# Tutorial 5 - FSP

Note: You will need the LTSA Tool http://bit.ly/ltsatool to do some of these exercises

**Q1** . Prove that the FSP for LOCK is same as the FSP for MUTEX. For your answer you may prove either by using traces for both the FSPs or by drawing LTS diagrams for both the FSPs.

```
//THE LOCK FSP
LOCK = (acquireLock -> releaseLock -> LOCK).

//THE MUTEX FSP
//define the states of the mutex as constants
const STATUS_FREE = 1
const STATUS_TAKEN = 0
MUTEX = MLOCK[STATUS_FREE], //initially the status is free
MLOCK[status:STATUS_TAKEN..STATUS_FREE] = (
                  when (status==STATUS_TAKEN) releaseLock -> MUTEX
                  | when (status==STATUS_FREE) acquireLock -> 
MLOCK[STATUS_TAKEN]
).
```

**Q2**. Consider the following FSPs:

```
//a process to generate alphabets and write them to file
ALPHAGEN = (genA -> genB -> writeA -> writeB -> STOP).

//a process to generate numbers and write them to file
NUMBERGEN = (gen1 -> gen2 -> write1 -> write2 -> STOP).
||ALPHA_NUM = (ALPHAGEN || NUMBERGEN).
```

a) Write any four possible traces for ||ALPHA_NUM process. Your traces must show possible interleavings between ALPHAGEN and NUMBERGEN.

**Q3**. A web server and a web client are described by the following FSPs:

```
WEB_SERVER = (listen -> accept -> service -> response -> WEB_SERVER).
WEB_CLIENT = (connect -> request -> wait -> receive -> WEB_CLIENT).
```

a) Write the FSP for a composite process ||WEB_CS where the WEB_CLIENT and WEB_SERVER synchronize on accept and request actions and also on response and receive actions.
b) Change the FSP for the WEB_CLIENT so that it terminates after receive.

**Q4**. Discuss the importance of modeling concurrent systems using approaches such as FSP. What are advantages and benefits of this modeling concurrent systems compared to writing the Java code directly without modeling the system?

**Q5**. Consider the following FSP and answer the questions below:

```
T1 = (r1.acquire -> r2.acquire -> doX -> r1.release -> r2.release -> T1).
T2 = (r2.acquire -> r1.acquire -> doY -> r2.release -> r1.release -> T2).
RESOURCE = (acquire -> release -> RESOURCE).

||T1T2 = (t1: T1 || t2:T2 || {t1,t2}::r1:RESOURCE || {t1,t2}::r2:RESOURCE).
```

a) using the LTSA tool find if there are any progress violations
b) Rewrite the correct FSP such that there are no progress violations

**Q6**. Two processes P and Q are defined as follows:

```
P = (a1 -> a2 -> b1 -> c1-> P).
Q = (a1 ->b1 -> b2 -> c1-> Q).
```

a) How many traces are possible when P is composed with Q? Write the possible traces.
b) Write a composite process PQ which is composed P and Q but show only actions a1 and c1.

**Q7**. A resource allocator controls access to a single shared resource. The FSP for the resource allocator is as follows:

```
//define the states of resource as constants
const AVAILABLE = 1
const TAKEN = 0
RESOURCE_ALLOCATOR = RES_ALLOC[AVAILABLE], //initially the resource is free
RES_ALLOC[status:TAKEN..AVAILABLE] = (when (status==AVAILABLE)
                                              acquire -> RES_ALLOC[TAKEN]
                                     |when(status==TAKEN)
                                              release -> RESOURCE_ALLOCATOR
                                     ).
```

(a) Write a Java monitor – *ResourceAllocator* for the above FSP complete with synchronization and condition synchronization.
(b) Write another version of the *ResourceAllocator* in Java but this time using a single Semaphore instead of condition variables.