Concurrency and Concurrent Programming

Problem Sheet 2 - Sample Answers

1. This question was relatively straightforward, and most people got it ok.

```
TX = (in1?x -> mid!1 -> mid!x -> TX
    | in2?x -> mid!2 -> mid!x -> TX)

RX = (mid?1 -> mid?x -> out1!x -> RX
    | mid?2 -> mid?x -> out2!x -> RX)
```

2. The solution to this question involves expanding the definitions given in question 1 using the interaction rules. Note that we conceal the mid channel, using the \ operator. In doing this, we hide all events which communicate along this channel.

```
(TX || RX) \{mid}

= ( in1?x -> mid!1 -> mid!x -> TX
  | in2?x -> mid!2 -> mid!x -> TX) || (mid?1 -> mid?x -> out1!x -> RX
                                     | mid?2 -> mid?x -> out2!x -> RX)  \{mid}

= ( in1?x -> (mid!1 -> mid!x -> TX) || ( mid?1 -> mid?x -> out1!x -> RX
                                     | mid?2 -> mid?x -> out2!x -> RX)  \{mid}
  | in2?x -> (mid!2 -> mid!x -> TX) || ( mid?1 -> mid?x -> out1!x -> RX
                                     | mid?2 -> mid?x -> out2!x -> RX))  \{mid}

= ( in1?x -> (mid!x -> TX) || (mid?x -> out1!x -> RX)
  | in2?x -> (mid!x -> TX) || (mid?x -> out2!x -> RX))  \{mid}

= ( in1?x -> TX || (out1!x -> RX)
  | in2?x -> TX || (out2!x -> RX))  \{mid}
```

Concealment operator - just crossed out of the solution

So the protocol effectively works like two buffers, and the inputs are all sent to the correct outputs.

Note that some people tried to continue this process by simplifying the expression to the following form.

```
( in1?x -> out1!x -> TX || RX
| in2?x -> out2!x -> TX || RX)
```

This is NOT correct, since we must consider the possible events of both processes, and doing so we see that in fact two events are possible, either the output by RX shown above, OR, the next input by process TX.

4. The secret of this question is to come up with a coding for zero and one so that transmission errors can be detected by the RECEIVING process. Any other solution invariably requires either the existence of a safe channel from TX to RX, or can only detect, but not correct the errors. The simplest such coding is:

1 => 1 0
0 => 0 1

A solution to the problem is thus:

TX = in?x -> X

X = mid!x -> mid!(1-x) -> (check?ok -> TX | check?fail -> X)

RX = mid?x -> (mid?(1-x) -> out!x -> check!ok -> RX
| mid?x -> check!fail -> RX)

(Or an equivalent solution using IF statements.)

Some people designed a communication protocol which used the main data channel to send signals "ok" and "fail" to the receiver. This protocol is would not work because the ok / fail signals are subject to faulty transmission in the same way as the data.

5. This question required you to give a case analysis based upon the CCP specification given in the answer to question 4.

CASES

(i) $x \neq (1-x)$. (i.e. The case where no 1's are dropped).

TX || RX \{ mid,check }

= (TX || RX) \{mid, check}

= (in?x -> X) || RX \{mid,check}

= in?x -> (X || RX) \{mid,check}

= in?x -> ((check?ok -> TX | check?fail -> X) ||
(out!x -> check!ok -> RX)) \{mid,check}

= in?x -> out!x -> ((check?ok -> TX | check?fail -> X) ||
(check!ok -> RX))

= in?x -> out!x -> (TX || RX) \{mid,check}

This is a buffer process!

(ii) x = (1-x) . (i.e. The case where 1's are dropped).

= (TX || RX) \{mid, check}

= (in?x -> X) || RX \{mid,check}

= in?x -> (X || RX) \{mid,check}

= in?x -> ((check?ok -> TX | check?fail -> X) ||
(check!fail -> RX)) \{mid,check}

= in?x -> (X || RX) \{mid,check}

(**) Hence, as long as bits are dropped, we have

= in?x -> (X || RX) \{mid,check}

and as soon as the bit is preserved, we have

= in?x -> out!x -> (TX || RX) \{mid,check}

Hence result.

PJA and PJW 12-11-90

Concurrency and Concurrent Programming

Problem Sheet 3 - Sample Answers

2. The basic problem here is fairly straightforward. The trick is to devise a pair of user processes which might deadlock, then a simple application of the CSP rules will prove that this deadlock occurs.

P = acquire -> μX. (print?x -> X
| release -> P)

V = acquire -> μX. (display?x -> X
| move?x -> X
| release -> V)

U1 = p.acquire -> v.acquire -> S
U2 = v.acquire -> p.acquire -> T

(p:P || v:V) || (U1 ||| U2)
= (p.acquire -> (p:X || v:V) ||(v.acquire -> S) ||| U2
[] v.acquire -> (p:P || v:X) ||U1 ||| (p.acquire -> T))

= (p.acquire -> v.acquire -> (p:X || v:X) ||((v.acquire -> S) ||| (p.acquire -> T)
[] (S ||| U2))
[] v.acquire -> p.acquire -> (p:X || v:X) ||((v.acquire -> S) ||| (p.acquire -> T)
[] (U1 ||| T)))

= (p.acquire -> v.acquire -> STOP
[] v.acquire -> p.acquire -> STOP)

This last line is possible since in the first and third line no events are possible, and, as you will recall from the notes, the mere possibility of deadlock ensures failure - the system thus deadlocks.

There are several possible ways to avoid deadlock, all of which place limitations on the way in which resources may be used. Possibilities include not allowing the acquisition of more than one resource, enforcing an ordering on such acquisitions (e.g. printer must be acquired before display unit), or using a resource manager which must be acquired before the individual resources are acquired.

3. The definitions for the transmitter and receiver are the same as in exercises 2. The only new definition is the wire, which most you of did satisfactorily using the non-deterministic or. The full set of definitions is thus:

TX = in?x -> X
X = left!x -> left!(1-x) -> check?y -> if y=ok then TX
else X

WIRE = (left?z -> if z=0 then right!0 -> WIRE
else (right!1 -> WIRE
[] right!0 -> WIRE))

RX = right?a -> right?b -> if a=b then check!fail -> RX
else out!a -> check!ok -> RX)

Some of you managed to produce shorter definitions of the wire, of the form:

WIRE = (left?x -> (right!x -> WIRE [] right!0 -> WIRE)

Although this is a valid definition, I don't think it is as good, since it doesn't clearly identify the part representing correct transmission, and the part representing error. Such a specification makes the ensuing proof more difficult. In the next part many people split the proof into several separate cases (zeroes and ones, correct and error transmission). This is sometimes a valid technique, but you must be careful not to lose the formality of the proof by using text to describe how the separate cases satisfy the proof. The interaction (etc) rules of CSP are powerful enough to reason about the whole system at once, and although

this sometimes makes a proof large, at least we can be sure that it is formally proved. The proof below is shown in full. In practice it would be possible to contract some of the steps together

```
(TX||WIRE||RX)\{left,right,check}

TX = (in?x -> X) ||

        (left?z -> if z=0 then right!0 -> WIRE
                   else ( right!1 -> WIRE
                        [] right!0 -> WIRE)) ||

                    right?a -> right?b -> if a=b then check!fail -> RX
                                          else out!a -> check!ok -> RX)

                                                          Expansion rule
= in?x -> ((left!x -> left!(1-x) -> check?y -> if y=ok then TX
                                               else X ) ||

        (left?z -> if z=0 then right!0 -> WIRE
                   else ( right!1 -> WIRE
                        [] right!0 -> WIRE)) ||

                    (right?a -> right?b -> if a=b then check!fail -> RX
                                           else out!a -> check!ok -> RX)\{left,right,check})

                                                          Expansion rule
                                                          (communication of x on the left channel hidden).
= in?x -> ((left!(1-x) -> check?y -> if y=ok then TX
                                     else X ) ||

        (if x=0 then right!0 -> WIRE
                else ( right!1 -> WIRE
                     [] right!0 -> WIRE)) ||

                    (right?a -> right?b -> if a=b then check!fail -> RX
                                           else out!a -> check!ok -> RX)\{left,right,check})

                                                          x takes values in { 1,0 } hence
= (in?0 -> ((left!1 -> check?y -> if y=ok then TX
                                  else X ) ||

        ( right!0 -> WIRE ) ||

                    (right?a -> right?b -> if a=b then check!fail -> RX
                                           else out!a -> check!ok -> RX))

| in?1 -> ((left!0 -> check?y -> if y=ok then TX
                                 else X ) ||

                   ( right!1 -> WIRE
                     [] right!0 -> WIRE) ||

                    (right?a -> right?b -> if a=b then check!fail -> RX
                                           else out!a -> check!ok -> RX))\{left,right,check})
```

Expansion rule + property of non-deterministic OR
(communication along the right channel hidden).

```
= (in?0 -> ((left!1 -> check?y -> if y=ok then TX
                                  else X ) ||

            WIRE ||

           ( right?b -> if 0=b then check!fail -> RX
                                  else out!a -> check!ok -> RX))

| in?1 -> ((left!0 -> check?y -> if y=ok then TX
                                  else X ) ||

           WIRE ||

           ((right?b -> if 1=b then check!fail -> RX
                                  else out!a -> check!ok -> RX)
          [] (right?b -> if 0=b then check!fail -> RX
                                  else out!a -> check!ok -> RX))\{left,right,check}))
```

Expansion rule
(communication along left channel hidden).

```
= (in?0 -> (( check?y -> if y=ok then TX
                         else X ) ||

           ( right!1 -> WIRE
           [] right!0 -> WIRE) ||

           ( right?b -> if 0=b then check!fail -> RX
                                  else out!a -> check!ok -> RX))

| in?1 -> (( check?y -> if y=ok then TX
                                  else X ) ||

           ( right!0 -> WIRE) ||

           ((right?b -> if 1=b then check!fail -> RX
                                  else out!a -> check!ok -> RX)
          [] (right?b -> if 0=b then check!fail -> RX
                                  else out!a -> check!ok -> RX))\{left,right,check}))
```

Expansion rule.
(communication along the right channel hidden).

```
= (in?0 -> (( check?y -> if y=ok then TX
                         else X ) ||

           WIRE ||

           ( check!fail -> RX
          [] out!a -> check!ok -> RX))

| in?1 -> (( check?y -> if y=ok then TX
                          else X ) ||

           WIRE ||

           (( out!a -> check!ok -> RX)
          [] ( check!fail -> RX))) \{left,right,check})
```

Expansion rule
(communication along the chack channel hidden).

= (in?0 -> (out!0 -> (check?ok -> TX) || (WIRE) || (check!ok -> RX)
[] X || WIRE || RX)

| in?1 -> (out!1 -> (check?ok -> TX || (WIRE) || (check!ok -> RX)
[] X || WIRE || RX))\{left,right,check})

= (in?0 -> (out!0 -> TX || WIRE || RX
[] X || WIRE || RX)

| in?1 -> (out!1 -> TX || WIRE || RX
[] X || WIRE || RX))\{left,right,check})

= in?x -> (out!x -> TX || WIRE || RX
[] X || WIRE || RX)\{left,right,check})

= in?x -> out!x -> TX || WIRE || RX\{left,right,check})

This last line is reached eventually, since the previous equation represents an infinite set of identical choices.

PJA and PJW 19-11-90

Concurrency and Concurrent Programming
Problem Sheet 4 - Sample Answers

1. There were two possible solutions to this question, depending upon whether or not you realised that there was an error in the given definition of process P. The line Y2:=Y2-1, should have read Y1:=Y1-1.

First the solution to the corrected algorithm

```
        P
start:  if y1 = (n-k) goto end;
        a.acquire ;
        y3 := y3 * y1 ;
        a.release ;
        y1 := y1 - 1 ;
        goto start ;
end:    halt
```

```
        Q
start:  if y2 = k goto end ;
        y2 := y2 + 1 ;
        loop until y1 + y2 =< n ;
        a.acquire ;
        y3 := y3 / y2 ;
        a.release ;
        goto start ;
end:    halt
```

Notes

The semaphores are used to enforce mutual exclusion around the commands which update y3. This is necessary in order to prevent the "lost update problem" (interference).

Some people also put references to y1 inside semaphores as well, because both P and Q use it. However, this is not useful, since only one process (process P) can ever **change** it. The fact that Q could be reading y1 while P is changing it, does not cause interference (Assuming that the physical read and write events are atomic). The only possible problem that can occur, is that the value read by Q is out of date, by the time the instruction is complete. However, in this particular algorithm, this is not a problem.

Some people used a semaphore to encapsulated the whole of the loop (in process Q). However, this would have the effect of preventing the termination of the loop, since y1 could never be decremented.

Now the solution to the uncorrected algorithm

```
        P
start:  if y1 = (n-k) goto end;
        a.acquire ;
        y3 := y3 * y1 ;
        a.release ;
        b.acquire ;
        y2 := y2 - 1 ;
        b.release ;
        goto start ;
end:    halt
```

```
        Q
start:  if y2 = k goto end ;
        b.acquire ;
        y2 := y2 + 1 ;
        b.release ;
        loop until y1 + y2 =< n ;
        a.acquire ;
        y3 := y3 / y2 ;
        a.release ;
        goto start ;
end:    halt
```

Notes

Some people used the same semaphore to enforce mutual exclusion on both y3 and y2. Although this is not wrong, it does unnecessarily reduce parallelism. For example when P is writing to y3, Q would be excluded from writing to y2, even though there is no possibility of interference. The above solution uses two semaphores, to overcome this.

2. There were several solutions to this question, the best of which used three parallel processes :
a card reader ,
a filter for to deal with *'s, and
a printer.

READER = READER $_1$

READER $_i$ = innewcard -> READCHAR $_i$

READCHAR $_i$ = read $_i$?char -> filt!char -> if i<80 then READCHAR $_{(i+1)}$
else filt!' ' -> outcard -> READER

FILTER = filt?ch -> if ch='*' then FILTER2
else print!ch -> FILTER

FILTER2 = filt?ch -> if ch='*' then print!'^' -> FILTER
else print!'*' -> print!ch -> FILTER

PRINTER = PRINTER $_i$

PRINTER $_i$ = print?x -> if x = eof then END $_i$
else output $_i$!x -> if i<125 then PRINTER $_{(i+1)}$
else newline -> PRINTER $_i$

END $_i$ = output $_i$!' ' -> if i<125 then END $_{i+1}$
else PRINTER

CONWAYPROBLEM = (READER || FILTER || PRINTER)

3. This question, was a little unclear. Most people just quoted the appropriate trace, when what was actually required was a derivation of the trace, using proof by induction.
Theorem:

$$traces(SEM) = \left\{ s \mid s \leq (<acquire, release>)^* \right\}$$

(where the trace definition is a regular expression)

Proof:
By definition

$$traces(\mu X \cdot F(X)) = \bigcup_{n \geq 0} traces(F^n(STOP))$$

Base Case

$$F^0(STOP) = <> = (<acquire, release>)^0$$

Induction Case
assume

$$traces(F^n(STOP)) = \left\{ s \mid s \leq (<acquire, release>)^n \right\}$$

Inductive Step

$$traces(F^{n+1}(STOP)) = traces(acquire \rightarrow release \rightarrow F^n(STOP))$$

$$traces(F^{n+1}(STOP)) = \{ <acquire>\hat{}t \mid t \in traces(release \rightarrow F^{n}(STOP)) \}$$
$$\cup \{ <> \}$$

$$traces(F^{n+1}(STOP)) = \{ <acquire>\hat{}t \mid t \leq \{ <release>\hat{}s \mid s \leq (<acquire, release>)^{n} \} \}$$
$$\cup \{ <> \}$$

se traces (F^n(STOP)) } EOR|

$$= \{ s \mid s \leq (<acquire, release>)^{n+1} \}$$

And hence by induction the theorem is proved.

4. This question is more complex than most of you thought, since you must solve the recursive equation to produce an explicit definition of the process trace.

The '+' and '*' operators are not standard to CSP. The + operator should be read as OR, and * should be read as meaning n for all n >= 0.

Theorem:

$$traces(GR) = \{ s \mid s \leq (<heathrow> + <victoria> + <in\ 10p, euston>)^{*} \}$$

(where the trace definition is a regular expression)

Proof:
By definition

$$traces(\mu X \cdot F(X)) = \bigcup_{n \geq 0} traces(F^{n}(STOP))$$

Base Case

$$F^{0}(STOP) = <> = (<heathrow> + <victoria> + <in\ 10p, euston>)^{0}$$

Induction Case
assume

$$traces(F^{n}(STOP)) = \{ s \mid s \leq (<heathrow> + <victoria> + <in\ 10p, euston>)^{n} \}$$

Inductive Step

$$traces(F^{n+1}(STOP)) = traces(heathrow \rightarrow F^{n}(STOP)$$
$$\mid victoria \rightarrow F^{n}(STOP)$$
$$\mid in\ 10p \rightarrow euston \rightarrow F^{n}(STOP))$$

$$traces(F^{n+1}(STOP)) = \{ <heathrow>\hat{}s \mid s \in traces(F^{n}(STOP)) \}$$
$$\cup \{ <victoria>\hat{}s \mid s \in traces(F^{n}(STOP)) \}$$
$$\cup \{ <in\ 10p>\hat{}s \mid s \in traces(euston \rightarrow F^{n}(STOP)) \}$$

$$\cup\{<>\}$$

$$traces(F^{n+1}(STOP)) = \{ <heathrow>\hat{}s \mid s \leq (<heathrow>+<victoria>+<in\ 10p, euston>)^n \}$$
$$\cup \{ <victoria>\hat{}s \mid s \leq (<heathrow>+<victoria>+<in\ 10p, euston>)^n \}$$
$$\cup \{ <in\ 10p>\hat{}t \mid t \leq \{ <euston>\hat{}s \mid s \leq (<heathrow>+<victoria>+<in\ 10p, euston>)^n \} \}$$
$$\cup \{<>\}$$

$$= \{ s \mid s \leq (<heathrow>+<victoria>+<in\ 10p, euston>)^{n+1} \}$$

And hence by induction the theorem is proved.

PJA and PJW 28-11-90

Concurrency and Concurrent Programming
Problem Sheet 5 - Sample Answers

1. This question turned out to be slightly more difficult than it was intended to be, and only a few people got it correct.

COPY = (in -> out -> COPY | off -> STOP)

Theorem:

$$traces(F^n(STOP)) = \left\{ s \mid s \leq <in, out>^m \hat{} <off> \wedge 0 \leq m \leq n \right\} \cup \left\{ t \mid t \leq <in, out>^n \right\}$$

Proof: By definition

COPY = F(COPY) where F(COPY) = (in -> out -> COPY | off -> STOP)

$$traces(COPY) = \bigcup_{n \geq 0} traces(F^n(STOP))$$

Base Case: (n = 0)

$$F^0(STOP) = \left\{ <> \right\} = \left\{ t \mid t \leq <in, out>^0 \right\}$$

Inductive Case

$$traces(F^{n+1}(STOP)) = traces(in \rightarrow out \rightarrow F^n(STOP) \mid off \rightarrow STOP)$$

$$= \left\{ <> \right\} \cup \left\{ <in> \hat{} t \mid t \in traces(out \rightarrow F^n(STOP)) \right\} \cup \left\{ <off> \hat{} t \mid t \in traces(STOP) \right\}$$

$$= \left\{ <> \right\} \cup \left\{ t \mid t \leq <in, out> \hat{} s \mid s \in traces(F^n(STOP)) \right\} \cup \left\{ <off> \right\}$$

$$= \left\{ t \mid t \leq <in, out> \hat{} \left\{ s \mid s \leq <in, out>^m \hat{} <off> \wedge 0 \leq m < n \bigcup t' \mid t' \leq <in, out>^n \right\} \right\} \cup \left\{ <off> \right\}$$

$$= \left\{ t \mid t \leq <in, out>^{m+1} \hat{} <off> \wedge 1 \leq m+1 \leq n+1 \right\}$$
$$\cup \left\{ t' \mid t' \leq <in, out>^{n+1} \right\}$$
$$\cup \left\{ <off> \right\}$$

$$= \left\{ s \mid s \leq <in, out>^{m+1} \hat{} <off> \wedge 0 \leq m+1 \leq n+1 \right\}$$
$$\cup \left\{ t' \mid t' \leq <in, out>^{n+1} \right\}$$

2. Only a few people recognised the connection between this question and question 1. i.e. process R1 is actually the same as the COPY process in question 1.

R = a -> R1

R1 = (g -> d -> R1 | r -> R)

R = F(R)
R1 = G(R1)

F(R) = (a -> R1)
G(R1) = (g -> d -> r1 | r -> R)

First consider the inductive proof of function F.

$$traces(R) = \cup_{n \geq 0} traces(F^n(STOP))$$

Inductive assumption

$$traces(F^n(STOP)) = \left\{ t \mid t \leq (<a> \hat{} k)^n \ \wedge \ k \in \cup_{p \geq 0} \{ s \mid s \leq <g, d>^q \hat{} <r> \wedge 0 \leq q < p \} \right\}$$
$$\cup \left\{ t \mid (t \leq (<a> \hat{} k)^m \hat{} <a> \hat{} j) \wedge 0 \leq m < n \ \wedge \ j \in \cup_{p \geq 0} \{ s \mid s \leq <g, d>^p \} \right\}$$

Base Case

$$traces(F^0(STOP)) = \left\{ <> \right\}$$
$$= \left\{ t \mid t \leq (<a> \hat{} k)^0 \ \wedge \ k \in \cup_{p \geq 0} \{ s \mid s \leq <g, d>^q \hat{} <r> \wedge 0 \leq q < p \} \right\}$$

Inductive Step

$$traces(F^{n+1}(STOP)) = traces(a \rightarrow R1)$$

$$= \left\{ <> \right\}$$

$$\cup\left\{<a>\right\}$$

$$\cup\left\{<a>\hat{}t \mid t \in traces(R1)\right\}$$

Now since R1 is essentially the same as the COPY process from Question 1, traces (R1) is the same as traces (COPY) except with events in, out and off replaced by g, d and r respctively. Hence :

$$=\left\{<>\right\}\cup\left\{<a>\right\}\cup\left\{<a>\hat{}t\right.$$

$$\left.\wedge t \in \cup_{p\geq 0}\left\{ \{s \mid s \leq <g,d>^{q}\hat{}<r>\hat{}y \mid y \in traces(F^{n}(STOP)) \wedge 0 \leq q < p\} \cup \{w \mid w \leq <g,d>^{p}\}\right\}\right\}$$

$$=\left\{<>\right\}\cup\left\{<a>\right\}\cup\left\{<a>\hat{}t\right.$$

$$\wedge t \in \cup_{p\geq 0}\left[\left[\right.\right.\left\{s \mid s \leq <g,d>^{q}\hat{}<r>\hat{}y \mid y \leq (<a>\hat{}k)^{n}\right.$$

$$\left.\wedge\ k \in \cup_{p'\geq 0}\{s' \mid s' \leq <g,d>^{q'}\hat{}<r> \wedge 0 \leq q' < p'\} \wedge 0 \leq q < p\right\} \cup \{w \mid w \leq <g,d>^{p}\}$$

$$\cup\left\{ \{s \mid s \leq <g,d>^{q}\hat{}<r>\hat{}y \mid y \leq (<a>\hat{}k)^{m}\hat{}<a>\hat{}j \wedge 0 \leq m < n\right.$$

$$\left.\left.\left.\wedge\ j \in \cup_{p'\geq 0}\{s' \mid s' \leq <g,d>^{p'}\} \wedge 0 \leq q < p\right\} \cup \{w \mid w \leq <g,d>^{p}\}\right]\right]\right\}$$

$$traces(F^{n+1}(STOP)) = \left\{ t \mid t \leq (<a>\hat{}k)^{n+1} \ \wedge\ k \in \cup_{p\geq 0}\{s \mid s \leq <g,d>^{q}\hat{}<r> \wedge 0 \leq q < p\}\right\}$$

$$\cup\left\{ t \mid (t \leq (<a>\hat{}k)^{m+1}\hat{}<a>\hat{}j) \wedge 0 \leq m+1 < n+1 \ \wedge\ j \in \cup_{p\geq 0}\left\{ \{s \mid s \leq <g,d>^{p}\}\right\}\right\}$$

Hence result proved.

4. This question is simpler, and most of you who tried it got the answer out ok.

$$\alpha GR = \alpha TM$$

From question one we know the traces for GR, and by similar reasoning can get the traces for TM.

$$traces(GR) = \left\{ s \mid s \leq (<heathrow>+<victoria>+<in\ 10p, euston>)^{*}\right\}$$

$$traces(TM) = \left\{ s \mid s \leq (<in\ 50p, heathrow>+<in\ 20p, victoria>+<in\ 10p, euston>)^{*}\right\}$$

Now, because the alphabets are equal, we know that:

$$traces(GR \,||\, TM) = traces(GR) \cap traces(TM)$$

$$= \left\{ s \mid \begin{array}{l} s \leq (<heathrow>+<victoria>+<in\ 10p, euston>)^{*} \wedge \\ s \leq (<in\ 50p, heathrow>+<in\ 20p, victoria>+<in\ 10p, euston>)^{*} \end{array}\right\}$$

$$= \left\{ s \mid s \leq <in\ 10p, euston>^{*}\right\}$$

Which we obtain by combining the two regular expressions into one representing the intersection of the individual expressions. But we also know, by the same reasoning as used in question 1, that

$$traces(\mu X \cdot (in\ 10p \rightarrow euston \rightarrow X))$$

$$= \left\{ s \mid s \leq <in\ 10p, euston>^{*} \right\}$$

And hence the required result is proved.

5. This question is quite straightforward, just requiring an application of Scott induction to consider the different traces of the process. In order to prove the property, what we really need to prove is the following.

For the process:

$$VM = \mu X \cdot (\ coin \rightarrow choc \rightarrow X$$
$$\mid\ choc \rightarrow coin \rightarrow X\ )$$

We must prove the property **P**:

$$P(tr) = \left[0 \leq (tr \downarrow coin + 1) - (tr \downarrow choc) \leq 2 \right]$$
$$= \left[-1 \leq (tr \downarrow coin) - (tr \downarrow choc) \leq 1 \right]$$

For all traces of the process, i.e. that VM **sat** P. We prove this using Scott induction by showing that STOP **sat** P and then that

$$\#X\ \textbf{sat}\ P \Rightarrow F(X)\ \textbf{sat}\ P\#$$

We first prove the base case, (using traces(STOP) = {<>}):

$$P(<>) = (-1 \leq <> \downarrow coin - <> \downarrow choc \leq 1)$$
$$= (-1 \leq 0 \leq 1) = true$$

For the step case, we assume

$$\#X\ \textbf{sat}\ P\#$$

and attempt to prove

$$\#F(X)\ \textbf{sat}\ P\#.Firstwe$$

must derive the possible traces of F(X), and then must prove that the property holds for each such trace.

$$traces(F(X)) = traces(coin \rightarrow choc \rightarrow X$$
$$\mid\ choc \rightarrow coin \rightarrow X\ )$$
$$= \left\{ <> \right\}$$
$$\cup \left\{ <coin>\hat{}s \mid s \in traces(choc \rightarrow X) \right\}$$
$$\cup \left\{ <choc>\hat{}s \mid s \in traces(coin \rightarrow X) \right\}$$
$$= \left\{ <> \right\}$$
$$\cup \left\{ <coin> \right\}$$
$$\cup \left\{ <coin, choc>\hat{}s \mid s \in traces(X) \right\}$$
$$\cup \left\{ <choc> \right\}$$

$$\cup\left\{ <choc, coin>\hat{}s \mid s \in traces(X) \right\}$$

In order to complete the induction we must prove that the property holds for each element of these sets.

$$P(<>) = true$$

$$P(<coin>) = \left[-1 \le (1-0) \le 1 \right] = true$$

$$P(<choc>) = \left[-1 \le (0-1) \le 1 \right] = true$$

$$P(<coin, choc>\hat{}s) = \left[-1 \le (1-1) + (sda coin - sda choc) \le 1 \right] = true$$

(Since the assumption guarantees that #P(s)# is true)

$$P(<choc, coin>\hat{}s) = \left[-1 \le (1-1) + (sda coin - sda choc) \le 1 \right] = true$$

And hence by induction, the property is satisfied.

PJA and PJW 28-11-90

Concurrency and Concurrent Programming
Problem Sheet 6 - Sample Answers

1. Almost none of you got this question right, forgetting, (as you did before), that in order to obtain an expression for the traces of a recursive process, we must solve a recursive equation. If we do not do this, then result we have is merely an implicit definition of the traces of the process.
Theorem:

$$traces(T) = \left\{ s \mid s \leq (<in\,1?x, mid\,!1, mid\,!x> + <in\,2?x, mid\,!2, mid\,!x>)^* \right\}$$

Proof is by Scott induction:
Base case.

$$traces(STOP) = <> = (<in\,1?x, mid\,!1, mid\,!x> + <in\,2?x, mid\,!2, mid\,!x>)^0$$

Inductive case, assuming:

$$traces(X) = \left\{ s \mid s \leq (<in\,1?x, mid\,!1, mid\,!x> + <in\,2?x, mid\,!2, mid\,!x>)^n \right\}$$

We must prove the inductive step:

$$traces(F(X)) = traces(in\,1?x \rightarrow mid\,!1 \rightarrow mid\,!x \rightarrow X \mid in\,2?x \rightarrow mid\,!2 \rightarrow mid\,!x \rightarrow X)$$

$$traces(F(X)) = \left\{ <in\,1?x> \hat{}\, s \mid s \in traces(mid\,!1 \rightarrow mid\,!x \rightarrow X) \right\} \cup \left\{ <in\,2?x> \hat{}\, s \mid s \in traces(mid\,!2 \rightarrow mid\,!x \rightarrow X) \right\} \cup \left\{ <> \right\}$$

$$= \left\{ <in\,1?x, mid\,!1> \hat{}\, s \mid s \in traces(mid\,!x \rightarrow X) \right\} \cup \left\{ <in\,2?x, mid\,!2> \hat{}\, s \mid s \in traces(mid\,!x \rightarrow X) \right\} \cup \left\{ <> \right\}$$

$$= \left\{ <in\,1?x, mid\,!1, mid\,!x> \hat{}\, s \mid s \in traces(X) \right\} \cup \left\{ <in\,2?x, mid\,!2, mid\,!x> \hat{}\, s \mid s \in traces(X) \right\} \cup \left\{ <> \right\}$$

$$= \left\{ s \mid s \leq (<in\,1?x, mid\,!1, mid\,!x> + <in\,2?x, mid\,!2, mid\,!x>)^{n+1} \right\}$$

By using the inductive assumption, and combining the two cases, and hence the theorem is proved.
Similarly for the receiver process, we can derive:

$$traces(T) = \left\{ s \mid s \leq (<mid?\,1, mid?x, out\,1!x> + <mid?\,2, mid?x, out\,2!x>)^* \right\}$$

Having proved the traces for T and R, it is possible to define the traces for (T||R) by using the rule provided (pg 72, section 2.3.3 L1). Hence :

$$traces(T||R) = \left\{ s \mid (s \upharpoonright \alpha T) \in traces(T) \wedge (s \upharpoonright \alpha R) \in traces(R) \wedge s \in (\alpha T \cup \alpha R)^* \right\}$$

However, because we want to hide all events in α mid T we must modify this law slightly as follows:

$$traces\left[(T||R) \setminus \alpha midT \right]$$

$$= traces\left[(T \setminus \alpha midT) || (R \setminus \alpha midT) \right]$$

$$= \left\{ s \mid (s \upharpoonright (\alpha T - \alpha midT)) \in traces(T) \right.$$
$$\wedge (s \upharpoonright (\alpha R - \alpha midT)) \in traces(R)$$
$$\left. \wedge\, s \in ((\alpha T - \alpha midT) \cup (\alpha R - \alpha midT))^* \right\}$$

$$= \left\{ s \mid (s \upharpoonright \{in\,1?x, in\,2?x\}) \in traces(T) \right.$$
$$\wedge (s \upharpoonright \{out\,1!x, out\,2!x\}) \in traces(R)$$
$$\left. \wedge\, s \in (\{in\,1?x, in\,2?x\} \cup \{out\,1!x, out\,2!x\})^* \right\}$$

3. In this question some of you failed to realise that the proof required involves the actual values transmitted down the channels, rather than just the number of values. Given this information the proof follows through fairly simply.

Again using Scott induction, we must prove the property for the base and step cases. For the base case we must prove the property for *STOP* .

$$traces(STOP) = \left\{ <> \right\}$$

$$P(<>) = \left(<> \downarrow right \leq^1 double^*(<> \downarrow left) \right)$$

$$= \left(<> \leq^1 <> \right) \equiv true$$

For the step case, we assume the property holds for X , and attempt to prove it for $F(X)$.

$$traces(F(X)) = \left\{ <> \right\}$$

$$\cup\left\{<left?x>\right\}$$

$$\cup\left\{<left?x \rightarrow right!(2x)>\right\}$$

$$\cup\left\{<left?x \rightarrow right!(2x)>\hat{}s \mid s \leq traces(X)\right\}$$

To complete the induction we must prove that the property holds for each of these possible traces.

$$P(<>) = true$$

$$P(<left?x>) = \left[<> \leq^1 double^* <x>\right] \equiv \left[<> \leq^1 <2x>\right] \equiv true$$

$$P(<left?x \rightarrow right!(2x)>) = \left[<2x> \leq^1 double^* <x>\right] \equiv \left[<2x> \leq^1 <2x>\right] \equiv true$$

$$P(<left?x \rightarrow right!(2x)>\hat{}s) = \left[<2x>\hat{}(s \downarrow right) \leq^1 double^*(<x>\hat{}(s \downarrow left))\right]$$

$$= \left[<2x>\hat{}(s \downarrow right) \leq^1 <x>\hat{}(double^*(s \downarrow left))\right]$$

$$\equiv \left[s \downarrow right \leq^1 double^*(s \downarrow left)\right] \equiv true$$

From the inductive assupmtion.

PJA and PJW 11-1-91