Communicating Process Architectures – 2002
James Pascoe, Peter Welch, Roger Loader and Vaidy Sunderam (Eds.)
IOS Press, 2002

Prioritised Dynamic Communicating Processes: Part II

Fred BARNES and Peter WELCH
Computing Laboratory, University of Kent, Canterbury, KENT. CT2 7NF
{frmb2,phw}@ukc.ac.uk

Abstract. This paper illustrates the work presented in 'Part I', giving additional examples of use of channel-types, extended rendezvous and FORKs that lean towards real applications. Also presented are a number of other additions and extensions to the occam language that correct, tidy up or complete facilities that have long existed. These include fixing the PRI ALT bug, allowing an unconditional SKIP guard as the last in a PRI ALT, replicator STEP sizes, run-time computed PAR replication counts, RESULT parameters and abbreviations, nested PROTOCOL definitions, inline array constructors and parallel recursion. All are available in the latest release (1.3.3) of KRoC, freely available (GPL/open source) from: www.cs.ukc.ac.uk/projects/ofa/kroc/ .

1 Introduction

The previous paper [1] presented a number of extensions to the occam [2, 3] language within the framework of KRoC/Linux [4]. This paper provides further examples for some of those extensions, specifically mobile channel-types, the extended rendezvous and the FORK.

Section 2 gives an example of a farmer-worker-harvester farm, implemented using FORK to create worker processes as needed. Also presented is a more traditional use of FORKs, using explicitly (compiler #PRAGMA) SHARED variables.

The extended rendezvous can be used to *intercept* a channel, without affecting the end-to-end synchronisation between the processes either side. When used with (mobile) channel-types, the extended rendezvous can be used to *re-wire* the process network (e.g. to plug in infrastructure for *distributed* occam channels – KRoC.net [5]) without affecting the synchronisation between the affected processes. Section 4 gives an example of this, along with a more complex example which uses the 'FORK' (dynamic parallel process creation) as well.

Section 5 gives details of a number of other additions to the occam language, which provide various new features and tidy up some old ones. These modifications have little or no impact on the syntax of the language, being mostly changes to the occam compiler (occ21) and the supporting run-time kernel (a heavily modified version of CCSP [6]).

2 Dynamic Process Farms

One application of FORK is for the dynamic creation and control of process *farms*. Figure 1 shows the process network for a worker-farm, with a pool.manager to control the number of FORKed processes running.

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

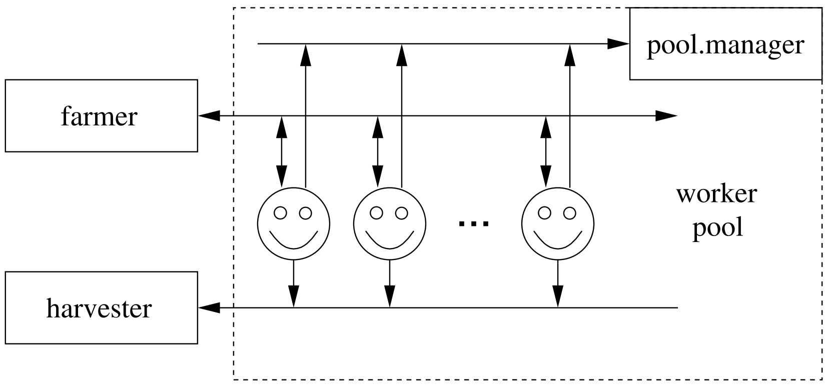

Figure 1: A `FORK`ed worker-farm process network

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?:              -- working (-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
      CLAIM signal
        signal[idle.count] ! -1        -- tell manager we're working
      ...  do work on 'job'  (will involve communicating results to 'out')
      CLAIM signal
        signal[idle.count] ! +1        -- tell manager we're done
:

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 (CLONE work.to.workers, CLONE work.from.workers,
                               CLONE signal.cli)
                n.idle := min.idle
            TRUE
              SKIP

          INT n:
          SEQ
            signal.svr[idle.count] ? n   -- working (-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. Also, no memory leaks occur as the space for such arrays is automatically recycled when the variables go out of scope or are overwritten.

3 Data Sharing and `FORK`ed Processes

With the introduction of dynamic process creation using the `FORK`, new opportunities for shared data *race-hazards* arise. Currently these can be handled through the use of explicit compiler directives to disable usage-checking (‘`#PRAGMA SHARED`’) and locks such as the `SEMAPHORE` and `CREW` [7] *user-defined types* [8] that provide, respectively, exclusive and CREW (*concurrent read exclusive write*) access to shared data. Data passed by reference to `FORK`ed processes must have parallel usage checking disabled, else that will be quite properly rejected. However, no checks are then made to ensure correct usage patterns patterns for lock *claims* and *releases* – or even that the necessary locks are passed and used at all! We are considering providing direct language support for parallel data sharing that will enforce secure (and very low overhead) control [7].

Meanwhile, here is an example of data sharing across `FORK`ed processes that has the necessary security explicitly programmed:

```
#USE "course.lib"
#INCLUDE "crew.inc"

PROC worker (VAL INT id, []INT data, CREW data.crew, SHARED CHAN BYTE out!)
  SEQ
    CLAIM out
      SEQ
        out.string ("worker ", 0, out!)
        out.int (id, 0, out!)
        out.string (" starting*n", 0, out!)
    WHILE TRUE
      SEQ
        claim.read.crew (data.crew)
        ...  read from ‘data’
        release.read.crew (data.crew)
        ...  local processing
        claim.write.crew (data.crew)
        ...  write results back to ‘data’
        release.write.crew (data.crew)
:

PROC example (CHAN BYTE kyb?, SHARED CHAN BYTE scr!, err!)
  INT n:
  [128]INT shared.data:
  #PRAGMA SHARED shared.data
  CREW data.crew:
  #PRAGMA SHARED data.crew
  SEQ
    initialise.crew (data.crew)

    CLAIM scr
      ask.int ("how many ? ", n, 4, kyb?, scr!)
    FORKING
      SEQ i = 0 FOR n
        FORK worker (i, data, data.crew, CLONE scr!)
:
```

This simply asks the user for a count, then launches that many '`worker`' processes. Each process launched claims the shared output channel '`out`' (to the screen), reports its existence then goes into an infinite processing loop – having released the shared output channel (controlled by the `CLAIM`). The first half of the loop claims *concurrent read* access to the data, reads the needed data for processing (not shown), then releases the read lock. Local processing on the data is then performed (not shown). The remainder of the loop claims *exclusive write* access to the data, writes any results (not shown), then releases the write lock.

The channel '`out`' in the '`worker`' process and the '`scr`' and '`err`' channels on the '`example`' process are an *anonymous* form of a `SHARED` channel-type, explained fully in section 5.9.

This example is such that the `FORKING` is not strictly required, since the following two processes are equivalent (and this equivalence holds for any parameters that the `PROC` '`P`' might take):

```
FORKING
  SEQ i = 0 FOR n
    FORK P (i)
```

```
PAR i = 0 FOR n
  P (i)
```

where we make use of the n-replicated `PAR` extension (section 5.4).

4 Extended Rendezvous and Channel-Types

Figure 2 shows a multiple client-server network that used 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!:
:

...  client and server PROCs

SHARED CHAN APP.LINK? link:       -- any-to-any mobile channel (section 5.9)
PAR
  PAR i = 0 FOR num.clients
    client (CLONE link!)          -- start client with CLONE of the output-end
  PAR i = 0 FOR num.servers
    server (CLONE link?)          -- start server with CLONE of the input-end
```

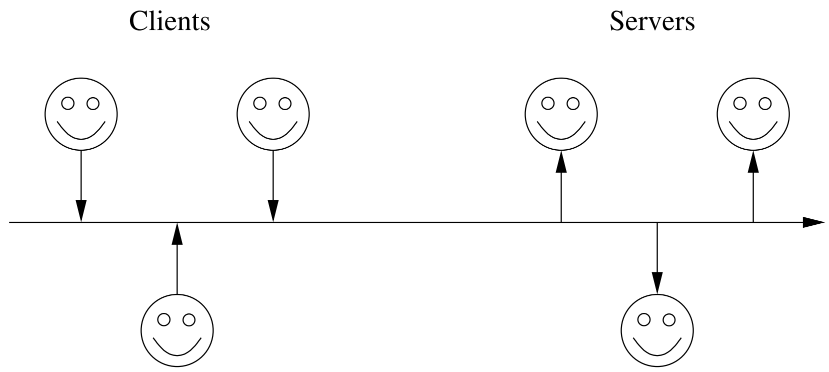

Figure 2: Multiple client-server network

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
      l.cli, l.svr := MOBILE APP.LINK    -- create one-to-one
                                        -- mobile channel-structure
      CLAIM out
        out ! l.svr                     -- communicate server-end (and lose it)

      ...  use 'l.cli' to communicate with a 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 the client
:
```

Figure 3 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`.

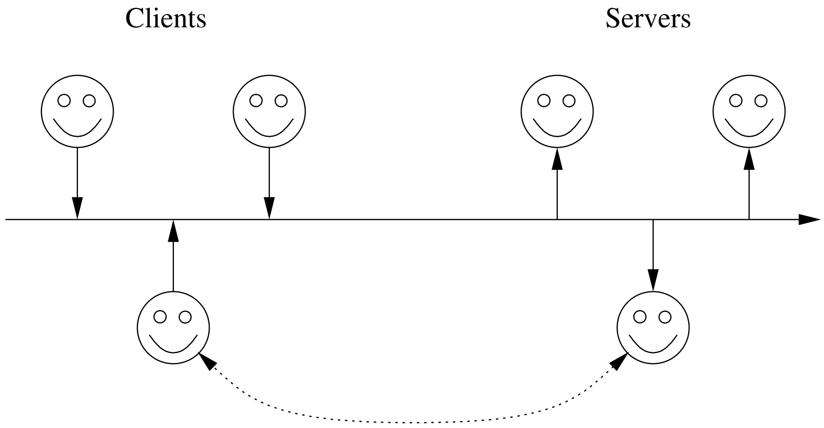

Figure 3: Multiple client-server network with a connected client and server

Using the extended rendezvous with channel-types opens up some interesting possibilities. Figure 4 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.

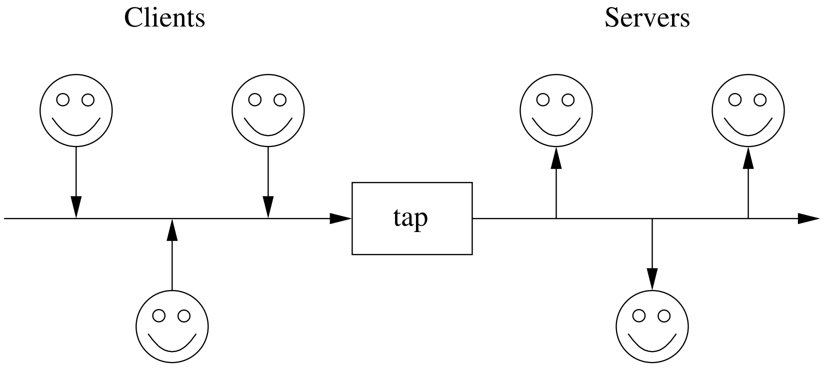

Figure 4: Multiple client-server network with a '`tap`' process

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

```
PROC tap.app.link (CHAN APP.LINK? in?, out!, SHARED LOG! to.log)
  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, CLONE to.log)
:
```

The '`tap.app.link`' 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. Figure 5 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. Figure 5 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’
:
```

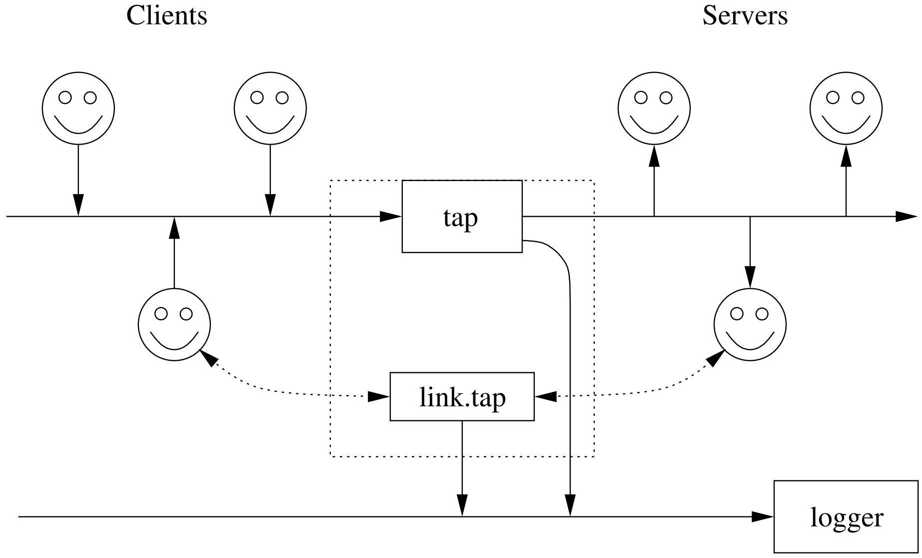

Figure 5: Multiple client-server network after forking a '`link.tap`'

`PAR` is used here to handle both channels in the chan-type independendly. 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. In order 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. Figure 6 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, descoping 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 [9] 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 [10] to implement the networking, as has been done by Goodacre [11] in a student project and by Schweigler [5] in his M.Sc. thesis. A similar functionality also exists in JCSP [12, 13, 14], which additionally allows the migration of processes.

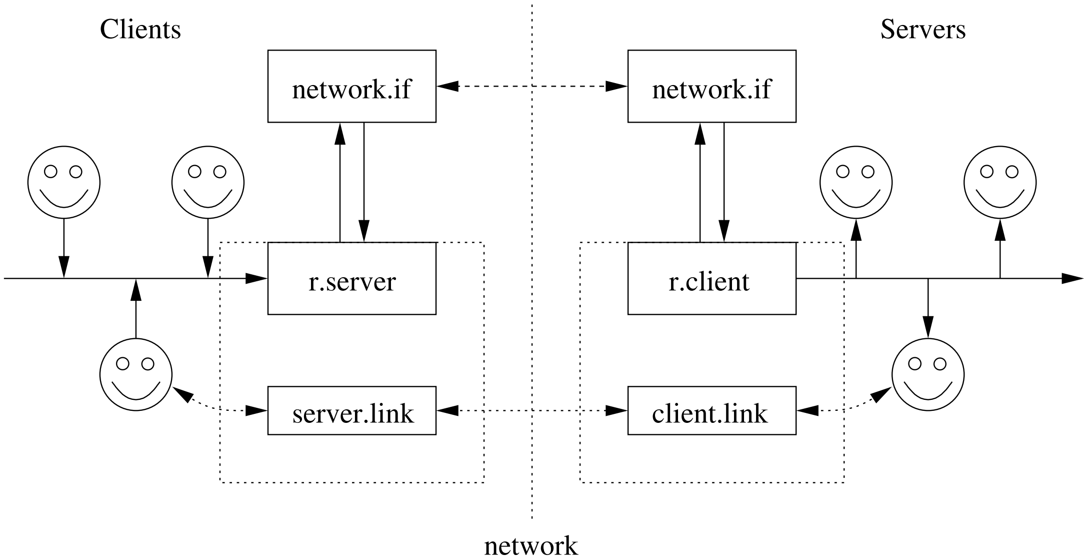

Figure 6: Remotely connected client-server network after communication and creation of link processes

5 Additional occam Extensions

This section describes a number of additional extensions to occam, which tidy-up minor deficiencies in the language as it stood and also adds to it (such as the aforementioned *n*-replicated `PAR`).

5.1 `STEP` *in Replicators*

One thing which occam has always lacked is a way of specifying a *step* size in replicators. Strictly speaking, it is not needed, since the same effect is easy to achieve with the appropriate abbreviation. Syntactically, a simple "`STEP` *exp*" may be added to replicators – for example, to sum the odd elements of an `INT` array:

```
INT odd.sum:
SEQ
  odd.sum := 0
  SEQ i = 1 FOR (SIZE array) / 2 STEP 2
    odd.sum := odd.sum + array[i]
```

The equivalent code, without a `STEP` in the replicator, would be:

```
INT odd.sum:
SEQ
  odd.sum := 0
  SEQ i = 1 FOR (SIZE array) / 2
    VAL INT i IS (2 * i) - 1:
    odd.sum := odd.sum + array[i]
```

In the same way as the *start* and *length* expressions in a replicator, variables used in the step expression are *fixed* – i.e. they may not be the target of assignment or communication in the replicated process. The implementation of different `STEP` sizes is handled with two new loop-end instructions. One of these is specific for a step-size of -1, the other handles arbitrary step sizes.

`STEP` expressions may be used in all replicators: `IF`, `ALT`, `SEQ` and `PAR`.

5.2 *Modified* ALT *Disabling Sequence*

The ALT is implemented by *enabling* and *disabling* instructions for each type of *guard*. When the ALT is entered, the guards are enabled one-by-one. After enabling, if none of the guards are ready, the ALTing process is descheduled – it will be rescheduled by a timeout (on a timeout guard), or by an outputting process in the case of channel guards. Once rescheduled (or if any ready guards were found during enabling), each guard is disabled. This is done sequentially from the first guard to the last guard (same as the enabling sequence). For each disabling guard, if the guard has *fired* (become ready), a pointer to the *guarded-process* is stored inside the ALTer's workspace, but only if no previous guards have become ready (determined by checking the guarded-process pointer in the ALTer).

The same scheme was used in the implementation of both ALTs and PRI ALTs, such that there was no visible difference between them. They are semantically quite different however.

A new set of disabling instructions (table 1) have been added to the underlying *virtual* transputer byte-code [15, 16], which *do not* check the existing state of the guarded-process pointer. i.e. if the guard is ready, these new instructions *fire* it regardless of any previously fired guards. This causes the last ready guard examined to be selected, always. The modified PRI ALT disabling sequence simply processes the guards in reverse order, such that the first ready guard under the PRI ALT is selected, but without the cost of the standard disabling test. This fixes an old bug in the PRI ALT, which allowed the possibility of the second of two (identical) guards to be wrongly selected (if the guard became ready between disabling of the first and second instances of it in the PRI ALT).

Mnemonic	Parameters	Description
NDISC	*process-addr, pre-cond, chan-addr*	disable channel
NDISS	*process-addr, pre-cond*	disable skip guard
NDIST	*process-addr, pre-cond, timeout*	disable timeout guard

Table 1: New ALT disabling instructions

The modified implementation of the normal ALT is similar to the existing one, except that we use the new instructions. This means that in the implementation of the standard ALT, the last ready guard will be selected, rather than the first (as was the case previously). This dramatically alters the behaviour between PRI ALT and ALT at run-time, in the hope that it will make program errors more obvious – i.e. those where the programmer should have used PRI ALT, but instead used just ALT.

The reverse disabling sequence for PRI ALT has only been made possible by the STEP extension for replicators, since previously there was no cheap way to run a replicated ALT backwards – it would have involved a subtraction for each replication.

5.3 *Modified* SKIP *in* ALT *Checking*

A further modification to ALTs has been implemented, which changes the way SKIP guards are checked. Previously, SKIP guards have always required an explicit pre-condition, often just set to TRUE. This restriction has been relaxed for SKIP guards *which appear as the last guard in a* PRI ALT. Any other placement of the SKIP guard must still use the explicit pre-condition.

When the pre-condition and guard are 'TRUE & SKIP' (or where the precondition evaluates to TRUE in the compiler), checks are made to ensure that it is *not* within a plain ALT and only used as the last guard within a PRI ALT. Any checks that fail generate a warning from the compiler, unless it is in *strict* mode, where it will generate an error instead. For example, the following will generate a compiler warning (or error in strict mode):

```
ALT
  c ? x
    ...  do something
  TRUE & SKIP
    ...  do something else
```

But the obvious polling idiom:

```
PRI ALT
  c ? x
    ...  do something
  SKIP
    ...  do something else
```

is now accepted.

5.4 *N-replicated* PAR*s*

This extension to the compiler allows a replicated PAR to have a non-constant replicator count. In terms of the language, there is almost no change, except that now we allow non-constant counts on PAR replicators. The drawback is that the compiler can no longer easily check the parallel usage of variables and channels inside the replicated process. This is a result of the way usage checking is performed – i.e. by brute force expansion of the replication (whose count value must therefore be known statically). In contrast, Southampton's Portable occam Compiler (SPOC) [17] uses an algebraic checker and thus does slightly better here (unless there's a *mod* operator involved).

The *n*-replicated PAR is implemented using the Brinch-Hansen style memory pools [18]. The workspace and vectorspace for the processes are allocated from the free-lists and put back once the process has finished. Unlike the FORK ([1]), memory is only returned to the free lists after all the replicated PAR processes have terminated. This is partly due to the way in which replicated PARs are handled inside the compiler. In theory, there should be no problem in releasing resources early – processes just need to *resign* [7] from the (implicit) PAR-barrier rather than *sync* on it. We will be looking into this.

Mobilespace is handled slightly differently, since it may not be returned to the free-lists after use – parts of it may have been *moved* elsewhere. Unlike FORKed mobilespaces, which sit on free-lists inside their respective enclosing mobilespaces, these mobilespaces are kept inside a dynamic array, which is referenced by the encompassing mobilespace. The difference is largely due to performance considerations – it is cheaper to perform a read-only array subscription than it is to pull a block off a free-list (which involves both a read and a write). There is a cost associated with the handling of this array though, since it must be able to extend dynamically – e.g., if the replicator is executed with successively increasing counts.

5.5 *Recursion in occam*

Recursion in occam has traditionally been prevented for two main reasons – one practical and one specifically invented to frustrate it. Firstly, the previous lack of dynamic memory would have imposed restrictions on the depth of recursion. Secondly, the scoping of names in occam is such that they only become visible at the end of their declaration. For PROCs, this means its own name is not valid inside its own code.

It is possible to fake recursion, often quite convincingly, by using the scoping of names to an advantage [19]. In a PROC called 'foo', any previously defined PROCs also called 'foo' are in scope and perfectly valid. However, this often breaks when the PROC involved is declared at the outermost level – most UNIX linkers do not care much for multiply defined symbols. Additionally, the depth of recursion is still restricted and there is not much scope for recovery at the *bottom-most* level – i.e. the first defined PROC of that name.

A version of recursion using a special locally defined PROC (with a very similar name) has been implemented for the Sparc version of KRoC by Wood in [20]. The Linux version of KRoC (i.e. the work presented here) supports recursion by a slightly different language mechanism, but with virtually the same (Brinch Hansen [18]) implementation for workspace and vectorspace. PROCs which wish to be recursive must indicate this in their name using the 'REC' or 'RECURSIVE' keywords, for example:

```
RECURSIVE PROC thing (...)
  ...  body of thing
:
```

This modifier simply brings the name 'thing' into scope early, thereby permitting its use within the body of 'thing'. Another example of parallel recursion is the 'sieve' process from the parallel recursive version of the Sieve of Eratosthenes:

```
RECURSIVE PROC sieve (VAL INT count, CHAN INT in?, out!)
  IF
    count = 0
      WHILE TRUE
        INT tmp:
        SEQ
          in ? tmp
          out ! tmp
    TRUE
      CHAN INT c:
      INT n:
      SEQ
        in ? n
        out ! n
        PAR
          filter (n, in?, c!)
          sieve (count - 1, c?, out!)
:
```

The 'count' parameter is used to limit the recursion. In the test-harness for this, it is set to a little under 4800 initially, enough to generate all the prime numbers less than 1 million.

The implementation for recursive workspace and vectorspace is handled using the standard free-lists, allocated dynamically on recursive instances. Mobilespace is implemented using a form of nested free-list. In the above example, if 'sieve' required mobilespace (which is not the case here) a *pointer-slot* would be allocated in its mobilespace for holding the mobilespace of the recursive instance. This is initialised to MOSTNEGINT on PROC entry, along with the mobiles already required, and allocated at the point of the recursive instance.

5.6 RESULT *Parameters and Abbreviations*

Result parameters were suggested by Barrett for occam 3 [21]. These are reference parameters which explicitly return results, as opposed to standard reference parameters, whose input/output behaviour is unknown. The *undefined-usage checker* [22] expects reference parameters to have defined arguments before the call and to leave defined data before returning. RESULT parameters change the behaviour of the undefinedness checker, which only checks for *definedness* at the point the PROC finishes, not when it is called. For example:

```
PROC sum.ints (CHAN INT in?, RESULT INT r)
  SEQ
    r := 0
    ...  read integers and modify r
:

...

INT x:
SEQ
  sum.ints (data.in?, x)
  data.out ! x
```

Without the RESULT parameter, the undefined-usage checker will complain about 'x' not being defined at the point of the call to 'sum.ints'. (In actual fact, this will compile without warning since the undefined-checker examines the body of 'sum.ints' and can see that 'x' is not read from before being written to. Separate compilation of a RESULT-less 'sum.ints' will generate this warning – or an error in strict mode).

Result abbreviations follow similar lines, i.e. only pragmatic changes in the compiler. Result abbreviations are less common than their parameter counterparts, but are encountered when a RESULT parameter is turned into an abbreviation in an INLINE PROC. Inlining the above code for example gives:

```
...

INT x:
SEQ

  --{{{  INLINE PROC
  CHAN INT in? IS data.in?:
  RESULT INT r IS x:
  SEQ
    r := 0
    ...  read integers and modify r
  --}}}

  data.out ! x
```

Both RESULT parameters and abbreviations involve no significant changes in the compiler code-generator and are handled in the same way as standard reference parameters and abbreviations – i.e. just dropping the RESULT keyword.

5.7 Nested PROTOCOL *Definitions*

One minor irritation of the existing occam was the inability to use user-defined PROTOCOLs as a component of another PROTOCOL. This is something we frequently wish to do, the alternative being to copy the relevant chunk of PROTOCOL.

An example of a nested *sequential* PROTOCOL is:

```
PROTOCOL PACKET IS INT; INT::[]BYTE:     -- id; data

PROTOCOL LINE.DATA
  CASE
    packet; PACKET
    timeout
:
```

Previously, this would not have been allowed by the compiler and we would have had to expand the declaration ouselves to:

```
PROTOCOL LINE.DATA
  CASE
    packet; INT; INT::[]BYTE:            -- id; data
    timeout
:
```

Variant (CASE) protocol nesting is handled slightly differently. Using the above 'LINE.DATA' protocol for example:

```
PROTOCOL INTERNAL
  CASE
    error; INT                           -- internal error
    FROM LINE.DATA                       -- include LINE.DATA cases
:
```

The use of the 'FROM' keyword is just to emphasize the point that we are *literally including* the cases *from* the 'LINE.DATA' protocol. There is no mysterious sub-typing or inheritance here. Variant protocol *inheritance* and usage have been investigated by Locke [23] and is possible to implement, but has not been yet. The proposed mechanism would allow a 'CHAN OF LINE.DATA' to be supplied as the argument to a 'CHAN OF INTERNAL' formal parameter. This is not the case as things stand however – 'INTERNAL' and 'LINE.DATA' are unrelated protocols.

When nesting variant protocols, the compiler checks that all the tags remain distinct, as it does for a flat protocol definition. Any conflicts result in a standard compiler error being produced.

5.8 Array Constructors

Array constructors add a simple new functionality to occam. It is useful mainly because the equivalent code looks somewhat peculiar. The syntax of array construction here is very similar to the similar list construction operation commonly found in *functional* languages (Miranda [24] for example). Their principle use is to initialise the elements of an array, for example:

```
[10]INT X:
SEQ
  X := [i = 0 FOR SIZE X | (3 * i) - 1]:

  ...  use 'X'
```

The array constructor is an expression, so it must follow existing rules for such – i.e. no side-effects. Although this example is trivial (we could have just used a simple SEQ loop to initialise the array), it may be used in communication, as a parameter, or even be subscripted or sliced.

The general form of the array constructor is as follows:

"[" *name* = *start* FOR *count* [STEP *stride*] "|" *expression* "]"

Only the *count* is used for generating the resulting array. The *start* and *stride* affect only the replicator *name* value in the *expression*. The *name*, *start*, *stride* and *count* types are all INTs.

Array constructors can be nested to create *n*-dimensional arrays. For example:

```
[100][100]REAL64 mesh:
SEQ
  mesh := [i = 0 FOR SIZE mesh |
            [j = 0 FOR SIZE mesh[i] |
              (SIN ((REAL64 TRUNC i) / scale)) *
              (COS ((REAL64 TRUNC j) / scale)) ]]

  ...  use computed 'mesh'
```

The array constructor is implemented by turning it (internally) into a VALOF process. These are essentially *in-line* FUNCTIONs, but due to their very peculiar syntax are not used much. The earlier example of a one-dimensional constructor assigned to 'X' is expanded by the compiler into:

```
[10]INT X:
SEQ
  X := ([10]INT temp:
        VALOF
          SEQ i = 0 FOR SIZE X
            temp[i] := (3 * i) - 1
          RESULT temp
        )

  ...  use 'X'
```

The expansion of the 'mesh' expression is much more convoluted, as is the case generally with inline VALOF processes. Array constructors provide this functionality, but in a nice, simple and consistent way.

In the same way as standard arrays (and array returning FUNCTIONs/VALOFs), array constructors may be subscripted or used in a slice. The compiler will generate *lazy evaluation* code to compute these where possible, including any necessary checks for array-bounds. For example:

```
INT v, i:
SEQ
  ...  compute 'i'
  v := [n = 10 FOR 50 STEP -1 | foo (n)][i]
```

simplifies to:

```
INT v, i:
SEQ
  ...  compute 'i'
  SEQ
    ...  check to ensure that 'i' is in the closed range [0-49]
    v := foo (10 + (i*(-1)))
```

Additionally, if 'foo' is an INLINE FUNCTION, then the code will reduce to just its body preceded by a VAL INT abbreviation for '(10 + (i*(-1)))'.

5.9 Anonymous Channel-Types

Anonymous channel-types are a convenient way of creating a shared *any-to-any* channel. Syntactically their declaration is like that of an ordinary occam channel, but with a 'SHARED' prefix. For example:

```
SHARED CHAN INT c:
P                                         -- process in the scope of ‘c’
```

Internally, the compiler turns the declaration of 'c' into a channel-type definition (suitably scoped), and a pair of channel-ends (along with suitable initialisation code) which will form the real shared channel. The types and variables created internally by the compiler have invalid occam names, which prevents the accidental de-scoping of a real variable by a compiler generated variable (which would be a serious error). The type and code generated for the above declaration is:

```
CHAN TYPE $anon.INT                       -- compiler generated type
  MOBILE RECORD
    CHAN INT x?:
:

SHARED $anon.INT! c$cli                   -- shared client-end
SHARED $anon.INT? c$svr                   -- shared server-end
SEQ
  c$cli, c$svr := MOBILE $anon.INT        -- allocate and initialise channel
  P
```

Within the body of process 'P', any occurrences of the SHARED channel 'c' will be replaced by either: the 'c$cli' or 'c$svr' end as appropriate; or the appropriate end with a subscription to access the 'real' channel (field 'x' in the CHAN TYPE definition). This selection is controlled by the usage of 'c', i.e. whether a 'SHARED CHAN INT' or a 'CHAN INT' is expected. In the latter case, the shared channel must be CLAIMed before use.

When used in PROC parameters, anonymous channel-types undergo a similar transformation, but apart from any needed type declaration no extra code is generated – only the type and name of the variable are changed. For example:

```
PROC foo (SHARED CHAN INT out!)
  CLAIM out                               -- grab channel-end
    SEQ i = 42 FOR 100 STEP -3            --   send 100 messages
      out ! i                             --   (no competitor interleaving)
:

PROC bar (SHARED CHAN INT out!)
  foo (out!)
:
```

is transformed into:

```
PROC foo (SHARED $anon.INT! out$cli)     -- transformed PROC header
  CLAIM out$cli                          -- transformed claim
    SEQ i = 42 FOR 100 STEP -3
      out$cli[x] ! i                     -- transformed communication
:

PROC bar (SHARED $anon.INT! out$cli)     -- transformed PROC header
  foo (out$cli)                          -- transformed PROC call
:
```

Anonymous (shared) channel-types are still subject to the same parallel usage and aliasing rules as named shared channels. Thus in order to effectively share between parallel processes, `CLONE`s must be used.

6 Conclusions and Further Work

This paper has demonstrated the use of the `FORK`, mobile channel-structure and extended-rendezvous additions to occam and KRoC/Linux, as well as a number of extensions to the existing syntax and semantics. The examples presented so far have been fairly simple, but their scope is far reaching. A new version of the occam web-server visible at [25] is already using these extensions successfully. Other demonstrator applications are on the way.

It is hoped that users of KRoC/Linux and occam will find these extensions useful and report feedback to the community.

References

[1] 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.

[2] Inmos Limited. occam 2.1 Reference Manual. Technical report, Inmos Limited, May 1995. Available at: `http://wotug.ukc.ac.uk/parallel/occam/documentation/`.

[3] A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1997. ISBN: 0-13-674409-5.

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

[5] M. Schweigler. The Distributed occam Protocol - A New Layer On Top Of TCP/IP To Serve occam Channels Over The Internet. Master's thesis, Computing Laboratory, University of Kent at Canterbury, September 2001. MSc Dissertation.

[6] J.Moores. CCSP – a Portable CSP-based Run-time System Supporting C and occam. In B.M.Cook, editor, *Architectures, Languages and Techniques for Concurrent Systems*, volume 57 of *Concurrent Systems Engineering series*, pages 147–168, Amsterdam, the Netherlands, April 1999. WoTUG, IOS Press. ISBN: 90-5199-480-X.

[7] 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.

[8] D.C.Wood and J.Moores. User-Defined Data Types and Operators in occam. In B.M.Cook, editor, *Architectures, Languages and Techniques for Concurrent Systems*, volume 57 of *Concurrent Systems Engineering Series*, pages 121–146. WoTUG, IOS Press, the Netherlands, April 1999. ISBN: 90-5199-480-X.

[9] Kevin Vella. *Seamless Parallel Computing on Heterogeneous Networks of Multiprocessor Workstations*. PhD thesis, The University of Kent at Canterbury, Canterbury, Kent. CT2 7NF, December 1998.

[10] Fred Barnes. *Socket, File and Process Libraries for occam*. Computing Laboratory, University of Kent at Canterbury, June 2000. Available at: `http://www.cs.ukc.ac.uk/people/rpg/frmb2/documents/`.

[11] I.N. Goodacre. occam NetChans, 2001. Project report.

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

[13] P.H.Welch, J.R.Aldous, and J.Foster. CSP networking for java (JCSP.net). In P.M.A.Sloot, C.J.K.Tan, J.J.Dongarra, and A.G.Hoekstra, editors, *Computational Science - ICCS 2002*, volume 2330 of *Lecture Notes in Computer Science*, pages 695–708. Springer-Verlag, April 2002. ISBN: 3-540-43593-X.

[14] P.H. Welch and B. Vinter. Cluster Computing and JCSP Networking. In James Pascoe, Peter Welch, Roger Loader, and Vaidy Sunderam, editors, *Communicating Process Architectures 2002*, WoTUG-25, Concurrent Systems Engineering, pages 213–232, IOS Press, Amsterdam, The Netherlands, September 2002.

[15] Inmos Limited. *The T9000 Transputer Instruction Set Manual*. SGS-Thompson Microelectronics, 1993. Document number: 72 TRN 240 01.

[16] 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.

[17] S. Wykes M. Debbage, M. Hill and Dennis Nicole. Southampton's portable occam compiler (SPOC). In R. Miles and A. Chalmers, editors, *Proceedings of WoTUG 17: Progress in Transputer and Occam Research*, volume 38 of *Concurrent Systems Engineering*. IOS Press, The Netherlands, April 1994. ISBN: 90-5199-163-0.

[18] Per Brinch Hansen. Efficient Parallel Recursion. *ACM SIGPLAN Notices*, 30(12):9–16, December 1995. Reprinted in: *The Origin of Concurrent Programming*, edited by Per Brinch Hansen, pp. 525-534, Springer, ISBN 0-387-95401-5. 2002.

[19] Michael D. Poole. Fixed Maximal Depth Recursion in occam. Number 16 in OUG Newsletter. IOS Press, Netherlands, January 1992.

[20] D.C. Wood. An Experiment with Recursion in occam. In P.H.Welch and A.W.P.Bakkers, editors, *Communicating Process Architectures, Proceedings of WoTUG 23*, volume 58 of *Concurrent Systems Engineering*, pages 193–204, Amsterdam, the Netherlands, September 2000. WoTUG, IOS Press. ISBN: 1-58603-077-9.

[21] Geoff Barrett. occam 3 Reference Manual. Technical report, Inmos Limited, March 1992. Available at: `http://wotug.ukc.ac.uk/parallel/occam/documentation/`.

[22] F.R.M. Barnes and P.H. Welch. Mobile Data, Dynamic Allocation and Zero Aliasing: an occam Experiment. In Majid Mirmehdi Alan Chalmers 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.

[23] T.S. Locke. Towards a Viable Alternative to OO – extending the occam/CSP programming model. In Majid Mirmehdi Alan Chalmers and Henk Muller, editors, *Communicating Process Architectures 2001*, volume 59 of *Concurrent Systems Engineering*, pages 329–349, Amsterdam, The Netherlands, September 2001. WoTUG, IOS Press. ISBN: 1-58603-202-X.

[24] Simon Thompson. *Miranda: The Craft of Functional Programming*. Addison Wesley, July 1995. ISBN: 0-201-42279-4.

[25] F.R.M. Barnes. The occam Web-Server Home Page, 2000. Available at: `http://wotug.ukc.ac.uk/ocweb/`.