Choice and Non-Determinism

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

Co631 (Concurrency)
Choice and Non-Determinism

Non-determinism ...

The **ALT** and **PRI ALT** ...

Control and real-time ...

 Resets and kills ...

 Memory cells ...

 Pre-conditioned guards ...

 Serial **FIFO** (‘ring’) buffer ...

 The replicated **ALT** ...

 Nested **ALTs** ...
Deterministic Processes (CSP)

So far, our parallel systems have been **deterministic**:

- the values in the output streams depend only on the values in the input streams;
- the semantics is scheduling independent;
- no race hazards are possible.

**CSP** parallelism, on its own, **does not introduce non-determinism**.

This gives a firm foundation for exploring real-world models which cannot always behave so simply.
Non-Deterministic Processes (CSP)

In the real world, it is sometimes the case that things happen as a result of:

- what happened in the past;
- when (or, at least, in what order) things happened.

In this world, things are scheduling dependent.

CSP \textit{(and occam-π)} addresses these issues \textit{explicitly}.

Non-determinism does not arise by default.
A Control Process

replace (in?, out!, inject?)

Coping with the real world - making choices ...

In **replace**, data normally flows from **in?** to **out!** unchanged.

However, if something arrives on **inject?**, it is output on **out!** - *instead of* the next input from **in?**.
A Control Process

The **out!** stream depends upon:

- The values contained in the **in** and **inject** streams;
- the **order** in which those values arrive.

The **out!** stream is **not** determined just by the **in?** and **inject?** streams - it is **non-deterministic**.
A Control Process

\[
\text{replace (in?, out!, inject?) = } \left\{ \begin{align*}
&\text{(inject?}x \rightarrow ((\text{in?}a \rightarrow \text{SKIP}) \lor (\text{out!}x \rightarrow \text{SKIP}))} \\
&\text{[PRI]} \quad \text{in?}a \rightarrow \text{out!}a \rightarrow \text{SKIP} \\
\end{align*} \right.
\]

Note: \([\text{PRI}]\) is the (external) choice operator of CSP. \([\text{PRI}]\) is a prioritised version - giving priority to the event on its left.
Another Control Process

Coping with the real world - making choices ...

In \textit{scale}, data flows from \textit{in?} to \textit{out!}, getting scaled by a factor of \textit{s} as it passes.

Values arriving on \textit{inject?} reset the \textit{s} factor.
Another Control Process

The *out*! stream depends upon:

- The values contained in the *in?* and *inject!* streams;
- The *order* in which those values arrive.

The *out*! stream is *not* determined just by the *in?* and *inject?* streams - it is *non-deterministic*. 
Another Control Process

scale (s, in?, out!, inject?) =
(inject?s --> SKIP
  [PRI]
  in?a --> out!s*a --> SKIP
);

Note: [] is the (external) choice operator of CSP.
    [PRI] is a prioritised version - giving priority to the event on its left.
A Real-Time Process

\[ \text{count} \ (\text{period}, \text{in}\?, \text{out}!) \]

Coping with the real world - making choices ...

\text{count} \ \text{observes passing time and messages arriving on} \ \text{period}\ \text{microseconds, it outputs (on} \ \text{out}) \ \text{the number of messages received during the previous} \ \text{period.} \]
A Real-Time Process

(a b c d e)

\[ \text{count (period, in?, out!)} \]

The \textbf{out!} stream depends upon:

- \textbf{When} values arrived on the \textbf{in?} stream (the values received are irrelevant).

The \textbf{out!} stream is \textbf{not} determined by the \textbf{in?} stream values - it is \textbf{non-deterministic}. 

\[ \begin{array}{ccc}
4 & 0 & 5 \\
7 & 2 & 5 \\
0 & 0 & 3 \\
3 & 5 & 0 \\
3 & 8 & 0 \\
\end{array} \]
A Real-Time Process

\[ \text{count} \left( \text{period}, \text{in?}, \text{out!} \right) = \]

standard CSP does not address time …

but \( \text{occam-}\pi \) does …
This is a \textit{resettable} version of the \textit{numbers} process. If nothing is sent down \textit{reset}, it behaves as before. But it may be \textit{reset} to continue counting from \textit{any} number at \textit{any} time.
Non-Deterministic Processes

To enable these, occam-$\pi$ introduces a new programming structure: the ALT ...

... which explicitly introduces non-determinism.

a very simple and elegant idea

will not frighten the horses …
Choice and Non-Determinism

Non-determinism ...

The ALT and PRI ALT ...

Control and real-time ...

Resets and kills ...

Memory cells ...

Pre-conditioned guards ...

Serial FIFO (‘ring’) buffer ...

The replicated ALT ...

Nested ALTs ...
Non-Deterministic Choice

ALT

<guard>
<process>
<guard>
<process>
<guard>
<process>

guarded processes
Non-Deterministic Choice

- A `<guard>` may be **ready** or **not-ready**.

- A **not-ready** `<guard>` may change to **ready** as a result of external activity.

- A **ready** `<guard>` may be executed.
Non-Deterministic Choice

ALT

\[
\begin{align*}
\langle \text{guard} \rangle & \langle \text{process} \rangle \\
\langle \text{guard} \rangle & \langle \text{process} \rangle \\
\langle \text{guard} \rangle & \langle \text{process} \rangle \\
\langle \text{guard} \rangle & \langle \text{process} \rangle \\
\langle \text{guard} \rangle & \langle \text{process} \rangle
\end{align*}
\]

\text{guarded processes}
Non-Deterministic Choice

An \textit{ALT} process executes as follows:

- **if no guard is ready**, the process is suspended until one, or more, become ready;

- **if one guard is ready**, execute it and then execute the process it was defending (end of \textit{ALT} process);

- **if more than one guard is ready**, one is \textit{arbitrarily chosen} and executes, followed by the process it was defending (end of \textit{ALT} process).

\textit{Note: only one of the guarded processes is executed.}
Non-Deterministic Choice

There are 3 types of `<guard>` ...

- input guards
- timeout guards
- SKIP guards
An input guard is **ready** if a process on the other end of the channel is trying to output to that channel and is waiting for its message to be taken.

Execution of this guard *if chosen* is just execution of the input process. Note that execution of this guard leaves it **not-ready** (until another process again outputs to the channel).
Non-Deterministic Choice

PROC crude.plex (CHAN INT in.0?, in.1?, in.2?, out!)
WHILE TRUE
    INT x:
    ALT
        in.0 ? x
        out ! x
        in.1 ? x
        out ! x
        in.2 ? x
        out ! x
    :
Non-Deterministic Choice

A timeout guard is **ready** if the time currently showing on the \textsc{timer} \( t \) is \textsc{after} the time indicated \( t \). Note that the time on a \textsc{timer} continually increments and that the time indicated cannot change while awaiting this timeout.

Execution of this guard \((\text{if chosen})\) is null. Note that execution of this guard leaves it **ready** (until the value of timeout is changed).
Non-Deterministic Choice

PROC watchdog (VAL INT period, CHAN INT in?, out!, CHAN BOOL panic!)

WHILE TRUE
    TIMER tim:
    INT t, x:
    SEQ
        tim ? t
        ALT
        in ? x
        out ! x
        tim ? AFTER t PLUS period
        panic ! TRUE
A **SKIP** guard is always ready.

Execution of this guard *if chosen* is null.
Non-Deterministic Choice

Both guards are ready – so an *arbitrary choice* is made!

Actually, such non-determinism is too much to be useful and the compiler issues warnings – *the programmer probably didn’t mean to write this*!

**SKIP** guards only become useful with *prioritised choice*, which comes next.
Deterministic Choice

PRI ALT

<guard>
<process>

<guard>
<process>

<guard>
<process>

guarded processes
A \textbf{PRI ALT} process executes as follows:

- if no guard is ready, the process is suspended until one, or more, become ready;

- if one guard is ready, execute it and then execute the process it was defending (end of \textbf{PRI ALT} process);

- if more than one guard is ready, the first one listed is chosen and executes, followed by the process it was defending (end of \textbf{PRI ALT} process).

Note: only one of the guarded processes is executed.
Example – Polling a Channel

PRI ALT
  in ? x
  ...
  message was pending
SKIP
  ...
  message was not pending

If no message was pending on the channel, the first guard is not-ready. But the second guard is (always) ready, so that guarded process is executed.

If a message was pending on the channel, the first guard is ready. So (always) is the second guard – but the first has priority and is taken.

A **SKIP** guard lets us poll channels to test if a message is pending and, if so, deal with it. *Beware polling though – it can lead to inefficient and poor design …*
Choice and Non-Determinism

- Non-determinism
- The \textbf{ALT} and \textbf{PRI ALT} ...
- Control and real-time ...
- Resets and kills ...
- Memory cells ...
- Pre-conditioned guards ...
- Serial \textbf{FIFO} (‘ring’) buffer ...
- The replicated \textbf{ALT} ...
- Nested \textbf{ALT}s ...
Example – a Control Process

Earlier example …

replace (in?, out!, inject?)

Coping with the real world - making choices …

In replace, data normally flows from in? to out! unchanged.

However, if something arrives on inject?, it is output on out! - instead of the next input from in?. 
Example – a Control Process

The `out!` stream depends upon:

- The *values* contained in the `in` and `inject` streams;
- the *order* in which those values arrive.

The `out!` stream is *not* determined just by the `in?` and `inject?` streams - it is *non-deterministic*. 
Example – a Control Process

```
PROC replace (CHAN INT in?, out!, inject?)
WHILE TRUE
  INT x, any:
  PRI ALT
    inject ? x  -- replace the
    PAR
    in ? any    -- next ‘in’
    out ! x     -- ‘inject’ value
    in ? x      -- normally
    out ! x     -- just copy through
```

Example – a Control Process

PROC replace (CHAN INT in?, out!, inject?)
WHILE TRUE
PRI ALT

INT x, any:
inject ? x     -- replace the
PAR            -- next ‘in’
in ? any       -- with the
out ! x        -- ‘reset’ value

INT x:
in ? x         -- normally
out ! x        -- just copy through
Locals + Guarded Processes

ALT or PRI ALT

\[
\begin{aligned}
&\text{local declarations} \\
&\text{guard} \\
&\text{process}
\end{aligned}
\]

\[
\begin{aligned}
&\text{local declarations} \\
&\text{guard} \\
&\text{process}
\end{aligned}
\]

\[
\begin{aligned}
&\text{local declarations} \\
&\text{guard} \\
&\text{process}
\end{aligned}
\]

Local declarations are optional

Guarded processes

Local declarations have scope only for the following guarded process
Example – another Control Process

Coping with the real world - making choices …

In \textit{scale}, data flows from \texttt{in?} to \texttt{out!}, getting scaled by a factor of \texttt{s} as it passes.

Values arriving on \texttt{inject?} reset the \texttt{s} factor.
Example – another Control Process

The `out!` stream depends upon:

- The **values** contained in the `in?` and `inject!` streams;
- the **order** in which those values arrive.

The `out!` stream is **not** determined just by the `in?` and `inject?` streams - it is **non-deterministic**.
Example – another Control Process

PROC scale (VAL INT s, CHAN INT in?, out!, inject?)

INT scale:
SEQ
(scale := s)
WHILE TRUE
PRI ALT
(inject ? scale -- get new scale
SKIP
INT x:
in ? x -- data
out ! scale*x -- scale it up

local declaration
Example – another Control Process

PROC scale (VAL INT s, CHAN INT in?, out!, inject?)
INITIAL INT scale IS s:
WHILE TRUE
PRI ALT
    inject ? scale      -- get new scale
    SKIP
    INT x:
    in ? x              -- data
    out ! scale*x      -- scale it up

initialising declaration

local declaration

simplification ...

8-Feb-07
Example – a Real-Time Process

Coping with the real world - making choices ...

\text{count} \text{ in} \? \text{ period} \text{ out}! \text{ observes passing time and messages arriving on period microseconds, it outputs (on out!) the number of messages received during the previous period.}
The **out!** stream depends upon:

- **When** values arrived on the **in?** stream (the values received are irrelevant).

The **out!** stream is *not* determined by the **in?** stream values - it is **non-deterministic**.
Example – a Real-Time Process

PROC count (VAL INT period, CHAN INT in?, out!)

INITIAL INT seen IS 0:
TIMER tim:
INT timeout:
SEQ
  tim ? timeout
  timeout := timeout PLUS period
WHILE TRUE
  PRI ALT
    tim ? AFTER timeout -- timeout
      SEQ
        out ! seen
        seen := 0
        timeout := timeout PLUS period
    INT any:
      in ? any -- data
        seen := seen + 1
  ;

Coding …
Choice and Non-Determinism

Non-determinism ...

The **ALT** and **PRI ALT** ...

Control and real-time ...

Resets and kills ...

Memory cells ...

Pre-conditioned guards ...

Serial **FIFO** (‘ring’) buffer ...

The replicated **ALT** ...

Nested **ALTs** ...
This is a **resettable** version of the **numbers** process. If nothing is sent down **reset**, it behaves as before. But it may be **reset** to continue counting from **any** number at **any** time.
Example – a Resettable Network

PROC numbers.reset (CHAN INT reset?, out!)
CHAN INT a, b, c, d:
PAR
  prefix (0, d?, a!)
  replace (a?, b!, reset?)
  delta (b?, out!, c!)
  succ (c?, d!)

Example – a Resettable Network
Example – a Resettable Network

PROC numbers.reset (CHAN INT reset?, out!)
  INITIAL INT n IS 0:
  WHILE TRUE
    SEQ
      PRI ALT -- poll reset channel
      reset ? n
      SKIP
      SKIP
      out ! n
      n := n PLUS 1
    :

serial implementation
Example – Resettable Integrator

PROC integrate.reset (CHAN INT in?, reset?, out!)
  INITIAL INT total IS 0:
  WHILE TRUE
    SEQ
    PRI ALT
      reset ? total
      SKIP
      INT x:
      in ? x
      total := total + x
      out ! total
    
::
Example – Resettable Integrator

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

parallel implementation
An Inertial Navigation Component

- **acc.in**: carries regular accelerometer samples;
- **vel.reset**: velocity *initialisation* and *corrections*;
- **pos.reset**: position *initialisation* and *corrections*;
- **pos/vel/acc**: regular outputs.
An Inertial Navigation Component

- **acc.in**: carries **regular** accelerometer samples;
- **vel.reset**: velocity **initialisation** and **corrections**;
- **pos.reset**: position **initialisation** and **corrections**;
- **pos/vel/acc**: **regular** outputs.
Half Inertial Navigation Component

- **in**: carries *regular* samples;
- **in.copy**: copy of the **in** stream;
- **out**: *regular* outputs (sample running sums);
- **reset**: running sum *initialisation* and *corrections*.

Build it from two components.
An Inertial Navigation Component

- **acc.in**: carries *regular* accelerometer samples;
- **vel.reset**: velocity *initialisation* and *corrections*;
- **pos.reset**: position *initialisation* and *corrections*;
- **pos/vel/acc**: *regular* outputs.
Example – Integrator (again)

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

serial implementation

\[ \int x + y + z \]

\[ x \]
With an Added Kill Channel

PROC integrate.kill (CHAN INT in?, out!, kill?)
    INITIAL INT total IS 0:
    INITIAL BOOL running IS TRUE:
    ... main loop
    :
With an Added Kill Channel

```
WHILE running ---- main loop
   PRI ALT
   INT any:
   kill ? any
       running := FALSE
   INT x:
   in ? x
   SEQ
       total := total + x
   out ! total

integrate.kill

serial implementation
```

x in
y
z
.
.
.

. out
x
x + y
x + y + z
.
.
.
Example – Integrator (again)

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

parallel implementation
With an Added Kill Channel

\[ x + y + z \]

\[
\begin{align*}
\text{PROC} \ intgr\_\text{kill} \ (\text{CHAN} \ INT \ in?, \ out!, \ kill?) \\
\text{CHAN} \ INT \ a, \ b, \ c, \ d: \\
\text{PAR} \\
\quad \text{killer} \ (in?, \ kill?, \ d!) \\
\quad \text{plus} \ (d?, \ c?, \ a!) \\
\quad \text{delta} \ (a?, \ out!, \ b!) \\
\quad \text{prefix} \ (0, \ b?, \ c!) \\
\end{align*}
\]
To shut down a network *gracefully* (without leaving some processes stranded – i.e. deadlocked), we *poison* all the components. The poison spreads through the normal dataflow.

For *integrate.kill*, the *killer* process injects poison upon receiving a *kill* signal, and then shuts down.
With an Added Kill Channel

This shutdown protocol generalises to work for any process network – see the paper:

“Graceful Termination, Graceful Resetting”
With an Added Kill Channel

The other processes check for poisonous input data – if found, they pass it on and die.

The \textbf{plus} process must wait for the poison to return from the feedback loop before dying.

The \textbf{delta} process only forwards the poison internally – unless it really wants to bring down the next component!
### Choice and Non-Determinism

- Non-determinism ...
- The **ALT** and **PRI ALT** …
- Control and real-time …
- Resets and kills …
- Memory cells …
- Pre-conditioned guards …
- Serial **FIFO** (‘ring’) buffer …
- The replicated **ALT** …
- Nested **ALTs** …
A Memory Cell

PROC mem.cell (CHAN INT in?,
              CHAN BOOL request?,
              CHAN INT out!)

-- WARNING: write before reading!
INT x:
WHILE TRUE
ALT
  in ? x
  SKIP
  BOOL any:
  request ? any
  out ! x
Asynchronous Communication

- A sends information to B.
- A can send at any time (it will never be blocked by B not being ready to receive).
- B can receive data at any time but, first, it has to request some (it will never be blocked by A not being able to send).
- The memory cell acts as a common pool of information.
We could relieve B from having to make requests by combining an auto-prompter with the memory cell.

PROC prompt (CHAN BOOL request!, CHAN INT in?, out!)

WHILE TRUE

INT x:
SEQ
request ! TRUE
in ? x
out ! x

:
Asynchronous Communication

- We could relieve B from having to make requests by combining an auto-prompter with the memory cell.

- But if auto-prompter gets its first request in before A sends anything, it will pick up garbage from the cell.

- Also, if B is not taking data, auto-prompter stores old (stale) data, while the memory-cell holds anything new that arrives. This is probably a bad thing. When B takes data, it wants the latest item that A has sent.
Asynchronous Communication

- We could relieve B from having to make requests by combining an auto-prompter with the memory cell.

- But if auto-prompter gets its first request in before A sends anything, it will pick up garbage from the memory cell. This is probably a bad thing.

- Also, if B is not taking data, auto-prompter stores old (stale) data, while the memory cell holds anything new that arrives. When B takes data, it wants the latest item that A has sent.

The auto-prompter not a good idea here ... (but see its later use the blocking-FIFO buffer).
Regular Events

PROC clock (VAL INT cycle, CHAN BOOL tick!)
    TIMER tim:
    INT t:
    SEQ
        tim ? t
        WHILE TRUE
            SEQ
                t := t PLUS cycle
                tim ? AFTER t
                tick ! TRUE
    : run this at high priority!!
Run all these at high priority

irregular data flow

regular data flow

clock (cycle)

tick
Another Memory Cell

- The implementation of \texttt{mem.cell} captured state information \textit{(the memory)} with a variable. This is OK for the demonstrated application \textit{(asynchronous communication)} … but a bit of a cheat if we want to \textit{model} a variable.

- The following implementation retains state information just by the topology \textit{(feedback loops)} of the internal connections. The internal components do not themselves retain state. They give a design for hardware implementation.
PROC mem.cell (CHAN INT in?, CHAN BOOL request?, CHAN INT out!)

-- WARNING: write before reading!!!
CHAN INT a, b, c:
PAR
    replace (c?, a!, in?)
    sample (a?, b!, request?, out!)
    INT any:
    prefix (any, b?, c!)


PROC replace (CHAN INT in?, out!, inject?)

WHILE TRUE

PRI ALT

INT x, any:

injection ? x  -- replace the
PAR

in ? any  -- next ‘in’

out ! x  -- ‘inject’ value

INT x:

in ? x  -- normally

out ! x  -- just copy through
PROC sample (CHAN INT in?, out!,
CHAN BOOL request?, CHAN INT answer!)

WHILE TRUE
PRI ALT
  BOOL any:
  request ? any
  INT x:
  SEQ
  in ? x
  PAR
  answer ! x   -- duplicate
  out ! x      -- output
  INT x:
  in ? x            -- normally
  out ! x         -- just copy through
  :
PROC vid.cell (CHAN INT in?, CHAN INT video.out!, CHAN BOOL request?, CHAN INT out!)

-- WARNING: write before reading or viewing!!!

CHAN INT a, b, c, d:
PAR
  replace (c?, a!, in?)
  delta (a?, video.out!, d!)
  sample (d?, b!, request?, out!)
  INT any:
  prefix (any, b?, c!)
:
Choice and Non-Determinism

Non-determinism ...
The **ALT** and **PRI ALT** ...
Control and real-time ...
Resets and kills ...
Memory cells ...
Pre-conditioned guards ...
Serial **FIFO** (‘ring’) buffer ...
The replicated **ALT** ...
Nested **ALTs** ...
Non-Deterministic Choice

ALT

\(<\text{guard}>\) \(<\text{process}>\) \(<\text{process}>\) \(<\text{guard}>\) \(<\text{process}>\) \(<\text{guard}>\) \(<\text{process}>\) \(<\text{guard}>\) \(<\text{process}>\) \(<\text{process}>\) \(<\text{guard}>\) \(<\text{process}>\)
Non-Deterministic Choice

An *ALT* process executes as follows:

- if no guard is ready, the process is suspended until one, or more, become ready;

- if one guard is ready, execute it and then execute the process it was defending (*end of ALT* process);

- if more than one guard is ready, one is *arbitrarily chosen* and executes, followed by the process it was defending (*end of ALT* process).

*Note: only one of the guarded processes is executed.*

**Revision:**
A **PRI ALT** process executes as follows:

- **if no guard is ready**, the process is suspended until one, or more, become ready;

- **if one guard is ready**, execute it and then execute the process it was defending *(end of PRI ALT process)*;

- **if more than one guard is ready**, the first one listed is *chosen* and executes, followed by the process it was defending *(end of PRI ALT process)*.

**Note:** only **one** of the guarded processes is executed.
Pre-Conditioned Guards

Any guard may be prefixed by a **BOOL** *pre-condition*:

\[
<\text{pre-condition}> \& <\text{guard}>
\]

When the **ALT** (or **PRI ALT**) starts execution, any *pre-conditions* on the guards are evaluated.

If a *pre-condition* turns out to be **FALSE**, **that guarded process is not chosen for execution** – even if the guard is (or becomes) ready.
Pre-Conditioned Guards

Any guard may be prefixed by a **BOOL** pre-condition:

\[
\langle \text{pre-condition}\rangle \; \& \; \langle \text{guard}\rangle
\]

\{ \langle \text{process}\rangle \}

For each execution of an **ALT** (or **PRI ALT**), any **pre-conditions** only need evaluating **once** – no rechecks are necessary.

A **pre-condition** is a **BOOL** expression, **whose variables cannot change** whilst waiting for a guard to become ready. No other process can change those variables (*simply because this process is observing them*).
INT a:

BOOL timing:

SEQ

... set a and timing

TIMER tim:

INT time.out, x:

SEQ

tim ? time.out

time.out := time.out PLUS 1000

ALT

in.0 ? x

out ! x

in.1 ? x

out ! x

(a = 42) & in.2 ? x

out ! x

RUN-TIME DECISION:

listen out for the in.2 channel?

set the timeout?
Choice and Non-Determinism

Non-determinism ...

The \textbf{ALT} and \textbf{PRI ALT} ...

Control and real-time ...

Resets and kills ...

Memory cells ...

Pre-conditioned guards ...

Serial \textit{FIFO} (‘ring’) buffer ...

The replicated \textbf{ALT} ...

Nested \textbf{ALT}s ...
Another (FIFO) Buffer Process

Recall that …

is a blocking **FIFO** buffer of capacity 6
Another (FIFO) Buffer Process

This is a great and simple design ... \textit{for hardware} ...

... where buffered data can flow \textit{in parallel} along the pipeline ...
Another (FIFO) Buffer Process

This is a great and simple design … *for hardware* …

… but not so good *for software* … where each item of buffered data must be copied (from process to process) $N$ times (where $N$ is the size of the buffer).
Another (FIFO) Buffer Process

So let’s do something better suited for software … that does not do all that copying. Let’s just have one process.

buffer has a capacity of max (say). A process may send data into the buffer until it is full. If it then tries to send more, it will be blocked until the buffer gets emptier.

A process may extract data (by first making a request) until the buffer is empty. If it then requests more, it will be blocked until the buffer gets some data.
Within the buffer, we declare an array (to hold up to max items) and three control variables:

- Number of items currently in the buffer: 5
- Index of the next free slot: 6
- Index of the oldest item in the buffer: 11
If **buffer** receives another item:

- **Number of items currently in the buffer**: 6
- **Index of the next free slot**: 6
- **Index of the oldest item in the buffer**: 12

The diagram shows a buffer with indices from 0 to max-1, with the oldest item at index 12 and the next free slot at index 6. The size of the buffer is 6.
And, then, is requested for and delivers an item:

- **Number of items currently in the buffer**: size 5
- **Index of the next free slot**: hi 12, lo 7
- **Index of the oldest item in the buffer**: max–1, max–2

...
And, then, receives another item:

Number of items currently in the buffer

Index of the next free slot

Index of the oldest item in the buffer

| Number of items currently in the buffer: 6 |
| Index of the next free slot: 7 |
| Index of the oldest item in the buffer: 0 |

holding

| max-1 |
| max-2 |
| 14 |
| 13 |
| 12 |
| 11 |
| 10 |
| 9 |
| 8 |
| 7 |
| 6 |
| 5 |
| 4 |
| 3 |
| 2 |
| 1 |
| 0 |
And another item:

Number of items currently in the buffer: 7

Index of the next free slot: 14

Index of the oldest item in the buffer: 7

Hold sizes:

max-1
max-2

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
And, then, is requested for and delivers an item:

- Number of items currently in the buffer: 6
- Index of the oldest item in the buffer: 8
- Index of the next free slot: 14

Buffer contents:
- max-1
- max-2
- 14
- 13
- 12
- 11
- 10
- 9
- 8
- 7
- 6
- 5
- 4
- 3
- 2
- 1
- 0
- \( \vdots \)
PROC buffer (CHAN INT in?, CHAN BOOL request?, CHAN INT out!)

[max]INT hold:
INT lo, hi, size :: -- size = hi - lo (modulo wrap-around)
SEQ
  lo, hi, size := 0, 0, 0
  WHILE TRUE
    ALT
      (size < max) & in ? hold[hi]
        SEQ
          hi := (hi + 1) \max
          size := size + 1
    BOOL any:
      (size > 0) & request ? any
        SEQ
          out ! hold[lo]
          lo := (lo + 1) \max
          size := size - 1

::
PROC buffer (CHAN INT in?, CHAN BOOL request?, CHAN INT out!)

[max]INT hold:
INT lo, hi, size : -- size = hi - lo (modulo wrap-around)
SEQ
   lo, hi, size := 0, 0, 0
WHILE TRUE
   ALT
      (size < max) & in ? hold[hi]
         SEQ
            hi := (hi + 1)\max
            size := size + 1
      BOOL any:
         (size > 0) & request ? any
            SEQ
            out ! hold[lo]
            lo := (lo + 1)\max
            size := size - 1

Note: the process taking items from this buffer has to make a request ... because output guards are not supported ... despite their semantic power.
PROC buffer (CHAN INT in?, CHAN INT out!)
[max]INT hold:
INT lo, hi, size : -- size = hi – lo (modulo wrap-around)
SEQ
lo, hi, size := 0, 0, 0
WHILE TRUE
ALT
(size < max) & in ? hold[hi]
SEQ
hi := (hi + 1)\max
size := size + 1
(size > 0) & out ! hold[lo]
SEQ
lo := (lo + 1)\max
size := size – 1
:}

Note: the process taking items from this buffer has to make a request ... because output guards are not supported... despite their semantic power.
Output guards require an independent mediator to resolve choices – because more than one process must make the same choice. For example:

Which communication should be done? Either is allowed. Both processes must reach the same decision.

We know how to solve this … but it costs!

By only allowing input guards, only one process is ever involved in any choice (i.e. if one process is ALTing, no process communicating with it can be ALTing).
To relieve the receiving process from the bother of making the requests, we can install an auto-prompter alongside the buffer:

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

```
new.buffer

buffer

prompt

out

in

req

seen before

request

out

in
```
To relieve the receiving process from the bother of making the requests, we can install an auto-prompter alongside the buffer:

![Diagram of buffer and prompter](image)

Just as when used like this with the mem.cell process, prompt holds old (stale) data. Meanwhile, the buffer holds anything new that arrives. *This is a good thing this time!*

Whatever takes data from new.buffer wants the oldest item put into it – it is, after all, a FIFO. 😊😊😊
To relieve the receiving process from the bother of making the requests, we can install an \textit{auto-prompter} alongside the \textit{buffer}:

\begin{center}
\begin{tikzpicture}
\node[draw,rounded corners] {buffer};
\node[draw,rounded corners, right of=buffer, xshift=2cm] {prompt};
\node[draw,rounded corners, below of=prompt, yshift=-2cm] {new.buffer};
\draw[->] (buffer) -- node[near start] {in} (prompt);
\draw[->] (prompt) -- node {req} (buffer);
\draw[->] (prompt) -- node {new.buffer} (new.buffer);
\draw[->] (new.buffer) -- node {out} (prompt);
\end{tikzpicture}
\end{center}

The \textit{prompt} process will be blocked making its first \textit{request} until something is put into the \textit{buffer}.

It then extracts that item and offers it out. When (if) that is taken, \textit{prompt} again requests from \textit{buffer}, which \textit{may} or \textit{may not} have accumulated more items.
To relieve the receiving process from the bother of making the requests, we can install an auto-prompter alongside the buffer:

An empty buffer always blocks a request, leaving new.buffer not trying to output anything.

An non-empty buffer always gives its oldest item, which prompt then offers on out.

So, new.buffer is just a FIFO with capacity $(\text{max} + 1)$. And it has single input/output lines – no request is needed.
To relieve the receiving process from the bother of making the requests, we can install an auto-prompter alongside the buffer:

\[
\text{PROC new.buffer (CHAN INT in?, out!)}
\]

\[
\text{CHAN BOOL req:}
\]

\[
\text{CHAN INT ans:}
\]

\[
\text{PAR}
\]

\[
\text{buffer (in?, req?, ans!)}
\]

\[
\text{prompt (ans?, req!, out!)}
\]

\[
\text{The capacity of new.buffer is (max + 1)}
\]
The top version is a more regular and simpler design. The bottom is more efficient for software – less copying of data.
PROC new.buffer (CHAN INT in?, out!)
  [max]CHAN INT c:
  PAR
    id (in?, c[0]!)
    PAR i = 0 FOR max - 1
      id (c[i]?, c[i+1]!)
    id (c[max - 1]?, out!)
  ::
This is the same as buffer, except that it does not block the source when it is full. Instead, it outputs a signal on the (BOOL) error line and discards the incoming item.

This type of buffer is used in a real-time system if it is important not to delay the source process if the receiver is slow and it is not crucial if we miss some items, so long as we know about it!
Exercise:

This is the similar to `overflow.buffer`; it also does not block the source when it is full. However, the incoming item (when full) is not discarded but **overwrites** the oldest item in the buffer. No error is reported for this (though another version could easily do that).

This type of buffer is used in a real-time system if it is important not to delay the source process if the receiver is slow and we don’t mind losing old items when full. Whatever it holds, it always holds the **latest** values received from the source.
Choice and Non-Determinism

- Non-determinism ...
- The **ALT** and **PRI ALT** ...
- Control and real-time ...
- Resets and kills ...
- Memory cells ...
- Pre-conditioned guards ...
- Serial **FIFO** (‘ring’) buffer ...
- The replicated **ALT** ...
- Nested **ALTs** ...
Consider a process with an array of input channels:

\[ \text{in}[0] \quad \text{in}[1] \quad \text{in}[2] \quad \text{in}[3] \]

And an internal data array of the same type and size as the input channel array.

The process needs to accept any message from any input channel, putting it into the corresponding element of its data array.
Consider a process with an array of input channels:

Before, we introduced the \textit{replicated PAR} for this. We knew that a message on \textit{one} channel was accompanied by a message on \textit{all} channels.

This time, we don’t know the frequency (\textit{if any}) with which messages will arrive from any channel.
We must await these inputs with an ALT:

\[
\text{in}[0] \quad ? \quad x[0] \\
\ldots \quad \text{deal with it} \\
\text{in}[1] \quad ? \quad x[1] \\
\ldots \quad \text{deal with it} \\
\text{in}[2] \quad ? \quad x[2] \\
\ldots \quad \text{deal with it} \\
\text{in}[3] \quad ? \quad x[3] \\
\ldots \quad \text{deal with it}
\]

But what if there were 40 channels in the array? Or 400 ... or 4000 ... ??!
The Replicated ALT

We must await these inputs with an ALT:

```
INT declaration

ALT i = 0 FOR 4
in[i] ? x[i]  
... deal with it

number of replications

This guarded process gets replicated

first value

in[0]
in[1]
in[2]
in[3]
```

**The Replicated ALT**

We must await these inputs with an **ALT**:

```
INT declaration

ALT i = 0 FOR 4
in[i] ? x[i]
... deal with it

number of replications

This guarded process gets replicated

first value

in[0]
in[1]
in[2]
in[3]
```
The Replicated **ALT**

We must await these inputs with an **ALT**:

\[
\text{ALT} \\
\text{in}[0] \ ? \ x[0] \\
\ldots \ \text{deal with it} \\
\text{in}[1] \ ? \ x[1] \\
\ldots \ \text{deal with it} \\
\text{in}[2] \ ? \ x[2] \\
\ldots \ \text{deal with it} \\
\text{in}[3] \ ? \ x[3] \\
\ldots \ \text{deal with it}
\]

\[\equiv\]

\[
\text{ALT } i = 0 \ \text{FOR 4} \\
\text{in}[i] \ ? \ x[i] \\
\ldots \ \text{deal with it}
\]
A Simple Multiplexor

This process just forwards any message it receives …

… but prefixes the message with the index of the channel on which it had been received …

… which will allow subsequent de-multiplexing. ☺ ☺ ☺
A Simple Multiplexor

PROC plex ([]CHAN INT in?, CHAN INT out!)
WHILE TRUE
ALT i = 0 FOR SIZE in?
INT x:
in[i] ? x
SEQ
out ! i
out ! x
...

This guarded process gets replicated
the array size
A Matching De-Multiplexor

This process recovers input messages to their correct output channels ... and assumes each message is prefixed by the correct target channel index ...

Each message must be a `<index, data>` pair, generated by a `plex` process (with the same number of inputs as this has outputs).
A Matching De-Multiplexor

PROC de.plex (CHAN INT in?, [CHAN INT out!)

WHILE TRUE
  INT i, x:
  SEQ
    in ? i
    in ? x
    out[i] ! x
  ::

This must be a legal index of the out array!
Multiplexor Application (Example)

Only a single wire available between the two machines...

machine.a

plex

......

machine.b

de.plex

\[ \text{in}[0] \rightarrow \text{out}[0] \]
\[ \text{in}[1] \rightarrow \text{out}[1] \]
\[ \vdots \]
\[ \text{in}[n-1] \rightarrow \text{out}[n-1] \]
If each message arriving at \texttt{plex} (and departing \texttt{de.plex}) is of type \texttt{THING}, then each message on the multiplexed channel consists of a channel array index (type \texttt{INT}) followed by a \texttt{THING}.
Multiplexor Application (Example)

Message structures should be documented somewhere!
In our example, we were fortunate that the messages to be multiplexed were type INT – the same as channel indices!

This lets us type the multiplexed channel: CHAN INT c:

Remembering that messages on c have form: INT; INT
However, suppose that the messages to be multiplexed were type \texttt{REAL64} …

Now, messages on the multiplexed channel: \texttt{CHAN ??? c:}
Multiplexor Application (Example)

*occam-π* introduces the concept of PROTOCOL, which enables rich message structures (containing possibly mixed types) to be declared for individual channels.

The compiler enforces strict adherence – we gain safety and auto-documentation (of those message structures).
We will return to this example in the chapter on message protocols.
Choice and Non-Determinism

Non-determinism …

The **ALT** and **PRI ALT** …

Control and real-time …

Resets and kills …

Memory cells …

Pre-conditioned guards …

Serial **FIFO** (‘ring’) buffer …

The replicated **ALT** …

Nested **ALTs** …
Nested ALTs and PRI ALTs

The inner ALT disappears and its *guarded processes* align with the *guarded processes* of the outer ALT.
Nested ALTs and PRI ALTs

PRI ALT

<guard 0>
<process 0>

ALT
<guard 1>
<process 1>
<guard 2>
<process 2>
<guard 3>
<process 3>

≡

PRI ALT

<guard 0>
<process 0>

<guard 1>
<process 1>
<guard 2>
<process 2>
<guard 3>
<process 3>

An ALT nested inside a PRI ALT gets prioritised ...
Nested ALTs and PRI ALTs

PRI ALT
<guard 0>

ALT
<guard 1>

<process 1>

<guard 2>

<process 2>

<guard 3>

<process 3>

≡

PRI ALT
<guard 0>

<guard 1>

<process 1>

<guard 2>

<process 2>

<guard 3>

<process 3>

... which is OK (an ALT can always be replaced by a PRI ALT)
Nested ALTs and PRI ALTs

A PRI ALT nested inside an ALT is illegal ...

ALT
<guard 0>
<process 0>
PRI ALT
<guard 1>
<process 1>
<guard 2>
<process 2>
<guard 3>
<process 3>

ALT
<guard 0>
<process 0>
<guard 1>
<process 1>
<guard 2>
<process 2>
<guard 3>
<process 3>
Nested ALTs and PRI ALTs

(a PRI ALT cannot always be replaced by an ALT)
Nested ALTs and PRI ALTs

ALT

\(<\text{guard } 0>\)
\(<\text{process } 0>\)

ALT \(i = 0\) FOR \(n\)
\(<\text{rep guard } i>\)
\(<\text{rep process } i>\)

\(<\text{guard } 1>\)
\(<\text{process } 1>\)

\(\equiv\)

Nested ALTs are mainly useful \(\ldots\) when the inner or outer is replicated.

ALT

\(<\text{guard } 0>\)
\(<\text{process } 0>\)

\(<\text{rep guard } 0>\)
\(<\text{rep process } 0>\)
\(\ldots\)
\(<\text{rep guard } (n-1)>\)
\(<\text{rep process } (n-1)>\)

\(<\text{guard } 1>\)
\(<\text{process } 1>\)
Nested ALTs and PRI ALTs

They enable us to **ALT** between *arrays* of guards and *individuals*.
**For example:**

```
PRI ALT
    tim ? AFTER timeout
    ... deal with it
    BOOL any:
    pause ? any
    pause ? any
    ALT i = 0 FOR SIZE a?
    INT x:
    a[i] ? x
    ... deal with it
```
**Alt**

For example:

\[
\text{ALT } i = 0 \text{ FOR SIZE } a? \\
\text{INT } x: \\
a[i] \ ? \ x \\
\ldots \text{ deal with it}
\]

\[
\text{ALT } i = 0 \text{ FOR SIZE } b? \\
\text{INT } x: \\
b[i] \ ? \ x \\
\ldots \text{ deal with it}
\]

**Als**

between two
arrays of guards.
For example:

**ALT**ing between a 2D array of guards.

\[
\text{ALT } i = 0 \text{ FOR SIZE } a\[i]?
\]

\[
\text{ALT } j = 0 \text{ FOR SIZE } a[i]?
\]

\[
\text{INT } x:
\]

\[
a[i][j] \ ? x
\]

... deal with it