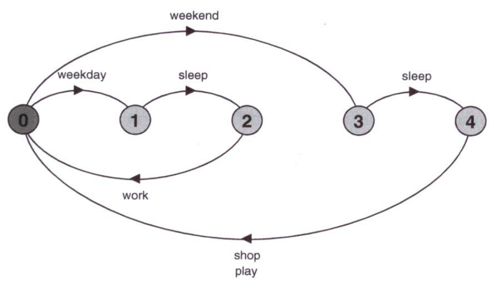
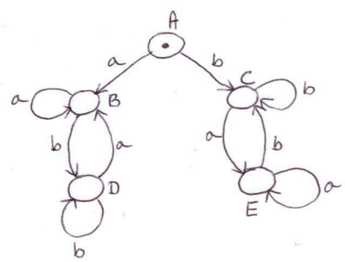
CS 3BB4. Sample solutions to the assignment 1.

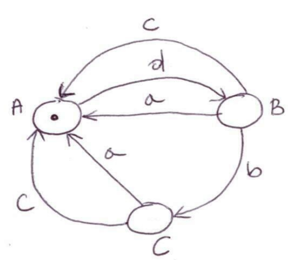
Total of this assignment is 140 pts. Each assignment is worth 10% of total.

If you think your solution has been marked wrongly, write a short memo stating where marking in wrong and what you think is right, and resubmit to me during class, office hours, or just slip under the door to my office. The deadline for a complaint is 2 weeks after the assignment is marked and returned

1.[10] For each one of the following three processes, give the Finite State Processes (FSP) description of the labelled transition graph.







Solution:

a.[2]
```
PERSON = (weekday -> sleep -> work -> PERSON
         |weekend -> sleep -> (shop -> PERSON | play -> PERSON)
         ).
```

b.[4]
```
A = (a -> B | b -> C)
B = (a -> B | b -> D)
C = (a -> E | b -> C)
D = (a -> B | b -> D)
E = (a -> E | b -> C)
```

c.[4]
```
A = (d -> B)
B = (a -> A | b -> C | c -> A)
C = (a -> A | c -> A)
```

2.[14] A miniature portable FM radio has three controls. An on/off switch turns the device on and off. Tuning is controlled by two buttons scan and reset which operates as follows. When the radio is turned on or reset is pressed, the radio is tuned to the top frequency of the FM band (108 MHz). When scan is pressed, the radio scans towards the bottom of the band (88 MHz). It stop scanning when it locks onto a station or it reaches the bottom (end). If the radio is currently tuned to a station and scan is pressed then it start to scan from the frequency of that station towards the bottom. Similarly, when reset is pressed the receiver tunes to the top.

a.[10] Model the radio as a FSP process RADIO.

b.[4] Provide an appropriate labelled transition system.
Hint: The alphabet of RADIO is {on, off, scan, reset, lock, end}.

Solution:
a.[10] FSP:

```
 *
 * The following process models a miniature portable Fm radio.
 *
 * The behavior of the radio is described as:
 *
 * (i) action 'on' turns the radio on at the top frequency of the
 * FM band (108 MHz).
 *
 * (ii) action 'off' turns the radio off.
 *
 * (iii) action 'scan' moves the tunning knob downwards one frequency
 * at the time. Scanning is only allowed as long as the botton frequency
 * (88 MHz) is not reached.
 *
 * (iv) action 'end' signals that the end frequency has been reached.
 *
 * (v) action 'lock' locks the tunning knob and stops scanning.
 *
 * (vi) action 'reset' tunes the radio to the top frequency of the FM
 * band (108 MHz).
 *
 * In addition, scanning starts from the  frequency the radio is
 * currently at.
 *
 * It is assumed that when the radio is scanning a new frequency no
 * other action than 'lock' is allowed. */

const Top = 108
const Bot = 88
range Band = Bot..Top

RADIO = (
 on → RADIO[Top] ),

 RADIO[frequency: Band] = (
  off → RADIO
  | reset → RADIO[Top]
  | when frequency > Bot scan[frequency] → SCANING[frequency-1] ),

 SCANING[frequency: Band] = (
  when frequency > Bot scan[frequency] → SCANING[frequency-1]
  | when frequency == Bot end → SCANING[Bot]
  | lock[frequency] → RADIO[frequency] ).
```

b.[4] LTS: Just use LTSA tool, however you may be forced to change the range of frequency because the system might crush.

3.[10] Program the radio of Question 2 in Java, complete with graphic display (if you can).

Java solutions are not provided.

4.[8] Model the following Road Deicing protocol as FSP. The road could be in one of the following states: Predicted Safe For Use, Predicted Unsafe For Traffic But Open, Closed. If road is 'Predicted Safe For Use', coming 'Predicted Ice Formation' changes its status to 'Predicted Unsafe For Traffic But Open'. If road is 'Predicted Unsafe For Traffic But Open', ice may melt (i.e. action 'Ice melts' occurs) and the road is again 'Predicted Safe For Use', or it becomes unsafe for use (action 'Unsafe for Use') and it is in the state 'Closed', or it is treated (action 'Road treated') and it is in the state 'Predicted Safe For Use' again. If the road is 'Closed', either 'Ice melts' or it is treated (action 'Road treated'), in both cases it becomes 'Predicted Safe For Use'.
Hint: The processes: ROAD-DEICING, PREDICTED-SAFE, PREDICTED-UNSAFE, CLOSED.

Solution:

```
ROAD-DEICING = ( predicted-safe -> PREDICTED-SAFE |
                 predicted-unsafe -> PREDICTED-UNSAFE )
PREDICTED-SAFE = ( predicted-ice-info -> PREDICTED-UNSAFE)
PREDICTED-UNSAFE = (ice-melts - > PREDICTED-SAFE |
                    road-treated - > PREDICTED-SAFE |
                    unsafe-for-use -> CLOSED)
CLOSED = (ice-melts - > PREDICTED-SAFE | road-treated - > PREDICTED-SAFE )
```

5.[12] Consider the following set of FSPs:

A = (a → (b → B) | c → (a → A | c → B) | c → C)
B = (b → (a → A | c → (a → A | b → C)))
C = (a → (b → (c → B)))

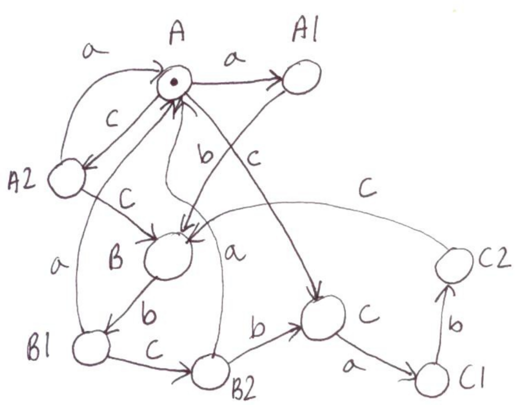
a. Construct an equivalent Labelled Transition System using the rules from page 16 of Lecture Notes 2.
b. Use LTSA to derive appropriate LTS, and, if different then yours, analyse and explain differences.

Solution:

a.[9] First we have to transform FSPs into the blue form from page 16 of Lecture Notes 2.

```
A  = (a -> A1 | c -> A2 | c -> C)
A1 = (b -> B)
A2 = (a -> A | c -> B)
B  = (b -> B1)
B1 = (a -> A | c -> B2)
B2 = (a -> A | b -> C)
C  = (a -> C1)
C1 = (b -> C2)
C2 = (c -> B)
```

Now we have:



b.[3] LTSA solution is not provided.

6.[8] ELEMENT = (up → down → ELEMENT) accepts an up action and then a down action. Using parallel composition '||' and the ELEMENT process describe a model that can accept up to four up actions before a down action.

Solution:

```
ELEMENT = (up->down->ELEMENT).

||COUNT(N = 4) = (forall[i:0..N-1] e[i]:ELEMENT)
                  /{ up/e[0].up, down/e[N-1].down,
                     forall[i:0..N-2] {e[i].down/e[i+1].up}
                   }@{up,down}.

/* look at the LTS that results from RUN */
```

```
/* there is an alternative rather neater recursive definition */

||COUNTR(N=4) =  if (N == 1) then
                    ELEMENT
                 else
                    (ELEMENT/{mid/down} || COUNTR(N-1)/{mid/up})
                 @{up,down}.
```

7.[8] Model the system from page 10 of Lecture Notes 3 as a composition of *FSP* processes. In this case, the entities that are represented by places in the Petri Nets model, must be represented by actions/transitions in *FSP* model.

Solution: (a possible one, bonus for using labelling)

```
COMP1 = (idle1 -> (read1 -> COMP1 | write1 -> COMP1))
COMP2 = (idle2 -> (read2 -> COMP2 | write2 -> COMP2))
MUT     = (write1 -> MUT | write2 -> MUT)
||TWPCOMP = COMP1||COMP2||MUT
```

8.[10] Model the system from page 13 of Lecture Notes 3 as a composition of *FSP* processes.

Solution (not unique):

We use two standard buffers, one for *message* and another for *acknowledgment*.

```
SENDER = proc.1 -> READY_TO_SEND
READY_TO_SEND = send_msg -> WAIT_FOR_ACK
WAIT_FOR_ACK = receive_ack -> ACK_RECEIVED
ACK_RECEIVED = proc.1 -> READY_TO_SEND

RECEIVER = READY_TO_RECEIVE
READY_TO_RECEIVE = recive_msg -> MSG_RECEIVED
MSG_RECEIVED = send_ack -> ACK_SENT
ACK_SENT = proc.1 -> READY_TO_RECEIVE

BUFFER = get -> put -> BUFFER

||SENDER_RECEIVER=(SENDER||RECEIVER||message:BUFFER||ack:BUFFER)
                /{send_msg/message.get, receive_message/message.put,
                  sent_ack/ack.get, receive_ack/ack.put}
```

9.[25] Consider the following simplified 'ancient' library system. Users can access the library by three desks; the request desk, the collection desk and return desk. In the library all books are kept in the stack and each book has an index card. A potential borrower enters the library system at the request desk where a particular book may be requested. If the book is in the library it is taken from the stack and the borrowed index is updated. The user gets the book at the collection desk. When a user returns a book, he does it so via the return desk; the book is put back in the stack and the index is appropriately updated.

a.[10] Model the system using Elementary Petri Nets.

b.[10] Model the system as a composition of *FSP* processes.

c.[5] Compare these two models, discuss similarities and differences.

a.[10]

b.[10]

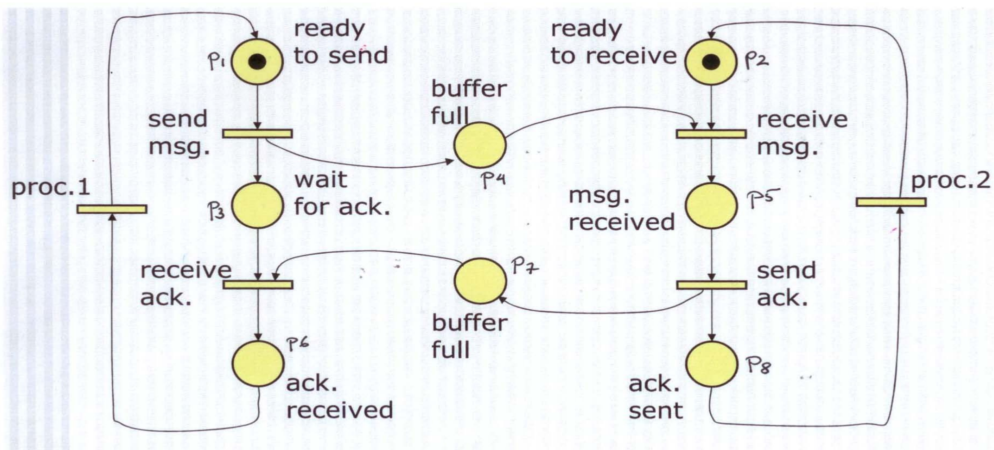
```
READER = going_to_lib -> ARRIVED

ARRIVED = (to_Request_Desk -> REQUEST_DESK
          |to_Return_Desk -> RETURN_DESK)

REQUEST_DESK = (book_is_borrowed -> msg_no_book -> COLLECTION_D_NO
               |book_is_update_index -> book_sent ->COLLECTION_D_YES)

COLLECTION_D_NO = (to_home_empty -> READER
                  |to_home_empty -> REQUEST_DESK
                  |to_home_empty -> RETURN_DESK)

COLLECTION_D_YES = (take_book -> READER
                   |take_book -> REQUEST_DESK
                   |take_book -> RETURN_DESK)

RETURN_DESK = (return_book -> READER
              |return_book -> REQUEST_DESK
              |return_book -> RETURN_DESK)

BOOK_STACK = (book_is_update_index -> book_sent->BOOK_STACK
             |return_book -> return_update_index ->
              book_to_Stack -> BOOK_STACK)

BOOK_INDEX = (book_is_update_index -> BOOK_INDEX
             |return_update_index -> BOOK_INDEX)


||LIBRARY = (READER || BOOK_STACJK || BOOK_INDEX)

Shared actions are in red.
```

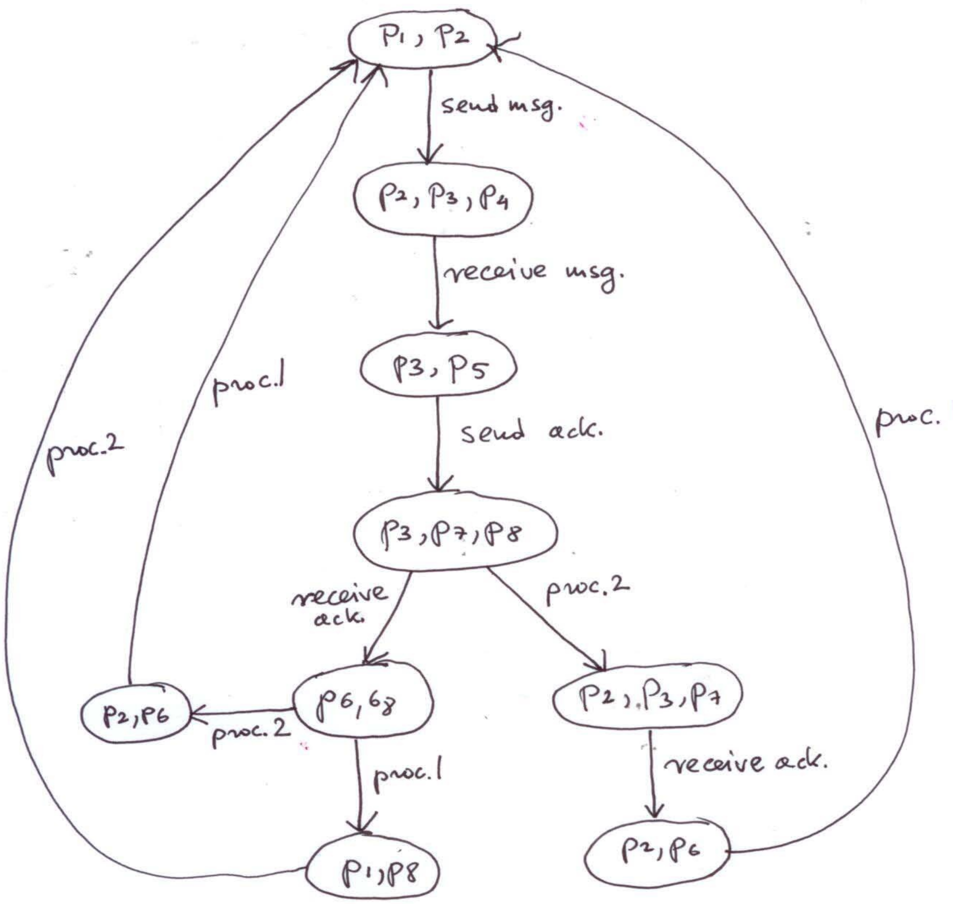
c.[5] Both solutions deal with only one reader and a generic book. In FSP we could extend it to many readers, many books and several books of with the same title, but it will make the solution rather complex and difficult to read. For Petri Nets we could use Coloured Petri Nets that will be discussed in a few weeks. Petri net solution allows some simultaneity but this is not an issue in this problem in this general setting. It will change when many users and many books are modelled. It is probably easier for most of people to find the first solution using Petri nets and then derive FSP model by modifying Petri nets model. This was the pattern use in this case. If synchronisation is non-standard, i.e. not via ‘buffers’ and/or cases modelled by ‘mutual exclusion’, finding a proper representation in FSP is often not easy.

Of course, this discussion depends heavily on the solutions provided.

10.[10] Construct *reachability graphs* (defined on page 18 of Lecture Notes 3) for both Petri nets from page 13 of Lecture Notes 3.

First rename places to have the names shorter, for example





11.[25] Consider four Labelled Transition Systems (Finite State Machines, Finite Automata) given below: P_1, Q_1, P_2 and Q_2. Tokens represent initial states. Show that:

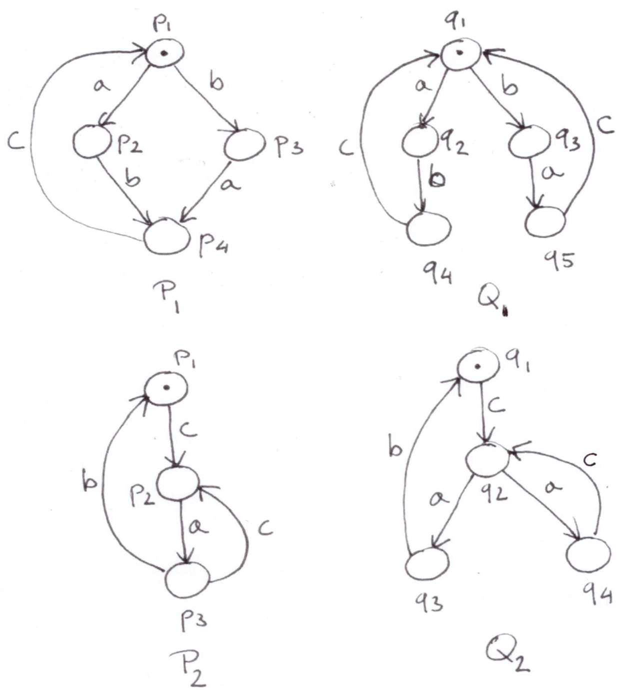
a. $P_1 \approx Q_1$, i.e. P_1 and Q_1 are *bisimilar*,

b. $P_2 \not\approx Q_2$, i.e. P_2 and Q_2 are *not bisimilar*,

c. Traces(P_1) = Traces(Q_1) = Pref(*give a proper regular expression*).

d. Traces(P_2) = Traces(Q_2) = Pref(*give a proper regular expression*).

e. Provide Finite State Processes (FSPs) equivalent to automata P_1, Q_1, P_2 and Q_2.



a.[6] Proof of $P_1 \approx Q_1$:

Clearly $p_1 \approx q_1$, as in both cases a and b can be executed. After a we are in p_2 and q_2 respectively. In both cases only b is allowed so $p_2 \approx q_2$. After b we are in p_3 and q_3 respectively. In both cases only a is allowed so $p_3 \approx q_3$. After a sequence ab we are now in p_4 and q_4. Now in both cases only c is allowed so $p_4 \approx q_4$. After a sequence ba we are now in p_4 and q_5. Again in both cases only c is allowed so $p_4 \approx q_5$. But this means $P_1 \approx Q_1$.

b.[6] Proof of $P_2 \not\approx Q_2$:

It suffices to consider the sequence ca. In P_2 it leads to the state p_3, while in Q_2 to the states q_3 and q_4. When in the state p_3, **both** b and c can be executed. However q_3 allows only b, while q_4 only c. Hence $p_3 \not\approx q_3$ and $p_3 \not\approx q_4$, i.e. $P_2 \not\approx Q_2$.

c.[4] For P_1 the set of all traces that start from *the state* p_1 and ends at *the state* p_1 is $((ab \cup ba)c)^*$, so Traces(P_1) = Pref($((ab \cup ba)c)^*$).
For Q_1 the set of all traces that start from *the state* q_1 and ends at *the state* q_1 is $(abc \cup bac)^*$, but obviously $(abc \cup bac)^* = ((ab \cup ba)c)^*$, so Traces($Q_1$) = Traces($P_1$) = Pref($((ab \cup ba)c)^*$).

d.[4] For P_2 the set of all traces that start from *the state* p_1 and ends at *the state* p_1 is $(c(ac)^*ab)^*$, so Traces(P_2) = Pref($(c(ac)^*ab)^*$).
For Q_2 the set of all traces that start from *the state* q_1 and ends at *the state* q_1 is also $(c(ac)^*ab)^*$, so Traces(Q_2) = Pref($(c(ac)^*ab)^*$).

e.[5]
```
P1   =  (a -> b -> P1p4 |b -> a -> P1p4)
P1p4 =  (c -> P1)

Q1 = ( a -> b -> c -> Q1 | b -> a -> c -> Q1)

P2   = (c -> P2p2)
P2p2 = (a -> P2p3)
P3p3 = (b - > P2 | c -> P2p2)

Q2   = (c -> Q2q2)
Q2q2 = (a -> c -> Q2q2 | a -> b -> Q2)
```