Some occam-$\pi$ Basics

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

Co631 (Concurrency)
Some occam-$\pi$ Basics

- Communicating processes ...
- A flavour of occam-$\pi$ ...
- Networks and communication ...
- Types, channels, processes ...
- Primitive processes ...
- Structured processes ...
- ‘Legoland’ …
Communicating Sequential Processes (CSP)

A mathematical theory for specifying and verifying complex patterns of behaviour arising from interactions between concurrent objects.

**CSP** has a formal, and *compositional*, semantics that is in line with our informal intuition about the way things work.

Claim
Why CSP?

- Encapsulates fundamental principles of communication.
- Semantically defined in terms of structured mathematical model.
- Sufficiently expressive to enable reasoning about deadlock and livelock.
- Abstraction and refinement central to underlying theory.
- Robust and commercially supported software engineering tools exist for formal verification.
Why CSP?

- CSP libraries available for Java (JCSP, CTJ).

- Ultra-lightweight kernels* have been developed yielding sub-microsecond overheads for context switching, process startup/shutdown, synchronized channel communication and high-level shared-memory locks.

- Easy to learn and easy to apply …

* not yet available for JVMs (or Core JVMs! )
Why CSP?

- After 5 hours teaching:
  - exercises with 20-30 threads of control
  - regular and irregular interactions
  - appreciating and eliminating race hazards, deadlock, etc.

- CSP is (parallel) architecture neutral:
  - message-passing
  - shared-memory
So, what is CSP?

**CSP** deals with *processes, networks* of processes and various forms of *synchronisation / communication* between processes.

A network of processes is also a process - so **CSP** naturally accommodates layered network structures (*networks of networks*).

We do not need to be mathematically sophisticated to work with **CSP**. **That sophistication is pre-engineered into the model.** We benefit from this simply by using it.
A **process** is a component that encapsulates some data structures and algorithms for manipulating that data.

Both its data and algorithms are **private**. The outside world can neither see that data nor execute those algorithms!  

The algorithms are executed by the process in its own thread (or threads) of control.

So, how does one process interact with another?
Processes

- The simplest form of interaction is *synchronised* message-passing along *channels*.

- The simplest forms of channel are *zero-buffered* and *point-to-point* (i.e. *wires*).

- But, we can have *buffered* channels (*blocking*/*overwriting*).

- And *any-1, 1-any* and *any-any* channels.

- And *structured multi-way synchronisation* (e.g. *barriers*) ...

- And high-level (e.g. *CREW*) *shared-memory locks* ...
Synchronised Communication

\[
(A (c) || B (c)) \setminus \{c\}
\]

A may write on c at any time, but has to wait for a read.

B may read from c at any time, but has to wait for a write.
Synchronised Communication

Only when both $A$ and $B$ are ready can the communication proceed over the channel $c$.

$\left( A (c) \parallel B (c) \right) \setminus \{c\}$
Some occam-$\pi$ Basics

Communicating processes ...

A flavour of occam-$\pi$ ...

Networks and communication ...

Types, channels, processes ...

Primitive processes ...

Structured processes ...

‘Legoland’ …
There must be a consistent (denotational) semantics that matches our intuitive understanding for Communicating Mobile Processes.

There must be as direct a relationship as possible between the formal theory and the implementation technologies to be used.

Without the above link (e.g. using C++/pthreads or Java/monitors), there will be too much uncertainty as to how well the systems we build correspond to the theoretical design.

Theory and practice must be flexible enough to cope with process mobility, location awareness, network growth and decay, disconnect and re-connect and resource sharing.

Computational overheads for managing (millions of) evolving processes must be sufficiently low so as not to be a show-stopper.

Massive concurrency – but no race hazards, deadlock, livelock or process starvation.
**occam-π**

- Process, communication, networks (**PAR**)
- Choice between multiple events (**ALT**)
- Mobile data types
- Mobile channel types
- Mobile process types
- Performance

+ shared channels, channel bundles, alias checking, no race hazards, dynamic memory, recursion, forking, no garbage, protocol inheritance, extended rendezvous, process priorities, …
An occam process may only use a channel parameter one-way (either for input or for output). That direction is specified (?) or (!), along with the structure of the messages carried – in this case, simple INTs. The compiler checks that channel usage within the body of the PROC conforms to its declared direction.
Processes and Channel-Ends

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

PROC integrate.kill (CHAN INT in?, out!, kill?)
  INITIAL INT total IS 0:
  INITIAL BOOL ok IS TRUE:
  ... main loop
  :

serial implementation
Choosing between Multiple Events

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

serial implementation

x
y
z
..
Parallel Process Networks

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

paral|lel   implementation
With an Added Kill Channel

PROC `integrate.kill` (CHAN INT \(\text{in}?\), \(\text{out}!\), \(\text{kill}?\))
CHAN INT \(a\), \(b\), \(c\), \(d\):
PAR
  poison (\(\text{in}?\), \(\text{kill}?\), \(d!\))
  plus (\(d?\), \(c?\), \(a!\))
  delta (\(a?\), \(\text{out}!\), \(b!\))
  prefix (0, \(b?\), \(c!\))

parallel implementation
Some *occam-π* Basics

Communicating processes ...

A flavour of *occam-π* ...

Networks and communication ...

Types, channels, processes ...

Primitive processes ...

Structured processes ...

‘Legoland’ …
... from the top

(components, networks and communication)
PROC P (CHAN INT a!, b?,
CHAN BOOL c?,
CHAN BYTE d!, e!)

...:

PROC Q (CHAN INT a?, b!, c?,
CHAN BOOL d!)

...:

PROC R (CHAN BYTE a?, b!)

...:
PROC S (CHAN INT a?, b!,
  CHAN BOOL c!,
  CHAN INT d!)
  ...
  :

PROC T (CHAN BYTE a?,
  CHAN BOOL b?,
  CHAN BYTE c?)
  ...
  :
CHAN INT f, g, h, m:
CHAN BOOL i, l:
CHAN BYTE j, k, n, o:
CHAN INT f, g, h, m:
CHAN BOOL i, l:
CHAN BYTE j, k, n, o:
PAR
  P (f!, m?, i?, j!, o!)
  Q (f?, g!, h?, i!)
  R (j?, k!)
  R (o?, n!)
  S (g?, h!, m!, l!)
  T (k?, l?, n?)
CHAN INT f, g, h, m:
CHAN BOOL i, l:
CHAN BYTE j, k, n, o:

PAR
P (f!, m?, i?, j!, o!)
Q (f?, g!, h?, i!)
R (j?, k!)
R (o?, n!)
S (g?, h!, m!, l!)
T (k?, l?, n?)
CHAN INT f, g, h, m:
CHAN BOOL i, l:
CHAN BYTE j, k, n, o:
PAR

P (f!, m?, i?, j!, o!)
Q (f?, g!, h?, i!)
R (j?, k!)
R (o?, n!)
S (g?, h!, l!, m!)
T (k?, l?, n?)

Picture & code agree
Good wiring !!!
Synchronised Unbuffered Communication

P0 → c → P1

CHAN INT c:
PAR
P0 (c!)
P1 (c?)
PROC P0 (CHAN INT out!)  
  
  out ! value  
  
  :  

PROC P1 (CHAN INT in?)  
  
  in ? x  
  
  :
Synchronised Unbuffered Communication

- Output \texttt{value} down the channel \texttt{out}
- This operation does not complete until the process at the other end of the channel inputs the information
- Until that happens, the outputting process sleeps (possibly forever!)
Input the next piece of information from channel \texttt{in} into the variable \texttt{x}.

This operation does not complete until the process at the other end of the channel outputs the information.

Until that happens, the inputting process sleeps (possibly forever!)

The inputting process can set “timeouts” on these inputs or choose between alternative inputs. [We will do this later]
Synchronised Unbuffered Communication ("Rendezvous")

- Unified concept of *synchronisation* and *unbuffered communication*.

- *Asynchronous* and *buffered* communication are easy to construct (later).

- Incoming communications are *selectable*.

- **Hardware model:** it is fast to implement.

- **Hardware model:** our intuition enables us to reason about it (see the *Legoland* slides).
Some occam-π Basics

Communicating processes ...
A flavour of occam-π ...
Networks and communication ...
Types, channels, processes ...
Primitive processes ...
Structured processes ...
‘Legoland’ ...

12-Jan-07 Copyright P.H.Welch
occam-$\pi$

... from the bottom
Types

Primitive types

INT, BYTE, BOOL
INT16, INT32, INT64
REAL32, REAL64

Arrays types (indexed from 0)

[100] INT
[32] [32] [8] BYTE
[] REAL64

Record types

(later ...)

The precision of the INT type depends on the word-length of the target processor (e.g. 32 bits for the Intel Pentium)

When the compiler or run-time system can work it out, we don’t have to specify array sizes.
Operators

+,-,*,,\ 
PLUS, MINUS, TIMES

INT\text{x}, INT\text{x} \rightarrow INT\text{x}
BYTE, BYTE \rightarrow BYTE

REAL\text{x}, REAL\text{x} \rightarrow REAL\text{x}

INT\text{x}, INT\text{x} \rightarrow BOOL
BYTE, BYTE \rightarrow BOOL
REAL\text{x}, REAL\text{x} \rightarrow BOOL

*, * \rightarrow BOOL

There is **strong typing** for all expressions ...

types must match

precisions must match
Operators

+\, -\, *, \, /, \, \backslash
PLUS, MINUS, TIMES

\textbf{NB:} this is \textit{modulo}

INTxx, INTxx \rightarrow INTxx
BYTE, BYTE \rightarrow BYTE

REALxx, REALxx \rightarrow REALxx

INTxx, INTxx \rightarrow BOOL
BYTE, BYTE \rightarrow BOOL
REALxx, REALxx \rightarrow BOOL

<\, \leq, \geq, >

=\, <>

*, * \rightarrow BOOL

There is \textbf{strong typing} for all expressions ...
Expressions

No *auto-coercion* happens between types: if \(x\) is a \texttt{REAL32} and \(i\) is an \texttt{INT}, then \(x + i\) is illegal ...

Where necessary, explicit *casting* between types must be programmed: e.g. \(x + (\texttt{REAL32 ROUND} \ i)\) ...

To cast between types, use the *target type name* as a prefix operator.

If *rounding mode* is significant, this must be specified (\texttt{ROUND} or \texttt{TRUNC}) following the *target type name* (as above).

No *precedence* is defined between operators, we must use brackets: e.g. \(a + (b*c)\) ...
Expressions

The operators +, -, *, and / trigger run-time errors if their results overflow.

In Java and C, such errors are ignored.

Therefore, the operators + and * are non-associative and we must use more brackets: e.g. a + (b + c) ...

The INT operators PLUS, MINUS and TIMES wrap-around (i.e. do not trigger run-time errors) if the results overflow.

The occam-π PLUS, MINUS and TIMES are the same as the Java/C +, - and *.

PLUS, MINUS and TIMES are mainly used for calculating timeouts.
Operators

AND, OR
BOOL, BOOL → BOOL

NOT
BOOL → BOOL

AFTER
INTxx, INTxx → BOOL

Boolean logic

time compare

AFTER relates to > in the same way as PLUS relates to +.

They both do arithmetic operations, but the former ignores overflow. If \( 0 < t \leq \text{MOSTPOS INTxx} \), then \((s \ \text{PLUS} \ t)\) is AFTER \( s \), even if wrap-around occurs and \((s \ \text{PLUS} \ t)\) is \( < s \).

There is strong typing for all expressions ...
Operators

not \and \or exclusive-or

\/, \/, \times\x

\sim

<<, >>

\text{bitwise logic}

\text{bitwise shifts}

\text{There is \textit{strong typing} for all expressions ...}
Values (named constants)

```c
VAL INT max IS 50:
VAL INT double.max IS 2*max:

VAL BYTE letter IS 'A':

VAL []BYTE hello IS "Hello*c*n":

VAL []INT mask IS [#01, #02, #04, #08,
                    #10, #20, #40, #80]:
```

All *declarations* end in a colon ...

A declaration cannot be used *before* it is made ...

Character literals have type *BYTE* (their *ASCII* value) ...

String literals have type *[]BYTE* ...
Values (named constants)

VAL INT max IS 50:
VAL INT double.max IS 2*max:

VAL BYTE letter IS 'A':

VAL []BYTE hello IS "Hello*c*n":

VAL []INT mask IS [#01, #02, #04, #08, #10, #20, #40, #80]:

The compiler fills in the sizes of the hello and mask arrays for us. We could have done this ourselves ([7]BYTE and [8]INT respectively).

Declarations are aligned at the same level of indentation ...

Long lines may be broken after commas, etc. ...
Variable Declarations

- INT a, b:
- [max]INT c:
- [double.max]BYTE d:

Two integers
50 integers
100 integers

Timer Declarations

- TIMER tim:

One timer

Channel Declarations

- CHAN BYTE p:
- [max]<<2]CHAN INT q:

A single channel
200 channels
**Process Abstractions**

```
PROC foo (VAL []BYTE s,
VAL BOOL mode,
INT result,
CHAN INT in?, out!,
CHAN BYTE pause?)
```

- **process body**
  - **VAL** (type) <id>:
    - value (data) parameters – local constants within the PROC body ...
  - **<type>** <id>:
    - reference (data) parameters – may be changed within the PROC body (with effect on the invoking process) ...
  - **CHAN** parameters – for communicating with other processes ...

```
foo (s, mode, result)
```
**Process Abstractions**

PROC foo (VAL [ ] BYTE s,
VAL BOOL mode,
INT result,
CHAN INT in?, out!,
CHAN BYTE pause?)

... : 

The process body is indented (two spaces) from its PROC header and closing colon.

We have just used the *three dot notation* as a place holder for the PROC body. Code (including any local declarations) goes here. The *three dots* are not part of occam-π syntax!

Note that the PROC body is indented (two spaces) from its PROC header and closing colon.
Some \textit{occam}-\textpi Basics

Communicating processes ...

A flavour of \textit{occam}-\textpi ...

Networks and communication ...

Types, channels, processes ...

Primitive processes ...

Structured processes ...

‘Legoland’ …
An **occam-π** Process *(syntax)*

Syntactically, an **occam-π** process consists of:

... an *optional* sequence of declarations (e.g. values, variables, timers, channels, procs, channel protocols*, ports*, data types*, channel types*, process types*, barriers*, ... )

... a single executable process

All the declarations – and the executable – are aligned at the same level of indentation.
Primitive Processes

Assignment
\[ a := c[2] + b \]

Input (synchronising)
\[ \text{in } ? a \]

Output (synchronising)
\[ \text{out } ! a + (2*b) \]

There are strong typing rules ...
Primitive Processes

What’s the time?

\[ \text{tim} \ ? \ t \]

Timeout (wait until specified time)

\[ \text{tim} \ ? \ \text{AFTER} \ (t \ \text{PLUS} \ 3000) \]

Null (do nothing)

\[ \text{SKIP} \]

Suspend (non-recoverable)

\[ \text{STOP} \]

+ \text{BARRIER} synchronisation, ...

(later)

where ...

\[
\begin{align*}
\text{TIMER} & \quad \text{tim}: \\
\text{INT} & \quad t:
\end{align*}
\]
A Brief History of Time

What’s the time?

```
tim ? t
```

`occam-π` time values are `INT`s delivered by `TIMER`s. These values increment by one every microsecond (for all current, 10/2006, implementations).

`occam-π` time values `cycle` through all `INT` values – from the most negative (`MOSTNEG INT`), through zero (0), to the most positive (`MOSTPOS INT`) and, then, back to the most negative again. `occam-π` time `starts` at an `arbitrary` `INT` value.
A Brief History of Time

For 32-bit INTs incrementing every microsecond, \texttt{occam-\pi} time values \texttt{cycle} every 72 minutes (roughly).

\begin{itemize}
\item \texttt{2,147,483,647}
\item \texttt{-2,147,483,648}
\item \texttt{0}
\end{itemize}
A Brief History of Time

Note that \( \text{occam-} \pi \) time values increment according to the rules for \( \text{PLUS} \) (wrap-around).
So, \((a \text{ AFTER } b)\) is **TRUE** if and only if the distance from \(b\) to \(a\) going **clockwise** – in the above diagram – is **less than** the distance going **anti-clockwise**.
Above, we have (q AFTER p), (r AFTER q) and (p AFTER r). Think of p, q and r as 2, 4 and 9 on a 12-hour clock face and ignore whether they represent am or pm.
Above, we have \((q \text{ AFTER } p), (r \text{ AFTER } q)\) and \((p \text{ AFTER } r)\). Note that, using normal arithmetic, we have \((q > p)\) and \((r > q)\), but not \((p > r)\).
Therefore, so long as our timeout periods are less than 36 minutes (i.e. half the time cycle) and we calculate absolute timeout values using PLUS, the AFTER operator always gives the expected time comparisons – even if the time wrap-around occurs.
Real-time systems tend to deal in *microseconds* or *milliseconds*, so **36 minutes** is a luxury! If we need to address longer timeouts, some extra (simple) programming effort is required.
A Brief History of Time

2,147,483,647

positive time

t PLUS period

t

-2,147,483,648

negative time

0

SEQ

tim ? t

tim ? AFTER (t PLUS period)

OK, provided period < 36 minutes
A Brief History of Time

2,147,483,647

–2,147,483,648

t PLUS period

positive time

negative time

OK, provided period < 36 minutes

SEQ

tim ? t

tim ? AFTER (t PLUS period)
A Brief History of Time

2,147,483,647

positive time

negative time

-2,147,483,648

OK, provided period < 36 minutes

SEQ

`tim ? t`

`tim ? AFTER (t PLUS period)`
Some *occam-π* Basics

- Communicating processes ...
- A flavour of *occam-π* ...
- Networks and communication ...
- Types, channels, processes ...
- Primitive processes ...
- Structured processes ...
- ‘Legoland’ ...

12-Jan-07  Copyright P.H.Welch
Structured Processes (SEQ and PAR)

SEQ

Do these 4 processes in the sequence written

PAR

Do these 4 processes in parallel
Here is a machine with internal variables $x$ and $\text{sum}$ – assume they are identical numeric types (e.g. $\text{INT}$).

Let’s assume the external channels carry the same type.

Consider the following fragment of code ...
Structured Processes (SEQ example)

SEQ

in ? sum
in ? x
sum := sum + x
out ! sum

Any change in the order of these processes impacts the semantics ...
Structured Processes (PAR example)

Here is another machine with internal variables $x.0$, $x.1$, $a$, $b$ and $c$ – assume they are identical numeric types (e.g. INT).

Let’s assume the external channels carry the same type.

Consider the following fragment of code ...
Structured Processes (**PAR** example)

The order in which these processes run does not matter ...

```
PAR
in.0 ? x.0
in.1 ? x.1
out ! a + b
c := a + (2*b)
```
Structured Processes (PAR rules)

- Change and observe a variable in parallel
- Input from a channel in parallel
- Output to a channel in parallel

Parallel processes may not ...
Structured Processes (PAR rules)

The effect of these rules is that the processes cannot interfere with each other’s state. If they need to interact, they must explicitly communicate.
Structured Processes (PAR rules)

No *data race hazards* are possible. The processes are safe to be scheduled *in any order* (e.g. on a single-core processor) or *in parallel* (e.g. on a multi-core processor).
Structured Processes (IF)

The `<boolean>` conditions are evaluated in sequence. Only the process underneath the first `TRUE` one is executed.

If all the tests are `FALSE`, a run-time error is raised.
Structured Processes (IF example)

The `<boolean>` conditions are evaluated in sequence. Only the process underneath the first `TRUE` one is executed.

```
IF
  x > 0
    screen ! 'p'
  x < 0
    screen ! 'n'
TRUE
  screen ! 'z'
```

If all the tests are `FALSE`, a run-time error is raised.
Structured Processes (WHILE)

WHILE <boolean>

Conventional “while-loop”

If the <boolean> is TRUE, the indented process is executed ... then ...

... the <boolean> is checked again ... if it is still TRUE, the indented process is executed again ... then ...

... etc. until ...

... the <boolean> is checked again ... and turns out to be FALSE ... in which case, this WHILE process terminates.
Structured Processes (WHILE example)

Here is a complete process (a ‘chip’) that doubles the values of the numbers flowing through it:

```plaintext
PROC double (CHAN INT in?, out!)
WHILE TRUE
  INT x:
  SEQ
    in ? x
    out ! 2*x
: runs forever ...☺☺☺
```
Structured Processes (PROC instance)

PROC foo (VAL []BYTE s,
VAL BOOL mode,
INT result,
CHAN INT in?, out!,
CHAN BYTE pause?)

...:

To create an instance, we must plug in correctly typed arguments – for example:

foo ("Goodbye World*c*n", TRUE, solution,
q[i]?, q[i+1]!, my.pause?)

VAL parameters must be passed expressions of the correct type. An expression could be a simple variable or literal.
Structured Processes (PROC instance)

PROC foo (VAL []BYTE s,
        VAL BOOL mode,
        INT result,
        CHAN INT in?, out!,
        CHAN BYTE pause?)

...:

To create an instance, we must plug in correctly typed arguments – for example:

    foo ("Goodbye World*c*n", TRUE, solution,
         q[i]?, q[i+1]!, my.pause?)

Reference parameters must be passed variables of the correct type. Changes to those parameters by the instanced process will be apparent in those variables when (if) the process instance terminates.
Structured Processes (**PROC instance**)

```plaintext
PROC foo (VAL []BYTE s,
       VAL BOOL mode,
       INT result,
       CHAN INT in?, out!,
       CHAN BYTE pause?)

To create an instance, we must plug in correctly typed arguments – for example:

```plaintext
foo ("Goodbye World*c*n", TRUE, solution,
     q[i]?, q[i+1]!, my.pause?)
```

*Channel* parameters must be passed the correct ends (? or !) of correctly typed *channels*. 
Structured Processes (PROC instance)

Process instances used in SEQuence with other processes are sometimes referred to as procedures. For example:

```
INT answer:
SEQ
  out.string ("The answer is ", 0, screen!)
  ... calculate answer
  out.int (answer, 0, screen!)
  out.string ("*c*n", 0, screen!)
```

The processes `out.string` and `out.int` are from the basic utilities library ("course.lib") supporting this course. They output their given string (respectively integer) as ASCII text to their channel parameter and terminate. Their middle parameter is a minimum fieldwidth.
Structured Processes (PROC instance)

Process instances used in PARallel with other processes are sometimes referred to as components (or just processes). For example:

PROC octople (CHAN INT in?, out!)
CHAN INT a, b:
PAR
  double (in?, a!)
  double (a?, b!)
  double (b?, out!)
:

This component scales by 8 the numbers flowing through it ...
Some occam-π Basics

Communicating processes ...

A flavour of occam-π ...

Networks and communication ...

Types, channels, processes ...

Primitive processes ...

Structured processes ...

‘Legoland’ …
‘Legoland’ Catalog

\[\begin{align*}
\text{id} & : \text{in} \rightarrow \text{out} \\
+ & : \text{in}.0 \rightarrow \text{out} \\
& \quad \text{in}.1 \rightarrow \text{out} \\
\text{n} & : \text{in} \rightarrow \text{out} \\
\text{succ} & : \text{in} \rightarrow \text{out} \\
\text{tail} & : \text{in} \rightarrow \text{out} \\
\text{black.hole} & : \text{in} \rightarrow \text{out}
\end{align*}\]
‘Legoland’ Catalog

\[
p \quad \text{in} \quad \text{id} \quad \text{out} \quad p
\]
\[
q \quad r \quad s \quad \ldots
\]

\[
p + 1 \quad \text{in} \quad \text{succ} \quad \text{out} \quad p + 1
\]
\[
q + 1 \quad r + 1 \quad s + 1 \quad \ldots
\]
‘Legoland’ Catalog

\[
\begin{align*}
\text{a} & \quad \text{b} & \quad \text{c} & \quad \text{d} & \quad \text{...} \\
\text{in.0} & \quad \text{+} & \quad \text{out} & \quad \text{a+p} & \quad \text{b+q} & \quad \text{c+r} & \quad \text{d+s} & \quad \text{...} \\
\text{in.1} & \quad \text{p} & \quad \text{q} & \quad \text{r} & \quad \text{s} & \quad \text{...}
\end{align*}
\]
'Legoland' Catalog
‘Legoland’ Catalog

\[
p \quad q \quad r \quad s \quad \ldots
\]

\[
p \quad q \quad r \quad s \quad \ldots
\]

\[
in \quad \rightarrow \quad out
\]

\[
in \quad \rightarrow \quad out
\]
‘Legoland’ Catalog

\[
p \quad q \quad r \quad s \quad \ldots
\]

in

black.hole
This is a catalog of fine-grained processes – think of them as pieces of hardware (e.g. chips).

They process data (INTs) flowing through them.

They are presented not because we suggest working at such fine levels of granularity …

… they are presented in order to build up fluency in working with parallel logic.
Parallel logic should become just as easy to manage as serial logic.

This is not the traditionally held view …

… but that tradition is **wrong**.

Let’s look at some **occam-π** code for these processes …
PROC id (CHAN INT in?, out!)
  WHILE TRUE
  INT x:
  SEQ
    in ? x
    out ! x
  :

PROC succ (CHAN INT in?, out!)
  WHILE TRUE
  INT x:
  SEQ
    in ? x
    out ! x + 1
  :

PROC black.hole (CHAN INT in?)
  WHILE TRUE
  INT x:
    in ? x
  :
PROC plus (CHAN INT in.0?, in.1?, out!)
  WHILE TRUE
  INT x.0, x.1:
  SEQ
    PAR
      in.0 ? x.0
      in.1 ? x.1
    out ! x.0 + x.1
  :

PROC delta (CHAN INT in?, out.0!, out.1!)
  WHILE TRUE
  INT x:
  SEQ
    in ? x
    PAR
      out.0 ! x
      out.1 ! x
  :

Note the parallel input ...

Note the parallel output ...
PROC prefix (VAL INT n, 
    CHAN INT in?, out!) 
SEQ 
    out ! n 
    id (in, out) 
: 

PROC tail (CHAN INT in?, out!) 
SEQ 
    INT any: 
    in ? any 
    id (in, out) 
: 

scope of 'any'
Theorem: \[ n \xrightarrow{\text{tail}} \]

\[ \equiv \]

\[ \text{id} \xrightarrow{\text{id}} \text{id} \]

Theorem: \[ \text{id} \xrightarrow{\text{id}} \text{id} \xrightarrow{\text{id}} \]

\[ \text{id} \xrightarrow{\text{id}} \text{id} \xrightarrow{\text{id}} \]

\[ \text{id} \xrightarrow{\text{id}} \text{id} \xrightarrow{\text{id}} \]

is a blocking **FIFO** buffer of capacity 6
Good News!

The good news is that we can ‘see’ this semantic equivalence with just one glance.

[CLAIM] **CSP** semantics cleanly reflects our intuitive feel for interacting systems.

This quickly builds up confidence …

Wot - no chickens ?!!

Try *Google*ing for this …
Good News!

Let’s build some simple circuits from these catalog components.

Can you see what they do … ?

And how to describe them in occam-π … ?
PROC numbers (CHAN INT out!)
CHAN INT a, b, c:
PAR
delta (a?, out!, b!)
succ (b?, c!)
prefix (0, c?, a!)
:

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

PROC pairs (CHAN INT in?, out!)
CHAN INT a, b, c:
PAR
delta (in?, a!, c!)
tail (a?, b!)
plus (b?, c?, out!)
:
Note: this pushes numbers out so long as the receiver is willing to take it.

Note: this outputs one number for every input it gets.

Note: this needs two inputs before producing one output. Thereafter, it produces one number for every input it gets.
Of course, these components also happen to have simple sequential implementations …

The parallel ones just shown were just to build fluency in CSP concurrency.

CSP (and occam-π) enables parallel and sequential logic to be built with equal ease.

In practice, sometimes parallel and sometimes sequential logic will be most appropriate – just choose the simplest.

Parallel logic is not, by nature, especially difficult.
Sequential Version

PROC numbers (CHAN INT out!)
  INT n:
  SEQ
    n := 0
    WHILE TRUE
      SEQ
        out ! n
        n := n + 1
      :
PROC integrate (CHAN INT in?, out!)
INT total:
SEQ
  total := 0
  WHILE TRUE
    INT x:
    SEQ
      in ? x
      total := total + x
    out ! total

Note: each declaration is as local as possible
‘Legoland’ Catalog

Let’s build some more circuits from the components just constructed (either the sequential or parallel versions).

If we build using the parallel ones, we have *layered* networks – circuits within circuits.

No problem!
pairs

fibonacci

0
1
1
2
3
5
8
13
...

numbers

integrate

pairs

squares

0
1
4
9
16
25
36
...

12-Jan-07 Copyright P.H.Welch
PROC fibonacci (CHAN INT out!)
CHAN INT a, b, c, d:
PAR
  delta (a?, b!, out!)
  pairs (b?, c!)
  prefix (0, d?, a!)
  prefix (1, c?, d!)
:

PROC squares (CHAN INT out!)
CHAN INT a, b:
PAR
  numbers (a!)
  integrate (a?, b!)
  pairs (b?, out!)
:

Note: the two numbers needed by \texttt{PairsInt} to get started are provided by the two \texttt{PrefixInt}s. Thereafter, only one number circulates on the feedback loop. If only one \texttt{PrefixInt} had been in the circuit, deadlock would have happened (with each process waiting trying to input).

\textbf{Note: the traffic on individual channels:}

\begin{align*}
\langle a \rangle &= [0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots] \\
\langle \text{out} \rangle &= [0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots] \\
\langle b \rangle &= [0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots] \\
\langle c \rangle &= [1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots] \\
\langle d \rangle &= [1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots]
\end{align*}
Note: the traffic on individual channels:

\[
\begin{align*}
\langle a \rangle &= [0, 1, 2, 3, 4, 5, 6, 7, 8, \ldots] \\
\langle b \rangle &= [0, 1, 3, 6, 10, 15, 21, 28, 36, \ldots] \\
\langle \text{out} \rangle &= [1, 4, 9, 16, 25, 36, 49, 64, 81, \ldots]
\end{align*}
\]
At this level, we have a network of 5 communicating processes.

PROC demo (CHAN BYTE out!)
    [4]CHAN INT c:
    PAR
        numbers(c[0]!)
        squares(c[1]!)
        fibonacci (c[2]!)
        times (c[3]!)
        lay.out (c?, out!)

In fact, 28 processes are involved: 18 non-terminating ones and 10 low-level transients (repeatedly starting up and shutting down for parallel input and output). **BUT we don’t need to know that to reason at this level … 😊😊😊**
At this level, we have a network of 5 communicating processes.

```plaintext
PROC demo (CHAN BYTE out!)
   [4]CHAN INT c:
   PAR
      numbers(c[0]!)
      squares(c[1]!)
      fibonacci (c[2]!)
      times (c[3]!)
      lay.out (c?, out!)
```

Fortunately, CSP semantics are compositional – which means that we only have to reason at each layer of the network in order to design, understand, code, and maintain it.