Adding Formal Verification to occam-π

Peter Welch[a], Matt Pedersen[b], Fred Barnes[a],
Carl Ritson[a] and Neil Brown[a]
[a] School of Computing, University of Kent, UK
[b] School of Computer Science, UNLV, USA

Complex Systems Modelling and Simulation infrastructure

CPA 2011, University of Limerick,
22nd. June, 2011

occam-π, the process algebra

Aim:

To enable formal verification of occam-π programs to be conducted within the language itself … *as a matter of course by the programmer*.

How:

Extend occam-π with *verification qualifiers* and *assertions*. Modify the compiler to generate *(minimal)* CSP_M code from programs using these qualifiers and assertions, bounce this off the FDR model checker and report back in terms of the source code.

occam-π, the process algebra

Why?

It's time!

autonomous ⇔ *emergent & complex*

(THESIS)

Why?

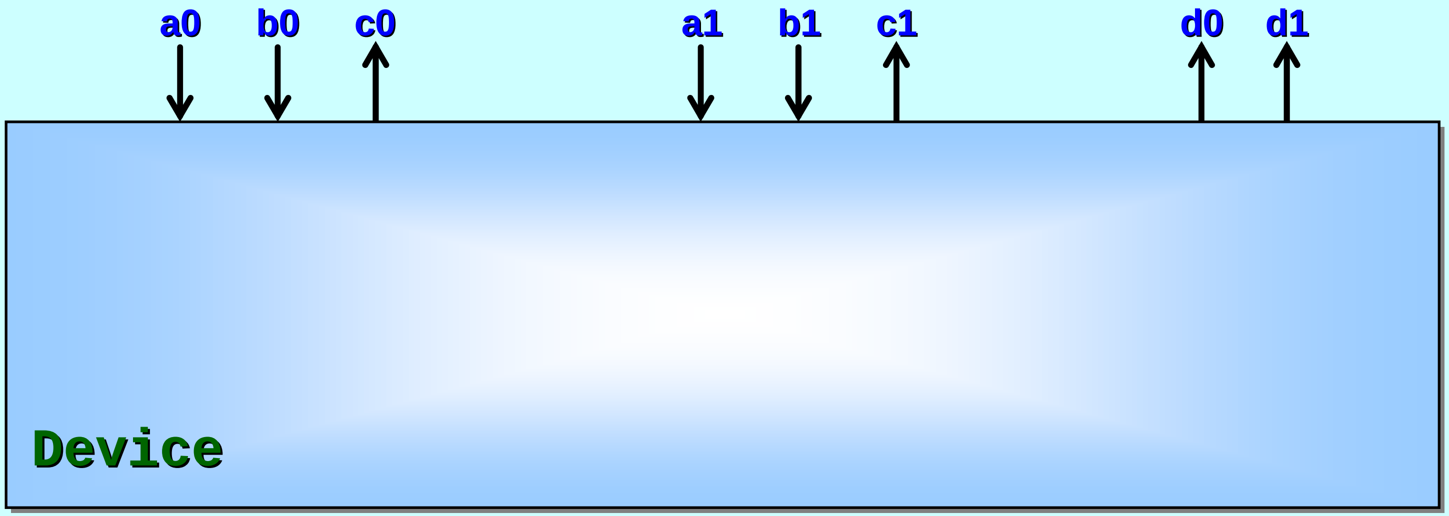
"Use of *autonomous* systems will require developing new methods to establish *'certifiable trust in autonomy'* through *Verification and Validation (V&V)* of the near-infinite state systems that result from high levels of adaptability; the lack of suitable V&V methods today prevents all but relatively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

Werner J.A. Dahm, Chief Scientist of the U.S. Air Force (AF/ST), "A Vision for Air Force Science & Technology (2010-2030)", May 2010

Example: *autonomous robot component*

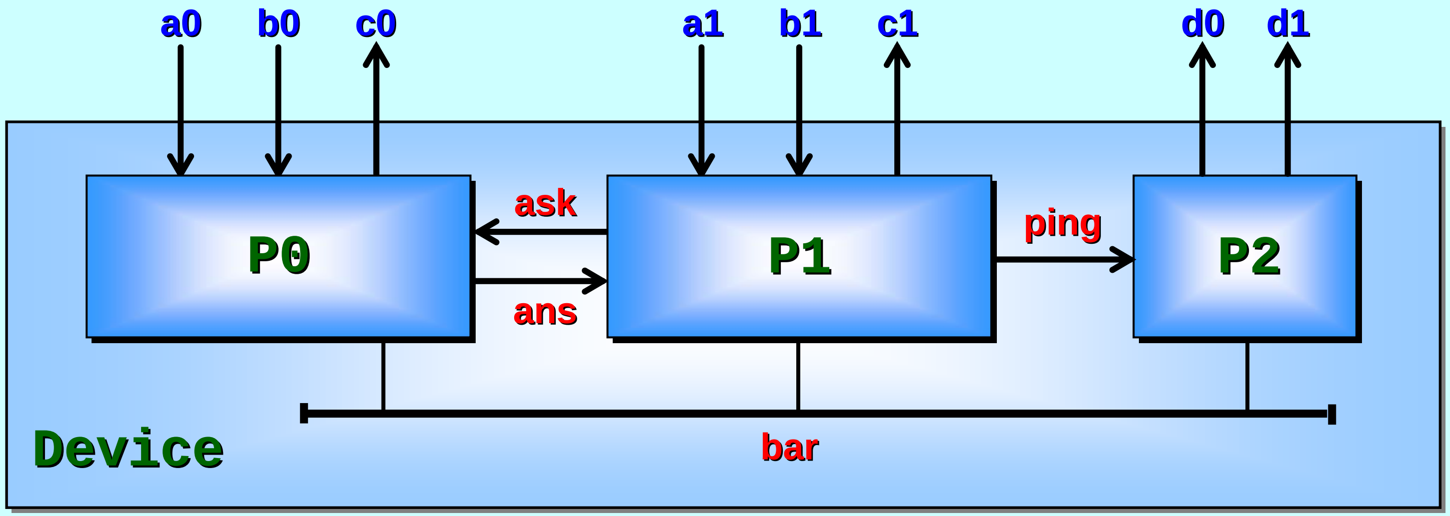
The following example has been developed from one first worked through in a single lesson of a graduate class in concurrency at UNLV in the spring of 2010.

Example: *autonomous robot component*



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

Example: *autonomous robot component*

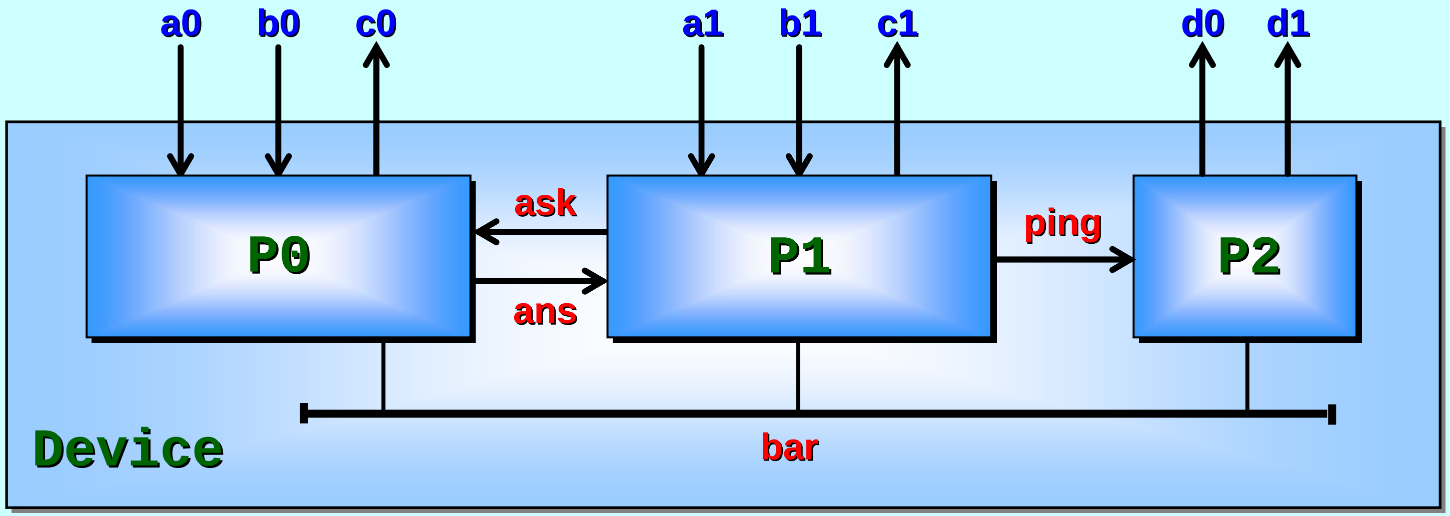


Device : real-time controller for 8 channels (4 input, 4 output).

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

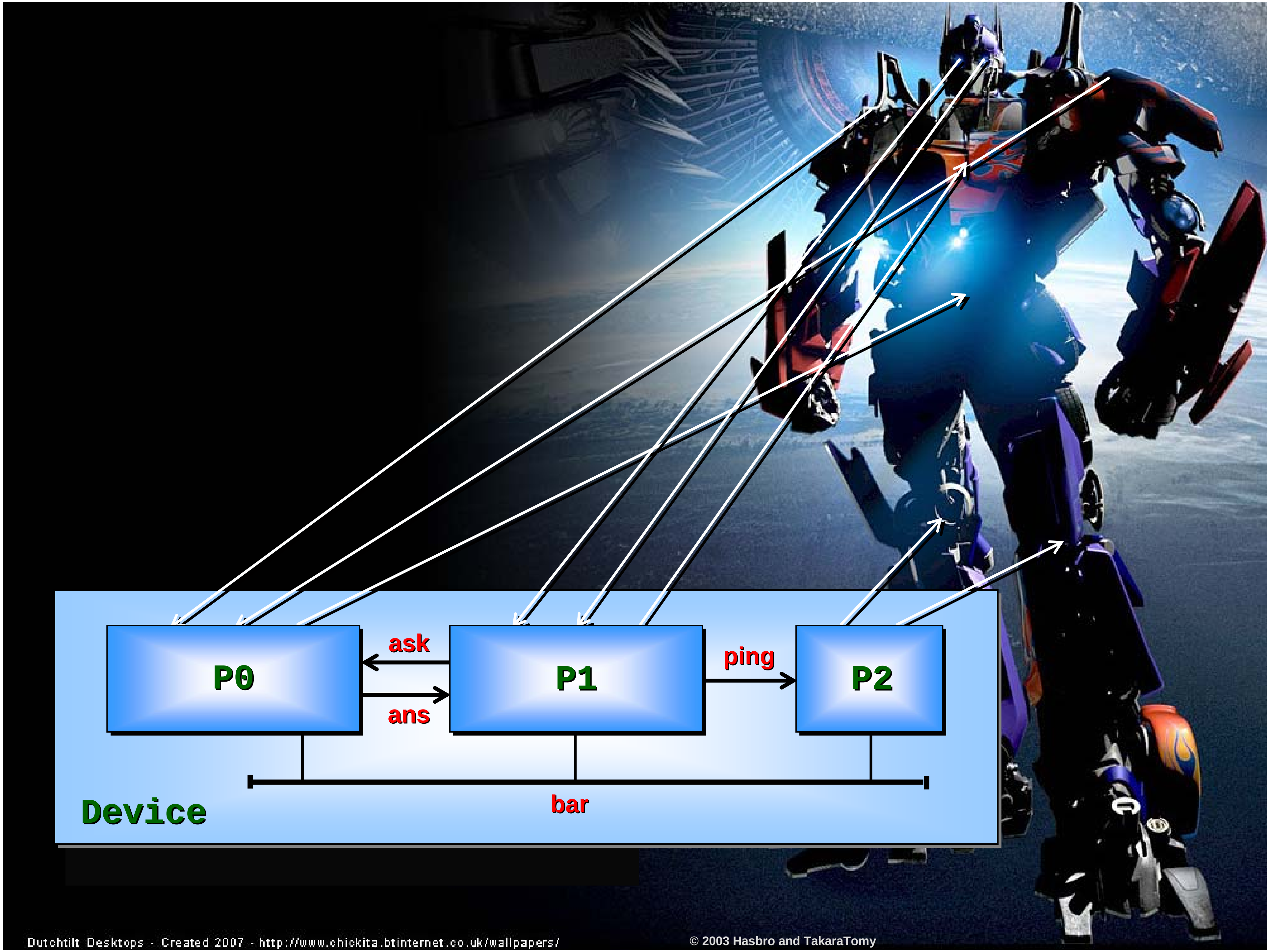
They exchange information over internal channels (**ask**, **ans**, **ping**) and all coordinate actions with an internal barrier (**bar**).

Example: *autonomous robot component*

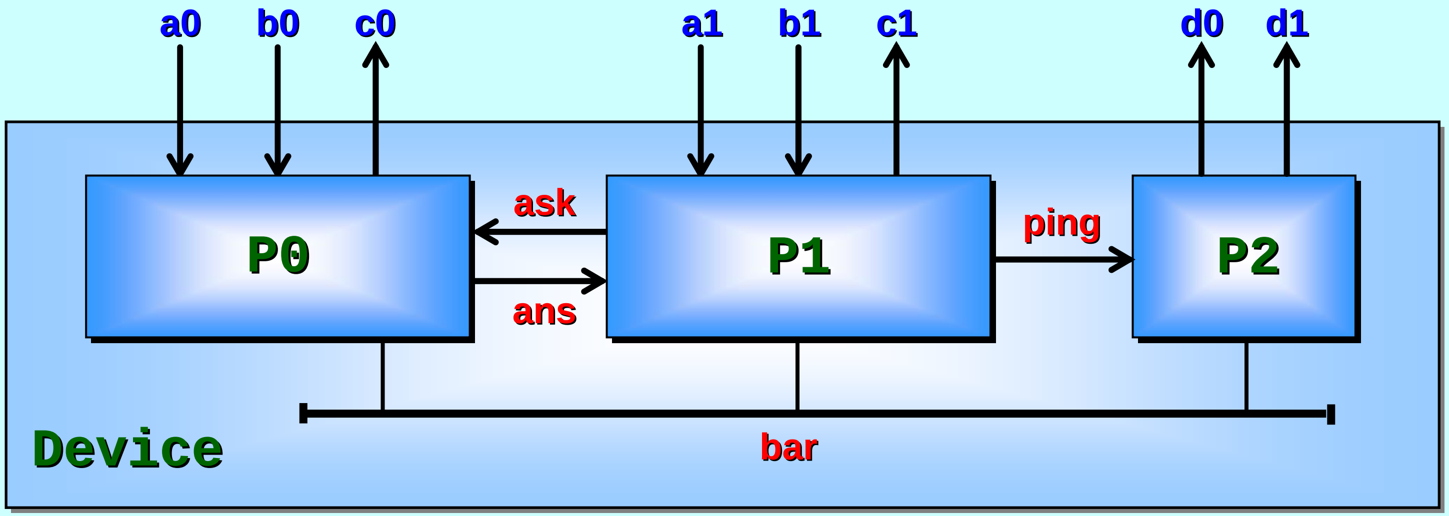


CSP semantics apply. *Channel communication* is unbuffered (sender waits for receiver and vice-versa). Any process *reaching a barrier* waits for *all* processes to *reach the barrier*.

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


ask
ans
ping
P0
P1
P2
bar
Device
Dutchtilt Desktops - Created 2007 - http://www.chickita.btinternet.co.uk/wallpapers/
© 2003 Hasbro and TakaraTomy

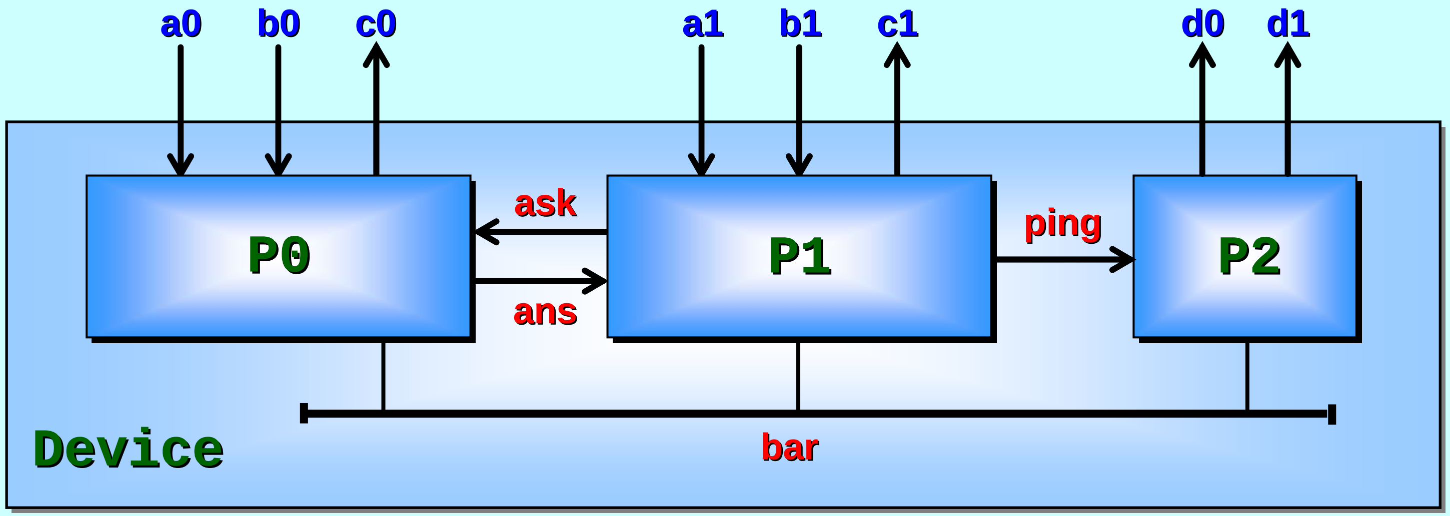
Behaviour: *two representations*



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.]

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

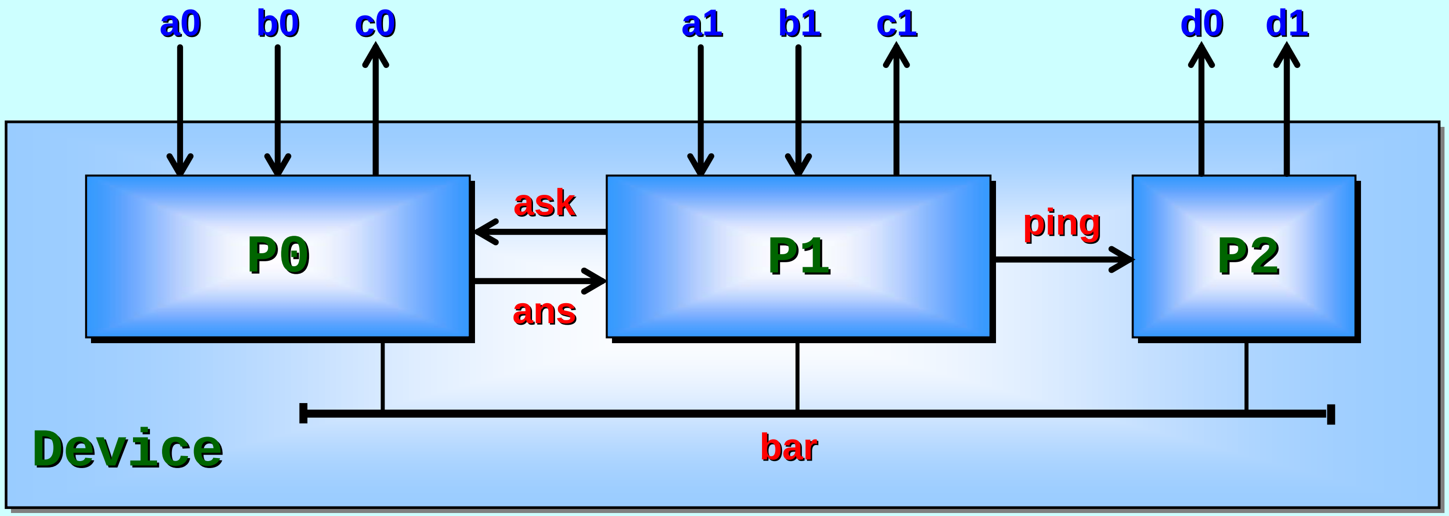
Behaviour: *one representation*



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.]

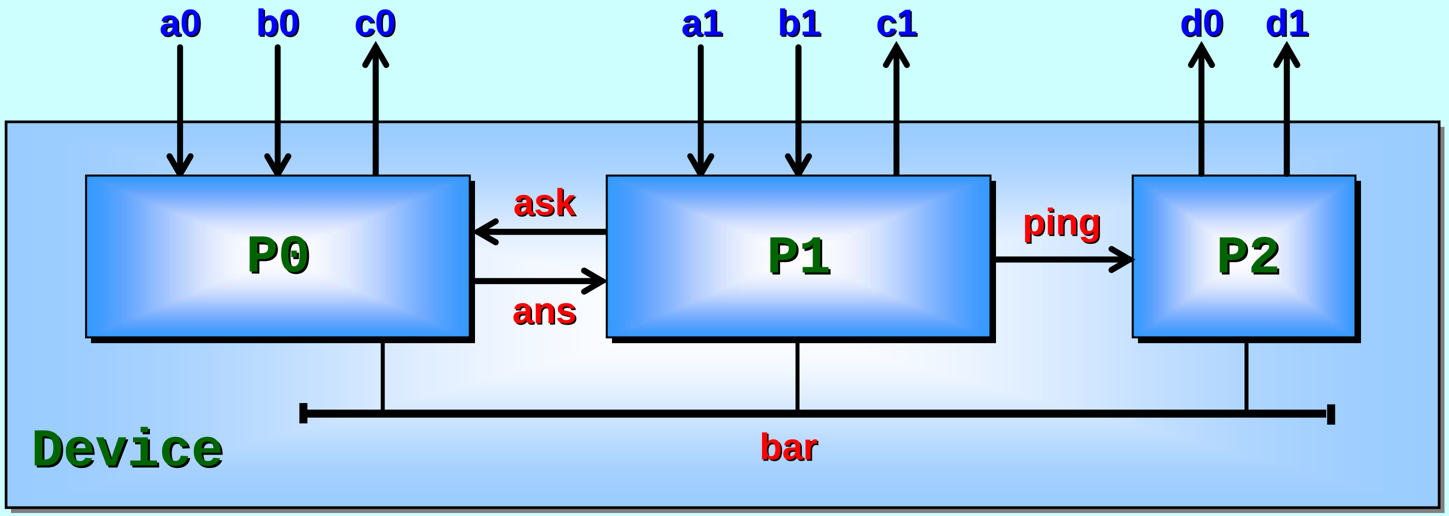
Behaviour: *what are we looking for?*



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**).]*

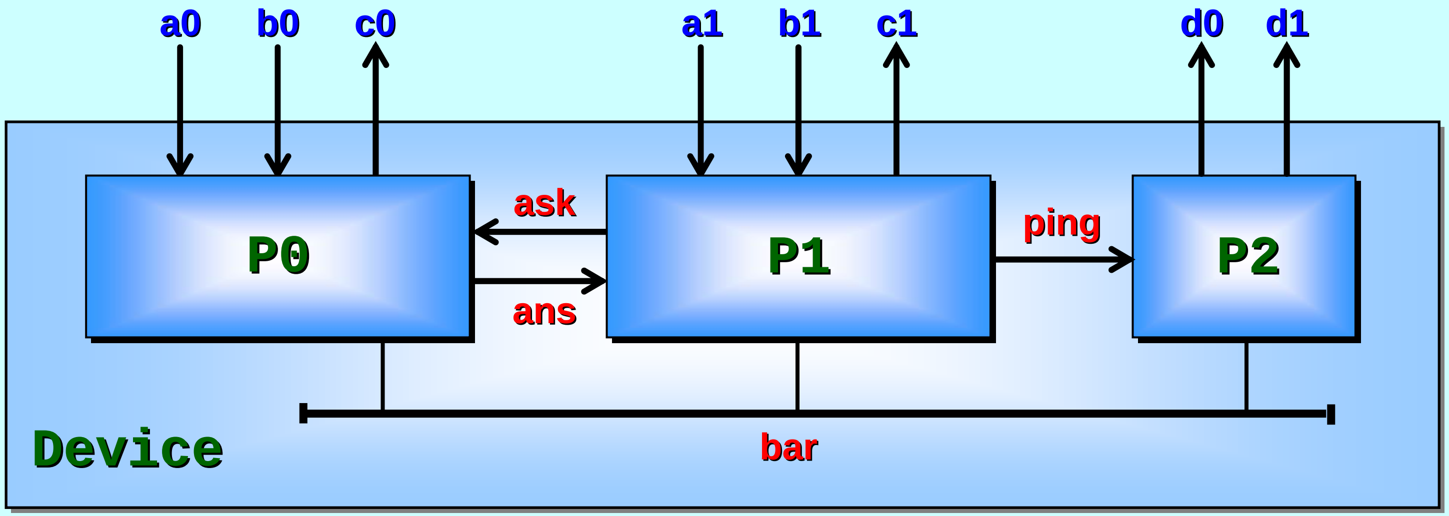
Behaviour: *what are we looking for?*



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

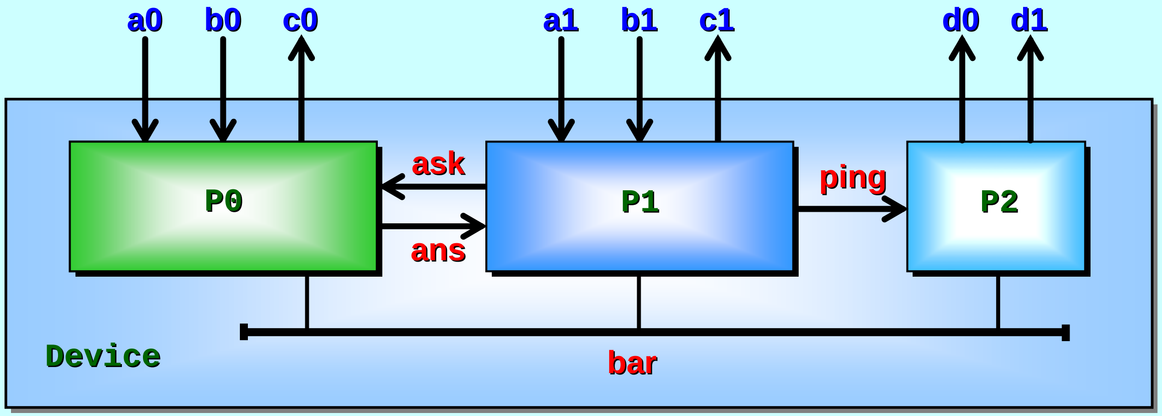
liveness: *will* it engage in correct sequences of external signals, as required?
[Some specs allow alternative sequences to be performed – all are correct, but an implementation must only do one and is free to choose.]

Behaviour: occam-π (executable)



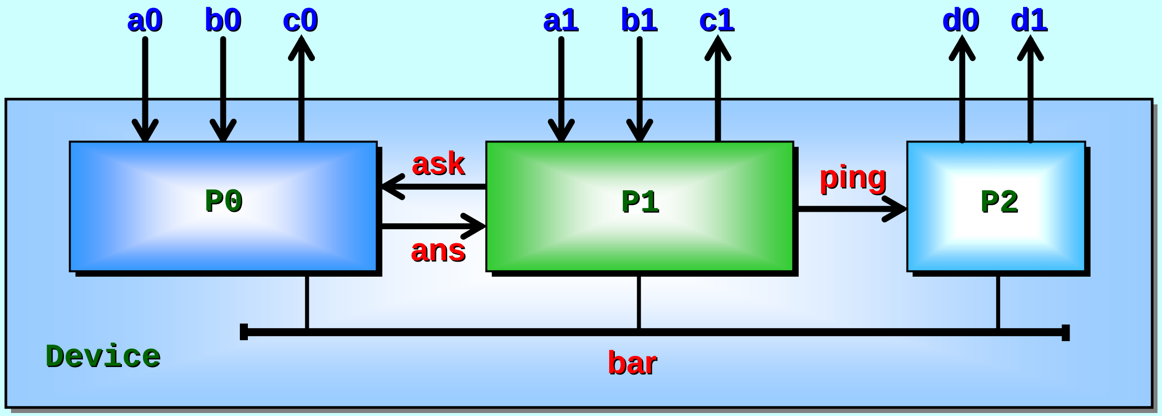
For the behaviour analysis in this example, data values and computations are not relevant. For simplicity, they are omitted in these codes, with all message content abstracted to zero.

Behaviour: occam-π *(executable)*



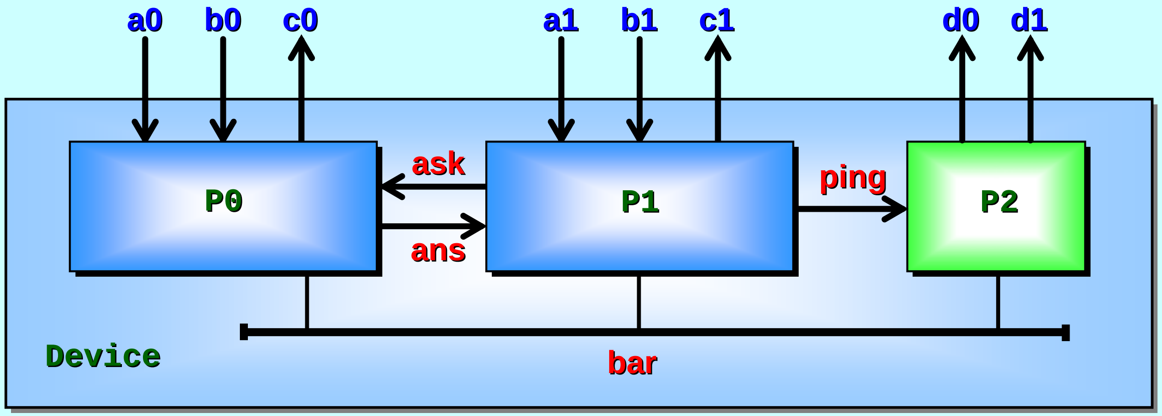

```
PROC P0 (CHAN INT a0?, b0?, c0!, ask?, ans!, BARRIER bar)
  WHILE TRUE
    INT x, y, z:
    SEQ
      ask ? x      -- take question
      a0 ? y
      ans ! 0      -- return answer (will depend on x and y)
      b0 ? z
      SYNC bar     -- wait for the others
      c0 ! 0
:
```

Behaviour: occam-π (executable)



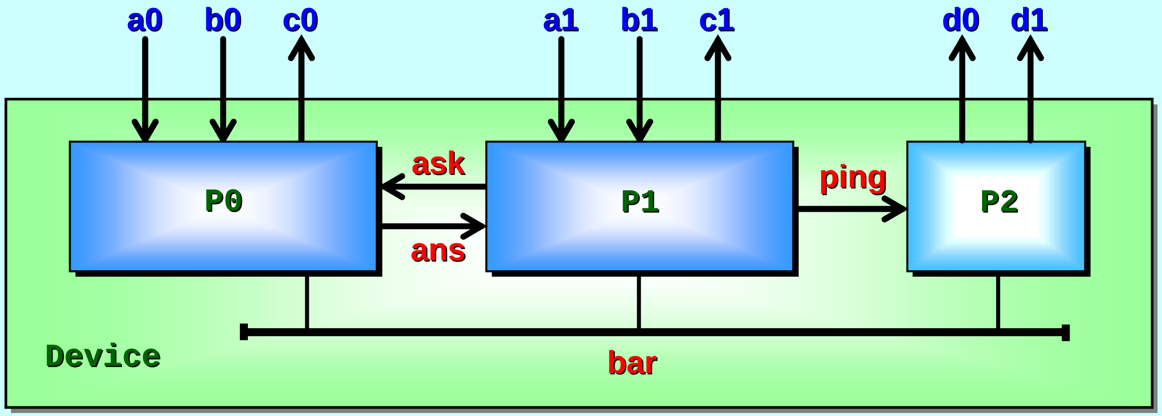

```
PROC P1 (CHAN INT a1?, b1?, c1!, ask!, ans?, ping!,
         BARRIER bar)
  WHILE TRUE
    INT x, y, z:
    SEQ
      ask ! 0       -- ask question
      ans ? x       -- wait for answer
      a1 ? y
      b1 ? z
      SYNC bar      -- wait for the others
      c1 ! 0
      ping ! 0      -- update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

Behaviour: 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
:
```

Behaviour: *occam-π (executable)*




```
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)
:
```

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 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
:
```

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
      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
:
```

What's first?

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 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
:
```

What's first?

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 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
:
```

What's first?

a0

<a0>

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 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
:
```

What's second?

<a0>

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 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
:
```

What's second?

<a0>

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 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
:
```

What's second?

b0 or **a1**

<a0>

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 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
:
```

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
      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
:
```

If **b0** second, then?

a1

<a0, b0, a1>

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 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
:
```

If **b0** second, then?

a1 then **b1**

<a0, b0, a1, b1>

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 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
:
```

If **b0** second, then?

a1 then **b1**

<a0, b0, a1, b1>

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 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
:
```

backtracking …

What's second?

b0 or **a1**

<a0>

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 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
:
```

If **a1** second, then?

<a0, a1>

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 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
:
```

If **a1** second, then?

b0 and **b1** *

<a0, a1>

(* any order)

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 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
:
```

If **a1** second, then?

b0 and **b1**

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

If **a1** second, then?

b0 and **b1**

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

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

What next?

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 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
:
```

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

What next?

c0 **c1** **d0** *

(* any order)

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 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
:
```

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

What happens when the sub-processes start looping?

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 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
:
```

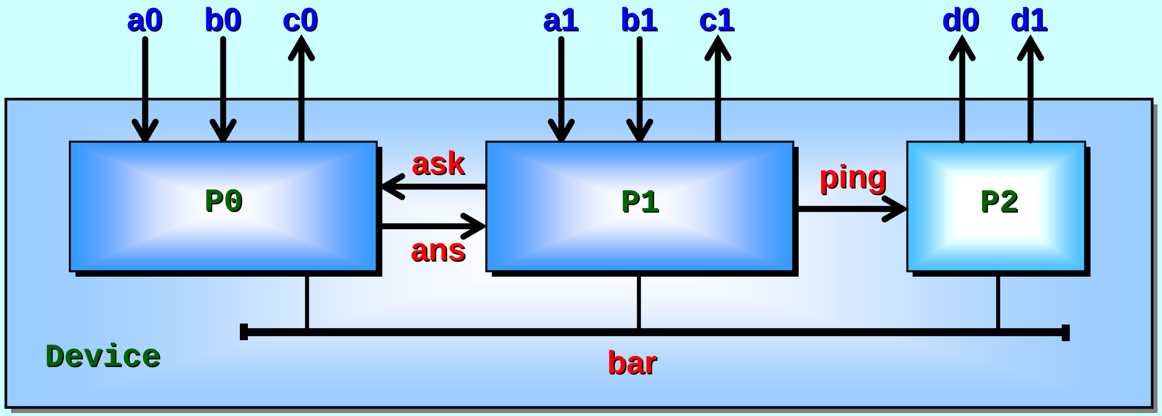
```
PROC P1 (CHAN INT a1?, b1?, c1!, ask!, ans?, ping!,
         BARRIER bar)
  WHILE TRUE
    INT x, y, z:
    SEQ
      ask ! 0      -- ask question
      ans ? x      -- wait for answer
      a1 ? y
      b1 ? z
      SYNC bar     -- wait for the others
➡     c1 ! 0
      ping ! 0     -- update neighbour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

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

Are there some more possible *first-7* signal sequences?

Formal

Behaviour: occam-π *(verifyable)*



With *verification qualifiers* and *assertions*, we can ask the occam-π 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-π source and using the FDR2 model checker.

Formal

Verify Qualifiers: *data*

If we generated CSP_M that fully reflected the semantics of the occam-π source code, we would quickly produce a system with too many states for any feasible *model checking*. For instance, a single **INT** variable has 4G 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
```

Formal

Verify Qualifiers: *data*

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

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

maps to:

```
Q (c, d) = c -> SKIP |~| d -> SKIP
```

Formal

Verify Qualifiers: *data*

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
:
```

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

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

Formal

Verify Qualifiers: *data*

or

If data values are significant, we qualify their types:

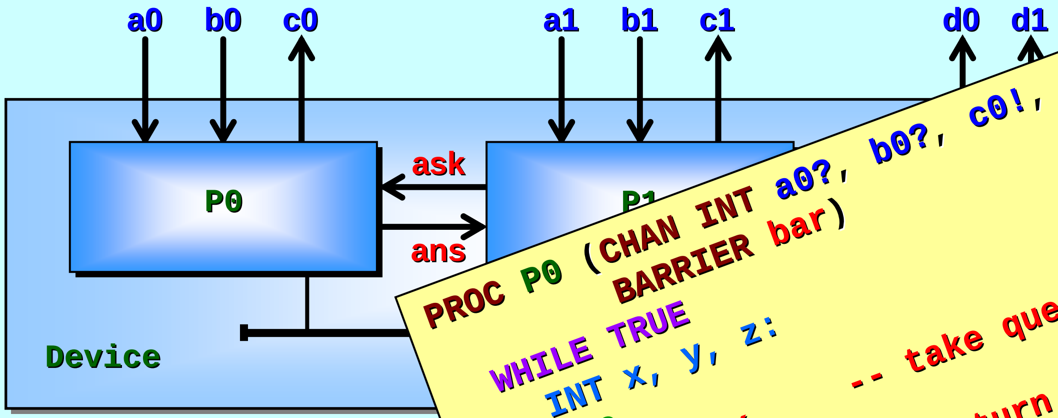
```
PROC Q (VAL VERIFY INT i, CHAN VERIFY INT c!, d!)
  IF
    i = 42
      c ! i
    TRUE
      d ! i
:
```

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

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

Formal

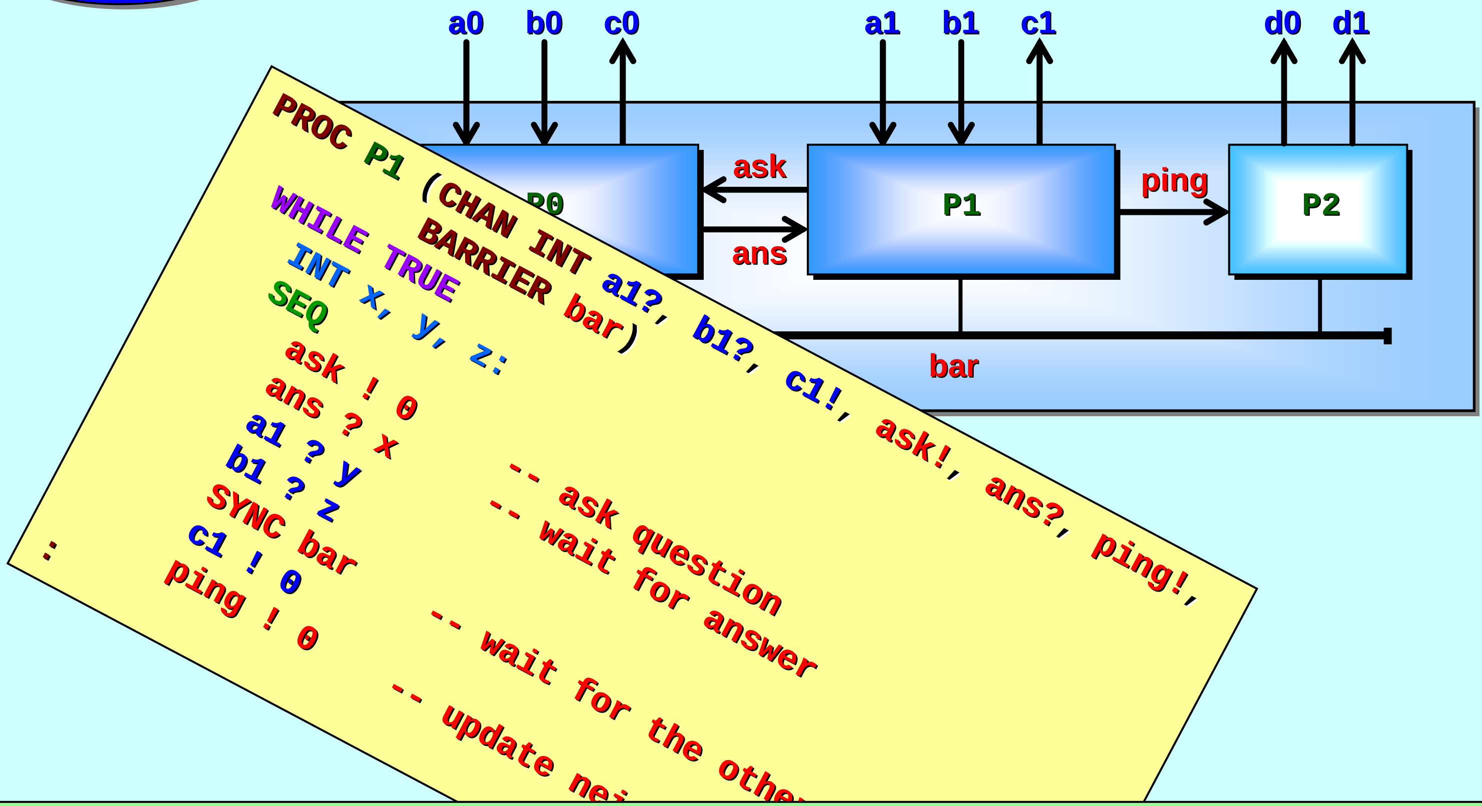
Compiling: occam-π ➔ CSP_M




```
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
      b0 ? z                  -- wait for others
      SYNC bar
      c0 ! 0
:
```

```
P0 (a0, b0, c0, ask, ans, bar) =
  let
    P0_0_ = ask -> a0 -> ans -> b0 -> bar -> c0 -> P0_0_
  within
    P0_0_
```

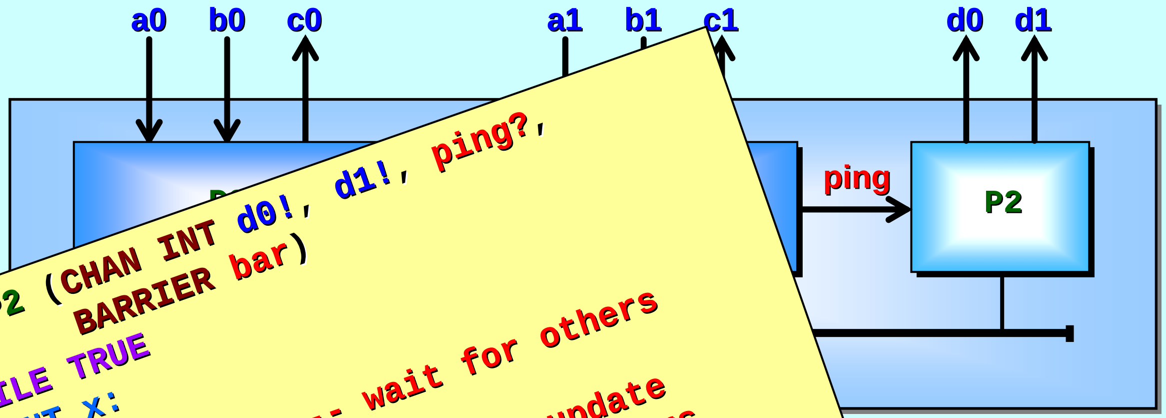
Formal Compiling: occam-π ➔ CSP_M




```
P1 (a1, b1, c1, ask, ans, ping, bar) =
  let
    P1_0_ = ask -> ans -> a1 -> b1 -> bar -> c1 -> ping -> P1_0_
  within
    P1_0_
```

Formal

Compiling: occam-π ➔ CSP_M

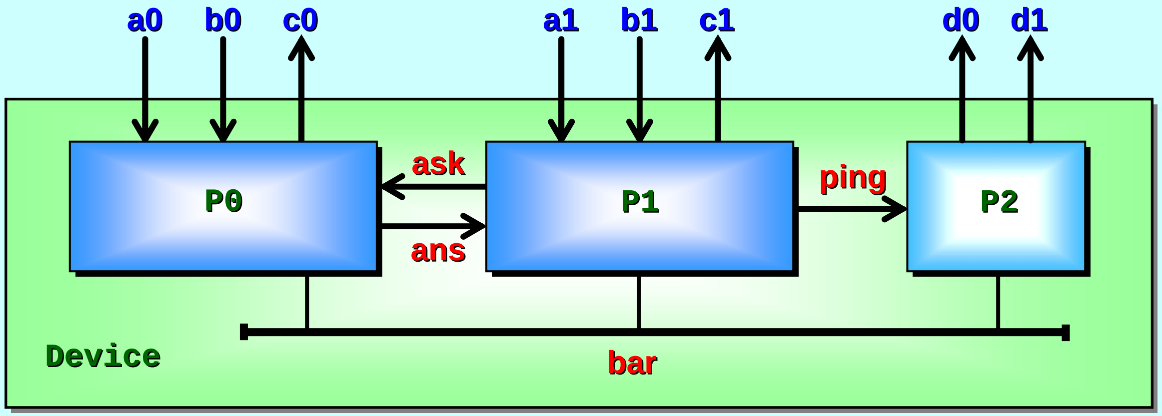



```
PROC P2 (CHAN INT d0!, d1!, ping?,
         BARRIER bar)
  WHILE TRUE
    INT x:
    SEQ
      SYNC bar          -- wait for others
      d0 ! 0
      ping ? x          -- receive update
      SYNC bar          -- wait for others
      d1 ! 0
      ping ? x          -- receive update
:
```

```
P2 (d0, d1, ping, bar) =
  let
    P2_0_ = bar -> d0 -> ping -> bar -> d1 -> ping -> P2_0_
  within
    P2_0_
```

Formal

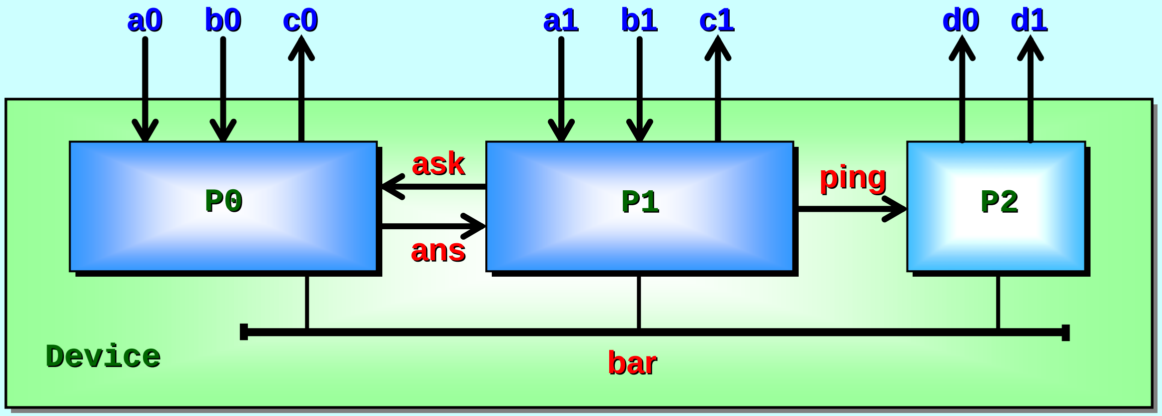
Compiling: occam-π ➔ CSP_M




```
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)
:
```

Formal

Compiling: occam-π → CSP_M




```
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_}
```

local channels are declared globally, used locally, hidden and not used again

Formal

Verify Assertions : occam-π

```
VERIFY <assertion>
VERIFY NOT <assertion>
```

<assertion>

```
DETERMINISTIC.F  <process>
DETERMINISTIC.FD <process>
DEADLOCK.FREE.F  <process>
DEADLOCK.FREE.FD <process>
LIVELOCK.FREE    <process>
TERMINATES       <process>
```

```
<process>  REFINES.T  <process>
<process>  REFINES.F  <process>
<process>  REFINES.FD <process>
```

where **<process>** is an instance of a **PROC**

Only **VAL VERIFY** operands need to be supplied (channels and barriers are supplied automatically)

Formal

Verify Assertions : occam-π

Without testing the system, we can assert straight away that **Device** is *deterministic* and *free from deadlock* and *livelock* – and that it doesn't *terminate*:

```
VERIFY DETERMINISTIC.FD Device
VERIFY DEADLOCK.FREE.FD Device
VERIFY LIVELOCK.FREE Device
VERIFY NOT TERMINATES Device
```

and the compiler says: " ✓ " !

☺ ☺ ☺ ☺ ☺

Formal

Verify Qualifiers: *processes*

To verify behaviours beyond determinism, deadlock and livelock freedom and termination, we need some way to express the behaviours we want. We can use occam-π for this, together with ***refinement***.

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

The occam-π compiler generates only CSP_M from such declarations – no executable code.

Within **VERIFY** processes, certain restrictions occam-π imposes (currently) can be removed – for instance, *output guards* and *barrier guards* are allowed.

Only **VERIFY** processes can invoke **VERIFY** processes.

Formal

Behaviour: occam-π *(verifyable)*

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

Intuition

Informal understanding

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

What next?

`c0` **`c1`** **`d0`** *

(* any order)

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

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
:
```

Informal understanding

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

What next?

c0 c1 d0 *

(* any order)

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

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

✓

… which verifies our intuition ☺☺☺

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
:
```

Informal understanding

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

What next?

c0 c1 d0 *

(* any order)

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

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

✓ **<a0, b0, a1, b1, d0, c0, c1>** is clearly a trace of **T0**. Therefore, it is also a trace of **Device**.

Formal Behaviour: occam-π (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>
```

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 ☺☺☺

Formal

Behaviour: occam-π (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>
```

What next?

c0 c1 d0 *

(* any order)

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

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

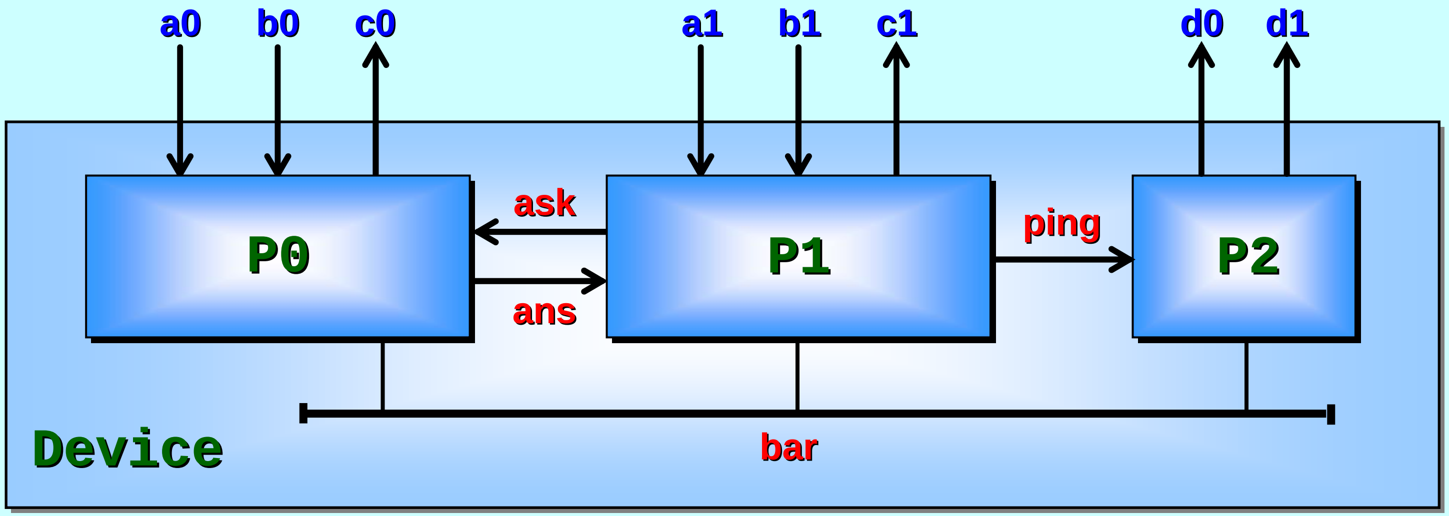
✗

At least one trace of **T1** is *not* a trace of **Device**. Comparing **T0** and **T1**, the fault lies in the mis-ordering of **d0** and **b1**.

Formal

Behaviour: occam-π (verifyable)

Safety



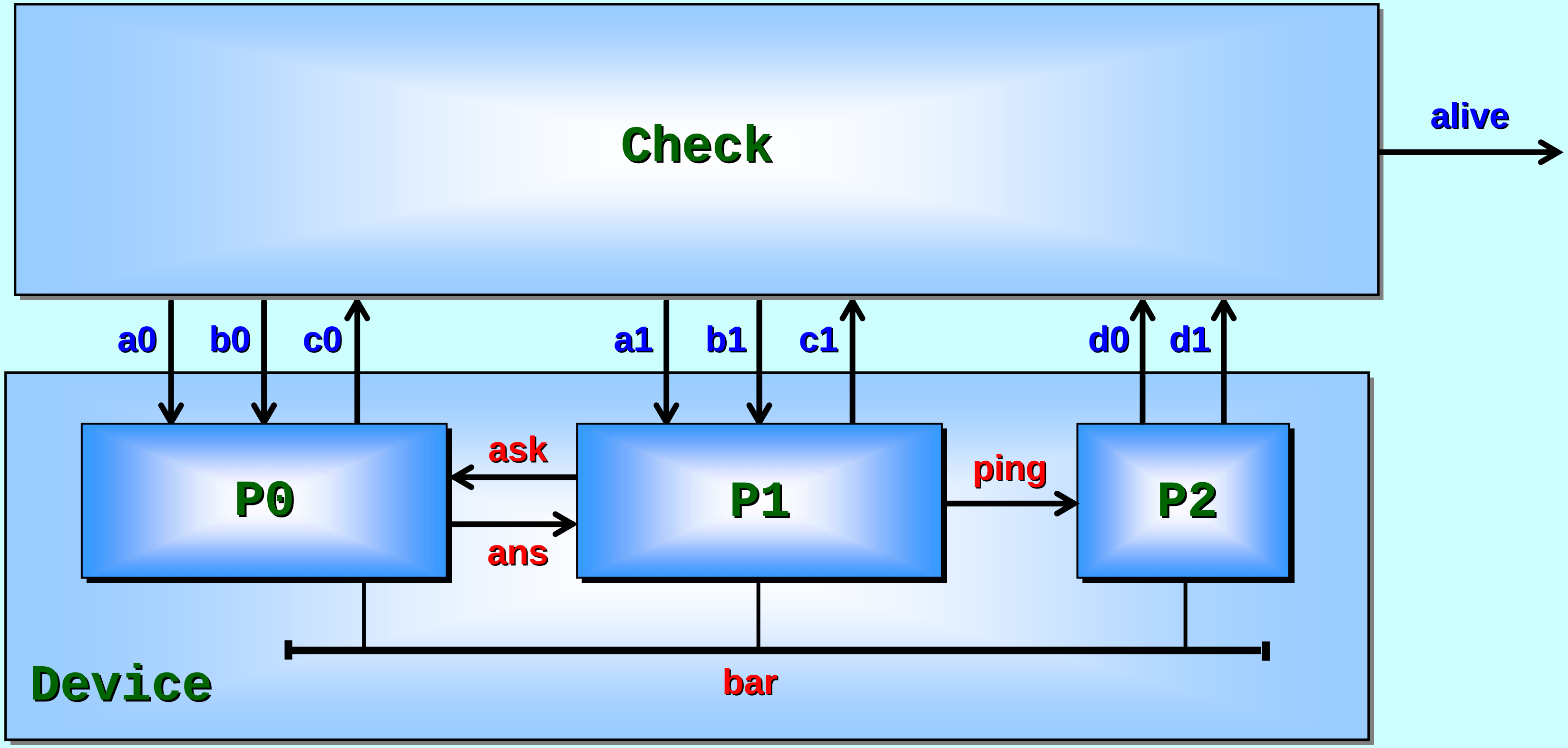
Let's ask a more difficult question about the continuous running of the system. Suppose the robot would do something *very bad* if its controller **Device** were ever to accept a signal *twice* on **a0** without a signal on **d0** or **d1** *in between*. Might this *ever* happen?

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 …

Formal

Behaviour: occam-π (verifyable)

Safety

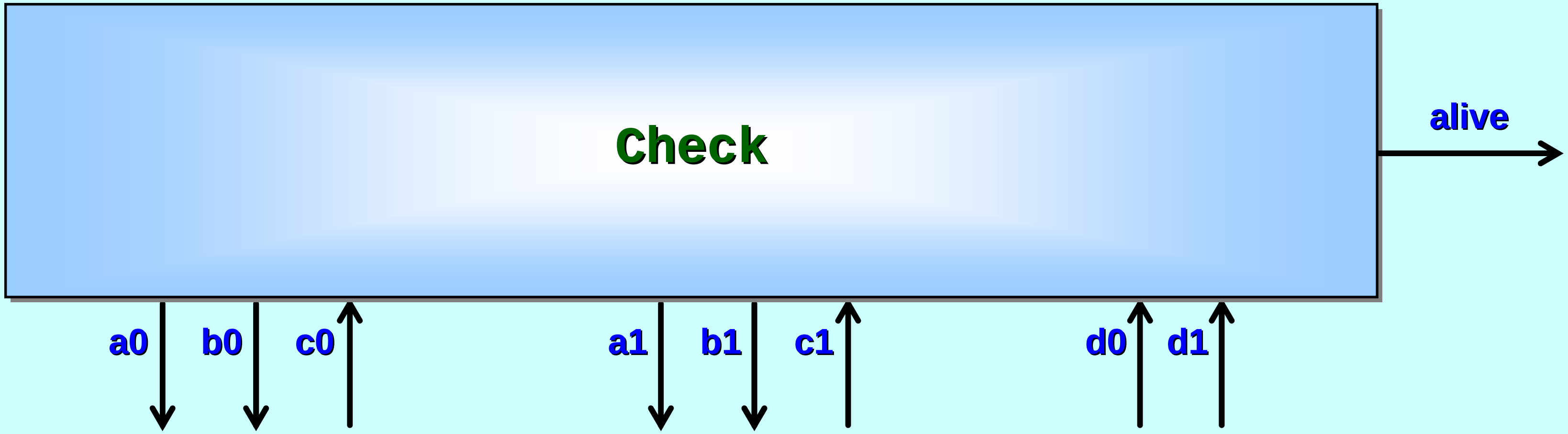


Simple: write a process that checks all signals to/from **Device**, looking for the bad scenario and deliberately deadlocks (the monitored system) if spotted. This is just programming …

Formal

Behaviour: occam-π (verifyable)

Safety



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 …

Formal

Behaviour: occam-π (verifyable)

Safety

```
VERIFY PROC Check (CHAN INT a0!, b0!, c0?, a1!, b1!, c1?,
                            d0?, d1?, alive!)
  --* n : the number of a0 signals since the last d0 or d1
  INITIAL VERIFY INT n IS 0:
  WHILE TRUE
    SEQ
      alive ! 0
      IF
        n >= 2
          STOP -- refuse all further signals (forcing deadlock)
        TRUE
          ...  process next signal (maintain n)
:
```

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: occam-π (verifyable)

Safety

```
{{{  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
}}}
```

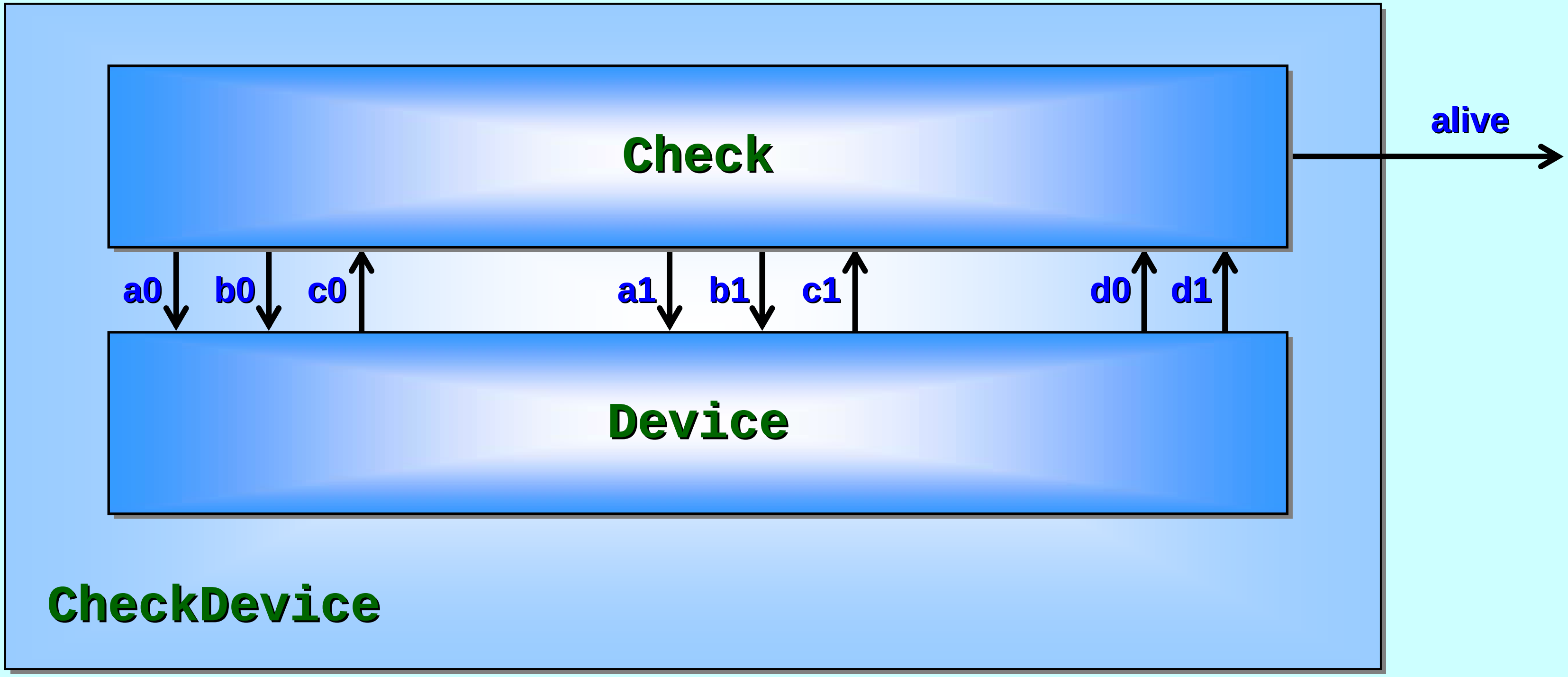
n = the number of a0 signals received since the last d0 or d1

This is an ALT with four input and four output guards

Formal

Behaviour: occam-π (verifyable)

Safety

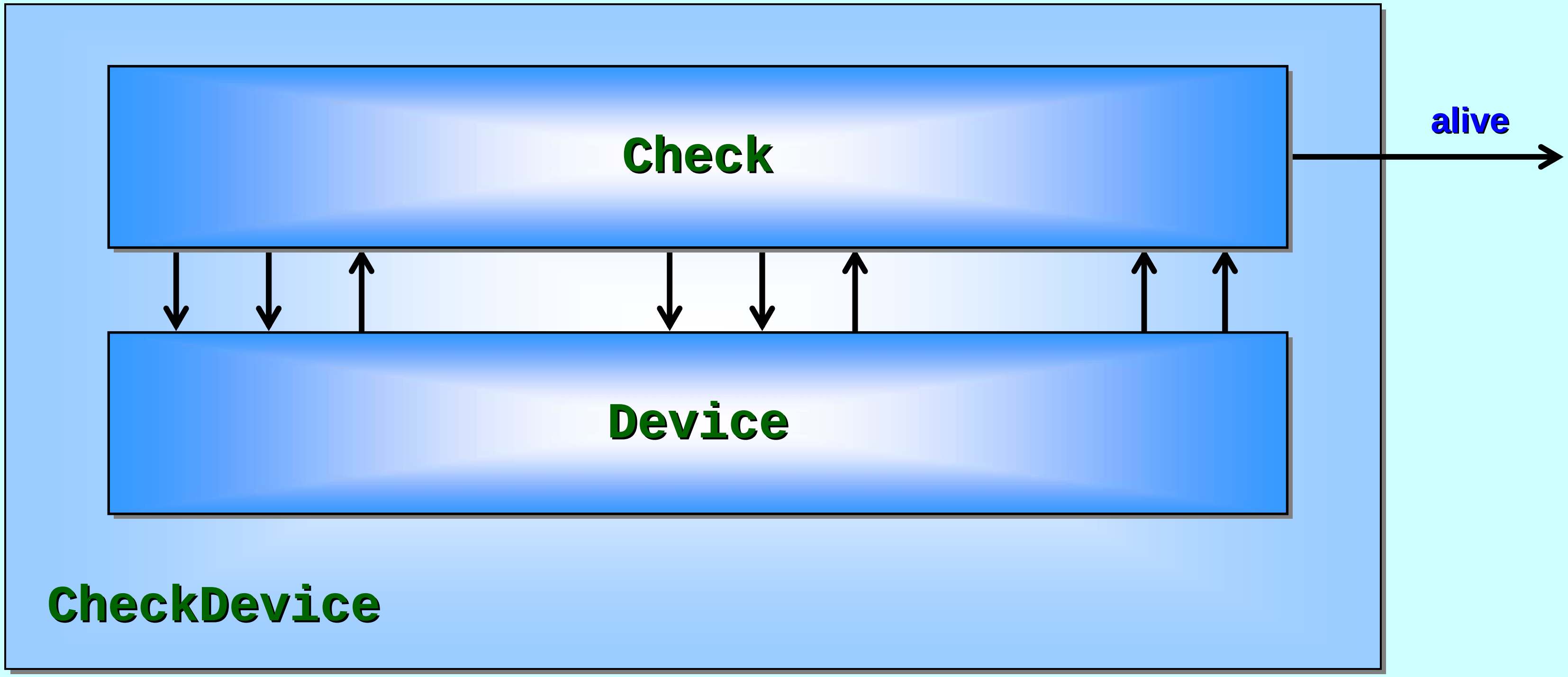



```
VERIFY PROC CheckDevice (CHAN INT alive!)
  CHAN INT a0, a1, b0, b1, c0, c1, d0, d1:
  PAR
    Check (a0!, b0!, c0?, a1!, b1!, c1?, d0?, d1?, alive!)
    Device (a0?, b0?, c0!, a1?, b1?, c1!, d0!, d1!)
:
```

Formal

Behaviour: occam-π (verifyable)

Safety



`VERIFY DEADLOCK.FREE.FD CheckDevice` ✓

If **`Check`** stops, **`CheckDevice`** will deadlock.

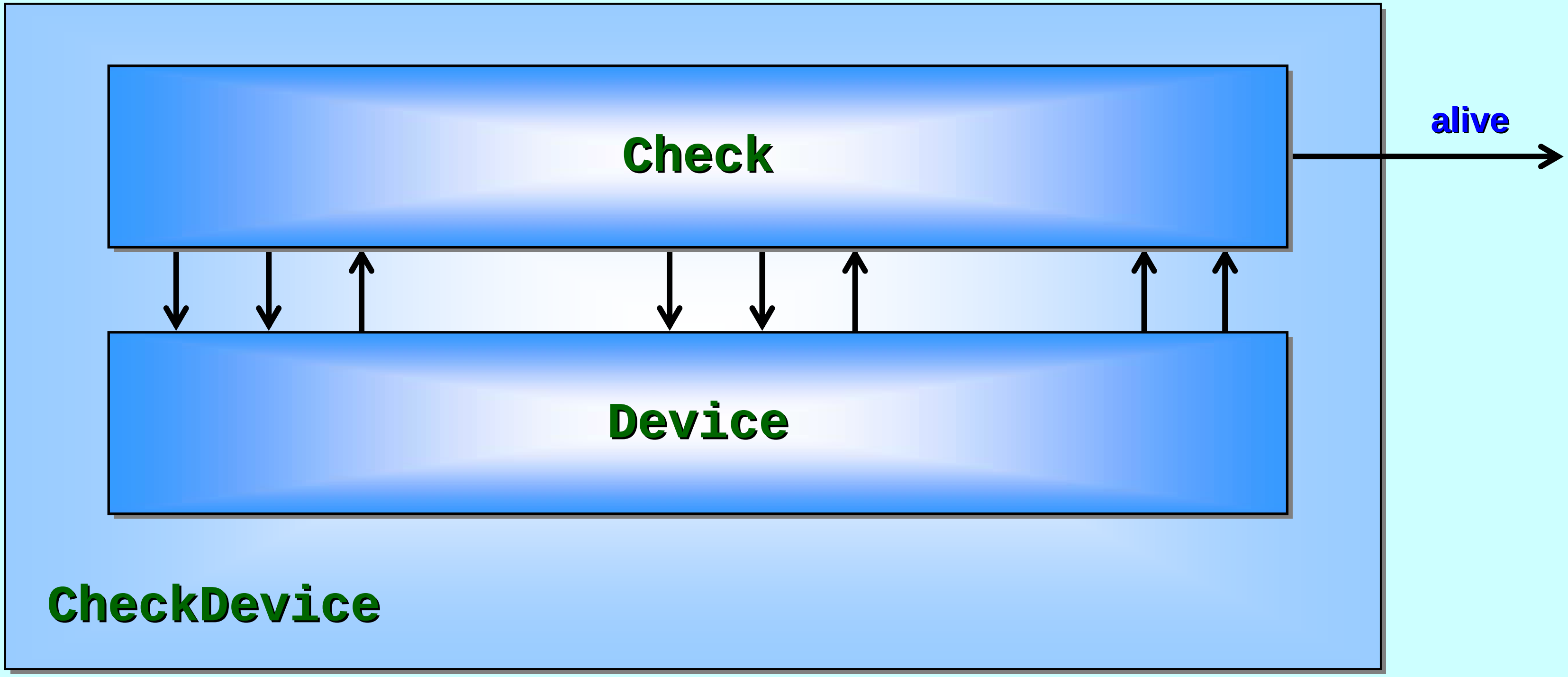
Therefore, **`Check`** never stops … *and the bad thing can't happen*.

Q.E.D.

Formal

Behaviour: occam-π (verifyable)

Safety



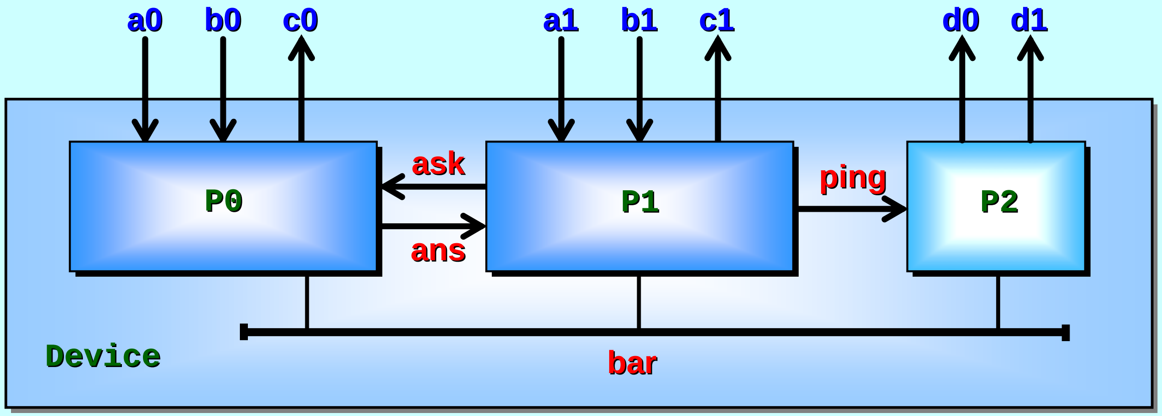
VERIFY DEADLOCK.FREE.FD CheckDevice ✓

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.

Formal

Behaviour: occam-π (verifyable)

Liveness



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.

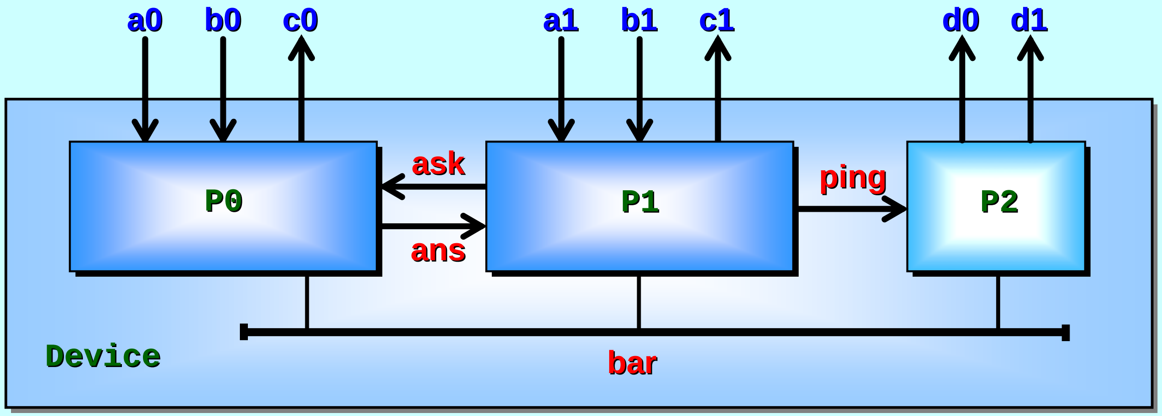
A *CSP failure* is a state that a system reaches (represented by its *trace* to that point) where it *may refuse to synchronise* with its environment on some given set of events.

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

Formal

Behaviour: occam-π (verifyable)

Liveness



Failure refinement makes 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.

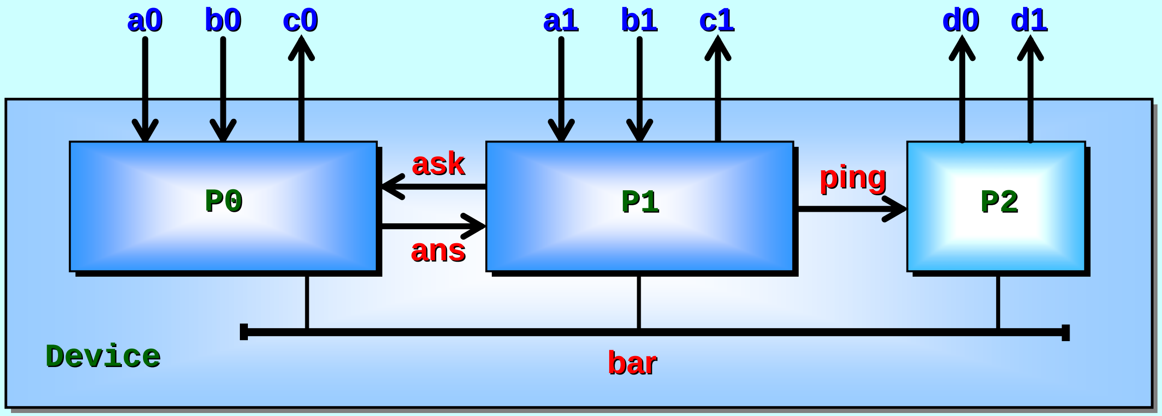
A *CSP* *failure* is a state that a system reaches (represented by its *trace* to that point) where it *may refuse to synchronise* with its environment on some given set of events.

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

Formal

Behaviour: occam-π (verifyable)

Liveness



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.

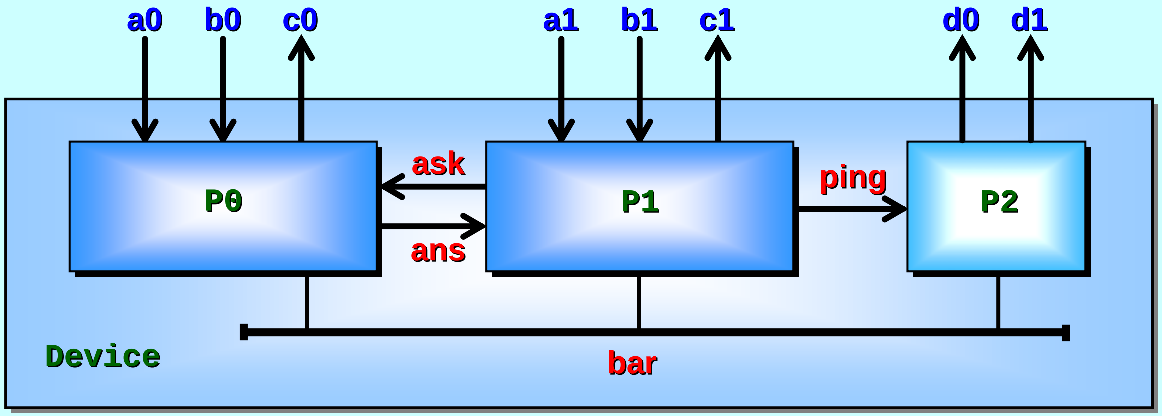
A *CSP* *failure* is a state that a system reaches (represented by its *trace* to that point) where it *may refuse to synchronise* with its environment on some given set of events.

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

Formal

Behaviour: occam-π (verifyable)

Liveness



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 …

Informal understanding

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

What next?

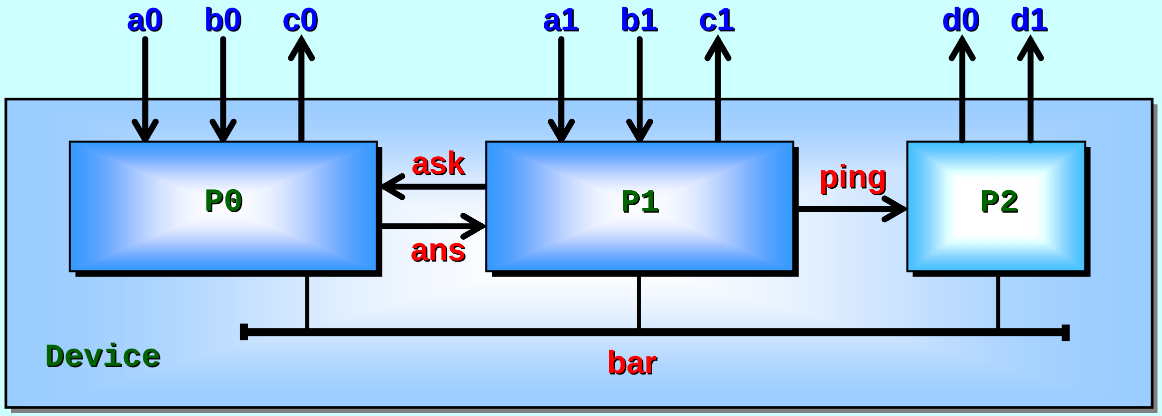
c0 **c1** **d0** *

(* any order)

Formal

Behaviour: *occam-π (verifyable)*

Liveness



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 …

Informal understanding

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

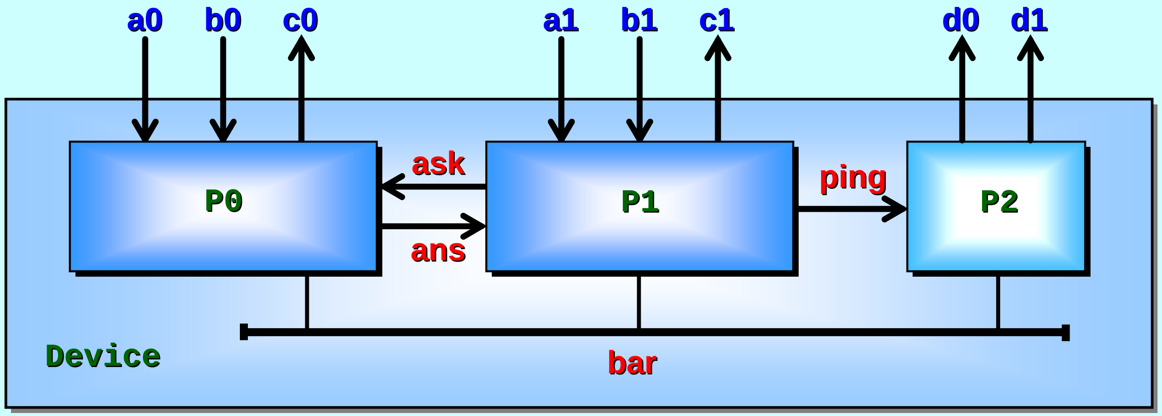
<c0> ||| <c1> ||| <d0>

interleave

Formal

Behaviour: occam-π *(verifyable)*

Liveness



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 …

Informal understanding

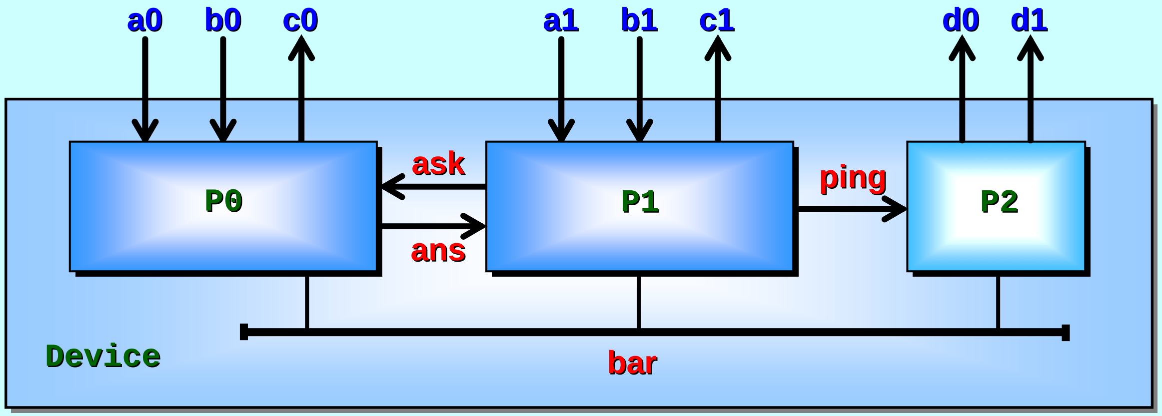
`<a0>`

`<b0> ||| <a1, b1>`

`<c0> ||| <c1> ||| <d0>`

interleave

Formal — Behaviour: occam-π *(verifyable)* — Liveness



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 …

Informal understanding

<a0>

<b0> ||| <a1, b1>

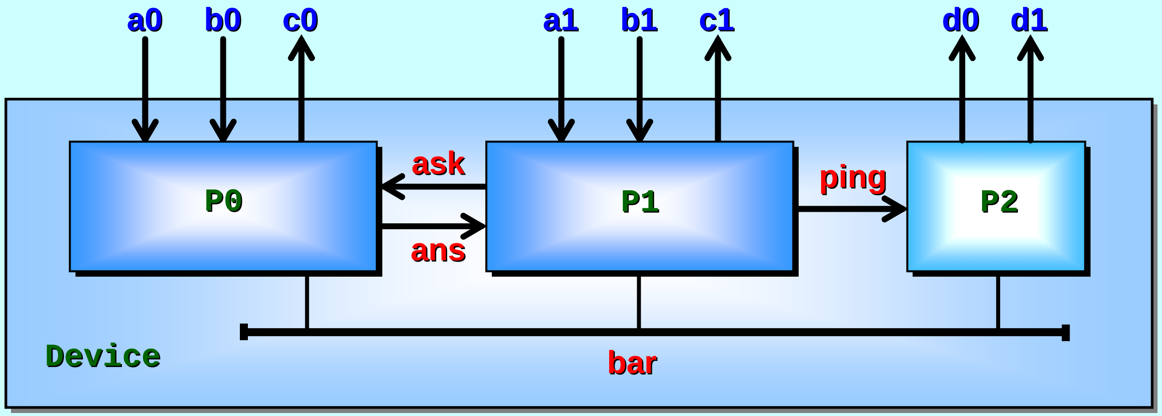
<c0> ||| <c1> ||| <d0>

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

Formal

Behaviour: occam-π (verifyable)

Liveness



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

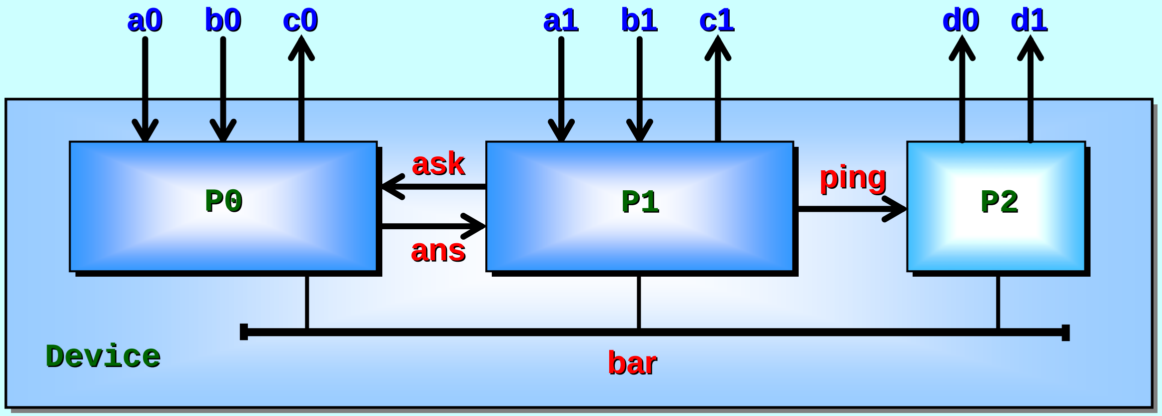
We can formalise the expression of those traces a bit better …

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

Formal

Behaviour: occam-π (verifyable)

Liveness



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

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

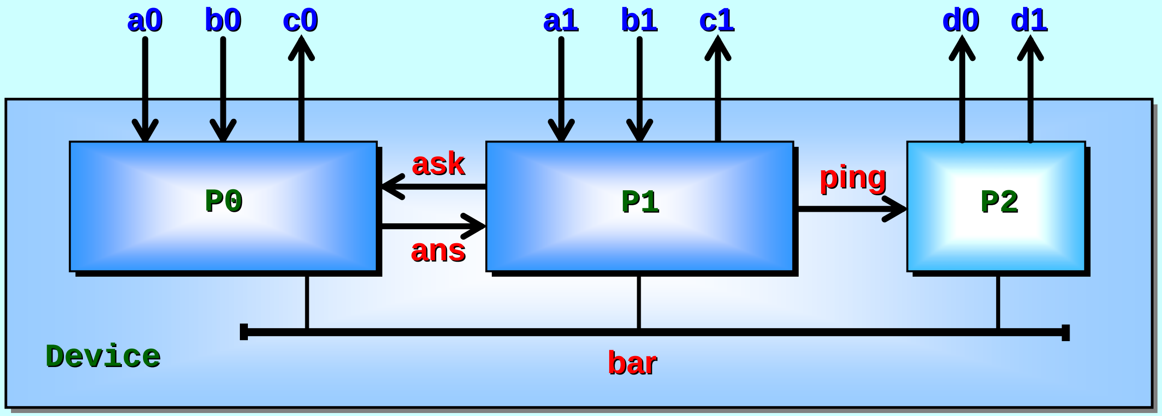
And the rest …

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

Formal

Behaviour: occam-π *(verifyable)*

Liveness



From such trace expressions, we can directly write down an occam-π process that offers all of them …

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

Formal

Behaviour: occam-π (verifyable)

Liveness

```
VERIFY PROC DeviceSpec (CHAN INT a0?, b0?, c0!,
                                 a1?, b1?, c1!, d0!, d1!)
  WHILE TRUE
    INT w, x, y, z:
    SEQ
      ...  phase 0
      ...  phase 1
:
```

From such trace expressions, we can directly write down an occam-π process that offers all of them …

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

Formal

Behaviour: occam-π *(verifyable)*

Liveness

```
{{{  phase 0
SEQ
  a0 ? w
  PAR
    b0 ? x
    SEQ
      a1 ? y
      b1 ? Z
  PAR
    c0 ! 0
    c1 ! 0
    d0 ! 0
}}}
```

From such trace expressions, we can directly write down an occam-π process that offers all of them …

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

Formal

Behaviour: occam-π (verifyable)

Liveness

```
{{{  phase 1
SEQ
  a0 ? w
  PAR
    b0 ? x
    SEQ
      a1 ? y
      b1 ? Z
  PAR
    c0 ! 0
    c1 ! 0
    d1 ! 0
}}}
```

From such trace expressions, we can directly write down an occam-π process that offers all of them …

This generation can be automated.

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

Formal

Behaviour: occam-π (verifyable)

Liveness

```
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>)
```

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: occam-π *(verifyable)*

Liveness

```
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: occam-π (verifyable)

Liveness

```
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
```

✓

Without this verification, we may be tempted to add another barrier (bar) sync at the end of each loop of **P0** and **P1** and half-loop of **P2**. The above *refinement* shows that the required pattern does indeed repeat cleanly and, so, this overhead is unnecessary.

Formal

Behaviour: occam-π (verifyable)

Liveness

```
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
```
✓

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: occam-π (verifyable)

Liveness

```
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
```

✓

We also have:

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

✓

But that's just icing on the cake! ☺ ☺ ☺

Formal

Verify Assertions : *compilation*

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_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.

Formal

Verify Assertions : *compilation*

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
```

The CSP_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.

Formal

Verify Assertions : *compilation*

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.

Formal

Verify Assertions : *compilation*

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].

Formal

Verify Assertions : *verified data*

The only arguments needed for CSP_M assertions are those for occam-π **VERIFY** data parameters. *Channels* and *barriers* can be supplied automatically. Non-**VERIFY** data parameters are irrelevant.

For example, if we need an assertion about:

```
PROC System (VAL VERIFY INT n, CHAN VERIFY INT out!)
```

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

```
VERIFY DEADLOCK.FREE.FD System (42, _)
```

where the underscore indicates arguments that are either irrelevant (*non-**VERIFY*** data) or automatic (*channels* and *barriers*).

Formal

Verify Assertions : *verification GUI*

Later, we plan an option for the occam-π compiler just to generate CSP_M code to be picked up by a GUI with facilities for interactive generation, checking and reporting of **VERIFY** assertions. These will be similar to those given by the FDR2 GUI, but processes and assertions will be in terms of the occam-π sources. FDR2, or some derivative, remains the underlying workhorse for model checking.

The GUI will allow flexible exploration of assertions with **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 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?

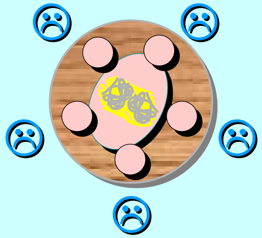
No

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 Dining Philosophers

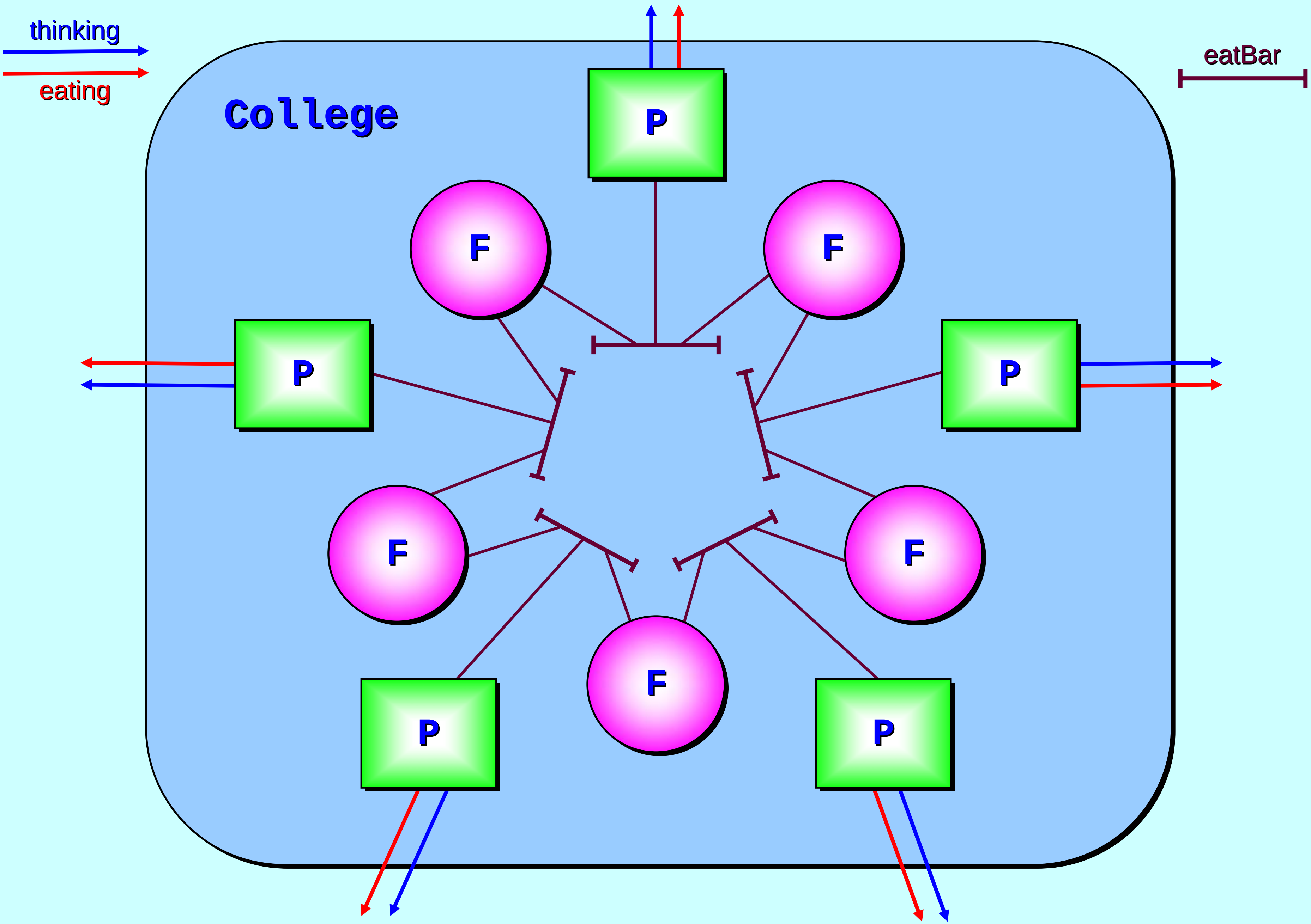
Extra Case Study ☺

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.

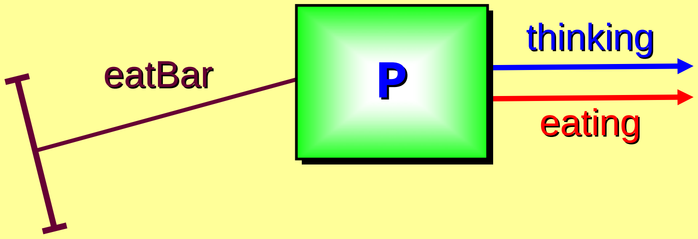
http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD310.PDF

Historical document

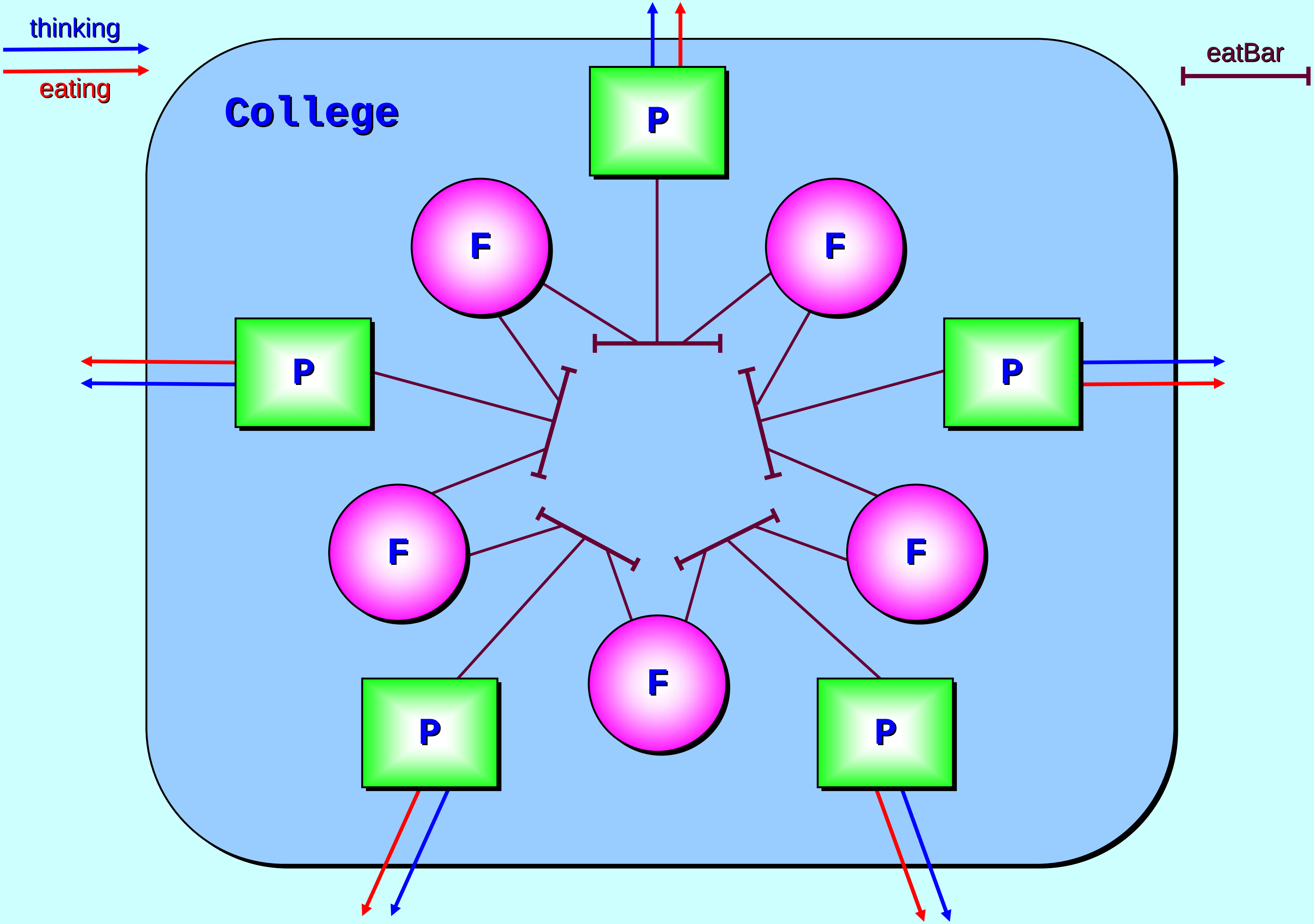


A new, really really neat, solution (Neil Brown / PHW)

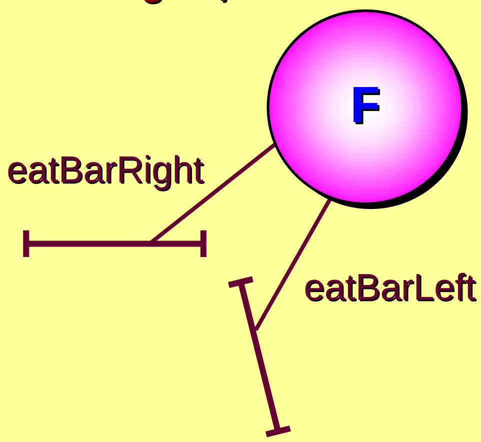
```
VERIFY PROC Phil (CHAN INT thinking!, eating!, BARRIER eatBar)
  WHILE TRUE
    SEQ
      thinking ! 0
      SYNC eatBar
      eating ! 0
      SYNC eatBar
:
```



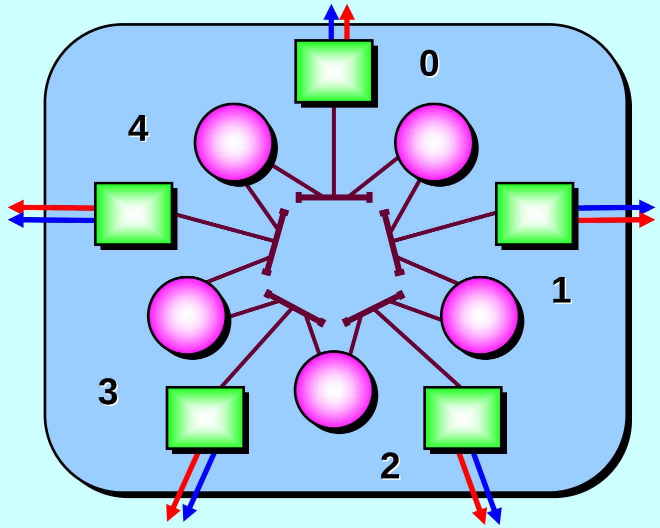

```
Phil (thinking, eating, eatBar) =
  let
    Phil_0_ =
      thinking -> eatBar ->
      eating -> eatBar -> Phil_0_
  within
    Phil_0_
:
```


thinking
eating
eatBar
College
P
F
F
P
P
F
F
F
P
P

```
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_
:
```



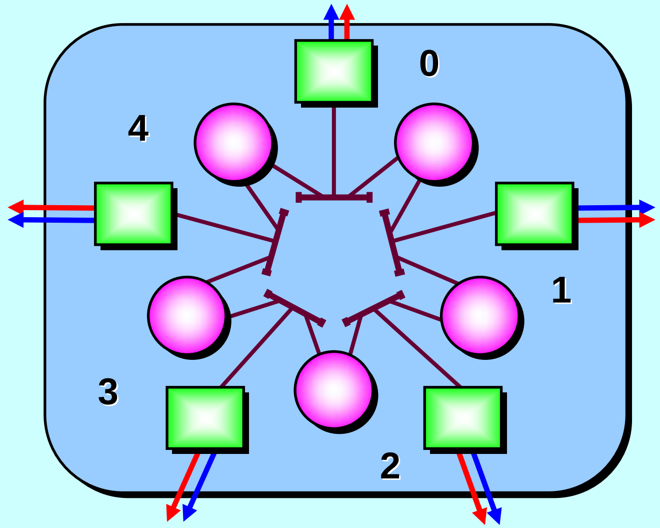

```
VAL INT nPhils IS 5:
```

```
nPhils = 5
```

```
VERIFY PROC Philosophers ([nPhils]CHAN INT thinking!, eating!,
                          [nPhils]BARRIER eatBar)
  PAR id = 0 FOR nPhils
    Phil (thinking[id]!, eating[id]!, eatBar[id])
:
```

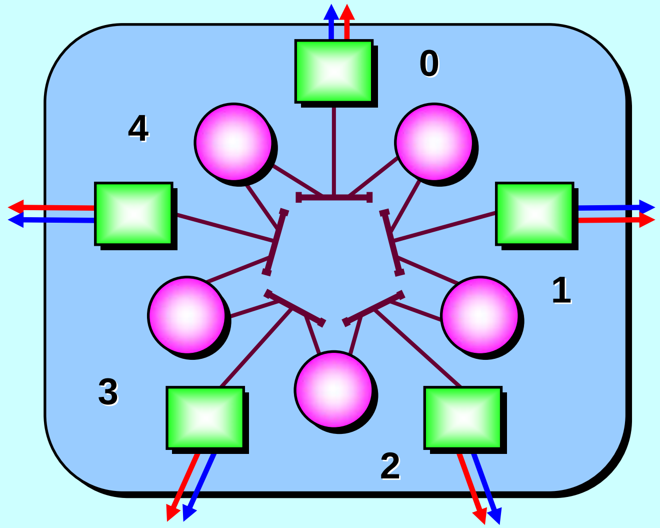
```
Philosophers (thinking, eating, eatBar) =
  ||| id : {0..(nPhils - 1)} @
    Phil (thinking.id, eating.id, eatBar.id)
```

… 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.




```
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 !!



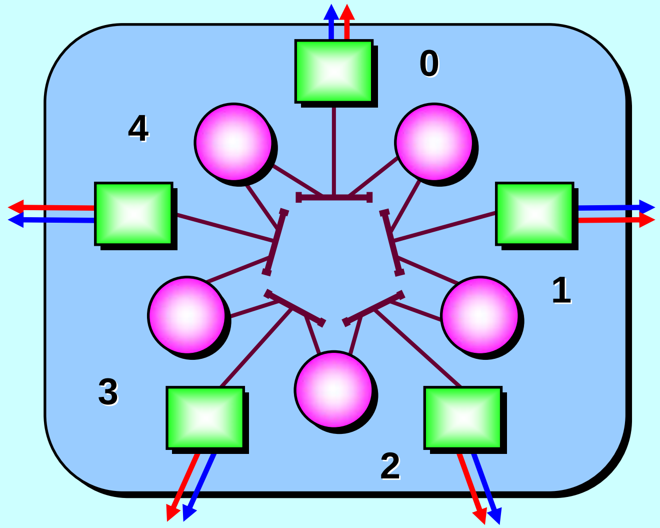

```
VAL INT nPhils IS 5:
```

```
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.

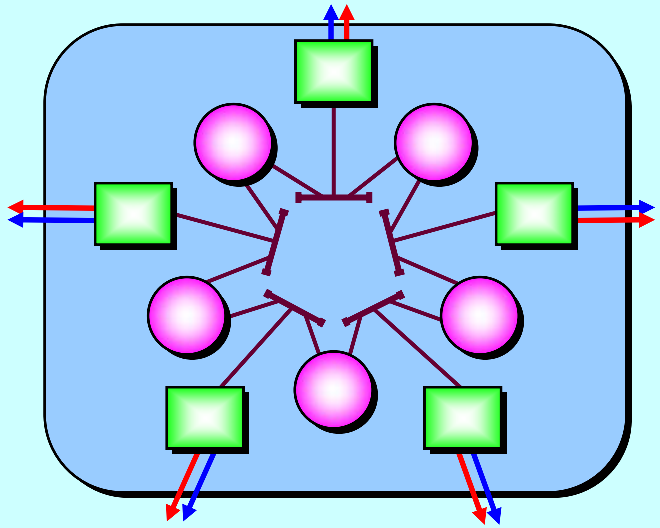



```
channel eatBarRight_r2_, eatBarLeft_r2_

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 !!

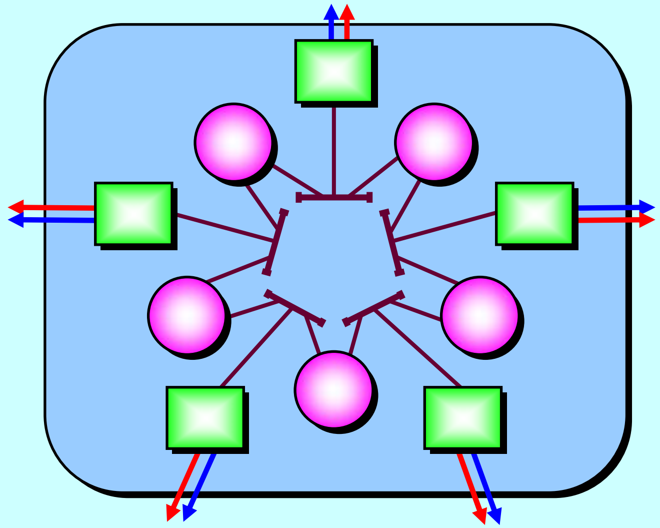
```
VAL INT nPhils IS 5:
```

```
nPhils = 5
```



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

```
channel eatBar_99_ : {0..(nPhils - 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
```

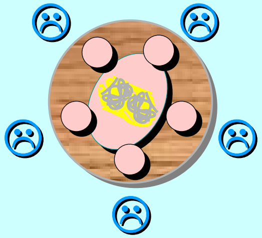
✓

```
VERIFY LIVELOCK.FREE College
```

✓ ☺ ☺ ☺

```
VERIFY NOT DETERMINISTIC.FD College
```

✓



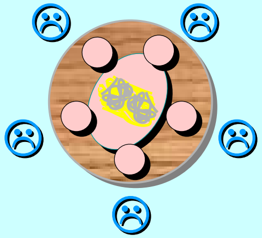
There is a problem though ☹

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 …

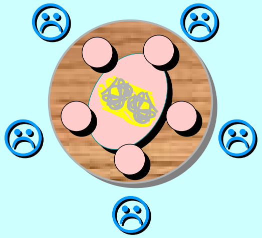
With our simpler college, we want to beat that scale! Further, we would like to verify a college of any number of philosophers … using induction.



Solving the problem ☺

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 *philospher-fork* pair. Next, build chains of *philospher-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.



Solving the problem ☺

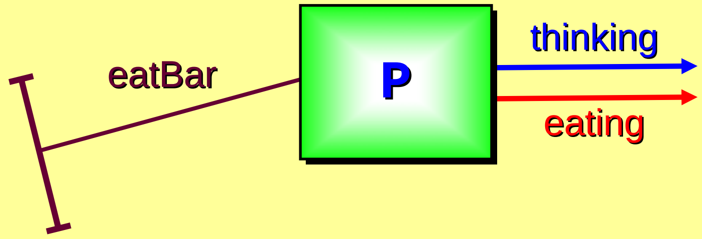
There are two further points that are needed: *hiding* and *compression*.

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.

Recall …

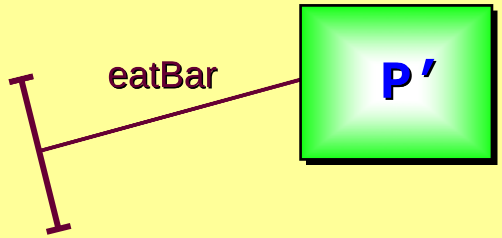
```
VERIFY PROC Phil (CHAN INT thinking!, eating!, BARRIER eatBar)
  WHILE TRUE
    SEQ
      thinking ! 0
      SYNC eatBar
      eating ! 0
      SYNC eatBar
:
```




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

Hide …

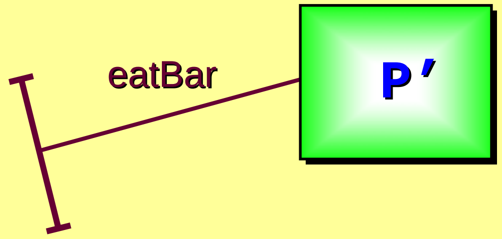
```
VERIFY PROC Phil. (BARRIER eatBar)
  CHAN INT thinking!, eating!:        -- channel *ends* only
  Phil (thinking!, eating!, eatBar)
:
```




```
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
:
```

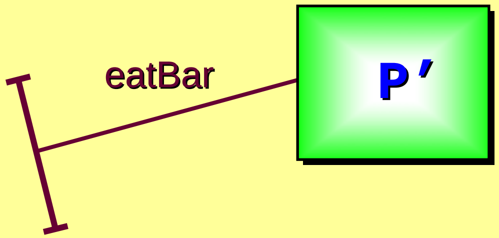



```
Phil’ (eatBar) =
  let
    Phil_0_ = eatBar -> eatBar -> Phil_0_
  within
    Phil_0_
:
```

… 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!:        -- channel *ends* only
  Phil (thinking!, eating!, eatBar)
:

VERIFY SIZE Phil

VERIFY SIZE Phil.
```

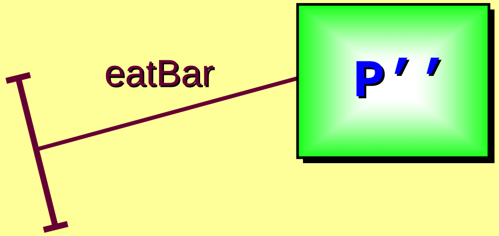


4 states, 4 transitions

… not won yet … need to compress!!

Compress …

```
VERIFY PROC Phil.. (BARRIER eatBar)
  NORMALISE                 -- reduce state machine to normal form
    Phil. (eatBar)
:

VERIFY SIZE Phil..
```



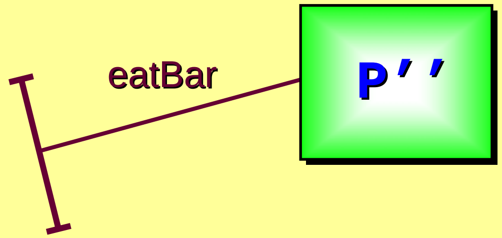
1 state, 1 transition

```
Phil'' (eatBar) = normalise (Phil' (eatBar))
```

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

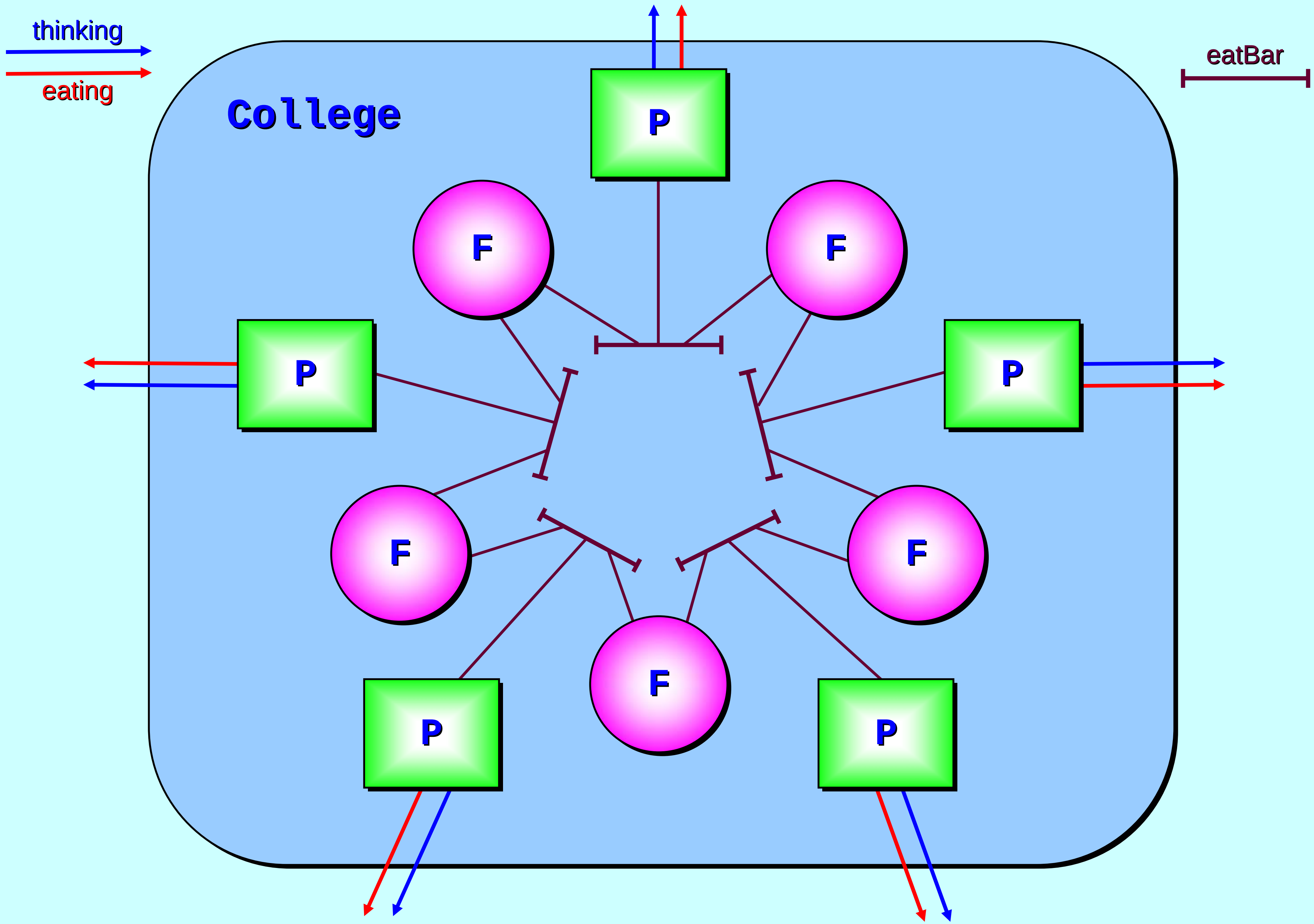
And is the same as …

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

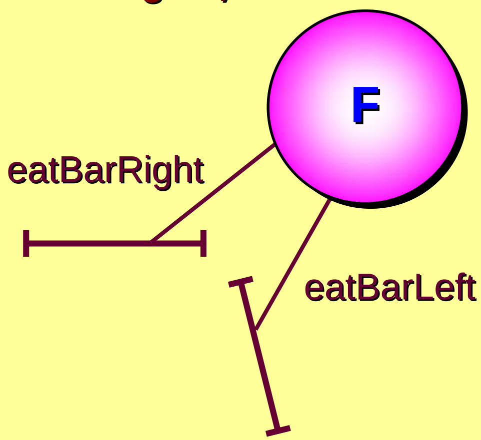



```
Phil'' (eatBar) =
  let
    Phil_0_ = eatBar -> Phil_0_
  within
    Phil_0_
:
```

… but without changing source code ☺

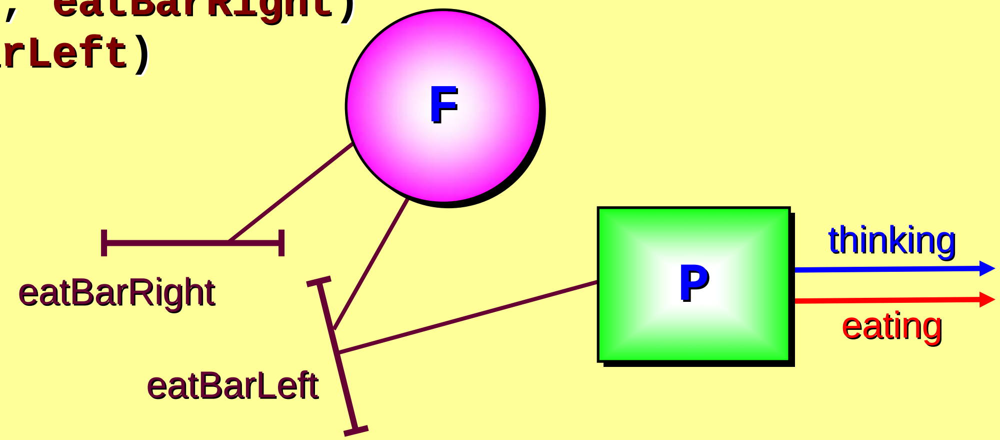

thinking
eating
College
eatBar
P
F
P
F
P
F
P
F
P
F

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

VERIFY SIZE Fork
```



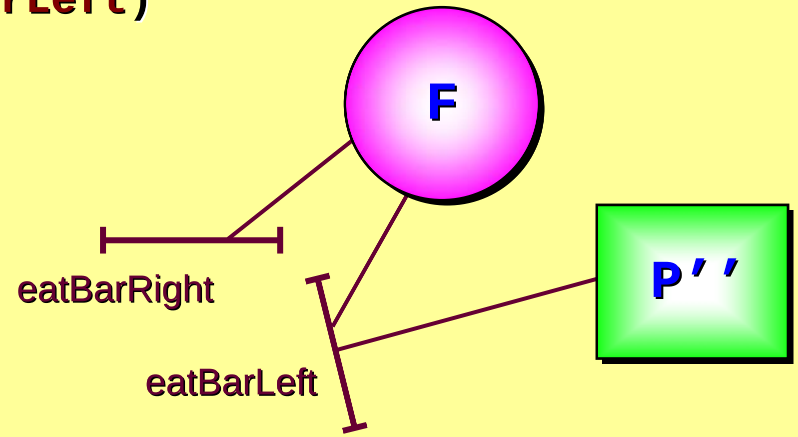
3 states, 4 transitions

```
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

```
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)
:
```

Generating the *CSP*$_M$ code for this requires an extra care …
(because FDR2 does something it shouldn't – claim!)

The following does not work correctly …

Now build a chain … using recursion

```
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?

```
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)*.

H(i) is:

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

We have *H(4)* and *H(5)*. Suppose *H(i)* for any *i >= 4*. Consider:

```
Chain (i+1, eatBarRight, eatBarLeft)
```

H(i+1)

This reduces to:

```
BARRIER eatBarMiddle:
PAR
  Chain (i, eatBarRight, eatBarMiddle)
  PhilFork. (eatBarMiddle, eatBarLeft)
```

By *H(i)*, this is EQUIVALENT.F to:

```
BARRIER eatBarMiddle:
PAR
  Chain (4, eatBarRight, eatBarMiddle)
  PhilFork. (eatBarMiddle, eatBarLeft)
```

But this is the same as:

```
Chain (5, eatBarRight, eatBarLeft)
```

Which, by *H(5)*, is EQUIVALENT.F to:

```
Chain (4, eatBarRight, eatBarLeft)
```

H(i) **is:**

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

Clearly *H(4)* and, by model checking, *H(5)*.

We have just shown that, for any *i >= 4*, *H(i)* implies *H(i+1)*.

By induction therefore, for all *i >= 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*?

```
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 `CollegeChain`s 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*).

```
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 `CollegeChain`s with size equal to or greater than *2* are *deadlock free*. Of course, with no reporting, they are hopelessly *livelocked* !

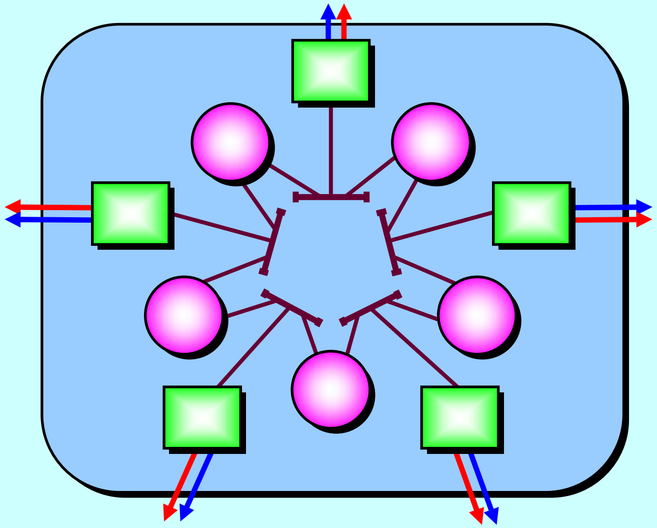
So … what about *reporting Colleges*?

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.*

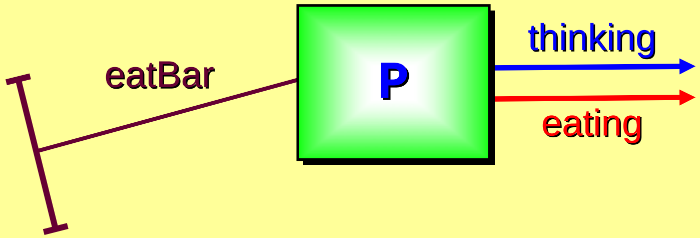
The following argument shows that a college with external reports is also *livelock free* …

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!

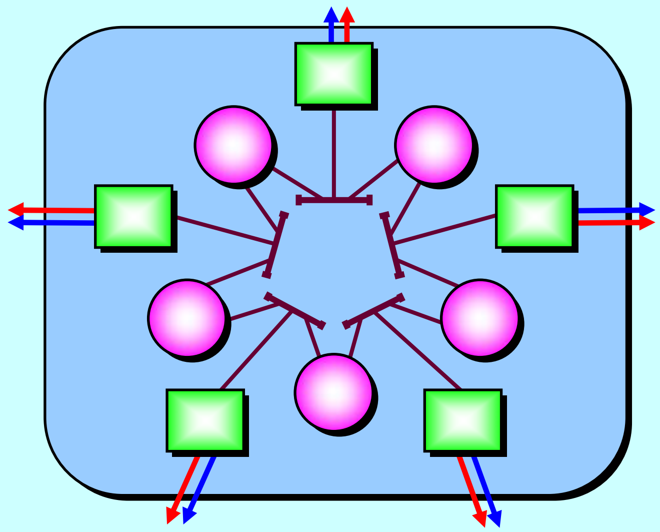


```
VERIFY PROC Phil (CHAN INT thinking!, eating!, BARRIER eatBar)
  WHILE TRUE
    SEQ
      thinking ! 0
      SYNC eatBar
      eating ! 0
      SYNC eatBar
:
```



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*** `eatBar`s. 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:

```
--* A chain of (length^level) philospher-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)
:
```

Finally, the Brute Force Approach

```
--* A chain of (length^level) philospher-fork pairs.
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, _, _)    ✗
VERIFY Chain2 (2, 2, _, _) EQUIVALENT.F Chain2 (3, 2, _, _)    ✓
VERIFY Chain2 (1, 10, _, _) EQUIVALENT.F Chain2 (2, 10,_, _)   ✓
```

And the *Colleges* …

Finally, the Brute Force Approach

```
--* 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 (0, 10, _, _)       ✓
VERIFY DEADLOCK.FREE.F CollegeChain2 (1, 10, _, _)       ✓
VERIFY DEADLOCK.FREE.F CollegeChain2 (10, 10,_, _)       ✓
VERIFY DEADLOCK.FREE.F CollegeChain2 (100, 10,_, _)      ✓
VERIFY DEADLOCK.FREE.F CollegeChain2 (1000, 10,_, _)     ✓
VERIFY DEADLOCK.FREE.F CollegeChain2 (2000, 10,_, _)     ✓
VERIFY DEADLOCK.FREE.F CollegeChain2 (2500, 10,_, _)
```

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 FDR2: *"broken pipe"* on the terminal launch window.

Finally, the Brute Force Approach

```
--* 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

occam-π / CSP_M

occam-π teams well with CSP_M to provide efficient executables and rich formal analysis.

This presentation reflects a proposal to extend 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.

* **http://portal.acm.org/citation.cfm?id=53255**

[A.W.Roscoe and C.A.R.Hoare, 1988]

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.

https://moodle.kent.ac.uk/external/course/view.php?id=31

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

Process Orientation

CSP / π-calculus
occam-π / JCSP

Because it scales

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 …

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 … ?

Observation

Any questions?

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

- they quickly develop a correct and intuitive understa… the primitive mechanisms *(e.g. processes, communication, sy… n, networks)* and higher level patterns *(e.g. client-server, ph… , I/O-PAR)* … ?

- they can use those primitives and patter… same fluency as they use serial computing primitives, without tr… dark hazards … ?

- they can develop their own pat… the standard ones don't apply … ?

- they can use formal met… y good behaviour *(e.g. freedom from deadlock and livelock… ness)*, without training in the underlying mathematics *(proc… , denotational semantics)* … ?

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

Yes, we can!