Higher Levels of Process Synchronisation

Peter H. Welch and David C. Wood

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

{P.H.Welch, D.C.Wood}@ukc.ac.uk

Abstract. Four new synchronisation primitives (SEMAPHOREs, RESOURCEs, BARRIERs and BUCKETs) were introduced in the KRoC 0.8beta release of occam for SPARC (SunOS/Solaris) and Alpha (OSF/1) UNIX workstations [1][2][3]. This paper reports on the rationale, application and implementation of two of these (SEMAPHOREs and BARRIERs). Details on the other two may be found on the web [4].

The new primitives are designed to support higher-level mechanisms of SHARING between parallel processes and give us greater powers of expression. They will also let greater levels of concurrency be safely exploited from future parallel architectures, such as those providing (virtual) shared-memory. They demonstrate that occam is neutral in any debate between the merits of message-passing versus shared-memory parallelism, enabling applications to take advantage of whichever paradigm (or mixture of paradigms) is the most appropriate.

The new primitives could be (but are not) implemented in terms of traditional channels, but only at the expense of increased complexity and computational overhead. The primitives are immediately useful even for uni-processors – for example, the cost of a *fair* ALT can be reduced from *O(n)* to *O(1)*. In fact, all the operations associated with new primitives have constant space and time complexities; and the constants are very low.

The KRoC release provides an Abstract Data Type interface to the primitives. However, direct use of such mechanisms still allows the user to misuse them. They must be used in the ways prescribed (in this paper and in [4]) else their semantics become unpredictable. No tool is provided to check correct usage at this level.

The intention is to bind those primitives found to be useful into higher level versions of occam. Some of the primitives (e.g. SEMAPHOREs) may never themselves be made visible in the language, but may be used to implement bindings of higher-level paradigms (such as SHARED channels and BLACKBOARDs). The compiler will perform the relevant usage checking on all new language bindings, closing the security loopholes opened by raw use of the primitives.

The paper closes by relating this work with the notions of virtual transputers, microcoded schedulers, object orientation and Java threads.

1 Channels are not enough

An occam channel is a primitive combining communication and synchronisation. As a synchronisation primitive, it applies to two processes at a time. Some applications require many processes to synchronise before any can continue – for example, the barrier synchronisations used by common shared-memory parallel algorithms.

Multi-way synchronisation is a fundamental idea in CSP [5], but is not implemented in occam. The computational arrangements for allowing any of the synchronising processes to back off (which CSP allows) is even more costly than allowing both parties to back off during channel synchronisation. However, just as allowing only the receiver to back off an offer to communicate enabled an efficient channel implementation in occam, a similarly drastic rule – allowing no parties to back off a multi-way synchronisation – makes possible an efficient implementation of the CSP `BARRIER`. Does such a restriction still leave a useful primitive? Just as for occam channels, the answer seems to be yes.

A different way of looking at channels is that they provide a peg on which to hang a blocked process. If we have lots of processes we wish to suspend for some common reason (e.g. they are waiting on a common event or for some shared resource, access to which is restricted by some rules), we either have to have lots of channels on which to hang them (and, later, organise their release) or we put them on a timer queue. Neither of these may be convenient or computationally light.

What are needed are different kinds of peg on which we may hang arbitrary numbers of processes ... plus the ability to retrieve them one at a time (`SEMAPHORE`s and `RESOURCE`s) ... or all at once (`BARRIER`s and `BUCKET`s) ... or some other way ...

2 Abstract Data Types

Each new primitive is presented as an Abstract Data Type. Each is implemented as an occam2.1 [6] `DATA TYPE`, together with a set of operations defined through `INLINE`d occam2.1 `PROC`s and `FUNCTION`s.

Full source code is provided in the KRoC 0.9beta occam system. Each primitive is accessed through a separate `#INCLUDE` file. KRoC releases may be found on the Internet Parallel Computing Archive:

```
 <URL:http://www.hensa.ac.uk/parallel/occam/projects/occam-for-all/kroc/>
<URL:ftp://unix.hensa.ac.uk/pub/parallel/occam/projects/occam-for-all/kroc/>
```

Although users have visibility of the data structures used for each primitive, advantage must *not* be taken of this visibility. Components of the data structures must *not* be accessed directly by user programs. Instances of the primitives may *only* be operated on by calling the `PROC`s and `FUNCTION`s provided.

3 SEMAPHOREs

3.1 SEMAPHORE Abstract Data Type

These implement classic counting semaphores:

- `DATA TYPE SEMAPHORE`

  Users may declare their own `SEMAPHORE` variables and pass them as reference parameters. One `SEMAPHORE` should be declared to control access to each shared resource (which could be a data or channel structure). `SEMAPHORE`s must not be duplicated by assignment or communication through channels.

- `PROC initialise.semaphore (SEMAPHORE s, VAL INT count)`

  Each `SEMAPHORE` must be initialised with this routine before it is used. The `count` value is the number of processes allowed simultaneous access to the shared resource. For exclusive access, set this to 1.

- `PROC claim.semaphore (SEMAPHORE s)`

  Before accessing the shared resource, a process must call this routine to claim the associated `SEMAPHORE`. If there are less than `count` (where `count` is the value with which the `SEMAPHORE` was initialised) processes using the shared resource, this process will be allowed through – i.e. the call will return immediately. Otherwise, the process will be blocked and put on the queue of processes associated with the `SEMAPHORE`.

- `PROC release.semaphore (SEMAPHORE s)`

  When a process has finished with the shared resource, it must call this routine to register its release of the associated `SEMAPHORE`. If there are processes waiting to claim that `SEMAPHORE`, the first process on that queue is re-scheduled – i.e. allowed through to use the resource.

3.2 The normal pattern of use

So, the normal pattern of use is:

```
...  thing declaration (where thing is to be SHARED by many processes)
#PRAGMA SHARED thing     -- suppress parallel usage checking
SEMAPHORE thing.s:
#PRAGMA SHARED thing.s   -- suppress parallel usage checking

SEQ
  initialise.semaphore (thing.s, 1)   -- for exclusive access (for example)
  PAR
    ...  process using thing
    ...  another process using thing
    ...  another process using thing
    ...  another process using thing
    ...  etc.
```

Within each '`process using thing`', each use must be protected within a `claim` and `release`:

```
SEQ
  claim.semaphore (thing.s)
  ...  now use thing
  release.semaphore (thing.s)
```

[Note: in the literature, `claim` is sometimes referred to as `wait` (or `P`) and `release` is sometimes called `signal` (or `V`).]

3.3 occam3 SHARED channels (via SEMAPHOREs)

The main motivation for implementing `SEMAPHORE`s is to support the occam3 `SHARED` channel [7]. This is a language construct to describe client-server applications, where multiple clients compete for exclusive access to a single server. In occam2, this has to be implemented through an array of channels (or channel-pairs for two-way interaction) over which the server performs a *fair* `ALT`. The problems with this are:

- the array of channels has to be declared and made visible to the server, which means that the number of clients has to be known at the point where the server is installed;
- the computational complexity of the server `ALT` is *O(n)*, where n is the number of clients. For not very large n, especially in a hard real-time application, this can become prohibitive.

On the other hand, with the `SEMAPHORE` implementation of a `SHARED` channel:

- there is a fixed-sized space overhead (3 words), regardless of the number of clients;
- the computational complexity of setting up and closing down each client-server transaction is *O(1)* – i.e. independent of the number of clients and the same order of magnitude as an ordinary context switch (sub-microsecond);
- the server does not know that the client end is `SHARED` – it sees an ordinary channel (or channel-pair). This means that a server may `ALT` over a set of `SHARED` (or ordinary) channels using normal mechanisms.

3.3.1 Client transactions over a SHARED channel

An occam3 `SHARED` channel (or channel-pair) connects any number of client processes with a server process. To use the `SHARED` channel, the client process must first claim it:

```
CLAIM c
  ...  use any of the channels within c
```

occam3 has a `CHAN TYPE` structure that allows us to group a collection of channels (each with differing `PROTOCOL`s and directions of use) as fields in a record:

```
CHAN TYPE CONNECT  -- CONNECT is the user-chosen name for this channel type
  RECORD
    CHAN OF REQUEST request:
    CHAN OF REPLY reply:
:
SHARED CONNECT c:
```

So, a typical transaction might look like:

```
CLAIM c
  --{{{  use any of the channels within c
  SEQ
    c[request] ! some.request
    c[reply] ? some.reply
    ...   follow-up questions and answers
  --}}}
```

Note that any attempted use of `c` outside a `CLAIM` body would be jumped on by the compiler. occam3 also forbids any synchronisation attempts inside the `CLAIM` body other than those involving `c`. In particular, a process is not allowed to accumulate resources through nested `CLAIM`s (which eliminates the classic danger of deadlock through partially acquired resources).

3.3.2 SHARED channels in occam2.1 (client end)

In occam2.1, the channel components need to be declared separately, together with a controlling semaphore:

```
CHAN OF REQUEST c.request:
#PRAGMA SHARED c.request      -- suppress parallel usage checking

CHAN OF REPLY c.reply:
#PRAGMA SHARED c.reply        -- suppress parallel usage checking

SEMAPHORE c.s:
#PRAGMA SHARED c.s            -- suppress parallel usage checking
```

The client transaction becomes:

```
SEQ
  claim.semaphore (c.s)
  --{{{  use any of the channels within c
  SEQ
    c.request ! some.request
    c.reply ? some.reply
    ...   follow-up questions and answers
  --}}}
  release.semaphore (c.s)
```

3.3.3 Server transactions over a SHARED channel

At the server end, occam3 establishes the client connection with an explicit:

```
GRANT c
  ...  use any of the channels within c
```

As for the clients, the server is not allowed to use `c` outside a `GRANT` body and any attempt would be disallowed by the compiler. However, servers *are* allowed to make further synchronisations (e.g. `CLAIM`s or other `GRANT`s) within a `GRANT` body.

In this example, a transaction matching the client `CLAIM` might be:

```
GRANT c
  --{{{  use any of the channels within c
  ...  local declarations
  SEQ
    c[request] ? some.request
    ...  compute the correct response
    c[reply] ! some.reply
    ...   follow-up questions and answers
  --}}}
```

3.3.4 SHARED channels in occam2.1 (server end)

The occam2.1 implementation for this `GRANT` is null. It simply maps to:

```
--{{{  use any of the channels within c
...  local declarations
SEQ
  c.request ? some.request
  ...  compute the correct response
  c.reply ! some.reply
  ...   follow-up questions and answers
--}}}
```

Note that, provided each `CLAIM` opens with a communication to the server, `ALT`ing between the `SHARED` channel and any other `ALT` guard (`SHARED` or not) is immediately possible by the server. If the transaction opens with a communication in the other direction, a dummy `request` will need to be added to allow the server to `ALT`.

Finally, some transaction bodies may contain no communications at all! For example, the server may be a `SHARED` signal-handler (where the signal is raised by a client simply making a `CLAIM` with a `SKIP` body). In this case again, a dummy `request` will need to be added to synchronise the client with the server.

3.4 Dining Philosophers (via SEMAPHOREs)

3.4.1 The classical occam model

Sometimes, `SEMAPHORE`s can be used to represent objects in their own right. For example, the forks in Dijkstra's classic Dining Philosophers system are simply binary `SEMAPHORE`s shared by the two philosophers whose place settings are on either side of the fork. In classic occam, they are simply modelled by a fork process such as:

```
PROC fork (CHAN OF BOOL left, right)
  WHILE TRUE
    ALT                        -- should be a ‘fair’ ALT
      BOOL any:
      left ? any               -- philosopher left picks up fork
        left ? any             -- philosopher left puts down fork
      BOOL any:
      right ? any              -- philosopher right picks up fork
        right ? any            -- philosopher right puts down fork
:
```

Similarly, the security guard (or butler), who only allows into the dining room up to four philosophers at a time, is a counting semaphore initialised to four. In classic occam, this is modelled:

```
PROC security ([5]CHAN OF BOOL down, up)
  VAL BYTE max IS 4:
  INITIAL BYTE n.sat.down IS 0:
  WHILE TRUE
    ALT i = 0 FOR 5                         -- should be a ‘fair’ ALT
      ALT
        --{{{  philosopher i wants to sit down
        BOOL any:
        (n.sat.down < max) & down[i] ? any  -- don’t allow more
          n.sat.down := n.sat.down + 1      -- than max at a time
        --}}}
        --{{{  philosopher i wants to stand up
        BOOL any:
        up[i] ? any                         -- always allow this
          n.sat.down := n.sat.down - 1
        --}}}
:
```

A philosopher interacts with two forks and the security guard:

```
  PROC philosopher (CHAN OF BOOL left, right,            -- forks
                    CHAN OF BOOL down, up)               -- security guard
    WHILE TRUE
      SEQ
        ...  think-a-while
        --{{{  get past the security guard
        down ! TRUE
        --}}}
        --{{{  pick up the forks
        PAR
          left ! TRUE       -- pick up left fork
          right ! TRUE      -- pick up right fork
        --}}}
        ...  eat-a-while
        --{{{  put down the forks
        PAR
          left ! TRUE       -- put down left fork (no wait)
          right ! TRUE      -- put down right fork (no wait)
        --}}}
        --{{{  notify security you have finished
        up ! TRUE
        --}}}
  :
```

The college consists of 5 philosophers, 5 forks and the security guard:

```
  PROC college ()
    [5]CHAN OF BOOL left, right, down, up:
    PAR
      security (down, up)
      PAR i = 0 FOR 5
        PAR
          philosopher (left[i], right[i], down[i], up[i])
          fork (left[i], right[(i + 1)\5])
  :
```

3.4.2 The occam2.1 model using SEMAPHOREs

With real `SEMAPHORE`s, there is no need for the fork and security *processes*:

```
PROC college ()

  [5]SEMAPHORE fork:
  #PRAGMA SHARED fork             -- suppress parallel usage checking

  SEMAPHORE security:
  #PRAGMA SHARED security         -- suppress parallel usage checking

  SEQ
    SEQ i = 0 FOR 5
      initialise.semaphore (fork[i], 1)      -- exclusive use
    initialise.semaphore (security, 4)       -- allow four at a time
    PAR i = 0 FOR 5
      philosopher (fork[i], fork[(i + 1)\5], security)
:
```

where the philosopher still interacts with two forks and the security guard:

```
PROC philosopher (SEMAPHORE left, right,     -- forks
                  SEMAPHORE security)        -- security guard
  WHILE TRUE
    SEQ
      ...  think-a-while
      --{{{  get past the security guard
      claim.semaphore (security)
      --}}}
      --{{{  pick up the forks
      PAR
        claim.semaphore (left)      -- pick up left fork
        claim.semaphore (right)     -- pick up right fork
      --}}}
      ...  eat-a-while
      --{{{  put down the forks
      PAR
        release.semaphore (left)    -- put down left fork (no wait)
        release.semaphore (right)   -- put down right fork (no wait)
      --}}}
      --{{{  notify security you have finished
      release.semaphore (security)
      --}}}
:
```

The `SEMAPHORE` implementations of the forks and security guard give us *fair* sharing. No philosopher can get locked out indefinitely by un-thinking neighbours racing back to the dining room and grabbing forks. The `SEMAPHORE` implementations don't need programming; they just need initialising! They give us more functionality and execute far faster than the original processes. However, their use in the above has nothing to do with occam3 `SHARED` channels, so such application requires care.

3.5 Instrumenting parallel systems (via SEMAPHOREs)

One common problem solved by `SHARED` channels is multiplexing data streams to single devices. For example, when animating the behaviour of a network of processes (for diagnostic or demonstration purposes), we want to print information to some display or file. Writing, installing and wiring up the necessary multiplexors to route the information coming from all the processes under inspection can be daunting ... we can't just put in print statements!

Or, at least, that used to be the case!! By making, for example, the screen channel `SHARED`, we can *just-put-in-print-statements* and we can do it within any number of parallel processes and have full control over the atomicity of any particular message.

The dining philosophers' college (from the previous section) will compile and run without deadlock, but is somewhat unexciting to watch – all the action is internal and we can't see it. The following modification instruments the `college` with a single reporting channel that is `SHARED` by all the `philosophers`:

```
PROC college (CHAN OF BYTE screen)

  #PRAGMA SHARED screen

  SEMAPHORE screen.s:
  #PRAGMA SHARED screen.s

  [5]SEMAPHORE fork:
  #PRAGMA SHARED fork

  SEMAPHORE security:
  #PRAGMA SHARED security

  SEQ
    SEQ i = 0 FOR 5
      initialise.semaphore (fork[i], 1)      -- exclusive use
    initialise.semaphore (security, 4)       -- allow four at a time
    initialise.semaphore (screen.s, 1)       -- exclusive use
    PAR i = 0 FOR 5
      philosopher (i, fork[i], fork[(i + 1)\5], security, screen, screen.s)

:
```

Each `philosopher` can now report its state, at any time, to the `screen` channel (provided, of course, it remembers to `claim` and `release` the guarding `SEMAPHORE` correctly). A full version of this code, together with a much more exciting animation, may be found on:

```
<URL:http://www.hensa.ac.uk/parallel/occam/projects/occam-for-all/hlps/>
<URL:ftp://unix.hensa.ac.uk/pub/parallel/occam/projects/occam-for-all/hlps/>
```

The animation was designed by a second year undergraduate (Nick Hollands) at Kent. The system contains some 52 processes (42 to 47 simultaneously active), with 25 driving the screen via a single `SHARED` channel.

4 BARRIERs

4.1 Barrier synchronisation

4.1.1 SPMD barriers

Barrier synchronisation is a common primitive in many models of parallel computing – in some cases, it is an essential element. In SIMD parallelism, there is global synchronisation between all processors after every instruction. In the slightly more flexible SPMD model, there is still just one barrier on which all processors synchronise; however, the point at which synchronisation takes place is application dependent and has to be programmed explicitly (and, usually, cyclically).

For example, SPMD parallelism has the general form:

```
...  shared global data
PAR i = 0 FOR n.processors
  WHILE TRUE                -- one identical serial process per processor
    SEQ
      ...  do something
      SYNC                  -- barrier: wait for all processors to get here
```

occam already imposes an implicit barrier synchronisation at the end of each `PAR` construct. This can be exploited to obtain the above model by moving the external `PAR` inside the serial control structure:

```
...  shared global data
WHILE TRUE
  PAR i = 0 FOR n.processors
    ...  do something
```

We now have a loop of parallel processes, each of which has to terminate, instead of a parallel set of loops that have to synchronise once per cycle. This would be disadvantageous if the start-up/shut-down overheads for parallel processes were large in comparison to their compute times, but this would not normally be the case for occam. A more serious problem arises if the processors use local state that has to survive the barrier:

```
...  shared global data
PAR i = 0 FOR n.processors
  INITIAL INT x IS 0:       -- local state
  WHILE TRUE
    SEQ
      ...  do something
      SYNC
```

This is, of course, a very common requirement. Bringing the parallelism inside the loop forces the set of local states into the global data space:

```
...  shared global data
INITIAL [n.processors]INT X IS [0 | i = 0 FOR n.processors]:
WHILE TRUE
  PAR i = 0 FOR n.processors
    INT x IS X[i]:
    ...  do something
```

which is not very pretty and threatens unnecessary run-time overhead! It also breaks the natural object-oriented encapsulation of local state that occam processes normally provide.

So, we need to introduce the explicit SYNC primitive to regain simplicity.

4.1.2 MIMD named barriers

However, occam is a MIMD parallel language and we don't want to be constrained by SPMD thinking. In particular, we want to obtain a structured and dynamic form of barrier synchronisation. For example, we want to allow our system to be composed of multiple sets of processes, each set with its own local barriers. We also want the flexibility of allowing the number of processes synchronising on any particular barrier to grow and shrink at run-time.

To achieve this, we need to be able to name barriers and associate them with particular sets of processes. A *named* BARRIER is simply a CSP event and its association with a set of processes is just its inclusion in their alphabets. Barrier synchronisation is event synchronisation, but with the restriction that processes cannot use it as a guard in a choice operator (i.e. an ALT guard in occam terms). There is no semantic problem in allowing the number of processes interested in a barrier to change dynamically.

There is no pragmatic problem either for an extended occam:

```
...  shared global data
PAR i = 0 FOR n.processors BARRIER b
  INITIAL BOOL running IS TRUE:
  WHILE running
    SEQ
      ...  do something
      SYNC b                -- named barrier
```

The named BARRIER is declared explicitly by the above PAR construct[1]. This is the only place where BARRIERs can be declared. The BARRIER is automatically in the alphabet of all components of the PAR (which means that when one component SYNCs on it, all components have to SYNC on it).

Since the BARRIER is named, it can be passed as a parameter to PROCs that are called in the body of the declaring PAR. Among other benefits, this allows the separate compilation of processes that are later instanced to synchronise on any BARRIERs the installer chooses.

[1]Declaring items in constructors already takes place in occam – for example, the control value i in this PAR. Some people say that channel declarations ought to be similarly bound to individual PARs.

Note that processes sharing the same BARRIER may terminate at different times – as can happen in the above example. Terminated processes do not block the barrier SYNCs of those that are still running. An elegant application of this principle is given later.

Since this is occam, we are not restricted to the replicated PAR of SPMD. For, example, the following system has three *different* processes synchronising on the named barrier:

```
...  shared global data
PAR BARRIER m
  ...  process A
  ...  process B
  ...  process C
```

We can also have *different groups* of processes synchronising on *different* barriers:

```
...  shared global data
PAR
  PAR BARRIER m
    ...  process A
    ...  process B
    ...  process C
  PAR BARRIER n
    ...  process P
    ...  process Q
    ...  process R
```

4.1.3 Dynamic enrollment in barriers

Finally, an BARRIER synchronising process may contain parallel sub-processes. Normal scoping rules imply that the sub-processes can see the BARRIER and may, therefore, SYNC on it. A logical policy would be to say that the number of processes taking part in the barrier automatically grows for the duration of those sub-processes. However, it is more flexible to be able to specify which sub-processes include the existing BARRIER in their alphabet (and are, therefore, obliged to SYNC if the barrier represented by the BARRIER needs to be overcome during their lifetime). There are two ways to get this: introduce either a *hiding* operator or an *enrolling* operator into the PAR construct. There are arguments both ways but, for now, we prefer the positive approach:

```
...  shared global data
PAR i = 0 FOR n.processors BARRIER b
  IF
    need.more.parallelism (i)
      PAR j = 0 FOR more ENROLL b
        ...  inner processes can (and, probably, better had) SYNC on b
    TRUE
      INITIAL BOOL running IS TRUE:     -- the original code
      WHILE running
        SEQ
          ...  do something
          SYNC b                        -- named barrier
```

Alternatively, components of inner `PAR`s may be enrolled individually in an outer `BARRIER`:

```
...  shared global data
PAR i = 0 FOR n.processors BARRIER m
  IF
    need.more.parallelism (i)
      PAR
        ENROLL m
          ...  process A (includes m in its alphabet)
        ENROLL m
          ...  process B (includes m in its alphabet)
        ...  process C (does not include m in its alphabet)
    TRUE
      INITIAL BOOL running IS TRUE:     -- the original code
      WHILE running
        SEQ
          ...  do something
          SYNC m                        -- named barrier
```

Processes `A` and `B` in the above have to partake in `SYNC m`, but process `C` does not and cannot! Explicit enrollment means that un-enrolled `BARRIER`s are automatically hidden from sub-components of the `PAR` and that any attempt to `SYNC` on them would be rejected by the compiler.

Finally, we note the equivalence in Figure 1.

occam2.x	occam2.x
`PAR ENROLL m`	`PAR`
`  ...  process A`	`  ENROLL m`
`  ...  process B`	`    ...  process A`
`  ...  process C`	`  ENROLL m`
	`    ...  process B`
	`  ENROLL m`
	`    ...  process C`

Figure 1: Factorisation of barrier enrollment across an inner PAR

4.2 BARRIER Abstract Data Type

The current KRoC release implements:

- `DATA TYPE BARRIER`

  Users may declare their own `BARRIER` variables and pass them as reference parameters. They should only be declared in association with the `PAR` construct that sets up the processes that synchronise on them. `BARRIER`s must not be duplicated by assignment or communication through channels.

occam2.x	occam2.1

... shared global data
PAR i = 0 FOR n.procs BARRIER b
  INITIAL BOOL running IS TRUE:
  WHILE running
    SEQ
      ... do something
      SYNC b   -- named barrier
	... shared global data
	BARRIER b:
	#PRAGMA SHARED b
	SEQ
	  initialise.barrier (b, n.procs)
	  PAR i = 0 FOR n.procs
	    SEQ
	      INITIAL BOOL running IS TRUE:
	      WHILE running
	        SEQ
	          ... do something
	          synchronise.barrier (b)
	      resign.barrier (b)

... shared global data
PAR BARRIER m
  ... process A
  ... process B
  ... process C
	... shared global data
	BARRIER m:
	#PRAGMA SHARED m
	SEQ
	  initialise.barrier (m, 3)
	  PAR
	    SEQ
	      ... process A
	      resign.barrier (m)
	    SEQ
	      ... process B
	      resign.barrier (m)
	    SEQ
	      ... process C
	      resign.barrier (m)

PAR ENROLL m
  ... process A
  ... process B
  ... process C
	SEQ
	  enroll.barrier (m, 2)
	  PAR
	    SEQ
	      ... process A
	      resign.barrier (m)
	    SEQ
	      ... process B
	      resign.barrier (m)
	    SEQ
	      ... process C
	      resign.barrier (m)
	  enroll.barrier (m, 1)

Figure 2: Mappings to occam2.1 from SPMD, MIMD and dynamic barriers

- `PROC initialise.barrier (BARRIER b, VAL INT count)`

  Each `BARRIER` must be initialised with this routine before starting the associated `PAR` construct. The `count` value is the number of processes in that `PAR`.

- `PROC resign.barrier (BARRIER b)`

  Each process in the associated `PAR` construct must execute this routine just before it terminates.

- `PROC synchronise.barrier (BARRIER b)`

  This may be called by any process in the associated `PAR` construct. The calling process will be blocked until all its sibling processes (in the `PAR`) have also called it or have resigned.

- `PROC enroll.barrier (BARRIER b, VAL INT count)`

  This needs to be called before and after nested `PAR` constructs, whose components are being enrolled on the `BARRIER`. Before the `PAR`, the `count` value is one less than the number of components being enrolled. After the `PAR`, the `count` is one.

4.3 Implementation of proposed barriers

Barrier synchronisation is normally intended to support physical concurrency. The syntax and semantics with which we have been experimenting are aimed at (virtual) shared-memory multi-processors. The current implementation is only for uni-processors, where barrier synchronisation is still a powerful tool for the management of processes.

Figure 2 shows the occam2.1 implementation of these barriers via the `BARRIER` primitives. Examples are given for SPMD, MIMD and for the dynamic enrollment of new processes in an existing barrier.

However, there is one loose end that needs to be nailed down! The last enrolled sub-process to resign should not really do so (as this may complete an external barrier that is not warranted). The last resignation should be folded with the subsequent re-enrollment and nothing should happen. With this prototype implementation, we don't know when resigning whether we are the last resignation. With an implementation via the run-time kernel, we will have this information and the loose end can be tied.

4.4 A simple example

The following is a complete KRoC occam2.1 program demonstrating a simple SPMD network of processes synchronising on an `BARRIER` barrier. Each process is cyclic, waiting for a varying amount of time before synchronising once per cycle. Each process is a `client` of the `SHARED screen` channel, which is protected by a `SEMAPHORE`. Each process announces to the `screen` when it tries to synchronise and when it succeeds in synchronising.

```
#INCLUDE "semaphore.inc"
#INCLUDE "barrier.inc"
#USE "utils"               -- in the course directory of the KRoC release
```

```
PROC barrier.test (CHAN OF BYTE keyboard, screen, error)

  #PRAGMA SHARED screen
  SEMAPHORE screen.s:
  #PRAGMA SHARED screen.s

  PROC client (VAL INT id, n.clients, BARRIER b,
               SEMAPHORE out.s, CHAN OF BYTE out)
    INT n:
    SEQ
      n := id
      WHILE TRUE
        SEQ
          --{{{  wait n seconds
          VAL INT seconds IS 1000000:
          TIMER tim:
          INT t:
          SEQ
            tim ? t
            tim ? AFTER t PLUS (n*seconds)
          --}}}
          --{{{  say ready to synchronise
          SEQ
            claim.semaphore (out.s)
            out.number (id, 0, out)
            out.string (" ready to synchronise*c*n", 0, out)
            release.semaphore (out.s)
          --}}}
          synchronise.barrier (b)
          --{{{  tell the world
          SEQ
            claim.semaphore (out.s)
            out.string ("==> ", 40, out)
            out.number (id, 0, out)
            out.string (" over the barrier ...*c*n", 0, out)
            release.semaphore (out.s)
          --}}}
          n := (n.clients + 1) - n    -- simple variation for the timeout
  :

  VAL INT n.clients IS 10:

  BARRIER b:
  #PRAGMA SHARED b

  SEQ
    initialise.semaphore (screen.s, 1)
    initialise.barrier (b, n.clients)
    PAR n = 0 FOR n.clients
      client (n + 1, n.clients, b, screen.s, screen)

:
```

It is hard to resist (and we don't) turning this into higher-level occam2.x:

```
#USE "utils"

PROC barrier.test (CHAN OF BYTE keyboard, SHARED CHAN OF BYTE screen, error)

  PROC client (VAL INT id, n.clients, BARRIER b, SHARED CHAN OF BYTE out)
    INT n:
    SEQ
      n := id
      WHILE TRUE
        SEQ
          ...  wait n seconds
          --{{{  say ready to synchronise
          CLAIM out
            SEQ
              out.number (id, 0, out)
              out.string (" ready to synchronise*c*n", 0, out)
          --}}}
          SYNC b
          --{{{  tell the world
          CLAIM out
            SEQ
              out.string ("==> ", 40, out)
              out.number (id, 0, out)
              out.string (" over the barrier ...*c*n", 0, out)
          --}}}
          n := (n.clients + 1) - n    -- simple variation for the timeout
  :

  VAL INT n.clients IS 10:

  PAR n = 0 FOR n.clients BARRIER b
    client (n + 1, n.clients, b, screen)

:
```

The point is not the modest reduction in code length but the absence of special `#PRAGMA`s, explicit `SEMAPHORE`s and explicit `SEMAPHORE` and `BARRIER` initialisation. It is these absences, the automatic initialisation (by the compiler) of all necessary primitives, the prevention (by the compiler) of any attempted `BARRIER` duplication via assignment or communication and the mandatory use of the `CLAIM` mechanism for the `SHARED` channel (by the compiler) that makes the high-level bindings secure and, hence, very desirable.

4.5 A shared accumulator (and implicitly parallel recursive lazy functional occam)

occam2 has a formal denotational semantics in terms of the traces, failures and divergences model of CSP [8][9]. This can be extended, in a natural way, to cover all the language extensions discussed here.

Otherwise, the state-of-the-art in parallel languages is somewhat bleak. There are no formal semantics on offer for *lightweight* threads libraries (even though they are becoming standardised) nor for Java threads (where, at least, the concept is bound into the language). Indeed, it is very difficult to find even informal descriptions – their semantics are mainly given by example.

Reference [10] describes ParC, a language extension for C giving explicit parallelism and synchronisation (with primitives not too far from those in the extended occam). In a section titled *The Meaning of Parallelism*, after analysing the issues for nearly one page, it reaches the following depressing conclusion:

> *"As a consequence, distinct execution of the same program may lead to different results, and even to different behaviours. For example, one execution may spawn many more activities than another, or one execution may terminate with a result while another enters an infinite loop. It is therefore impossible to specify the exact semantics of ParC programs. In the absence of formal semantics, we make do with a set of rules to guide the implementation of a ParC system."*

Such conclusions give no optimism for a sound engineering basis for parallel computing and raise fundamental questions as to its viability. Fortunately, the scientific insights of occam and CSP, along with those of BSP, are becoming available to a wider audience. Unfortunately, resistance to the concept of sound engineering is hard to overestimate in today's mainstream computer culture.

Anyway, the example in this section is taken from [10] but expressed now in terms of occam. The problem is to add up the elements of an array in *O(log n)* time.

4.5.1 Natural barrier solution

The first solution is expressed in only modestly upgraded occam, making use of the natural barriers marking the termination of PAR constructs:

```
PROC sum ([]INT A)              -- with implicit barrier synchronisation
  -- assume : (SIZE A) = (2**N), for some N >= 0
  -- spec : A'[0] = SIGMA {A[i] | i = 0 FOR SIZE A}
  INITIAL INT n IS SIZE A:  -- INVARIANT:
  INITIAL INT stride IS 1:  --   (n*stride) = (SIZE A)
  WHILE n > 1
    SEQ
      n, stride := n >> 1, stride << 1
      PAR i = 0 STEP stride FOR n
        A[i] := A[i] + A[i + (stride >> 1)]
:
```

The *modest* extensions are the INITIALising declarations (of occam3 and used previously), the STEP in the replicator (very convenient for numeric algorithms and straightforward to implement) and the variable number of replications in the PAR construct (no semantic but a serious implementation problem, although a unified virtual shared-memory address space simplifies matters considerably).

The algorithm is simple. In the first loop, a process is spawned for all the even elements in the array (stride is 2 and n is half the array size); this process adds into its element the value of its odd (senior) neighbour. In the second loop, a process is spawned for every fourth element (stride = 4) that accumulates the contents of their neighbour two (i.e. half-a-stride) above them; every fourth element now holds the sum of all the elements within its stride. This continues for *log(n)* cycles until stride reaches the size of the array (and n drops to 1); after which A[0] holds the complete sum and the loop terminates.

A slight drawback is that the array size must be a power of two. Other sizes could be handled but the simplicity of the code would be damaged.

Parallel security is easy to establish. Each parallel process updates element A[i], where each i is different (so no race-hazard there). The process updating A[i] uses the value in an element half-a-stride away. But no other process is looking at these half-stride values since they are all separated by a stride (so no race-hazard there). *QED*. Getting such proofs checked mechanically (e.g. by the compiler) looks possible and will ultimately be necessary.

Of course, the fine granularity of the parallelism in the above example would scuttle any hoped-for performance gain from current parallel architectures, although future designs (such as the ParaPC [11][12]) could lap it up. In the meantime, the example serves as a *model* for combining operations that are computationally more intensive than addition and that current architectures may be able to exploit.

4.5.2 Explicit barrier solution

The second solution commutes the PAR and the WHILE constructs, catering for those who fear the costs of starting up and shutting down processes. The result is a conventional SPMD algorithm with explicit barrier synchronisation, which occam can now comfortably express:

```
PROC sum ([]INT A)              -- with explicit barrier synchronisation
  -- assume : (SIZE A) = (2**N), for some N >= 0
  -- spec : A'[0] = SIGMA {A[i] | i = 0 FOR SIZE A}
  PAR i = 0 FOR SIZE A BARRIER b
    INT accumulate IS A[i]:
    INITIAL INT n IS (i = 0) -> SIZE A, i:
    INITIAL INT stride IS 1:
    WHILE (n /\ 1) = 0          -- even (n)
      SEQ
        accumulate := accumulate + A[i + stride]
        n, stride := n >> 1, stride << 1
        SYNC b
:
```

We have sneaked conditional expressions into the language (since they simplify one of the initialising declarations above). The syntax used is just that already implemented for the occam configuration language in the SGS-Thomson Toolset:

boolean-expression `->` *expression*`,` *expression*

This simply yields the first or second *expression*, depending on the value of the *boolean-expression*. Both *expression*s must, of course, yield the same type. Note that occam2 already can express the above:

`[`*expression*`,` *expression*`][INT` *boolean-expression*`]`

although it is not quite so understandable! The order of the *expression*s must be reversed since `FALSE` and `TRUE` map (under `INT`) to 0 and 1 respectively. Also, the current implementation will evaluate both *expression*s at run-time before discarding the unselected one (unless *boolean-expression* is constant).

The conditional expression is a distraction ... forget them! Please compare the new version of `sum` with the old. This time processes are set up once for each element of the array (abbreviated locally to `accumulate`). The odd processes immediately terminate. That leaves the even processes adding their odd (senior) neighbours to themselves, exactly as before. At the end of each loop, the active processes synchronise on the barrier and half of them drop out. Recall that that does not prevent the remaining processes synchronising on their next loop. Events continue until there is only one process left, which accumulates the final answer into `A[0]` and terminates. There are now no processes left and the outer `PAR` construct terminates and the `PROC` returns.

Security against race-hazards is somewhat harder to prove than before. An elegant way to establish this would be to find some semantic-preserving transformations (that can be mechanised) to change the first version into the second. This is left as an exercise for the reader.

One optimising transformation on the second version that is too tempting to resist is as follows. Since the odd processes terminate without ever doing anything, don't set them up in the first place! This is achieved simply by changing the `PAR` constructor:

```
PROC sum ([]INT A)              -- with explicit barrier synchronisation
  ...  same specification
  PAR i = 0 STEP 2 FOR (SIZE A) >> 1 BARRIER b
    ...  same replicated process (but only half as many of them!)
:
```

4.5.3 Recursive solution with only local synchronisations

Finally, for those who find barrier synchronisation a little unnatural, here is a taste of some much wilder ideas. The following code also sums its array in *O(log n)* time but:

- is (first order) functional, relying on the compiler and run-time system to extract the parallelism that is always implicit in occam expressions (which are free from side-effects and can, therefore, be executed in any order or concurrently);

- is recursive – however, implementation techniques that enable variable PAR replication also enable recursion;
- has no global synchronisations, only local synchronisations implied by (add) operators requiring the (two) processes computing their operands to terminate before they can operate;
- is efficient in that the (parallel) execution tree for small array fragments has been preset by standard loop unrolling – however, the implementation of table lookup at run-time needs to be made *lazy* for the way we have chosen to express the unravelled loop to work sensibly;
- has automatic parallel security, derived from the semantics of occam expressions;
- handles arrays of any size – not just powers of two.

It's also pretty neat:

```
INT FUNCTION sum (VAL []INT A) IS

  -- spec : returns SIGMA {A[i] | i = 0 FOR SIZE A}

  (SIZE A) <= 8) ->

    [A[0],
     A[0] + A[1],
     A[0] + (A[1] + A[2]),
     (A[0] + A[1]) + (A[2] + A[3]),
     (A[0] + A[1]) + (A[2] + (A[3] + A[4])),
     (A[0] + (A[1] + A[2])) + (A[3] + (A[4] + A[5])),
     (A[0] + (A[1] + A[2])) + ((A[3] + A[4]) + (A[5] + A[6])),
     ((A[0] + A[1]) + (A[2] + A[3])) + ((A[4] + A[5]) + (A[6] + A[7]))]
     [(SIZE A) - 1],

      sum ([A FOR (SIZE A) >> 1]) + sum ([A FROM (SIZE A) >> 1 ])]:
```

Now, all we need is a ParaPC on which to run it.

5 Implementation overview and platform independence

The new primitives have been introduced as abstract data types and with no changes to the KRoC kernel. This means they will automatically run on any KRoC system.

The routines operating on the primitives are programmed as INLINE PROCs and use transputer ASM blocks. This means that, with certain restrictions, they will also run on real transputers (using standard occam Toolsets).

5.1 SEMAPHOREs

A SEMAPHORE is an occam2.1 RECORD with three fields: one holds a count and the others hold front and back pointers to a process queue. Processes are held on this queue using the same workspace link fields that hold them on the run-queue (a process can never be on a SEMAPHORE-queue and the run-queue at the same time). This means that no space needs to be reserved to manage this queue (other than the front and back pointers).

A process claiming a SEMAPHORE will be put on its queue (and blocked) if its count is zero. Otherwise the count is decremented.

A process releasing a SEMAPHORE will re-schedule the first process from its queue if that queue is not null. Otherwise, it increments the count.

Both these operations work in constant time.

5.1.1 Transputer restrictions

Scheduling of processes on and off the run-queue is managed using the normal transputer scheduling instructions – run-queue registers are not modified directly. This means they will be secure on a transputer even in the presence of high-priority process pre-emption (caused by transputer link, event or timer interrupts).

However, manipulation of the SEMAPHORE queues themselves (or their counters) can be corrupted by process pre-emption. No danger arises if all processes sharing the same SEMAPHORE also share the same transputer priority. Otherwise, the low-priority processes must protect their claims and releases by first popping into high-priority – for example:

```
PRI PAR
  claim.semaphore (s)
  SKIP
```

T9000 and ST20-derived transputers have extended instruction sets that include SEMAPHORE operations providing claim.semaphore and release.semaphore semantics (called signal and wait). If asked, we can provide a version of the SEMAPHORE abstract data type that exploits them (and the above work-around will not be necessary).

5.2 BARRIERs

An BARRIER is an occam2.1 RECORD with four fields: two integer counts (for the number of processes registered and the number that have not yet synchronised) and the front and back pointers to a process queue.

A process synchronising on an BARRIER decrements its synchronisation count. If this has not reached zero, it attaches itself to the BARRIER-queue and blocks. Otherwise, it releases *all* the processes on the BARRIER-queue (by simply concatenating it on to the run-queue) and resets the synchronisation count back to the number currently registered. This is a constant time operation.

A process resigning an BARRIER decrements both its registration count and its synchronisation count. If the latter has reached zero, it releases *all* the processes on the BARRIER-queue (just like a synchronising process) and resets the synchronisation count. Again, this is a constant time operation.

A process enrolling on an BARRIER just increments the registration and synchronisation counts equally. No rescheduling of processes takes place.

5.2.1 Transputer restrictions

Transputers have no atomic (non-preemptable) instructions for concatenating a process queue on to a run-queue. Therefore, to implement `BARRIER` operations, we have been updating run-queue registers through a sequence of instructions. Transputer link, event and timer interrupts may preempt this sequence and try to schedule processes on to the same run-queue – with bad consequences!

Therefore, our `BARRIER`s may be used safely on transputers provided:

- only low-priority processes use `BARRIER`s and they protect `synchronise.barrier` and `resign.barrier` operations by popping into high-priority;
- only high-priority processes handle transputer link, event or timer interrupts. [NB: the term *transputer event* refers to the electronic assertion of its event pin by some external device, which has nothing to do with the `BARRIER` primitive in this document.]

These are not severe restrictions – the second point above should properly be a design rule in any case. Of course, if the application had no need for interrupt handling (which implies that it is uni-processor), the first point can be ignored (unless, of course, the `BARRIER` is shared between high and low priority processes).

6 Discussion

6.1 Summary

This document has described some new synchronisation primitives for occam and some higher-level language bindings that make them secure. The new primitives are directly usable within occam2.1 through the abstract data types released from KRoC 0.8beta onwards. The primitives complement, but do not replace, the traditional concept of channel communication for networks of synchronising processes. In particular, they provide an implementation for `SHARED` channels (as proposed for occam3), as well as a range of higher-level and relaxed forms of safely `SHARED` resource that will allow more parallelism to be extracted from applications. Details of these higher-level mechanisms for sharing will be reported separately.

6.2 Performance and optimisation

The released primitives have not yet been benchmarked, but we believe that none of the operations cost more in time than about two or three context-switches (i.e. between one and two micro-seconds on a SPARC-20). For portability, the prototypes have not been burnt into the KRoC kernel, but have been implemented with transputer instructions (which KRoC uses as an abstract intermediate code). It is straightforward to move the implementation of the primitives into the KRoC kernel and this will be done later for those that prove useful. This should reduce the overheads for each operation towards a single context-switch. It will also nail down the loose end currently exposing a minor security problem in `BARRIER`s (when the number of processes engaged in a barrier synchronisation grows temporarily).

6.3 Virtual transputers

Working with virtual transputers has its benefits over working with real ones: modern micro-processors mean they run faster and we *can* experiment with new instructions with relative ease. For example, to move the primitives into the kernel, the abstract (or virtual) transputer machine used by KRoC will have to extended with new instructions for the manipulation of process queues. We want to do this anyway for other reasons – such as a full implementation of `PRI PAR` (that allows any number of prioritised components).

When KRoC goes multi-priority and multi-processor, we want no constraints on our use of the primitives (such as are necessary with the existing microcode on real transputers). With control over the virtual architecture, this should be possible (and inexpensive) to arrange.

The long-term (medium-term?) architectural goal, for which the new primitives would give major benefit, is shared-memory (real or virtual) multi-processing. To support this further, we are investigating a richer form of `BARRIER` that implements the contradictory-sounding *non-blocking barrier synchronisation*, recently proposed for MPI-2. This is a two-phase barrier synchronisation, where the first phase registers that the process is ready to synchronise (but doesn't block) and the second phase does block (unless and until all the other processes in the barrier have registered their first phase). With the right algorithm, processes may be able to sail through most such barriers without ever blocking!

We also want to extend the virtual instruction set to provide type information that will allow KRoC to target Java Byte Code. Additionally, this type information makes possible a Transputer Byte Code *verifier* that enables the distribution of occam processes as compact binaries (*occlets*) with the same (or better) object-level security as Java. Note that KRoC translates these Byte Codes to target object code for execution – it doesn't interpret them at run-time. In that sense, it already provides *just-in-time* compilation (but we do realise there is a little bit more involved than that).

6.4 Microcode, methods and objects

The algorithms underlying the new synchronisation primitives correspond to new *micro-code* for the virtual transputer. Just like the real micro-code that implements channel synchronisation on real transputers, great care has to be taken in its design – these algorithms are significantly harder to get right than algorithms of a similar size at the occam application level.

The reason for this is that algorithms within occam processes are naturally object-oriented – in the *literal* sense of the term. By this, we mean that they directly express the behaviour of objects from their own point of view – not as a set of procedure calls that are made by (and, therefore, oriented towards) external agents. This is achieved through implementing objects as *active* processes that run concurrently with other objects, each with their own thread (or threads) of control. Of course, this is something for which occam was specifically designed.

Most *object-oriented* programming languages allow the encapsulation of object data and algorithms, but only provide for the expression of those algorithms through a set of *passive* methods (which are no different to procedures). Objects interact by calling each other's methods and, so far as algorithm design is concerned, there is no paradigm shift from traditional procedural programming.

The problem is that expressing the behaviour of an object through a set of externally-called methods is unnatural (it's literally not object-oriented!), and it gets especially hard in a multi-threaded (or multi-processing) environment. The reason is that the semantics of concurrently operated methods (even the `synchronized` methods of Java) do not compose. This means that in order to design/understand two methods of some object, we have to design/understand both at the same time. Of course, this gets worse the more methods an object contains and puts a limit on the complexity of system that can be designed in this way.

Unfortunately, we cannot build a system purely from active objects! An active object cannot directly interfere with another active object (this would break the principle of data abstraction and introduce all manner of semantic chaos) and it cannot directly call on it (because active objects are active and don't provide passive facilities). So, we need some passive medium through which they can interact.

Fortunately, we only need a small variety of passive objects to construct this medium and these can be hidden in *micro-code* and/or burnt into a high-level language. Then, the system engineer only needs to work with active (truly object-oriented) objects, whose semantics do compose and put no limit on the complexity of the design.

Hence, we have occam and the virtual transputer, where the necessary (but hard to program) passive objects are its `CHAN`nels, `SEMAPHORE`s, `RESOURCE`s, `BARRIER`s and `BUCKET`s and, hopefully, not too many more!

The most complex in this list is the `RESOURCE` [4]. The non-compositional nature of the semantics of its implementation bites as we are obliged to think simultaneously about its `claim` and `release` algorithms – they cannot be understood individually. The same is true for all the other primitives, including the original occam `CHAN`nel whose *input* and *output* methods (especially in the context of `ALT`s) are so elegant, but are completely interdependent. It is an immense relief we don't have to understand this, or program like this, at the occam level.

6.5 Java threads and occam

These ideas were examined in the context of Java at the Java Threads Workshop, which took place at the University of Kent last September (1996). Java allows both passive and active objects at the user-program level, with passive being the default mechanism everyone learns first. Java threads are not based upon CSP, but on the earlier concept of *monitors*. Thread synchronisation can only be achieved through calling `synchronized` methods. These methods belong to passive objects and have to be programmed – which is hard.

The workshop has stimulated the development (by at least three research groups) of CSP class libraries that package a range of passive hard-to-program objects (like channels, shared channels, buffers, shared buffers, barriers and buckets). Multi-threaded Java applications can now be developed that interact with each other in the occam/CSP style and use only active easy-to-program objects. Details from this workshop can be found on:

`<URL:http://www.hensa.ac.uk/parallel/groups/wotug/java/>`
`<URL:ftp://unix.hensa.ac.uk/pub/parallel/groups/wotug/java/>`

6.6 If only ...

These ideas could have been introduced at low cost over ten years ago – they do not require any special hardware technology. They remain vital today because they enhance any multi-processing (or multi-threaded) technology that doesn't have them – and that seems to include most everything. Now if only we had had them since 1985, ...

References

[1] P.H. Welch and D.C. Wood. KRoC – the Kent Retargetable occam Compiler. In B. O'Neill, editor, *Proceedings of WoTUG 19*, Amsterdam, March 1996. WoTUG, IOS Press. ISBN 90-5199-261-0.

[2] M.D. Poole. *occam for all*: Two approaches to retargeting the INMOS occam compiler. In B. O'Neill, editor, *Proceedings of WoTUG 19*, Amsterdam, March 1996. WoTUG, IOS Press. ISBN 90-5199-261-0.

[3] *occam-for-all* group. Beta-release WWW site for KRoC. <URL:http:// www.hensa.ac.uk/parallel/occam/projects/occam-for-all/kroc/>.

[4] P.H. Welch and D.C. Wood. SEMAPHOREs, RESOURCEs, EVENTs and BUCKETs (documentation). <URL:http:// www.hensa.ac.uk/ parallel/ occam/ projects/ occam-for-all/ hlps/>.

[5] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[6] SGS-Thomson Microelectronics Ltd. occam2.1 reference manual, 1995. <URL:http://www.hensa.ac.uk/parallel/occam/documents/> (printed version available from `M.D.Poole@ukc.ac.uk`).

[7] Geoff Barrett. occam3 reference manual (draft), March 1992. <URL:http:// www.hensa.ac.uk/parallel/occam/documents/>.

[8] M.H. Goldsmith, A.W. Roscoe, and B.G.O. Scott. Denotational semantics for occam2 (part 1). *Transputer Communications*, 1(2):65–91, John Wiley and Sons Ltd., November 1993. ISSN 1070-454X.

[9] M.H. Goldsmith, A.W. Roscoe, and B.G.O. Scott. Denotational semantics for occam2 (part 2). *Transputer Communications*, 2(1):25–67, John Wiley and Sons Ltd., March 1994. ISSN 1070-454X.

[10] Y.Ben-Asher, D.G.Feitelson, and L.Rudolf. ParC – an extension of C for shared memory parallel processing. *Software – Practice and Experience*, 26(5):581–612, John Wiley and Sons Ltd., May 1996. ISSN 0038-0644.

[11] P.H. Welch. Parallel hardware and parallel software – a reconciliation. In *Proceedings of the ZEUS'95 and NTUG'95 Conference, Linkoping, Sweden*, pages 287–301, Amsterdam, May 1995. IOS Press. ISBN 90-5199-22-7.

[12] B. Cook and R. Peel. The Para-PC, an analysis. In B. O'Neill, editor, *Proceedings of WoTUG 19*, pages 89–102, Amsterdam, March 1996. WoTUG, IOS Press. ISBN 90-5199-261-0.