What is CSP?

CSP is a language for processes which interact, both with us and with other processes that may be in parallel with them.

Communicating **S**equential **P**rocesses was invented by Tony Hoare in the mid to late 1970's, in two distinct versions:

- *(CACM, 1978)* A simple imperative programming language with communication between named, sequential components.
- *(Various papers leading up to Hoare's 1985 book)*
  A rather abstract looking notation in which virtually everything had disappeared except for constructs directly related to communication.

This is a course on concurrent systems based on the second of these.

Fundamental concepts

We are in a world of interaction.

A process might be part of a distributed computing network, part of a microprocessor, a vending machine or ATM, a piece of a railway or telephone network.

A CSP process is completely described by the ways in which it can communicate with its external environment.

The most important first step is therefore to choose the *alphabet* of communications: an appropriate set of atomic interactions for the world we are modelling.

The alphabet of all events is conventionally written Σ (Sigma).

In the abstracted world of CSP, these are assumed to be instantaneous: the instant when an interaction is agreed.

Example alphabets

- $\{up, down, iszero\}$ simple counter
- $\{in.x, out.x \mid x \in T\}$ a unit that inputs and outputs values (of type T) on one channel each.
- $\{pay.x, change.x \mid x \in M\}$
  $\cup \{cheddar.w, gouda.w, parmesan.w, \ldots \mid w \in W\}$
  (M is set of money amounts, W is set of weights) a cheese shop.
- $\{getup, gotobed, eat.lunch, eat.breakfast, work, play \ldots\}$ The events in someone's day.

$STOP$

The simplest process is the one that does nothing:

$STOP$

performs no actions at all.

It may seem useless, but we will find that it is useful in programs and specifications, and also provides a simple model of a *deadlocked* system:

Any process, however complex its construction, that can do nothing is equivalent to $STOP$.

Prefixing

If P is a process and $a \in \Sigma$ is any communication

$$a \rightarrow P$$

offers a until the environment accepts it, and then behaves like P.

$$up \rightarrow down \rightarrow STOP$$

$$pay.\pounds 5 \rightarrow gouda.500g \rightarrow$$
$$cheddar.1lb \rightarrow change.\pounds 1.23 \rightarrow STOP$$

Recursion

By returning to their initial states, processes can communicate for ever:

$$P_1 = up \to down \to P_1$$

$$P_2 = up \to down \to up \to down \to P_2$$

$$P_u = up \to P_d$$

$$P_d = down \to P_u$$

If $F(\cdot)$ is an operator from processes to processes, we can define a recursive process by an equation $P = F(P)$ or in-line $\mu\, P.F(P)$. Both create a process which runs the context $F(P)$, such that each occurrence of the name P behaves the same as the whole process.

Mutual recursions, like P_u and P_d, define several processes in terms of each other.

Choice

If $A \subseteq \Sigma$ is any set of events and $P(a)$ is a process for each $a \in A$, then

$$?x : A \rightarrow P(x)$$

offers its environment the choice of A and then behaves like the appropriate $P(a)$. For example:

$$RUN_A = ?x : A \rightarrow RUN_A$$

$$REPEAT = ?x : \Sigma \rightarrow x \rightarrow REPEAT$$

(RUN_A is a process to remember, since it is one of the standard simple processes like $STOP$.)

If $A = \{\}$, then $?x : A \rightarrow P(x) = STOP$ and $RUN_A = STOP$.

Guarded alternative

Where the different $P(x)$'s are not so conveniently similar we can write:

$$(a \to P(a) \mid b \to P(b) \mid \ldots \mid z \to P(z))$$

For example

$$COUNT_0 = up \to COUNT_1$$

$$COUNT_{n+1} = (up \to COUNT_{n+2} \mid down \to COUNT_n)$$

We can clearly create any state machine where each state offers a list of actions with successor states: rather like a menu structure on a computer.

Channels

An event in Σ consists of a channel name plus zero or more data components: up, $cheddar.1lb$, $send.a.b.m$.

In *machine-readable* CSP each channel must be declared with its data type

```
channel up, down
channel cheddar, gouda, parmesan: Weights
channel send, receive: User.User.Message
```

It is always a good idea to follow the type discipline this implies, even when not using the machine readable version.

The data components are sometimes used to transmit data between processes, and sometimes to create arrays of channels.

Input and output

Often a process will want to allow either all

$$c?x \to P(x)$$

or some

$$c?x : A \to P(x)$$

of the communications on one of its channels. These are just forms of $?y : B \to P(y)$ that have the effect of allowing a process to *input* along a channel.

When outputs happen on channels, we often write them $c!x \to P$ rather than $c.x \to P$, though these are nearly synonymous.

(The only differences appear when inputs and outputs are mixed in the same communication: $c?x!y$ and $c?x.y$ have different meanings....see books!)

Example

Provided they are on distinct channels, we allow channel inputs and outputs in the guarded alternative construct:

$$CS(0) = pay?x \rightarrow CS(x)$$

$$\begin{aligned} CS(x) = (&cheddar?w : \{z \in W \mid z \times V_C \leq x\} \rightarrow CS(x - w \times V_C) \\ &\mid gouda?w : \{z \in W \mid z \times V_G \leq x\} \rightarrow CS(x - w \times V_G) \\ &\mid parmesan?w : \{z \in W \mid z \times V_P \leq x\} \rightarrow CS(x - w \times V_P) \\ &\mid pay?y \rightarrow CS(x + y) \\ &\mid change!x \rightarrow CS(0)) \end{aligned}$$

Parameters

Note the use of process parameters to maintain and calculate process state.

Parameters appear in a variety of places depending on the programmer's taste: $P(x, y)$, $P^{x,y}$, $P_y(x)$ etc., but are always treated *declaratively* (i.e., you can't assign to them): programming the data components of CSP systems is essentially *functional* programming.

Two buffer processes

Buffers are processes that act as loss-free, order-preserving handlers of data:

$$COPY \;=\; left?x : T \to right!x \to COPY$$

is a one-place buffer, and the infinite parameterised mutual recursion

$$B^{\infty}_{\langle\rangle} = left?x : T \to B^{\infty}_{\langle x\rangle}$$

$$B^{\infty}_{s\hat{}\langle y\rangle} = (left?x : T \to B^{\infty}_{\langle x\rangle\hat{}s\hat{}\langle y\rangle} \mid right!y \to B^{\infty}_{s})$$

can hold any number of items.

External choice operator

The guarded alternative operator is a bit clumsy because it uses an arbitrary length list of explicitly guarded (by communication) processes.

The *external choice* operator $\Box$ "generalises" it:

$$P \Box Q$$

offers the environment the choice between the initial actions of P and Q, and behaves like the one whose action is picked.

Every guarded alternative can be replaced by $\Box$ (one in place of each "$|$" in the text).

Note that this is a larger change to the syntax structure than it is to the written text.

External *versus* guarded choice

Almost all practical uses of $\Box$ are in places where guarded alternative would have worked (i.e. between disjointly guarded processes), but

- we always use the $\Box$ form anyway, and
- we have to understand what $P \Box Q$ means in the general case.

It is best to think of guarded alternative as being a stepping-stone to understanding $\Box$, rather than having a proper place in the language.

If $A \cap B = \{\}$, it is obvious that

$$(?x : A \to P(x)) \Box (?x : B \to Q(x)) = ?x : A \cup B \to R(x)$$

where $R(x)$ is $P(x)$ or $Q(x)$ depending on whether x is in A or B.

But what happens when $A \cap B \neq \{\}$?

Ambiguity in $P \Box Q$

What happens if the environment selects an initial event common to P and Q?

The implementation doesn't have the information to know which way to jump.

It would be asymmetric and give extra overhead to make it prioritised: e.g., we would have to wait to make sure P could not do an event before allowing Q to act.

The answer, therefore, is that the choice is *nondeterministic*: the process is free to pick either path and the environment has no control over which. For example, in

$$(a \rightarrow b \rightarrow STOP) \Box (a \rightarrow c \rightarrow STOP)$$

the environment cannot control whether it is offered b or c after a.

Nondeterministic choice

Unpredictable behaviour arises naturally in various ways in CSP, and we need to be able to reason about it cleanly.

Therefore there is a *nondeterministic choice* operator. $P \sqcap Q$ can behave like P or like Q as it wishes: the environment has no control. (It must, however, pick one.)

$$(a \rightarrow b \rightarrow STOP) \,\Box\, (a \rightarrow c \rightarrow STOP)$$

$$a \rightarrow (b \rightarrow STOP \sqcap c \rightarrow STOP)$$

If S is a nonempty set of processes, then $\bigsqcap S$ can choose to behave like any member of S:

$$\bigsqcap \{P_1, \ldots, P_n\} = P_1 \sqcap \ldots \sqcap P_n$$

Though these operators are not ones we would often expect to find in programs representing complete implementations, they have many uses.

$Chaos_A$

RUN_A has no nondeterminism: it always offers the whole of A.

Here are two other processes with the same traces but rather different behaviour:

- $Chaos_A = STOP \sqcap ?x : A \to Chaos_A$
  The process that can accept or refuse any element of A.
- $DF_A = \bigsqcap \{a \to DF_A \mid a \in A\}$
  Can always select any member of A, but *must* select one. This is the most nondeterministic *deadlock free* process over A.

Refinement

P refines Q if every behaviour of P is one of Q. It is equivalent to

$$P \sqcap Q = Q$$

and written

$$P \sqsupseteq Q$$

You can never tell, using P, that you are not in fact dealing with Q.

Anywhere that Q works, P is guaranteed to work as well, so P represents an improvement (not necessarily proper) on Q.

Refinement will prove to be exceptionally important throughout this course!

Conditional choice

$\Box$ and $\sqcap$ are both ways of selecting between pairs of alternatives, neither of which is found in "ordinary" programming languages.

There is also a way that just about all languages have:

$$if\ b\ then\ P\ else\ Q$$

has the obvious meaning, and can be written equivalently as

$$P \lhd b \rhd Q$$

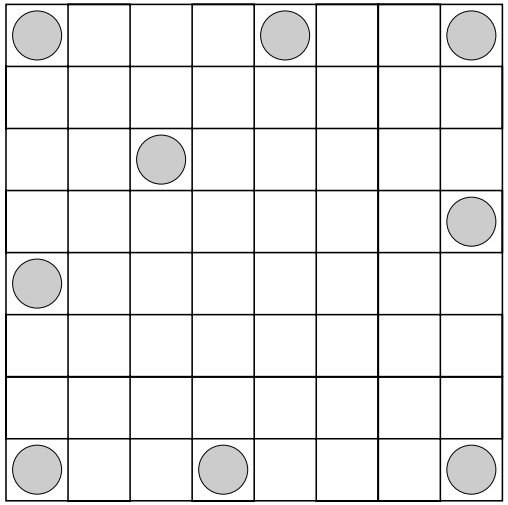
(designed to be an infix operator like $\Box$ and $\sqcap$).

b will depend on parameters and input values. $P \lhd b \rhd STOP$ can be abbreviated $b \& P$

Example

$$\begin{aligned} Counter(n,m) = \quad & n > 0 \& (down \rightarrow Counter(n-1,m)) \\ \Box \; & n < 7 \& (up \quad \rightarrow Counter(n+1,m)) \\ \Box \; & m > 0 \& (left \quad \rightarrow Counter(n,m-1)) \\ \Box \; & m < 7 \& (right \rightarrow Counter(n,m+1)) \end{aligned}$$

This collapses 9 different cases to 1:



Machine readable CSP

A CSP script mixes process and non-process definitions, plus channel and type declarations, comments etc: it is a functional program with processes (and finite sets).

Written in ASCII, so operators look slightly different: `->` `[]` `|~|`

Guarded alternative not supported, and non-channel inputs $?x : A \to P(x)$ can only be written using *indexed* external choice:

```
[] x:A @ x -> P(x)
```

$\sqcap\{P(x) \mid x \in I\}$ is similar: `|~| x:I @ P(x)`

Conditionals achieved by `if..then..else..` and pattern matching, with

```
if b then P else STOP
```

abbreviated `b & P` (see *Counter* definition earlier).

ProBE

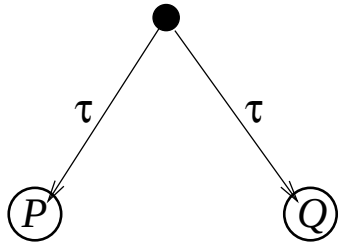
Process **B**ehaviour **E**xplorer.

A tool that allows us to explore the actions of a CSP process, by giving a choice of communications, and calculating the process that results from the one we pick, and so on.....

It is a CSP *animator*, or *simulator*.

Good for helping you understand what you have written, and seeing if it behaves as you expected!

Note that nondeterministic choice `P|~|Q` is implemented via `tau` (τ) actions, which are *internal* and *invisible* to the environment.



Operational semantics

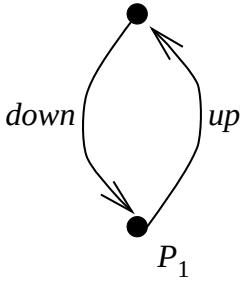
ProBE actually calculates the *operational semantics* of a process P: the graph

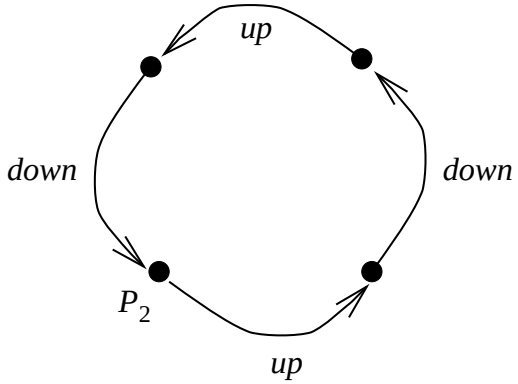
- whose root is P,
- whose nodes are CSP terms that P can become after some actions, and
- where the edges from each node Q are the initial actions (τ or in Σ) that Q can perform.

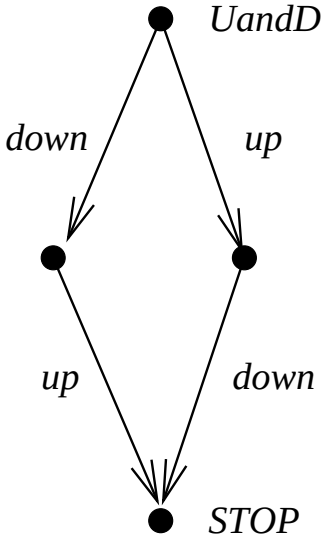
In this way a process is identified with a picture of its behaviour as a simple *state machine*, *automaton* or *labelled transition system*.

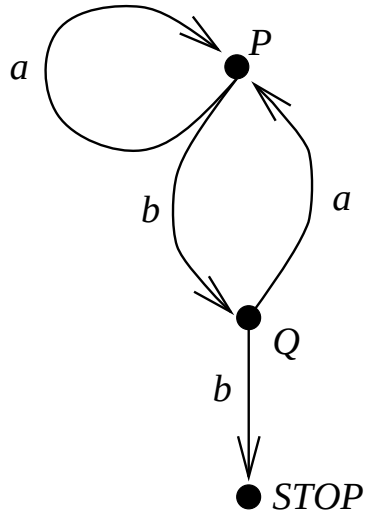
The formalities of this can be studied in Chapter 9, but in this course we will be using these pictures only as an *informal* aid to understanding CSP.

Examples


down
up
P_1


up
down
down
P_2
up


UandD
down
up
up
down
STOP


P
a
b
a
Q
b
STOP