Department of Computer Science
University of Bristol

# COMS20001 - Concurrent Computing

www.ole.bris.ac.uk/bbcswebdav/courses/COMS20001_2018/content

Lecture 06

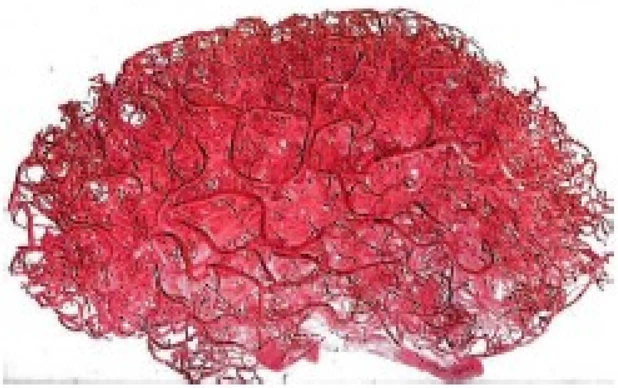
# CSP Abstraction: Events, Processes, Traces and Refinement

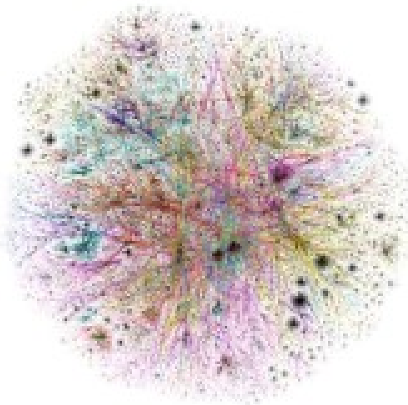
Sion Hannuna | hannuna@cs.bris.ac.uk
Tilo Burghardt | tilo@cs.bris.ac.uk
Dan Page | daniel.page@bristol.ac.uk

24 Slides

# Recap: The Natural World is NOT serial ☺

- …**NATURE is massively concurrent** !
  - natural networks tend to be continuously evolving, yet they are robust, efficient and long-lived
  - Concurrency is one of nature's core design mechanisms – and one of ours!



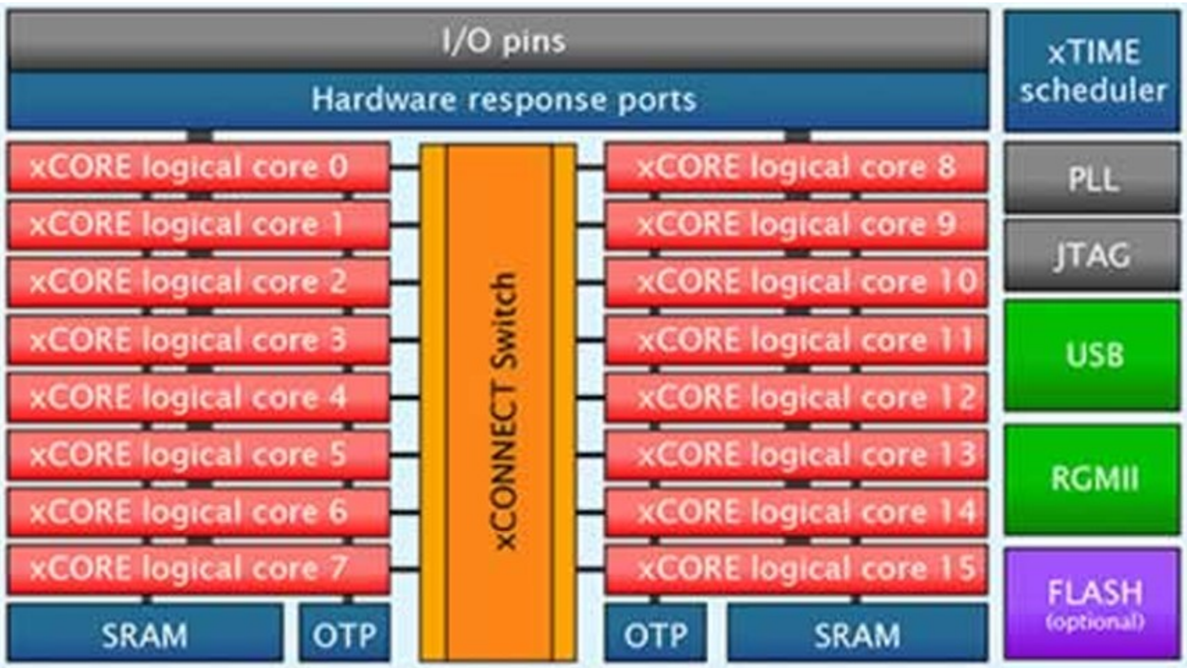


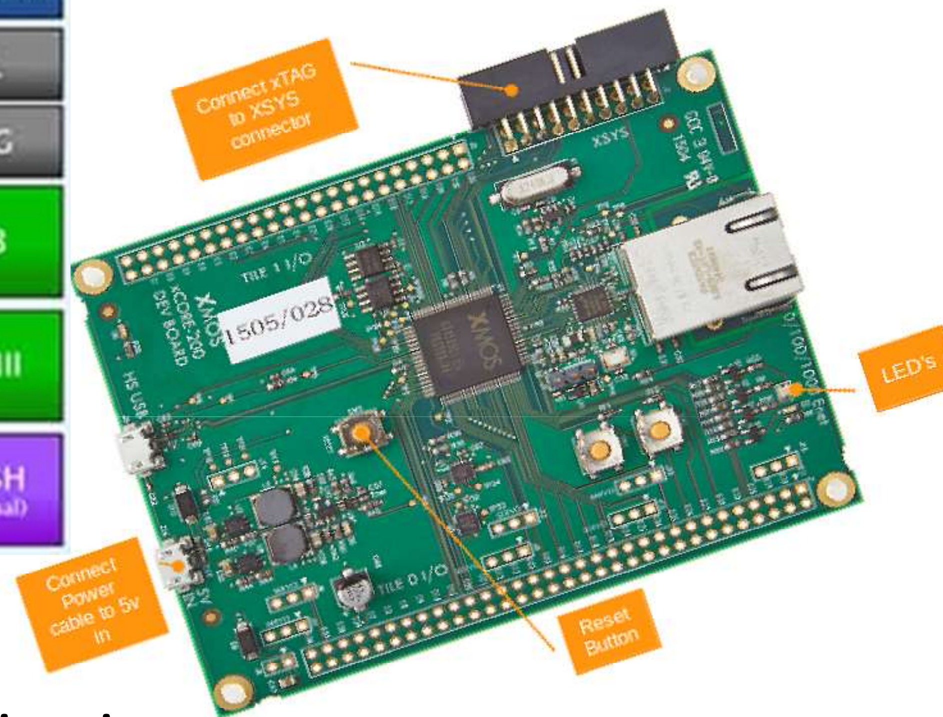
- in many cases **computing models phenomena of the real world**

→ computers are built as part of the physical world and can harvest natural concurrency for their own performance

→ concurrency can often help simplifying the modelling of systems

# Recap: XMOS xCore200 Explorer Kit




- 16 logical cores on 2 xCORE tiles
- 32 channels for cross-core communication
- 512KB internal single-cycle SRAM (max 256KB per tile)
- 6 servo interfaces, 3D accelerometer, Gigabit Ethernet interface, 3-axis gyroscope, USB interface,  xTAG debug adaptor, …

# Need for Formalisation

It is often difficult and complex to design, analyse and implement concurrent systems.

→ there is a need for a systematic approach for describing concurrent systems in a concise way;
→ we want to understand how theory lays the foundation for the programming language XC;

## Aims and Objectives:

→ *understand the basics of a theory of concurrency and interaction*
→ *learn aspects of the process algebra CSP that provides a systematic description of concurrent systems and their properties*
→ *understand and compute important properties of small-scale concurrent systems (e.g. safety, liveness, freedom from spinning)*

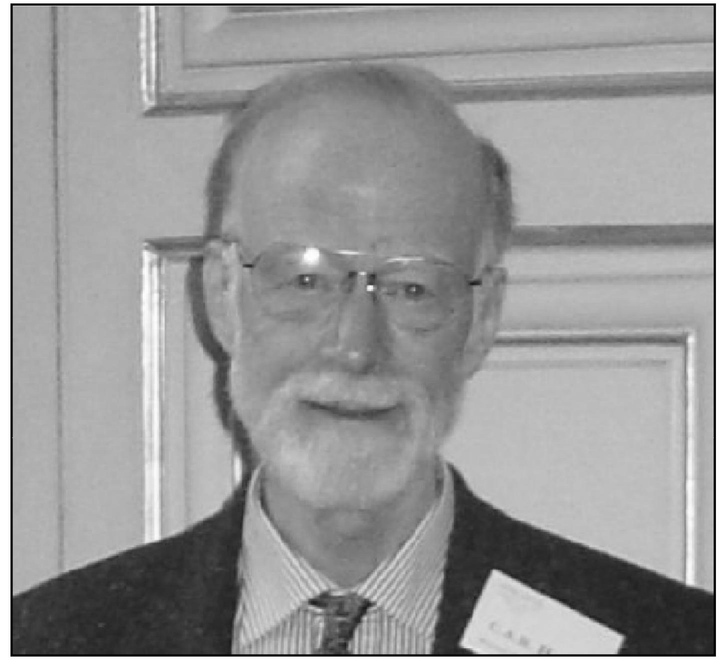
# Communicating Sequential Processes (CSP)

## CSP

... theoretical notation (language) for modelling sets of independent, communicating processes (i.e. concurrent systems)

... pioneered by C.A.R. Hoare, Oxford University, 1980s

... builds on paradigms of 'threads' and 'message passing'

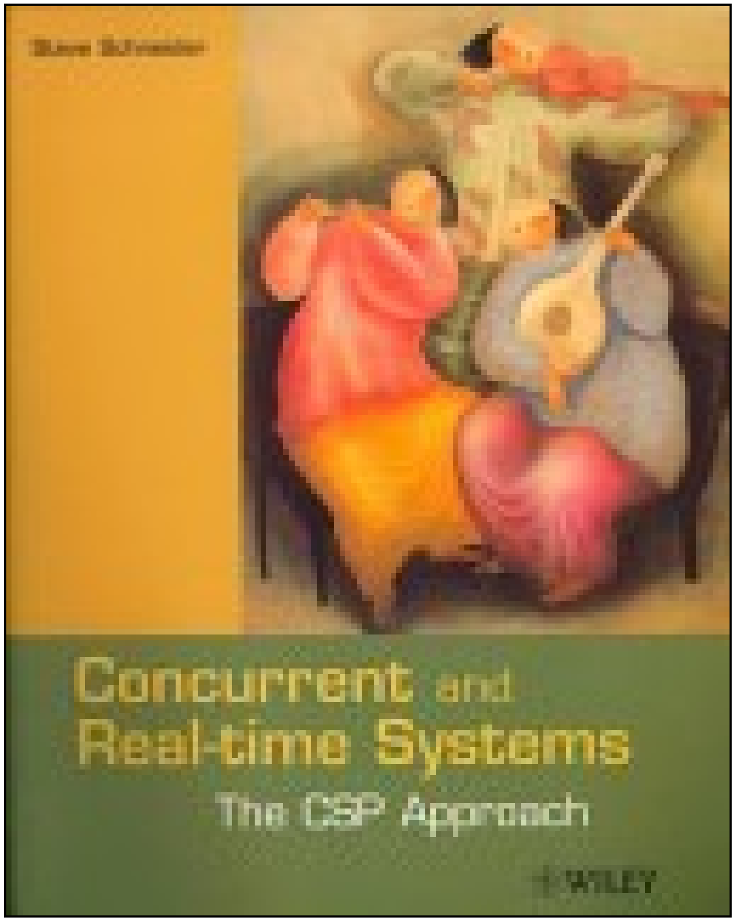


Sir C.A.R. Hoare

→ abstracts the concept of communicating processes

# General Reading: First Chapters of Schneider's CSP book

Steve Schneider
***Concurrent and Real-time Systems: The CSP approach.***
John Wiley & Sons Ltd, 2000,
ISBN: 0-471-62373-3




This book is in the library.
We will only cover the basics of CSP.
Thus, only the first few chapters will be relevant to the course.

# CSP Basics: Abstraction of Processes

→ **Idea:** We reduce a process description (e.g. as defined in XC as a thread) to just the fundamental interactive behaviours , i.e. the events that the process exhibits.

- **CSP Processes**
  … are independent, sequential entities (such as XC threads)
  … engage in events (e.g. operations visible to the environment)
  … may communicate with other processes via common events
  … are completely described by their possible event sequences

- **Primitive Processes**
  … represent fundamental, predefined behaviours such as:
  ***STOP*** (i.e. a process that communicates nothing – deadlock )
  ***SKIP*** (i.e. a process that terminated successfully – work done )

# CSP Basics: Fundamental Events

- **Events**

  … represent visible behaviours of a process (e.g. communications), which are atomic (indivisible) and instantaneous.

  The set of all possible atomic events of a process ***P*** is its alphabet (or interface) written as ***α(P)***

## Example Processes and their events:

| Process | Events |
|---|---|
| *SimpleVendingMachine* | *coin*, *choc* |
| *ComplexVendingMachine* | *in1p*, *in2p*, *small*, *large*, *out1p* |
| $P$ | $a, b, c$ |

# CSP Basics: Operators - Prefixing

- **Prefixing**

  ... describes a process as an event followed by another process:

  **SVM = coin → STOP**
  **SVM = coin → (choc → STOP)**

  ~~SVM = coin → choc (Not valid in strict CSP)~~
  ~~SVM1 = SVM2 → SVM3 (Not valid in strict CSP)~~

- Timing is not described:

  **Lecture = start → (end → STOP)**

  ...for convenience we sometimes leave out (strictly required) brackets (do this only when no strict CSP is asked for)...

  **Lecture = start → end → STOP**

# CSP Basics: Operators - Recursion

- **Recursion**

  … uses prefixing to describe a process as a closed sequence of events:

```
Clock = tock → Clock
Clock = tock → (tock → Clock)
Clock = tock → (tock → (tock → Clock))
...
```

```
SVM = coin → (choc → SVM)
```

- ***Note:*** Recursion may be written as Mutual Recursion:

```
SVM = coin → SVM′
SVM′ = choc → SVM
```

# CSP Basics: Operators – Choice (Guarded Alternative)

- **Choice**

  … describes a process as a set of alternative prefix notations, where the prefix event serves as a guard.

  **`Wait = green → Walk | red → Wait`**

  …read the above as *"green then Walk choice red then Wait"*

  ~~`Wait = Walk | Wait`~~ `(Not valid)`

- Choices can be made amongst any (finite) number of events:

  **`P = k → A | r → B | ... | f → Z`**

- Choices are made either by the process (internal – no external control!) or by the environment (external – control).

# CSP Basics: Operators – Menu Choice

- Guarded alternative and mutual recursion can represent any deterministic DFA using a finite number of equations (process definitions).

- **Menu Choice**

  … provides a notation that allows for a choice amongst an infinite number of alternatives:

  $$\mathbf{P} = \mathbf{a}_1 \rightarrow \mathbf{P}_1 \mid \mathbf{a}_2 \rightarrow \mathbf{P}_2 \mid \ldots \mid \mathbf{a}_n \rightarrow \mathbf{P}_n$$

  written as $\mathbf{P} = \mathbf{x}: \mathbf{A} \rightarrow \mathbf{P(x)}$ given $\mathbf{A} = \{\mathbf{a}_1, \ldots, \mathbf{a}_n\}$

- $\mathbf{P}$ can perform any event $\mathbf{a}_\mathbf{x}$ in alphabet $\mathbf{A}$ and then acts like the process $\mathbf{P(x)}$
- read as ”*x from A then P of x*”.

# CSP Basics: Process Termination & Composition

- **Successful Termination**

  **`SKIP`**: does nothing except perform √ (‘tick') to indicate successful termination

  √: an event outside the normal alphabet, **Σ**.
  It is visible, but not controllable by the environment

- **Sequential Composition**

  … decomposes a process into a process chain **`P;Q`**
  … act like **`P`** until **`P`** terminates successfully (performs a √), then act like **`Q`**

# Interim Summary

- Communicating Sequential Processes enable a systematic specification and analysis approach for concurrent systems.

- Sequential processes can be described with the following concepts:

  | | |
  |---|---|
  | $\alpha(P)$ | The alphabet of process $P$, the set of $P$'s events |
  | $SKIP$ | Successful termination, \`√', \`tick' |
  | $STOP$ | Deadlock |
  | $P;Q$: | Sequential composition |
  | $a \rightarrow P$ | Prefixing, perform event $a$ then act like $P$ |
  | $x: A \rightarrow P(x)$ | Menu choice (guarded alternative) |

# Process Description of a Student

```
α(Student) = {yr1, yr2, yr3, pass, fail, graduate}

PerfectStudent = yr1 → pass → yr2 → pass → yr3 →
                 pass → graduate → SKIP
```

---

```
Student = StudentYr1

StudentYr1 =
yr1 → (pass → StudentYr2 | fail → StudentYr1)

StudentYr2 =
yr2 → (pass → StudentYr3 | fail → StudentYr2)

StudentYr3 =
yr3 → (pass → graduate → SKIP | fail → StudentYr3)
```

# CSP Trace: Notation of an Event Sequence

## A Trace

… is a **finite sequence** (i.e. list) of events, representing a particular behaviour of a process up to a point in time.

- write a trace as comma-separated lists of events, enclosed in angle brackets <>
- empty brace <> (read as ‘empty’ or ‘nil’ ) contains no events

■ Process $TICKET$ with alphabet $A = \{wells, bath, ticket, pound\}$ defined by:

$$
\begin{aligned}
TICKET &= wells \rightarrow pound \rightarrow ticket \rightarrow TICKET \\
&\mid\ bath \rightarrow pound \rightarrow pound \rightarrow ticket \rightarrow TICKET
\end{aligned}
$$

(One) Trace of $TICKET$ is: $\langle bath, pound, pound, ticket\rangle$.

# CSP Traces Set: All Possible Traces of a Process

- A process can have many different behaviours.
- We don't know in advance which trace will be generated by a process.

---

- However, we can note the set of ALL POTENTIAL TRACES of a process to describe its potential behaviour, noted as:

  **`traces(processName)`**

## Examples:

```
traces(STOP) = { <> }
traces(SKIP) = { <>, <√> }
traces(coin → STOP) = { <>, <coin> }
traces(CLOCK) = { <>, <tock>, <tock,tock>, ... }
```

# Example: Traces Sets as Signatures of Behaviour

$$
\begin{aligned}
Driver1 = approach \rightarrow (&left \rightarrow STOP \\
\mid\ &ahead \rightarrow STOP \\
\mid\ &right \rightarrow STOP)
\end{aligned}
$$

$$
\begin{aligned}
Driver2 = approach \rightarrow (&left \rightarrow STOP \\
\mid\ &ahead \rightarrow STOP)
\end{aligned}
$$

$traces(Driver1) =$
$\{\langle\rangle, \langle approach\rangle, \langle approach, left\rangle, \langle approach, right\rangle, \langle approach, ahead\rangle\}$

$traces(Driver2) =$
$\{\langle\rangle, \langle approach\rangle, \langle approach, left\rangle, \langle approach, ahead\rangle\}$

# CSP Trace Refinement

Definition of **refinement** relation $\sqsubseteq_T$ on processes:

$$\boxed{P \sqsubseteq_T Q \text{ if and only if } traces(P) \supseteq traces(Q)}$$

Pronounce $P \sqsubseteq_T Q$ as "P is refined by Q".

Subscript $T$ indicates that refinement is w.r.t. traces.

(– CSP has other forms of refinement too.)

Meaning: $P$ is refined by $Q$, if $Q$ exhibits at most the behaviour exhibited by $P$, *possibly less.*

$a \rightarrow b \rightarrow STOP \sqsubseteq_T a \rightarrow STOP$

For any process $P$, $P \sqsubseteq_T STOP$.

# Motivation: Safety via Trace Refinement

- Trace refinement can be used to specify a behavioural hull (i.e. a safety specification, the maximum allowed set of behaviours), and check that no other behaviour is possibly exhibited by an implementation.
- Example: $Driver1 \sqsubseteq_T Driver2$

> Safety properties (making safety specifications):
>
> Nothing bad can happen.
>
> $Specification \sqsubseteq_T Implementation$
>
> However, does not require anything good to happen either.
>
> $AnySpec \sqsubseteq_T STOP$

# CSP Process Interaction via Alphabetised Parallel

## Process Interaction

…means processes simultaneously perform events,
i.e. events must become joint/synchronized activities.

- Interaction allows to place a process P into an environment of other (concurrently existing) processes (e.g. Q etc).

$P\ {}_A\|_B\ Q$

$A$ and $B$ are alphabets.

$A = \alpha(P)$, $B = \alpha(Q)$ is one possiblity.

$P$ and $Q$ can only perform events from $A$ and $B$ respectively.

All events common to both $A$ and $B$ must be offered by $P$ and $Q$ simultaneously to be able to occur.

Events common to both $A$ and $B$ are performed as one event.

# Interaction Example: Customer and SVM

$$A = \{coin, choc, toffee\}$$

$$SVM = coin \rightarrow (choc \rightarrow SVM \mid toffee \rightarrow SVM)$$
$$Customer = coin \rightarrow choc \rightarrow Customer$$

$$SVM \ {}_{A}\|_{A} \ Customer$$

# Interaction Example: Student and College

Process $STUDENT$ with alphabet:
$S = \{yr1, yr2, yr3, pass, graduate, fail\}$

$$\begin{aligned}
STUDENT &= yr1 \rightarrow (pass \rightarrow YEAR2 \mid fail \rightarrow STUDENT) \\
YEAR2 &= yr2 \rightarrow (pass \rightarrow YEAR3 \mid fail \rightarrow YEAR2) \\
YEAR3 &= yr3 \rightarrow (pass \rightarrow graduate \rightarrow STOP \mid fail \rightarrow YEAR3)
\end{aligned}$$

$$\begin{aligned}
COLLEGE &= fail \rightarrow STOP \mid pass \rightarrow C1 \\
C1 &= fail \rightarrow STOP \mid pass \rightarrow C2 \\
C2 &= fail \rightarrow STOP \mid pass \rightarrow prize \rightarrow STOP
\end{aligned}$$

with $\alpha(COLLEGE) = \{pass, fail, prize\} = C$
and $\alpha(STUDENT) = \{yr1, yr2, yr3, pass, graduate, fail\} = S$

Combine student and college: $STUDENT\ {}_{S}\|_{C}\ COLLEGE$

- Which events do student and college synchronise on?
- What happens if the student fails?
- NOTE: $COLLEGE$ stops after $fail$!

# Looking ahead...

Concurrent System Design