SFWRENG-3BB4 Assignment 1

Matthew Farah, ⟨farahm11@mcmaster.ca⟩, 400468251

October 7, 2025

Note that all code Java and LTS code is provided at the end in an appendix

1 Consider the following simple hotel reservation system. A customer makes a request for a room. If a room is available, a confirmation is sent to the customer, otherwise, the customer is put on a reservation list. If a room has been confirmed, the customer may either use it, pay for the room, and leave with the whole transaction being archived. However, the customer may also cancel their reservation. When a customer is on the waiting list a room may become available, in which case a confirmation is sent to the customer. The customer may also give up on waiting and cancel their reservation. Model this reservation system as a Finite State Process (FSP). Note that this process always stops, so you must include a STOP process. Also, provide an appropriate Labelled Transition System (LTS) using an LTSA tool.

1.a

1.a.I FSP Description

$$\begin{aligned}
\text{REQUEST} &= (\text{room_available} \rightarrow \text{confirm} \rightarrow \text{CHOICE} \mid \text{room_unavailable} \rightarrow \text{WAITLIST}) \\
\text{CHOICE} &= (\text{use} \rightarrow \text{pay} \rightarrow \text{leave} \rightarrow \text{archive} \rightarrow \text{STOP} \mid \text{cancel} \rightarrow \text{STOP}) \\
\text{WAITLIST} &= (\text{room_available} \rightarrow \text{confirm} \rightarrow \text{CHOICE} \mid \text{room_unavailable} \rightarrow \text{WAIT}) \\
\text{WAIT} &= (\text{cancel} \rightarrow \text{STOP} \mid \textit{wait}\text{WAITLIST})
\end{aligned}$$

1.a.II LTS Representation

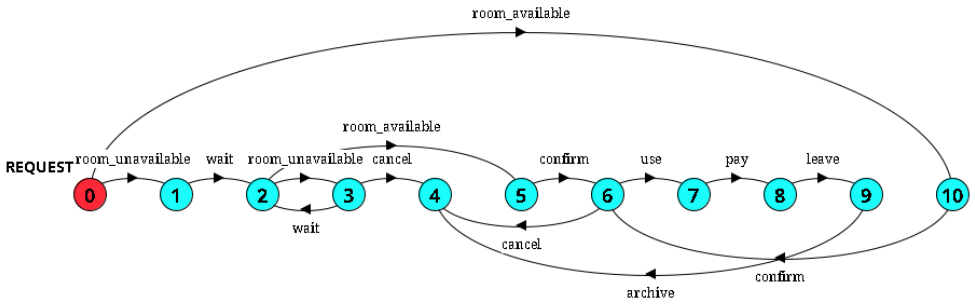

Figure 1: LTS produced by LTSA tool for Question 1

2 Consider the following three processes:

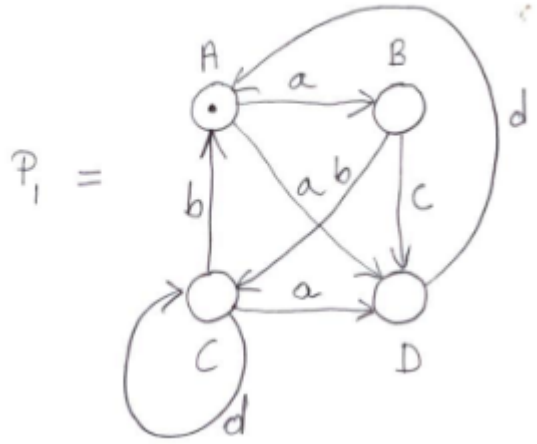

Figure 2: Question 2-a LTS 1

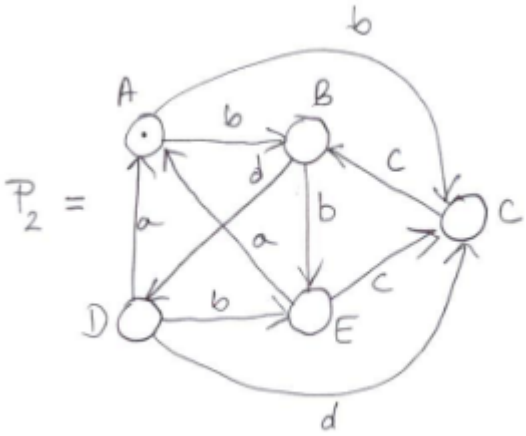

Figure 3: Question 2-a LTS 2

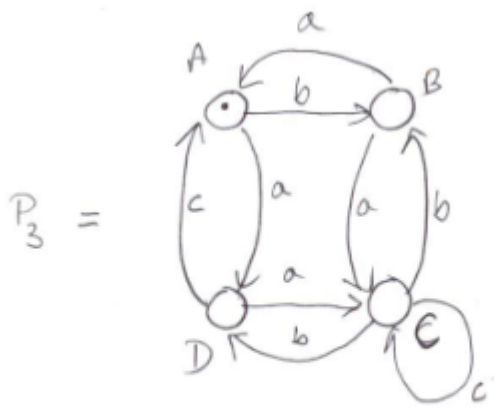

Figure 4: Question 2-a LTS 3

2.a For each of the following three processes, give an FSP description of the labelled transition graph. Dots indicate initial states.

2.a.I FSP Description for P1

$$
\begin{aligned}
&A = (a \rightarrow B \mid a \rightarrow D)\\
&B = (b \rightarrow C \mid c \rightarrow D)\\
&C = ((a \rightarrow D \mid b \rightarrow A) \mid d \rightarrow C)\\
&D = (d \rightarrow A)
\end{aligned}
$$

2.a.II FSP Description for P2

$$
\begin{aligned}
&A = (b \rightarrow B \mid b \rightarrow C)\\
&B = (b \rightarrow E \mid d \rightarrow d)\\
&C = (c \rightarrow B)\\
&D = ((a \rightarrow A \mid d \rightarrow C) \mid b \rightarrow E)\\
&E = (a \rightarrow A \mid c \rightarrow C)
\end{aligned}
$$

2.a.III FSP Description for P2

$$
\begin{aligned}
&A = (a \rightarrow D \mid b \rightarrow B)\\
&B = (a \rightarrow A \mid a \rightarrow C)\\
&C = ((b \rightarrow B \mid b \rightarrow D) \mid c \rightarrow C)\\
&D = (a \rightarrow C \mid c \rightarrow A)
\end{aligned}
$$

2.b Use an LTSA tool to transform your FSPs back into LTSs. Compare the results and discuss any differences.

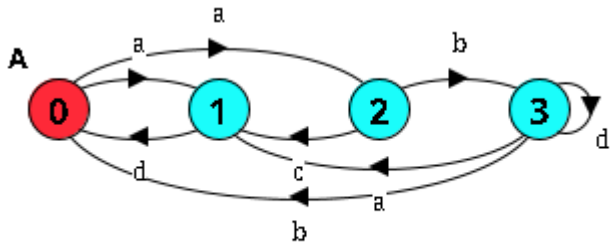

Figure 5: Question 2-b LTS 1

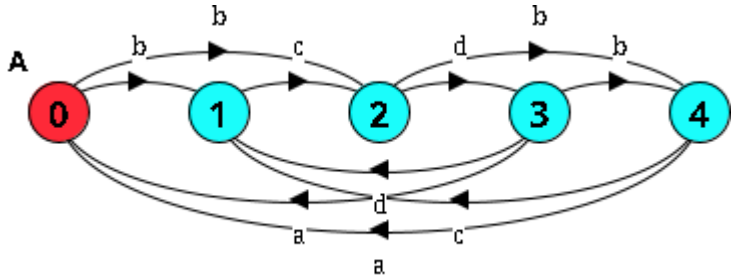

Figure 6: Question 2-b LTS 2

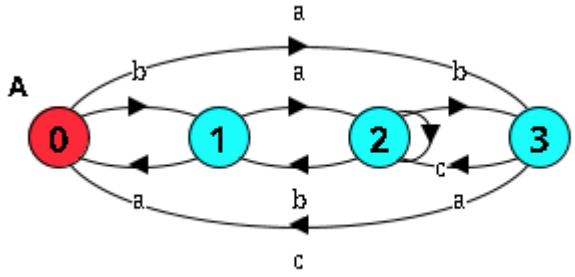

Figure 7: Question 2-b LTS 3

Each of the LTSs produced by the LTS tool are functionally equivalent to the ones given in the question, apart from the fact that the states are numbered rather than labelled alphabetically and arranged horizontally. In the code I used to produce the LTSs, the only difference of note compares to the FSPs I derived was that I had to unnest choices whenever I nested them to get it to compile.

3 A miniature portable FM radio has three controls. An on/off switch which turns the device on and off. Tuning is controlled by two buttons: Scan and reset. When the device is turned on or reset is pressed, the radio is tuned to the top freqyency of the FM band (108MHz). When scan is pressed, the radio scans towards the bottom of the band (88MHz), It stops 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, it starts to scan from the frequency of that station towards the bottom. Similarly, when reset is pressed, the receiver tunes to the top. Model the radio as an FSP (RADIO) and provide an appropriate LTS. (Hint: The alphabet of RADIO is {on, off, reset, lock, end}).

3.a

3.a.I FSP Description

$$
\begin{aligned}
\text{RADIO} &= OFF \\
\text{OFF} &= (\text{on} \rightarrow \text{TOP}) \\
\text{TOP} &= (\text{scan} \rightarrow \text{SCAN} \mid \text{reset} \rightarrow \text{TOP} \mid \text{lock} \rightarrow \text{ON} \mid \text{off} \rightarrow \text{OFF}) \\
\text{ON} &= (\text{scan} \rightarrow \text{SCAN} \mid \text{reset} \rightarrow \text{ON} \mid \text{off} \rightarrow \text{OFF} \mid \text{lock} \rightarrow \text{ON}) \\
\text{SCAN} &= (\text{lock} \rightarrow \text{ON} \mid \text{end} \rightarrow \text{BOTTOM} \mid \text{reset} \rightarrow \text{TOP} \mid \text{off} \rightarrow \text{OFF}) \\
\text{BOTTOM} &= (\text{scan} \rightarrow \text{SCAN} \mid \text{reset} \rightarrow \text{TOP} \mid \text{off} \rightarrow \text{OFF})
\end{aligned}
$$

3.a.II LTS Representation

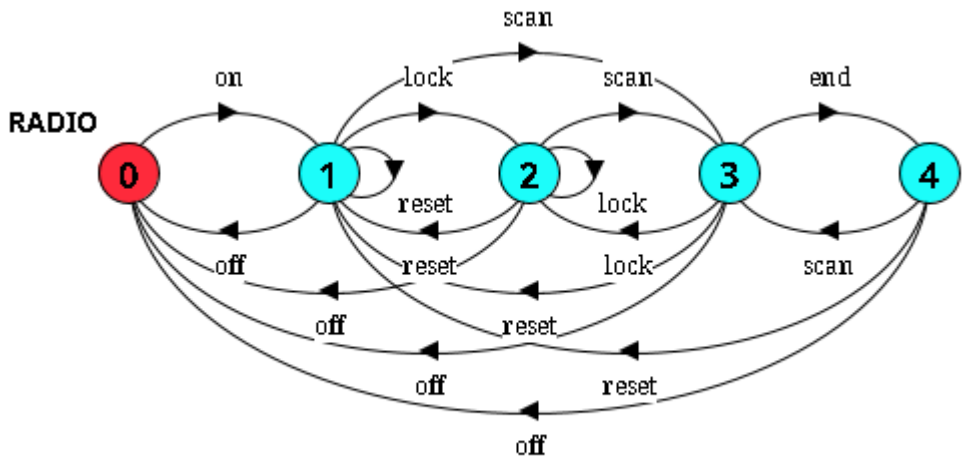

Figure 8: Question 3 LTS Generated by LTSA

4 Program the radio in Question 3 in Java, complete with a graphic display (if possible)

Code provided in appendix

5 A drinks Machine machine charges 15¢ for a can of Sugarola, 20¢ for a can of SugarolaDiet, and 25¢ for a can of SugarolaSuperDiet. The machine accepts 5¢, 10¢, and 25¢ denomination coins and provides change. Model the machine as an FSP process, DRINKS.

$$
\begin{aligned}
\text{const SUGAROLA} &= 15 \\
\text{const DIET} &= 20 \\
\text{const SUPERDIET} &= 25 \\
\text{range CREDIT} &= 0..25 \\
\text{DRINKS} &= \text{CREDIT}[0], \\
\text{CREDIT[c:CREDIT]} &= (\text{when}(c+5 \leq 25)\text{nickel} \rightarrow \text{CREDIT[c+5]} \\
&\quad |\text{when}(c+10 \leq 25)\text{dime} \rightarrow \text{CREDIT[c+10]} \\
&\quad |\text{when}(c \geq \text{SUGAROLA})\text{sugarola} \rightarrow \text{CHANGE[c-SUGAROLA]} \\
&\quad |\text{when}(c \geq \text{DIET})\text{diet} \rightarrow \text{CHANGE[c-DIET]} \\
&\quad |\text{when}(c \geq \text{SUPERDIET})\text{superdiet} \rightarrow \text{CHANGE[c-SUPERDIET]}), \\
\text{CHANGE[r:CREDIT]} &= (\text{when}(r \geq 10)\text{change10} \rightarrow \text{CHANGE[r-10]} \\
&\quad |\text{when}(r \geq 5)\text{change5} \rightarrow \text{CHANGE[r-5]} \\
&\quad |\text{CREDIT}[0].
\end{aligned}
$$

6 Consider the following set of FSPs

$$
\begin{aligned}
\text{A} &= (((\text{a} \rightarrow (\text{b} \rightarrow \text{A})) \mid (\text{c} \rightarrow (\text{a} \rightarrow \text{C} \mid \text{c} \rightarrow \text{B})) \mid (\text{c} \rightarrow \text{C}))) \\
\text{B} &= (\text{b} \rightarrow (\text{a} \rightarrow \text{B} \mid \text{c} \rightarrow (\text{a} \rightarrow \text{A} \mid \text{b} \rightarrow \text{B}))) \\
\text{C} &= ((\text{a} \rightarrow (\text{b} \rightarrow (\text{c} \rightarrow \text{B}))) \mid (\text{a} \rightarrow \text{C}))
\end{aligned}
$$

6.a Construct an equivalent LTS using the rules from page 16 of lecture notes 2

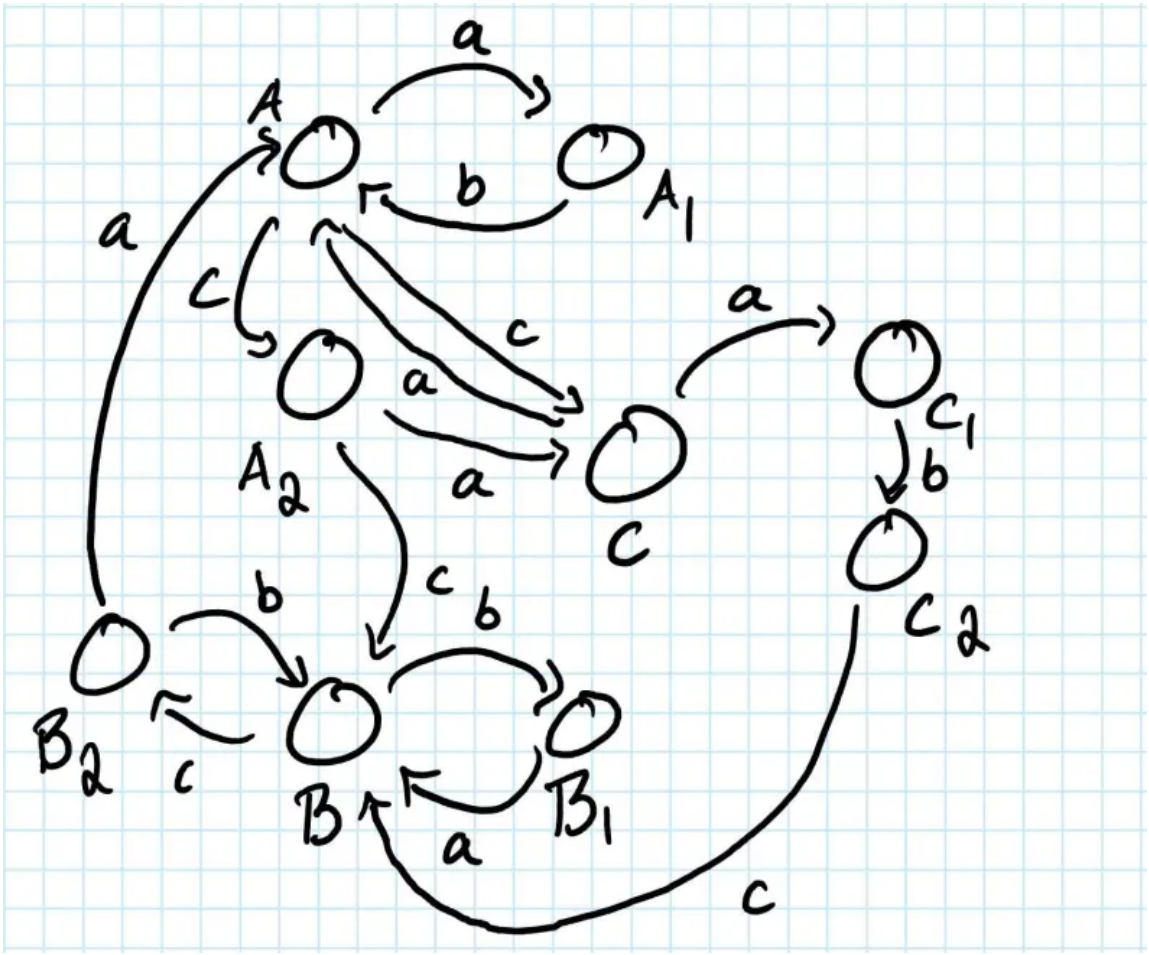

Figure 9: Question 6-a LTS made by me

6.b Use an LTSA tool to produce an appropriate LTS. If it differs from yours, analyse and explain those differences.

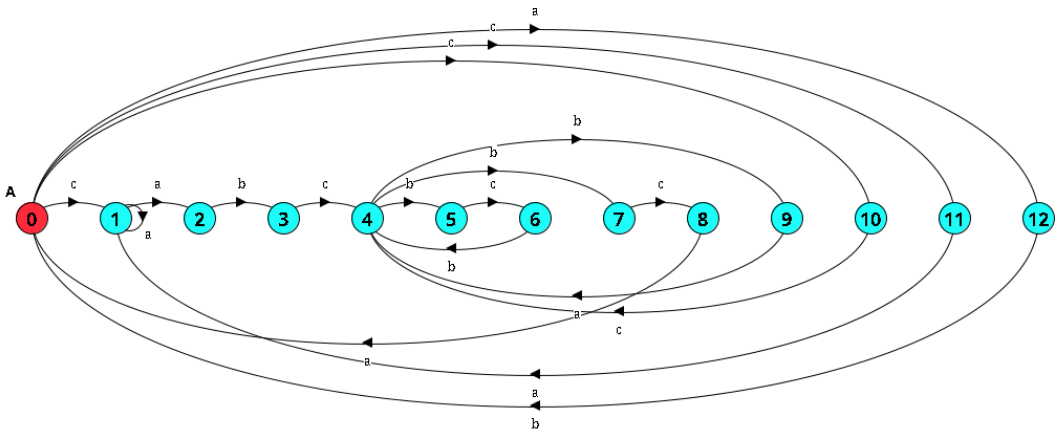

Figure 10: Question 6-b LTS made by LTSA

The solution I produced does differ from the one produced by the LTSA tool in the sense that the LTSA tool used more states to represent the behaviour of the FSP (13) than I did (9). However, both LTSs are functionally equivalent. I suspect the reason why the LTSA tool produced more states is because it chose to always create a new state every time a there existed a nested transition, whereas I chose to reuse states where ever possible to produce a slightly more concise, although equivalent, LTS.

7 Consider the following two processes, ||S1 and S2:

$$\begin{aligned} \mathrm{P} &= (\mathrm{a} \rightarrow \mathrm{b} \rightarrow \mathrm{d} \rightarrow P) \\ \mathrm{Q} &= (\mathrm{c} \rightarrow \mathrm{b} \rightarrow \mathrm{e} \rightarrow Q) \\ ||\mathrm{S1} &= (\mathrm{P} \,||\, \mathrm{Q}) \end{aligned}$$

7.a Show that the processes ||S1 and S2 generate the same LTSs. I.e. LTS(||S1) = LTS(S2) (or equivalently, they exhibit the same behaviour)

Using an LTSA tool, we can see that ||S1 and S2 generate LTSs which produce the exact same traces. Thus, we can see that their behaviour must be equivalent. You can see the generated LTSs below:

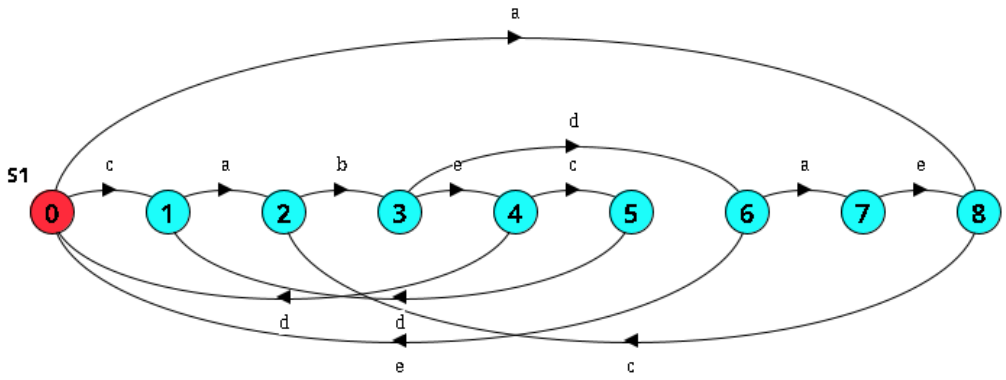

Figure 11: LTS for ||S1 for Question 7

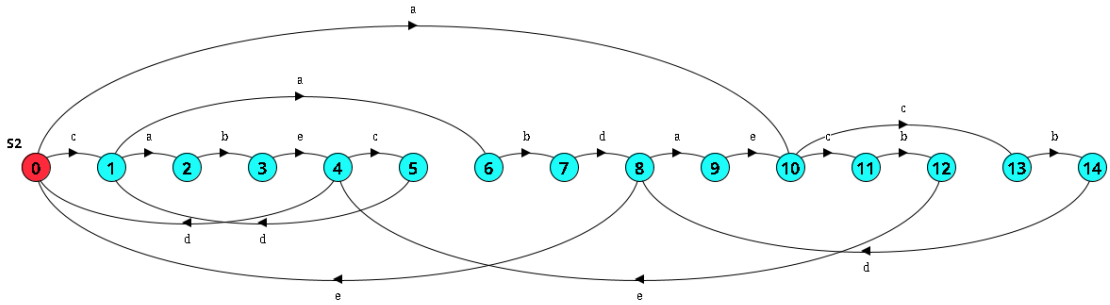

Figure 12: LTS for S2 for Question 7

7.b Using a method presented on page 17 of Lecture Notes 3 and pages 10-11 of Lecture Notes 4, transform the processes ||S1 and S2 into appropriate Petri nets. Are these nets identical? Explain the differences. Which one allows simultaneity?

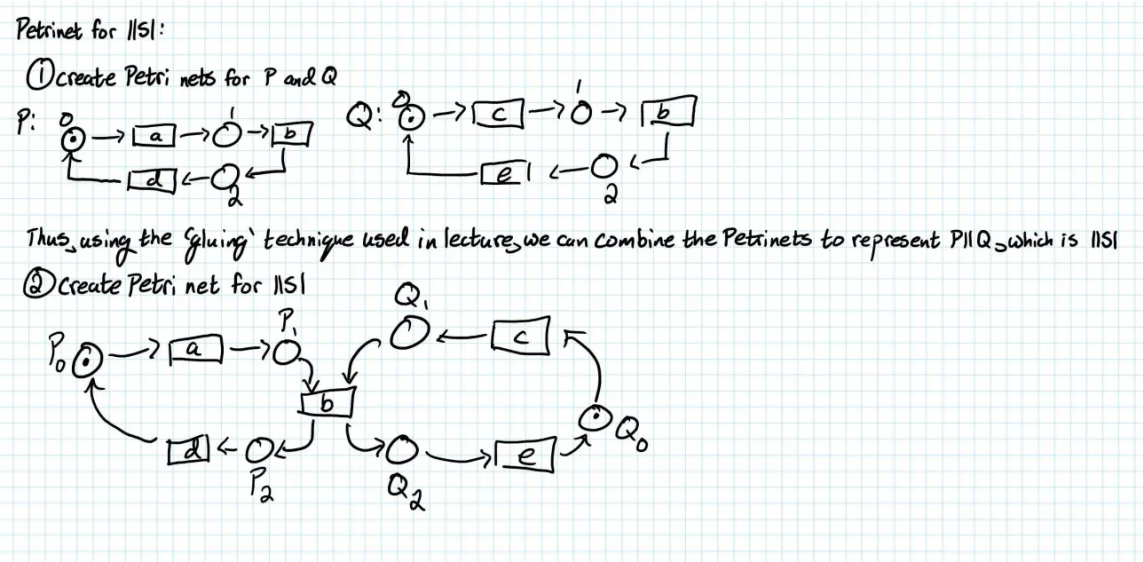

Figure 13: Petri net for ||S1 for Question 7

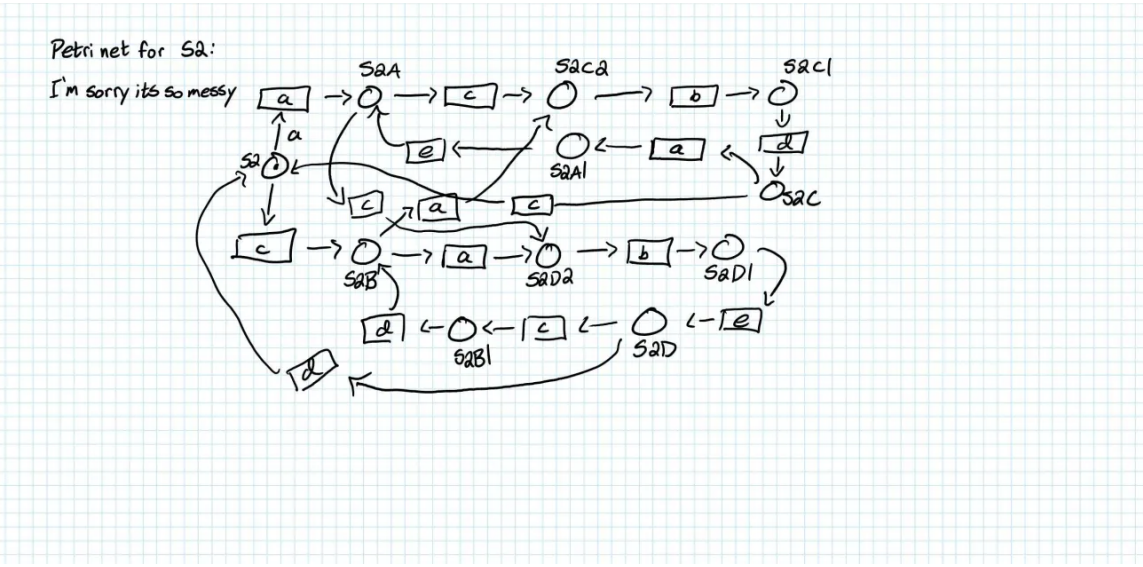

Figure 14: Petri net for S2 for Question 7

The nets are clearly not identical. Superficially, the net for ||S1 is much simpler with far fewer places and transitions than S2, however, we can also notice that the net for ||S1 contains two, rather than a single, initial token, and has a transition (the b transition) which is shared by different places, whereas S2 does not. Due to the fact ||S1 has multiple tokens, only ||S1 allows is capable of modelling simultaneity.

8 Model the Petri net N_1 (given below) as the composition of FSP processes

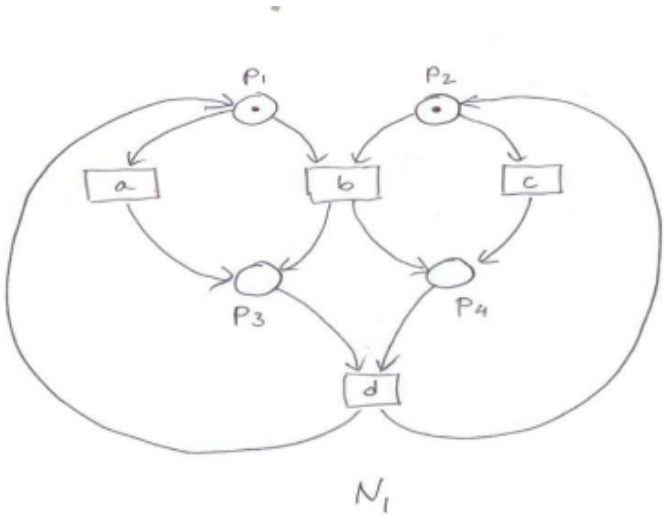

Figure 15: Petri net for Question 8

We can model N_1 with the following FSP:

$$
\begin{aligned}
P_1 &= (\mathrm{a} \to \mathrm{d}P_1 \mid \mathrm{b} \to \mathrm{d} \to P_1) \\
P_2 &= (\mathrm{c} \to \mathrm{d}P_2 \mid \mathrm{b} \to \mathrm{d} \to P_2) \\
||N_1 &= (P_1||P_2)
\end{aligned}
$$

9 Model the system from page 10 of lecture notes 3 as the composition of FSP processes. The entities represented by places in the Petri net model must be represented as actions/transitions in the FSP model

The Petri-net in the slide is given by:

Modeling Mutual Exclusion

- Two computers, one printer/data base, etc.
- Synchronization is added.

Example I

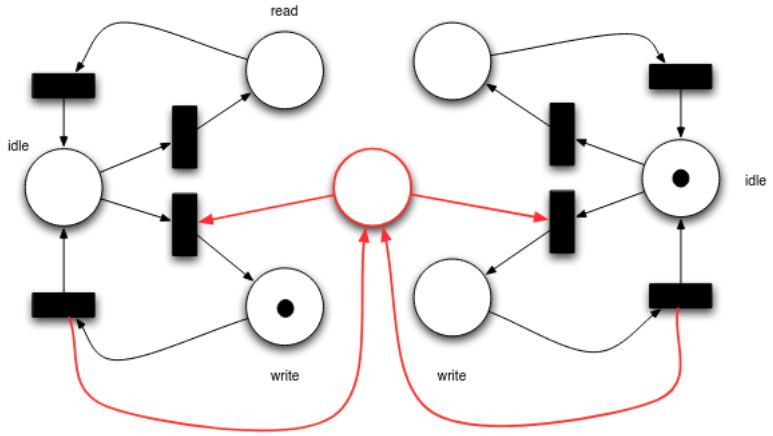

Figure 16: Petri net for Question 9

We can describe this system (denoted SYSTEM) by the parallel composition of the processes LOCK, EXECUTE, and PAUSE. The FSP for this system is given below:

$$
\begin{aligned}
\text{LOCK} &= (\text{acquire} \rightarrow \text{release} \rightarrow \text{LOCK}) \\
\text{EXECUTE} &= (\text{read} \rightarrow \text{PAUSE} \mid \text{acquire} \rightarrow \text{write} \rightarrow \text{release} \rightarrow \text{PAUSE}) \\
\text{PAUSE} &= (\text{idle} \rightarrow \text{EXECUTION}) \\
||\text{SYSTEM} &= (\text{LOCK}||\text{EXECUTE}||\text{PAUSE})
\end{aligned}
$$

10 A roller coaster control system only permits its car to depart when it's full. Passengers arriving at the departure platform are registered with the roller-coaster controller by a turnstile. The controller signals the car to depart when there are enough passengers on the platform to fill the car to its maximum capacity of M passengers. Ignore the synchronization detail of passengers embarking from the platform and car departure. The roller-coaster consists of three processes: TURNSTILE, CONTROL, and CAR. TURNSTILE and CONTROL interact by the shared action 'passenger' indicating the arrival of a passenger. CONTROL and CAR interact by the shared action 'depart' which signals the car to depart. Provide an FSP description of each process and the overall composition.

Since we're not given what the actual maximum capacity of the car is, I'm referencing it ambiguously in the FSP. In an actual implementation, all we would need to do is just substitute *some_number* with an integer.

$$
\begin{aligned}
\text{MAXIMUM\_CAPACITY} &= \textit{some\_number} \\
\text{TURNSTILE} &= (\text{passenger} \rightarrow \text{TURNSTILE}) \\
\text{CAR} &= (\text{depart} \rightarrow \text{CAR}) \\
\text{CONTROL} &= \text{CONTROL}[0] \\
\text{CONTROL}[i : 0..\text{MAXIMUM\_CAPACITY}] &= (\text{when } i < \text{MAXIMUM\_CAPACITY}) \text{ passenger} \rightarrow \text{CONTROL}[i+1] \\
&\quad | \ (\text{when } i = \text{MAXIMUM\_CAPACITY}) \text{ depart} \rightarrow \text{CONTROL}[0] \\
||\text{RIDE} &= (\text{TURNSTILE} \ || \ \text{CONTROL} \ || \ \text{CAR})
\end{aligned}
$$

11 Construct a reachability graph (defined on page 18 of lecture notes 3) for the Petri net given in Question 8.

From Question 8, we were able to determine that the Petri net can be described by the FSP:

$$
\begin{aligned}
P_1 &= (\text{a} \rightarrow \text{d}P_1 \mid \text{b} \rightarrow \text{d} \rightarrow P_1) \\
P_2 &= (\text{c} \rightarrow \text{d}P_2 \mid \text{b} \rightarrow \text{d} \rightarrow P_2) \\
||N_1 &= (P_1||P_2)
\end{aligned}
$$

Drawing this out, we can see that P_1 and P_2 are given by the LTSs:

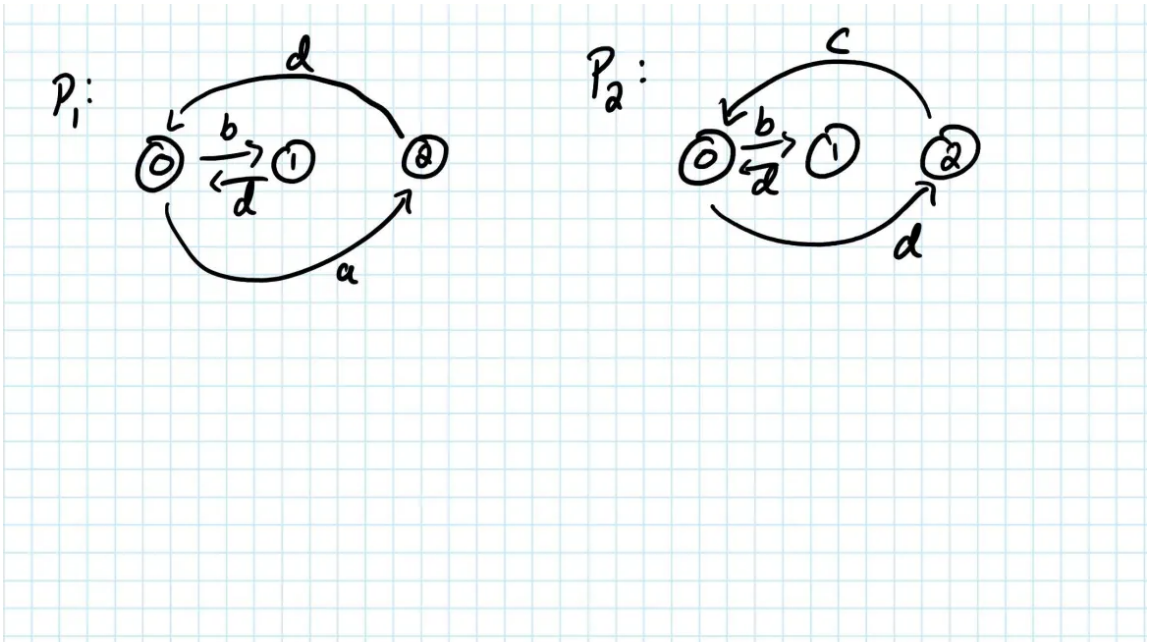

Figure 17: LTSs for Question 9

Using this, we can take the cross product of the states in each LTS to arrive at the reachability graph for N_1:

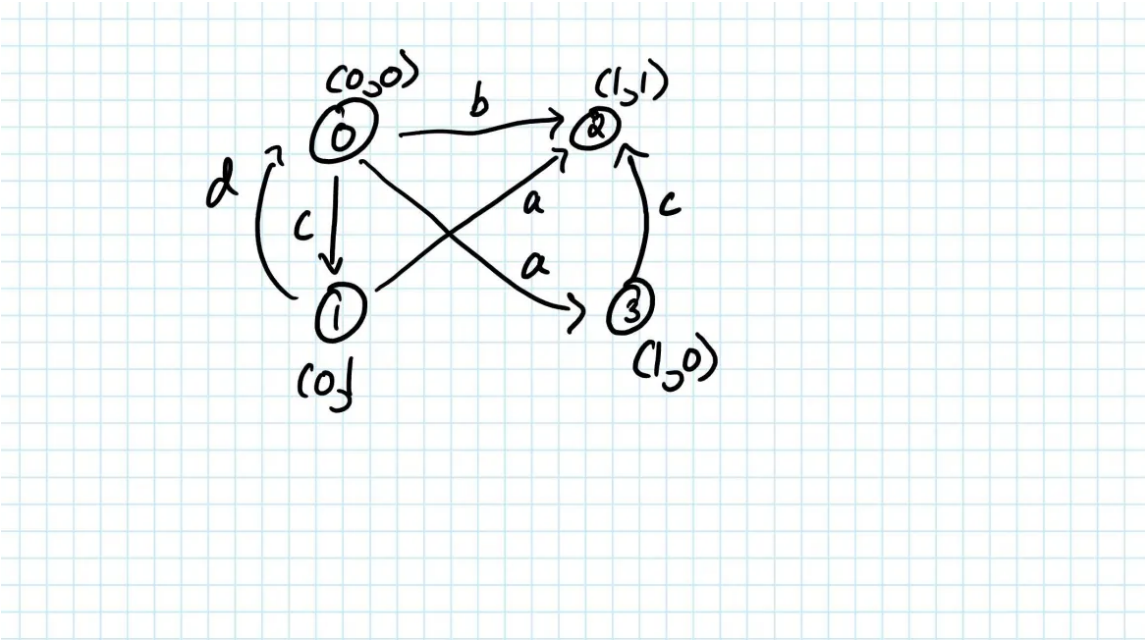

Figure 18: Reachability graph for Question 9

12 Consider the three LTSs given below, P_1, P_2, and P_3 (where tokens represent initial states):

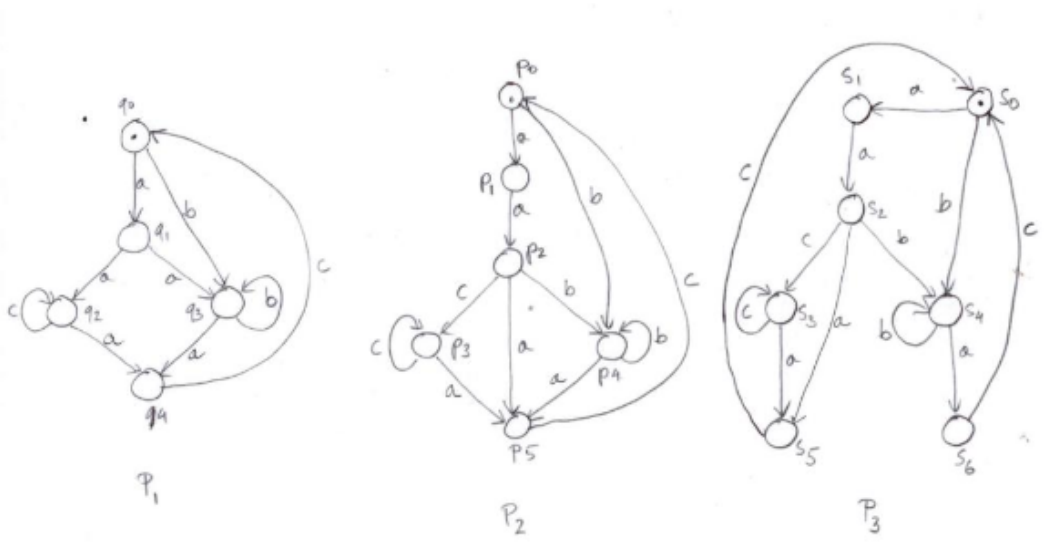

Figure 19: LTSs for Question 12

12.a Show that $P_2 \approx P_3$ (P_2 and P_3 are bisimilar)

By definition of bisimilarity, we know that P_1 and P_3 are bisimilar if all states in P_2 reachable by some trace from the initial state is bisimilar to all states reachable by the same trace in P_3. Using this definition, we can see that:

- $p_0 \approx s_0$ since they are both initial states (reachable by the zero-length string) and can both execute the action 'a'.
- $p_1 \approx s_1$ since they are both reachable by the trace 'a' and both can execute the 'a' action.
- $p_2 \approx s_2$ since both p_2 and s_2 are reachable by the trace 'aa' and both can execute the 'a', 'b', and 'c' actions.
- $p_3 \approx s_3$ since p_3 and s_3 are both reachable by the trace 'aac' and both can execute the 'a' and 'c' actions.
- $p_4 \approx s_4$ since both are reachable by the traces 'aab' and 'b' and both can execute the 'a' and 'b' actions.
- $p_5 \approx s_4$ and s_6. We know that p_5 is reachable by the traces 'aaa', 'aacc*a', 'bb*a', and 'aaa'. As for s_5 and s_6, we know s_5 is reachable by 'aacc*a' and 'aaa' and s_6 is reachable by the traces 'bb*a' and 'aab*a'. Thus, all traces from the initial state that reach p_5 also reach either s_5 or s_6. Now since, p_5 can execute actions 'c', and s_5 and s_6 can also execute 'c', the same set of actions can be executed for the same traces, this making p_5 bisimilar to both s_5 and s_6.

Therefore, since all states in P_2 are bisimilar to all states reachable by the same trace in P_3, P_2 and P_3 are bisimilar and thus $P_2 \approx P_3$.

12.b Show that $P_1 \not\approx P_2$ (P_1 and P_2 are NOT bisimilar)

We can prove that P_1 and P_2 are not bisimilar by simply illustrating a single example of a trace in P_1 which leads to a state in P_1 which isn't bisimilar to any state in P_2 with the same trace.

Consider the trace 'a' in P_1. This leads us to a state q_1. q_1 is capable of non-deterministically executing the action a, leading to two different states in P_1 (q_2 and q_3). The same trace, 'a', leads us to state p_1 in P_2, which is only capable of deterministically executing a single 'a' transition. Therefore, since two states reachable by the same trace ('a') are not bisimilar, then P_1 and P_2 are not bisimilar and thus $P_1 \not\approx P_2$.

12.c Show that $P_1 \not\approx P_3$ (P_1 and P_3 are NOT bisimilar)

We can prove that P_1 and P_3 are not bisimilar by the same methedology as earlier.

Consider once again the trace 'a' in P_1. This takes us to the state q_1, which as stated before, can non-deterministically execute action 'a' and lead to states q_2 and q_3 in P_1. The trace 'a' in P_3 leads to state s_1 which only has a single, deterministic 'a' transition leading to a single other state, s_2. Therefore, these two states, which are reachable by the same trace, are not bisimilar, P_1 and P_3 are not bisimilar and thus $P_1 \not\approx P_3$.

12.d Show that Traces(P_1) = Traces(P_2) = Traces(P_3) = Pref(*Some regular expression*)

12.d.I Traces(P_1)

To find all the traces of P_1 we must find the set of all traces which return us to the initial state. The trace 'a' takes us to the state q_1 from q_0 (our initial state). From q_1, we can non-deterministically take 'a' to either q_2 or q_3 (making our current trace 'aa'). At q_2, we can execute c an indefinite number of times while remaining at q_2 or take an 'a' transition to q_4. At q_3, we can similarly execute 'b' an indefinite number of times or take an 'a' transition to end up at state q_4. Thus, q_4 is reachable by the traces (aab*a—aac*a). q_4 connects us back to our initial state with the 'c' transition and therefore (aa(b*—c*)ac)* is a set of traces of P_1. However, there also exists a 'b' transition connecting q_0 to q_3, and using our prior knowledge of q_3, we can ascertain that the set of all traces of P_1 is given by Pref(((aa(c*—b*)—(bb*))ac)*).

12.d.II Traces(P_2)

We use the same strategy as before to ascertain the traces of P_2. The trace 'a' takes us from the initial state p_0 to p_1 and taking 'a' again leads us to p_2, giving us out current trace of 'aa'. From p_2, we can take either an 'a', 'b', or 'c' transition to the states p_5, p_4, or p_3 respectively. Taking the 'b' transition to p_4, we can take the 'b' transition an indefinite number of times, remaining at p_4 or take an 'a' transition to p_5. Similarly, from p_3 we can take a c transition an indefinite number of times befre transitioning to p_5 by executing 'a'. Thus the traces 'aaa', 'aacc*a', and 'aabb*a' all take us to p_5. Since p_5 takes us back to the initial state with a 'c' transition, we know (aa(cc*—bb*)ac)* are all traces of our system. However, there also exists a 'b' transition frm p_0 to p_4, and using our knowledge of p_4, we can therefore conclude that the set of all traces of P_2 can be given by Pref(((aa(c*—b*)—(bb*))ac)*).

12.d.III Traces(P_3)

Using the same process, we see that 'a' takes us from s_0 to s_1, and another 'a' moves us from s_1 to s_2. From s_2, we can take an 'a', 'b', or 'c' transition to states s_5, s_4, or s_3 respectively. At s_3, we can take a 'c' transition an indefinite number of times, remaining at s_3, before taking an 'a' transition to s_5. Likewise, at s_4, we can take the 'b' transition an indefinite number of times before taking an 'a' transition to s_6. From both s_5 and s_6, we can take a 'c' transition to return back to our initial state s_0. However, we can also take a 'b' transition directly from s_0 to s_4, and using our prior description of s_4, we can conclude that the traces of P_3 is given by Pref(((aa(c*—b*)—(bb*))ac)*).

Thus, the traces of P_1, P_2, and P_3 are all equivalent and given by Pref(((aa(c*—b*)—(bb*))ac)*).

13 Appendix

13.a Question 1 LTSA Code

```
REQUEST = (room_available -> confirm -> CHOICE | room_unavailable ->
    wait -> WAITLIST),
CHOICE = (use -> pay -> leave -> archive -> STOP | cancel -> STOP),
WAITLIST = (room_available -> confirm -> CHOICE | room_unavailable ->
    WAIT),
WAIT = (cancel -> STOP | wait -> WAITLIST).
```

13.b Question 2 LTSA Code

13.b.I

```
A = (a -> B | a -> D),
B = (b -> C | c -> D),
C = (a -> D | b -> A | d -> C),
D = (d -> A).
```

13.b.II

```
A = (b -> B | b -> C),
B = (b -> E | d -> D),
C = (c -> B),
D = (a -> A | d -> C | b -> E),
E = (a -> A | c -> C).
```

13.b.III

```
A = (a -> D | b -> B),
B = (a -> A | a -> C),
C = (b -> B | b -> D | c -> C),
D = (a -> C | c -> A).
```

13.c Question 3 LTSA Code

```
RADIO = OFF,

OFF = (on -> ON_TOP),

ON_TOP = (scan -> SCANNING
         | reset -> ON_TOP
         | off -> OFF
         | lock -> ON_STATION),

ON_STATION = (scan -> SCANNING
             | reset -> ON_TOP
             | off -> OFF
             | lock -> ON_STATION),

SCANNING = (lock -> ON_STATION
           | end -> ON_BOTTOM
           | reset -> ON_TOP
           | off -> OFF),

ON_BOTTOM = (scan -> SCANNING
            | reset -> ON_TOP
            | off -> OFF).
```

13.d Question 4 Java Code

```
import javax.swing.*;
import java.awt.*;

public class RadioSimulator extends JFrame {
    enum State { OFF, ON_TOP, ON_STATION, SCANNING, ON_BOTTOM }
    private State state = State.OFF;
    private JLabel display;
    private int frequency = 108;
    private Timer scanTimer;

    public RadioSimulator() {
        super("Mini FM Radio");
        display = new JLabel("State: OFF", SwingConstants.CENTER);
        display.setFont(new Font("Arial", Font.BOLD, 20));
        add(display, BorderLayout.CENTER);

        JPanel buttons = new JPanel(new GridLayout(1, 5));
        JButton onBtn = new JButton("On");
        JButton offBtn = new JButton("Off");
```

```
JButton scanBtn = new JButton("Scan");
JButton resetBtn = new JButton("Reset");
JButton lockBtn = new JButton("Lock");
buttons.add(onBtn);
buttons.add(offBtn);
buttons.add(scanBtn);
buttons.add(resetBtn);
buttons.add(lockBtn);
add(buttons, BorderLayout.SOUTH);

scanTimer = new Timer(300, e -> {
    if (state == State.SCANNING) {
        if (frequency > 88) {
            frequency--;
        } else {
            state = State.ON_BOTTOM;
            scanTimer.stop();
        }
        update();
    }
});

onBtn.addActionListener(e -> {
    if (state == State.OFF) {
        state = State.ON_TOP;
        frequency = 108;
    }
    update();
});
offBtn.addActionListener(e -> {
    state = State.OFF;
    scanTimer.stop();
    update();
});
scanBtn.addActionListener(e -> {
    if (state == State.ON_TOP || state == State.ON_STATION ||
        state == State.ON_BOTTOM) {
        state = State.SCANNING;
        scanTimer.start();
    }
    update();
});
resetBtn.addActionListener(e -> {
    if (state != State.OFF) {
        state = State.ON_TOP;
        frequency = 108;
```

```
                scanTimer.stop();
            }
            update();
        });
        lockBtn.addActionListener(e -> {
            if (state == State.SCANNING) {
                state = State.ON_STATION;
                scanTimer.stop();
            }
            update();
        });

        setSize(400, 200);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setVisible(true);
    }

    private void update() {
        if (state == State.OFF) display.setText("State: OFF");
        else display.setText("State: " + state + "  Frequency: " +
            frequency + " MHz");
    }

    public static void main(String[] args) {
        SwingUtilities.invokeLater(RadioSimulator::new);
    }
}
```

13.e Question 6 LTSA Code

```
A = (a-> b -> A | c -> a -> C | c -> c -> B | c -> C),
B = (b -> a -> B | b -> c -> a -> A | b -> c -> b -> B),
C = (a -> b -> c -> B | a -> C).
```

13.f Question 7 LTSA Code

13.f.I

```
P = ( a -> b -> d -> P).
Q = ( c -> b -> e -> Q ).
||S1 = (P||Q).
```

13.f.II

```
S2 = ( a -> S2A | c -> S2B ) ,
S2A = ( c -> b -> d -> S2C | c -> b -> e -> S2D ) ,
S2B = ( a -> b -> d -> S2C | a -> b -> e -> S2D ) ,
S2C = ( e -> S2 | a -> e -> S2A ) ,
S2D = ( d -> S2 | c -> d -> S2B ) .
```