SolutionSerie 3 - Safety and Synchronization

Exercise 1 (2 Points)

Answer the following questions (0.5 point each):

1. What are safety properties? How are they modeled in FSP? **Answer:**

   A safety property P is a deterministic process that assert that any trace including actions in the alphabet of P is accepted by P. It is possible define a safety property using the keyword property and defining the legal action trace that the model has to follow.

2. Is the busy-wait mutex protocol fair? Deadlock-free? Justify your answer. **Answer:**

   This protocol is fair because a process always gives priority to the other process to enter the critical section. So they, in theory, can alternately enter in the critical section. It's formally proven to be deadlock free (see the FSP in the lecture).

3. Can you ensure safety in concurrent programs without using locks? **Answer:**

   To ensure safety you need locks (of some sort) because you have to guarantee that only one thread at the same time can use shared resources. The only situation where you don't need locks is when there is no shared mutable state.

4. The Java language designers decided to implement concurrency based on monitors. What is the main reason behind this decision? **Answer:**

   The notion of monitors is similar to the notion of objects.

Exercise 2 (1 point)

Consider the following process definitions. Are S1 and S2 equivalent? Why?

```
P = (a->b->P).
Q = (c->b->Q).
||S1 = (P || Q).
S2 = (a->c->b->S2|c->a->b->S2).
```

Answer:

Yes. The only possible action traces are: "a, c, b" or "c, a, b" for both processes.

Exercise 3 (3 points)

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 operate as follows. When the radio is turned on or reset is pressed, the radio is tuned to the top frequency of the FM band. When scan is pressed, the radio scans towards the bottom of the band. It stops scanning when it locks on to a station or it reaches bottom (end). If the radio is currently tuned to a station and scan is pressed then it starts to scan from the frequency of that station towards bottom. Similarly, when reset is pressed the receiver tunes to top.

Using the alphabet $\{\text{on}, \text{off}, \text{scan}, \text{reset}, \text{lock}, \text{end}\}$ model the FM radio as an FSP process called RADIO and generate the corresponding LTS using the LTSA tool.

```
RADIO = OFF,
OFF = (on-> ON),
ON = (off->OFF |
      reset -> TOP |
      scan -> SCANNING),
BOTTOM = (off -> OFF | scan -> SCANNING),
TOP = (off -> OFF | scan -> SCANNING),
SCANNING = (lock -> LOCKED | end -> BOTTOM | reset -> TOP),
LOCKED = (off -> OFF | scan -> SCANNING).
```

Exercise 4 (4 points)

Consider the following problem specifications:

1. We have a stack that has two slots.
2. The stack operations *pop* and *push* should be atomic. In other words, the stack is locked while being *pop*ped or *push*ed.
3. The input space is 0..1 (You can only *push* 0s and 1s).
4. There is one producer process that *push*es 0s and 1s repeatedly.
5. There is one consumer that *pop*s the top of the stack.
6. Throw an error for *push*ing to a full stack or *pop*ping from an empty stack.

Write an FSP description for this problem and verify your model using LTSA. The solution where stack error is a recognized and modeled action.

```
range BIT = 0..1

LOCK = ( lock -> unlock -> LOCK ).

STACK = S0,
S0 = ( push[v:BIT] -> ok -> S1[v]
         | pop[u:BIT] -> error -> S0),
S1[v:BIT] = ( pop[v] -> ok -> S0
                    | push[u:BIT] -> ok -> S2[v][u] ),
S2[v:BIT][u:BIT] = ( pop[u] -> ok -> S1[v]
                    | push[x:BIT] -> error -> S2[v][u]).

set Ops = { pop[BIT], push[BIT], error, ok }

PUSH = ( lock -> push[v:BIT] ->
          ( ok -> unlock -> PUSH
          | error -> unlock -> PUSH ) ) +Ops.

POP = ( lock -> pop[v:BIT]->
          ( ok -> unlock -> POP
          | error -> unlock -> POP ) ) +Ops.

||Stack2 = ({a,b}::STACK || a:PUSH || b:POP || {a,b}::LOCK).
```

Another solution (not perfect but acceptable) where the stack error is a model ERROR.

```
range InputR = 0..1
const N = 2
range SizeR = 0..N
set StackAlpha = { push[InputR], pop[InputR] }

STACK = STACK[0][0][0],
STACK[size:SizeR][top:InputR][bottom:InputR] = (
when(size<N) push[x:InputR] -> STACK[size+1][x][top]
|when(size>0) pop[top] -> STACK[size-1][bottom][0]
|when(size==0) pop[top] -> ERROR
|when(size==N) push[x:InputR] ->ERROR
).

PRODUCER = (push[x:InputR]-> PRODUCER)+StackAlpha.

CONSUMER = (pop[x:InputR]->CONSUMER) +StackAlpha.

||PROCESS = ({p,c}::STACK||p:PRODUCER||c:CONSUMER).
```