Adding Formal Verification to occam-$\pi$

Peter Welch\textsuperscript{a}, Matt Pedersen\textsuperscript{b}, Fred Barnes\textsuperscript{a}, Carl Ritson\textsuperscript{a} and Neil Brown\textsuperscript{a}

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

CPA 2011, University of Limerick, 22\textsuperscript{nd}. June, 2011
**occam-\(\pi\), the process algebra**

**Aim:**

To enable formal verification of \textit{occam-\(\pi\)} programs to be conducted within the language itself ... \textit{as a matter of course by the programmer}.

**How:**

Extend \textit{occam-\(\pi\)} with \textit{verification qualifiers} and \textit{assertions}. Modify the compiler to generate \textit{(minimal)} \textit{CSP}_M code from programs using these qualifiers and assertions, bounce this off the \textit{FDR} model checker and report back in terms of the source code.
Why?

It’s time!

Why?

“Use of autonomous systems will require developing new methods to establish ‘certifiable trust in autonomy’ through Verification and Validation (V&V) of the near-infinite state systems that result from high levels of adaptability; the lack of suitable V&V methods today prevents all but relatively low levels of autonomy from being certified for use … (This) will require access to as-yet undeveloped methods for establishing certifiably reliable V&V.”

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 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.

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

CSP: for formal analysis
[FDR2 model checker + other (simple) formal reasoning.]

OcCam-π: for compiling to a runnable system.
[memory overheads <= 32 bytes per process / synchronisation overheads of order tens of nanoseconds / eats multicore nodes for breakfast.]
**Behaviour: one representation**

- **occam-π**: for compiling to a runnable system.
  
  
  `[memory overheads <= 32 bytes per process / synchronisation overheads of order tens of nanoseconds / eats multicore nodes for breakfast.]`

- **occam-π**: for formal analysis.
  
  
  `[verify qualifiers and (FDR) assertions + other (simple) formal reasoning.]`
**Behaviour: what are we looking for?**

**deadlock:** might it ever stop?
[e.g. P₀ and P₂ want to synchronise on bar, but P₁ wants to ping.]

**livelock:** might it get busy ... but refuse all external signals?
[e.g. P₀, P₁ and P₂ 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-\pi} (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 (will depend on x and y)
      b0 ? z
      SYNC bar    -- wait for the others
      c0 ! 0

::
\end{verbatim}
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 the others

start 0

ping ? x -- receive update

sync bar -- wait for the others

start 0

ping ? x -- receive update

:
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: \texttt{occam-\pi} (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     -- return answer
    ans ! 0    -- wait for 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     -- wait for others
    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}

What patterns of \textit{external (blue)} signalling are possible from \textit{Device}?
Behaviour: **occam-\(\pi\)** (executable)

**Informal Intuitive**

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

\[
\text{WHILE TRUE} \\
\text{INT } x, y, z: \\
\text{SEQ} \\
\quad \text{ask } ? x \quad -- \text{take question} \\
\quad a_0 ? y \quad -- \text{return answer} \\
\quad \text{ans } ! 0 \quad -- \text{return answer} \\
\quad \text{b}_0 ? z \quad -- \text{wait for others} \\
\quad \text{SYNC bar} \quad -- \text{wait for others} \\
\quad \text{c}_0 ! 0 \\
:
\]

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

\[
\text{WHILE TRUE} \\
\text{INT } x, y, z: \\
\text{SEQ} \\
\quad \text{ask } ! 0 \quad -- \text{ask question} \\
\quad \text{ans } ? x \quad -- \text{wait for answer} \\
\quad \text{a}_1 ? y \\
\quad \text{b}_1 ? z \\
\quad \text{SYNC bar} \quad -- \text{wait for the others} \\
\quad \text{c}_1 ! 0 \\
\quad \text{ping } ! 0 \quad -- \text{update neighbour} \\
:\n\]

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

\[
\text{WHILE TRUE} \\
\text{INT } x: \\
\text{SEQ} \\
\quad \text{SYNC bar} \quad -- \text{wait for others} \\
\quad d_0 ! 0 \quad -- \text{wait for others} \\
\quad \text{ping } ? x \quad -- \text{receive update} \\
\quad \text{SYNC bar} \quad -- \text{wait for others} \\
\quad d_1 ! 0 \quad -- \text{receive update} \\
\quad \text{ping } ? x \quad -- \text{receive update} \\
:\n\]

What’s first?
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             -- 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
    \rightarrow ask ! 0     -- ask question
    ans ? x     -- wait for answer
    a1 ? y
    bl ? 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
:\
\end{verbatim}

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

**Informal**

**Intuitive**

\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}

What’s first?

\(<a0>\)
Behaviour: \texttt{occam-\pi} (executable)

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

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

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

\textbf{Behaviour:} \texttt{occam-\pi} (executable)

\textbf{Informal Intuitive}

\textbf{What's second?}

\textbf{<a0>
Behaviour: \textit{occam-π} (executable)

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

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

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

What's second?

<\texttt{a0}>
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

Informal

Intuitive

What's second?

b0 or a1

< a0 >
Behaviour: \textit{occam-\pi} (executable)

**Informal Intuitive**

**Behaviour:**

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

WHILE TRUE

INT $x, y, z$:

SEQ

\begin{align*}
\text{ask } ? & \ x & \text{-- take question} \\
\text{a0 } ? & \ y & \text{-- return answer} \\
\text{ans } ! & \ 0 \\
\text{b0 } ? & \ z \\
\text{SYNC bar} & \text{-- wait for others} \\
\text{c0 } ! & \ 0 \\
\end{align*}

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

WHILE TRUE

INT $x, y, z$:

SEQ

\begin{align*}
\text{ask } ! & \ 0 & \text{-- ask question} \\
\text{ans } ? & \ x & \text{-- wait for answer} \\
\text{a1 } ? & \ y \\
\text{b1 } ? & \ z \\
\text{SYNC bar} & \text{-- wait for the others} \\
\text{c1 } ! & \ 0 \\
\text{ping } ! & \ 0 & \text{-- update neighbour} \\
\end{align*}

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

WHILE TRUE

INT $x$:

SEQ

\begin{align*}
\text{SYNC bar} & \text{-- wait for others} \\
\text{d0 } ! & \ 0 \\
\text{ping } ? & \ x & \text{-- receive update} \\
\text{SYNC bar} & \text{-- wait for others} \\
\text{d1 } ! & \ 0 \\
\text{ping } ? & \ x & \text{-- receive update} \\
\end{align*}

If $b0$ second, then?

\(<a0, b0>\)
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 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
    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?
  a1

< a0, b0, a1 >
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

:

---

**Informal Intuitive**

If b0 second, then?

then a1

then b1

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

\begin{itemize}
\item \textbf{PROC P0} (CHAN INT a0? , b0? , c0! , ask? , ans! ,
\hspace{1cm} \textup{BARRIER bar})
\item \textbf{WHILE TRUE}
\item INT x, y, z:
\item \textbf{SEQ}
\item ask ? x \hfill \text{-- take question}
\item a0 ? y \hfill \text{-- return answer}
\item ans ! 0 \hfill \text{-- return answer}
\item b0 ? z
\item \textbf{SYNC bar} \hfill \text{-- wait for others}
\item c0 ! 0
\end{itemize}

\begin{itemize}
\item \textbf{PROC P2} (CHAN INT d0! , d1! , ping? ,
\hspace{1cm} \textup{BARRIER bar})
\item \textbf{WHILE TRUE}
\item INT x:
\item \textbf{SEQ}
\item d0 ! 0 \hfill \text{-- wait for others}
\item SYNC bar \hfill \text{-- wait for others}
\item ping ? x \hfill \text{-- receive update}
\item d1 ! 0 \hfill \text{-- wait for others}
\item ping ? x \hfill \text{-- receive update}
\end{itemize}

\begin{itemize}
\item \textbf{PROC P1} (CHAN INT a1? , b1? , c1! , ask! , ans? , ping!,
\hspace{1cm} \textup{BARRIER bar})
\item \textbf{WHILE TRUE}
\item INT x, y, z:
\item \textbf{SEQ}
\item ask ! 0 \hfill \text{-- ask question}
\item ans ? x \hfill \text{-- wait for answer}
\item a1 ? y
\item b1 ? z
\item \textbf{SYNC bar} \hfill \text{-- wait for the others}
\item c1 ! 0 \hfill \text{-- update neighbour}
\item ping ! 0
\end{itemize}

\begin{itemize}
\item \textbf{If} \begin{itemize}
\item \begin{itemize}
\item \textbf{b0}
\end{itemize}
\item \begin{itemize}
\item \textbf{second, then?}
\end{itemize}
\item \begin{itemize}
\item \textbf{b1}
\end{itemize}
\item \begin{itemize}
\item then
\end{itemize}
\item \begin{itemize}
\item \textbf{a1}
\end{itemize}
\item \begin{itemize}
\item \textbf{b1}
\end{itemize}
\item \begin{itemize}
\item \textbf{<a0, b0, a1, b1>}
\end{itemize}
\end{itemize}
\end{itemize}
PROCPROC P1P1 ((CHAN INTCHAN INT a1?a1?, , b1?b1?, , c1!c1!, , ask!ask!, , ans?ans?, , ping!ping!,,
BARRIERBARRIER barbar))

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

::

PROCPROC P0P0 ((CHAN INTCHAN INT a0?a0?, , b0?b0?, , c0!c0!, , ask?ask?, , ans!ans!,,
BARRIERBARRIER barbar))

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

::

PROCPROC P2P2 ((CHAN INTCHAN INT d0!d0!, , d1!d1!, , ping?ping?,
BARRIERBARRIER barbar))

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

::

Behaviour: occam-π (executable)

What’s second?

b0  or  a1

InformalIntuitive

backtracking ...
**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 \( a_1 \) second, then?

\(<a_0, a_1>\)
Informal

Intuitive

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

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

\textbf{WHILE TRUE}

\textbf{INT x, y, z:}

\textbf{SEQ}

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

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

\textbf{WHILE TRUE}

\textbf{INT x, y, z:}

\textbf{SEQ}

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

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

\textbf{WHILE TRUE}

\textbf{INT x:}

\textbf{SEQ}

\textbf{SYNC bar} -- wait for others
\textbf{d0 ! 0}
\textbf{ping ? x} -- receive update
\textbf{SYNC bar} -- wait for others
\textbf{d1 ! 0}
\textbf{ping ? x} -- receive update

\begin{itemize}
  \item If \textbf{a1} second, then?
  \item \textbf{b0} and \textbf{b1}*
  \item \textbf{<a0, a1>}
  \item (* any order)
\end{itemize}
Behaviour: \textit{occam-π (executable)}

**Informal Intuitive**

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

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

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

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

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

\[
\text{WHILE TRUE} \\
\text{INT x:} \\
\text{SEQ} \\
\begin{align*}
\text{SYNC bar} & \quad \text{-- wait for others} \\
\text{d0 ! 0} & \quad \text{-- wait for others} \\
\text{ping ? x} & \quad \text{-- receive update} \\
\text{SYNC bar} & \quad \text{-- wait for others} \\
\text{d1 ! 0} & \quad \text{-- receive update} \\
\text{ping ? x} & \quad \text{-- receive update} \\
\end{align*}
\]

\[
\text{If } \begin{cases} \textbf{a1} \text{ second, then?} \\
\end{cases} \\
\begin{cases} \text{b0} \quad \text{and} \quad \text{b1} \\
\end{cases} \\
\begin{cases} \langle a0, a1, b0, b1 \rangle \\
\langle a0, a1, b1, b0 \rangle \\
\end{cases}
\]

Copyright \textcopyright 1989 P. H. Welch and J. B. Pedersen
Behaviour: \texttt{occam-\Pi} (executable)

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

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

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

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

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

```occam
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 \( b1 \)

\( \langle a0, a1, b0, b1 \rangle \)

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

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

\textbf{PROC P1} (CHAN INT \texttt{a1?}, \texttt{b1?}, \texttt{c1!}, \texttt{ask!}, \texttt{ans?}, \texttt{ping!},
\texttt{BARRIER bar})
\begin{verbatim}
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
    cl ! 0
    ping ! 0 -- update neighbour
\end{verbatim}

\textbf{PROC P2} (CHAN INT \texttt{d0!}, \texttt{d1!}, \texttt{ping?},
\texttt{BARRIER bar})
\begin{verbatim}
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}

\begin{itemize}
  \item \texttt{<a0, b0, a1, b1>}
  \item \texttt{<a0, a1, b0, b1>}
  \item \texttt{<a0, a1, b1, b0>}
\end{itemize}

What next?
Behaviour: \texttt{occam-\pi} (executable)

**Informal Intuitive**

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

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

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

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

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

\[
\begin{align*}
\text{WHILE TRUE} & \\
\text{INT } x: & \\
\text{SEQ} & \\
\text{SYNC bar } & \quad \text{-- wait for others} \\
\text{d0 } ! 0 & \quad \text{-- wait for others} \\
\text{ping } ? x & \quad \text{-- receive update} \\
\text{SYNC bar } & \quad \text{-- wait for others} \\
\text{d1 } ! 0 & \quad \text{-- receive update} \\
\text{ping } ? x & \quad \text{-- receive update} \\
\end{align*}
\]

- \(\langle a0, b0, a1, b1 \rangle\)
- \(\langle a0, a1, b0, b1 \rangle\)
- \(\langle a0, a1, b1, b0 \rangle\)

What next?

- \(c0\)
- \(c1\)
- \(d0\)

(* any order)
Behaviour: \texttt{occam-\pi} (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      -- return answer
        ans ! 0     -- wait for others
        b0 ? z      -- wait for others
        SYNC bar
        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
        c1 ! 0
        ping ! 0    -- update neighbour
    :

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

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

What happens when the sub-processes start looping?
Behaviour: \texttt{occam-}\pi (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

Could \texttt{P0} signal \texttt{again} on \texttt{a0} before \texttt{P2} gave its first \texttt{d0}?

Are there some more possible \texttt{first-7} signal sequences?
With **verification qualifiers** and **assertions**, we can ask the **occam-\(\pi\)** compiler to **model check** the previous intuition (which was only about the opening behaviour of the system) and answer the open questions (and more) about its continuous behaviour.

The compiler does this by generating **CSP\(_M\)**, a **declarative** (**functional**) language, from the **occam-\(\pi\)** source and using the **FDR2** model checker.
If we generated $CSP_M$ that fully reflected the semantics of the source code, we would quickly produce a system with too many states for any feasible model checking. For instance, a single $\text{INT}$ variable has $4^G$ possible states!

**By default**, therefore, data values are ignored when generating the $CSP_M$. For instance:

```
PROC P (VAL INT i, CHAN INT c!) 
   c! i 
: 
```

maps just to:  

```
P (c) = c -> SKIP  
```
**Verify Qualifiers: data**

*occam*-π code dependant on tests of *untracked* run-time values map to non-deterministic choice:

```plaintext
PROC Q (VAL INT i, CHAN INT c!, d!) IF i = 42 c! i TRUE d! i :
```

maps to:

```plaintext
Q (c, d) = c -> SKIP |~| d -> SKIP
```
If data values are significant, we qualify their types:

```
PROC Q (VAL VERIFY INT i, CHAN INT c!, d!)
  IF
    i = 42
    c ! i
    TRUE
    d ! i
  ::

Q (i, c, d) =
  if i == 42 then c -> SKIP else d -> SKIP
```

Such data variables are **tracked** and the above now maps to:
Verify Qualifiers: data

If data values are significant, we qualify their types:

```plaintext
PROC Q (VAL VERIFY INT i, CHAN VERIFY INT c!, d!)
  IF
    i = 42
    c! i
    TRUE
    d! i
  :;
```

Such data variables and channel messages are tracked and the above now maps to:

```plaintext
Q (i, c, d) =
  if i == 42 then c!i --> SKIP else d!i --> SKIP
```
**Compiling: occam-π → CSP**

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 -- wait for others
SYNC bar

: c0 ! 0

\[
P0 \ (a0, b0, c0, ask, ans, bar) =
\]

let

\[
P0_0_ = ask \to a0 \to ans \to b0 \to bar \to c0 \to P0_0_
\]

within

\[
P0_0_
\]
Compiling: occam-π → CSP_{M}

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

::

P1 (a1, b1, c1, ask, ans, ping, bar) =
let
P1_0_ = ask → ans → a1 → b1 → bar → cl → ping → P1_0_
within
P1_0_

Formal Compiling: occam-π → CSP_{M}

Compiling: occam-π → CSP_{M}
P2 (d0, d1, ping, bar) =
let
p2_0_ = bar -> d0 -> ping -> bar -> d1 -> ping -> p2_0_
within
p2_0_
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)
::
channel ask_0_, ans_0_, ping_0_, bar_0_

Device (a0, b0, c0, a1, b1, c1, d0, d1) =

let Device_0_ =

  ( P0 (a0, b0, c0, ask_0_, ans_0_, bar_0_)  
    [! {ask_0_, ans_0_, bar_0_} !]  
    P1 (a1, b1, c1, ask_0_, ans_0_, ping_0_, bar_0_) )  
  \ {ask_0_, ans_0_}

within

  ( Device_0_  [! {ping_0_, bar_0_} !]  
    P2 (d0, d1, ping_0_, bar_0_) )  
  \ {ping_0_, bar_0_}
Verify Assertions: \texttt{occam-\pi}

\begin{itemize}
  \item \texttt{VERIFY <assertion>}
  \item \texttt{VERIFY NOT <assertion>}
\end{itemize}

\texttt{<assertion>}

\begin{itemize}
  \item \texttt{DETERMINISTIC.F <process>}
  \item \texttt{DETERMINISTIC.FD <process>}
  \item \texttt{DEADLOCK.FREE.F <process>}
  \item \texttt{DEADLOCK.FREE.FD <process>}
  \item \texttt{LIVELOCK.FREE <process>}
  \item \texttt{TERMINATES <process>}
\end{itemize}

\begin{itemize}
  \item \texttt{<process> REFINES.T <process>}
  \item \texttt{<process> REFINES.F <process>}
  \item \texttt{<process> REFINES.FD <process>}
\end{itemize}

Only \texttt{VAL VERIFY} operands need to be supplied (channels and barriers are supplied automatically)

where \texttt{<process>} is an instance of a \texttt{PROC}
Without testing the system, we can assert straight away that \textbf{Device} is \textit{deterministic} and \textit{free from deadlock} and \textit{livelock} – and that it doesn’t \textit{terminate}:

\begin{verbatim}
VERIFY DETERMINISTIC.FD Device
VERIFY DEADLOCK.FREE.FD Device
VERIFY LIVELOCK.FREE Device
VERIFY NOT TERMINATES Device
\end{verbatim}

and the compiler says: “✔”!
To verify behaviours beyond determinism, deadlock and livelock freedom and termination, we need some way to express the behaviours we want. We can use $\text{occam-}\pi$ for this, together with refinement.

```
VERIFY PROC P (...) ...
```

The $\text{occam-}\pi$ compiler generates only $\text{CSP}_M$ from such declarations –– no executable code.

Within $\text{VERIFY}$ processes, certain restrictions $\text{occam-}\pi$ imposes (currently) can be removed –– for instance, output guards and barrier guards are allowed.

Only $\text{VERIFY}$ processes can invoke $\text{VERIFY}$ processes.
Formal

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

**Intuition**

**Informal understanding**

\(<a_0, b_0, a_1, b_1>\)
\(<a_0, a_1, b_0, b_1>\)
\(<a_0, a_1, b_1, b_0>\)

What next?

\(c_0\) \(c_1\) \(d_0\)

(* any order)
**Formal Behaviour:** \texttt{occam-π} \textit{(verifyable)}

```plaintext
VERIFY PROC T0 (CHAN INT a0?, b0?, c0!, a1?, b1?, c1!, d0!, d1!)
  INT x:
  SEQ
    a0 ? x
    b0 ? x
    a1 ? x
    b1 ? x
    d0 ! 0
    c0 ! 0
    c1 ! 0
  STOP
```

**Informal understanding**

- `<a0, b0, a1, b1>`
- `<a0, a1, b0, b1>`
- `<a0, a1, b1, b0>`
- `<a0, a1, b0, b1>`

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

```
VERIFY T0 REFINES.T Device
```

... which verifies our intuition ☺☺☺☺☺☺

What next?

- `c0`
- `c1` (* any order *)
- `d0`
Formal Behaviour: **occam-π (verifyable)**

```
VERIFY PROC T0 (CHAN INT a0?, b0?, c0!, a1?, b1?, c1!, d0!, d1!)
  INT x:
  SEQ
    a0 ? x
    b0 ? x
    a1 ? x
    b1 ? x
    d0 ! 0
    c0 ! 0
    c1 ! 0
  STOP
: 
```

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

```
VERIFIY T0 REFINES.T Device
```

Informal understanding

```
<a0, b0, a1, b1>
<a0, a1, b0, b1>
<a0, a1, b1, b0>
```

What next?

```
c0  cl  d0
(* any order)
```

\(<a0, b0, a1, b1, d0, c0, c1>\) is clearly a trace of **T0**. Therefore, it is also a trace of **Device**.
**Formal Behaviour:** occam-$\pi$ *(verifyable)*

```
VERIFY PROC T1 (CHAN INT a0?, b0?, c0!, a1?, b1?, c1!, d0!, d1!)
  INT x:
  SEQ
    a0 ? x
    b0 ? x
    a1 ? x
    d0 ! 0
    b1 ? x
    c0 ! 0
    c1 ! 0
  STOP
```

Informal understanding

- `<a0, b0, a1, b1>`
- `<a0, a1, b0, b1>`
- `<a0, a1, b1, b0>`
- `<a0, a1, b0, b1>`

What next?

- `c0`
- `c1`
- `d0`

(* any order*)

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

```
VERIFY T1 REFINES.T Device
```

... which verifies our intuition ☺☺☺

24-Jun-11 Copyleft (GPL) P.H.Welch and J.B.Pedersen 54
**Formal Behaviour: **\textit{occam-\pi} (verifyable)

\begin{verbatim}
VERIFY PROC T1 (CHAN INT a0?, b0?, c0!, a1?, b1?, c1!, d0!, d1!)
  INT x:
  SEQ
    a0 ? x
    b0 ? x
    a1 ? x
    \textcolor{red}{d0 ! 0}
    b1 ? x
    c0 ! 0
    c1 ! 0
    STOP
\end{verbatim}

Informal understanding

\begin{itemize}
  \item \textcolor{red}{<a0, b0, a1, b1>}
  \item \textcolor{red}{<a0, a1, b0, b1>}
  \item \textcolor{red}{<a0, a1, b1, b0>}
\end{itemize}

What next?

\begin{itemize}
  \item c0
  \item c1
  \item \textcolor{red}{d0} (* any order)
\end{itemize}

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

\begin{verbatim}
VERIFY T1 REFINES.T Device
\end{verbatim}

At least one trace of \textcolor{red}{T1} is \textit{not} a trace of \textit{Device}. Comparing \textcolor{red}{T0} and \textcolor{red}{T1}, the fault lies in the mis-ordering of \textcolor{red}{d0} and \textcolor{red}{b1}. 

24-Jun-11
Copyleft (GPL) P.H.Welch and J.B.Pedersen
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 accept a signal twice on a0 without a signal on d0 or d1 in between. Might this ever happen?

Simple: write a process that checks all signals to/from Device, looking for the bad scenario and deliberately deadlocks (the monitored system) if spotted. This is just programming …
**Behaviour:** \textit{occam-π} (verifyable)

**Formal**

**Safety**

**Simple:** write a process that checks all signals to/from \textit{Device}, looking for the bad scenario and deliberately deadlocks (the monitored system) if spotted. This is just programming …
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 **Device** were ever to signal **twice** on **a0** without a signal on **d0** or **d1 in between**. Might this **ever** happen?

**Simple:** write a process that checks all signals to/from **Device**, looking for the bad scenario and deliberately deadlocks (the monitored system) if spotted. This is just programming …
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 Device were ever to signal twice on a0 without a signal on d0 or d1 in between. Might this ever happen?
Formal Behaviour: \texttt{occam-\pi} (verifyable)

```plaintext
{{{{ process next signal (maintain n) }}}}
INT x:
ALT
    a0 ! 0
    n := n + 1
    b0 ! 0
    SKIP
    c0 ? x
    SKIP
    a1 ! 0
    SKIP
    b1 ! 0
    SKIP
    c1 ? x
    SKIP
    d0 ? x
    n := 0
    d1 ? x
    n := 0
}}}
```

- \( n = \) the number of \( a0 \) signals received since the last \( d0 \) or \( d1 \)
- This is an \texttt{ALT} with four input and four output guards
**Formal Behaviour**: \textit{occam-\pi} (verifyable)

**Safety Check**

```
VERIFY PROC CheckDevice (CHAN INT alive!)
  CHAN INT a0, a1, b0, b1, c0, c1, d0, d1:
  PAR
    Check (a0!, b0!, c0?, a1!, b1!, c1?, d0?, d1?, alive!)
    Device (a0?, b0?, c0!, a1?, b1?, c1!, d0!, d1!)
:
```
Formal Behaviour: \textit{occam-\pi} (verifyable)

\begin{itemize}
\item Check
\item Device
\end{itemize}

If \textbf{Check} stops, \textbf{CheckDevice} will deadlock. Therefore, \textbf{Check} never stops \ldots and the bad thing can't happen.

\textbf{Q.E.D.}
Formal Behaviour: occam-π (verifyable)

Safety

CheckDevice

Check

Device

alive

VERIFY DEADLOCK.FREE.FD CheckDevice

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 is not there at run-time and, therefore, has no impact on performance.
So far, our checks have concerned safety—namely that our system will not do harm (incorrect things). This is not enough! After all, the STOP 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).
Failure refinement makes a powerful statement! Process $P$ can only do traces of $Q$ (so it is 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 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 traces of **P** are traces of **Q**) and (all failures of **P** are failures of **Q**).

We can describe "**P failure refines Q**" in a positive way: whenever **Q** stays alive (engaging with its environment), so does **P** (and in the same way). So, if **Q** is a specification explicitly defining the required patterns of synchronisation, **P** will provide them.
Recall our informal understanding of (at least some of) the opening traces of Device (slides 18-35) …

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 18-35) …

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 18-35) …

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 18-35) …

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

\[
\langle a_0 \rangle; (\langle b_0 \rangle ||| \langle a_1, b_1 \rangle); (\langle c_0 \rangle ||| \langle c_1 \rangle ||| \langle d_0 \rangle)
\]
We can formalise the expression of
those traces a bit better …

And, still using our intuitive understanding,
guess the next cycle of events …

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

\begin{align*}
\langle a0 \rangle; (\langle b0 \rangle ||| \langle a1, b1 \rangle); (\langle c0 \rangle ||| \langle c1 \rangle ||| \langle d0 \rangle); \\
\langle a0 \rangle; (\langle b0 \rangle ||| \langle a1, b1 \rangle); (\langle c0 \rangle ||| \langle c1 \rangle ||| \langle d1 \rangle)
\end{align*}
We can formalise the expression of those traces a bit better …

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 …
From such trace expressions, we can directly write down an \textit{occam-\pi} process that offers all of them …
From such trace expressions, we can directly write down an \textit{occam-$\pi$} process that offers all of them ...
From such trace expressions, we can directly write down an \textit{occam-\pi} process that offers all of them …

\[
\langle \text{a0} \rangle; (\langle \text{b0} \rangle ||| \langle \text{al}, \text{bl} \rangle); (\langle \text{c0} \rangle ||| \langle \text{cl} \rangle ||| \langle \text{d0} \rangle); \rangle^* 
\]

{{{{
\begin{verbatim}
{{{{  phase 0
  SEQ
    a0 ? w
  PAR
    b0 ? x
    SEQ
      a1 ? y
      b1 ? z
    PAR
    c0 ! 0
    c1 ! 0
    d0 ! 0
}}}
\end{verbatim}
}}}}}
From such trace expressions, we can directly write down an \texttt{occam-\pi} process that offers all of them …

\[
\begin{align*}
&\langle a_0 \rangle; (\langle b_0 \rangle ||| \langle a_1, b_1 \rangle); (\langle c_0 \rangle ||| \langle c_1 \rangle ||| \langle d_0 \rangle); \\
&\langle a_0 \rangle; (\langle b_0 \rangle ||| \langle a_1, b_1 \rangle); (\langle c_0 \rangle ||| \langle c_1 \rangle ||| \langle d_1 \rangle)
\end{align*}
\]

\textit{This generation can be automated.}
**DeviceSpec** is an explicit specification of all signal patterns we expect (or need) **Device** to be able to perform:

\[
\left( \langle a_0 \rangle; (\langle b_0 \rangle || \langle a_1, b_1 \rangle); (\langle c_0 \rangle || \langle c_1 \rangle || \langle d_0 \rangle); \right)^* \\
\left( \langle a_0 \rangle; (\langle b_0 \rangle || \langle a_1, b_1 \rangle); (\langle c_0 \rangle || \langle c_1 \rangle || \langle d_1 \rangle) \right)
\]

**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.
**Formal Behaviour:** \texttt{occam-π (verifyable)}

```
VERIFY PROC DeviceSpec (CHAN INT a0?, b0?, c0!, a1?, b1?, c1!, d0!, d1!)
...
```

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

```
(<a0>; (<b0> ||| <a1, b1>); (<c0> ||| <c1> ||| <d0>);)*
(<a0>; (<b0> ||| <a1, b1>); (<c0> ||| <c1> ||| <d1>))
```

However:

```
VERIFY Device REFINES.FD DeviceSpec
```

This is all we need. Any traces performed by **Device** are allowed by **DeviceSpec** – so it’s safe. Any failures reached by **Device** are allowed by **DeviceSpec** – so it’s as "alive" as **DeviceSpec** (which was built always to offer everything in the specified trace pattern).
**Formal Behaviour:** \texttt{occam-\pi\, (verifyable)}

\begin{verbatim}
VERIFY PROC DeviceSpec (CHAN INT a0?, b0?, c0!, a1?, b1?, c1!, d0!, d1!)

::

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

\[
\begin{align*}
(<a_0>; (b_0 || <a_1, b_1>); (<c_0> || <c_1> || <d_0>);)^* \\
(<a_0>; (b_0 || <a_1, b_1>); (<c_0> || <c_1> || <d_1>))
\end{align*}
\]

However:

\[
\text{VERIFY Device REFINES.FD DeviceSpec}
\]

Without this verification, we may be tempted to add another barrier (\texttt{bar}) \texttt{sync} at the end of each loop of \texttt{P0} and \texttt{P1} and half-loop of \texttt{P2}. The above \textit{refinement} shows that the required pattern does indeed repeat cleanly and, so, this overhead is unnecessary.
Formal behaviour: \textit{occam-\pi} (verifyable)

\begin{verbatim}
VERIFY PROC DeviceSpec (CHAN INT a0?, b0?, c0!, a1?, b1?, c1!, d0!, d1!)
  ...
:

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

\[
\langle a_0 \rangle; (\langle b_0 \rangle ||| \langle a_1, b_1 \rangle); (\langle c_0 \rangle ||| \langle c_1 \rangle ||| \langle d_0 \rangle); (\langle a_0 \rangle; (\langle b_0 \rangle ||| \langle a_1, b_1 \rangle); (\langle c_0 \rangle ||| \langle c_1 \rangle ||| \langle d_1 \rangle)\star
\]

However:

\begin{verbatim}
VERIFY DeviceREFINES.FD DeviceSpec
\end{verbatim}

\checkmark

Rather than being deduced after implementation, DeviceSpec may be part of the specification for 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.
**Formal Behaviour:** \textit{occam-\pi} (verifyable)

\begin{verbatim}
VERIFY PROC DeviceSpec (CHAN INT a0?, b0?, c0!, a1?, b1?, c1!, d0!, d1!)

::

DeviceSpec is an explicit specification of all signal patterns we expect (or need) \textit{Device} to be able to perform:

\begin{verbatim}
(<a0>; (<b0> || <a1, b1>); (<c0> || <c1> || <d0>);)
\end{verbatim}

However:

\begin{verbatim}
VERIFY Device REFINES.FD DeviceSpec
\end{verbatim}

We also have:

\begin{verbatim}
VERIFY DeviceSpec REFINES.FD Device
\end{verbatim}

But that’s just icing on the cake! ☺☺☺
For simplicity, most process arguments are omitted in VERIFY assertions – the occam-π compiler supplies all necessary events:

```
VERIFY DEADLOCK.FREE.FD Device

channel a0_42_, b0_42_, c0_42_, a1_42_,
     b1_42_, c1_42_, d0_42_, d1_42_

assert Device (a0_42_, b0_42_, c0_42_, a1_42_,
          b1_42_, c1_42_, d0_42_, d1_42_)
  : [ deadlock free [FD] ]
```

The CSP\textsubscript{M} channel names are generated from the occam-π CHAN and BARRIER parameter names of the asserted process, suffixed by a unique number generated by the compiler.
For simplicity, most process \textit{arguments} are omitted in \texttt{VERIFY} assertions – the \texttt{occam-\pi} compiler supplies all necessary \textit{events}:

\begin{verbatim}
VERIFY NOT TERMINATES Device

assert not SKIP [FD=
Device (a0_42_, b0_42_, c0_42_, a1_42_,
b1_42_, c1_42_, d0_42_, d1_42_) \ Events
\end{verbatim}

The $CSP_M$ channel names are generated from the \texttt{occam-\pi} \texttt{CHAN} and \texttt{BARRIER} parameter names of the asserted process, suffixed by a unique number generated by the compiler.
For simplicity, most process arguments are omitted in `VERIFY` assertions – the `occam-π` compiler supplies all necessary events:

```
VERIFY NOT TERMINATES Device

assert not SKIP [FD=
    Device (a0_42_, b0_42_, c0_42_, a1_42_,
    b1_42_, c1_42_, d0_42_, d1_42_) \ Events
```

Subsequent assertions about the same process may reuse channels previously generated.
For simplicity, most process arguments are omitted in VERIFY assertions – the occam-π compiler supplies all necessary events:

```
VERIFY Device REFINES.FD DeviceSpec

assert DeviceSpec (a0_42_, b0_42_, c0_42_, a1_42_,
                  b1_42_, c1_42_, d0_42_, d1_42_)

[FD=
 Device (a0_42_, b0_42_, c0_42_, a1_42_,
         b1_42_, c1_42_, d0_42_, d1_42_)
```

Subsequent assertions about the same process may reuse channels previously generated. [Note: processes in refinement assertions should have the same parameter signatures, though the formal names can be different].
The only arguments needed for $\text{CSP}_M$ assertions are those for $\text{occam-}\pi$ $\text{VERIFY}$ data parameters. $\text{Channels}$ and $\text{barriers}$ can be supplied automatically. Non-$\text{VERIFY}$ data parameters are irrelevant.

For example, if we need an assertion about:

\begin{verbatim}
PROC System (VAL VERIFY INT n, CHAN VERIFY INT out!)
\end{verbatim}

we must supply a value for $n$, since we have declared it relevant:

\begin{verbatim}
VERIFY DEADLOCK.FREE.FD System (42, _)
\end{verbatim}

where the underscore indicates arguments that are either irrelevant ($\text{non-}\text{VERIFY}$ data) or automatic ($\text{channels}$ and $\text{barriers}$).
Later, we plan an option for the \texttt{occam-\pi} compiler just to generate \texttt{CSP}_M code to be picked up by a \texttt{GUI} with facilities for interactive generation, checking and reporting of \texttt{VERIFY} assertions. These will be similar to those given by the \texttt{FDR2 GUI}, but processes and assertions will be in terms of the \texttt{occam-\pi} sources. \texttt{FDR2}, or some derivative, remains the underlying workhorse for model checking.

The \texttt{GUI} will allow flexible exploration of assertions with \texttt{VERIFY} data values. It will also prove useful when some assertions take a long time to check … rather than wait for all checks to complete during compilation (as a single batch of assertions to \texttt{FDR2}).
Reflection on Case Study (Device)

Further study:

All sorts of *what-ifs* on the behaviour of the system can 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 …**
The story of The Dining Philosophers is due to Edsger Dijkstra – one of the founding fathers of Computer Science.

It illustrates a classic problem in concurrency: how to share resources safely between competing consumers.

http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD310.PDF

Historical document
A new, really really neat, solution (Neil Brown / PHW)
VERIFICATION PROTOCOL Phil (CHAN INT thinking!, eating!, BARRIER eatBar)
  WHILE TRUE
    SEQ
      thinking ! 0
      SYNC eatBar
      eating ! 0
      SYNC eatBar
  :

Phil (thinking, eating, eatBar) =
  let
    Phil_0_ =
      thinking -> eatBar ->
      eating -> eatBar -> Phil_0_
  within
    Phil_0_
  :

VERIFY PROC Fork (BARRIER eatBarRight, eatBarLeft)
  WHILE TRUE
  ALT
    SYNC eatBarRight
    SYNC eatBarRight
    SYNC eatBarLeft
    SYNC eatBarLeft
  ::

Fork (eatBarRight, eatBarLeft) =
  let
    Fork_0_ =
      eatBarRight -> eatBarRight -> Fork_0_ [ ]
      eatBarLeft -> eatBarLeft -> Fork_0_
    within
      Fork_0_ 
  :
VAL INT nPhil IS 5:

nPhil = 5

VERIFY PROC Philosophers ([nPhil]CHAN INT thinking!, eating!, [nPhil]BARRIER eatBar)

PAR id = 0 FOR nPhil
    Phil (thinking[id]!, eating[id]!, eatBar[id])

Philosophers (thinking, eating, eatBar) =
    ||| id : {0..(nPhil - 1)} @
    Phil (thinking.id, eating.id, eatBar.id)
... except that FDR2 uses *much less memory and time* if replicated (or merely repeated) processes take *no parameters*, but instead use *event renaming* to wire up the different instances.

```haskell
channel thinking_r0_, eating_r0_, eatBar_r0_

Philosophers (thinking, eating, eatBar) =
  let
    Philosophers_0 = Phil (thinking_r0_, eating_r0_, eatBar_r0_)
  within
    id : {0..(nPhils - 1)} @
    Philosophers_0 [[
      thinking_r0_ <- thinking.id,
      eating_r0_ <- eating.id,
      eatBar_r0_ <- eatBar.id
    ]]
```

*Note: the three declared channels are not actually used!!*
VAL INT nPhils IS 5:

nPhils = 5

VERIFY PROC Forks ([nPhils]BARRIER eatBar)
  PAR id = 0 FOR nPhils
    VAL INT right IS id:
    VAL INT left IS (id + 1)\nPhils:
    Fork (eatBar[right], eatBar[left])

Forks (eatBar) =
  || id : {0..(nPhils - 1)} @
    [{ eatBar.id, eatBar.((id + 1)%nPhils) }]
    Fork (eatBar.id, eatBar.((id + 1)%nPhils))
... except that **FDR2** uses *much less memory and time* if replicated (or merely repeated) processes take *no parameters*, but instead use *event renaming* to wire up the different instances.

```plaintext
channel eatBarRight_r2_, eatBarLeft_r2_

Forks (eatBar) =
  let
    Forks_0 = Fork (eatBarRight_r2_, eatBarLeft_r2_)
  within
  || id : {0..(nPhils - 1)} @
  [{ eatBar.id, eatBar.(id + 1)%nPhils }]
  Forks_0 [[
    eatBarRight_r2_ <- eatBar.id,
    eatBarLeft_r2_ <- eatBar.(id + 1)%nPhils
  ]]

Note: the two declared channels are not actually used !!
```
VAL INT nPhilhs IS 5:

nPhilhs = 5

VERIFY PROC College ([nPhilhs]CHAN INT thinking!, eating!)
   [nPhilhs]BARRIER eatBar:
   PAR
      Philosophers (thinking!, eating!, eatBar)
      Forks (eatBar)
   :

channel eatBar_99_ : {0..(nPhilhs - 1)}

College (thinking, eating) =
  (Philosophers (thinking, eating, eatBar_99_) [\{|| eatBar_99_ |} |]
  Forks (eatBar_99_)) \ {\{| eatBar_99_ |}
  :
VERIFY PROC College ([nPhils]CHAN INT thinking!, eating!)
   [nPhils]BARRIER eatBar:
   PAR
   Philosophers (thinking!, eating!, eatBar)
   Forks (eatBar)

VERIFY DEADLOCK.FREE.FD College ✓
VERIFY LIVELOCK.FREE College ✓ 😊 😊 😊
VERIFY NOT DETERMINISTIC.FD College ✓
The previous model check verifies properties of a college with precisely 5 philosophers. The FDR2 model check is almost instant.

Scaling to 10 philosophers puts a strain on my laptop – it gets very hot and takes a few minutes. Scaling to 20 fails.

In the FDR2 manual, Bill Roscoe explains how to verify a college with 10^200 philosophers … we had better follow his guidelines … and tackle the black art of compression in model checking …

With our simpler college, we want to beat that scale! Further, we would like to verify a college of any number of philosophers … using induction.
The first guideline is not to build the philosophers and forks as separate sub-systems and, then, the college as their parallel combination. This is what we did and it doesn’t let us use inductive reasoning very well.

Instead, first build a philosopher-fork pair. Next, build chains of philosopher-fork pairs using recursion (e.g. a chain of length $n$ is a chain of length $(n-1)$ plus one more pair). Verify properties of the chain, for any $n$. Finally, add one more pair that connects both ends of a chain and get the college. Verify the college using verified properties of the chain.
There are two further points that are needed: *hiding* and *compression*.

First, note that the *thinking* and *eating* reports from the *philosophers* play no role in the deadlock / livelock properties of the *college*. Each philosopher engages on its own *thinking* and *eating* channels with the environment of the *college*. The forks do not engage with those channels.

Therefore, no *thinking* or *eating* report can block the operations of the *college*. Verifying deadlock and livelock freedom in a college with the *thinking* and *eating* events *hidden* will also verify the result for a college that doesn’t hide them.
VERIFY PROC Phil (CHAN INT thinking!, eating!, BARRIER eatBar)
   WHILE TRUE
   SEQ
   thinking ! 0
   SYNC eatBar
   eating ! 0
   SYNC eatBar
   :

Phil' (thinking, eating, eatBar) =
   let
   Phil_0_ =
      thinking -> eatBar ->
      eating -> eatBar -> Phil_0_
   within
   Phil_0_
   :

Recall …
VERIFY PROC Phil. (BARRIER eatBar)
  CHAN INT thinking!, eating!:
  Phil (thinking!, eating!, eatBar)

channel thinking_h0_, eating_h0_

Phil' (eatBar) =
  Phil (thinking_h0_, eating_h0_, eatBar)
  \ { | thinking_h0_, eating_h0_ |}
**Which is the same as ...**

```
VERIFY PROC Phil. (BARRIER eatBar)
   WHILE TRUE
   SEQ
     SYNC eatBar
     SYNC eatBar
   :
```

Which is the same as …

Phils’ (eatBar) =
    let
      Phil_0_ = eatBar -> eatBar -> Phil_0_
    within
      Phil_0_
    :

…but without changing source code 😊
We can ask for the size of the state transition machine generated by FDR …

```
VERIFY PROC Phil. (BARRIER eatBar)
  CHAN INT thinking!, eating!:
    Phil (thinking!, eating!, eatBar)
:
VERIFY SIZE Phil
VERIFY SIZE Phil.
```

4 states, 4 transitions

… not won yet … need to compress!!
VERIFY PROC Phil. (BARRIER eatBar)
NORMALISE -- reduce state machine to normal form
Phil. (eatBar)
:
VERIFY SIZE Phil..

... a big win!! Adding such a (non-reporting, compressed) philosopher to any system cannot increase the number of states.
And is the same as ...

VERIFY PROC Phil.. (BARRIER eatBar)
  WHILE TRUE
    SYNC eatBar
:

Phil’’ (eatBar) =
  let
    Phil_0_ = eatBar -> Phil_0_
  within
    Phil_0_
:

... but without changing source code 😊
VERIFY PROC Fork (BARRIER eatBarRight, eatBarLeft)
  WHILE TRUE
  ALT
    SYNC eatBarRight
    SYNC eatBarRight
    SYNC eatBarLeft
    SYNC eatBarLeft

VERIFY SIZE Fork

3 states, 4 transitions
VERIFY PROC PhilFork (CHAN INT thinking!, eating!, BARRIER eatBarRight, eatBarLeft)

PAR

    Phil (thinking!, eating!, eatBarRight)
    Fork (eatBarRight, eatBarLeft)

:

VERIFY SIZE PhilFork

\(6\) states, \(9\) transitions
VERIFY PROC PhilFork. (BARRIER eatBarRight, eatBarLeft)
  PAR
  Phil.. (eatBarRight)
  Fork (eatBarRight, eatBarLeft)
:
VERIFY SIZE PhilFork.

VERIFY PhilFork. EQUIVALENT.FD Fork

3 states, 4 transitions

... PhilFork. is the same as Fork 😊
Now build a chain … using recursion

\[
\text{VERIFY PROC } \text{Chain} \ (\text{VAL VERIFY INT length}, \quad -- \ \text{assume } \geq 1 \\
\quad \text{BARRIER eatBarRight, eatBarLeft})
\]

\[
\text{IF} \\
\quad \text{length} = 1 \\
\quad \text{PhilFork.} \ (\text{eatBarRight, eatBarLeft}) \\
\quad \text{TRUE} \\
\quad \text{NORMALISE} \\
\quad \text{BARRIER eatBarMiddle:} \\
\quad \text{PAR} \\
\quad \text{Chain} \ (\text{length} - 1, \text{eatBarRight, eatBarMiddle}) \\
\quad \text{PhilFork.} \ (\text{eatBarMiddle, eatBarLeft})
\]

Generating the \(\text{CSP}_M\) code for this requires an extra care … \\
(because \(\text{FDR2}\) does something it shouldn’t – claim!)

The following does not work correctly …
Now build a chain ... using recursion

```plaintext
channel eatBarMiddle

Chain (length, eatBarRight, eatBarLeft)
  if length == 1 then
    PhilFork' (eatBarRight, eatBarLeft)
  else
    normalise (
      ( Chain (length - 1, eatBarRight, eatBarMiddle)
        [ | {eatBarMiddle} | ]
      PhilFork' (eatBarMiddle, eatBarLeft)
    ) \ {eatBarMiddle}
  )

:;
```

eatBarMiddle events “hidden” inside the recursive instance of Chain get confused with the eatBarMiddle connecting that instance with PhilFork’. 😞 😞 😞

We have to manufacture lots of eatBarMiddle events ...
Now build a chain … using recursion

```
channel eatBarMiddle : Int

Chain (length, eatBarRight, eatBarLeft)
  if length == 1 then
    PhilFork' (eatBarRight, eatBarLeft)
  else
    normalise ( (Chain (length - 1, eatBarRight, eatBarMiddle)
      [\ {eatBarMiddle.length}\]
    PhilFork' (eatBarMiddle, eatBarLeft))
    \ {eatBarMiddle.length}
  )
: 
```

… and use a different one for each length. Now we are OK! ☺ ☺ ☺

But it really should not be up to us to declare and use this infinite set of hidden events. Why doesn’t FDR2 just rename hidden events to unique names that cannot be expressed by the FDR2 coder? Not doing so seems to break the semantics of hiding … ???
What’s happening with the sizes?

VERIFY PROC Chain (VAL VERIFY INT length, -- assume >= 1
                  BARRIER eatBarRight, eatBarLeft)

  IF
    length = 1
    PhilFork. (eatBarRight, eatBarLeft)
  TRUE
  NORMALISE
    BARRIER eatBarMiddle:
    PAR
      Chain (length - 1, eatBarRight, eatBarMiddle)
      PhilFork. (eatBarMiddle, eatBarLeft)

:  

VERIFY SIZE Chain (1, _, _)  -->  3 states, 4 transitions
VERIFY SIZE Chain (2, _, _)  -->  1 state, 2 transitions
VERIFY SIZE Chain (3, _, _)  -->  1 state, 2 transitions
VERIFY SIZE Chain (4, _, _)  -->  1 state, 2 transitions
How similar are they and might they deadlock?

```plaintext
VERIFY PROC Chain (VAL VERIFY INT length,          -- assume >= 1
    BARRIER eatBarRight, eatBarLeft)

IF
    length = 1
    PhilFork. (eatBarRight, eatBarLeft)
    TRUE
    NORMALISE
        BARRIER eatBarMiddle:
        PAR
            Chain (length - 1, eatBarRight, eatBarMiddle)
            PhilFork. (eatBarMiddle, eatBarLeft)

:
```

From Chain (2, _, _) upwards, they can certainly *livelock* – infinite sequences of *eatBarMiddle* events!

So, deadlock and refinement checking must only be done with the *failures* model.
How similar are they and might they deadlock?

```
VERIFY PROC Chain (VAL VERIFY INT length,          -- assume >= 1
BARRIER eatBarRight, eatBarLeft)

::

VERIFY DEADLOCK.FREE.F Chain (1, _, _) ✔
VERIFY DEADLOCK.FREE.F Chain (2, _, _) ✔
VERIFY DEADLOCK.FREE.F Chain (3, _, _) ✔
VERIFY DEADLOCK.FREE.F Chain (4, _, _) ✔
VERIFY DEADLOCK.FREE.F Chain (5, _, _) ✔

VERIFY Chain (1, _, _) EQUIVALENT.F Chain (2, _, _) ✗
VERIFY Chain (2, _, _) EQUIVALENT.F Chain (3, _, _) ✗
VERIFY Chain (3, _, _) EQUIVALENT.F Chain (4, _, _) ✗
VERIFY Chain (4, _, _) EQUIVALENT.F Chain (5, _, _) ✔
```

Let \(H(i)\) be the hypothesis that:

```
Chain (4, _, _) EQUIVALENT.F Chain (i, _, _)
```

Clearly \(H(4)\) and, by model checking, \(H(5)\).
We have $H(4)$ and $H(5)$. Suppose $H(i)$ for any $i \geq 4$. Consider:

$$H(i+1)$$

This reduces to:

$$BARRIER \text{ eatBarMiddle:}$$
$$PAR$$
$$Chain \ (i, \text{ eatBarRight, eatBarMiddle})$$
$$PhilFork. \ (\text{eatBarMiddle, eatBarLeft})$$

By $H(i)$, this is $EQUIVALENT \cdot F$ to:

$$BARRIER \text{ eatBarMiddle:}$$
$$PAR$$
$$Chain \ (4, \text{ eatBarRight, eatBarMiddle})$$
$$PhilFork. \ (\text{eatBarMiddle, eatBarLeft})$$

But this is the same as:

$$Chain \ (5, \text{ eatBarRight, eatBarLeft})$$

Which, by $H(5)$, is $EQUIVALENT \cdot F$ to:

$$Chain \ (4, \text{ eatBarRight, eatBarLeft})$$
Clearly $H(4)$ and, by model checking, $H(5)$.

We have just shown that, for any $i \geq 4$, $H(i)$ implies $H(i+1)$.

By induction therefore, for all $i \geq 4$, we have $H(i)$.

All chains of (no reporting) philosopher-fork pairs with lengths equal to or greater than 4 are failures equivalent. Further, all such chains are deadlock free (since model checking gave us that for chains of lengths 1 through 4).

But … what about Colleges?
But … what about *Colleges*?

```plaintext
VERIFY PROC CollegeChain (VAL VERIFY INT size) -- assume >= 2
NORMALISE
[2]BARRIER eatBar:
    PAR
        PhilFork. (eatBar[0], eatBar[1])
        Chain (size - 1, eatBar[1], eatBar[0])
:
```

We can immediately deduce that all `CollegeChains` with size equal to or greater than 5 are *failures equivalent* (since their `Chain` sub-components have lengths equal to or greater than 4 and are *failures equivalent*).

```plaintext
VERIFY DEADLOCK.FREE.F CollegeChain (2, _, _) ✔
VERIFY DEADLOCK.FREE.F CollegeChain (3, _, _) ✔
VERIFY DEADLOCK.FREE.F CollegeChain (4, _, _) ✔
VERIFY DEADLOCK.FREE.F CollegeChain (5, _, _) ✔
```

Hence, all `CollegeChains` with size equal to or greater than 2 are *deadlock free*. Of course, with no reporting, they are hopelessly *livelocked*!
So … what about reporting Colleges?

An earlier argument showed that a deadlock free result for a college with external reports hidden implies a deadlock free result for a college with external reports (since the external reporting cannot cause internal blocking). So all reporting colleges of any size are deadlock-free.

The following argument shows that a college with external reports is also livelock free …
From simple code inspection, a Phil process cannot engage in two eatBar events (internal) without an (external) intervening report.

This could be model-checked, using techniques discussed earlier, if it was felt necessary!

```
VERIFY PROC Phil (CHAN INT thinking!, eating!, BARRIER eatBar)
  WHILE TRUE
  SEQ
    thinking ! 0
    SYNC eatBar
    eating ! 0
    SYNC eatBar
: 
```
From simple code inspection, a Phil process cannot engage in two eatBar events (internal) without an (external) intervening report.

This could be model-checked, using techniques discussed earlier, if it was felt necessary!

For the college not to be *livelock free* ... it must be possible for it to engage in an infinite sequence of internal events ... and the only internal events are eatBar s. Suppose that this happens!

If the college has size $n$, it has only $n$ eatBars. After at most $(n+1)$ eatBar events, at least one must have occurred at least twice. But the Phil process engaging with that eatBar must (by the above) have made an external report ... so the college is not livelocked.

This is a contradiction! So the supposition is false – and the college is *livelock free*. 
Finally, the Brute Force Approach

In Roscoe’s book, chains are not built up one-at-a-time like this (possibly because the standard dining philosophers solution analysed does not collapse as nicely as this one, when reporting is hidden?). Instead, they are built up in powers of 10. We can do this too:

```verbatim
/* A chain of (length^level) philosopher-fork pairs. */
VERIFY PROC Chain2 (VAL VERIFY INT level, length,
                   BARRIER eatBarRight, eatBarLeft)
  IF level = 0
    PhilFork. (eatBarRight, eatBarLeft)
    TRUE
    NORMALISE
    [length-1]BARRIER eatBar:
    PAR
    Chain2 (level - 1, length, eatBarRight, eatBar[0])
    PAR id = 1 FOR length - 2
    Chain2 (level - 1, length, eatBar[id - 1], eatBar[id])
    Chain2 (level - 1, length, eatBar[length - 2], eatBarLeft)
  : 
```
--- A chain of \((\text{length}^{\text{level}})\) philosopher-fork pairs.

\begin{verbatim}
VERIFY PROC Chain2 (VAL VERIFY INT level, length,
                   BARRIER eatBarRight, eatBarLeft)
...

VERIFY DEADLOCK.FREE.F Chain2 (0, 10, _, _) ✓
VERIFY DEADLOCK.FREE.F Chain2 (1, 10, _, _) ✓
VERIFY DEADLOCK.FREE.F Chain2 (10, 10, _, _) ✓
VERIFY DEADLOCK.FREE.F Chain2 (100, 10, _, _) ✓
VERIFY DEADLOCK.FREE.F Chain2 (1000, 10, _, _) ✓
VERIFY Chain2 (1, 2, _, _) EQUIVALENT.F Chain2 (2, 2, _, _) X
VERIFY Chain2 (2, 2, _, _) EQUIVALENT.F Chain2 (3, 2, _, _) ✓
VERIFY Chain2 (1, 10, _, _) EQUIVALENT.F Chain2 (2, 10, _, _) ✓
\end{verbatim}

Finally, the Brute Force Approach

And the Colleges ...

Finally, the Brute Force Approach

```latex
\begin{verbatim}
\texttt{---\star A college of size (length^{level}) + 1.}
\texttt{VERIFY PROC CollegeChain2 (VAL VERIFY INT level, length)}
\texttt{  NORMALISE}
\texttt{   [2]BARRIER eatBar:}
\texttt{   PAR}
\texttt{      PhilFork. (eatBar[0], eatBar[1])}
\texttt{      Chain2 (level, length, eatBar[1], eatBar[0])}
\texttt{ ::}
\texttt{VERIFY DEADLOCK.FREE.F CollegeChain2 (0, 10, _, _)} ✔
\texttt{VERIFY DEADLOCK.FREE.F CollegeChain2 (1, 10, _, _)} ✔
\texttt{VERIFY DEADLOCK.FREE.F CollegeChain2 (10, 10, _, _)} ✔
\texttt{VERIFY DEADLOCK.FREE.F CollegeChain2 (100, 10, _, _)} ✔
\texttt{VERIFY DEADLOCK.FREE.F CollegeChain2 (1000, 10, _, _)} ✔
\texttt{VERIFY DEADLOCK.FREE.F CollegeChain2 (2000, 10, _, _)} ✔
\texttt{VERIFY DEADLOCK.FREE.F CollegeChain2 (2500, 10, _, _)} ✔
\end{verbatim}
```

\textbf{FDR2} verifies the first four above almost instantly. The college of size \(10^{1000} + 1\) takes around 8 seconds and \(10^{2000} + 1\) around 20 seconds. The last one crashes \textbf{FDR2}: “\textit{broken pipe}” on the terminal launch window.
Finally, the Brute Force Approach

```plaintext
--* A college of size (length^level) + 1.

VERIFY PROC CollegeChain2 (VAL VERIFY INT level, length)
   NORMALISE
   [2]BARRIER eatBar:
   PAR
      PhilFork. (eatBar[0], eatBar[1])
      Chain2 (level, length, eatBar[1], eatBar[0])
   :

VERIFY DEADLOCK.FREE.F CollegeChain2 (2000, 10, __, __) ✓
```

The same arguments as before reveal that removing the report hiding from these colleges leaves them deadlock and livelock free.

For the college with \((10^{2000} + 1)\) philosophers, all we need is a universe large enough to contain the computer on which to run it.

We may actually need several parallel universes. Establishing the barrier syncs and channel communications between them is an open question.
Reflection

\textit{occam-\pi} / \textit{CSP}_M

\textit{occam-\pi} teams well with \textit{CSP}_M to provide efficient executables and rich formal analysis.

This presentation reflects a proposal to extend \textit{occam-\pi} to include \textit{verification assertions} (about \textit{deadlock}, \textit{livelock}, \textit{determinism} and \textit{refinement}). Its compiler will generate suitably abstracted \textit{CSP}_M and interact with the \textit{FDR2} model checker, feeding back results in terms of the source \textit{occam-\pi} program.

Together with the ancient formal \textit{Laws of occam Programming} *, this moves \textit{occam-\pi} towards a process algebra in its own right.

* \url{http://portal.acm.org/citation.cfm?id=53255}

Reflection

Observation

Formal verification of the behaviour of concurrent processes can be achieved — by students — even though they engaged in only simple reasoning themselves.

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

Aside: model checking found an error overlooked in developing the (Device) 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]
Reflection

Class experience

The (Device) 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.

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.
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

Sequence, variables, assignment, parameters, concurrency, channels, synchronisation, ...

Fundamental primitives for software engineering

All are important. All are simple. All are available.
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
- Because it simplifies
- Because it scales

**CSP / π-calculus**

**occam-π / JCSP**

for complexity

for performance
A Thesis (for which we have experimental evidence)

Complex and high-performance systems cannot avoid concurrent design, implementation and reasoning.

Common concurrency bugs are intermittent – not repeatable on demand. Untestable in practice.

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

But also we can (and we should) teach formal analysis and verification of this concurrency at the same time …
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 … ?
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 … ?

Any questions?