Replicators
(components and test-rigs)

Peter Welch (p.h.welch@kent.ac.uk)
Computing Laboratory, University of Kent at Canterbury

Co631 (Concurrency)
Replicators *(components and test-rigs)*

Replicated **PAR** and **SEQ** ...

The **SORT PUMP** ...

Component testing ...

Stateless components ...

The **SORT GRID** ...

Replicated **IF** ...

Replicator **STEP** sizes ...
Consider a process with an array of input channels:

\[
\begin{array}{cccc}
\text{in}[0] & \text{in}[1] & \text{in}[2] & \text{in}[3] \\
\end{array}
\]

And an internal data array, \( x \), of the same type and size as the input channel array.

The process needs to input one message from each input channel into the corresponding element of its data array.
These inputs are to be done in parallel:

\[
\begin{array}{c}
\text{in[0]} & \text{in[1]} & \text{in[2]} & \text{in[3]} \\
\top & \top & \top & \top \\
\hline
\top & \top & \top & \top \\
0 & 1 & 2 & 3 \\
\end{array}
\]

**Golden Rule:** When communications can be done in parallel, do them in parallel.
These inputs are to be done *in parallel*:

```
in[0]  \rightarrow  x[0]
in[1]  \rightarrow  x[1]
in[3]  \rightarrow  x[3]
```

But what if there were 40 channels in the array? Or 400 ... or 4000 ... ??!
These inputs are to be done *in parallel*:

```c
INT declaration
PAR i = 0 FOR 4
in[i] ? x[i]
```

This process gets replicated.
These inputs are to be done *in parallel*:

\[
\begin{align*}
\text{PAR } & i = 0 \text{ FOR 4} \\
i[n] & \equiv x[i]
\end{align*}
\]
Just in case they really had to be done *in sequence*:

\[
\begin{array}{c}
\text{INT declaration} \\
\text{SEQ } i = 0 \text{ FOR } 4 \\
in[i] \text{ ? } x[i]
\end{array}
\]

This process gets replicated.
Just in case they really had to be done *in sequence*:

\[
\begin{array}{c}
\text{SEQ } i = 0 \text{ FOR 4} \\
\text{in}[i] \rightarrow x[i]
\end{array}
\]
The replicated **SEQ** is like a very clean **for**-loop.

**INT declaration**

**first value**

**SEQ** i = start **FOR** count

<process i>

**In Java or C:**

```java
for (int i = start; i < (start + count); i++) {
    <code i>
}
```

**Must not change the value of i, start or count**
The replicated **PAR** has no correspondence in Java or C.

```plaintext
INT declaration
first value
number of replications
PAR i = start FOR count
<process i>
```

**In Java or C:**

... **silence**
Applying the replicated PAR.

The first example showed parallel replication of a *primitive* process (an input process).

But, earlier, we’ve seen parallel composition of long-lived *structured* processes (like continuously active ‘chips’).

The next example shows parallel replication of such a process to build a *parallel sorting engine*. 
Replicators *(components and test-rigs)*

- Replicated **PAR** and **SEQ** ...
- The **SORT PUMP** ...
- Component testing ...
- Stateless components ...
- The **SORT GRID** ...
- Replicated **IF** ...
- Replicator **STEP** sizes ...
A diagram showing a sorting process labeled `sort.pump`. The process takes unsorted items (grouped) and outputs sorted items (grouped). The unsorted items include numbers such as 8, 99, 42, 4, 6, 21, 55, 7, etc., while the sorted items include 4, 6, 7, 8, 21, 42, 55, 99, etc. The diagram demonstrates the transformation from unsorted to sorted.
sort.pump

in

8, 99, 42, 4, 6, 21, 55, 7, 255, 65, 92, 32, 54, 255, 111, 10, 5, 60, 48, 69, 7, 255, ...

4, 6, 7, 8, 21, 42, 55, 99, 255, 32, 54, 65, 92, 255, 5, 7, 10, 48, 60, 69, 111, 255, ...

out
sort.pump

in

8, 99, 42, 4, 6, 21, 55, 7, 255, 65, 92, 32, 54, 255, 111, 10, 5, 60, 48, 69, 7, 255...

out

4, 6, 7, 8, 21, 42, 55, 99, 255, 32, 54, 65, 92, 255, 5, 7, 10, 48, 60, 69, 111, 255...

Note: 255 is used here to mark the end of each group.
For the efficient application of this device, we need a long-running source of groups of items that need sorting. We also need to specify an upper limit on the size of groups.
An example is a simple image smoothing filter: each pixel is replaced by the median value of its (9) neighbours. Finding median values implies sorting. Each n-by-m image generates (n*m) groups of 9 numbers for sorting.
The sort.pump is implemented as a pipeline of simpler cell processes. (We’ll see what they do presently.)

To sort groups up to a maximum size of $\max$, we need at least $(\max - 1)$ cells.

So, if $\max$ is 16, we need 15 cells … which means we need 14 internal channels … which we have indexed above from 0 through 13.
VAL INT max IS 16:

PROC sort.pump (CHAN BYTE in?, out!)
[max-2]CHAN BYTE c:
PAR
cell (in?, c[0]!)
PAR p = 1 FOR max-3
  cell (c[p-1]?, c[p]!)
cell (c[max-3]?, out!)
:
VAL INT max IS 16:

PROC sort.pump (CHAN BYTE in?, out!)
[max-2]CHAN BYTE c:
PAR
  cell (in?, c[0]!)
  PARA P = 1 FOR max-3
    cell (c[p-1]?, c[p]!)
  cell (c[max-3]?, out!)
  :

So, we have (max-1) cells altogether.
So, we can sort groups up to size max.
All each cell has to do is drag heavy items backwards. In particular, as each group flows through, the last one out must be the heaviest in the group.

To do this, two variables (or registers) are needed: one to hold the largest item seen so far and one to hold the next item to arrive.
The **cell** inputs the first item of a group into **largest**.

Then, it compares each **next** item against **largest**, outputting the smaller and keeping the larger.

When the **end.marker** arrives, it just outputs the **largest** followed by that **end.marker**.
VAL BYTE end.marker IS 255:    -- assume > data items

PROC cell (CHAN BYTE in?, out!)
  WHILE TRUE
    BYTE largest:
    SEQ
      in ? largest
      WHILE largest <> end.marker
        BYTE next:
        SEQ
          in ? next
          IF next
            -- output smaller, keep larger
            largest >= next
            out ! next
            TRUE        -- i.e. largest < next
            SEQ
              out ! largest
              largest := next
          ELSE
            out ! end.marker
  :
VAL BYTE end.marker IS 255: -- assume > data items

PROC cell (CHAN BYTE in?, out!)
  WHILE TRUE
    BYTE largest:
    SEQ
      in ? largest
      WHILE largest <> end.marker
        BYTE next:
        SEQ
          in ? next
          IF largest >= next
            out ! next
            TRUE -- i.e. largest < next
              SEQ
                out ! largest
                largest := next
          out ! end.marker
  : 

Note: this algorithm requires a potential data item (255) reserved for the end.marker. This constraint can be removed – later.
VAL INT max IS 16:

Each **cell** holds back largest item it sees, so ...

max items injected here

(max-1) lightest output first

(max-2) lightest output first

lightest output first
VAL INT max IS 16:

As the end.marker flows through, it pushes out the heaviest item, which pushes out the next heaviest, etc…
VAL INT max IS 16:

The group, therefore, flows out in ascending sorted order. 😊
VAL INT max IS 16:

If the cells are implemented on separate pieces of silicon (i.e. *we have a physically parallel engine*), the speed at which data flows through is the *slowest* of:

- the speed at which data is offered;
- the cycle speed for each cell;
- the inter-cell communication speed.

The speed is independent of the number of cells – which means that it is independent of the number of items being sorted. We have an *O*(n) sorting engine: *sort.pump*. 😊 😊 😊
VAL INT max IS 16:

In fact, \texttt{sort.pump} is a parallel version of \texttt{bubble-sort}, one of the simplest known sorting algorithms. Its performance on a \textit{serial} processor is $O(n^2)$, which is poor compared to more complex sorts (such as \texttt{quick-sort}, which is $O(n \cdot \log(n))$).

If data is supplied in $O(n)$ time (as in the above, where the numbers are supplied \textit{one-at-a-time}), then a processing complexity of $O(n)$ cannot be beat!

\textbf{Lesson:} when considering a \textit{parallel} design, don’t start from the most efficient known \textit{serial} algorithm – it’s probably optimised the wrong way. \textit{Rethink} – look for the simplest approach.
Note: the capacity of \textit{sort.pump} is \((2*\text{max} - 2)\) items, each \textit{cell} holding 2 of them.

So, \textit{sort.pump} \textit{can be} processing (parts of) two or three groups (up to \textit{max} size) at the same time.

It will only operate efficiently so long as there is a continuous supply of groups to be sorted.

For example, if only one group were pushed through, only half the \textit{cells} would ever be operating at the same time.
Replicators (components and test-rigs)

Replicated PAR and SEQ ...

The SORT PUMP ...

Component testing ...

Stateless components ...

The SORT GRID ...

Replicated IF ...

Replicator STEP sizes ...
1) Place component (e.g. `sort.pump`) on `bench`.
Component Testing

1) Place component (e.g. \texttt{sort.pump}) on \texttt{bench}.

2) Design \texttt{test.rig} through which we can interact meaningfully with component.
1) Place component (e.g. `sort.pump`) on `bench`.

2) Design `test.rig` through which we can interact meaningfully with component.

3) Wire it up and start experimenting …
1) Place component (e.g. `sort.pump`) on `bench`.

2) Design `test.rig` through which we can interact meaningfully with component.

3) Wire it up and start experimenting …
Typical Test-Rig Design

probe.in → probe.out


keyboard.manage → a

test.rig
This process filters keyboard input for ‘bad’ characters (e.g. control-chars, carriage-return), issuing an error report for any found, and compresses/encodes ‘good’ characters (e.g. visible-chars) for onward transmission.
Typical Test-Rig Design

probe.out

window (16, 2)

probe.in

window (48, 2)

b

m[1]

m[2]

m[3]

a

test.rig

m[0]

keyboard

history

screen.plex

keyboard.manage

screen
This process multiplexes an array of input streams to a single output stream.
Typical Test-Rig Design

probe.out

window (16, 2)

probe.in

window (48, 2)

b

m[1]

m[2]

m[3]

history

test.rig

a

m[0]

keyboard.

manage

screen.

plex

keyboard

screen
This process is a fixed-size delay line. Each item input pushes one item out. It holds the last \( \max \) items received. Every cycle, it dumps its entire holding array (with screen position control-chars). This lets us see what’s in the data stream.

\( (x, y) \) specifies coordinates defining the start position on the screen for the dump items.
Typical Test-Rig Design

probe.out

window (16, 2) → a → m[1] → history → m[2] → screen.plex

keyboard.manage

keyboard

screen

probe.in
This process lays out a *history* of the items received. It uses the bottom two-thirds of the screen.
Design Guidelines

- Don’t try to cram too much functionality into any process: **One function ⇔ One process**
- **Multiple functions ⇔ Multiple processes**
- Each process is programmed from its own point-of-view. Think of each process as an independent **serial** program, with a variety of input and output channels.
- **Concurrency then makes design simple!** ☺ ☺ ☺
- Try to build that **test.rig** as a **single serial** process and we will get a mess … ☹ ☹ ☹
PROC bench (CHAN BYTE keyboard?, screen!, error!)
CHAN BYTE a, b:
PAR
  sort.pump (a?, b!)
  test.rig (keyboard?, screen!, a!, b?)
:
PROC test.rig (CHAN BYTE keyboard?, screen!, probe.out!, probe.in?)
CHAN BYTE a, b:
[4]CHAN BYTE m:
PAR
keyboard.manage (keyboard?, a!, m[0]!)
window (16, 2, a?, probe.out!, m[1]!) -- (16, 2) => top-left
window (48, 2, probe.in?, b!, m[2]!) -- (48, 2) => top-right
history (b?, m[3]!)
screen.plex (m?, out!)
But ... what if we want to see what's going on inside the sort.pump?
As things stand, we can’t see inside the cell processes in the pump.
We need to wire up the cells to report their changing states.
VAL INT max IS 16:

PROC sort.pump (CHAN BYTE in?, out)

[max-2]CHAN BYTE c:
PAR
    cell (in?, c[0]!)
    PAR p = 1 FOR max-3
    cell (c[p-1]?, c[p]!)
    cell (c[max-3]?, out!)
:


VAL INT max IS 16:

PROC sort.inside (CHAN BYTE in?, out!,
               []CHAN BYTE report!)

[max-2]CHAN BYTE c:
PAR
 reporting.cell (in?, report[0]!, c[0]!)
PAR p = 1 FOR max-3
 reporting.cell (c[p-1]?, report[i]!, c[p]!)
 reporting.cell (c[max-3]?, report[max-3]!, out!)
VAL BYTE end.marker IS 255:  -- assume > data items

PROC cell (CHAN BYTE in?, out!)
  WHILE TRUE
    BYTE largest:
    SEQ
      in ? largest
      WHILE largest <> end.marker
        BYTE next:
        SEQ
          in ? next
          IF
            -- output smaller, keep larger
            largest >= next
            out ! next
            TRUE    -- i.e. largest < next
              SEQ
                out ! largest
                largest := next
              out ! end.marker
          :)
VAL BYTE end.marker IS 255: -- assume > data items

PROC reporting.cell (CHAN BYTE in?, report!, out!)
  WHILE TRUE
    BYTE largest:
    SEQ
      ... report ! '~'; '~'
      in ? largest
      ... report ! '~'; largest
    WHILE largest <> end.marker
      BYTE next:
      SEQ
        in ? next
        ... report ! next; largest
      IF -- output smaller, keep larger
        largest >= next
        out ! next
      TRUE -- i.e. largest < next
        SEQ
          out ! largest
          largest := next
        ... report ! '~'; largest
      out ! end.marker
PROC bench (CHAN BYTE keyboard?, screen!, error!)  
CHAN BYTE a, b:  
[max-1]CHAN BYTE report:  
PAR  
sort.pump (a?, report!, b!)  
test.rig (keyboard?, screen!, a!, report?, b?)  
:
Replicators *(components and test-rigs)*

Replicated **PAR** and **SEQ** ...

The **SORT PUMP** ...

Component testing ...

Stateless components ...

The **SORT GRID** ...

Replicated **IF** ...

Replicator **STEP** sizes ...
Let's simplify the logic within a cell process ...

Val BYTE end.marker IS 255:  -- assume > data items

Proc cell (chan byte in?, out!)
  WHILE TRUE
    byte largest:
      seq
        in ? largest
        WHILE largest <> end.marker
          byte next:
            seq
              in ? next
              IF -- output smaller, keep larger
                largest >= next
                out ! next
                true -- i.e. largest < next
                seq
                  out ! largest
                  largest := next
              out ! end.marker
        ;
  ;
Here is the serial logic (a loop within a loop).
Let’s simplify the logic within a cell process …

Here is the parallel logic …
Let’s simplify the logic within a cell process …

VAL BYTE hi IS 255:  -- assume > data items
VAL BYTE lo IS 0:  -- assume < data items

Here is the parallel logic …

The largest (so far) is trapped in the feedback loop.
This process copies data through, substituting \texttt{a} for \texttt{b} ...

\begin{verbatim}
PROC substitute (VAL BYTE a, b, CHAN BYTE in?, out!)
  WHILE TRUE
    BYTE x:
    SEQ
      in ? x
      IF
        x = a
        out ! b
        TRUE
        out ! x
    :
  
\end{verbatim}
And finally, let's simplify the logic within a **cell** process ... 

The **largest** (so far) is trapped in the **feedback loop**.

Here is the **parallel** logic ...
This is a primitive comparator …

PROC greater (CHAN BYTE in.0?, in.1?, small!, large!)
  WHILE TRUE
    BYTE x.0, x.1:
    SEQ
      PAR
        in.0 ? x.0
        in.1 ? x.1
      IF
        x.0 < x.1
        PAR
          small ! x.0
          large ! x.1
      TRUE
        PAR
          small ! x.1
          large ! x.0
  :

Hence, the asymmetric design of its icon.

Note: \texttt{gt} is symmetric on its input channels, but not on its output channels!
Stateless Components

All the primitive process components in the ‘Legoland’ catalogue (\texttt{id, succ, plus, delta, prefix, tail, …}) plus the ones just presented (\texttt{substitute, greater}) are \textit{stateless}.

This means they are mathematical functions. They transform input values to output values without reference to past events: the same inputs yield the same outputs. \textit{They have no memory} – no state.

Memory emerges when they are connected in circuits with feedback loops (\texttt{numbers, integrate, cell, …}).

Stateless components are trivial to reason about – we don’t have to think about loops! They are also easy to cast into silicon – as, of course, are circuits built from them.
PROC substitute (VAL BYTE a, b, CHAN BYTE in?, out!)

WHILE TRUE

BYTE x:
SEQ
  in ? x
  IF
    x = a
    out ! b
    TRUE
    out ! x
  :

loop-free logic
PROC greater (CHAN BYTE in.0?, in.1?, small!, large!)
WHILE TRUE
BYTE x.0, x.1:
SEQ
  PAR
    in.0 ? x.0
    in.1 ? x.1
  IF
    x.0 < x.1
    PAR
      small ! x.0
      large ! x.1
  TRUE
    PAR
      small ! x.1
      large ! x.0
  :

loop-free logic
Replicators (components and test-rigs)

Replicated PAR and SEQ ...

The SORT PUMP ...

Component testing ...

Stateless components ...

The SORT GRID ...

Replicated IF ...

Replicator STEP sizes ...
On a **serial** processor, **bubble-sort** takes $O(n^2)$ computation time, where $n$ is the number of items being sorted. Cleverer algorithms (such as **quick-sort** or **shell-sort**) take $O(n*\log(n))$.

With $O(n)$ processing elements, the **sort-pump** takes $O(n)$ computation time, with respect to each group of $n$ items being sorted. If we only present data serially (i.e. one item at a time), supply takes $O(n)$ time ... so **sort-pump** cannot be beaten!

**But we do need a continuous supply of groups.**

**Question:** with $O(n^2)$ processing elements, can we sort groups of $n$ items in $O(1)$ time? Of course, we will have to present data in parallel (i.e. $O(1)$ time) and have a continuous supply.

**Answer:** Yes. **And it’s easy!**
If the comparators are implemented on separate pieces of silicon (*i.e. we have a physically parallel engine*), the speed at which data flows through is the *slowest* of:

- the speed at which data is offered;
- the cycle speed for each comparator;
- the inter-cell communication speed.

The speed is independent of the number of comparators – which means that it is independent of the number of items being sorted.

Each group of data *enters* and *exits* the grid *in parallel*. All comparators operate *in parallel*. After each *unit time* cycle, a sorted group emerges. We have an *O(1)* sorting engine:

```
sort.grid.
```
For groups up to size 16, we need 16 rows of (gt) comparators. The even rows have 8 each and the odd rows have 7.

**Coding:** to keep things easy, let’s first program an even-odd pair of rows …
PROC even.odd ([max]CHAN BYTE in?, out!)  
[max-2]CHAN BYTE c:  
PAR  
gt (in[0]?, in[1]?, out[0]!, c[0]!)  
PAR i = 2 FOR (max/2) - 2 STEP 2  
gt (in[i]?, in[i+1]?, c[i-1]!, c[i]!)  
gt (in[max-2]?, in[max-1]?, c[max-3]!, out[max-1]!)  
PAR i = 1 FOR (max/2) - 1 STEP 2  
gt (c[i-1]?, c[i]?, out[i]!, out[i+1]!)  
:

See replicator STEP sizes (later) …
PROC sort.grid ([max]CHAN BYTE in?, out!)

[(max/2)-1][max]CHAN BYTE c:

PAR

even.odd (in?, c[0]!)

PAR i = 0 FOR (max/2) - 2

even.odd (c[i]?, c[i+1]!)

even.odd (c[(max/2)-2]? , out!)

:
Exercise:

Build a test-rig for `sort.grid`...
Replicators *(components and test-rigs)*

- Replicated **PAR** and **SEQ** ...
- The **SORT PUMP** ...
- Component testing ...
- Stateless components ...
- The **SORT GRID** ...
- Replicated **IF** ...
- Replicator **STEP** sizes ...
Summary of Replicators (SEQ, PAR)

One New Replicator (IF)
The replicated `SEQ` is like a very clean `for`-loop.

In Java or C:

```java
for (int i = start; i < (start + count); i++) {
    <code i>
}
```

Must not change the value of `i`, `start` or `count`.
The replicated PAR has no correspondence in Java or C.

INT declaration

first value

PAR i = start FOR count

<process i>

number of replications

In Java or C:

... silence
Replicated IFs

So far, we have seen the *occam*-\(\pi\) process constructors \texttt{SEQ}, \texttt{PAR}, \texttt{IF} and \texttt{WHILE}. (Still to come are \texttt{ALT} and \texttt{CASE}.)

We have seen how \texttt{SEQ} and \texttt{PAR} can be \textit{replicated}. So, also, can the \texttt{IF} and (later) the \texttt{ALT}. Here is a \textit{replicated IF}:

\begin{verbatim}
INT declaration
first value
number of replications

IF i = 0 FOR 4
x[i] = 42
index := i

This \textit{conditional-process} gets replicated
\end{verbatim}
Replicated IFs

So far, we have seen the \texttt{occam-\pi} process constructors \texttt{SEQ}, \texttt{PAR}, \texttt{IF} and \texttt{WHILE}. (Still to come are \texttt{ALT} and \texttt{CASE}.)

We have seen how \texttt{SEQ} and \texttt{PAR} can be \texttt{replicated}. So, also, can the \texttt{IF} and (later) the \texttt{ALT}. Here is a \texttt{replicated IF}:

\begin{verbatim}
IF i = 0 FOR 4
  x[i] = 42
  index := i
\end{verbatim}

\begin{verbatim}
  x[0] = 42
  index := 0
  x[1] = 42
  index := 1
  x[2] = 42
  index := 2
  x[3] = 42
  index := 3
\end{verbatim}
Replicated IFs

This code searches the first 4 elements of the array \( x \) for the value 42. The search is \textit{sequential}, starting from element 0 and proceeding upwards. If successful, the variable \( \text{index} \) is set to the (first) index of the \( x \) array element equal to the target. If unsuccessful, this code will crash!

\[
\begin{array}{c}
\text{IF } i = 0 \text{ FOR 4} \\
\quad x[i] = 42 \\
\quad \text{index} := i \\
\end{array}
\]
To avoid that crash, we need a final condition that catches the flow of control should all the other conditions fail:

\[
\text{IF}\quad \begin{array}{l}
  x[0] = 42 \\
  \text{index} := 0 \\
  x[1] = 42 \\
  \text{index} := 1 \\
  x[2] = 42 \\
  \text{index} := 2 \\
  x[3] = 42 \\
  \text{index} := 3 \\
  \text{TRUE} \\
  \text{index} := -1
\end{array}
\]

To express this using an IF-replicator (which we need if we were searching the through n elements, where n is known only at run-time), we need a nested IF ...

where index is set to -1, an illegal array index, used here to indicate that the search failed.
Nested **IF**s

The inner **IF** disappears and its *conditional processes* align with the *conditional processes* of the outer **IF**.
Nested IFs

Nested IFs are mainly useful ... when the inner or outer is replicated.
Nested IFs

They enable us to IF between sequenced and individual conditions.
Replicated IFs

```
IF
x[0] = 42
  index := 0
x[1] = 42
  index := 1
x[2] = 42
  index := 2
x[3] = 42
  index := 3
TRUE
  index := -1
```

```
IF i = 0 FOR 4
  x[i] = 42
  index := i
TRUE
  index := -1
```

where `index` is set to `-1`, an **illegal array index**, used here to indicate that the **search failed**.
Bounded Linear Search (occam-$\pi$)

The ‘bounded linear search’ is the only common use for a replicated IF – but it is a good one!

Problem: find the index of the first element of some array, $x$, that matches some.condition():

```
IF
  IF i = 0 FOR SIZE x
  some.condition (x[i])
  ... we found it at index i
  TRUE
  ... we didn’t find it
```

Note: the above code searches (potentially) the whole array. We can restrict the search by setting first and replicate values (of the replicated IF) appropriately.
Bounded Linear Search (Java / C)

**Problem:** find the index of the first element of some array, `x`, that matches `some.condition()`:

```java
int i = 0;
bool found = false;
for (i = 0; i < x.length; i++) {
    if (someCondition (x[i])) {
        found = true;
        break;
    }
}
if (found) {
    ... we found it at index i
} else {
    ... we didn’t find it
}
```
Problem: find the index of the first element of some array, $x$, that matches some condition:

Actually, this can be expressed in almost a compact form as in *occam-π* ... but we need to resort to a labelled block with non-local break-out:

```java
BLS: {
    for (int i = 0; i < x.length; i++) {
        if (someCondition(x[i])) {
            ... we found it at index i
            break BLS;
        }
    }
    ... we didn’t find it
}
```
Replicators (components and test-rigs)

Replicated **PAR** and **SEQ** ...

The **SORT PUMP** ...

Component testing ...

Stateless components ...

The **SORT GRID** ...

Replicated **IF** ...

Replicator **STEP** sizes ...
Replicator **STEP** Sizes

Normally, the replicator control value increments by 1 for each replicated instance.

However, we may define an arbitrary **STEP** size for this increment:

\[
\text{INT declaration} \\
\langle rep \rangle \ i = \text{start} \ \text{FOR} \ \text{count} \ \text{STEP} \ \text{size} \\
\langle \text{process} \ i \rangle \\
\text{May not change the value of } i, \ \text{start, count or size}
\]
The `<rep>` constructor is one from: SEQ, PAR, IF and (later) ALT.

The `start`, `count` and `size` may be any INT expressions. The values of `i` and any variables in `start`, `count` and `size` cannot be changed by the replicated process.

```
<rep> i = start FOR count STEP size
```

May not change the value of `i`, `start`, `count` or `size`
Summary: a replicated **SEQ** is a very clean **for-loop**.

```java
{ int i = start;
  for (int ii = 0; ii < count; ii++) {
    <code i>
    i += size;
  }
}
```

**INT declaration**

**first value**

**number of replications**

**increment**

**SEQ** 

```
i = start
FOR count
STEP size
```

**<process i>**

**In Java or C:**

**Must not use ii**

**Must not change the value of i, start, count or size**
The replicated PAR has no correspondence in Java or C.

\[
\text{INT declaration} \rightarrow \text{first value} \rightarrow \text{number of replications} \rightarrow \text{increment} \rightarrow \text{PAR} \ i = \text{start} \ \text{FOR} \ \text{count} \ \text{STEP} \ \text{size} \ \text{<process} \ i> \rightarrow \text{In Java or C:} \ldots \text{silence}
\]
The replicated IF gives a ‘Bounded Linear Search’

```
INT declaration first value number of replications
IF IF i = start FOR count STEP size
    <condition i>
    <found-process i>
    TRUE
    <not-found-process>
```

Unless we know that the search will succeed, we must nest the replicated IF inside a plain IF to catch any failure.
BLS: {
    int i = start;
    for (int ii = 0; ii < count; ii++) {
        if (condition) {
            found-code
            break BLS;
        }
    }
    i += size;
}

The expression and code must not use \(i\) and must not change the value of \(i\), \(start\), \(count\) or \(size\).