Sudoku in CSP!

		9		2	3			7
		5		6	7	8		
			8					
	1					9		3
				5				
7		2					1	
					5			
			9	4		2		
6		4	2	7		5		

See CSP_M coding in example files.

Deadlock-free routing

It is common to want a service for sending messages between nodes of a network: imagine that the nodes send and receive messages for users (who might be people or other processes). $send.s.(r, m)$ and $receive.r.(s, m)$ represent message m sent from s to r.

Internal packets are of the form (s, r, m).

Most naive attempts to create a routing service lead to deadlock!

Deadlocking ring

$$D_i = send.i?(b, m) \rightarrow D'_i(i, b, m)$$
$$\Box\ ring.i?(a, b, m) \rightarrow D'_i(a, b, m)$$

$$D'_i(a, b, m) = receive.i!(a, m) \rightarrow D_i$$
$$\lhd b = i \rhd$$
$$ring.(i \oplus 1)!(a, b, m) \rightarrow D_i$$

Everybody sending a message at once deadlocks this....and adding more buffering doesn't remove this possibility.

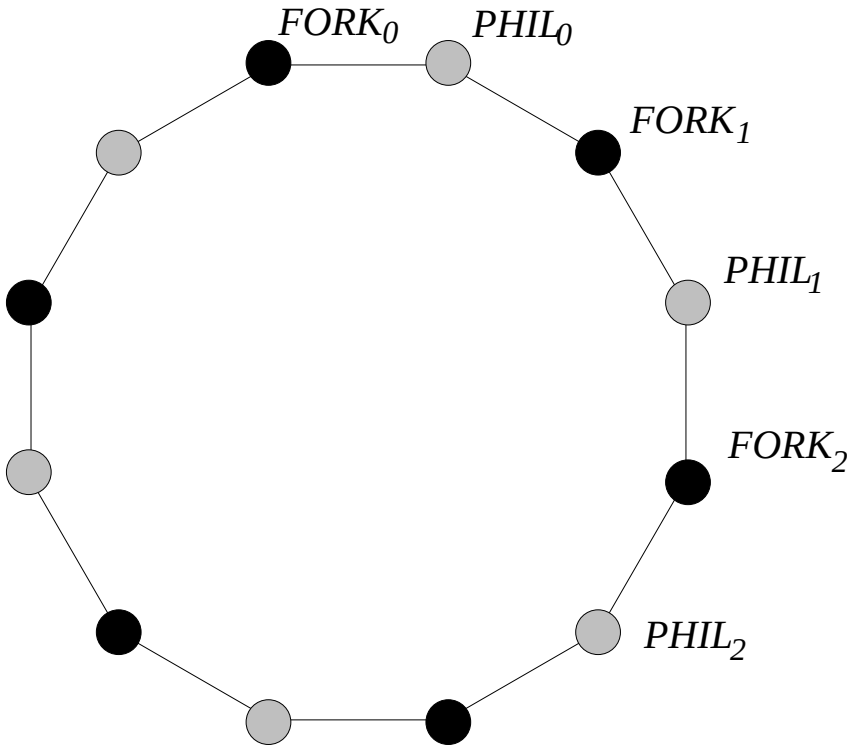
Let's examine deadlock in the abstract.....

... in parallel networks without 3-or-more way synchronisation....

where each component process is deadlock free.

Communication graphs

The communication graph of a network is formed by putting in an edge when alphabets intersect.



Obviously this plays a large part in hunting deadlocks.

Deadlock states and ungranted requests

Since deadlock is a static phenomenon, we can examine the fixed state in which it occurs.

Each component process is in a state which we can add to the communication graph.

Because of our assumptions each process must be able to perform some action. Such actions must, be dependent one other processes (or this would not be a deadlock).

Thus each process has one or more ungranted requests to other processes: it is asking for a communication that the other cannot currently agree to.

Ungranted requests are the building blocks of deadlock!

Strong conflict

Say two processes are in conflict if each has an ungranted request to the other, and strong conflict if one of them has no other options.

Good networks can (almost???) always be (re-)designed to be free of strong conflict.

A strong conflict free network satisfying our assumptions has a proper cycle (length at least 3) of ungranted requests.

Therefore if the network is a tree it is deadlock free....

... and methods of eliminating deadlocks from more complex networks usually concentrate on proving there are no cycles of ungranted requests.

Routing in a tree network

In a tree network there is a unique path between two nodes: so that is the way to send a message! The obvious node definition is

$$NodeE1(n) = \Box\{pass.(n', n)?(a, b, m) \rightarrow NodeF1(n, a, b, m) \mid n' \in tnbrs(n)\}$$
$$\Box\ send.n?(b, m) \rightarrow NodeF1(n, n, b, m)$$

$$NodeF1(n, a, b, m) = receive.n.(a, m) \rightarrow NodeE1(n)$$
$$\lhd n = b \rhd$$
$$pass.(n, tnext(n, b)).(a, b, m) \rightarrow NodeE1(n)$$

The internal channel is $pass$.

But strong conflicts can arise on this channel when a pair want to send a message to each other...this leads to deadlock.

Deadlock-free tree

This can be solved by making swapping a pair of messages a possibility:

$$NodeE2(n) = \Box\{pass.(n', n)?(a, b, m) \to NodeF2(n, a, b, m) \mid n' \in tnbrs(n)\}$$
$$\Box\ send.n?(b, m) \to NodeF2(n, n, b, m)$$

$$NodeF2(n, a, b, m) = receive.n.(a, m) \to NodeE2(n)$$
$$\lhd n = b \rhd$$
$$(pass.(n, tnext(n, b)).(a, b, m) \to NodeE2(n)$$
$$\Box\ swap.(n, tnext(n, b))!(a, b, m) \to$$
$$swap.(tnext(n, b), n)?(a', b', m') \to NodeF2(n, a', b', m')$$
$$\Box\ swap.(tnext(n, b), n)?(a', b', m') \to$$
$$swap.(n, tnext(n, b))!(a, b, m) \to NodeF2(n, a', b', m')$$

Generalisation

We can use the deadlock-free tree as the core of a non-tree routing network by ensuring that all ungranted requests arise within the tree.

See the book.

Routing in a ring

Assume that all messages pass round in one direction.

As we have seen, this can easily create deadlock.

Two approaches:

- Token ring: messages are always held on one of a fixed number of tokens that pass constantly round the ring.
- Non-blocking ring: each node has two places, and only accepts a message on $send$ when both are empty. Thus the ring never becomes full.

See book and example files.

Routing in an ordered network

Suppose that the nodes of our network are in a partial order, and that messages only head downhill $>$.

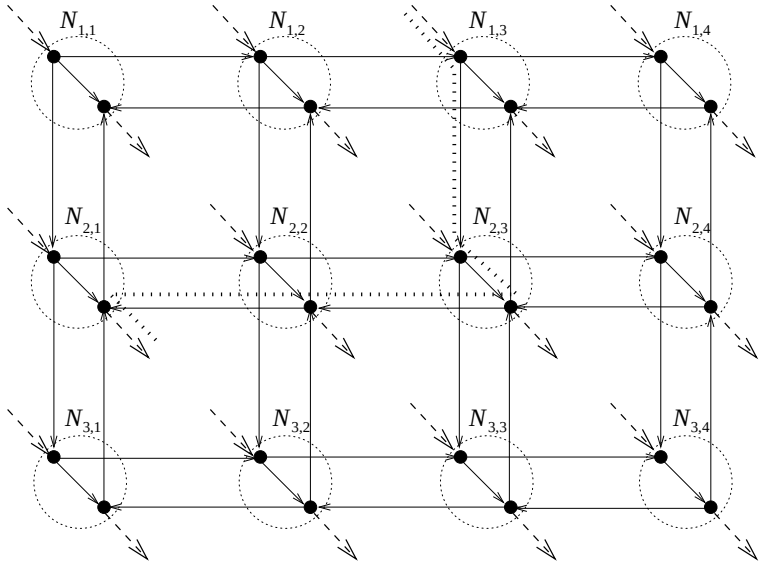
Then we can design nodes so that they input from all greater neighbours if from any.

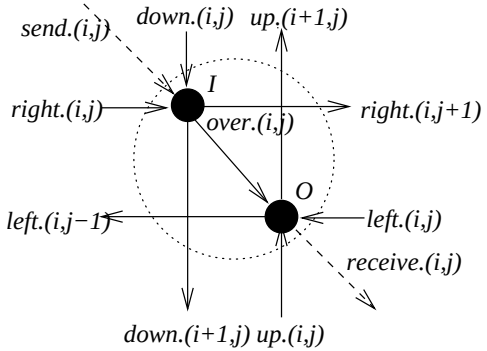
Then there can be no cycle of ungranted requests: a minimal member of a cycle cannot refuse one neighbour while requesting input from another.

But it seems unlikely that this can be much use for general-purpose routing: if you can send a message from A to B you can't do the reverse!

The mad postman (Yantchev)

The solution is to divide each physical node into two logical ones:





The routing algorithm

Divide each node $N_{i,j}$ into the parallel composition of two processes $I_{i,j}$ and $O_{i,j}$. If every message enters the pair via $I_{i,j}$ with destination (k, l) it is then

- routed to $I_{m,n}$ through the Is, where $m = max(i, k)$ and $n = max(j, l)$;
- passed to $O_{m,n}$;
- routed to $O_{k,l}$ through the Os

So making each node parallel (or equivalent to parallel) is the key to deadlock-free routing here.

The input nodes I

$$
\begin{aligned}
I_{i,j} = {} & \mathit{send}.(i,j)?((x,y),m) \to I'_{i,j}((i,j),m) \\
& \Box\ \mathit{down}.(i,j)?(x,y)?p \to I'_{i,j}((x,y),p) \\
& \Box\ \mathit{right}.(i,j)?(x,y)?p \to I'_{i,j}((x,y),p) \\
I'_{i,j}((x,y),p) = {} & \mathit{right}.(i+1,j)!(x,y)!p \to I_{i,j} \\
& \lhd i < x \rhd \\
& (\mathit{down}.(i,j+1)!(x,y)!p \to I_{i,j} \\
& \lhd j < y \rhd \\
& \mathit{over}.(i,j)!(x,y)!p \to I_{i,j})
\end{aligned}
$$

The output nodes O

$$O_{i,j} = over.(i,j)?(x,y)?p \rightarrow O'_{i,j}((x,y),p)$$
$$\Box\ \mathit{up}.(i,j)?(x,y)?p \rightarrow O'_{i,j}((x,y),p)$$
$$\Box\ \mathit{left}.(i,j)?(x,y)?p \rightarrow O'_{i,j}((x,y),p)$$

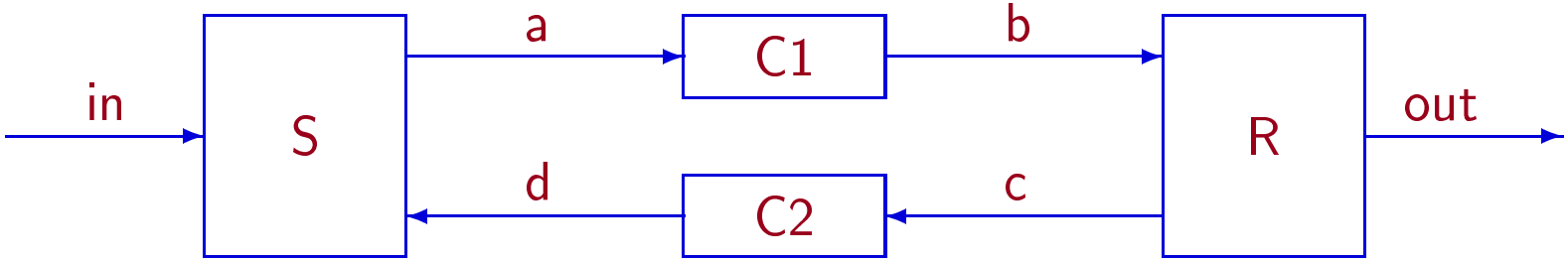
$$O'_{i,j}((x,y),p) = \mathit{left}.(i-1,j)!(x,y)!p \rightarrow O_{i,j}$$
$$\lhd i > x \rhd$$
$$(\mathit{up}.(i,j-1)!x!y!p \rightarrow O_{i,j}$$
$$\lhd j > y \rhd$$
$$receive.(i,j)!p \rightarrow O_{i,j})$$

Communications protocols

Let's assume we have a medium that can lose or duplicate messages, but cannot corrupt or reorder them. It can send structured (i.e., not just bits) messages and acknowledgements,

Note that corruption can be turned into loss via checksums: corrupted messages are discarded.

Alternating Bit Protocol The simplest of a family of protocols for this situation is the *alternating bit protocols*, whose network is



Messages and acknowledgements are tagged alternately with 0 and 1 to distinguish new ones from repetitions and duplications.

This is usually described using real-time features such as time-outs, but it is possible to construct a version whose correctness is independent of timing details.

Error-prone channels

In the following there is no limit on loss or duplication

$$C(in, out) = in?x \rightarrow C'(in, out, x)$$

$$\begin{aligned} C'(in, out, x) = {} & out!x \rightarrow C(in, out) && \text{(correct transmission)} \\ & \sqcap\ out!x \rightarrow C'(in, out, x) && \text{(potential duplication)} \\ & \sqcap\ C(in, out) && \text{(loss of message)} \end{aligned}$$

The channels for the protocol would be $C1 = C(a, b)$ and $C2 = C(c, d)$.

No matter how many errors one can tolerate, it is clear we could not tolerate an infinite unbroken sequence of them: the system would never get anywhere. It is useful to have a version with bounded error behaviour.

$$C_N(in, out, r) = in?x \to C'(in, out, x, r)$$

$$\begin{aligned} C'_N(in, out, x, r) = {} & out!x \to C_N(in, out, N) \\ & \lhd r = 0 \rhd \\ & (out!x \to C_N(in, out, N) \\ & \sqcap\ out!x \to C'_N(in, out, x, r-1) \\ & \sqcap\ C_N(in, out, r-1)) \end{aligned}$$

Controlled errors

Or alternatively one where we can see the errors happening:

$$CE(in, out) = in?x \rightarrow CE'(in, out, x)$$

$$\begin{aligned} CE'(in, out, x) &= out!x \rightarrow CE(in, out) && \text{(correct transmission)} \\ &\Box\ dup \rightarrow out!x \rightarrow CE'(in, out, x) && \text{(potential duplication)} \\ &\Box\ loss \rightarrow C(in, out) && \text{(loss of message)} \end{aligned}$$

Basic sender and receiver

Each has a parameter saying which bit the next *new* acknowledgement/message will have.

$$S(s) = in?x \rightarrow S'(s,x)$$

$$\begin{aligned} S'(s,x) = {} & a.s.x \rightarrow S'(s,x) \\ & \Box\, d.s \rightarrow S(1-s) \\ & \Box\, d.(1-s) \rightarrow S'(s,x) \end{aligned}$$

$$\begin{aligned} R(s) = {} & b.s?x \rightarrow out!x \rightarrow R(1-s) \\ & \Box\, b.(1-s)?x \rightarrow R(s) \\ & \Box\, c!(1-s) \rightarrow R(s) \end{aligned}$$

Note that each is always willing both to send and receive.

(Un)fairness

Even if the channel processes behave perfectly, the system built from the above can fail to make progress: there is nothing to stop (repeated) messages or (repeated) acknowledgements infinitely excluding the other.

This can be avoided by restricting R so that it never does infinitely many of one without the other. Namely, it is made *fair*.

You can either do this by putting a regulator process in parallel with R, or by redesigning it: perhaps making channels b and c alternate.

See also...

Example file on Sliding Window Protocol, a development on ABP in which the sends and acknowledgements of multiple messages are interleaved, so we don't have to wait for one message to be acked before sending another.