Occam the ideal language ?

Occam:

- Designed for MIMD Message Passing machines.
  - Described as machine code for the Transputer.
- Includes low level support for concurrent processes.
- Includes built in support for message passing.

Although Occam itself contains the elements of an ideal language for portable concurrent programming with communicating processes, its implementation on Transputers falls short of the ideal.

Let us look further at Occam.

Primitive Processes

All programs are built from the following primitive processes:

- Assignment (assign expression e to variable v)

```
v := e
```

- Input (assign a value to variable v from channel c)

```
c ? v
```

- Output (output expression e via channel c)

```
c ! e
```

- No Operation (do nothing and then terminate)

```
SKIP
```

- Error (do nothing and then don't terminate)

```
STOP
```

Process Constructions

- Sequence (standard sequential construction - note indentation)

```
SEQ
  P1
  P2
  P3
  ...

SEQ
  c1 ? x
  x := x + 1
  c2 ! x
```

- Parallel (low level support for parallelism)

```
PAR
  P1
  P2
  ...

PAR
  c1 ? x
  c1 ! y * z
```

- Loop (standard sequential loop structure)

```
WHILE condition
  P

WHILE x <= 256
  SEQ
    c ! x
    x := x * 2
```

- Replication - Sequence (repeat n times in sequence for different i)

```
SEQ i = 0 FOR n
  P

SEQ i = 1 FOR 15
  A[i] := A[i-1] + i
```

- Replication - Parallel (replicate n times in parallel for different i)

```
PAR i = 0 FOR n
  P

PAR i = 0 FOR 16
  A[i] := B[i] + C[i]
```

- Conditional (executes at most one process)

```
IF
  condition1
    P1
  condition2
    P2
  ...

IF
  x > 255
    x := 255
  x < 0
    x := 0
  TRUE
    SKIP
```

- **Selection** (executes at most one process)

```
CASE expression1
  expression2
    P1
  expression3,expression4
    P2
  ...

CASE day
  saturday, sunday
    play()
  tuesday, wednesday, thursday, friday
    work()
```

- Alternation (executes at most one process)

```
ALT
  c1 ? v1
    P1
  c2 ? v2
    P2
  ...

WHILE TRUE
  ALT
    c1 ? x
      c3 ! x
    c2 ? x
      c3 ! x
```

Occam Channels

Occam channels provide unbuffered unidirectional point to point communication of values between two concurrent processes.

- Declaration

```
CHAN OF protocol channel :

CHAN OF BYTE screen :
PAR
  screen !  A

PROTOCOL packet IS INT16; INT16::[]BYTE :
CHAN OF packet link :
INT16 address, length :
[256]BYTE buffer :
PAR
  link ? address;length::buffer
```

Occam Channels

Occam channels provide unbuffered ...communication ...

- Unbuffered Communication
  - As communication is unbuffered it is also synchronized. Communication cannot take place until both processes are ready.
  - Thus we can use the communication of dummy values for interprocess synchronization.
- Buffered Communication
  - We can provide buffered communication by explicitly including a buffer process. We are then forced to consider the required buffer size.
  - N.B. Buffered communication can simulate unbuffered communication by forcing an acknowledgment after each item is transferred.

Occam Channels

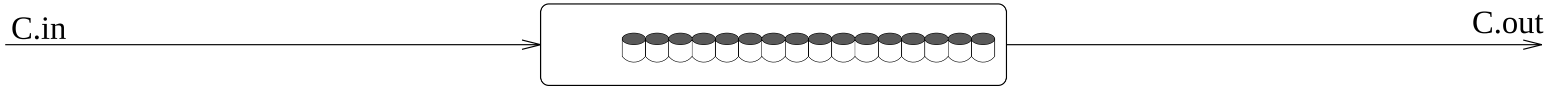
Occam channels provide . . . unidirectional point to point communication . . .

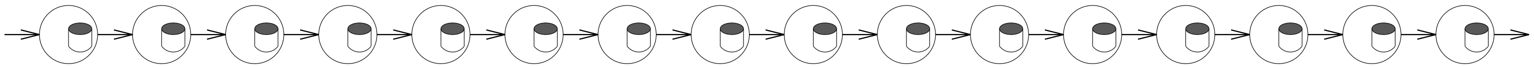
- Uni-directional Communication
  - Only one of the two communicating processes may write to a channel and only one may read from it. Bi-directional connections are constructed from two channels.
- Point to Point Communication
  - Broadcast (one to many) communication

    This can be achieved with processes which output the same data over more than one channel.
  - Many to one communication

    This is achieved with a multiplexor process which accepts data from more than one channel.

Occam Buffer Process





```
PAR
  WHILE TRUE
    SEQ
      c.in ? x[0]
      c[0] ! x[0]
  WHILE TRUE
    SEQ
      c[14] ? x[15]
      c.out ! x[15]
  PAR i = 1 FOR 14
    WHILE TRUE
      SEQ
        c[i-1] ? x[i]
        c[i]   ! x[i]
```

Occam Granularity

- Occam has been designed in order to encourage programs using large amounts of concurrency.
- Thus the *grain* size of a concurrent process is small.
- We must match this with a low overhead for initiating concurrent processes.

  We would like for the execution time of:

```
PAR
  A := B + C
  D := E + F
```

  to be comparable to the execution time of:

```
SEQ
  A := B + C
  D := E + F
```