Concurrent Computing

Lecturers: Prof. Majid Mirmehdi (majid@cs.bris.ac.uk)
Dr. Tilo Burghardt (tilo@cs.bris.ac.uk)
Dr. Simon Hollis(simon@cs.bris.ac.uk)
Dr. Daniel Page (page@cs.bris.ac.uk)
Web: http://www.cs.bris.ac.uk/Teaching/Resources/COMS20001

LECTURE 6

CSP ABSTRACTION:

EVENTS, PROCESSES, TRACES , REFINEMENT

[Many thanks to Kerstin Eder, major parts of these lecture slides are taken from or based on materials originally prepared by her.]

Need for Formalisation

→ without a formal approach to modelling their structure, it is often difficult & complex to analyse and implement concurrent systems

→ need for a systematic approach to model concurrent systems

We want to understand how theory lays foundation for programming language XC

- understand basic theory of concurrency and interaction
- learn aspects of process algebra CSP that provides a systematic approach for concurrent systems
- model and analyse tiny-scale concurrent systems.

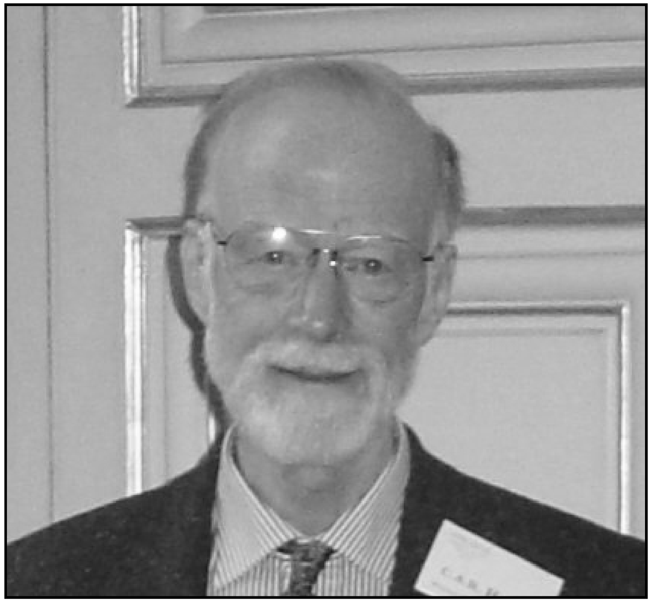
Communicating Sequential Processes (CSP)

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'



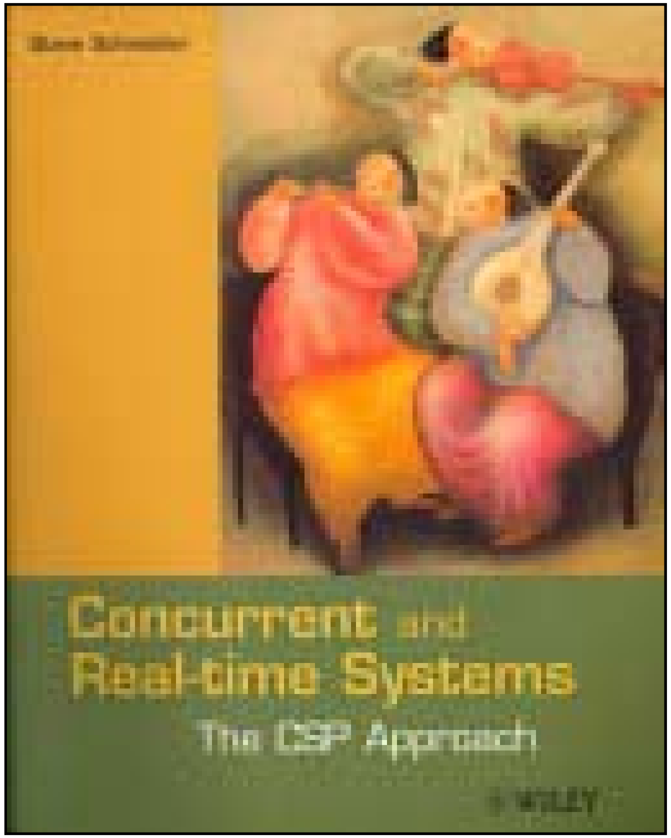
Sir C.A.R. Hoare

→ abstracts the concept of communicating processes

Literature Reference on CSP

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.



CSP – Abstraction of Processes

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

- **Processes**

  … are independent, sequential entities (such as XC threads)
  … engage in events (e.g. visible operations)
  … may communicate with other processes via (common) events
  … completely described by the event sequences it engages in

- **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 – Fundamental Events

- **Events**

  … represent visible behaviours of a process (e.g. communications or interactions), 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 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 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 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 Operators – Menu Choice I

- 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:

  $$\mathbf{P = a_1 \rightarrow P_1 \mid a_2 \rightarrow P_2 \mid \ ... \ \mid a_n \rightarrow P_n}$$

  … is written $\mathbf{P = x\!: A \rightarrow P(x)}$ given $\mathbf{A = \{a_1, a_2, \ldots, a_n\}}$

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

CSP Operators – Menu Choice II

- Menu choice can be used to represent every construct seen so far.
- **Choice:**

  $\mathbf{a_1 \rightarrow P_1 \mid a_2 \rightarrow P_2 \mid \ldots \mid a_n \rightarrow P_n}$

  $\mathbf{x : A \rightarrow P(x)}$

  $\mathbf{A = \{a_1, a_2, \ldots, a_n\}}$

  For each $\mathbf{x}$, $\mathbf{P(a_x) = P_x}$

- **Prefixing:**

  $\mathbf{a_1 \rightarrow P}$

  $\mathbf{x: A \rightarrow P(x)}$

  $\mathbf{A = (a_1)}$

  $\mathbf{P(a_1) = P}$

$\mathbf{STOP = x: \{\} \rightarrow P(x)}$

where $\mathbf{P}$ can remain undefined

CSP 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, design and analysis approach for concurrent systems.

- Sequential processes can be used to describe any deterministic DFA and some infinite state machines using a finite set of equations.

- Sequential processes can be described with the following concepts:

α(P)	The alphabet of process ***P***, the set of ***P***'s events
SKIP	Successful termination, `√', `tick‘
STOP	Deadlock
P;Q:	Sequential composition
a → P	Prefixing, perform event ***a*** then act like process ***P***
x: A → P(x)	Menu choice (generalised guarded alternative)

CSP Example: Undergraduate Career

Process Description of a Student

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

PerfectStudent =
yr1 → pass → yr2 → pass → yr3 → pass → graduate → STOP

Student = StudentYr1

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

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

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

CSP Trace: Notation of an Event Sequence

A Trace

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

- write a trace as comma-separated lists of events, enclosed in angle brackets $\langle\rangle$
- empty brace $\langle\rangle$ (read as '**empty**' or '**nil**') contains no events

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

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

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

Processes and Traces

- 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 as Behavioural Signatures

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

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

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

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

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.*

$a \to b \to STOP \sqsubseteq_T a \to STOP$

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 can be exhibited by an implementation.
- Example: $\mathit{Driver1} \sqsubseteq_T \mathit{Driver2}$

Safety properties (making safety specifications):

Nothing bad can happen.

$\mathit{Specification} \sqsubseteq_T \mathit{Implementation}$

However, does not require anything good to happen either.

$\mathit{AnySpec} \sqsubseteq_T \mathit{STOP}$

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*!