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

Matt Pedersen[a] and Peter Welch[b]
[a]School of Computer Science, UNLV, USA
[b]School of Computing, University of Kent, UK

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

Curricula for Concurrency and Parallelism
SPLASH 2010, 17th. Oct

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

Not only

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

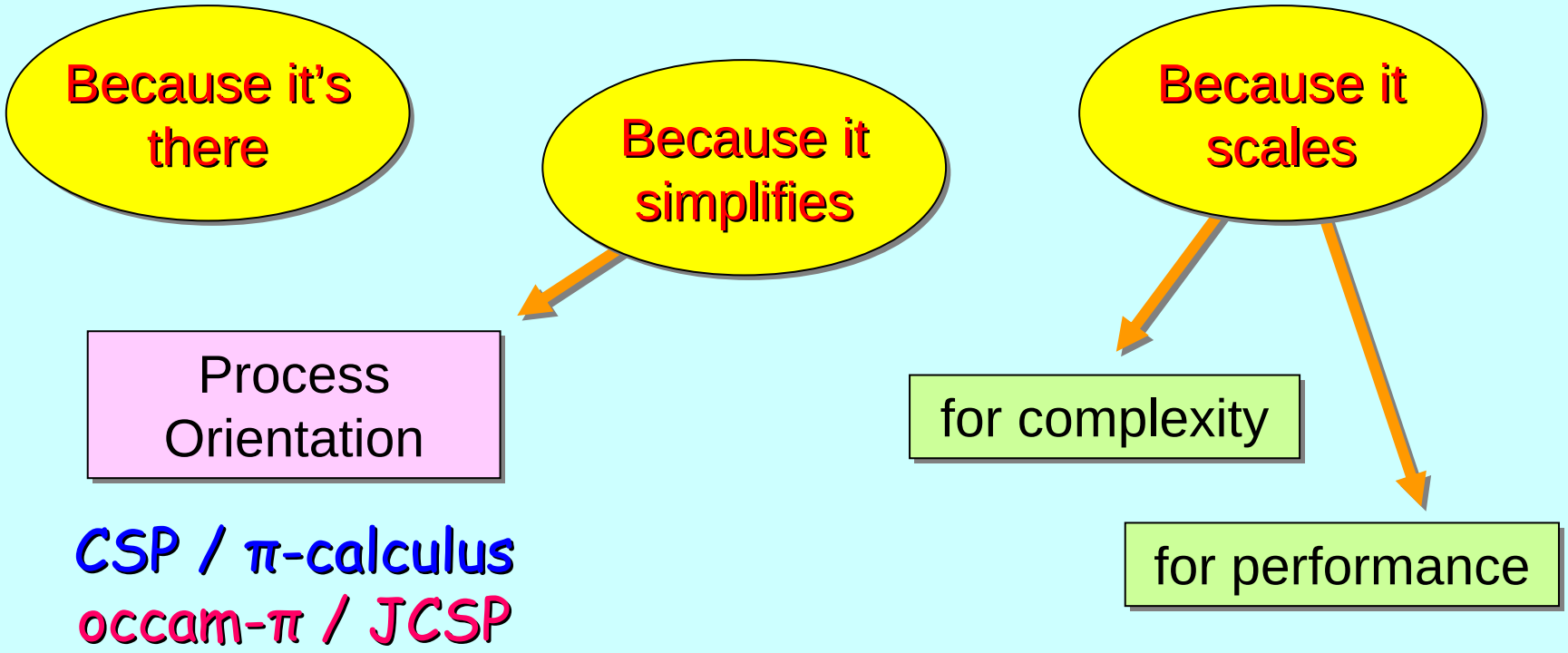
But also

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

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

Not only

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



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 of software engineering

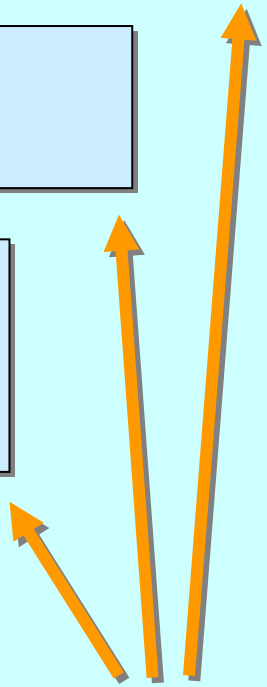
All are important. All are simple. All are available.

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.***



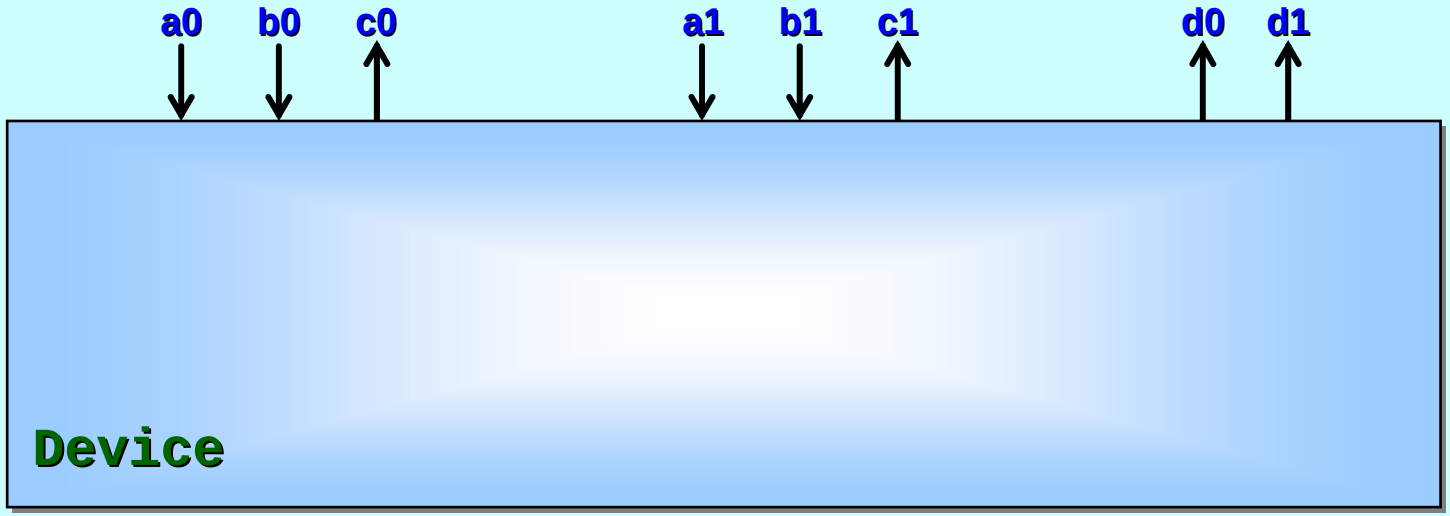
But also

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

Example: *autonomous robot component*

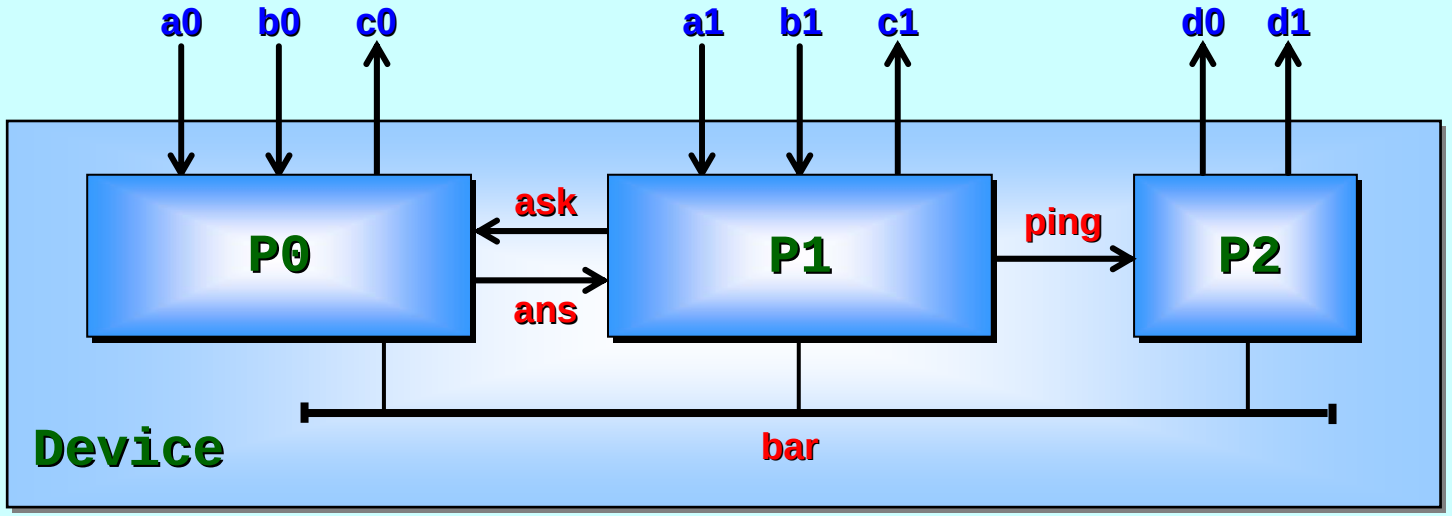
The following example has been developed from one first worked through in a single lesson of a graduate class in concurrency at UNLV in the spring of 2010.

Example: *autonomous robot component*



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

Example: *autonomous robot component*

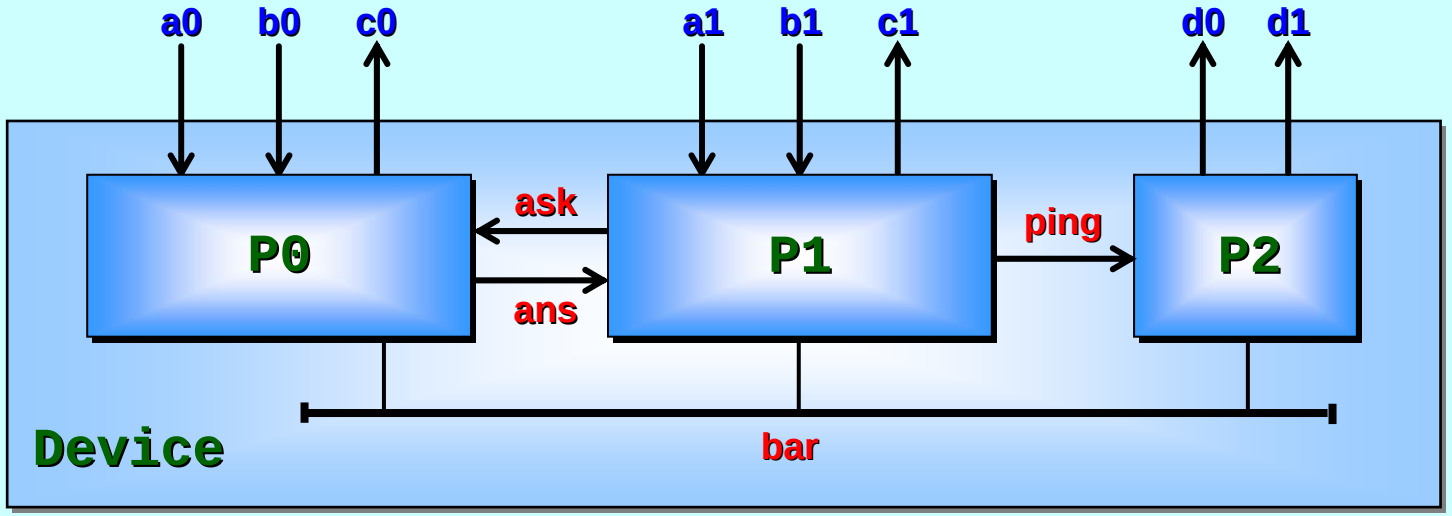


Device : real-time controller for 8 channels (4 input, 4 output).

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

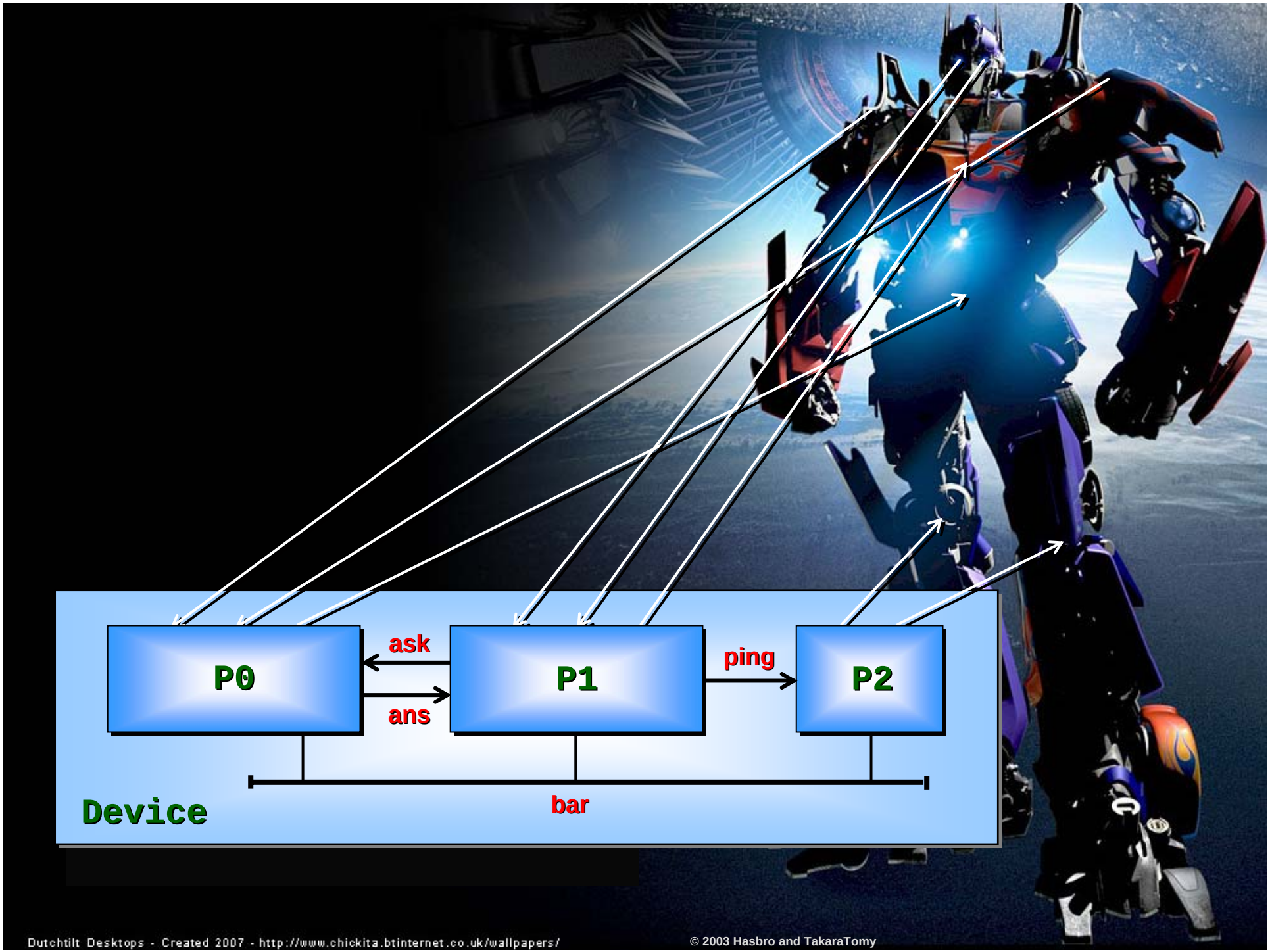
They exchange information over internal channels (**ask**, **ans**, **ping**) and all coordinate actions with an internal barrier (**bar**).

Example: *autonomous robot component*

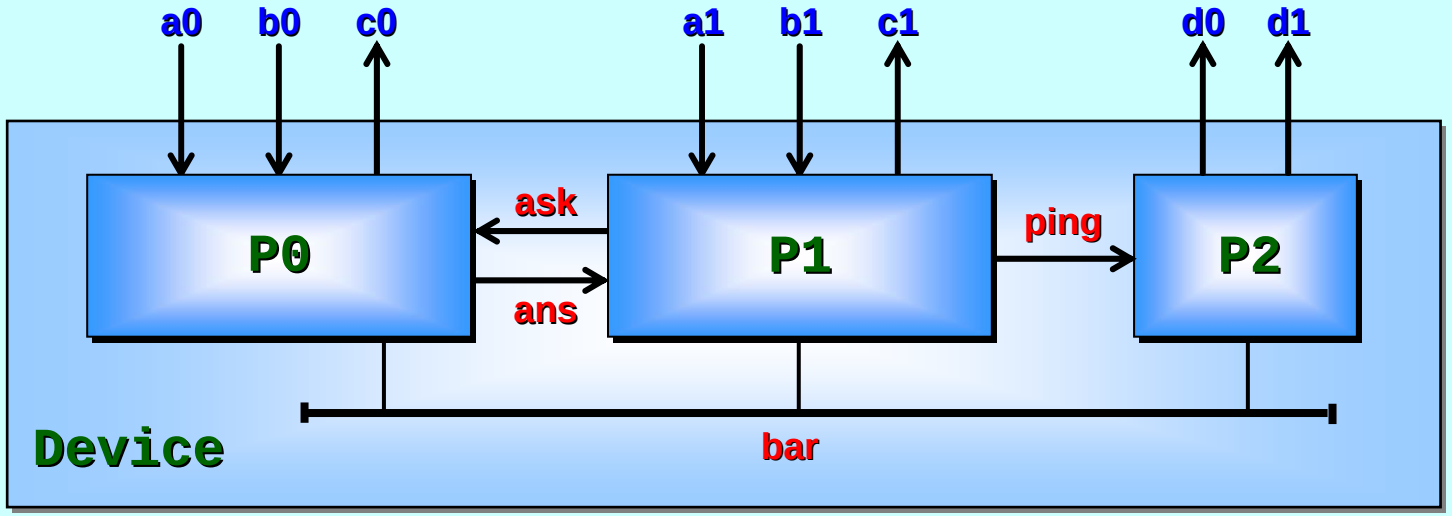


CSP semantics apply. *Channel communication* is unbuffered (sender waits for receiver and vice-versa). Any process *reaching a barrier* waits for *all* processes to *reach the barrier*.

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


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

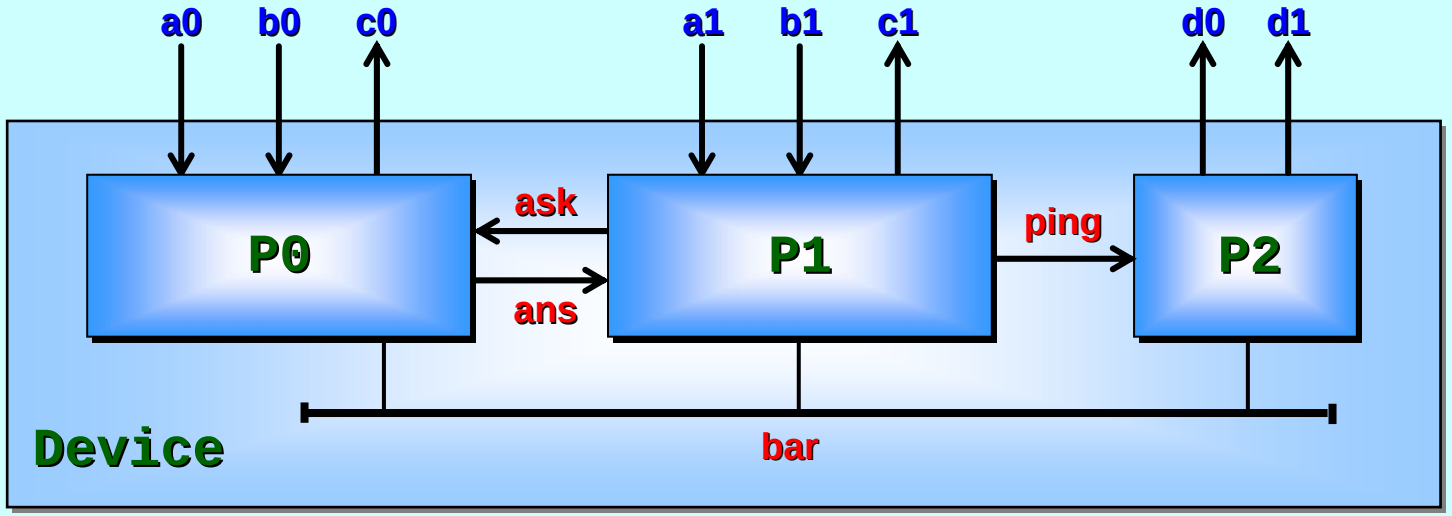
Behaviour: *two representations*



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: *two representations*

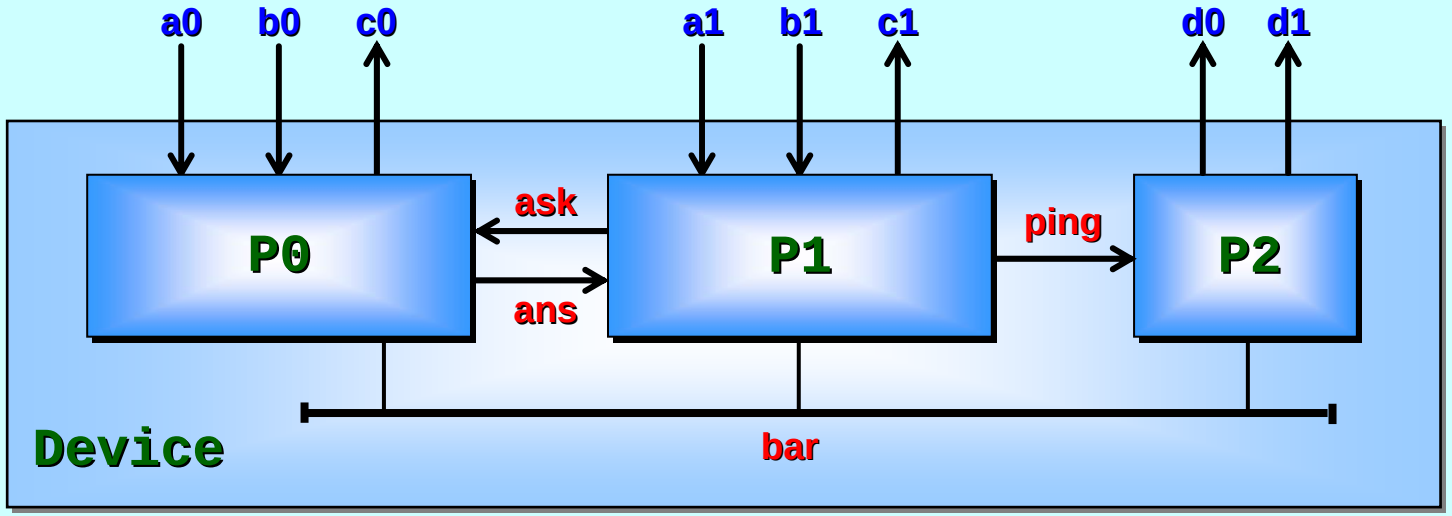


occam-π: for compiling to a runnable sy…
[memory overheads <= 32 b… …ds of order…

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

C…
[FD… …r (simple) formal reasoning.]

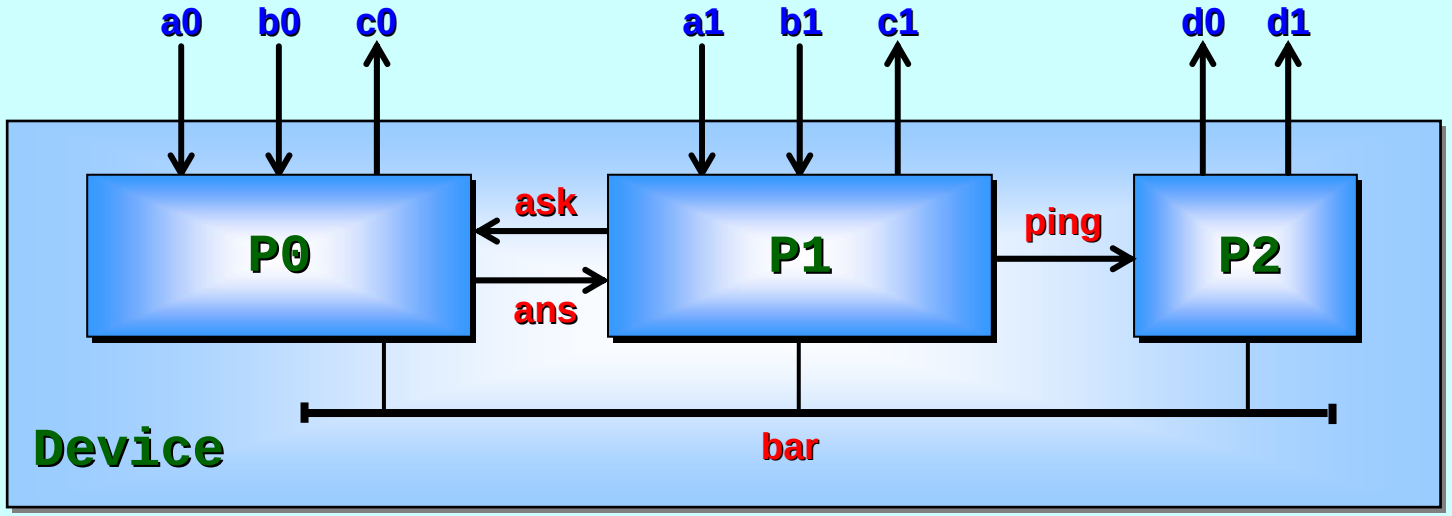
Behaviour: *what are we looking for?*



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?*

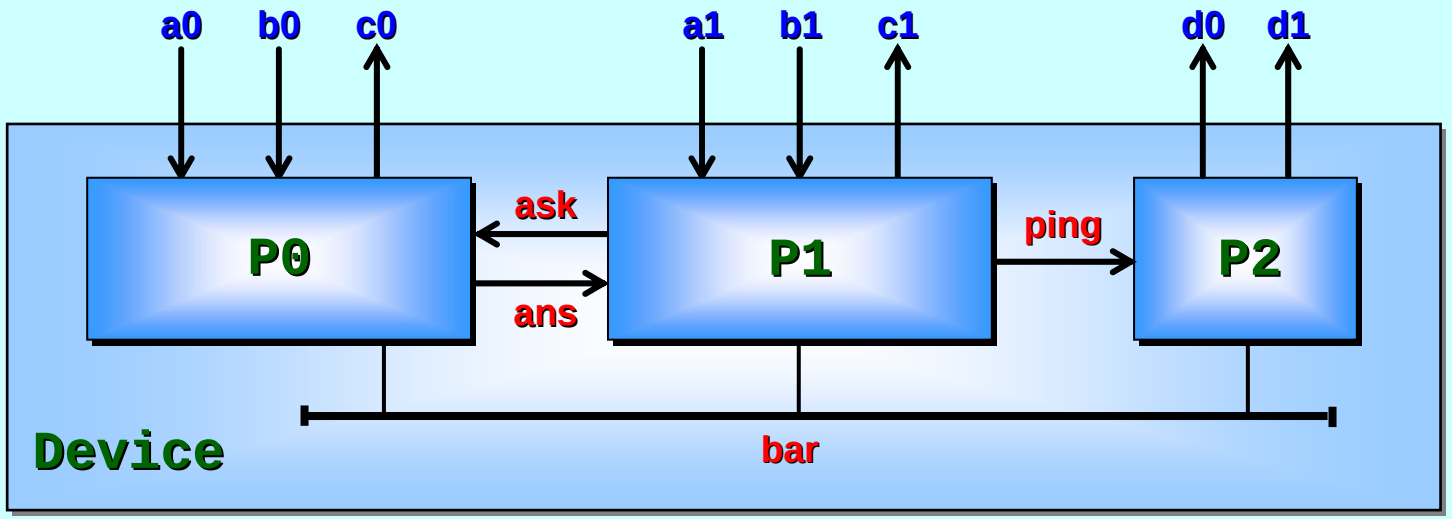


safety: *might* it ever engage in an incorrect sequence of external signals?

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

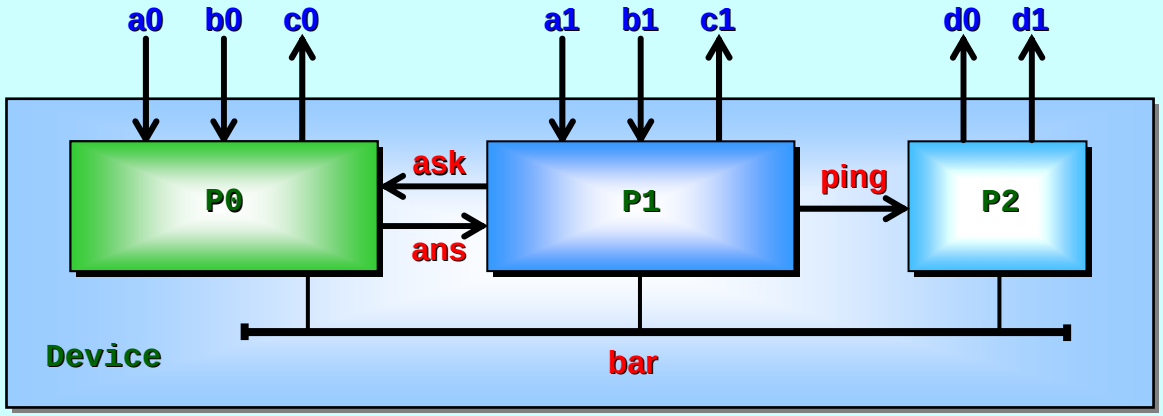
[Some specs allow alternative sequences to be performed – all are correct, but an implementation must only do one and is free to choose.]

Behaviour: occam-π *(executable)*



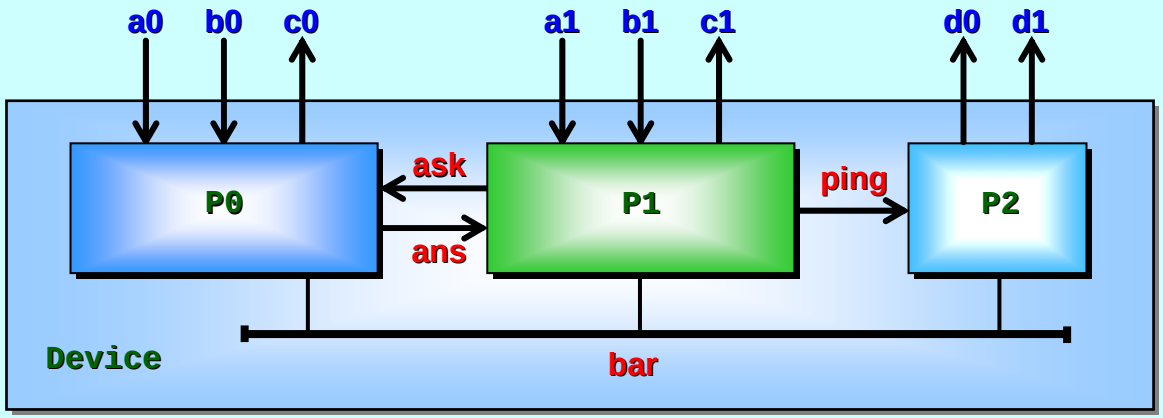
For the behaviour analysis in this example, data values and computations are not relevant. For simplicity, they are omitted in these codes, with all message content abstracted to zero.

Behaviour: occam-π *(executable)*



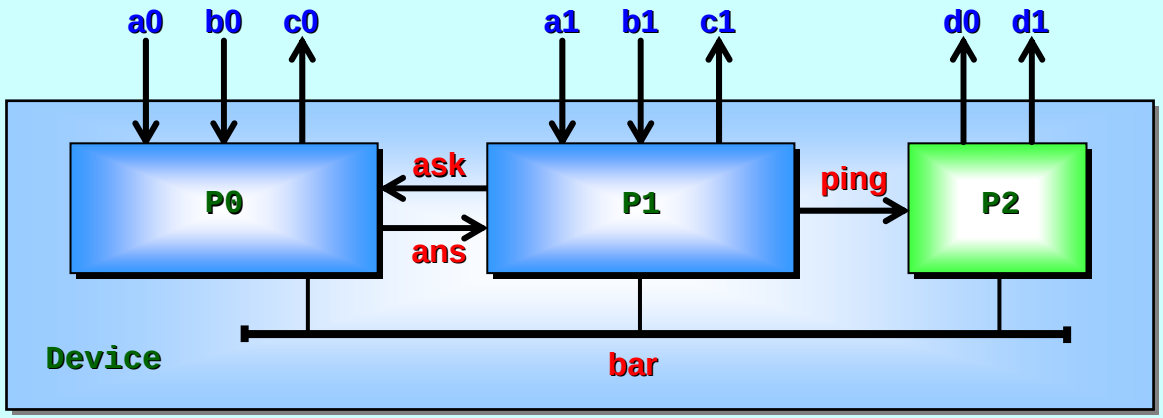

```
PROC P0 (CHAN INT a0?, b0?, c0!, ask?, ans!, BARRIER bar)
  WHILE TRUE
    INT x, y, z:
    SEQ
      ask ? x      -- take question
      a0 ? y
      ans ! 0      -- return answer (will depend on x and y)
      b0 ? z
      SYNC bar     -- wait for the others
      c0 ! 0
:
```

Behaviour: occam-π *(executable)*



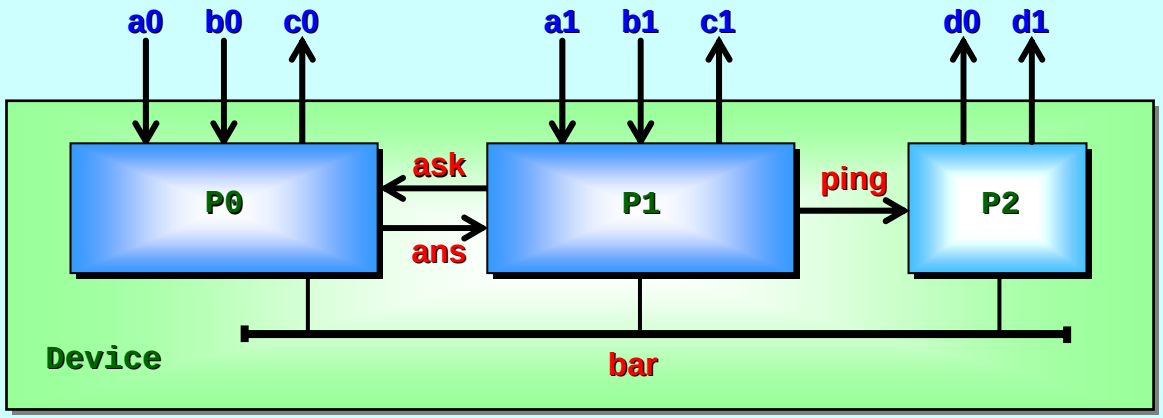

```
PROC P1 (CHAN INT a1?, b1?, c1!, ask!, ans?, ping!,
         BARRIER bar)
  WHILE TRUE
    INT x, y, z:
    SEQ
      ask ! 0      -- ask question
      ans ? x      -- wait for answer
      a1 ? y
      b1 ? z
      SYNC bar     -- wait for the others
      c1 ! 0
      ping ! 0     -- update neighbour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

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

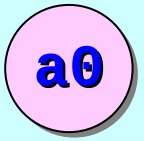
Behaviour: occam-π (executable)

```
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]
```

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?

(* 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
:
```

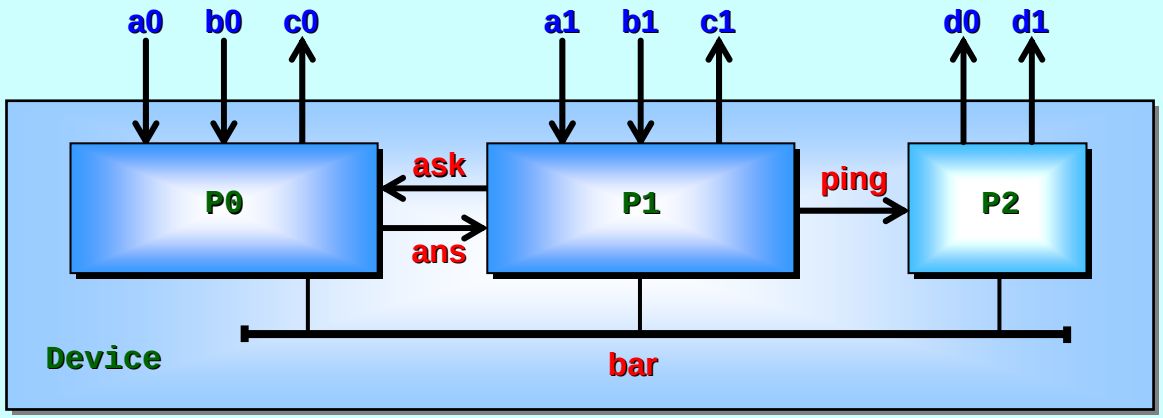
```
PROC P1 (CHAN INT a1?, b1?, c1!, ask!, ans?, ping!,
         BARRIER bar)
  WHILE TRUE
    INT x, y, z:
    SEQ
      ask ! 0      -- ask question
      ans ? x      -- wait for answer
      a1 ? y
      b1 ? z
      SYNC bar     -- wait for the others
      c1 ! 0
      ping ! 0     -- update neighbour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

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

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

Formal

Behaviour: CSP-M (verifyable)

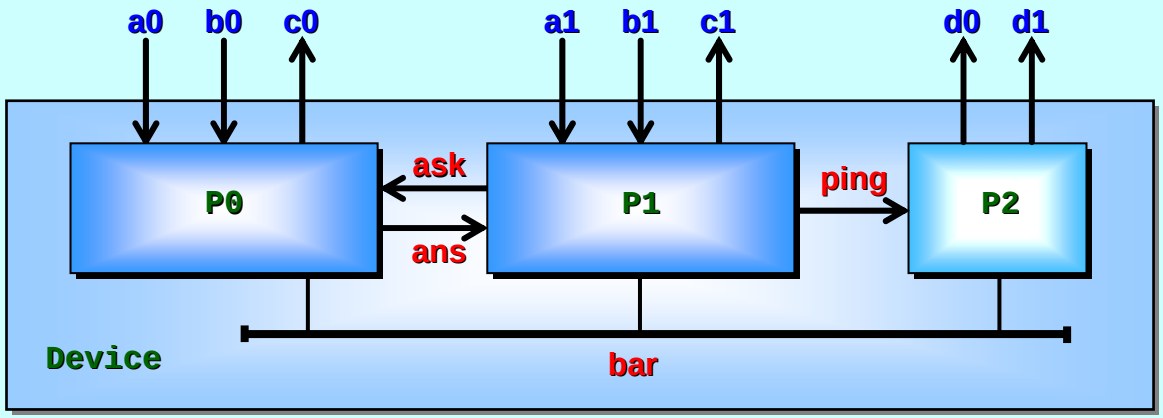


We can formally verify the previous intuition (which was only about the opening behaviour of the system) and answer the open questions (and more) about its continuous behaviour with a *CSP* representation.

We use *CSP-M*, the machine readable form used by the FDR2 model checker. *CSP-M* is a *declarative* (*functional*) language – *loops* map to *tail recursions*. Students who enjoy programming have no problem learning new syntax *(it's particularly easy when the semantics remain unchanged)* – but they need to be told why!

Formal

Behaviour: CSP-M (verifyable)



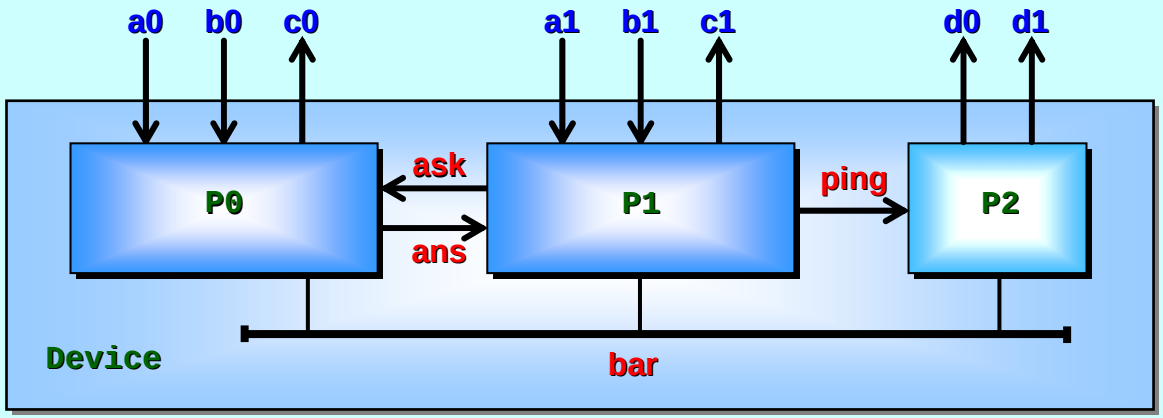
CSP-M lets us abstract the channel communications further by omitting the data sent (always zero in our example) and the direction of communication (irrelevant here).

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

In the following CSP-M, we further simplify things by omitting process parameters and accessing all channels from global declaration. *[We could have done this with the occam-π …]*

Formal

Behaviour: CSP-M (verifyable)



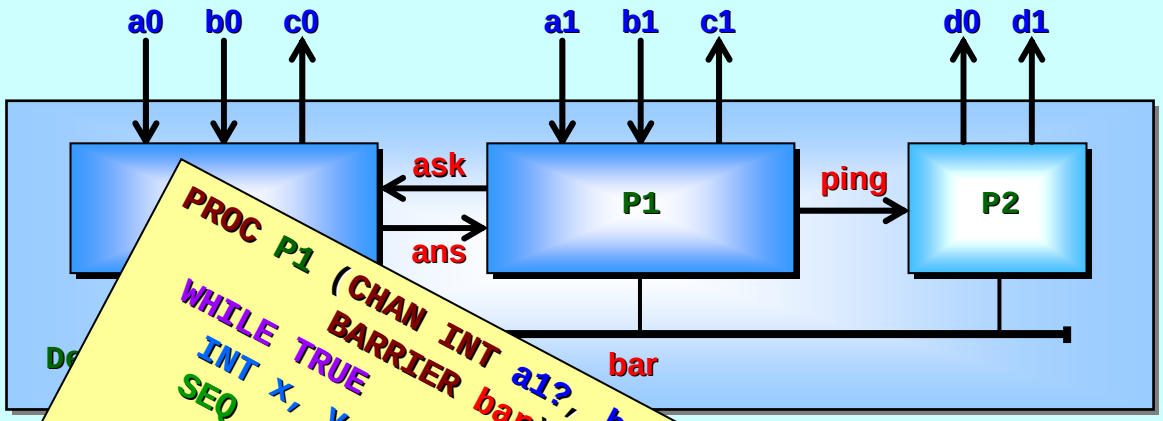

```
channel a0, b0, c0, a1, b1, c1, d0, d1, ask
```

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

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

Formal

Behaviour: CSP-M (verifyable)




```
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 answ
      a1 ? y
      b1 ? z
      SYNC bar
      c1 ! 0                 -- the others
      ping ! 0               -- update neighbour
:
```

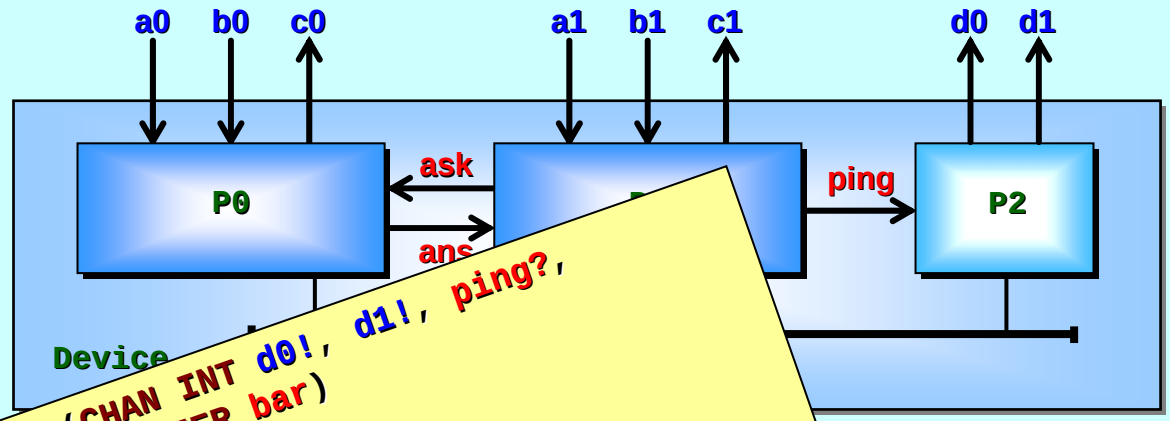
```
channel a                                    sk, ans, ping, bar
```

```
P0 = ask
```

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

Formal

Behaviour: CSP-M *(verifyable)*

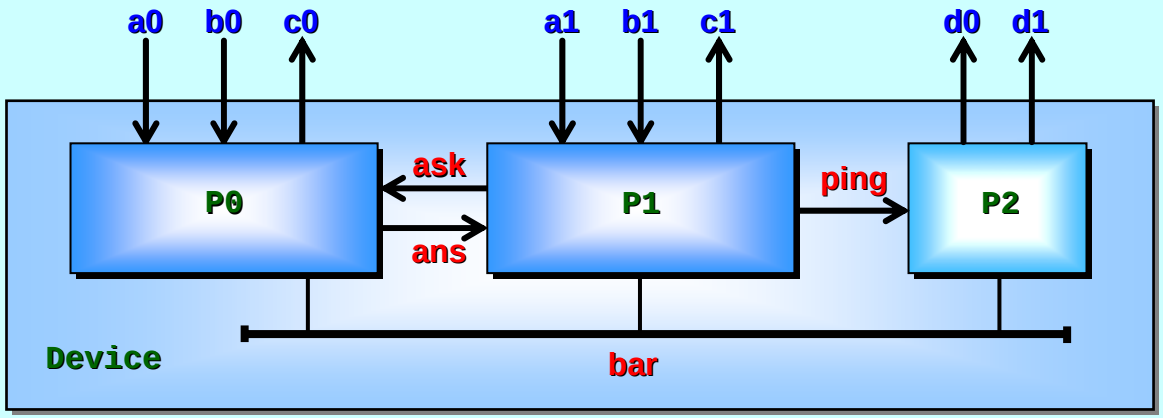



```
PROC P2 (CHAN INT d0!, d1!, ping?,
         BARRIER bar)
  WHILE TRUE
    INT x:
    SEQ
      SYNC bar          -- wait for others
      d0 ! 0            -- receive update
      ping ? x          -- wait for others
      SYNC bar
      d1 ! 0            -- receive update
      ping ? x
```

```
channe                                          ask, ans, ping, bar
P0 = ask                                        -> P0
P1 = ask                                        -> bar -> c1 -> ping -> P1
P2 = bar -> d0 -> ping -> bar -> d1 -> ping -> P2
```

Formal

Behaviour: CSP-M (verifyable)




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

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

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

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

```
P0P1 = (P0 [| {ask, ans, bar} |] P1) \ {ask, ans}
Device = (P0P1 [| {ping, bar} |] P2) \ {ping, bar}
```

Formal

Behaviour: CSP-M *(verifyable)*

Loading the system below into FDR2, we discover straight away that **Device** is *free from deadlock and livelock* – just click the buttons!

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

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

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

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

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

Device = (P0P1 [| {ping, bar} |] P2) \ {ping, bar}
```

Formal

Behaviour: CSP-M *(verifyable)*

Intuition

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

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.

```
T0 = a0 -> b0 -> a1 -> b1 -> d0 -> c0 -> c1 -> STOP
T1 = a0 -> b0 -> a1 -> d0 -> b1 -> c0 -> c1 -> STOP
```

And ask: does each *trace refine* **Device**?

Process **P** *trace refines* **Q** if all *traces* of **P** are *traces* of **Q**.

```
Q [T= P
```

Formal

Behaviour: CSP-M *(verifyable)*

Intuition

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

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.

```
T0 = a0 -> b0 -> a1 -> b1 -> d0 -> c0 -> c1 -> STOP
T1 = a0 -> b0 -> a1 -> d0 -> b1 -> c0 -> c1 -> STOP
```

FDR2 reports that **T0** *trace refines* **Device** … but **T1** *does* not – which confirms our intuition. ☺☺☺

Device [T= T0 ✓ **Device [T= T1** ✗

Formal

Behaviour: CSP-M *(verifyable)*

Intuition

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

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.

```
T0 = a0 -> b0 -> a1 -> b1 -> d0 -> c0 -> c1 -> STOP
T1 = a0 -> b0 -> a1 -> d0 -> b1 -> c0 -> c1 -> STOP
```

Clearly, **[a0, b0, a1, b1, d0, c0, c1]** is a trace of **T0**.
Therefore, it is also a trace of **Device**.

```
Device [T= T0
```

✓

Formal

Behaviour: CSP-M *(verifyable)*

Intuition

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

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.

```
T0 = a0 -> b0 -> a1 -> b1 -> d0 -> c0 -> c1 -> STOP
T1 = a0 -> b0 -> a1 -> d0 -> b1 -> c0 -> c1 -> STOP
```

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**.

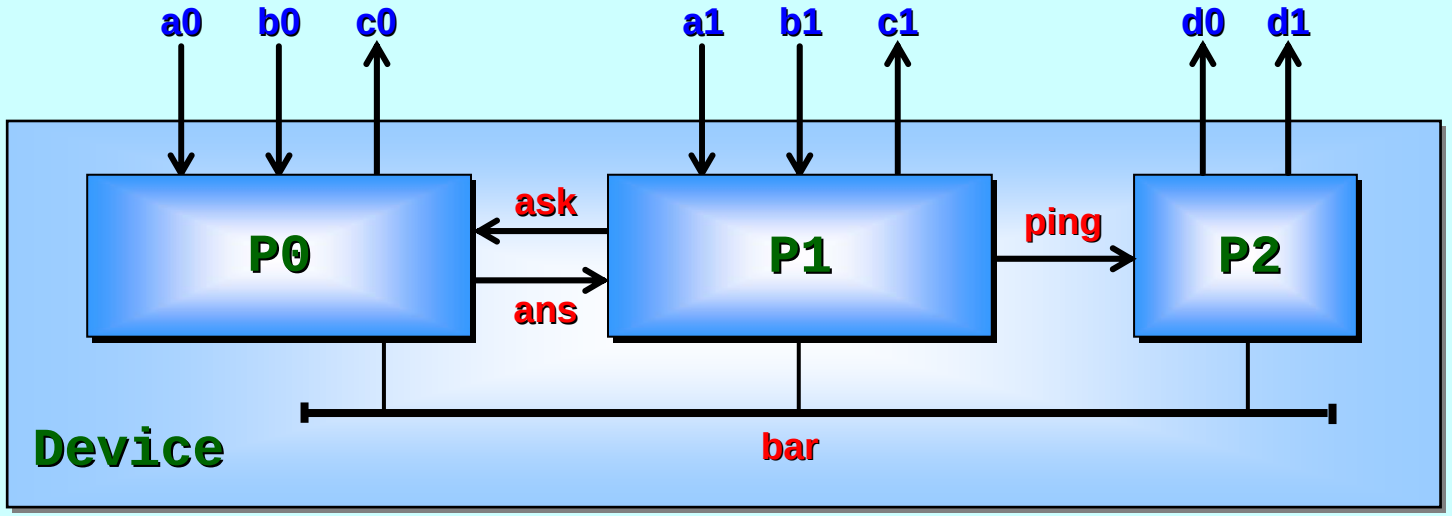
```
Device [T= T1
```

✗

Formal

Behaviour: CSP-M *(verifyable)*

Safety



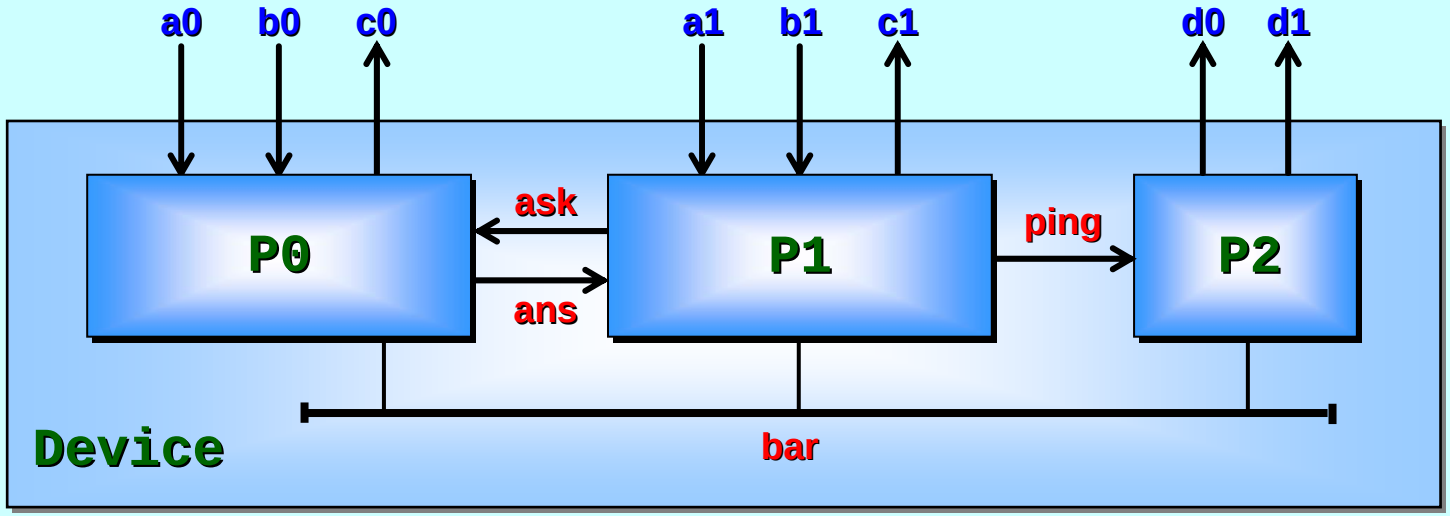
Let's ask a more difficult question about the continuous running of the system. Suppose the robot would do something *very bad* if its controller **Device** were ever to signal *twice* on **a0** without a signal on **d0** or **d1** *in between*. Might this ever happen?

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

Formal

Behaviour: CSP-M (verifyable)

Safety



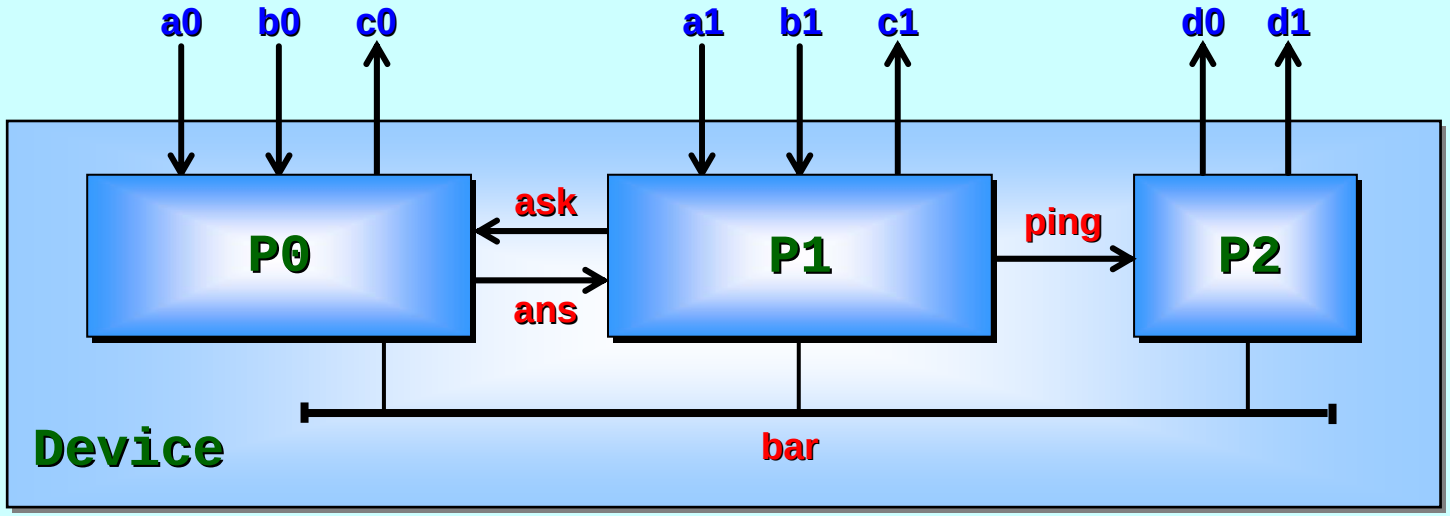

```
Check (n) =
  if n >= 2 then STOP else
    a0 -> Check (n+1) [] d0 -> Check (0) [] d1 -> Check (0) []
    a1 -> Check (n)   [] b0 -> Check (n) [] b1 -> Check (n) []
    c0 -> Check (n)   [] c1 -> Check (n)
```

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

Formal

Behaviour: CSP-M *(verifyable)*

Safety



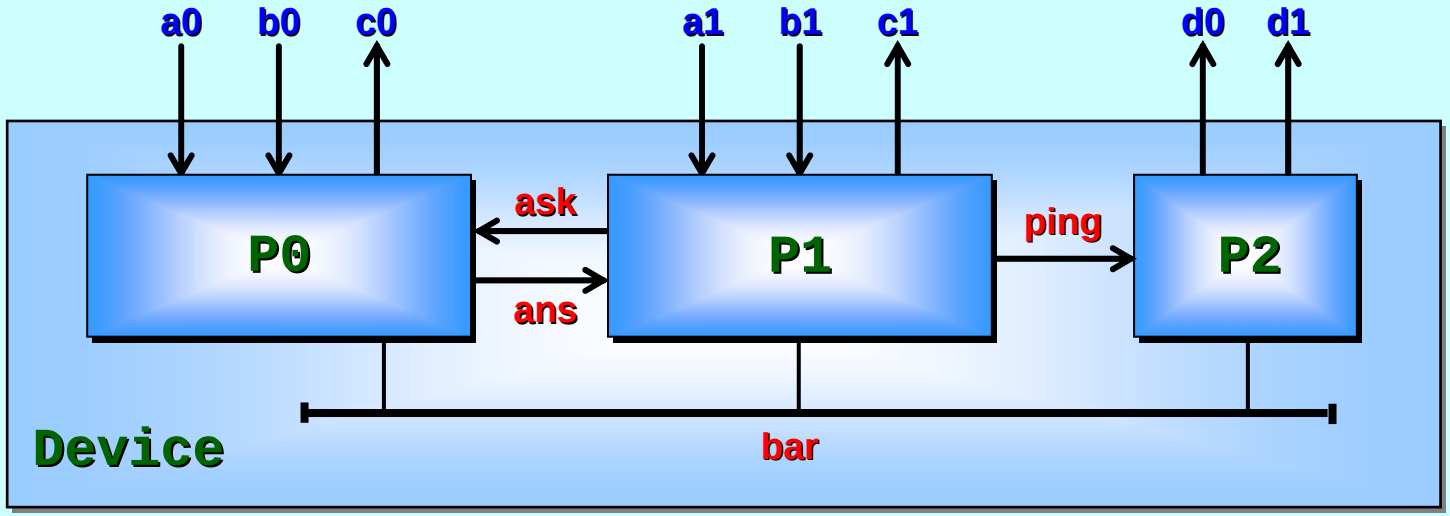

```
Check (n) =
  if n >= 2 then STOP else
    a0 -> Check (n+1) [] d0 -> Check (0) [] d1 -> Check (0) []
    a1 -> Check (n)   [] b0 -> Check (n) [] b1 -> Check (n) []
    c0 -> Check (n)   [] c1 -> Check (n)
```

The operator "**[]**" means wait for one or more of the operand processes *to become able* to run … choose one of them and run.

Formal

Behaviour: CSP-M *(verifyable)*

Safety



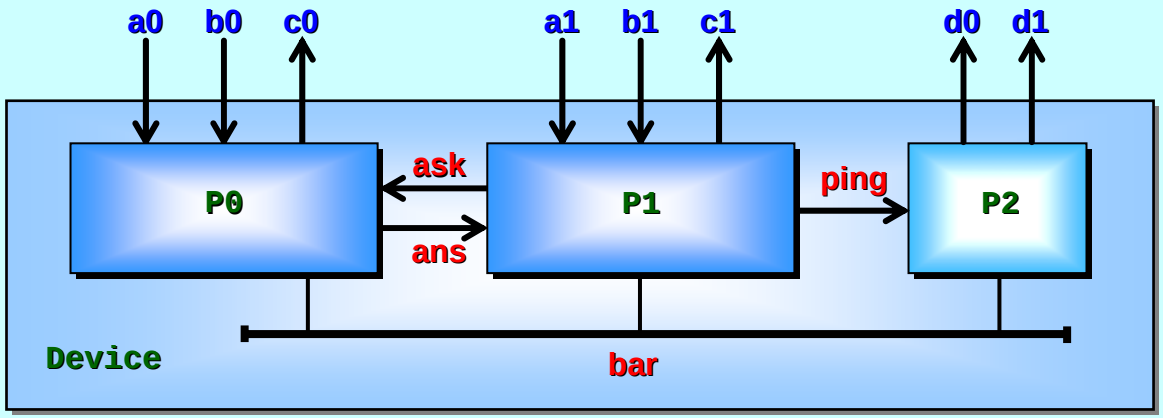

```
Check (n) =
  if n >= 2 then STOP else
    a0 -> Check (n+1) [] d0 -> Check (0) [] d1 -> Check (0) []
    a1 -> Check (n)   [] b0 -> Check (n) [] b1 -> Check (n) []
    c0 -> Check (n)   [] c1 -> Check (n)
```

The parameter to **Check** records how many **a0** signals have been received since the last **d0** or **d1**, stopping if this reaches 2.

Formal

Behaviour: CSP-M (verifyable)

Safety




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

```
CheckDevice =
  Device [| {a0, b0, c0, a1, b1, c1, d0, d1} |] Check (0)
```

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

FDR2 reports **CheckDevice** is deadlock free.

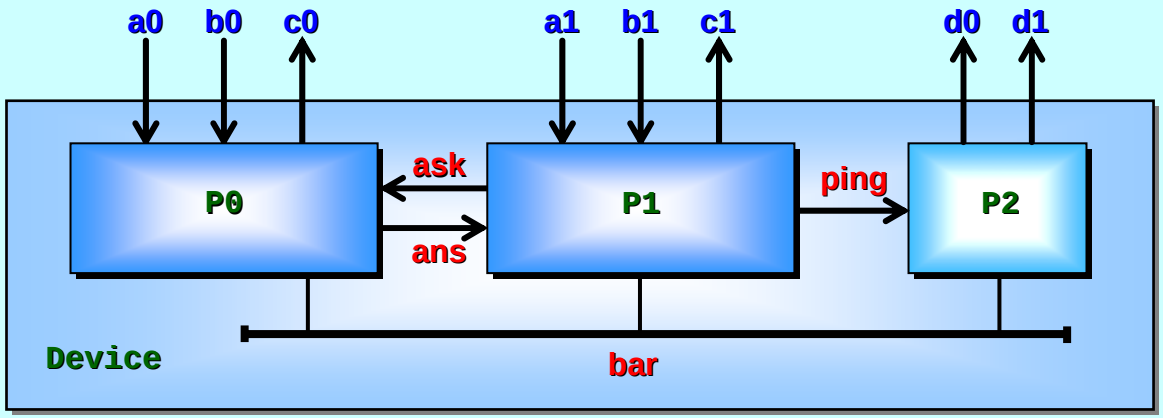
Therefore, **Check (0)** never stops *(& the bad thing can't happen)*.

Q.E.D.

Formal

Behaviour: CSP-M *(verifyable)*

Safety



Note: protocol checking monitors, such as **Check**, are sometimes used live to ensure adherence at run-time (e.g. in device drivers). We are using **Check** purely for static analysis – it has no role at run-time and, therefore, no impact on performance.

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

FDR2 reports **CheckDevice** is deadlock free.

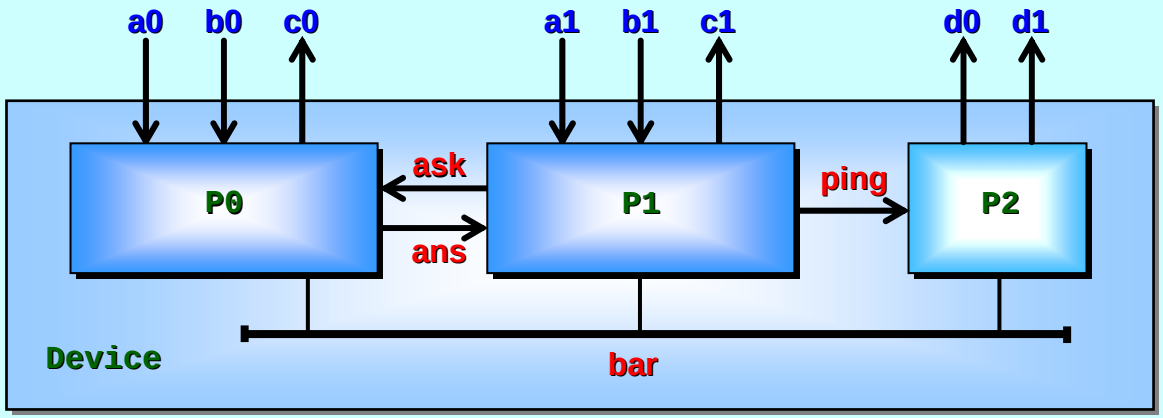
Therefore, **Check (0)** never stops *(& the bad thing can't happen)*.

Q.E.D.

Formal

Behaviour: CSP-M (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.

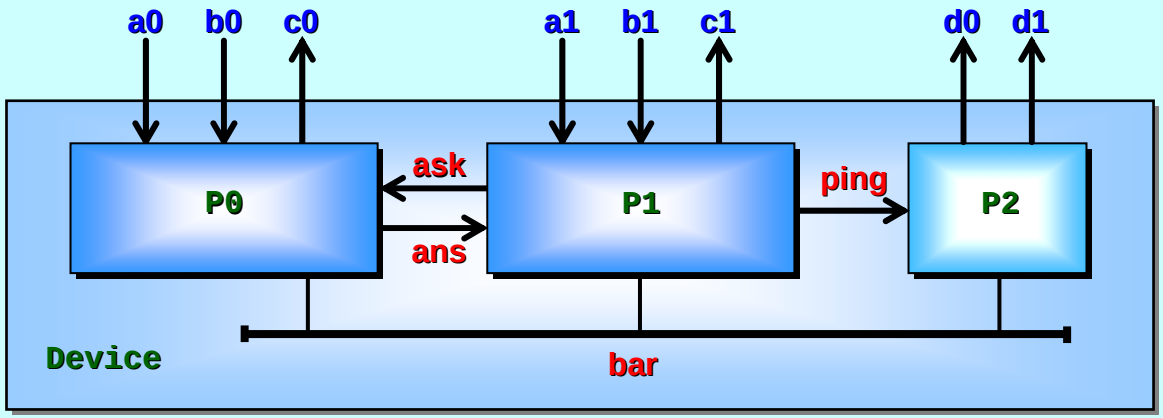
A *CSP failure* is a state that a system reaches (represented by its *trace* to that point) where it *may refuse to synchronise* with its environment on some given set of events.

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: CSP-M (verifyable)

Liveness



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

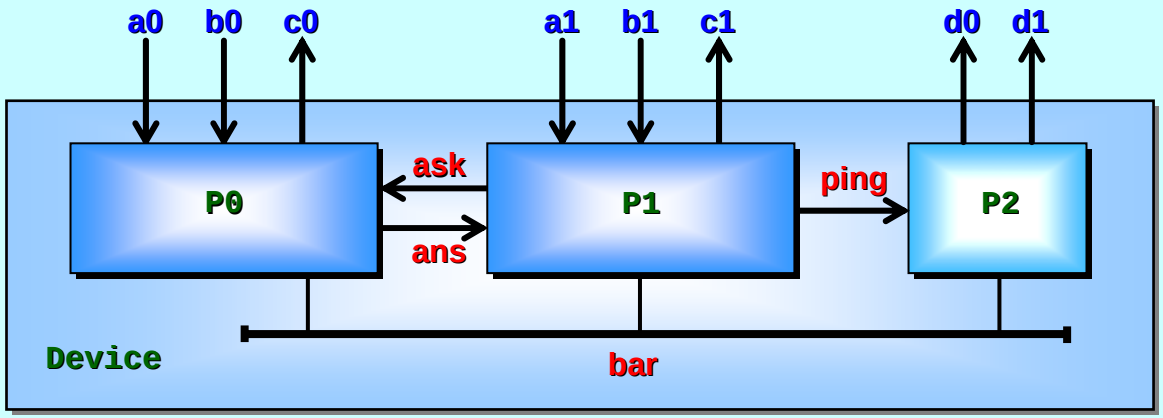
Process **P** *failure refines* **Q** if (all its *traces* are *traces* of **Q**) and (all its *failures* are *failures* of **Q**).

This is a powerful statement! **P** can only do *traces* of **Q** (so its safe). More: the *failures* of **P** are allowed by **Q**. If **P** and **Q** execute the same trace to a state where their environment offers a set of events that **Q** will not refuse, then **P** also will not refuse.

Formal

Behaviour: CSP-M (verifyable)

Liveness



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

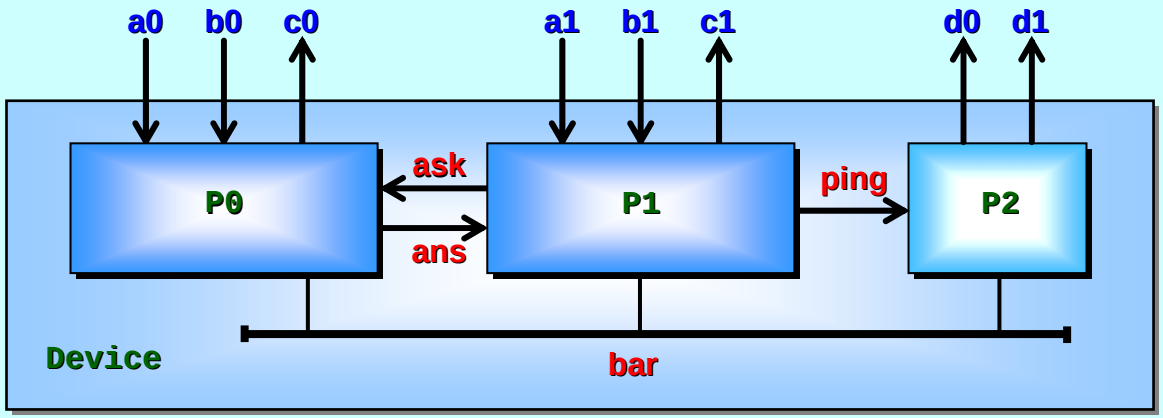
Process **P** *failure refines* **Q** if (all its *traces* are *traces* of **Q**) and (all its *failures* are *failures* of **Q**).

Whenever **Q** stays alive (engaging with its environment), so does **P** (and in the same way). If **Q** is a specification directly written to express the required patterns of synchronisation, **P** will fulfil them.

Formal

Behaviour: CSP-M *(verifyable)*

Liveness



Recall our informal understanding of (at least some of) the opening traces of **Device** *(slides 20-37)* …

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

Informal understanding

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

What next?

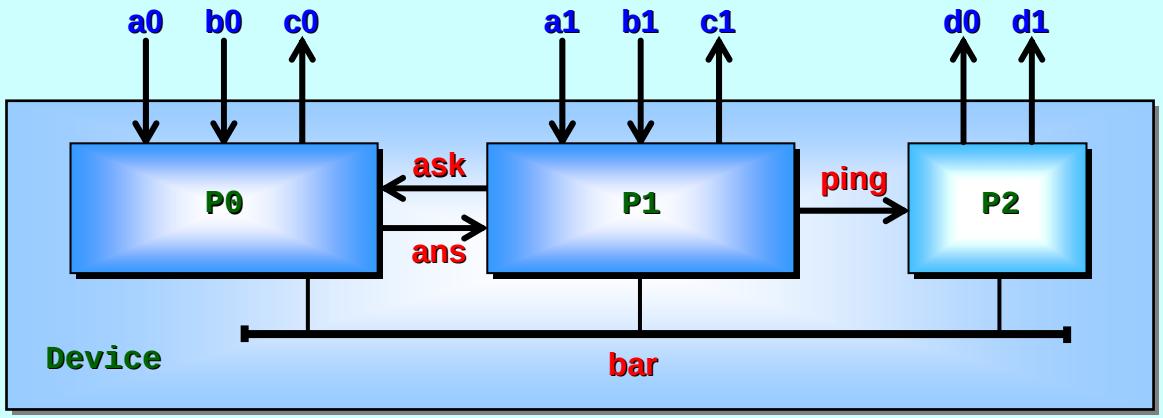
c0 **c1** **d0** *

(* any order)

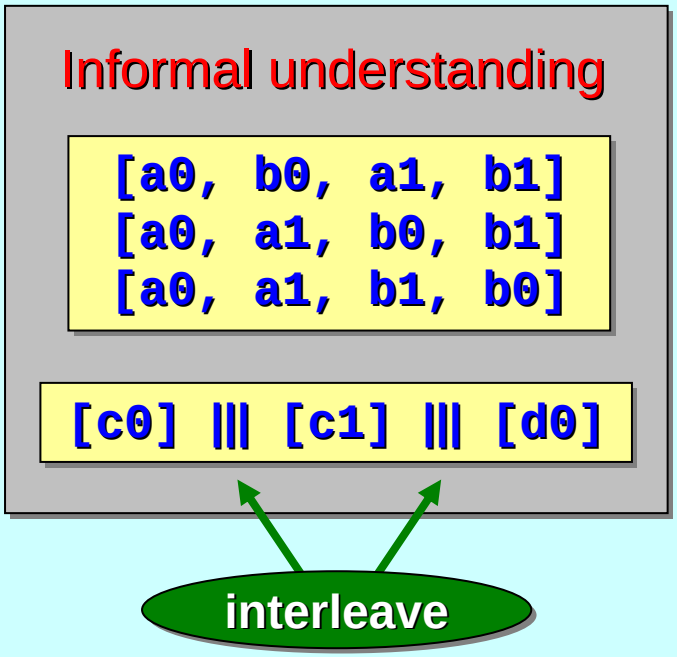
Formal

Behaviour: CSP-M *(verifyable)*

Liveness



Recall our informal understanding of (at least some of) the opening traces of **Device** *(slides 20-37)* …

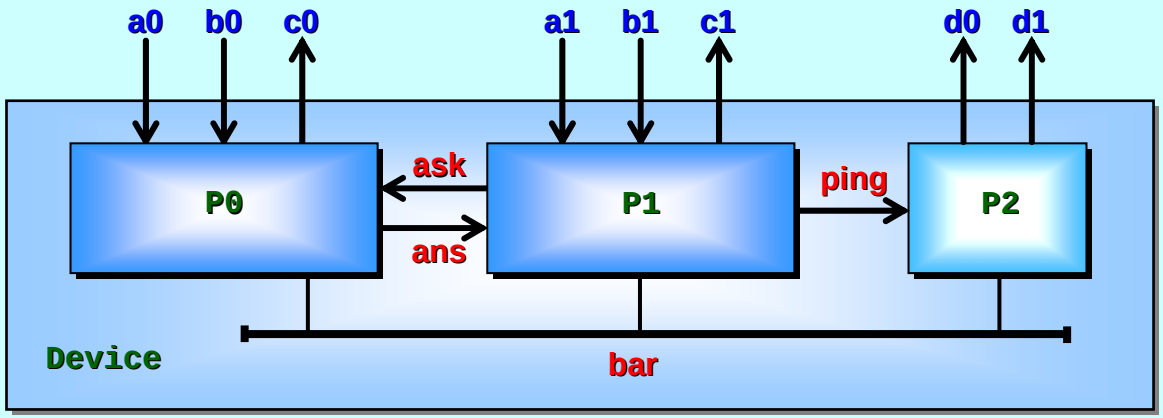


We can formalise the expression of those traces a bit better …

Formal

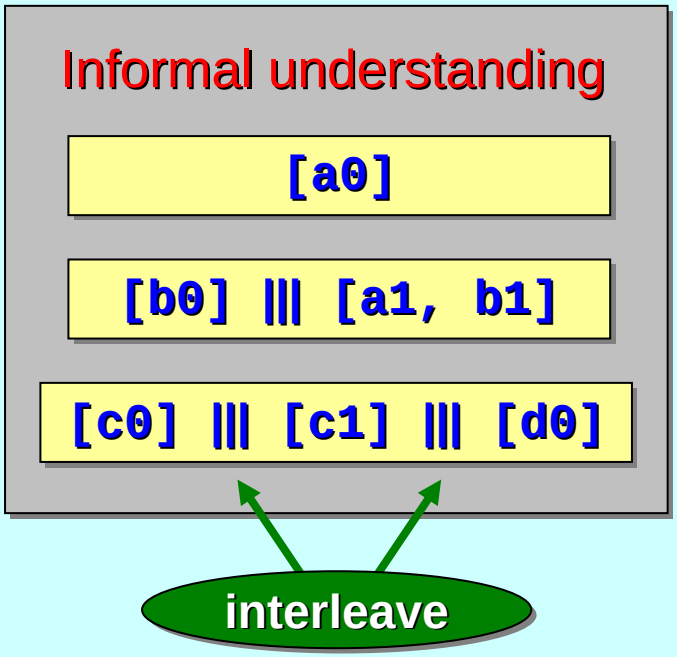
Behaviour: CSP-M *(verifyable)*

Liveness



Recall our informal understanding of (at least some of) the opening traces of **Device** *(slides 20-37)* …

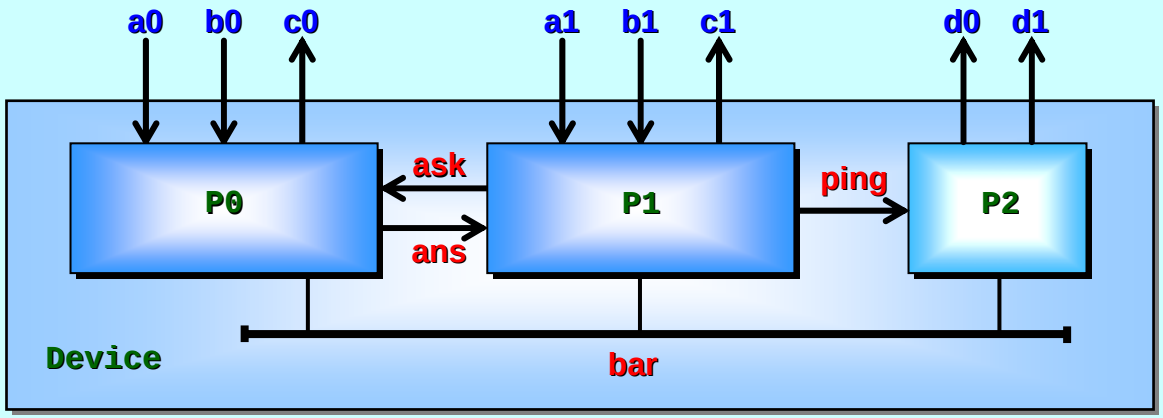
We can formalise the expression of those traces a bit better …



Formal

Behaviour: CSP-M *(verifyable)*

Liveness



Recall our informal understanding of (at least some of) the opening traces of **Device** *(slides 20-37)* …

Informal understanding

```
[a0]
```

```
[b0] ||| [a1, b1]
```

```
[c0] ||| [c1] ||| [d0]
```

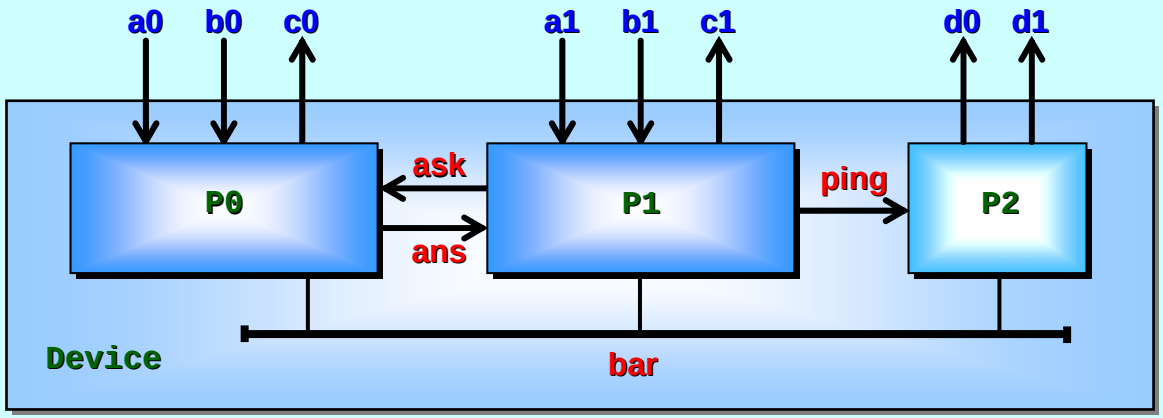
We can formalise the expression of those traces a bit better …

```
[a0]; ([b0] ||| [a1, b1]); ([c0] ||| [c1] ||| [d0])
```

Formal

Behaviour: CSP-M *(verifyable)*

Liveness



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

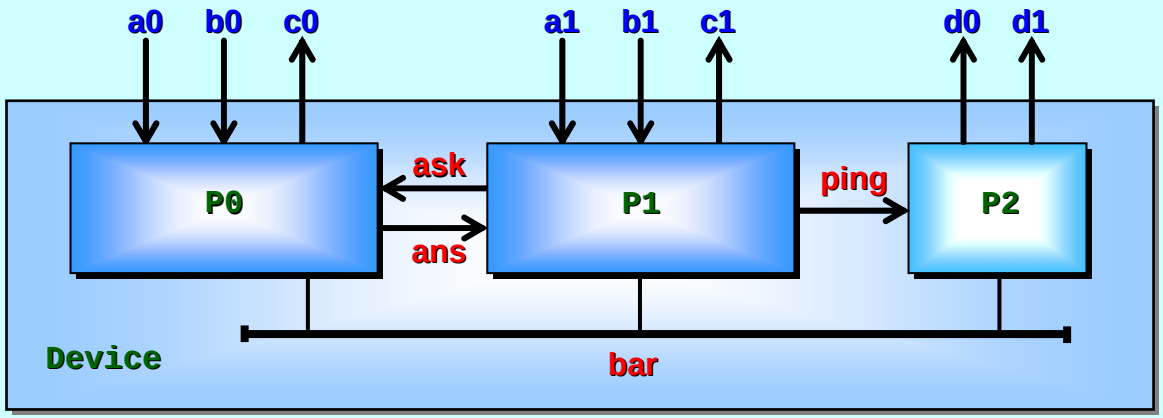
We can formalise the expression of those traces a bit better …

```
[a0]; ([b0] ||| [a1, b1]); ([c0] ||| [c1] ||| [d0]);
[a0]; ([b0] ||| [a1, b1]); ([c0] ||| [c1] ||| [d1])
```

Formal

Behaviour: CSP-M *(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 …

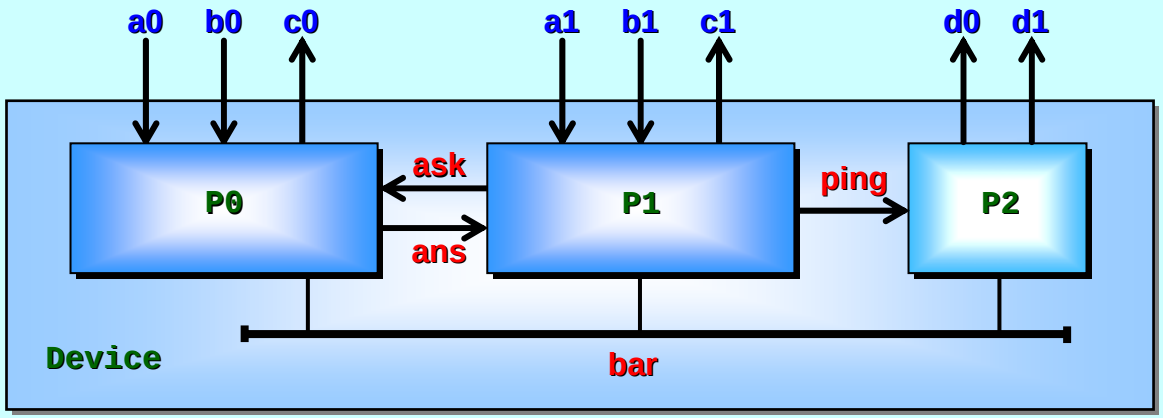
And the rest …

```
( [a0]; ([b0] ||| [a1, b1]); ([c0] ||| [c1] ||| [d0]);
  [a0]; ([b0] ||| [a1, b1]); ([c0] ||| [c1] ||| [d1]) )*
```

Formal

Behaviour: CSP-M (verifyable)

Liveness




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

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

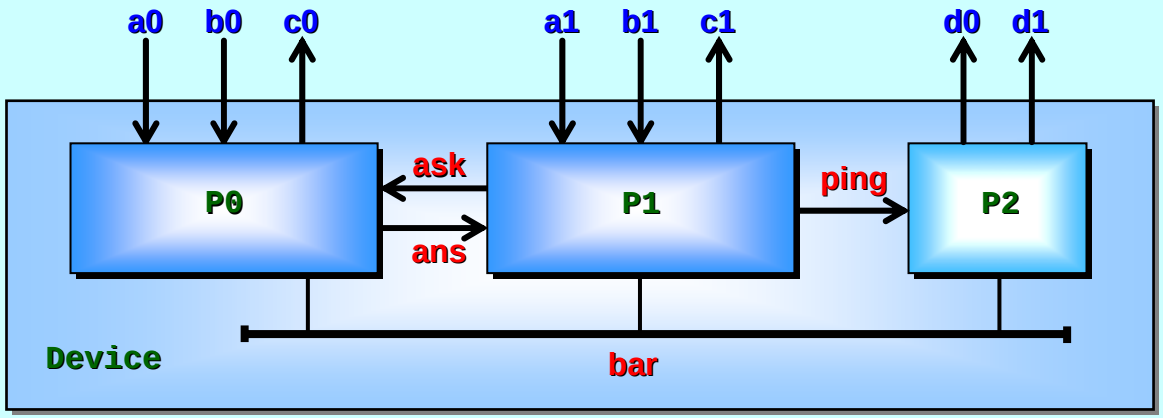
This generation can be automated.

```
( [a0]; ([b0] ||| [a1, b1]); ([c0] ||| [c1] ||| [d0]);
  [a0]; ([b0] ||| [a1, b1]); ([c0] ||| [c1] ||| [d1]) )*
```

Formal

Behaviour: CSP-M (verifyable)

Liveness




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

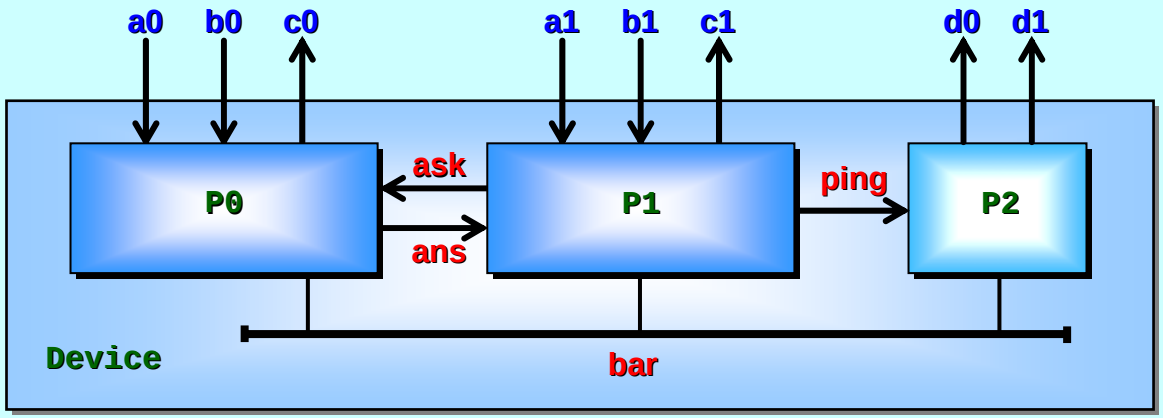
DeviceSpec is an explicit specification of all signal patterns we expect (or need) **Device** to be able to perform.

FDR2 reports **Device** *failure refines* **CheckDevice**. ☺ ☺ ☺
In fact, the reverse is also true – they have exactly the same *traces* and *failures*.

Formal

Behaviour: CSP-M (verifyable)

Liveness



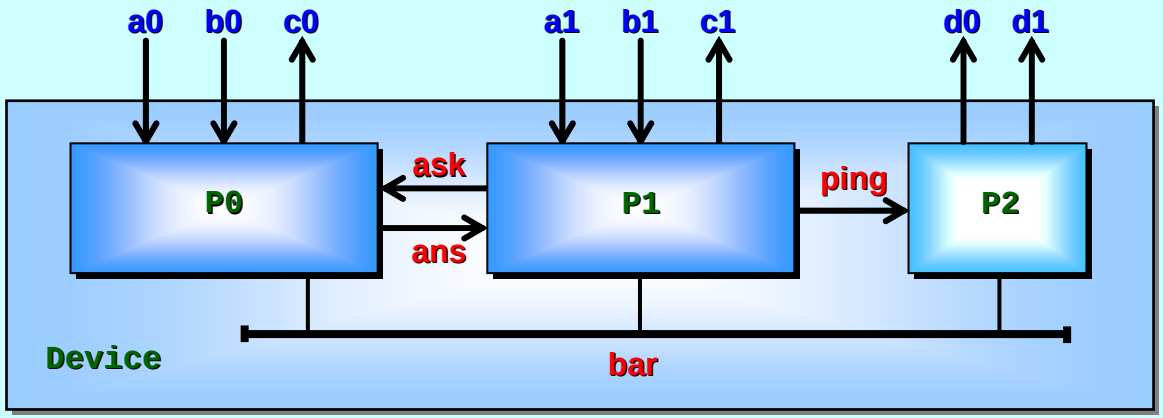

```
DeviceSpec =
  a0 -> (b0 -> SKIP ||| a1 -> b1 -> SKIP);
  (c0 -> SKIP ||| c1 -> SKIP ||| d0 -> SKIP);
  a0 -> (b0 -> SKIP ||| a1 -> b1 -> SKIP);
  (c0 -> SKIP ||| c1 -> SKIP ||| d1 -> SKIP); DeviceSpec
```

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.

Formal

Behaviour: CSP-M (verifyable)

Liveness



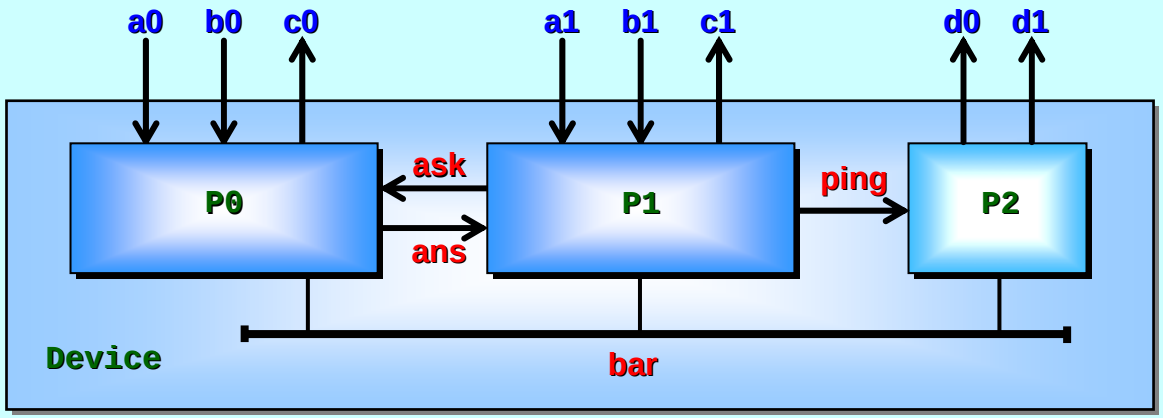

```
DeviceSpec =
  a0 -> (b0 -> SKIP ||| a1 -> b1 -> SKIP);
  (c0 -> SKIP ||| c1 -> SKIP ||| d0 -> SKIP);
  a0 -> (b0 -> SKIP ||| a1 -> b1 -> SKIP);
  (c0 -> SKIP ||| c1 -> SKIP ||| d1 -> SKIP); DeviceSpec
```

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: CSP-M (verifyable)

Liveness



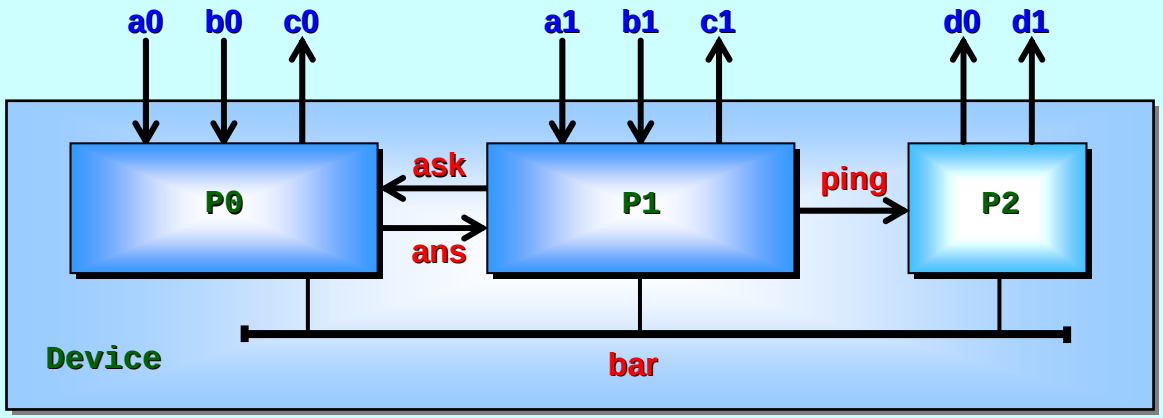

```
DeviceSpec =
  a0 -> (b0 -> SKIP ||| a1 -> b1 -> SKIP);
  (c0 -> SKIP ||| c1 -> SKIP ||| d0 -> SKIP);
  a0 -> (b0 -> SKIP ||| a1 -> b1 -> SKIP);
  (c0 -> SKIP ||| c1 -> SKIP ||| d1 -> SKIP); DeviceSpec
```

One way to ensure this is to add another barrier (bar) at the end of each loop of **P0** and **P1** and half-loop of **P2**. The *failures equivalence* of **Device** and **DeviceSpec** shows that the pattern does indeed repeat cleanly and, so, this overhead is not necessary.

Formal

Behaviour: CSP-M (verifyable)

Liveness




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

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

Reflection

Class experience

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

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

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

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

Reflection

Class experience

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

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

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

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

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

Reflection

Class experience

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

If the (internal) **ping** communications were removed, does **Check** still hold?	No
Do the **a0** and **a1** signals strictly alternate?	Yes
Do the **b0** and **b1** signals strictly alternate?	No
If we added an extra **bar** sync at the end of each cycle in **P0** and **P1** and half-cycle in **P2**, would it make any difference?	No
If the elevator cabin is not at a floor, might the floor doors to the elevator shaft still open?	Another exercise …

Reflection

occam-π / CSP-M

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

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

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

* `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 has been achieved – *by students* – even though they engaged in only simple reasoning themselves.

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

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

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

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

TOPLAS, [April, 2010]

Final Observation

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

- they quickly develop a correct and intuitive understanding of the primitive mechanisms *(e.g. processes, communication, synchronisation, networks)* and higher level patterns *(e.g. client-server, phased barrier, I/O-PAR)* … ?
- they can use those primitives and patterns with the same fluency as they use serial computing primitives, without tripping over dark hazards … ?
- they can develop their own patterns when the standard ones don't apply … ?
- they can use formal methods to verify good behaviour *(e.g. freedom from deadlock and livelock, safety, liveness)*, without training in the underlying mathematics *(process algebra, denotational semantics)* … ?
- they can do this as normal everyday practice, without any sense of fear … ?

Final Observation

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 tri… 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!