Prof. Dr. Oscar Nierstrasz
Pascal Gadient, Leonel Merino

# Solution
# Series 03 — 04.10.2017 – v1.0
# Safety & Synchronization

## Exercise 1 (2 Points)

Answer the following questions:

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

   *A safety property P consists of (textual or mathematical) definitions describing properties of a model. These definitions can be implemented within a deterministic process. This process asserts that any trace with actions in the alphabet of P, is accepted by P (i.e. not violating any safety constraints). It is possible to define a safety property using the keyword "property" and defining the legal action trace that the model has to follow.*[1]

b. 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).*

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

   *No, to ensure safety you need some sort of locks, because you have to guarantee that only one thread at a time can access shared resources. The sole case where you don't need locks is, obviously, when there is no shared mutable state.*

d. The Java language designers decided to implement concurrency based on monitors. What is the main reason behind this decision? What other options except monitors could have been chosen? (Hint: Consider slide 28 of lecture 1) **Answer:**

   *The notion of monitors is similar to the notion of objects. The concepts of semaphores and message passing could also be used for ensuring safety and liveness, but because of their complex configurations these are not that easily adaptable to existing language specifications. For example, imagine the Java language designers relied on semaphores instead of monitors. Accordingly you would have to provide every time an additional number specifying the amount of concurrent threads that are allowed in the critical section, when you use the synchronized keyword.*

## Exercise 2 (1 point)

Consider the following process definitions. Are T1 and T2 equivalent? Why?

```
R = (a->c->R).
S = (b->c->S).
||T1 = (R || S).
T2 = (a->b->c->T2|b->a->c->T2).
```

[1] You can find an example in section 1.8 at
https://www.doc.ic.ac.uk/ltsa/eclipse/help/index.html?appendix_b___fsp_language_spec.htm

**Answer:**

*Yes. The only possible action traces are: "a, b, c" or "b, a, c" 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. **Answer:**

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

## Exercise 4 (4 points)

Consider the following problem specification:

a. We have a stack that has two slots.

b. The stack operations *pop* and *push* should be atomic. In other words, the stack is locked while being *pop*ped or *push*ed.

c. The input space is 0..1 (You can only *push* 0s and 1s).

d. There is one producer process that *push*es 0s and 1s repeatedly.

e. There is one consumer that *pop*s the top of the stack.

f. 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. **Answer:**

*In the solution below, we modeled stack errors as actions (i.e. transitions). Another, although less favorable approach, would be to represent stack errors with a common explicit "error state".*

```
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).
```