Concurrency, Intuition and Formal Verification: Yes, We Can!

Matt Pedersen\textsuperscript{a} and Peter Welch\textsuperscript{b}

\textsuperscript{a}School of Computer Science, UNLV, USA
\textsuperscript{b}School of Computing, University of Kent, UK

phw@kent.ac.uk
matt@cs.unlv.edu

Curricula for Concurrency and Parallelism
SPLASH 2010, 17\textsuperscript{th}. Oct
**A Thesis** *(for which we have experimental evidence)*

**Not only**

*can we (and *should* we) teach concurrency at the start of the undergraduate CS curriculum …*

**But also**

*we *can* (and we *should*) teach formal analysis and verification of this concurrency at the same time …*
A Thesis (for which we have experimental evidence)

Not only can we (and should we) teach concurrency at the start of the undergraduate CS curriculum …

Because it’s there
Process Orientation

Because it simplifies
CSP / $\pi$-calculus

Because it scales
for complexity

for performance
CSP / $\pi$-calculus

occam-$\pi$ / JCSP
A Thesis (for which we have experimental evidence)

Not only can we (and should we) teach concurrency at the start of the undergraduate CS curriculum …

Because it’s there

Sequence, variables, assignment, parameters, \textit{concurrency}, \textit{channels}, \textit{synchronisation}, …

Fundamental primitives of software engineering

All are important. All are simple. All are available.
Complex and high-performance systems cannot avoid concurrent design, implementation \textit{and reasoning}.

Common concurrency bugs are intermittent – not repeatable on demand. \textit{Untestable in practice}.

We stand on the shoulders of giants (who made the theory and model checkers). \textit{We verify programs just by writing programs … it becomes everyday practice.}

But also \textit{we can (and we should) teach formal analysis and verification of this concurrency at the same time …}
Example: autonomous robot component

The following example has been developed from one first worked through in a single lesson of a graduate class in concurrency at UNLV in the spring of 2010.
Example: autonomous robot component

Device: real-time controller for 8 channels (4 input, 4 output).
**Example: autonomous robot component**

Device: real-time controller for 8 channels (4 input, 4 output).

There are 3 sub-components: **P0** (weapons systems), **P1** (vision processing) and **P2** (motion stabilizer).

They exchange information over internal channels (**ask**, **ans**, **ping**) and all coordinate actions with an internal barrier (**bar**).
Example: autonomous robot component

They exchange information over internal channels (ask, ans, ping) and all coordinate actions with an internal barrier.

CSP semantics apply. Channel communication is unbuffered (sender waits for receiver and vice-versa). Any process reaching a barrier waits for all processes to reach the barrier.
**Behaviour: two representations**

- **occam-π**: for compiling to a runnable system.
  
  \[\text{memory overheads} \leq 32\ \text{bytes per process} / \text{synchronisation overheads of order tens of nanoseconds} / \text{eats multicore nodes for breakfast.}\]

- **CSP**: for formal analysis.
  
  \[\text{FDR2 model checker + other (simple) formal reasoning.}\]
Behaviour: *two representations*

**occam-π**: for compiling to a runnable system.

(memory overheads \(\leq 32\) bytes per process / synchronisation overheads of order tens of nanoseconds / eats multicore nodes for breakfast.)

**CSP**: for formal analysis.

[FDR2 model checker + other (simple) formal reasoning.]

**occam-π** syntax / semantics has an injective mapping to **CSP**. Our students had little trouble shifting between them. A tool exists to generate **CSP** automatically from **occam-π** … not yet ready for use in the classroom.
Behaviour: what are we looking for?

deadlock: might it ever stop?
[e.g. $P_0$ and $P_2$ want to synchronise on $bar$, but $P_1$ wants to $ping$.]

livelock: might it get busy … but refuse all external signals?
[e.g. $P_0$, $P_1$ and $P_2$ start engaging in an infinite sequence of internal channel or barrier synchronisations (on $ask$, $ans$, $ping$ and $bar$).]
Behaviour: what are we looking for?

safety: might it ever engage in an incorrect sequence of external signals?

liveness: will it engage in correct sequences of external signals, as required?

[Some specs allow alternative sequences to be performed – all are correct, but an implementation must only do one and is free to choose.]
For the behaviour analysis in this example, data values and computations are not relevant. For simplicity, they are omitted in these codes, with all message content abstracted to zero.
Behaviour: \texttt{occam-π (executable)}

\begin{figure}[h]
\centering
\begin{tikzpicture}[node distance=3cm,auto,>=latex]
\node (P0) [circle,draw,fill=green!20] {	exttt{P0}};
\node (P1) [circle,draw,fill=blue!20, right of=P0] {	exttt{P1}};
\node (P2) [circle,draw,fill=blue!20, right of=P1] {	exttt{P2}};
\draw[->] (P0) -- node[above] {ask} (P1);
\draw[->] (P1) -- node[above] {ping} (P2);
\draw[->] (P0) -- node[below] {ans} (P1);
\draw[->] (P0) -- node[below] {bar} (P2);
\end{tikzpicture}
\end{figure}

\begin{verbatim}
PROC P0 (CHAN INT a0?, b0?, c0!, ask?, ans!, BARRIER bar)
    WHILE TRUE
        INT x, y, z:
        SEQ
            ask ? x -- take question
            a0 ? y
            ans ! 0 -- return answer \textit{(will depend on x and y)}
            b0 ? z
            SYNC bar -- wait for the others
            c0 ! 0

::
\end{verbatim}
Behaviour: **occam-π** (executable)

```
PROC P1 (CHAN INT a1?, b1?, c1!, ask!, ans?, ping!,
         BARRIER bar)

WHILE TRUE
  INT x, y, z:
  SEQ
    ask ! 0     -- ask question
    ans ? x     -- wait for answer
    a1 ? y
    b1 ? z
    SYNC bar    -- wait for the others
    c1 ! 0
    ping ! 0   -- update neighbour
```

---

![Diagram of P0, P1, P2 with channels a0, b0, c0, a1, b1, c1, d0, d1 and device connections]

---

**Device**

```
<table>
<thead>
<tr>
<th>a0</th>
<th>b0</th>
<th>c0</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>b1</td>
<td>c1</td>
</tr>
<tr>
<td>d0</td>
<td>d1</td>
<td></td>
</tr>
</tbody>
</table>
```

---

**P0**

---

**P1**

---

**P2**

---

**bar**

---

**ping**
Behaviour: \textit{occam-π} (executable)

```
PROC P2 (CHAN INT d0!, d1!, ping?, BARRIER bar)
  WHILE TRUE
    INT x:
    SEQ
      SYNC bar    -- wait for the others
      d0 ! 0
      ping ? x    -- receive update
      SYNC bar    -- wait for the others
      d1 ! 0
      ping ? x    -- receive update
```

Diagram:

- Devices: a0, b0, c0, a1, b1, c1
- Channels: d0, d1
- Processors: P0, P1, P2
- Devices connected to processors via arrows
- Channels connected between processors with arrows
- Processor P2 has an ask and answer channel to devices a1, b1, c1
- Processor P2 has a channel to send and receive updates via d0, d1
- Processor P2 has a channel to synchronize with other processors
- Processor P2 has a channel to synchronize with a barrier

Diagram shows the flow of data between devices and processors.
PROC Device (CHAN INT a0?, b0?, c0!, a1?, b1?, c1!, d0!, d1!)
CHAN INT ask, ans, ping:
BARRIER bar:
PAR ENROLL bar
   P0 (a0?, b0?, c0!, ask?, ans!, bar)
   P1 (a1?, b1?, c1!, ask!, ans?, ping!, bar)
   P2 (d0!, d1!, ping?, bar)
:

Behaviour: occam-π (executable)
**Informal Intuitive**

**Behaviour: occam-$\pi$ (executable)**

```
PROC P0 (CHAN INT a0?, b0?, c0!, ask?, ans!, BARRIER bar)
  WHILE TRUE
    INT x, y, z:
    SEQ
      ask ? x        -- take question
      a0 ? y
      ans ! 0        -- return answer
      b0 ? z
      SYNC bar       -- wait for others
      c0 ! 0
:

PROC P1 (CHAN INT a1?, b1?, c1!, ask!, ans?, ping!, BARRIER bar)
  WHILE TRUE
    INT x, y, z:
    SEQ
      ask ! 0        -- ask question
      ans ? x        -- wait for answer
      a1 ? y
      b1 ? z
      SYNC bar       -- wait for the others
      c1 ! 0
      ping ! 0       -- update neighbour
:

PROC P2 (CHAN INT d0!, d1!, ping?, BARRIER bar)
  WHILE TRUE
    INT x:
    SEQ
      SYNC bar       -- wait for others
      d0 ! 0
      ping ? x       -- receive update
      SYNC bar       -- wait for others
      d1 ! 0
      ping ? x       -- receive update
:
```

What patterns of external (blue) signalling are possible from Device?
**Informal Intuitive**

**Behaviour:** *occam-π* (executable)

**PROC P0** (CHAN INT a0?, b0?, c0!, ask?, ans!,
BARRIER bar)

WHILE TRUE

INT x, y, z:
SEQ
  ask ? x  -- take question
  a0 ? y  -- return answer
  ans ! 0  -- wait for others
  b0 ? z
  SYNC bar  -- wait for others
  c0 ! 0
:

**PROC P1** (CHAN INT a1?, b1?, c1!, ask!, ans?, ping!,
BARRIER bar)

WHILE TRUE

INT x, y, z:
SEQ
  ask ! 0  -- ask question
  ans ? x  -- wait for answer
  a1 ? y
  b1 ? z
  SYNC bar  -- wait for the others
  c1 ! 0
  ping ! 0  -- update neighbour
:

**PROC P2** (CHAN INT d0!, d1!, ping?,
BARRIER bar)

WHILE TRUE

INT x:
SEQ
  SYNC bar  -- wait for others
  d0 ! 0
  ping ? x  -- receive update
  SYNC bar  -- wait for others
  d1 ! 0
  ping ? x  -- receive update
:

What’s first?
Informal Intuitive

Behaviour: \textit{occam-\pi} (executable)

\begin{verbatim}
PROC P0 (CHAN INT a0?, b0?, c0!, ask?, ans!,
         BARRIER bar)
  WHILE TRUE
    INT x, y, z:
    SEQ
      \rightarrow ask ? x    -- take question
        a0 ? y
        ans ! 0    -- return answer
        b0 ? z
      SYNC bar    -- wait for others
      c0 ! 0

PROC P1 (CHAN INT a1?, b1?, c1!, ask!, ans?, ping!,
         BARRIER bar)
  WHILE TRUE
    INT x, y, z:
    SEQ
      \rightarrow ask ! 0    -- ask question
        ans ? x    -- wait for answer
        a1 ? y
        b1 ? z
      SYNC bar    -- wait for the others
      c1 ! 0
      ping ! 0    -- update neighbour

PROC P2 (CHAN INT d0!, d1!, ping?,
         BARRIER bar)
  WHILE TRUE
    INT x:
    SEQ
      \rightarrow SYNC bar    -- wait for others
        d0 ! 0
        ping ? x    -- receive update
      SYNC bar    -- wait for others
        d1 ! 0
        ping ? x    -- receive update

What's first?
\end{verbatim}
Behaviour: \textit{occam-π} (executable)

**Informal Intuitive**

\textbf{Process P0 (CHAN INT a0?, b0?, c0!, ask?, ans!, BARRIER bar)}

\textbf{While True}

\textbf{Int} x, y, z:

\textbf{Seq}

\begin{itemize}
  \item \textbf{ask} ? x \hspace{1em} \texttt{-- ask question}
  \item a0 ? y \hspace{1em} \texttt{-- return answer}
  \item ans ! 0
  \item b0 ? z
  \item \textbf{SYNC bar} \hspace{1em} \texttt{-- wait for others}
  \item c0 ! 0
\end{itemize}

\textbf{:}

\textbf{Process P1 (CHAN INT a1?, b1?, c1!, ask!, ans?, ping!, BARRIER bar)}

\textbf{While True}

\textbf{Int} x, y, z:

\textbf{Seq}

\begin{itemize}
  \item \textbf{ask} ! 0 \hspace{1em} \texttt{-- ask question}
  \item ans ? x \hspace{1em} \texttt{-- wait for answer}
  \item a1 ? y
  \item b1 ? z
  \item \textbf{SYNC bar} \hspace{1em} \texttt{-- wait for the others}
  \item c1 ! 0
  \item ping ! 0 \hspace{1em} \texttt{-- update neighbour}
\end{itemize}

\textbf{:}

\textbf{Process P2 (CHAN INT d0!, d1!, ping?, BARRIER bar)}

\textbf{While True}

\textbf{Int} x:

\textbf{Seq}

\begin{itemize}
  \item \textbf{SYNC bar} \hspace{1em} \texttt{-- wait for others}
  \item d0 ! 0
  \item ping ? x \hspace{1em} \texttt{-- receive update}
  \item \textbf{SYNC bar} \hspace{1em} \texttt{-- wait for others}
  \item d1 ! 0
  \item ping ? x \hspace{1em} \texttt{-- receive update}
\end{itemize}

\textbf{:}

\textbf{What’s first?}

\begin{itemize}
  \item \textbf{a0}
  \item \textbf{[a0]}
\end{itemize}
Behaviour: *occam-π* (executable)

**PROC P0** (CHAN INT a0?, b0?, c0!, ask?, ans!,
BARRIER bar)

WHILE TRUE

INT x, y, z:
SEQ
    ask ? x    -- take question
    a0 ? y    -- return answer
    ans ! 0    -- wait for others
    b0 ? z    -- wait for others
    SYNC bar    -- wait for others
    c0 ! 0

**PROC P1** (CHAN INT a1?, b1?, c1!, ask!, ans?, ping!,
BARRIER bar)

WHILE TRUE

INT x, y, z:
SEQ
    ask ! 0    -- ask question
    ans ? x    -- wait for answer
    a1 ? y    -- wait for the others
    b1 ? z
    SYNC bar    -- update neighbour
    c1 ! 0
    ping ! 0

**PROC P2** (CHAN INT d0!, d1!, ping?,
BARRIER bar)

WHILE TRUE

INT x:
SEQ
    SYNC bar    -- wait for others
    d0 ! 0
    ping ? x    -- receive update
    SYNC bar    -- wait for others
    d1 ! 0
    ping ? x    -- receive update

What's second?

[a0]
Informal
Intuitive

Behaviour: \textit{occam-π (executable)}

\begin{align*}
\text{PROC P0} \ (\text{CHAN INT a0?, b0?, c0!, ask?, ans!, BARRIER bar}) \\
\text{WHILE TRUE} \\
\text{INT} \ x, y, z: \\
\text{SEQ} \\
\quad \text{ask ?} \ x & \quad -- \text{take question} \\
\quad a0 ? y & \quad -- \text{return answer} \\
\quad \text{ans !} \ 0 & \quad -- \text{wait for others} \\
\quad b0 ? z & \quad -- \text{wait for others} \\
\quad \text{SYNC bar} & \quad -- \text{wait for the others} \\
\quad \text{c0 !} \ 0 & \quad -- \text{update neighbour} \\
\end{align*}

\begin{align*}
\text{PROC P2} \ (\text{CHAN INT d0!, d1!, ping?, BARRIER bar}) \\
\text{WHILE TRUE} \\
\text{INT} \ x: \\
\text{SEQ} \\
\quad \text{SYNC bar} & \quad -- \text{wait for others} \\
\quad \text{d0 !} \ 0 & \quad -- \text{receive update} \\
\quad \text{ping ?} \ x & \quad -- \text{wait for others} \\
\quad \text{SYNC bar} & \quad -- \text{wait for others} \\
\quad \text{d1 !} \ 0 & \quad -- \text{receive update} \\
\quad \text{ping ?} \ x & \quad -- \text{receive update} \\
\end{align*}

\begin{align*}
\text{PROC P1} \ (\text{CHAN INT a1?, b1?, cl!, ask!, ans?, ping!, BARRIER bar}) \\
\text{WHILE TRUE} \\
\text{INT} \ x, y, z: \\
\text{SEQ} \\
\quad \text{ask !} \ 0 & \quad -- \text{ask question} \\
\quad \text{ans ?} \ x & \quad -- \text{wait for answer} \\
\quad a1 ? y & \quad -- \text{wait for the others} \\
\quad b1 ? z & \quad -- \text{wait for the others} \\
\quad \text{SYNC bar} & \quad -- \text{update neighbour} \\
\quad \text{cl !} \ 0 & \quad -- \text{update neighbour} \\
\quad \text{ping !} \ 0 & \quad -- \text{update neighbour} \\
\end{align*}

What's second?

[a0]
Behaviour: \texttt{occam-π (executable)}

\textbf{PROC P0} (CHAN INT \texttt{a0?, b0?, c0!, ask?, ans!},
BARRIER \texttt{bar})
\begin{verbatim}
WHILE TRUE
  INT x, y, z:
  SEQ
    ask ? x     \texttt{-- take question}
    a0 ? y
    ans ! 0     \texttt{-- return answer}
    b0 ? z
    SYNC bar    \texttt{-- wait for others}
    c0 ! 0
\end{verbatim}

\textbf{PROC P1} (CHAN INT \texttt{a1?, b1?, c1!, ask!, ans?, ping!},
BARRIER \texttt{bar})
\begin{verbatim}
WHILE TRUE
  INT x, y, z:
  SEQ
    ask ! 0     \texttt{-- ask question}
    ans ? x     \texttt{-- wait for answer}
    a1 ? y
    bl ? z
    SYNC bar    \texttt{-- wait for the others}
    cl ! 0
    ping ! 0    \texttt{-- update neighbour}
\end{verbatim}

\textbf{PROC P2} (CHAN INT \texttt{d0!, d1!, ping?},
BARRIER \texttt{bar})
\begin{verbatim}
WHILE TRUE
  INT x:
  SEQ
    SYNC bar    \texttt{-- wait for others}
    d0 ! 0
    ping ? x    \texttt{-- receive update}
    SYNC bar    \texttt{-- wait for others}
    d1 ! 0
    ping ? x    \texttt{-- receive update}
\end{verbatim}

\textbf{What's second?}

\begin{itemize}
  \item \texttt{b0}
  \item \texttt{a1}
\end{itemize}

\textit{[a0]}
Behaviour: *occam-π* (executable)

**PROC P0** (CHAN INT \(a_0?, \ b_0?, \ c_0!, \ ask?, \ ans!,\)
BARRIER bar)

WHILE TRUE

INT \(x, \ y, \ z:\)

SEQ

ask ? \(x\) -- take question
\(a_0 ? \ y\)
ans ! 0 -- return answer
\(b_0 ? \ z\)
SYNC bar -- wait for others
\(c_0 ! 0\)

:

**PROC P1** (CHAN INT \(a_1?, \ b_1?, \ c_1!, \ ask!, \ ans?, \ ping!,\)
BARRIER bar)

WHILE TRUE

INT \(x, \ y, \ z:\)

SEQ

ask ! 0 -- ask question
ans ? \(x\) -- wait for answer
\(a_1 ? \ y\)
\(b_1 ? \ z\)
SYNC bar -- wait for the others
\(c_1 ! 0\)
ping ! 0 -- update neighbour

:

**PROC P2** (CHAN INT \(d_0!, \ d_1!, \ ping?\),
BARRIER bar)

WHILE TRUE

INT \(x:\)

SEQ

SYNC bar -- wait for others
\(d_0 ! 0\)
ping ? \(x\) -- receive update
SYNC bar -- wait for others
\(d_1 ! 0\)
ping ? \(x\) -- receive update

:

If \(b_0\) second, then?

\([a_0, b_0]\)
**Informal Intuitive**

**Behaviour: occam-π (executable)**

**PROC P0** (CHAN INT a0?, b0?, c0!, ask?, ans!,
BARRIER bar)

WHILE TRUE
  INT x, y, z:
  SEQ
    ask ? x -- take question
    a0 ? y
    ans ! 0 -- return answer
    b0 ? z
  SYNC bar -- wait for others
  c0 ! 0
:

**PROC P1** (CHAN INT a1?, b1?, c1!, ask!, ans?, ping!,
BARRIER bar)

WHILE TRUE
  INT x, y, z:
  SEQ
    ask ! 0 -- ask question
    ans ? x -- wait for answer
    a1 ? y
    b1 ? z
  SYNC bar -- wait for the others
  c1 ! 0
  ping ! 0 -- update neighbour
:

**PROC P2** (CHAN INT d0!, d1!, ping?,
BARRIER bar)

WHILE TRUE
  INT x:
  SEQ
     SYNC bar -- wait for others
     d0 ! 0
     ping ? x -- receive update
     SYNC bar -- wait for others
     d1 ! 0
     ping ? x -- receive update
:

If **b0** second, then?
If **al**

[a0, b0, al]
Behaviour: \textit{occam-\pi} (executable)

\begin{itemize}
  \item PROC \textsc{P0} (CHAN INT $a_0\?, b_0\?, c_0\!, \text{ask}\?, \text{ans}\!,$
              \text{BARRIER bar})
    \item WHILE TRUE
      \item INT $x, y, z$
      \item SEQ
        \item ask ? $x$ \hfill -- take question
        \item $a_0 \? y$
        \item ans ! 0 \hfill -- return answer
        \item $b_0 \? z$
        \item SYNC bar \hfill -- wait for others
        \item $c_0 \! 0$
  \end{itemize}

\begin{itemize}
  \item PROC \textsc{P2} (CHAN INT $d_0\!, d_1\!, \text{ping}\?$,
        \text{BARRIER bar})
    \item WHILE TRUE
      \item INT $x$
      \item SEQ
        \item \rightarrow SYNC bar \hfill -- wait for others
        \item $d_0 \! 0$
        \item \rightarrow ping ? $x$ \hfill -- receive update
        \item \rightarrow SYNC bar \hfill -- wait for others
        \item $d_1 \! 0$
        \item ping ? $x$ \hfill -- receive update
  \end{itemize}

\begin{itemize}
  \item PROC \textsc{P1} (CHAN INT $a_1\?, b_1\?, c_1\!, \text{ask}\!, \text{ans}\?, \text{ping}\!$,\n       \text{BARRIER bar})
    \item WHILE TRUE
      \item INT $x, y, z$
      \item SEQ
        \item \rightarrow ask ! 0 \hfill -- ask question
        \item ans ? $x$ \hfill -- wait for answer
        \item $a_1 \? y$
        \item $b_1 \? z$
        \item SYNC bar \hfill -- wait for the others
        \item $c_1 \! 0$
        \item ping ! 0 \hfill -- update neighbour
  \end{itemize}

\begin{itemize}
  \item If $b_0$ second, then?
  \item \begin{itemize}
        \item $a_0, b_0, a_1, b_1$
  \end{itemize}
  \item then $b_1$
\end{itemize}
Behaviour: \textit{occam-π} (executable)

\begin{align*}
\text{PROC P0} & \quad (\text{CHAN INT } a0?, \ b0?, \ c0!, \ ask?, \ ans!, \ \text{BARRIER bar}) \\
\text{WHILE TRUE} & \\
\text{INT } x, \ y, \ z: & \\
\text{SEQ} & \\
\text{ask } ? \ x & \quad \text{-- take question} \\
a0 ? \ y & \quad \text{-- return answer} \\
\text{ans } ! \ 0 & \\
b0 ? \ z & \text{-- wait for others} \\
\text{SYNC bar} & \quad \text{-- wait for others} \\
c0 ! \ 0 & \\
\end{align*}

\begin{align*}
\text{PROC P1} & \quad (\text{CHAN INT } a1?, \ b1?, \ c1!, \ ask!, \ ans?, \ \text{ping!}, \ \text{BARRIER bar}) \\
\text{WHILE TRUE} & \\
\text{INT } x, \ y, \ z: & \\
\text{SEQ} & \\
\text{ask } ! \ 0 & \quad \text{-- ask question} \\
\text{ans } ? \ x & \quad \text{-- wait for answer} \\
a1 ? \ y & \\
b1 ? \ z & \text{-- wait for the others} \\
\text{SYNC bar} & \quad \text{-- wait for the others} \\
c1 ! \ 0 & \quad \text{-- update neighbour} \\
\text{ping } ! \ 0 & \\
\end{align*}

\begin{align*}
\text{PROC P2} & \quad (\text{CHAN INT } d0!, \ d1!, \ \text{ping?}, \ \text{BARRIER bar}) \\
\text{WHILE TRUE} & \\
\text{INT } x: & \\
\text{SEQ} & \\
\text{SYNC bar} & \quad \text{-- wait for others} \\
d0 ! \ 0 & \\
ping ? \ x & \quad \text{-- receive update} \\
\text{SYNC bar} & \quad \text{-- wait for others} \\
d1 ! \ 0 & \\
ping ? \ x & \quad \text{-- receive update} \\
\end{align*}

If \begin{align*}
\text{\textbullet ~ b0} & \quad \text{second, then?} \\
\text{\textbullet ~ a1} & \quad \text{then} \\
\text{\textbullet ~ b1} & \end{align*}
\begin{align*}
\text{[a0, b0, a1, b1]} & \end{align*}
Behaviour: \textit{occam-π (executable)}

\begin{verbatim}
PROC P0 (CHAN INT a0?, b0?, c0!, ask?, ans!,
         BARRIER bar)
  WHILE TRUE
    INT x, y, z:
    SEQ
      ask ? x     -- take question
      a0 ? y
      ans ! 0     -- return answer
      b0 ? z
      SYNC bar    -- wait for others
      c0 ! 0
: 

PROC P1 (CHAN INT a1?, b1?, c1!, ask!, ans?, ping!,
         BARRIER bar)
  WHILE TRUE
    INT x, y, z:
    SEQ
      ask ! 0     -- ask question
      ans ? x     -- wait for answer
      a1 ? y
      b1 ? z
      SYNC bar    -- wait for the others
      c1 ! 0
      ping ! 0    -- update neighbour
: 

PROC P2 (CHAN INT d0!, d1!, ping?,
         BARRIER bar)
  WHILE TRUE
    INT x:
    SEQ
      SYNC bar    -- wait for others
      d0 ! 0
      ping ? x    -- receive update
      SYNC bar    -- wait for others
      d1 ! 0
      ping ? x    -- receive update
: 
\end{verbatim}

Informal

Intuitive

What's second?

backtracking ...
**Behaviour:** \texttt{occam-\pi} (executable)

**Informal**

**Intuitive**

- **PROC P0** (CHAN INT \texttt{a0}, \texttt{b0}, \texttt{c0}, \texttt{ask}, \texttt{ans}, BARRIER \texttt{bar})
  
  \[
  \text{WHILE TRUE} \\
  \text{INT } x, y, z; \\
  \text{SEQ} \\
  \text{ask } ? x \quad \text{-- take question} \\
  \text{a0 } ? y \quad \text{-- return answer} \\
  \text{ans } ! 0 \quad \text{-- wait for others} \\
  \text{b0 } ? z \quad \text{-- wait for others} \\
  \text{SYNC bar} \quad \text{-- wait for others} \\
  \text{c0 } ! 0 \quad \text{-- wait for others} \\
  \]

- **PROC P1** (CHAN INT \texttt{a1}, \texttt{b1}, \texttt{c1}, \texttt{ask}, \texttt{ans}, \texttt{ping}, BARRIER \texttt{bar})
  
  \[
  \text{WHILE TRUE} \\
  \text{INT } x, y, z; \\
  \text{SEQ} \\
  \text{ask } ! 0 \quad \text{-- ask question} \\
  \text{ans } ? x \quad \text{-- wait for answer} \\
  \text{a1 } ? y \quad \text{-- wait for the others} \\
  \text{bl } ? z \quad \text{-- wait for the others} \\
  \text{SYNC bar} \quad \text{-- wait for the others} \\
  \text{cl } ! 0 \quad \text{-- update neighbour} \\
  \text{ping } ! 0 \quad \text{-- update neighbour} \\
  \]

- **PROC P2** (CHAN INT \texttt{d0}, \texttt{d1}, \texttt{ping}, BARRIER \texttt{bar})
  
  \[
  \text{WHILE TRUE} \\
  \text{INT } x; \\
  \text{SEQ} \\
  \text{SYNC bar} \quad \text{-- wait for others} \\
  \text{d0 } ! 0 \quad \text{-- wait for others} \\
  \text{ping } ? x \quad \text{-- receive update} \\
  \text{SYNC bar} \quad \text{-- wait for others} \\
  \text{d1 } ! 0 \quad \text{-- receive update} \\
  \text{ping } ? x \quad \text{-- receive update} \\
  \]

If \textbf{\textit{a0, a1}} second, then?
Informal
Intuitive

Behaviour: \texttt{occam-Π (executable)}

\begin{verbatim}
PROC P0 (CHAN INT a0?, b0?, c0!, ask?, ans!,
        BARRIER bar)
  WHILE TRUE
    INT x, y, z:
    SEQ
      ask ? x     -- take question
      a0 ? y
      ans ! 0     -- return answer
      b0 ? z
      SYNC bar    -- wait for the others
      c0 ! 0
  :

PROC P1 (CHAN INT a1?, b1?, c1!, ask!, ans?, ping!,
        BARRIER bar)
  WHILE TRUE
    INT x, y, z:
    SEQ
      ask ! 0     -- ask question
      ans ? x     -- wait for answer
      a1 ? y
      bl ? z
      SYNC bar    -- wait for the others
      cl ! 0
      ping ! 0    -- update neighbour
  :

PROC P2 (CHAN INT d0!, d1!, ping?,
        BARRIER bar)
  WHILE TRUE
    INT x:
    SEQ
      SYNC bar    -- wait for others
      d0 ! 0
      ping ? x    -- receive update
      SYNC bar    -- wait for others
      d1 ! 0
      ping ? x    -- receive update
  :
\end{verbatim}

If \( a_1 \) second, then? \( b_0 \) and \( b_1 \) *

**\([a_0, a_1]\) (\ast any order)**
**Behaviour: occam-π (executable)**

**Informal Intuitive**

**PROC P0** (CHAN INT a0?, b0?, c0!, ask?, ans!, BARRIER bar)

WHILE TRUE

INT x, y, z:

SEQ

ask ? x -- take question
a0 ? y
ans ! 0 -- return answer
b0 ? z
SYNC bar -- wait for others
c0 ! 0

:

**PROC P1** (CHAN INT a1?, b1?, c1!, ask!, ans?, ping!, BARRIER bar)

WHILE TRUE

INT x, y, z:

SEQ

ask ! 0 -- ask question
ans ? x -- wait for answer
a1 ? y
b1 ? z
SYNC bar -- wait for the others
c1 ! 0
ping ! 0 -- update neighbour

:

**PROC P2** (CHAN INT d0!, d1!, ping?, BARRIER bar)

WHILE TRUE

INT x:

SEQ

SYNC bar -- wait for others
d0 ! 0
ping ? x -- receive update
SYNC bar -- wait for others
d1 ! 0
ping ? x -- receive update

:

If a1 second, then?

b0 and b1

[a0, a1, b0, b1]

[a0, a1, b1, b0]
Informal Intuitive

**Behaviour: occam-\(\pi\) (executable)**

PROC P0 (CHAN INT a0?, b0?, c0!, ask?, ans!,
          BARRIER bar)
WHILE TRUE
  INT x, y, z:
  SEQ
    ask ? x     -- take question
    a0 ? y
    ans ! 0     -- return answer
    b0 ? z
    SYNC bar    -- wait for others
    c0 ! 0
:

PROC P1 (CHAN INT a1?, b1?, c1!, ask!, ans?, ping!,
          BARRIER bar)
WHILE TRUE
  INT x, y, z:
  SEQ
    ask ! 0     -- ask question
    ans ? x     -- wait for answer
    a1 ? y
    b1 ? z
    SYNC bar    -- wait for the others
    c1 ! 0
    ping ! 0    -- update neighbour
:

PROC P2 (CHAN INT d0!, d1!, ping?,
          BARRIER bar)
WHILE TRUE
  INT x:
  SEQ
    SYNC bar    -- wait for others
    d0 ! 0
    ping ? x    -- receive update
    SYNC bar    -- wait for others
    d1 ! 0
    ping ? x    -- receive update
:

If a1 second, then? b0 and b1

[a0, a1, b0, b1]
[a0, a1, b1, b0]
Behaviour: \textit{occam-\pi} (executable)

**PROC P0** (CHAN INT \(a0?, \ b0?, \ c0!, \ ask?, \ ans!,\)
BAR)</br>WHILE TRUE
INT \(x, \ y, \ z:\)
SEQ
 \(\text{ask} \ ? \ x\quad\text{-- take question}\)
 \(a0 \ ? \ y\quad\text{-- return answer}\)
 \(\text{SYNC} \ \text{bar}\quad\text{-- wait for others}\)
 \(\text{c0} ! 0\quad\text{-- update neighbour}\):

**PROC P2** (CHAN INT \(d0!, \ d1!, \ ping?,\)
BAR)</br>WHILE TRUE
INT \(x:\)
SEQ
 \(\text{SYNC} \ \text{bar}\quad\text{-- wait for others}\)
 \(d0 ! 0\quad\text{-- wait for others}\)
 \(\text{ping} \ ? \ x\quad\text{-- receive update}\)
 \(\text{SYNC} \ \text{bar}\quad\text{-- wait for others}\)
 \(d1 ! 0\quad\text{-- receive update}\):

**PROC P1** (CHAN INT \(a1?, \ b1?, \ c1!, \ ask!, \ ans?, \ ping!,\)
BAR)</br>WHILE TRUE
INT \(x, \ y, \ z:\)
SEQ
 \(\text{ask} \ ! 0\quad\text{-- ask question}\)
 \(\text{ans} \ ? \ x\quad\text{-- wait for answer}\)
 \(a1 \ ? \ y\quad\text{-- wait for the others}\)
 \(b1 \ ? \ z\quad\text{-- update neighbour}\)

\([a0, b0, a1, b1]\)
\([a0, a1, b0, b1]\)
\([a0, a1, b1, b0]\)

What next?
**Informal Intuitive**

**Behaviour: occam-$\pi$ (executable)**

**PROC P0** (CHAN INT $a0?$, $b0?$, $c0!$, ask?, ans!, BARRIER bar)

WHILE TRUE
  
  INT $x$, $y$, $z$:
  
  SEQ
  
  ask ? $x$ -- take question
  
a0 ? $y$
  
  ans ! 0 -- return answer
  
b0 ? $z$
  
  SYNC bar -- wait for others
  
c0 ! 0

```
[a0, b0, a1, b1]
[a0, a1, b0, b1]
[a0, a1, b1, b0]
```

**PROC P1** (CHAN INT $a1?$, $b1?$, $c1!$, ask!, ans?, ping!, BARRIER bar)

WHILE TRUE

  INT $x$, $y$, $z$:

  SEQ

  ask ! 0 -- ask question

  ans ? $x$ -- wait for answer

  a1 ? $y$

  b1 ? $z$

  SYNC bar -- wait for the others

  c1 ! 0

  ping ! 0 -- update neighbour

```
[a0, b0, a1, b1]
[a0, a1, b0, b1]
[a0, a1, b1, b0]
```

**PROC P2** (CHAN INT $d0!$, $d1!$, ping?, BARRIER bar)

WHILE TRUE

  INT $x$:

  SEQ

 SYNC bar -- wait for others

  d0 ! 0

  ping ? $x$ -- receive update

  SYNC bar -- wait for others

  d1 ! 0

  ping ? $x$ -- receive update

```
[a0, b0, a1, b1]
[a0, a1, b0, b1]
[a0, a1, b1, b0]
```

What next?

- $c0$
- $cl$
- $d0$

(*) any order
**Behaviour: occam-π (executable)**

**PROC P0** (CHAN INT a0?, b0?, c0!, ask?, ans!,
BARRIER bar)

WHILE TRUE
   INT x, y, z:
   SEQ
      ask ? x    -- take question
      a0 ? y
      ans ! 0    -- return answer
      b0 ? z
      SYNC bar   -- wait for others
      c0 ! 0

::

**PROC P1** (CHAN INT a1?, b1?, c1!, ask!, ans?, ping!,
BARRIER bar)

WHILE TRUE
   INT x, y, z:
   SEQ
      ask ! 0     -- ask question
      ans ? x     -- wait for answer
      a1 ? y
      b1 ? z
      SYNC bar    -- wait for the others
      c1 ! 0     -- update neighbour

::

**PROC P2** (CHAN INT d0!, d1!, ping?,
BARRIER bar)

WHILE TRUE
   INT x:
   SEQ
      SYNC bar    -- wait for others
d0 ! 0
      ping ? x    -- receive update
      SYNC bar    -- wait for others
d1 ! 0
      ping ? x    -- receive update

::

That’s 18 possible orderings of the first 7 signals.

What happens when the sub-processes start looping?
Informal

Intuitive

Behaviour: **occam-\pi** (executable)

**Behaviour:**

**PROCP**

PROC P0 (CHAN INT a0?, b0?, c0!, ask?, ans!,

BARRIER bar)

WHILE TRUE

INT x, y, z:

SEQ

ask ? x  -- take question

a0 ? y

ans ! 0  -- return answer

b0 ? z

SYNC bar  -- wait for others

c0 ! 0

:

PROC P2 (CHAN INT d0!, d1!, ping?,

BARRIER bar)

WHILE TRUE

INT x:

SEQ

SYNC bar  -- wait for others

d0 ! 0

ping ? x  -- receive update

SYNC bar  -- wait for others

d1 ! 0

ping ? x  -- receive update

:

PROC P1 (CHAN INT a1?, b1?, c1!, ask!, ans?, ping!,

BARRIER bar)

WHILE TRUE

INT x, y, z:

SEQ

ask ! 0  -- ask question

ans ? x  -- wait for answer

a1 ? y

b1 ? z

SYNC bar  -- wait for the others

c1 ! 0

ping ! 0  -- update neighbour

:

Could P0 signal **again** on a0 before P2 gave its first d0?

Are there some more possible **first-7** signal sequences?
We can formally verify the previous intuition (which was only about the opening behaviour of the system) and answer the open questions (and more) about its continuous behaviour with a CSP representation.

We use CSP-M, the machine readable form used by the FDR2 model checker. CSP-M is a declarative (functional) language – loops map to tail recursions. Students who enjoy programming have no problem learning new syntax (it's particularly easy when the semantics remain unchanged) – but they need to be told why!
**CSP-M** lets us abstract the channel communications further by omitting the data sent (always zero in our example) and the direction of communication (irrelevant here).

**CSP** processes synchronise only on **events**, which capture the notions of point-to-point channels and multiway barriers. **CSP-M** calls them all **channels**.

In the following **CSP-M**, we further simplify things by omitting process parameters and accessing all channels from global declaration. *We could have done this with the occam-$\pi$ ...*
**Behaviour: CSP-M (verifyable)**

```
channel a0, b0, c0, a1, b1, c1, d0, d1, ask, ans, ping, barbar

P0 = ask -> a0 -> ans -> b0 -> bar -> c0 -> P0

PROC P0 =
  WHILE TRUE
  INT x, y, z:
  SEQ
    ask ? x          -- take question
    a0 ? y
    ans ! 0
    b0 ? z
    ans ! 0
    SYNC bar
    c0 ! 0
    bar
```

Device

P0 = ask --> a0 --> ans --> b0 --> bar --> c0 --> P0
**Formal**

**Behaviour: CSP-M (verifyable)**

P0 = ask

PROC P1 (CHAN INT a1?, b1?, c1!, ask!, ans?, ping?, bar)

WHILE TRUE

INT x, y, z:

SEQ

ask ! 0
ans ? x
al ? y
bl ? z
sync bar

cl ! 0
ping ! 0

-- ask question
-- wait for ans?
-- wait for the others

update neighbour

P1 = ask -> ans -> a1 -> b1 -> bar -> c1 -> ping -> P1

channel a0 a1 a2 a3

channel b0 b1 b2 b3

channel c0 c1 c2 c3

channel d0 d1 d2 d3
Behaviour: *CSP-M* (verifyable)

```
channel, channel a0, b0, c0, a1, b1, c1, d0, d1, ask, ans, ping, bar

PROC P0
    WHILE TRUE
        INT x:
        SEQ
            SYNC bar
            d0 ! 0
            ping ? x
            -- receive update
            d1 ! 0
            ping ? x
            -- receive update

P0 = ask, ans, ping, bar
```

```
P1 = ask, ans, ping, bar
```

```
P2 = bar --> d0 --> ping --> bar --> d1 --> ping --> P2
```
Formal

**Behaviour: CSP-M (verifyable)**

channel a0, b0, c0, a1, b1, c1, d0, d1, ask, ans, ping, bar

P0 = ask → a0 → ans → b0 → bar → c0 → P0

P1 = ask → ans → a1 → b1 → bar → c1 → ping → P1

P2 = bar → d0 → ping → bar → d1 → ping → P2

POP1 = (P0 || {ask, ans, bar}) || P1) \ {ask, ans}

Device = (POP1 || {ping, bar}) || P2) \ {ping, bar}
Loading the system below into **FDR2**, we discover straight away that **Device** is *free from deadlock and livelock* – just click the buttons!

![Smiley faces]

**Formal**

**Behaviour:** CSP-M *(verifyable)*

```plaintext
channel a0, b0, c0, al, b1, cl, d0, d1, ask, ans, ping, bar

P0 = ask -> a0 -> ans -> b0 -> bar -> c0 -> P0

P1 = ask -> ans -> al -> b1 -> bar -> cl -> ping -> P1

P2 = bar -> d0 -> ping -> bar -> d1 -> ping -> P2

P0P1 = (P0 || {ask, ans, bar} || P1) \ {ask, ans}

Device = (P0P1 || {ping, bar} || P2) \ {ping, bar}
```
To check whether particular event sequences \((\text{traces})\) may initially be performed by \textbf{Device} ... e.g.

Define processes that have no choice in the matter ... e.g.

\[
\begin{align*}
T_0 &= a_0 \rightarrow b_0 \rightarrow a_1 \rightarrow b_1 \rightarrow d_0 \rightarrow c_0 \rightarrow c_1 \rightarrow \text{STOP} \\
T_1 &= a_0 \rightarrow b_0 \rightarrow a_1 \rightarrow d_0 \rightarrow b_1 \rightarrow c_0 \rightarrow c_1 \rightarrow \text{STOP}
\end{align*}
\]

And ask: does each \underline{\text{trace refine Device}}?

Process \(\textbf{P} \text{ trace refines } \textbf{Q} \) if all \(\text{traces} \) of \(\textbf{P} \) are \(\text{traces} \) of \(\textbf{Q} \).

\[
Q \ [T= P]
\]
To check whether particular event sequences (traces) may initially be performed by Device … e.g.

Define processes that have no choice in the matter … e.g.

\[
\begin{align*}
T_0 &= a_0 \rightarrow b_0 \rightarrow a_1 \rightarrow b_1 \rightarrow d_0 \rightarrow c_0 \rightarrow c_1 \rightarrow \text{STOP} \\
T_1 &= a_0 \rightarrow b_0 \rightarrow a_1 \rightarrow d_0 \rightarrow b_1 \rightarrow c_0 \rightarrow c_1 \rightarrow \text{STOP}
\end{align*}
\]

FDR2 reports that \textbf{T0 trace refines Device} … but \textbf{T1 does not} – which confirms our intuition. ☺☺☺

\[ \text{Device } [T= T_0] \quad \checkmark \quad \text{Device } [T= T_1] \quad \times \]
Behaviour: **CSP-M** (verifyable)

To check whether particular event sequences (traces) may initially be performed by **Device** ... e.g.

Define processes that have no choice in the matter ... e.g.

\[
\begin{align*}
T_0 &= a_0 \rightarrow b_0 \rightarrow a_1 \rightarrow b_1 \rightarrow d_0 \rightarrow c_0 \rightarrow c_1 \rightarrow \text{STOP} \\
T_1 &= a_0 \rightarrow b_0 \rightarrow a_1 \rightarrow d_0 \rightarrow b_1 \rightarrow c_0 \rightarrow c_1 \rightarrow \text{STOP}
\end{align*}
\]

Clearly, \([a_0, b_0, a_1, b_1, d_0, c_0, c_1]\) is a trace of **T0**. Therefore, it is also a trace of **Device**.
To check whether particular event sequences (*traces*) may initially be performed by Device ... e.g.

Define processes that have no choice in the matter ... e.g.

\[T_0 = a_0 \rightarrow b_0 \rightarrow a_1 \rightarrow b_1 \rightarrow d_0 \rightarrow c_0 \rightarrow c_1 \rightarrow \text{STOP}\]
\[T_1 = a_0 \rightarrow b_0 \rightarrow a_1 \rightarrow d_0 \rightarrow b_1 \rightarrow c_0 \rightarrow c_1 \rightarrow \text{STOP}\]

At least one trace of \(T_1\) is *not* a trace of Device. Comparing \(T_0\) and \(T_1\), the fault lies in the mis-ordering of \(d_0\) and \(b_1\).
Let’s ask a more difficult question about the continuous running of the system. Suppose the robot would do something very bad if its controller were ever to signal twice on a0 without a signal on d0 or d1 in between. Might this ever happen?

**Simple:** write a process checking the signals to/from Device, looking for the bad scenario and deadlocks if spotted. This is just programming …
**Behaviour:** CSP-M *(verifyable)*

**Formal**

**Safety**

```
Check (n) =
  if n >= 2 then STOP else
  a0 -> Check (n+1) []
  a1 -> Check (n)
  c0 -> Check (n)
  d0 -> Check (0)
  d1 -> Check (0)
  b0 -> Check (n)
  b1 -> Check (n)
  c1 -> Check (n)
```

**Simple:** write a process checking the signals to/from **Device**, looking for the bad scenario and deadlocks if spotted. This is just programming …
The operator "[]" means wait for one or more of the operand processes to become able to run ... choose one of them and run.
The parameter to Check records how many a0 signals have been received since the last d0 or d1, stopping if this reaches 2.
**Behaviour: CSP-M (verifyable)**

Formal Safety

Check \((n)\) =

if \(n \geq 2\) then STOP else

- \(a_0 \rightarrow \text{Check} \ (n+1) \) []
- \(d_0 \rightarrow \text{Check} \ (0) \) []
- \(d_1 \rightarrow \text{Check} \ (0) \) []
- \(a_1 \rightarrow \text{Check} \ (n) \) []
- \(b_0 \rightarrow \text{Check} \ (n) \) []
- \(b_1 \rightarrow \text{Check} \ (n) \) []
- \(c_0 \rightarrow \text{Check} \ (n) \) []
- \(c_1 \rightarrow \text{Check} \ (n) \)

CheckDevice =

Device [ | \{a_0, b_0, c_0, a_1, b_1, c_1, d_0, d_1\} | | Check \ (0)\]

If Check \((0)\) stops, CheckDevice will deadlock.

FDR2 reports CheckDevice is deadlock free.

Therefore, Check \((0)\) never stops (*& the bad thing can't happen*).
**Note:** protocol checking monitors, such as Check, are sometimes used live to ensure adherence at run-time (e.g. in device drivers). We are using Check purely for static analysis – it has no role at run-time and, therefore, no impact on performance.

If Check (0) stops, CheckDevice will deadlock. FDR2 reports CheckDevice is deadlock free.

Therefore, Check (0) never stops (*& the bad thing can’t happen*).
So far, our checks have concerned safety – namely that our system will not do harm (incorrect things). This is not enough! After all, the process does not do incorrect things – it does nothing. STOP trace refines every process. Trace refinement is not enough.

A CSP failure is a state that a system reaches (represented by its trace to that point) where it may refuse to synchronise with its environment on some given set of events.

Process $P$ failure refines $Q$ if (all traces of $P$ are traces of $Q$) and (all failures of $P$ are failures of $Q$).
A **CSP failure** is a state that a system reaches (represented by its trace to that point) where it **may refuse to synchronise** with its environment on some given set of events.

Process $P$ **failure refines** $Q$ if (all its traces are traces of $Q$) and (all its failures are failures of $Q$).

This is a powerful statement! $P$ can only do traces of $Q$ (so its safe). **More:** the failures of $P$ are allowed by $Q$. If $P$ and $Q$ execute the same trace to a state where their environment offers a set of events that $Q$ will not refuse, then $P$ also will not refuse.
A **CSP failure** is a state that a system reaches (represented by its trace to that point) where it may refuse to synchronise with its environment on some given set of events.

Process **P failure refines Q** if (all its traces are traces of Q) and (all its failures are failures of Q).

Whenever Q stays alive (engaging with its environment), so does P (and in the same way). If Q is a specification directly written to express the required patterns of synchronisation, P will fulfil them.
Recall our informal understanding of (at least some of) the opening traces of Device (slides 20-37) …

We can formalise the expression of those traces a bit better …
Recall our informal understanding of (at least some of) the opening traces of Device (slides 20-37) ...

We can formalise the expression of those traces a bit better ...

\[ [a_0, b_0, a_1, b_1] \]
\[ [a_0, a_1, b_0, b_1] \]
\[ [a_0, a_1, b_1, b_0] \]

\[ [c_0] \| | [c_1] \| | [d_0] \]
Recall our informal understanding of (at least some of) the opening traces of Device \((slides 20-37)\) …

We can formalise the expression of those traces a bit better …
Recall our informal understanding of (at least some of) the opening traces of Device (slides 20-37) …

We can formalise the expression of those traces a bit better …

\[
[a0]; ([b0] ||| [a1, b1]); ([c0] ||| [c1] ||| [d0])
\]
And, *still using our intuitive understanding*,
guess the next cycle of events …

We can formalise the expression of
those traces a bit better …

\[ [a_0]; ([b_0] ||| [a_1, b_1]); ([c_0] ||| [c_1] ||| [d_0]);
[a_0]; ([b_0] ||| [a_1, b_1]); ([c_0] ||| [c_1] ||| [d_1]) \]
And, *still using our intuitive understanding*,
guess the next cycle of events …

We can formalise the expression of
those traces a bit better …

And the rest …

\[
\left( [a_0]; ([b_0] || [a_1, b_1]); ([c_0] || [c_1] || [d_0]); \right)^* \\
\left( [a_0]; ([b_0] || [a_1, b_1]); ([c_0] || [c_1] || [d_1]); \right)
\]
Behaviour: **CSP-M (verifyable)**

From such trace expressions, we can directly write down a CSP process that offers all of them to its environment …

This generation can be automated.
In fact, the reverse is also true – they have exactly the same traces and failures.

FDR2 reports Device failure refines CheckDevice. ☺☺☺

DeviceSpec is an explicit specification of all signal patterns we expect (or need) Device to be able to perform.

DeviceSpec =

\[
\begin{align*}
    a0 & \rightarrow (b0 \rightarrow \text{SKIP} ||| a1 \rightarrow b1 \rightarrow \text{SKIP}); \\
    (c0 \rightarrow \text{SKIP} ||| c1 \rightarrow \text{SKIP} ||| d0 \rightarrow \text{SKIP}); \\
    a0 & \rightarrow (b0 \rightarrow \text{SKIP} ||| a1 \rightarrow b1 \rightarrow \text{SKIP}); \\
    (c0 \rightarrow \text{SKIP} ||| c1 \rightarrow \text{SKIP} ||| d1 \rightarrow \text{SKIP}); \\
\end{align*}
\]

DeviceSpec is an explicit specification of all signal patterns we expect (or need) Device to be able to perform.

DeviceSpec ==

\[
\begin{align*}
    a0 & \rightarrow (b0 \rightarrow \text{SKIP} ||| a1 \rightarrow b1 \rightarrow \text{SKIP}); \\
    (c0 \rightarrow \text{SKIP} ||| c1 \rightarrow \text{SKIP} ||| d0 \rightarrow \text{SKIP}); \\
    a0 & \rightarrow (b0 \rightarrow \text{SKIP} ||| a1 \rightarrow b1 \rightarrow \text{SKIP}); \\
    (c0 \rightarrow \text{SKIP} ||| c1 \rightarrow \text{SKIP} ||| d1 \rightarrow \text{SKIP}); \\
\end{align*}
\]
**Behaviour: CSP-M** *(verifyable)*

Device was not implemented as **DeviceSpec** because of the three independent functions (**weapons systems**, **vision processing** and **motion stability**) it had to perform. **Process-oriented design** led to its three communicating sub-systems.
Whilst our intuition indicated that the first two lines of `DeviceSpec` reflected the initial behaviour of `Device`, it was unclear whether the pattern repeated cleanly as its sub-components started looping.
Behaviour: CSP-M (verifyable)

Formal

DeviceSpec =
  a0 -> (b0 -> SKIP |||| a1 -> b1 -> SKIP);
  (c0 -> SKIP |||| c1 -> SKIP |||| d0 -> SKIP);
  a0 -> (b0 -> SKIP |||| a1 -> b1 -> SKIP);
  (c0 -> SKIP |||| c1 -> SKIP |||| d1 -> SKIP); DeviceSpec

One way to ensure this is to add another barrier (bar) at the end of each loop of P0 and P1 and half-loop of P2. The failures equivalence of Device and DeviceSpec shows that the pattern does indeed repeat cleanly and, so, this overhead is not necessary.
DeviceSpec =
  a0 -> (b0 -> SKIP ||| a1 -> b1 -> SKIP);
  (c0 -> SKIP ||| c1 -> SKIP ||| d0 -> SKIP);
  a0 -> (b0 -> SKIP ||| a1 -> b1 -> SKIP);
  (c0 -> SKIP ||| c1 -> SKIP ||| d1 -> SKIP); DeviceSpec

Rather than being deduced after implementation, DeviceSpec may be part of the specification for the behaviour of Device. We certainly need assurance of the behaviour of Device to use it securely with other components. All its patterns of synchronisation (for safety and liveness questions) can be trivially deduced from DeviceSpec.
Reflection

Class experience

The case study presented was developed from one first worked through in a single lesson of a graduate class in concurrency at UNLV in the spring of 2010.

They had previously studied a range of concurrency approaches, including process-oriented material from the Kent “Concurrency Design and Practice” course (presented at last year’s workshop).

They were comfortable with using occam-π in non-trivial projects (thousands of interacting processes), so the example system here would be considered fairly simple.

Nevertheless, it was appreciated that relying just on intuitive understanding is unsafe – especially if the application were safety critical.
Reflection

Class experience

During the exercise, students were given an overview (through examples) of CSP-M syntax, with semantics defined by relating back to occam-π syntax and semantics.

The functional nature of CSP-M, compared with the imperative nature of occam-π, was no particular problem.

Working with FDR2 through its GUI was not very sexy (by modern GUI standards) – but easy enough.

Checking their own (initial) test sequences for Device signals was very simple. Correct confirms/rejects were obtained.

Writing safety-checking processes (like Device) for long term dangers was harder – but they warmed to this with practice.
Class experience

What-ifs on the behaviour of the system could be explored and answered without running any code ... e.g.

If the (internal) ping communications were removed, does Check still hold?  No

Do the a0 and a1 signals strictly alternate?  Yes

Do the b0 and b1 signals strictly alternate?  No

If we added an extra bar sync at the end of each cycle in P0 and P1 and half-cycle in P2, would it make any difference?  No

If the elevator cabin is not at a floor, might the floor doors to the elevator shaft still open?  Another exercise ...
**Reflection**

*occam-π / CSP-M*

*occam-π* teams well with *CSP-M* to provide efficient executables and rich formal analysis.

Of course, it would be better if only one syntactic representation were needed. We are working on extending *occam-π* to include verification assertions (about *deadlock*, *livelock*, *determinism* and *refinement*). Its compiler will generate suitably abstracted *CSP-M* and interact with the *FDR2* model checker, feeding back results in terms of the source *occam-π* program.

Together with the ancient formal *Laws of occam Programming*, this moves *occam-π* towards a process algebra in its own right.

Reflection

Observation

Formal verification of the behaviour of concurrent processes has been achieved – by students – even though they engaged in only simple reasoning themselves.

The complexity of synchronisation and communication analysed went far beyond the embarrassingly parallel.

Aside: model checking found an error overlooked in developing the case study on paper (the need for ping) … which shows the necessity for formal checks (especially when those responsible think they won’t make mistakes!).

Further reading: Santa Claus: Formal Analysis of a Process Oriented Solution.*

* http://doi.acm.org/10.1145/1734206.1734211

TOPLAS, [April, 2010]
Final Observation

Can we teach students (those who love to program, anyway) concurrency so that:

- they quickly develop a correct and intuitive understanding of the primitive mechanisms (e.g. processes, communication, synchronisation, networks) and higher level patterns (e.g. client-server, phased barrier, I/O-PAR) … ?

- they can use those primitives and patterns with the same fluency as they use serial computing primitives, without tripping over dark hazards … ?

- they can develop their own patterns when the standard ones don’t apply … ?

- they can use formal methods to verify good behaviour (e.g. freedom from deadlock and livelock, safety, liveness), without training in the underlying mathematics (process algebra, denotational semantics) … ?

- they can do this as normal everyday practice, without any sense of fear … ?
Final Observation

Can we teach students *(those who love to program, anyway)* concurrency so that:

- they quickly develop a correct and intuitive understanding of the primitive mechanisms *(e.g. processes, communication, synchronisation, networks)* and higher level patterns *(e.g. client-server, phased barrier, I/O-PAR)* …?

- they can use those primitives and patterns with the same fluency as they use serial computing primitives, without tripping over dark hazards …?

- they can develop their own patterns when the standard ones don’t apply …?

- they can use formal methods to verify good behaviour *(e.g. freedom from deadlock and livelock, safety, liveness)*, without training in the underlying mathematics *(process algebra, denotational semantics)* …?

- they can do this as normal everyday practice, without any sense of fear …?

Yes, we can!