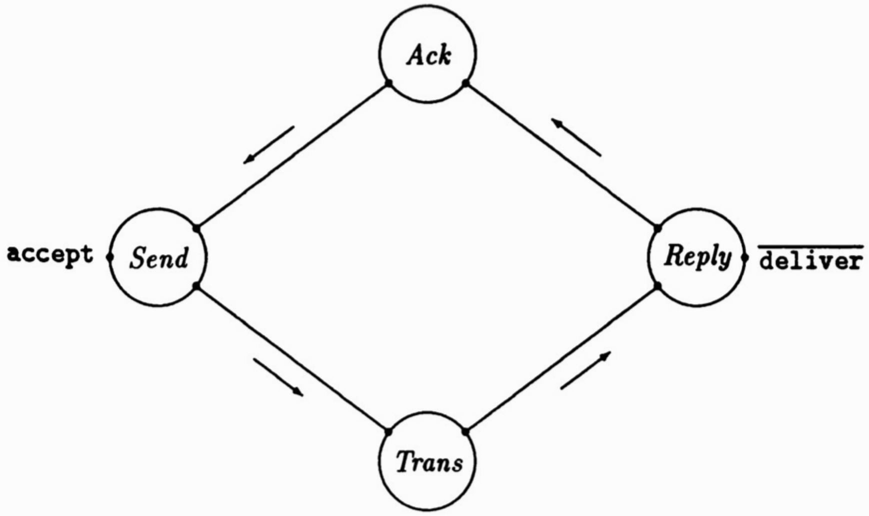
Tutorial 10 11. 12

1. Model the Alternating - Bit protocol using CCS language.

6.3 The alternating-bit protocol

A communications protocol is a discipline for transmission of messages from a source to a destination. Sometimes the protocol is designed to perform routing through large networks; sometimes it is designed to ensure reliable transmission under possibly adverse conditions. Typically, the adverse conditions pertain to the transmission medium, which may lose, duplicate or corrupt messages; for example, the sender cannot assume that a transmission was successful until an acknowledgment is obtained from the receiver. The problem is further compounded if acknowledgments themselves may be lost, duplicated or corrupted.

Here we are interested in a system where the transmission medium consists of communication lines which behave as unbounded buffers (see Section 1.1), except that they are unreliable. The transmission system may be depicted by the following flow graph, in which we indicate the direction of information flow by arrows:



The source (to the left) and the destination (to the right) are not shown. Here *Trans* and *Ack* are the unreliable communication lines, while *Send* and *Reply* are the agents which represent the protocol or discipline.

A famous protocol is called the Alternating-Bit (AB) protocol, and we shall now give a simple narrative description of it. We shall assume that the *Trans* and *Ack* lines may lose or duplicate (but not corrupt) messages, and are 'buffers' with an unbounded message capacity. (Other versions may take them to have bounded capacity.) The name of the protocol refers to the method used; messages are sent tagged with the bits 0 and 1 alternately, and these bits also constitute the acknowledgments.

The sender works as follows. After accepting a message, it sends it with bit b along the *Trans* line and sets a timer. There are then three possibilities:

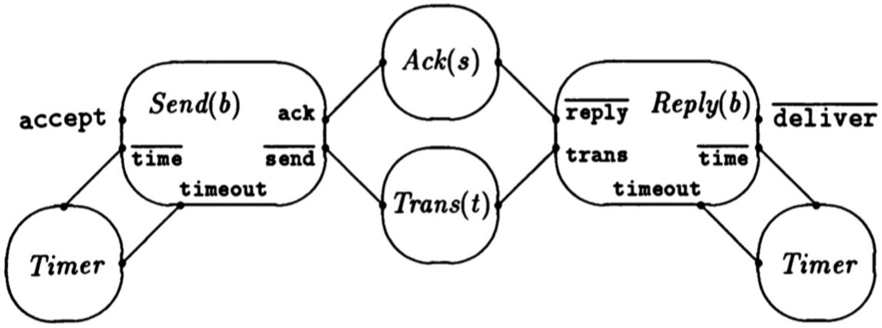
- it may get a 'time-out' from the timer, upon which it sends the message again with b;
- it may get an acknowledgment b from the *Ack* line, upon which it is ready to accept another message (which it will send with bit $\hat{b} = 1 - b$);
- it may get an acknowledgment $\hat{b}$ (resulting from a superfluous retransmission of the previous message) which it ignores.

The replier works in a dual manner. After delivering a message it acknowledges it with bit b along the *Ack* line and sets a timer. There are then three possibilities:

- it may get a 'time-out' from the timer, upon which it acknowledges again with b;
- it may get a new message with bit $\hat{b}$ from the *Trans* line, upon which it is ready to deliver the new message (which it will acknowledge with bit $\hat{b}$);
- it may get a superfluous transmission of the previous message with bit b, which it ignores.

There are subtle variations of the discipline. In one, which was the original design, the replier sets no timer and only returns one acknowledgment per message received. This is simpler, but loses the pleasant duality which we prefer to keep.

It is easy to specify how this protocol, and others, should behave – namely, it should simply accept and deliver messages alternately. This is just to say that it should behave like a perfect buffer of capacity one. This we shall prove. (As an exercise later, we raise the question of whether the buffer capacity need be exactly one.) First, of course, we have to define the system as a composite agent of our calculus. The flow graph is as follows:



Note that each agent (except *Timer*) is parameterised. *Send*(b) and *Reply*(b) are sending and replying with bit b; *Trans*(t) and *Ack*(s) are currently holding bit-sequences $t, s \in \{0,1\}^*$. Since the content of messages is unimportant for the discipline, we are omitting this content (otherwise, of course, each member of t would be a message tagged with a bit).

We now give the definitions, which you should compare with the informal description given earlier. We choose not to use the variable binding convention in this example, since the only values involved are boolean; thus, for example, we write $\overline{\mathtt{send}_b}$ rather than $\overline{\mathtt{send}}(b)$.

$$\begin{aligned}
Send(b) &\stackrel{\text{def}}{=} \overline{\mathtt{send}_b}.\overline{\mathtt{time}}.Sending(b)\\
Sending(b) &\stackrel{\text{def}}{=} \mathtt{timeout}.Send(b) + \mathtt{ack}_b.\mathtt{timeout}.Accept(\hat{b})\\
&\qquad\quad + \mathtt{ack}_{\hat{b}}.Sending(b)\\
Accept(b) &\stackrel{\text{def}}{=} \mathtt{accept}.Send(b)
\end{aligned}$$

$$\begin{aligned}
Reply(b) &\stackrel{\text{def}}{=} \overline{\mathtt{reply}_b}.\overline{\mathtt{time}}.Replying(b)\\
Replying(b) &\stackrel{\text{def}}{=} \mathtt{timeout}.Reply(b) + \mathtt{trans}_{\hat{b}}.\mathtt{timeout}.Deliver(\hat{b})\\
&\qquad\quad + \mathtt{trans}_b.Replying(b)\\
Deliver(b) &\stackrel{\text{def}}{=} \overline{\mathtt{deliver}}.Reply(b)
\end{aligned}$$

$$Timer \stackrel{\text{def}}{=} \mathtt{time}.\overline{\mathtt{timeout}}.Timer$$

If we were to use the variable binding convention, then the second equation would be written

$$\begin{aligned}
Sending(b) &\stackrel{\text{def}}{=} \mathtt{timeout}.Send(b) + \mathtt{ack}(x).Check(x,b)\\
Check(x,b) &\stackrel{\text{def}}{=} \textbf{if } x = b \textbf{ then } \mathtt{timeout}.Accept(\hat{b}) \textbf{ else } Sending(b)
\end{aligned}$$

We now turn to the communication lines. Instead of defining *Trans* and *Ack* by equations – although this is not difficult – it is a little easier to define them by giving all their transitions. In the following, we shall represent concatenation of sequences by juxtaposition; so by sbt, for example, we mean the concatenation of s, b and t (where $s, t \in \{0,1\}^*$, $b \in \{0,1\}$).

$$\begin{array}{ll}
Ack(bs) \xrightarrow{\overline{\mathtt{ack}_b}} Ack(s) & Trans(sb) \xrightarrow{\overline{\mathtt{trans}_b}} Trans(s) \\
Ack(s) \xrightarrow{\mathtt{reply}_b} Ack(sb) & Trans(s) \xrightarrow{\mathtt{send}_b} Trans(bs) \\
Ack(sbt) \xrightarrow{\tau} Ack(st) & Trans(tbs) \xrightarrow{\tau} Trans(ts) \\
Ack(sbt) \xrightarrow{\tau} Ack(sbbt) & Trans(tbs) \xrightarrow{\tau} Trans(tbbs)
\end{array}$$

The last two lines represent loss and duplication, respectively, of any bit in transit.

First we shall compose $Send(b)$ with its *Timer*, and likewise $Reply(b)$, since this leads to a simplification. It is easy to show that, if we define

$$Send'(b) \stackrel{\text{def}}{=} (Send(b) \mid Timer)\backslash\{\mathtt{time}, \mathtt{timeout}\}$$

(with similar priming of the other compositions) then the following equations hold among $Send'(b)$, $Sending'(b)$ and $Accept'(b)$ – in which we now drop the primes:

$$\begin{array}{rcl}
Send(b) & = & \overline{\mathtt{send}_b}.Sending(b) \\
Sending(b) & = & \tau.Send(b) + \mathtt{ack}_b.Accept(\hat{b}) + \mathtt{ack}_{\hat{b}}.Sending(b) \\
Accept(b) & = & \mathtt{accept}.Send(b)
\end{array}$$

and similarly, by composing the *Timer* into $Reply(b)$:

$$\begin{array}{rcl}
Reply(b) & = & \overline{\mathtt{reply}_b}.Replying(b) \\
Replying(b) & = & \tau.Reply(b) + \mathtt{trans}_{\hat{b}}.Deliver(\hat{b}) + \mathtt{trans}_b.Replying(b) \\
Deliver(b) & = & \overline{\mathtt{deliver}}.Reply(b)
\end{array}$$

We are now ready to build the complete system, in which we imagine that a message has just been delivered, a new message is just about to be accepted, and the lines are empty:

$$AB \stackrel{\text{def}}{=} Accept(\hat{b}) \parallel Trans(\varepsilon) \parallel Ack(\varepsilon) \parallel Reply(b)$$

where ε is the empty sequence, and $\parallel$ denotes restricted Composition as usual.

Source: Robin Milner. Communication and Concurrency

2. Model a counter with zero tests using CCS language.

Attempt 1 – infinite alphabet

* define actions: i – increment, d – decrement, t – test for zero

* the easiest solution – one process for each value of the counter: Z represents zero, A_i corresponds to counter $= i$ for $i \in \mathbb{N}_+$

* the processes work as follows:

$$Z \overset{def}{=} i.A_1 + z.Z$$

$$A_1 \overset{def}{=} i.A_2 + d.Z$$

$$A_{k+1} \overset{def}{=} i.A_{k+2} + d.A_k \quad \text{for } k \in \mathbb{N}_+$$

* the graph of transitions

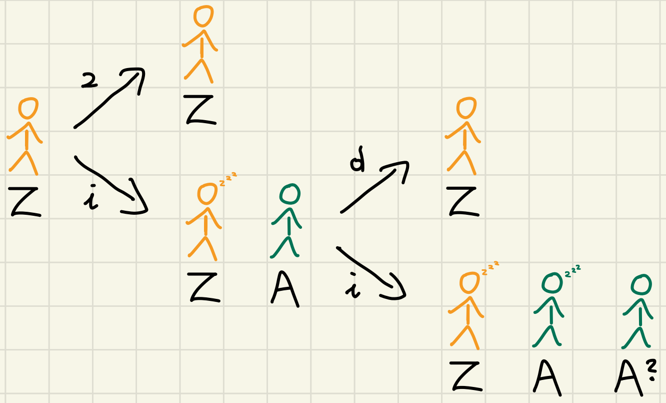
$$\begin{tikzcd} Z \arrow[loop left, "z"] \arrow[r, bend left, "i"] & A_1 \arrow[l, bend left, "d"] \arrow[r, bend left, "i"] & A_2 \arrow[l, bend left, "d"] \arrow[r, bend left, "i"] & A_3 \arrow[l, bend left, "d"] \arrow[r, bend left, "i"] & \dots \arrow[l, bend left, "d"] \end{tikzcd}$$

Problem: the language is infinite

Attempt 2 – finite alphabet

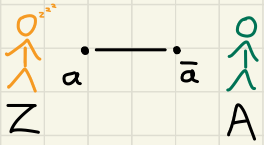
* only a finite number of processes and actions
* we still have to be able to distinct between all posible counter values
* which language operation can allow this ?
  choice / parallel computation / restriction / renaming

Idea :



* incrementation – current active process "falls asleep" (becomes inactive) ; a new active process is generated
* decrementation – current active process dies and the last one to fall asleep wakes up
* how to model the waking up operation and how many different processes do we need ?

* for process A to be able to "wake up" Z the two processes needs to communicate



let us use port a for this purpose

* now, how such a communication may look like? we can define it as follows

$$Z = z.Z + i.(A \mid a.Z)$$

after incrementation two processes are generated

A is the active one

Z "waits behind" port a (to be called by A)

$$A = d.\bar{a}.0 + i.\_$$

to "wake up" Z after decrementation, A calls $\bar{a}$

we need to allow the incrementation of A, but how to do it? (*)

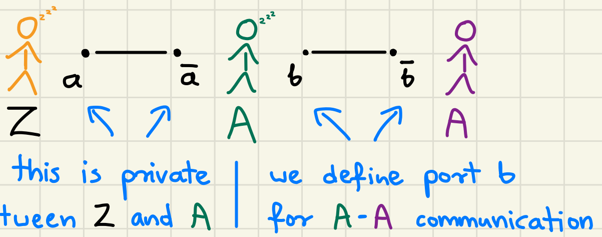
* to ensure that only A can wake up Z, we need to make the communication on port a private for the two processes

$$Z = z.Z + i.(A \mid a.Z) \setminus a$$

port a is now hidden from the outside world

* at this point, we need to decide what to do after the incrementation of A

* notice that if we create another copy of A, we will need to define a new port for the communication with the first A copy


a
$\bar{a}$
b
$\bar{b}$
Z
A
A
this is private between Z and A
we define port b for A-A communication

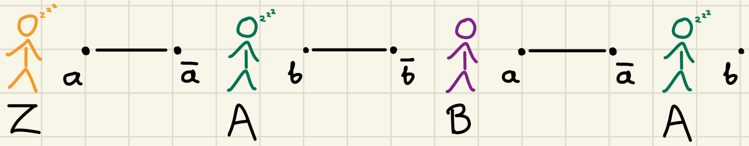
thus, the definition of A will look like this

$$A = d.\bar{a}.0 + i.(A \mid b.A)$$

which won't work, because right now starting from the second copy of A, these processes wait to be woken up on port b, while they call port $\bar{a}$ when we decrease them (we will reach deadlock)

* hence, we will define a new process B that gets created when we increment A

* graphically it will look as follows



which can be formally defined as

$$Z \overset{\text{def}}{=} z.Z + i.(A \mid a.Z) \backslash a$$

$$A \overset{\text{def}}{=} d.\bar{a}.0 + i.(B \mid b.A) \backslash b$$

$$B \overset{\text{def}}{=} d.\bar{b}.0 + i.(A \mid a.B) \backslash a$$

* we can also look at the transition graph

$$\begin{tikzcd}
0 & Z \arrow[d, "i"] & \sim & (0 \mid Z)\backslash a & (\bar{a}.0 \mid a.Z)\backslash a \arrow[l, "\tau"'] \\
1 & (A \mid a.Z)\backslash a \arrow[d, "i"] \arrow[urrr, "d"] & \sim & ((0 \mid A)\backslash b \mid a.Z)\backslash a & ((\bar{b}.0 \mid b.A)\backslash b \mid a.Z)\backslash a \arrow[l, "\tau"'] \\
2 & ((B \mid b.A)\backslash b \mid a.Z)\backslash a \arrow[d, "i"] \arrow[urrr, "d"] & & & \cdots \\
3 & (((A \mid a.B)\backslash a \mid b.A)\backslash b \mid a.Z)\backslash a \arrow[d, "i"] \arrow[urrr, "d"] & & & \\
\vdots & \vdots & & &
\end{tikzcd}$$