

PROC generator (CHAN BUF.MGR? s.out!,
                CHAN BUF.MGR! c.out!)
  BUF.MGR? buf.svr:
  BUF.MGR! buf.cli:
  SEQ

    -- allocate mobile channel structure
    buf.cli, buf.svr := MOBILE BUF.MGR

    s.out ! buf.svr           -- move server-end
    c.out ! buf.cli           -- move client-end
:

CHAN BUF.MGR? svr.chan:
CHAN BUF.MGR! cli.chan:
PAR
  generator (svr.chan!, cli.chan!)
  server (svr.chan?)
  client (cli.chan?)
```

Fig. 3 *Simple mobile channel type demonstration program*

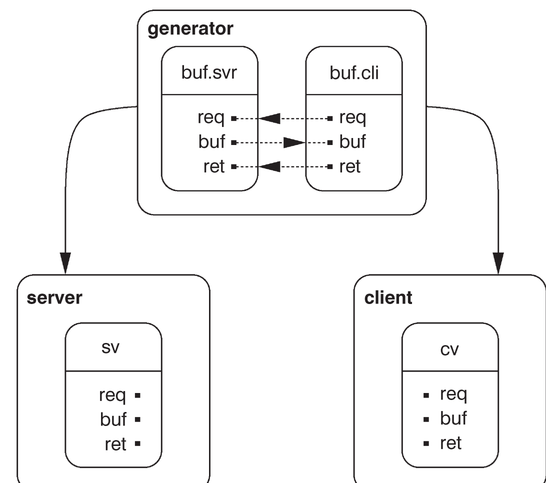

Fig. 4 *Process states after channel allocation*

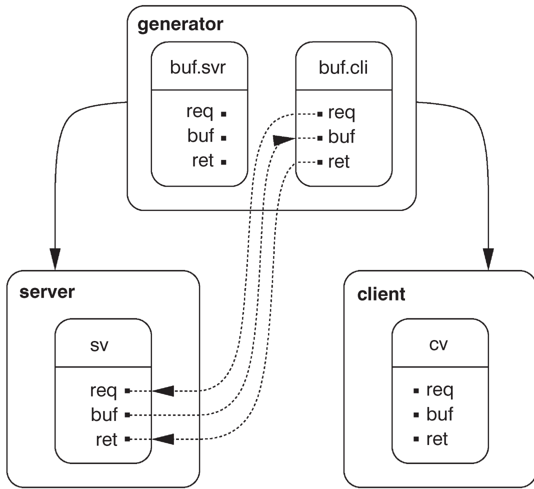

Fig. 5 *Process states after moving server-end*

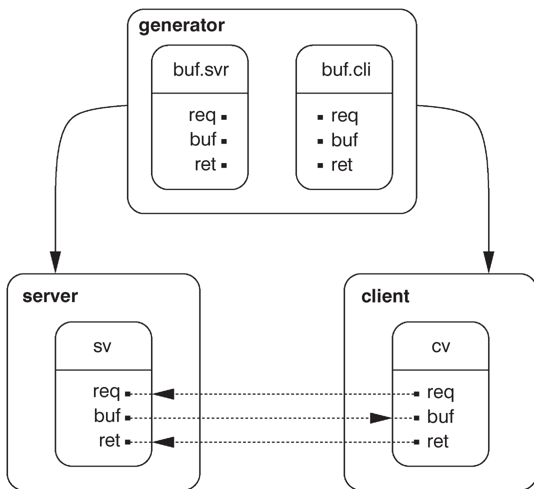

Fig. 6 *Final process states after moving client-end*

over the channels connecting them. This final network configuration is shown in Fig. 6.

This example is not particularly exciting, but demonstrates how channel-types can be used. The main point is that a process having one end of a channel (or channel bundle) does not need to know where the remote end is. Indeed, one of the general points of occam/CSP is that we should not need to know where channels are connected, only what protocols and usage patterns they have [Note 2]. This is what gives us a compositional semantics for occam concurrency and scalable complexity.

4.3 Shared channel-types

The channel-types presented so far provide a general mechanism for moving channels around networks, neatly grouped according to function. A common thing we want to do with channels is to share them. Sharing of channels (and any other variable, parameter or abbreviation) is currently performed using a compiler directive to turn-off usage checking and a (user-defined) `SEMAPHORE` [17, 29] type to provide mutual exclusion. We would clearly wish to avoid this approach, since it removes any opportunity for the compiler to check affected code for aliasing and parallel usage.

We solve this problem, for channel types at least, by allowing the declaration of `SHARE`d channel type variables and subsequently enforcing their safe use. Either the client or server ends may be shared, or both, providing the full set of *one-to-one*, *any-to-one*, *one-to-any* and *any-to-any* channel arrangements (similar to those in JCSP [30]). For example, an *any-to-one* channel type pair can be created with:

```
SHARED BUF.MGR! s.cli:
BUF.MGR? u.svr:
SEQ
  s.cli, u.svr := MOBILE BUF.MGR
  ...  use 's.cli' and 'u.svr'
```

The '`SHARE`d' attribute changes the nature of the variable, such that it is only assignable or communicable with other shared channel-types of the same `CHAN TYPE` and endianism. This prevents the accidental mix-up of shared and nonshared channel-types, which would be disastrous.

Before the channels inside a *shared* channel-structure may be used, the whole channel-structure (or rather the relevant channel type *end* of it), must be `CLAIM`ed. This follows a style similar to that presented in occam3 [20], although what we are `CLAIM`ing is a little different.

For example, a new '`client`' which operates on client-shared channel-types might look like:

```
PROC client.2 (SHARED BUF.MGR! cv)

  MOBILE []BYTE b:

  CLAIM cv           -- claim it
    SEQ
      cv[req] ! 1518 -- send desired buffer size
      cv[buf] ? b    -- get buffer
      ...  use 'b'
      cv[ret] ! b    -- move buffer back
:
```

While this process is in the body of the `CLAIM`, other clients are blocked from using the channel-structure ('`cv`'). The same applies to shared *server-ends* too.

The usage rules for `CLAIM` differ slightly depending on whether the `CLAIM`ed channel-structure is a *server* or *client* end. Once it has claimed the shared *client* end of a channel type, a process may only communicate on the channels within a `CLAIM`ed structure and must not `CLAIM` anything else. Assignment, function-calls and timeouts are still permitted, as are `PROC` calls (on the condition that any channels used are part of the `CLAIM`ed structure).

The rules for the server `CLAIM` are slightly different. Once a process `CLAIM`s a shared *server* end it must not make any nested `CLAIM`s on other (shared) *server end*s. It may however `CLAIM` client-shared ends and act as a client to other servers. Usage of other channels within the body of a server `CLAIM` is unrestricted. The issue of cyclic deadlock exists here, when a loop of client-server relationships form, but this can be avoided by careful design [26].

In the case of long-running transactions, an *any-to-any* channel becomes less useful, since both the *client* and *server* must remain in the `CLAIM` for the duration of the transaction, preventing other clients and servers interacting. However, what we can do is create another *any-to-any* channel type, which only communicates the *one-to-one*

Note 2: However, there are dangers in allowing network topologies to be set up dynamically [28], particularly in regards to deadlock/livelock analysis. We must therefore take care to constrain the networks we can now construct dynamically to those we can analyse.

channel types for each client/server pair to use privately. For example:

```
CHAN TYPE C.BUF.MGR
  MOBILE RECORD
    CHAN BUF.MGR? svr?:
:
```

This allows an *any-to-any* channel carrying `BUF.MGR?`s to be created. When a client process wishes to engage in a transaction with a server process, it can dynamically create the `BUF.MGR` channels, communicate the server-end and use the client-end locally. The '`client`' process in this case could be written as:

```
PROC client.3 (SHARED C.BUF.MGR! c.cv)

  BUF.MGR! cv:
  BUF.MGR? sv:

  SEQ

    -- allocate channels
    cv, sv := MOBILE BUF.MGR

    -- claim connection to (any) server
    CLAIM c.cv
      c.cv[svr] ! sv  -- move out server end
    MOBILE []BYTE b:
    SEQ
      cv[req] ! 1518  -- send desired buffer size
      cv[buf] ? b     -- get buffer
      ...  use 'b'
      cv[ret] ! b     -- move buffer back
:
```

The corresponding server process simply inputs the server-end manufactured in the client then uses that for communication:

```
PROC server.3 (SHARED C.BUF.MGR? c.sv)
  BUF.MGR? sv:
  SEQ

    -- claim connection from (any) client
    CLAIM c.sv
      c.sv[svr] ? sv         -- move in server end

    MOBILE []BYTE b:
    INT s:
    SEQ
      sv[req] ? s            -- input size
      b := MOBILE [s]BYTE    -- allocate buffer
      sv[buf] ! b            -- move to client
      sv[ret] ? b            -- take back
:
```

The network connecting these together is:

```
SHARED C.BUF.MGR! cli.c:
SHARED C.BUF.MGR? svr.c:
SEQ

  -- allocate channels
  cli.c, svr.c := MOBILE C.BUF.MGR

  PAR i = 0 FOR 4    -- 4 clients and 4 servers
    PAR
      client.3 (cli.c)
      server.3 (svr.c)
```

This example creates a network in which four client processes and four server processes are plugged into a *both ends shared* channel-structure (of type '`C.BUF.MGR`'). In a *boom*, clients will be queuing to get on to the shared channel to find a server. In a *recession*, servers will queue to find a client. Clients and servers use *separate* queues.

In a more realistic example, each client and each server would be making many transactions – i.e. looping. Instead of having to allocate a new `BUF.MGR` bundle of channels each time, a client could reuse the same bundle. To do that, the server would have to return the `BUF.MGR` server-end it was using. For that to be possible, `BUF.MGR` would need to be extended with an extra channel on which it could transport *its own* server-end:

```
RECURSIVE CHAN TYPE BUF.MGR.2
  MOBILE RECORD
    CHAN INT req?:              -- integer
    CHAN MOBILE []BYTE buf!:    -- dynamic array
    CHAN MOBILE []BYTE ret?:    -- dynamic array
    CHAN BUF.MGR.2? finish!:    -- own server end
:
```

The channel type needs to be declared *recursive* because of the usual name scoping rules in occam – otherwise, the name of an item being declared does not become useable until after the declaration. To return the server-end down itself, `server.3` need only say:

```
sv[finish] ! sv
```

After which, of course, its variable `sv` is *undefined* (but available for re-assignment from the `c.sv[svr]` channel). To retrieve that server-end, `client.3` must execute:

```
cv[finish] ? sv
```

4.4 Non-dynamic non-mobile channel-types

In a less dynamic environment, such as an embedded system, real dynamic memory may be scarce or not available. Having the benefits of channel types is still desirable though. There are two solutions to this. The first is to restrict the size of the dynamic memory pool, making the '`MOBILE`' allocation operator a possible descheduling point (although this is currently unimplemented). The second is to make the type non-mobile, which causes it to be allocated statically in the local workspace of a process. The second option is examined here.

Making the channel type non-mobile means that we can no longer *move* it around, which rules out communication and assignment. Because of this, a single name can refer permanently to both ends of the channel. The type and variable declarations are:

```
CHAN TYPE FOO
  RECORD
    CHAN INT req ? :
    CHAN BYTE resp!:
:

FOO c:
```

Because no direction is specified on the type of '`c`', it must be added whenever '`c`' is subsequently referenced (just as for normal channels). Channel type parameters still carry the direction in the type since they can only ever refer to one end. For example:

```
PROC foo.svr (FOO? link)
  INT x:
  SEQ
    link[req] ? x
    ...
    out.string (.., 0, link[resp]!)
:
```

```
...  other process declarations

FOO c:
PAR
  foo.svr (c?)
  ...  rest of the network
```

When using a channel field directly for communication, no direction specifier is needed since the channel-direction is specified in the channel-structure (plus it doesn't fit into the language syntax for communication). The direction should still be specified when the channel is used as a parameter, however, as shown in the preceeding example.

5 The extended rendezvous

The extended rendezvous is a mechanism for allowing the inputting process of a communication to execute a process with the communicated data, while the outputting process remains suspended. A new *extended-input* process implements this. It is syntactically similar to an ordinary input, but with '`??`' instead of '`?`'. However, it is followed by a compulsory indented process (the *extended-rendezvous*), which is executed while the outputting process remains blocked. Anything may be done during extended rendezvous *except*, of course, trying to communicate with the blocked outputter.

One application is to tap a channel in a way that does not affect the synchronisation between the processes either side. This is useful when we wish to 'inspect' the data flowing round a process network. Channels connecting existing processes can be 'tapped' without changing the semantics of that process network (assuming that the processes monitoring the tapped output channels guarantee always to take it).

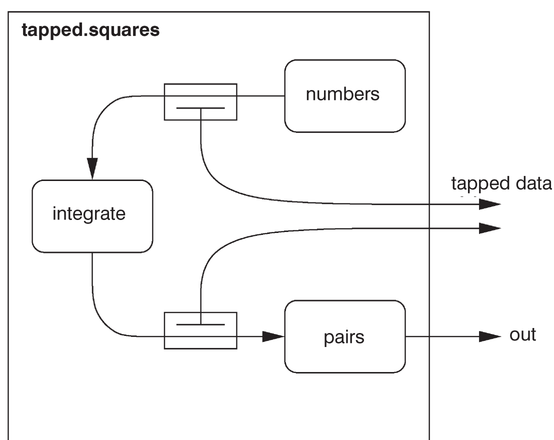

Fig. 7 *Tapped 'squares' process pipeline*

Fig. 7 shows how to inspect communication internal to the '`squares`' process pipeline of [31]. Here is the code for the '`tap`' process, assuming `INT` dataflow:

```
PROC tap (CHAN INT in?, out!, tap!)
  WHILE TRUE
    INT x:
    SEQ
      in ?? x
        out ! x
      tap ! x
:
```

5.1 Semantics

The semantics of the extended rendezvous are quite simple. Consider the following input and output processes running in parallel:

```
c ! 42          c ?? b
                  ...  extended rendezvous (no c?)
```

This has the same semantics as the following pair of processes:

```
SEQ             SEQ
  c ! 42          c ? b
  c.ack ? any     ...  extended rendezvous (no c?)
                  c.ack ! TRUE
```

where '`c.ack`' is an extra `CHAN BOOL` on which the processes synchronise. The implementation is quite different [32], but the semantics remain as presented here.

5.2 ALTs and CASE inputs

An extended rendezvous may be used as a guard in an `ALT`:

```
ALT
  c ?? x
    ...  extended rendezvous (no c?)
    ...  guarded process (optional)
  d ? y
    ...  guarded process
```

If the first guard is chosen, the outputting process is rescheduled after the extended rendezvous. The guarded process (at the same level of indentation as the extended rendezvous process) is then executed. Often there is nothing left to do after the rendezvous process. Instead of writing `SKIP`, the guarded process may *in this case* be omitted.

A similar construction is needed when using variant (`CASE`) protocols:

```
PROTOCOL CONTROL
  CASE
    data; INT
    stop
:

...

in ?? CASE
  INT x:
  data; x
    ...  extended rendezvous (no in?)
    ...  case process (optional)
  stop
    ...  extended rendezvous (no in?)
    ...  case process (optional)
```

In both tag cases, the second indented process is executed after the outputting process has resumed. This second process is always optional and if not present is assumed to be *Skip*.

5.3 Usage restrictions

There is only one usage restriction on extended-input: the channel undergoing the extended input may not be used in the *extended-process*. Any attempt to use the channel involved would immediately deadlock, since the process on the other end of the channel is suspended. This restriction is enforced by the compiler.

5.4 Efficiency and other uses

The extended rendezvous is implemented in such a way that it does not impact, in any way, the performance of ordinary (i.e. nonextended) communications. This is achieved by masquerading the process performing the extended-input as an ALTer, whether part of an ALT construct or not. This avoids any modifications to the implementation of channel output instructions due to the way ALTing processes are handled at run-time. A full description of the implementation for the extended rendezvous can be found in [32].

One particular, though not initially conceived, use of the extended rendezvous is in the correct handling of the *stop* error mode. By default, KRoC compiles occam programs in *halt* error mode i.e. when a process causes a run-time error, the whole (occam) system is killed (additionally reporting the location of the error [33]). In stop error mode, run-time errors made by an occam process simply cause that process to stop running, without affecting the operation of other occam processes; the error is thus contained. Of course, without specific application-level protection to guard against committed interaction with potentially stopped processes, run-time errors – even though localised – will eventually lead to deadlock.

There is a long-standing problem associated with stop error mode however, concerned with variant (CASE) protocol handling particularly those protocols containing data-less tags. The implementation of variant protocols is trivial: the tag names are enumerated by the compiler and communicated as BYTEs. The variant (or single-tag) input process inputs the BYTE, then performs a CASE selection on the tag. If the outputting process sends a tag value which the inputting process does not handle, a run-time error is generated by the inputting process (the implicit STOP for unhandled CASEs). If the output consists only of the tag, without any data, this is communicated and the outputting process continues – under the false impression that the communication has succeeded.

The extended rendezvous now provides a simple solution to this problem. If an inputting process does not handle all data-less variants in a CASE protocol, the initial BYTE input for the tag is performed as an extended input. The inputting process then performs the CASE selection on the inputted tag, only allowing the outputting process to continue after a selection has been successfully made. In the cases of unhandled tags, this will leave the output process blocked, while the inputting process generates a run-time error and, in stop error mode, simply stops. Thus, we now correctly implement, for example, the CSP equation:

$$((c!apple \parallel c?banana) \setminus \{c\}) = STOP$$

6 Dynamic process creation (the FORK)

The FORK is a way of launching a dynamic process which runs parallel to the dispatching process. The early ideas about FORK were to allow an arbitrary process to be spawned (that process being indented under the FORK). This was causing too many headaches in the implementation however, so a more restricted approach has been taken for now: the parallel creation of a PROC instance. This provides a nice way of giving the launched process an initial state: its parameters. The more general FORK would need to provide a way of handling scoping and parallel usage for free variables in the FORKed process. Controlling this through the parameters of a PROC is much simpler.

The lifetime of a FORKed PROC and its dispatching process are controlled through a special FORKING process constructor. This acts as a barrier which ensures any FORKed processes are complete before leaving the FORKING block. For example, here is part of much simplified code from a *dynamic* version of the occam web-server [24]:

```
PROC fe.proc (VAL INT n, D.CONN conn,
              SHARED C.CONN! to.sw)
  ...
:

PROC fe.farm (CHAN D.CONN in?,
              SHARED C.CONN! to.sw)
  D.CONN local:
  FORKING
    INITIAL INT c IS 0:
    WHILE TRUE
      SEQ
      in ? local
      FORK fe.proc (c, local, to.sw)
      c := c + 1
:
```

The 'fe.farm' process sits in a loop accepting D.CONN's (connection in the web-server) from its 'in' channel. For each D.CONN received, an instance of 'fe.process' is created. These processes are actually pooled for recycling, see [16] for details.

The need for the 'FORKING' block may not seem immediately obvious, here less so than in other cases, but the implementation requires it. A more obvious case is where we share data with FORKed processes using the '#PRAGMA SHARED name' compiler directive (to turn off usage and alias checking). In these cases we must guarantee (completely) that shared variables remain in scope for the whole lifetime of any FORKed process.

6.1 Semantics of FORK parameters

Unlike nonFORKed PROCs, whose parameters follow a *renaming* semantics, the parameters in a FORKed PROC have to follow a *channel communication* semantics. This has different semantic effects to an ordinary PROC call. We allow only the following types of parameters:

- VAL data-types: these are *copied* into the FORKed process, regardless of size. This differs from traditional VAL parameter passing, which will abbreviate (rename) items larger than four bytes (one word).
- MOBILE data-types and MOBILE channel type-ends: these are *moved* into the FORKed process – i.e. the FORKING process loses them. If the FORKING process does not want to lose them, it must pass a CLONEd argument. (For channel-types, only explicitly SHARED ends may be CLONEd.)
- Reference parameters: which have been explicitly shared (with a compiler #PRAGMA) and which are declared outside the FORKING block. This ensures that they remain in scope for the lifetime of a FORKed PROC. Variables declared within the FORKING block may not be passed by reference.

The copying of VAL data-types is *required*, as we wish to allow the code on the left (below):

```
FORKING                    INT x:
  INT x:                   SEQ
  SEQ                        x := 42
    x := 42                  PAR
    FORK P (x)                 P (x)   -- VAL param
    x := x + 1                 SEQ
    Q (x)                        x := x + 1
                                 Q (x)
```

The left column code (above) is not equivalent to the right column code (which is, of course, illegal since ‘x’ is both read and assigned in parallel). The FORKING block is in fact equivalent to the following process sub-network:

```
CHAN INT c:
PAR
  INT x:
  SEQ
    x := 42
    c ! x
    x := x + 1
    Q (x)

  INT x:
  SEQ
    c ? x
    P (x)
```

Semantically, there is no difference between parameter passing VALs by communication (as we have done here), and parameter passing using INITIAL formals. INITIAL formal parameters [20, 34] should be used, but these are not currently supported by the compiler.

Some more interesting FORKs (i.e. with a loop), can also be expressed in our extended occam. For example, the following two (columns of) processes are equivalent:

```
FORKING                    RECURSIVE PROC dispatch
  WHILE TRUE                            (CHAN BOOL c?)
    SEQ                      SEQ
      P ()                     BOOL any:
      FORK foo (4, 99)         c ? any
      Q ()                     PAR
                                 foo (4, 99)
                                 dispatch (c?)
                           :

                           CHAN BOOL c:
                           PAR
                             dispatch (c?)

                             WHILE TRUE
                               SEQ
                                 P ()
                                 c ! TRUE
                                 Q ()
```

6.2 Dynamic process farms

One application of FORK is for the dynamic creation and control of process *farms*. Fig. 8 shows the process network for a worker-farm, with a pool.manager to control the number of FORKed processes running.

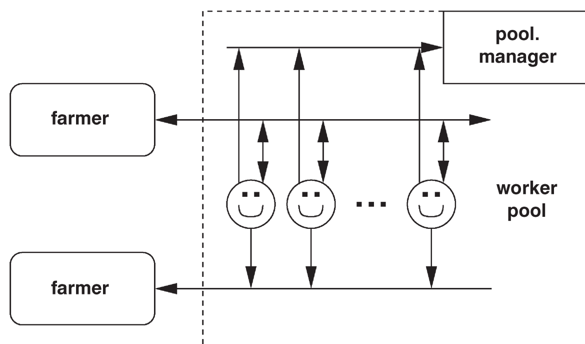

Fig. 8 *Forked worker-farm process network*

The ‘farmer’ generates work packets (maybe by receiving them from an external source, not shown) and distributes them to a pool of ‘worker’s. The system arranges for a minimum (‘min.idle’) number of ‘worker’ processes to always be available for processing new jobs. New ‘worker’s are started by the ‘pool.manager’ process, which maintains a count of the number of idle processes, FORKing more at the start of the loop if needed (which will always be the case the first time round, providing that ‘min.idle’ is greater than zero). In this code, the number of worker processes will only ever increase (to suit demand).

The channel type based code which implements these processes is as follows:

```
CHAN TYPE WORK.IN       -- server view (farmer)
  MOBILE RECORD
    CHAN BOOL request?:
    CHAN MOBILE []BYTE work.packet!:
:

CHAN TYPE WORK.OUT    -- server view (harvester)
  MOBILE RECORD
    CHAN MOBILE []BYTE result?:
:

CHAN TYPE SIGNAL   -- server view (pool.manager)
  MOBILE RECORD
    CHAN INT idle.count?:  -- busy (-1) or idle (+1)
:

PROC worker (SHARED WORK.IN! in,
             SHARED WORK.OUT! out,
             SHARED SIGNAL! signal)
  WHILE TRUE
    MOBILE []BYTE job:
    SEQ
      CLAIM in
        SEQ
          in[request] ! TRUE
          in[work.packet] ? job

      -- tell manager we're working
      CLAIM signal
        signal[idle.count] ! -1

      ...  do work on 'job' (communicates on 'out')

      -- tell manager we're done
      CLAIM signal
        signal[idle.count] ! +1
:

PROC harvester (WORK.OUT? from.workers)
  WHILE TRUE
    MOBILE []BYTE result:
    SEQ
      from.workers[result] ? result
      ...  consume result
:

PROC farmer (WORK.IN? to.workers)
  WHILE TRUE
    MOBILE []BYTE work:
    SEQ
      ...  manufacture work
      BOOL any:
      to.workers[request] ? any
      to.workers[work.packet] ! work
:
```

```
PROC pool.manager (VAL INT min.idle,
    SHARED WORK.IN! work.to.workers,
    SHARED WORK.OUT! work.from.workers)

  SHARED SIGNAL! signal.cli:
  SIGNAL? signal.svr:
  SEQ
    signal.cli, signal.svr := MOBILE SIGNAL

    FORKING
      INITIAL INT n.idle IS 0:
      WHILE TRUE
        SEQ
          IF
            n.idle < min.idle
              SEQ
                SEQ i = 0 FOR min.idle - n.idle
                  FORK worker (work.to.workers,
                               work.from.workers,
                               signal.cli)
                n.idle := min.idle
            TRUE
              SKIP

          INT n:
          SEQ
            signal.svr[idle.count] ? n
            -- n is busy (-1) or idle (+1)
            n.idle := n.idle + n
:
```

The code which sets this network up is as follows:

```
VAL INT min.idle IS ...:
SHARED WORK.IN! i.cli:
WORK.IN? i.svr:
SHARED WORK.OUT! o.cli:
WORK.OUT? o.svr:

SEQ
  i.cli, i.svr := MOBILE WORK.IN
  o.cli, o.svr := MOBILE WORK.OUT

  PAR
    farmer (i.svr)
    pool.manager (min.idle, i.cli, o.cli)
    harvester (o.svr)
```

Adding functionality to shut-down worker processes and to limit the number idle to some maximum is trivial and is left as an exercise for the reader.

Note that the `MOBILE BYTE[]` arrays are communicated efficiently *by reference* and that no aliasing dangers (e.g. through *parallel reference*) are possible. No memory leaks occur as the space for such arrays is automatically recycled when the variables go out of scope or are overwritten.

7 Extended rendezvous and channel-types

Fig. 9 shows a multiple client–server network that uses a shared *any-to-any* channel to enable a client and server to find each other. Here is example network code for this:

```
CHAN TYPE APP.LINK   -- client/server channel type
  MOBILE RECORD
    CHAN INT next.event?:
    CHAN MOBILE []BYTE event.data!:
:
```

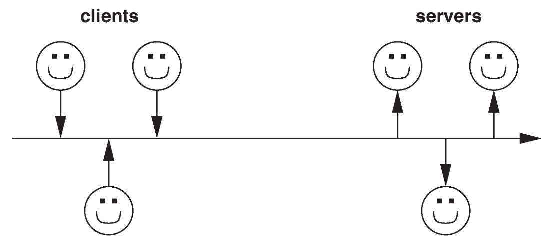

Fig. 9 *Multiple client–server network*

```
...  client and server PROC definitions

-- shared (at both ends) channel declaration
SHARED CHAN APP.LINK? link:

PAR
  PAR i = 0 FOR num.clients
    client (link!)      -- start a client
  PAR i = 0 FOR num.servers
    server (link?)      -- start a server
```

A client seeking a server makes a shared mobile channel-structure (`APP.LINK`) and outputs the *server-end* of this (of type '`APP.LINK?`') towards the set of servers hopefully waiting on the shared channel:

```
PROC client (SHARED CHAN APP.LINK? out!)
  WHILE TRUE
    APP.LINK? l.svr:        -- server-end
    APP.LINK! l.cli:        -- client-end
    SEQ
      -- create one-to-one mobile
      -- channel-structure
      l.cli, l.svr := MOBILE APP.LINK

      CLAIM out
        -- communicate server-end (and lose it)
        out ! l.svr

      ...  use 'l.cli' to communicate with server
:
```

Here is an outline for one of the servers:

```
PROC server (SHARED CHAN APP.LINK? in?)
  WHILE TRUE
    APP.LINK? svr:          -- server-end
    SEQ
      CLAIM in
        in ? svr  -- get server-end from a client
      ...  use 'svr' to communicate with client
:
```

Fig. 10 shows the network after a client and server have communicated, now connected (directly) by a *private* one-to-one channel-structure of type `APP.LINK`.

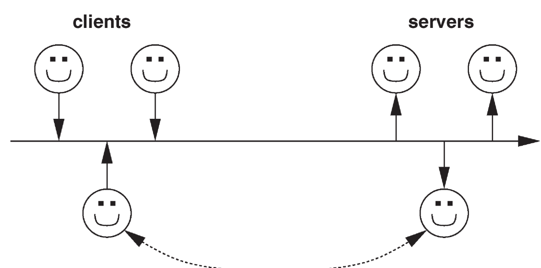

Fig. 10 *Multiple client-server network with connected client and server*

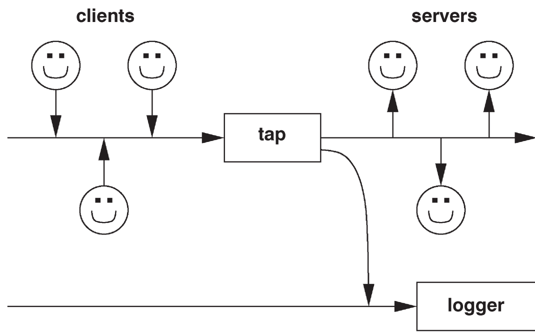

Fig. 11 *Multiple client–server network with 'tap' process*

Using the extended rendezvous with channel-types opens up some interesting possibilities. Fig. 11 shows a multiple client–server network with a tap process. Clients and servers still see shared channel-ends plugged into them, carrying the same *server-end channel structures* as before. This version of '`tap`' is special in that it intercepts and keeps the channel-end being passed, creates a new channel structure (of the appropriate type) and communicates the new server end to the original destination. This code uses *any-to-one* and *one-to-any* channels. Since the tap process in this example does not interfere with the synchronisation between the clients and servers, they (clients and servers) can only see the link as an *any-to-any* channel: they cannot detect the tap! Note that *no change* has been made to the client and server processes.

The '`tap`' process here (for the '`APP.LINK`' channel type) is:

```
PROC tap (CHAN APP.LINK? in?, out!,
          SHARED LOG! to.log)
  FORKING
    WHILE TRUE
      APP.LINK? c.svr, l.svr:
      APP.LINK! l.cli:
      SEQ
        l.cli, l.svr := MOBILE APP.LINK
        in ?? c.svr
          out ! l.svr
        FORK link.tap (c.svr, l.cli, to.log)
:
```

The '`tap`' process `FORK`s '`link.tap`' each time a client communicates a server-end to one of the servers. Note the use of the extended rendezvous to prevent the client being aware that its output line is being tapped. Fig. 12 shows the network after a client has communicated with a server.

The `FORK`ed '`link.tap`' process connects the two processes, and can be implemented so that its presence is also undetectable to the client and server processes connected either side. Fig. 12 also shows a '`logger`' process, to which the `FORK`ed '`link.tap`' processes report. A simple form of the '`link.tap`' process could be:

```
PROC link.tap (APP.LINK? from.cli, APP.LINK! to.svr,
               SHARED LOG! to.log)
  PAR
    WHILE TRUE
      INT e:
      from.cli[next.event] ?? e
        to.svr[next.event] ! e
        CLAIM to.log
          ...  report event on 'to.log'
    WHILE TRUE
      MOBILE []BYTE b:
      to.svr[event.data] ?? b
        from.cli[event.data] ! b
        CLAIM to.log
          ...  report event on 'to.log'
:
```

`PAR` is used here to handle both channels in the channel type independently. This is also non-terminating, which in a real-life situation is probably undesirable. For real-life protocols, the point at which the client and server processes either side 'let go' of the channels should be deducible from the data communicated. Sometimes the usage pattern may be that the channels only ever get used once, in which case the '`WHILE TRUE`'s can be reduced to '`SEQ`'. `ALT`ing implementations are also perfectly valid and probably desirable when we wish to arrange termination by inspection of the data.

The '`link.tap`' need not be so simple however. It might be the case that the clients and servers reside on different machines, with functionally dummy '`server`' and '`client`' processes at either end, incorporating the '`tap`' and necessary network infrastructure. In this case, communication of the channel-end would result in a network-aware process being created on either side to handle communication. To create the remote network-handling channel (and possibly the whole remote 'server' as well), some form of networking infrastructure needs to be available. As long as the network-handling processes synchronise properly over the network, the 'client' and 'server' at either side will see the link as synchronous and will be unaware of the networking. Fig. 13 shows what such a network might look like.

Since the extended rendezvous can be used to intercept channels, without requiring modifications in the (originally) connected processes, this provides a simple method for distributing existing occam programs amongst nodes on a network. The only modifications required would be in the code which sets up the process network, which could be reduced to just a single '`#USE`' compiler directive. The `USE`d code would implement the network-aware versions of existing processes, de-scoping the original local versions. This works equally well for code with and without channel-types.

Building the infrastructure to support such a distributed system is not the direct concern of this work, which merely provides a new way of doing it, hopefully much simpler, more secure and more efficient than was previously possible. Vella [35] provides a lot of insight into building such systems. The work there was done on the Sparc version of KRoC, in the assembler kernel. For the Linux/i386 version of KRoC, we can use the occam socket library [36] to implement the networking, as has been done by Goodacre [37] in a student project and by Schweigler [38] in his MSc thesis. A similar functionality

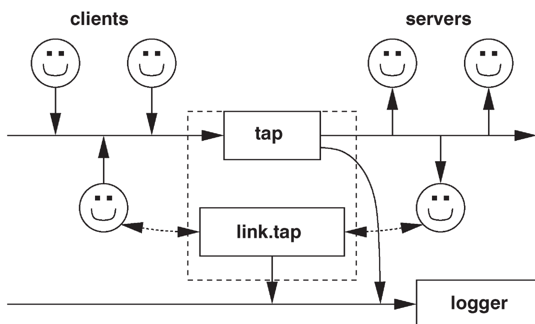

Fig. 12 *Multiple client–server network after forking 'link.tap'*

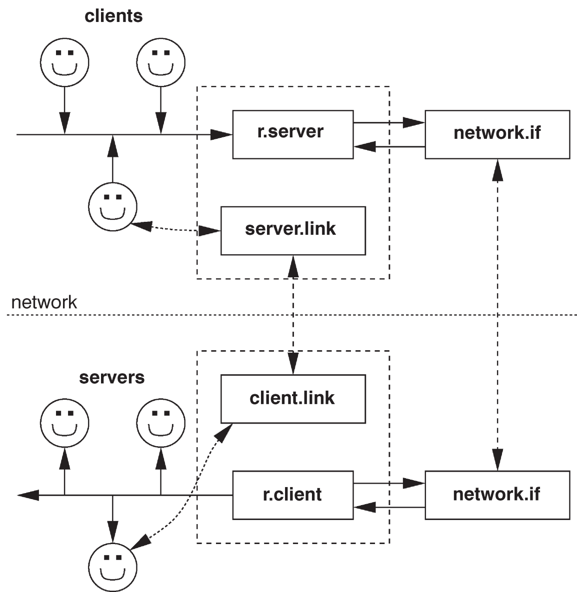

Fig. 13 *Remotely connected client–server network after communication and creation of link processes*

also exists in JCSP [30, 39, 40], which additionally allows the migration of processes (Section 9).

8 Process priority

A major requirement of real-time control applications is a set of cyclic processes, one for each *control-law*, managed so that each process completes each cycle within a fixed time. The rate at which each process cycles will be constant, but will generally be different for different processes.

Transputer hardware [6, 41] supported two levels of priority, low and high, with fast pre-emptive scheduling. The original KRoC [13] only supported a single level of priority, quietly implementing '`PRI PAR`' as just '`PAR`'. Without additional programming (at the application level) to construct more priorities [42, 43], this is not sufficient to manage securely more than one such 'control-law' per occam program, even at very low processor loadings. Efficient classical solutions (e.g. *rate-monotonic* or *deadline* scheduling [44]) require multiple and time-varying priorities. KRoC now provides 32 levels of priority.

Even so, for KRoC running on top of Linux or Solaris, we are at the mercy of the host operating-system and the way in which it performs scheduling between OS processes. There are ways of forcing *run-to-completion* behaviour for OS processes (*FIFO-scheduling*), but at the expense of other OS processes (including interrupt handlers) and the requirement for superuser privileges. We are investigating *Raw Metal occam* operating environments (RMoX) in which occam systems run without any OS support and overheads, and for which all scheduling is under the total control of the KRoC kernel.

Standard CSP [3] does not include a treatment of process priority, and omits `PRI ALT`, which has always existed in occam. Even the denotational semantics for occam [45, 46] expressly omit any treatment of priority. *CSPP* [47, 48] addresses this deficiency by providing a well-defined semantics for `PRI PAR` and occam priority issues in general.

8.1 API for priorities

Rather than implement priority in terms of `PRI PAR`, we have gone for a more general – but lower level – approach. The current implementation supports 32 distinct levels of priority, 0 being the highest and 31 being the lowest. The number of priority levels is limited to be efficient in the implementation, but is extensible with only a marginal increase in overheads.

To inspect or change its own priority, a process may use the following compiler built-ins:

```
INT FUNCTION GETPRI ()
PROC SETPRI (VAL INT p)
PROC INCPRI ()
PROC DECPRI ()
```

The use of '`SETPRI`'/'`INCPRI`'/'`DECPRI`' within a `FUNCTION` body is not allowed, to prevent nondeterminism in the resulting priority when evaluating expressions. '`GETPRI`' is wholly nonside-effecting, so can be used safely in expressions. The run-time implementation quietly ignores out-of-range values, mapping them to the lowest and highest priorities as appropriate. '`INCPRI`' is really a shorthand way of writing:

```
SETPRI (GETPRI () - 1)
```

and similarly for '`DECPRI`', which is

```
SETPRI (GETPRI () + 1)
```

A process may change its own priority arbitrarily but it cannot change the priority of any other process. Changing from a low to a high priority (decreasing priority level '`p`') will generally always succeed immediately, although there is a small possibility of descheduling. A change from a high to a low priority (increasing priority level '`p`') will result in a reschedule if another process is waiting at the target priority level or higher.

Even if priority is not used directly, its provision causes a slight increase in run-time overheads. Most of all, this is due to the need to save and restore a process's priority when rescheduling. Table 1 shows the results for the 'commstime' benchmark, whose process network is shown in Fig. 14.

Table 1: Results for 'commstime' benchmark on 800 MHz Pentium-3, values in nanoseconds

Translator	Channel cost (INT comm.)	Process start/stop cost
tranpc (old KRoC)	233	196
tranx86	112	52
tranx86 (inlining)	52	28
tranx86 (priority)	120	108
tranx86 (pri + inline)	77	79
tranx86 (pri + −P)	119	116
tranx86 (pri + inline + −P)	75	67

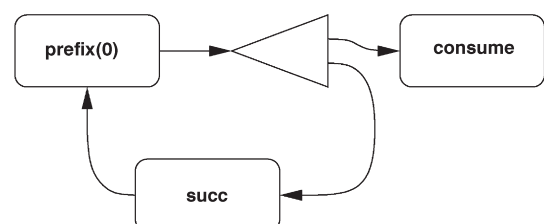

Fig. 14 *Process network for 'commstime' benchmark*

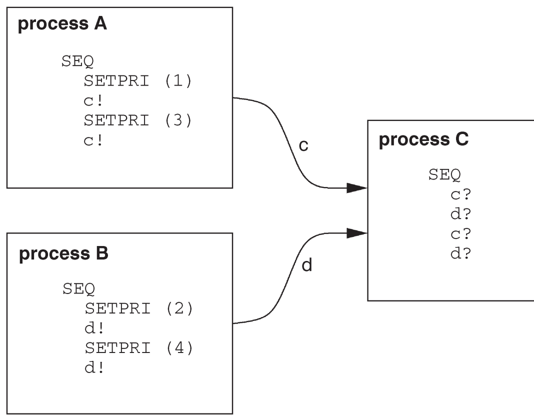

Fig. 15 *Priority benchmark process network showing loop body code*

8.2 Benchmarking priority handling

Fig. 15 shows the process network and code for a simple priority benchmark program. The benchmark is comprised of three processes, two interleaving producers and a consumer. The consumer is run at the lowest priority level (31) while the producers alternate between priority levels 1 through 4. The arrangement of priority and communication in this program forces the scheduling to happen in a deterministic way, although *in general* one cannot use priority to guarantee determinism.

Both the `A` and `B` processes sit in loops changing priority and communicating with the `C` process. At the point where the loops start, `A` and `B` are blocked in channels '`c`' and '`d`' at priorities 1 and 2, respectively. Process `C` at priority level 31 is the only runnable process, which starts by communicating on '`c`', thereby waking up `A`, which gets rescheduled immediately because it is of a higher priority.

Fig. 16 shows the execution trace of the benchmark, indicating the timed (and looping) section. The priority overheads are calculated by subtracting the time required for a priority-free version of the loops from the prioritised version. There are a total of eight context switches here, four for rescheduling when a process blocks in a channel (`A` and `B` processes), and four for rescheduling a higher-priority process with which process `C` communicates. There are also other overheads associated with changing priority, since the current run-queue must be saved and a new one loaded. The nonprioritised version of the loops use an average of eight context switches, the exact number depends on the scheduling order of processes. It is also sensitive to the policy of rescheduling blocked processes, i.e. whether we continue running, it continues running, or neither continue running.

The priority overhead for this benchmark loop is 752 ns, measured on an 800 MHz Pentium 3. Interestingly, with the '`-p`' flag (check sync flags and timer on backward jumps), the overhead is reduced to 728 ns, again attributed to cache effects. Overall, the run-queue is changed 12 times in the prioritised version, giving an average overhead of 63 ns for a priority-level change (around 50 machine cycles).

9 Mobile process types

Mobile processes exist in many other technologies (such as *applets*, *agents* and distributed operating systems). occam offers the prospect of supporting this with much higher security and lower overheads.

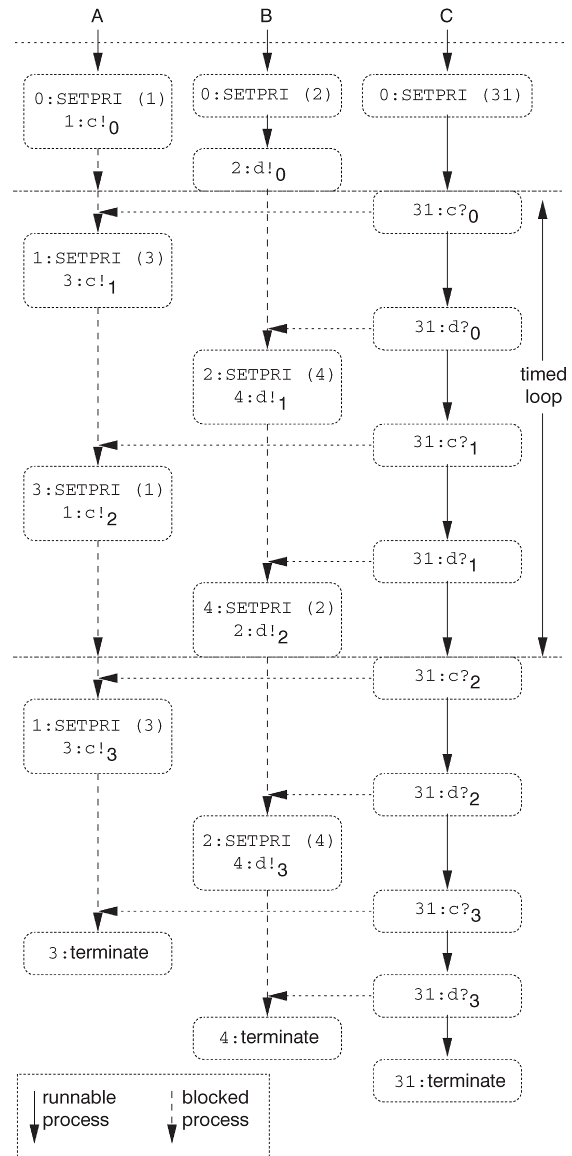

Fig. 16 *Execution trace for priority benchmark program*

This is an initial proposal for mobile processes in occam – at the time of writing, nothing in this Section has been fully implemented. They will have application both for internal concurrency and for distributed systems.

Mobile process types for occam follow naturally from the other mobile mechanisms for data [18, 19] and channels (Section 4). Mobilising processes presents new challenges since processes are active components, whereas data and channels are passive.

The method for mobile processes presented here introduces the concept of a process type to occam. This is effectively a type signature for a `PROC` – for example:

```
PROC TYPE IO.KILL (CHAN INT in?, out!, kill?):
```

declares a process type called '`IO.KILL`', that defines an interface with three channels, two for input and one for output, all carrying simple `INT`s.

Process types such as this are used in two places in our proposal for occam. First within the definition of a *mobile* process (Section 9.1), where an implementation is specified for a particular process type. Secondly, for declaring mobile process *variables* (Section 9.2) which, after initialisation, can be activated using an interface defined by one of the process types it implements.

9.1 Mobile processes

Mobile processes are active entities that combine some data (their state) with code to initialise and operate on that data. The state of a mobile process consists of standard occam declarations.

State initialisation is defined through PROC-style constructors with arbitrary signatures. The code for mobile process activation is also implemented in a PROC-style manner, using named process types.

The following code shows an example mobile process declaration, for a *serial running-sum integrator*, that offers an interface of type 'IO.KILL' (above):

```
MOBILE PROC integrate.kill

  INT total:  -- private data

  CONSTRUCT ( )
    total := 0

  CONSTRUCT (VAL INT i)
    total := i

  IMPLEMENTS IO.KILL (CHAN INT in?, out!,
                      CHAN INT kill?)

    INITIAL BOOL running IS TRUE:
    WHILE running

      PRI ALT

        INT any:
        kill ? any
          running := FALSE

        INT v:
        in ? v
          SEQ
            total := total + v
            out ! total
:
```

This declares a mobile process called 'integrate.kill', whose state consists of the single variable 'total'. Two state initialisation ('CONSTRUCT') processes are provided, one which takes no arguments (initialising the total to zero), and one which takes a single INT argument which is assigned to the total. The main body of code is introduced with 'IMPLEMENTS IO.KILL (...)', which provides the three-channel 'IO.KILL' interface. A mobile process may implement multiple interfaces, but they must all be different.

9.2 Process variables

Mobile process variables provide the basic mechanism for using process types, allowing a *mobile process* to be created, then subsequently used (activated), assigned or communicated (but only when not active). These operations are based on the variable's process type, they are independent of the particular mobile process ('MOBILE PROC') providing the implementation.

The initialisation of mobile process variables, through the name of a mobile-process, follows a similar style to other dynamic mobile initialisers (such as for mobile channel-types). For example, a simple declaration and initialisation using the earlier 'IO.KILL' type and 'integrate.kill' mobile process, is:

```
IO.KILL x:
SEQ
  x := MOBILE integrate.kill ()
  ...  process using 'x'
```

Initially, the variable 'x' is undefined and treated by the compiler as such. The initialisation is performed via an assignment, which dynamically allocates an instance of 'integrate.kill', initialised using its zero-argument CONSTRUCT block. The compiler checks that 'integrate.kill' does indeed implement 'IO.KILL'.

Once initialised (defined), a mobile process variable may either be communicated or assigned, or activated using the interface specified by the type. The syntax for activating a process variable is the same as that used for instancing standard PROCs—except the name referred to is a mobile process variable and not a PROC definition.

9.3 Communication of mobile processes

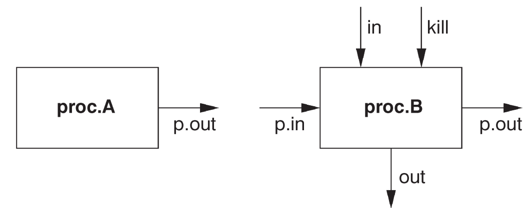

Fig. 17 *Mobile process communicating processes*

Fig. 17 shows two processes that communicate mobile processes (of type 'IO.KILL'). The following code shows their implementations:

```
PROC proc.A (CHAN IO.KILL p.out!)
  IO.KILL x:
  SEQ
    -- 'x' is not yet defined
    x := MOBILE integrate.kill (42)
    -- 'x' is now defined
    p.out ! x
    -- 'x' is now undefined
:

PROC proc.B (CHAN IO.KILL p.in?, p.out!,
             CHAN INT in?, out!, kill?)
  IO.KILL y:
  SEQ
    -- 'y' is not yet defined
    p.in ? y
    -- 'y' is now defined
    y (in?, out!, kill?)      -- activation
    -- 'y' is still defined
    p.out ! y
    -- 'y' is now undefined
:
```

Fig. 18 shows part of a mobile process network. It contains communicating instances of 'proc.A' and 'proc.B'. An 'integrate.kill' mobile process is constructed in 'proc.A' and passed on to 'proc.B', where it is plugged into locally provided channels and activated. When that

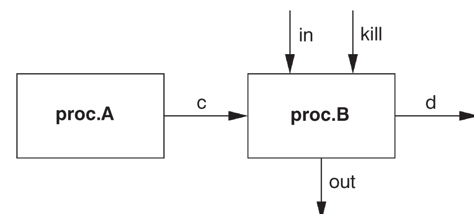

Fig. 18 *Mobile process network*

activation terminates, it moves on again—taking its state (its running 'total') with it.

Although this example is simple, the process communication mechanism it demonstrates is powerful. A mobile process can be communicated and activated without limitation, accumulating and disseminating information at each port of call. Here is the code for Fig. 18:

```
CHAN IO.KILL c, d:
CHAN INT in, out, kill:
...  other channels
PAR
  proc.A (c!)
  proc.B (c?, d!, in?, out!, kill?)
  ...  other processes
```

Note that whilst a mobile process is active (e.g. 'y' in 'proc.B'), it cannot be assigned or communicated. This is enforced by occam's normal usage/aliasing rules.

Unlike dynamic process mechanisms in other languages (the 'CSProcess' within JCSP for example), occam's mobile processes do not have a reference to themselves (Java's 'this' reference in JCSP). This prevents a process from being able to communicate itself down a channel which, although an interesting idea, we propose to ban. Parallel usage and aliasing checks will forbid any such attempts (that could be made by passing a process-variable as a parameter to its own instance).

9.4 Assignment of mobile processes

Assignment of mobile processes follows naturally from communication, such that the process moves from the source variable to the destination variable. For example:

```
IO.KILL x, y:
SEQ
  x := MOBILE integrate.kill ( )
  y := x
  y (in?, out!, kill?)
```

After the assignment 'y := x', x is undefined and any attempt to use it will result in a compiler error. In less obvious circumstances, the compiler generates run-time checks to ensure that 'x' is defined before its value is assigned to 'y', those checks having a negligible overhead.

In a similar way to mobile data-types [19], the 'CLONE' operator may be used to generate a copy of the mobile, leaving its operand (the mobile process being CLONEd) defined. For example:

```
IO.KILL x, y:
SEQ
  x := MOBILE integrate.kill ()
  y := CLONE x

  CHAN INT d:
  PAR
     x (in?, d!, kill.a?)
    y (d?, out!, kill.b?)
```

This code implements a double integrator. The implementation of CLONE for 'integrate.kill' is trivial: it involves allocating workspace (a very small unit-time cost) for the new process and copying only the *state* (in this case, the single 'INT total') from the old process.

For processes with a more complicated state, such as one containing other mobile variables, the CLONE must perform a *deep-copy* to prevent aliasing, handled by CLONEing each mobile variable in the state. The one case for which this is invalid is when an unshared channel-bundle end (Section 4) is part of the state. To avoid the problem, we also propose to ban the use of unshared mobile channel bundles in the state of a mobile process; this can be easily checked by the compiler. Shared channel bundles are perfectly acceptable however.

10 Conclusions and future work

At the time of writing, the extensions presented here (except Section 9) are almost ready for release. There are one or two things which are currently unhandled by the compiler, but these will be finished off in version 1.3.3 of KRoC/Linux [15]. A new experimental occam webserver (which uses FORKs and channel-types) is currently running live at [24] and successfully serving pages.

It is also hoped that improvement in the translator's handling of priority will yield better benchmark times for 'commstime' (part of the KRoC release), results for which are shown in Table 1. This is especially true for inlining, where the presence of priority often disables inline translations for some instructions.

It has been pointed out that allowing run-time failures (with OS-grade loadable occam processes [49]) presents the possibility of dynamic memory being 'lost' (inside the terminated process or processes). Although it is not currently implemented, it is possible to recover the memory. The solution is also required for correct loading/saving of these loadable occam processes, although the current (somewhat naïve) implementation of this works. The required addition is in theory quite simple: generate in-line tables of workspace offsets for channels and dynamic-pointers. This will allow the run-time system to free dynamic memory used by an evicted process. It is hoped to implement this in the near future. Without it, saving and restoring loadable processes has the potential to go wrong.

Another aspect of dynamic occam proposed here are the mobile process (or agents) extensions (Section 9). The question of run-time failure also needs to be addressed for these mobile processes.

However, we have positive answers to general security issues related to incoming mobile processes especially those downloaded from networked channels. Access to resources in occam is always abstracted through explicit plumbing, such as a channel bundle. If we don't want a downloaded mobile to access the file-system (for example), we simply don't plug the necessary channels into its activation, and it can't find them itself.

Finally, as mentioned at the start of Section 8, we are investigating *Raw Metal occam* operating environments (RMoX) in which occam systems run without any OS support and overheads, and for which all scheduling is under the total control of the KRoC kernel. We are currently using the Flux OSKit [50] to provide the boot mechanism and access to a flat (physical) memory space, beyond that very little else of the OSKit is used. This work-in-progress, is being done in collaboration with Brian Vinter from the University of Southern Denmark, and aided by several final-year and MSc students at UKC. This currently utilises many of the extensions presented here, with the exception of the mobile processes described in Section 9.

11 Acknowledgments

The authors would like to thank EPSRC for funding this work in the form of a research studentship, and the anonymous reviewers who provided useful and detailed feedback on an earlier revision of this work. Also many

thanks to the people who have been patient with the various new features of KRoC/Linux, submitting valuable bug reports and providing useful thoughts, in particular: David Wood, Christian Jacobsen, Mario Schweigler, Adam Sampson, Adrian Lawrence and Hiroshi Nakahara.

The proposal for mobile processes strongly follows ideas first spelt out by Tom Locke [28].

12 References

1 INMOS LIMITED: 'Occam 2.1 reference manual'. Tech. rep., May 1995. Available at: http://www.wotug.org/occam

2 HOARE, C.A.R.: 'Communicating sequential processes', *Commun. ACM 21*, 1978, **8**, pp. 666–677

3 HOARE, C.A.R.: 'Communicating sequential processes' (Prentice-Hall, London, 1985)

4 ROSCOE, A.W.: 'The theory and practice of concurrency' (Prentice Hall, London, 1997)

5 HOMEWOOD, M., MAY, D., SHEPHERD, D., and SHEPHERD, R.: 'The IMS T800 transputer', *IEEE Micro*, 1987, pp. 10–26

6 MAY, M.D., THOMPSON, P.W., and WELCH, P.H.: 'Networks, routers and transputers' (IOS Press, Amsterdam, 1993)

7 INMOS LIMITED: 'The T9000 transputer hardware reference manual'. SGS-Thomson Microelectronics, 1993

8 JOY, B., GOSLING, J., and STEELE, G.: 'The Java language specification' (Addison-Wesley, Reading, MA, 1996)

9 JONES, R., and LINS, R.: 'Garbage collection: Algorithms for automatic dynamic memory management' (Wiley, New York, 1996, reprint 1997)

10 KERNIGHAN, B.W., and RITCHIE, D.M.: 'The C progamming language', (Prentice Hall, Englewood Cliffs, 1978, 1st edn.)

11 AMERICAN NATIONAL STANDARDS INSTITUTE: 'Programming languages—C, 1999'. 1999, ISO/IEC 9899:1999

12 STROUSTRUP, B.: 'The C++ programming language' (Addison-Wesley, Reading, MA, 1997, edn.)

13 WELCH, P.H., and WOOD, D.C.: 'The Kent retargetable occam compiler'. Proceedings of WoTUG 19 Parallel processing developments, in O'NEILL, B. (Ed.): 'Concurrent systems engineering' (IOS Press), Amsterdam, March 1996, Vol. 47, pp. 143–166

14 OFA PROJECT: 'Occam For All: Case for support, Feb.1995, Available online at: http://wotug.ukc.ac.uk/parallel/occam/projects/occam-for-all/ accessed December 2002

15 WELCH, P.H., MOORES, J., BARNES, F.R.M., and WOOD, D.C.: 'The KRoC Home Page'. 2000, Available at: http://www.cs.ukc.ac.uk/projects/ofa/kroc/ accessed December 2002

16 BARNES, F.R.M., and WELCH, P.H.: 'Prioritised dynamic communicating processes: Part II'. Proceedings of WoTUG 25 Communicating process architectures, in PASCOE, J., WELCH, P., LOADER, R., and SUNDERAM, V. (Eds.): 'Concurrent systems engineering' (IOS Press), Amsterdam, September 2002, pp. 353–370

17 WELCH, P.H., and WOOD, D.C.: 'Higher levels of process synchronisation'. Proceedings of WoTUG 20 Parallel programming and Java, in BAKKERS, A. (Ed.): 'Concurrent systems engineering' (IOS Press), Amsterdam, The Netherlands, April 1997, pp. 104–129

18 BARNES, F.R.M., and WELCH, P.H.: 'Mobile data types for communicating processes'. Proceedings of the 2001 international conference on Parallel and distributed processing techniques and applications (PDPTA'2001), (CSREA press), June 2001, Vol. 1, pp. 20–26

19 BARNES, F.R.M., and WELCH, P.H.: 'Mobile data, dynamic allocation and zero aliasing: an occam experiment'. Proceedings of WoTUG 24, Communicating process architectures, in CHALMERS, A., MIRMEHDI, M., and MULLER, H. (Eds.): 'Concurrent systems engineering' (IOS Press), Amsterdam, September 2001, Vol. 59, pp. 243–264

20 BARRETT, G.: 'Occam 3 reference manual'. Tech. Rpt., Inmos Limited, Mar. 1992, (http://wotug.ukc.ac.uk/parallel/occam/documentation) accessed December 2002

21 MULLER, H.L., and MAY, D.: 'A simple protocol to communicate channels over channels'. EUROPAR '98 Parallel Processing, LNCS 1470, September1998, Southampton, UK, Springer Verlag, pp. 591–600

22 MAY, D., and MULLER, H.: 'Copying, moving and borrowing semantics'. Proceedings of WoTUG 14 Communicating process architectures, in CHALMERS, A., MIRMEHDI, M., and MULLER, H. (Eds.): 'Concurrent systems engineering' (IOS Press), Amsterdam, September 2001, Vol. 59, pp. 15–26

23 BARNES, F.R.M.: 'Various extensions to the occam compiler'. 2001, (http://www.cs.ukc.ac.uk/projects/ofa/kroc/occ21-extensions.html) accessed December 2002

24 BARNES, F.R.M.: 'The occam Web-Server Home Page'. 2000, (http://wotug.ukc.ac.uk/ocweb/.) accessed December 2002

25 PLOEG, E., SUNTER, J.P.E., BAKKERS, A.W.P., and ROEBBERS, H.W.: 'Dedicated multipriority scheduling'. Proceedings of WoTUG-17: Progress in transputer and occam research, in MILES, R., and CHALMERS, A. (Eds.): 'Transputer and occam engineering' (IOS Press), The Netherlands, April 1994, Vol. 38, pp. 18–31

26 WELCH, P.H., JUSTO, G.R.R., and WILLCOCK, C.J.: 'Higher-level paradigms for deadlock-free high-performance systems'. Transputer applications and systems'93 in GREBE, R., HEKTOR, J., HILTON, S., JANE, M., and WELCH, P. (Eds.): 'Proceedings of the 1993 World Transputer Congress' (IOS Press, Netherlands), Aachen, Germany, September 1993, Vol. 2, pp. 981–1004, (http://www.cs.ukc.ac.uk/pubs/1993/279)

27 MAY, D., and MULLER, H.L.: 'Using channels for multimedia communication'. Tech. Rpt., Department of Computer Science, University of Bristol, Feb. 1998

28 LOCKE, T.S.: 'Towards a viable alternative to OO – extending the occam/CSP programming model'. Proceedings of WoTUG 24 Communicating process architectures, in CHALMERS, A., MIRMEHDI, M., and MULLER, H. (Eds.): 'Concurrent systems engineering' (IOS Press), Amsterdam, Sept. 2001, Vol. 59, pp. 329–349

29 WOOD, D.C., and MOORES, J.: 'User-defined data types and operators in occam'. Proceedings of WoTUG 22 Architectures, languages and techniques for concurrent systems, in COOK, B. (Ed.): 'Concurrent systems engineering' (IOS Press), Amsterdam, April 1999, Vol. 57, pp. 121–146

30 WELCH, P.H.: 'Process oriented design for Java – concurrency for all, PDPTA 2000' (CSREA Press) June 2000, Vol. 1, pp. 51–57. See also http://www.cs.ukc.ac.uk/projects/ofa/jcsp/ accessed December 2002

31 WELCH, P.H.: 'An occam approach to transputer engineering'. Proceedings of the 3rd. ACM conference on Hypercube concurrent computers and applications, Pasadena, CA, January 1988, see also: http://www.cs.ukc.ac.uk/pubs/1988/245

32 BARNES, F.R.M., and WELCH, P.H.: 'Prioritised dynamic communicating processes: Part I'. Proceedings of WoTUG 25 Communicating process architectures, in PASCOE, J., WELCH, P., LOADER, R., and SUNDERAM, V. (Eds.): 'Concurrent systems engineering' (IOS Press), Amsterdam, Sept. 2002, Vol. 60, pp. 321–351

33 WOOD, D.C., and BARNES, F.R.M.: Post-mortem debugging in KRoC'. Communicating process architectures, Proceedings of WoTUG 23, in WELCH, P., and BAKKERS, A. (Ed.): 'Concurrent systems engineering' (IOS Press), Amsterdam, Sept. 2001, Vol. 60, pp. 329–349

34 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, Dec. 2000

35 VELLA, K.: 'Seamless parallel computing on heterogeneous networks of multiprocessor workstations'. PhD Thesis, The University of Kent at Canterbury, Canterbury, Kent CT2 7NF, Dec. 1998

36 BARNES, F. 'Socket, file and process libraries for occam'. Computing Laboratory, University of Kent at Canterbury, June 2000, (http://www.cs.ukc.ac.uk/people/rpg/frmb2/documents/.) accessed December 2002

37 GOODACRE, I.N.: 'Occam net chans'. Project Rpt., Computing Laboratory, University of Kent, April 2001

38 SCHWEIGLER, M.: 'The distributed occam Protocol – A new layer on top of TCP/IP to serve occam channels over the internet'. MSc Thesis, Computing Laboratory, University of Kent at Canterbury, Sept. 2001

39 WELCH, P.H., ALDOUS, J.R., and FOSTER, J.: 'CSP networking for Java (JCSP.net)'. Proceedings of ICCS Computational science, in SLOOT, P., TAN, C., DONGARRA, J., and HOEKSTRA, A. (Eds.): Lecture Notes in Computer Science (Springer-Verlag, April 2002), Vol. 2330, pp. 695–708 (http://www.cs.ukc.ac.uk/pubs/2002/1382)

40 WELCH, P.H., and VINTER, B.: Cluster computing and JCSP networking'. Proceedings of WoTUG-25 Communicating process architectures, in PASCOE, J., WELCH, P., LOADER, R., and SUNDERAM, V. (Eds.): 'Concurrent Systems Engineering' (IOS Press), Amsterdam, Sept. 2002, Vol. 60, pp. 203–222

41 INMOS LIMITED: 'The T9000 transputer instruction set manual'. SGS-Thomson Microelectronics, 1993, Document 72 TRN 240 01

42 SUNTER, J.P.E., WIJBRANS, K.C.J., and BAKKERS, A.W.P.: 'Co-operative priority scheduling in occam'. Proceedings of the 13th occam user group technical meeting: Real-time Systems with transputers, ZEDAN, H. (Ed.): 'Transputer and occam Engineering' (IOS Press), Amsterdam, Sept. 1990, pp.175–185

43 WELCH, P.H.: 'Multipriority scheduling for transputer-based real-time control'. Proceedings of the 13th occam user group technical meeting: Real-time systems with transputers, in ZEDAN, H. (Ed.): 'Transputer and occam Engineering' (IOS Press), Amsterdam, Sept. 1990, pp. 198–214

44 LIU, C.L., and LAYLAND, J.W.: 'Scheduling algorithms for multiprogramming in a hard real-time environment', *J. ACM 20*, 1973, **1**, pp. 46–61

45 GOLDSMITH, M.H., ROSCOE, A.W., and SCOTT, B.G.O.: 'Denotational semantics for Occam2, Part 1', *Transput. Commun.*, 1993, **1**, (2), pp. 65–91

46 GOLDSMITH, M.H., ROSCOE, A.W., and SCOTT, B.G.O.: 'Denotational Semantics for Occam2, Part 2', *Transput. Commun.*, 1994, **2**, (1), pp. 25–67

47 LAWRENCE, A.E.: 'Extending CSP: denotational semantics', *IEE Proc. Softw.*, 2003, **150**, pp. 51–60

48 LAWRENCE, A.E.: 'CSP extended: imperative state and true concurrency', *IEE Proc. Softw.*, 2003, **150**, pp. 61–69

49 BARNES, F.R.M.INMOS LIMITED: 'Dynamic occam processes'. Fringe-session presentation at CPA-2000, 2000 (http://frmb.home.cern.ch/frmb/pubs/dynoccam-slides.ps) accessed December 2002

50 FORD, B., BACK, G., BENSON, G., LEPREAU, J., LIN, A., and SHIVERS, O.: 'The flux OSKit: A substrate for kernel and language research'. Proceedings of symposium on operating systems principles, 1997, pp. 38–51 (Software available from: http://www.cs.utah.edu/flux/oskit/) accessed December 2002