Running side by side

We need to understand how processes run in parallel.

Processes interact by handshaken communication (in which both parties have to agree).

The simplest CSP parallel operator $P \parallel Q$ makes two processes agree on *everything*.

Recall that

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

The event a plays "ping pong" in

$$(a \to REPEAT) \parallel REPEAT$$

so it is equivalent to $\mu\, p.a \to p$.

This illustrates how one process can input from another by offering a choice of actions.

Properties

$P \parallel Q$ is symmetric, associative and distributive, and it has the following step law

$$\begin{aligned} ?x : A \to P \parallel ?x : B \to Q \ &= \\ ?x : A \cap B \to (P \parallel Q) & \end{aligned} \qquad \langle\parallel\text{-step}\rangle$$

Its traces are especially easy to calculate:

$$traces(P \parallel Q) = traces(P) \cap traces(Q)$$

Parallel into sequential

CSP makes no distinction between "parallel" and "sequential" processes, so that we can, for example, write $(P \parallel Q) \,\square\, R$.

Indeed, every parallel process is equivalent to a sequential one.

A combination of two sequential processes under $\parallel$ can be turned into a single sequential process using distributive laws, $\langle\parallel$-step$\rangle$ and UFP.

$$(a \to REPEAT) \parallel REPEAT$$

$$\begin{aligned} &= (a \to REPEAT) \parallel (?x : \Sigma \to x \to REPEAT) \\ &= a \to \underline{(REPEAT \parallel (a \to REPEAT))} \\ &= a \to a \to \underline{(a \to REPEAT \parallel REPEAT)} \end{aligned}$$

Since we have seen the final underlined term before, and it is guarded, we can *derive* a recursion to which it is provably equivalent via UFP:

$$R = a \to a \to R$$

Same can be done to any $P \parallel Q$, often producing a mutual recursion whose equivalent to a selection of compositions $P' \parallel Q'$ for states P' of P and Q' of Q.

Deadlock

A pair of processes put in parallel might not agree on any action:

$$(\mu\, P.a \to b \to a \to p) \parallel (\mu\, Q.a \to b \to Q)$$

deadlocks after the trace $\langle a, b, a\rangle$.

It is equivalent to $a \to b \to a \to STOP$.

$STOP$ is playing its role of an idealised deadlocked process.

Deadlock can appear nondeterministically:

$$P = a \to P \sqcap b \to P$$

(a deadlock free process) implies

$$P \parallel P = Chaos_{\{a,b\}}$$

which can deadlock or not as it wishes.

This example shows that $\parallel$ is not idempotent.

Deadlock will appear in more interesting ways later.

Alphabetised parallel

When two processes are in parallel, we would not usually expect them to share *all* communications.

Some will be shared, and each will probably have communications with the rest of the world.

If X and Y are subsets of Σ, $P\ {}_X\|_Y\ Q$ is the combination where P and Q are assigned *alphabets* X and Y:

- P must perform every communication in X (and no others), and
- Q must perform every communication in Y.

$X \cap Y$ are communications between P and Q, with $X \setminus Y$ and $Y \setminus X$ their independent actions.

Example

$$(a \to b \to b \to STOP)\ {}_{\{a,b\}}\|_{\{b,c\}}\ (b \to c \to b \to STOP)$$

has the behaviour

$$a \to b \to c \to b \to STOP$$

Initially the only possible event is a (as the left hand side blocks b); then both sides agree on b; and so on.

Alphabets

Since the alphabet of a process is usually the set of actions it can perform, why do we need them?

- Because processes sometimes can't perform all the actions we think they can.
  It is vital that we know clearly whether P needs Q's agreement to some action.
- Because sometimes it is useful to give a process a bigger alphabet so it can stop another one performing some actions.

Intrinsic *versus* explicit alphabets

In Hoare's book *all* processes have alphabets (like types) which means that they do not have to be quoted explicitly in the parallel operator.

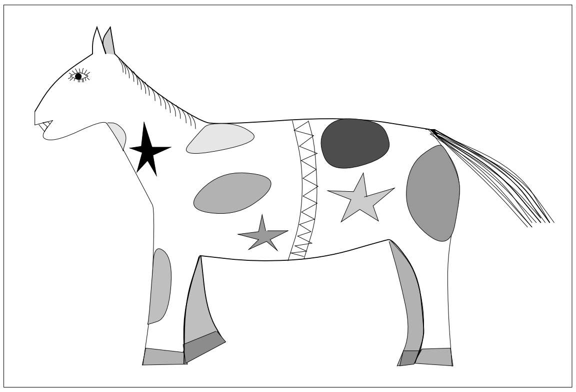
αP is then P's alphabet (which may be strictly larger than the set of events P can perform).

This creates extra work in various other places and saves it in defining $\|$.

Which version you use is a matter of taste: we choose not to have these intrinsic alphabets.

Sometimes we still use the notation αP, but only informally and where it is quite unambiguous.

An example process



A pantomime horse is the parallel composition of two processes: the actors at the front and the back. The structure is

$$Front\ {}_F\|_B\ Back$$

where $\quad F = \{forward, backward, nod, neigh\}$
$\quad\quad\quad\ \ B = \{forward, backward, wag, kick\}$

One pair of actors

With this behaviour:

$$\begin{aligned} Front &= forward \to Front' \\ &\quad \Box\ nod \to Front \\ Back &= backward \to Back' \\ &\quad \Box\ wag \to Back \end{aligned}$$

then the horse will never move whatever $Front'$ and $Back'$ are (since they are never reached). It will simply nod and wag for ever: equivalent to $RUN_{\{nod,wag\}}$.

Step law of ${}_X\|_Y$

Suppose

$$\begin{aligned} P &= ?x : A \to P' \\ Q &= ?x : B \to Q' \end{aligned}$$

$$\begin{aligned} C = \ & (A \cap (X \setminus Y)) && P \text{ by itself} \\ & \cup (B \cap (Y \setminus X)) && Q \text{ by itself} \\ & \cup (A \cap B \cap X \cap Y) && \text{interactions} \end{aligned}$$

Then

$$P\ {}_X\|_Y\ Q = ?x : C \to \left(\begin{array}{l} P' \llparenthesis x \in X \rrparenthesis P \\ {}_X\|_Y \\ Q' \llparenthesis x \in Y \rrparenthesis Q \end{array} \right) \qquad \langle {}_X\|_Y\text{-step}\rangle$$

More laws of ${}_X\|_Y$

$$P\ {}_X\|_Y\ (Q \sqcap R) = (P\ {}_X\|_Y\ Q) \sqcap (P\ {}_X\|_Y\ R) \quad \langle {}_X\|_Y\text{-dist}\rangle$$

$$P\ {}_X\|_Y\ Q = Q\ {}_Y\|_X\ P \quad \langle {}_X\|_Y\text{-sym}\rangle$$

$$(P\ {}_X\|_Y\ Q)\ {}_{X\cup Y}\|_Z\ R = P\ {}_X\|_{Y\cup Z}\ (Q\ {}_Y\|_Z\ R) \quad \langle {}_X\|_Y\text{-assoc}\rangle$$

Note the effects of alphabets on the structure of the symmetry and associative laws.

Iterated parallel

The law $\langle {}_X\|_Y\text{-assoc}\rangle$ begins to show how clumsy it is to build up a network using the binary form of ${}_X\|_Y$ since a lot of compound alphabets have to be formed.

Intrinsic alphabets (αP) provide one solution (and this is the best argument for them).

Another solution is to define a multi-way parallel operator:

$$\begin{aligned}
\|_{i=1}^{2}(P_i, X_i) &= P_1\ {}_{X_1}\|_{X_2}\ P_2 \\
\|_{i=1}^{n+2}(P_i, X_i) &= (\|_{i=1}^{n+1}(P_i, X_i))\ {}_{X_1\cup\ldots\cup X_{n+1}}\|_{X_{n+2}}\ P_{n+2}
\end{aligned}$$

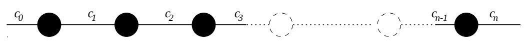
Example

$$\begin{aligned}
COPY'(c,d) &= c?x : T \to d.x \to COPY'(c,d) \\
X_r &= c_r.T \cup c_{r+1}.T\ (= \{\mid c_r, c_{r+1} \mid\})
\end{aligned}$$

(with $c_0, c_1, \ldots c_n$ all distinct) then

$$\|_{r=0}^{n-1}(COPY'(c_r, c_{r+1}), X_r)$$

is a chain of n one-place buffer processes.



It is natural to think of the channels c_0 and c_n as external, because they are only in the alphabet of one process each, and the rest are internal.

Note how we can draw the *communication graph* of a network by looking for intersections between alphabets.

Sequentialising again

Once again, parallel processes can be turned into sequential ones using laws and the UFP rule. For example, in

$$CC_0 = COPY'(a,b)\ {}_{\{\mid a,b\mid\}}\|_{\{\mid b,c\mid\}}\ COPY'(b,c)$$

the initial events of the two processes are $\{\mid a \mid\}$ and $\{\mid b \mid\}$. The full calculation of the overall initials is thus

$$\begin{aligned}
&(\{\mid a \mid\} \cap \{\mid b \mid\} \cap \{\mid a,b \mid\} \cap \{\mid b,c \mid\}) \\
&\cup(\{\mid a \mid\} \cap (\{\mid a,b \mid\} \setminus \{\mid b,c \mid\})) \\
&\cup(\{\mid b \mid\} \cap (\{\mid b,c \mid\} \setminus \{\mid a,b \mid\}))
\end{aligned}$$

$$= \{\} \cup \{\mid a \mid\} \cup \{\} = \{\mid a \mid\}$$

Thus CC_0 equals (by $\langle {}_X\|_Y\text{-step}\rangle$)

$$a?x \to ((b!x \to COPY'(a,b))\ {}_{\{\mid a,b\mid\}}\|_{\{\mid b,c\mid\}}\ COPY'(b,c))$$

Call the parallel combination here $CC_1(x)$.

More transformation

Applying $\langle\|\text{-step}\rangle$ again shows that $CC_1(x)$ equals

$$b!x \to (COPY'(a,b)\ {}_{\{\mid a,b\mid\}}\|_{\{\mid b,c\mid\}}\ c!x \to COPY'(b,c))$$

Call the parallel combination here $CC_2(x)$, and we can prove it equals

$$\begin{aligned}
&a?y \to (b!y \to COPY'(a,b)\ {}_{\{\mid a,b\mid\}}\|_{\{\mid b,c\mid\}}\ c!x \to COPY'(b,c)) \\
&\Box\ c!x \to (COPY'(a,b)\ {}_{\{\mid a,b\mid\}}\|_{\{\mid b,c\mid\}}\ COPY'(b,c))
\end{aligned}$$

which, naming the first parallel combination $CC_3(y,x)$, equals

$$\begin{aligned}
&a?y \to CC_3(y,x) \\
&\Box\ c!x \to CC_1(y)
\end{aligned}$$

Closing it off

Similarly, $CC_3(y, x)$ equals

$$c!x \to ((b!y \to COPY'(a, b)) \;{}_{\{|a,b|\}}\|_{\{|b,c|\}}\; COPY'(b, c))$$
$$= c!x \to CC_1(y)$$

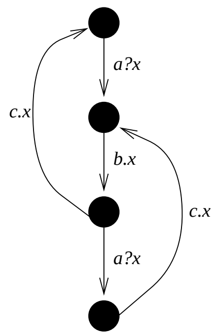
Since all the parallel combinations discovered have already been explored, there is nothing more to do.

We have *derived* a guarded mutual recursion that CC_0, $CC_1(x)$, $CC_2(x)$ and $CC_3(y, x)$ satisfy.

$$\begin{array}{rcl} CC'_0 & = & a?x \to CC'_1(x) \\ CC'_1(x) & = & b!x \to CC'_2(x) \\ CC'_2(x) & = & (c!x \to CC'_0) \,\Box\, (a?y \to CC'_3(y, x)) \\ CC'_3(y, x) & = & c!x \to CC'_1(y) \end{array}$$

UFP proves equality.

Picture of CC_0



Note that this picture doesn't give quite enough details about how data is managed.

Usually this is solved by having separate nodes in the graph for each parameter value.

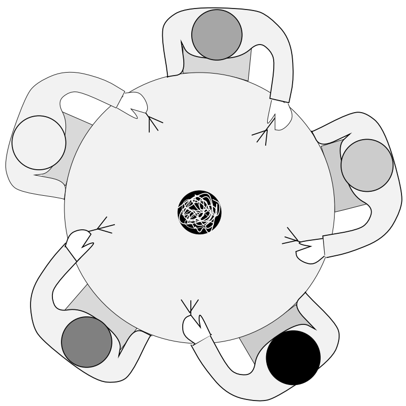
This sort of expansion is tedious for larger examples. Fortunately FDR is much better at it than we are: much of what it does is building enormous expansions like this (current limit is in region 10^8 to 10^9 terms).

Traces

The traces of $P \;{}_X\|_Y\; Q$ are just those which combine a trace of P and a trace of Q so that all communications in $X \cap Y$ are shared.

$$traces(P \;{}_X\|_Y\; Q) =$$
$$\{s \in (X \cup Y)^* \mid s \upharpoonright X \in traces(P)$$
$$\wedge\, s \upharpoonright Y \in traces(Q)\}$$

The dining philosophers



Five philosophers share a dining table at which they have allotted seats.

In order to eat, a philosopher must pick up the forks on either side. These are only put down when he or she has eaten.

Dining philosopher processes

There is one process for each fork...

$$
\begin{aligned}
FORK_i \quad = \quad & (picksup.i.i \to putsdown.i.i \to FORK_i) \\
& \Box\ (picksup.i{\ominus}1.i \to putsdown.i{\ominus}1.i \to FORK_i)
\end{aligned}
$$

... and one for each philosopher

$$
\begin{aligned}
PHIL_i \quad = \quad & thinks.i \to sits.i \\
& \to picksup.i.i \to picksup.i.i{\oplus}1 \\
& \to eats.i \to putsdown.i.i{\oplus}1 \\
& \to putsdown.i.i \to getsup.i \to PHIL_i
\end{aligned}
$$

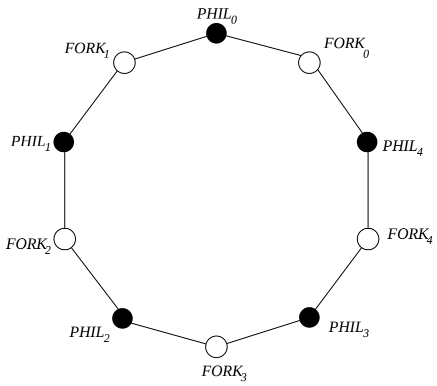
The alphabet of each process is the set of events it can use: AF_i or AP_i.

Forming the college

Setting $(P_1, A_1) = (FORK_0, AF_0)$,
$(P_2, A_2) = (PHIL_0, AP_0)$,
$(P_3, A_3) = (FORK_1, AF_1)$, etc.,

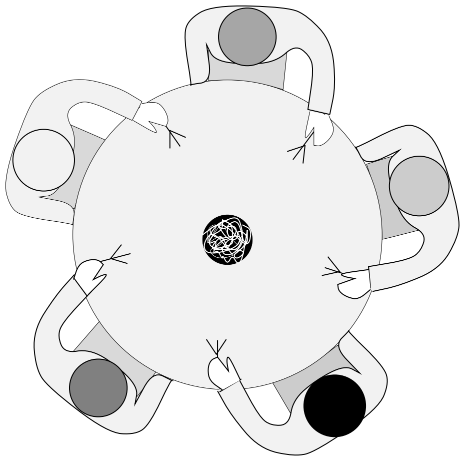
the system is $\Vert_{i=1}^{10} (P_i, A_i)$

and has communication graph



Deadlock

If all philosophers get hungry at once and pick up fork each, they deadlock and starve to death.



Interleaving

$\Vert$ and ${}_X\Vert_Y$ make all partners allowed to communicate a given event a synchronise on it.

The opposite is true of parallel composition by *interleaving*,

$$P \mid\mid\mid Q$$

P and Q run completely independently of each other, and any event of $P \mid\mid\mid Q$ occurs in exactly one of P and Q.

If they can both do event a, then we get nondeterminism similar to that created by $P \Box Q$: $\mid\mid\mid$ allows either to perform a.

If $P = ?x : A \to P'$ and $Q = ?x : B \to Q'$ then

$$
\begin{aligned}
& P \mid\mid\mid Q \ = \\
& \quad ?x : A \cup B \to \quad (P' \mid\mid\mid Q) \sqcap (P \mid\mid\mid Q') \\
& \qquad\qquad\qquad\quad \lessdot x \in A \cap B \gtrdot \\
& \qquad\qquad\qquad\quad (P' \mid\mid\mid Q) \lessdot x \in A \gtrdot (P \mid\mid\mid Q')
\end{aligned}
\qquad \langle \mid\mid\mid\text{-step}\rangle
$$

More laws

$|||$ is symmetric, associative and distributive.

$$P \;|||\; Q \;=\; P \;|||\; Q \qquad \langle|||\text{-sym}\rangle$$

$$(P \;|||\; Q) \;|||\; R \;=\; P \;|||\; (Q \;|||\; R) \qquad \langle|||\text{-assoc}\rangle$$

$$P \;|||\; (Q \sqcap R) \;=\; (P \;|||\; Q) \sqcap (P \;|||\; R) \qquad \langle|||\text{-dist}\rangle$$

Example

An array of printers:

$$\begin{aligned} \mathit{Printer}(n) &= \mathit{input}?x \to \mathit{print}.n!x \to \mathit{Printer}(n) \\ \mathit{Printroom} &= |||_{n=1}^{4} \mathit{Printer}(n) \end{aligned}$$

This is nondeterministic because

- a user has no control over which printer his files appear on, and
- the output events of the printers are named.

Interleaving and recursion

Recursing through $|||$ allows us to spawn off capabilities which remain around while further calls are made.

Processes with the same behaviour as $COUNT_0$ can be built as single rather than mutual recursions:

$$\mathit{Ctr} = \mathit{up} \to (\mathit{Ctr} \;|||\; \mathit{down} \to \mathit{STOP})$$

or, more subtly

$$\begin{aligned} \mathit{Ctr}' &= \mathit{up} \to (\mathit{Ctr}' \;|||\; \mu\, P.\mathit{down} \to \mathit{up} \to P) \\ \mathit{Ctr}'' &= \mathit{up} \to (\mathit{Ctr}'' \;|||\; \mathit{down} \to \mathit{Ctr}'') \end{aligned}$$

Using interleaving

The uses of interleaving on the previous two slides are rather cunning: it is nice when you can achieve effects like these, but you have to be sure of yourself.

The most common everyday use is as a substitute for ${}_X\|_Y$ in cases where X and Y are disjoint.

It saves the work of defining alphabets, for example:

$$\begin{aligned} \mathit{FORKS} &= \mathit{FORK}_0 \;|||\; \mathit{FORK}_1 \;|||\; \ldots \;|||\; \mathit{FORK}_4 \\ \mathit{PHILS} &= \mathit{PHIL}_0 \;|||\; \mathit{PHIL}_1 \;|||\; \ldots \;|||\; \mathit{PHIL}_4 \\ \mathit{AFS} &= \{\!|\, \mathit{picksup}, \mathit{putsdown} \,|\!\} \\ \mathit{SYSTEM} &= \mathit{FORKS} \;{}_{\mathit{AFS}}\|_{\Sigma}\; \mathit{PHILS} \end{aligned}$$

Traces of $P \mid\mid\mid Q$

If s and t are two traces, $s \mid\mid\mid t$ is the *set* of all their interleavings:

$$\begin{array}{rcl} \langle\rangle \mid\mid\mid s & = & \{s\} \\ s \mid\mid\mid \langle\rangle & = & \{s\} \\ \langle a\rangle^\frown s \mid\mid\mid \langle b\rangle^\frown t & = & \{\langle a\rangle^\frown u \mid u \in s \mid\mid\mid \langle b\rangle^\frown t\} \\ & & \cup\{\langle b\rangle^\frown u \mid u \in \langle a\rangle^\frown s \mid\mid\mid t\} \end{array}$$

Given this

$$\begin{array}{rcl} traces(P \mid\mid\mid Q) & = & \bigcup\{s \mid\mid\mid t \mid s \in traces(P) \\ & & \qquad \wedge\, t \in traces(Q)\} \end{array}$$

Generalised parallel

$\|$, ${}_X\|_Y$ and $\mid\mid\mid$ are all special cases of a single operator:

$$P \underset{X}{\|} Q$$

runs P and Q, making them synchronise on events in X and allowing all others freely. Thus (provided P and Q don't communicate outside X and Y)

$$P \;{}_X\|_Y\; Q = P \underset{X \cap Y}{\|} Q$$

and in general

$$\begin{array}{rcl} P \| Q & = & P \underset{\Sigma}{\|} Q \\ P \mid\mid\mid Q & = & P \underset{\{\}}{\|} Q \end{array}$$

This is *generalised* or *interface* parallel.

The use of $\underset{X}{\|}$

Mostly, $P \underset{X}{\|} Q$ is used where it is equivalent to $P \;{}_Y\|_Z\; Q$ where $X = Y \cap Z$.

As a binary operator it seems more natural to use and is more economical (you have to define one small set rather than two big ones).

However the *indexed* form is less useful: it insists that all nodes synchronise on the same set.

In general, N processes have $N(N-1)/2$ binary interfaces and only N alphabets, so the indexed alphabetised form has the advantage here as well.

For these reasons, system definitions in machine-readable CSP most often use the binary operator $\underset{X}{\|}$, and the indexed form of ${}_X\|_Y$.

More interesting uses

$P \underset{X}{\|} Q$ can achieve effects beyond those obtainable using earlier operators.

It can synchronise *some*, but *not all* of the events of P and Q.

$$COUNT_0 \underset{\{up\}}{\|} COUNT_0$$

allows *twice* as many down's as up's.

$$COPY \underset{\{\mid left \mid\}}{\|} COPY$$

outputs each item is receives twice.

Such uses are, however, rare.

See the text for the laws and trace semantics of $\underset{X}{\|}$.

Parallel composition as conjunction

It is obvious that parallel operators are needed to model parallel system implementations.

But they also have other capabilities, one of which is to build trace specifications.

If $R_1(tr)$ and $R_2(tr)$ are two trace specifications with characteristic processes P_1 and P_2, then the characteristic process of $R_1(tr) \wedge R_2(tr)$ is $P_1 \parallel P_2$, because

$$traces(P_1 \underset{\Sigma}{\parallel} P_2) = traces(P_1) \cap traces(P_2)$$

Specifications that are conjunctions of simpler parts may therefore be built up using $\parallel$.

Specifications with restricted alphabets

The specifications on the previous slide had to be of complete traces, even though they might be properties of behaviour in a proper subset of Σ.

If $R(tr)$ is a trace specification on Σ^*, $R'(tr)$ is one on A^*, and P, P' are their characteristic processes ($traces(P') \subseteq A^*$), then that of

$$R(tr) \wedge R'(tr \upharpoonright A) \qquad \text{is}$$

$$P \underset{A}{\parallel} P'$$

Thus if we start off with the most liberal trace specification $P_0 = RUN_\Sigma$, we can build up a compound specification out of parts (say Q_n) that can each have its own natural alphabet (say X_n), by adding on pieces

$$P_n = P_{n-1} \underset{X_n}{\parallel} Q_n$$

Example

Imagine an airlock, with two doors, to regions 1 and 2, that can be opened and closed:

$$\{door\_open.d, door\_close.d \mid d \in \{1,2\}\}$$

valves for equalising the pressure with region 1 or 2. Each valve can be closed or open.

$$\{valve\_open.d, valve\_close.d \mid d \in \{1,2\}\}$$

and we might have an event $tock$ representing the regular passage of time.

The correct behaviour of systems like this is usually best understood in terms of a series of simple specifications on subsets of events.

Properties of the airlock

- Each door, and each valve alternates between opening and closing (4 specifications in all) and is initially closed, e.g.

$$\mu P.door\_open.1 \to door\_close.1 \to P$$

- Only one valve is open at once.

$$\mu P.valve\_open?x \to valve\_close!x \to P$$

- If a valve is open then the opposite door is closed.
- Only one door is open at once.
- No door opens until its valve has been open for 2 time units.

The entire specification can be built up from these parts.

See book exercises for more examples.

Machine-readable parallel operators

- $P \; {}_X\|_Y \; Q$ is written `P [X||Y] Q`
- $P \; ||| \; Q$ is written `P ||| Q`
- $P \underset{X}{\|} Q$ is written `P [|X|] Q`

($P \parallel Q$ can easily be modelled using the others, for example `P[|Events|] Q`.)

Indexed versions are written as follows:

- $\|_{i=1}^{N}(P_i, A_i)$ is `|| i:{1..N} @ [A(i)] P(i)`
- $|||_{i=1}^{N} P_i$ is `||| i:{1..N} @ P(i)`
- $\underset{X}{\|}_{i=1}^{N} P_i$ is `[|X|] i:{1..N} @ P(i)`

FDR and parallel systems

- Often get state explosion in parallel systems: N two-state processes can have 2^N states!
- FDR compiles all the parallel component processes, and uses much faster algorithms to run the combination, but state explosion often puts a ceiling on how large a system you can consider.
  Approximately 600,000,000 states is the current record: several days on a big PC, FDR 2.64, Sept 1999.
  (To run checks of more than approx 10,000,000 states you need to have lots of disc space and configure FDR 2.64 to use it.)
- Finding ways round the state explosion problem is a major topic of research.
- **We can now model and check some interesting non-trivial examples.**