# Chapter 1 - exercises

This is not really meant as an exercise, but as a way for you to get a first contact with the LTSA tool, which you will be using extensively in this course.

**1.1** Start the LTSA, and type the following in the Edit window:

```
BOMB = (start -> timeout -> explode -> STOP).
```

This is a simplified model of a bomb. The timer of the bomb is started, and when it expires (action timeout) the bomb explodes. Now from the "Build" menu option, select "Parse". This lets you know if your specification contains syntax errors.

If you have no syntax errors, then select "Check – Compile". This will generate the state machine (Labelled Transition System - LTS) that corresponds to your specification. You want to see what it looks like? Select "Window – Draw". Is this what you expected?

You can also experiment with actually "animating" the model of the bomb. Select "Check - Run – Default". An animator window comes up. On the right, you have the actions that can be performed by the Bomb. The "ticked" ones are those that are eligible at the current state. Select an action that you would like the bomb to perform (by clicking in its corresponding box). Can all actions be selected? What happens when you select an eligible action? Does anything change on the displayed LTS?

**1.2** Perform all of the above steps for the following specification of a lamp:

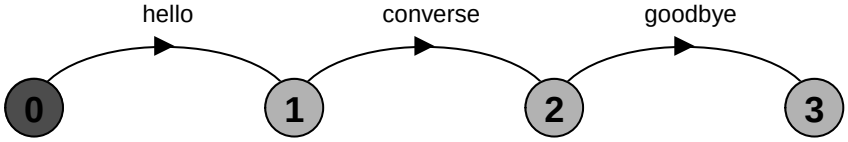
```
LAMP = (switch_on -> switch_off -> LAMP).
```

Can you see an important difference from the first model?

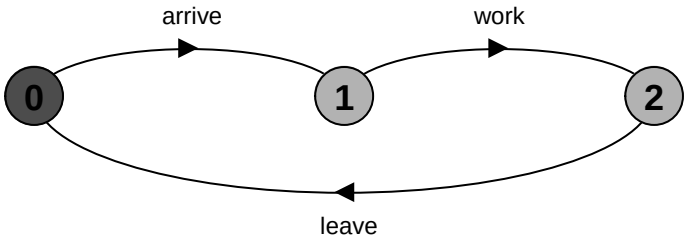
## Chapter 2 - exercises

2.1 For each of the following processes, give the Finite State Process (FSP) description of the Labeled Transition System (LTS) graph. The FSP process descriptions may be checked by generating the corresponding state machines using the analysis tool, *LTSA*.

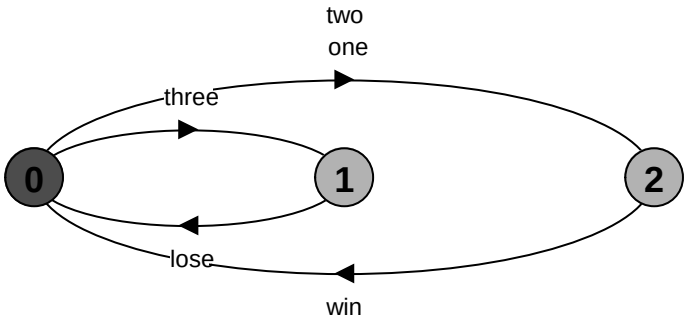
I. MEETING:



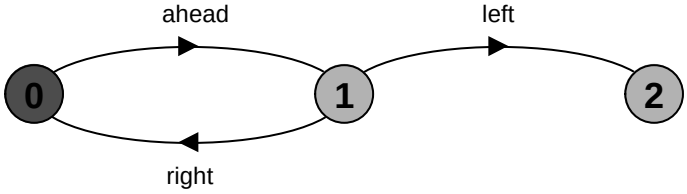

II. JOB:



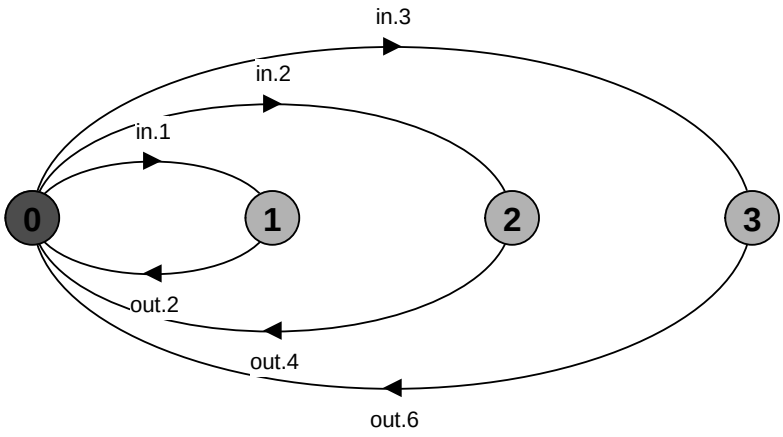

III. GAME:



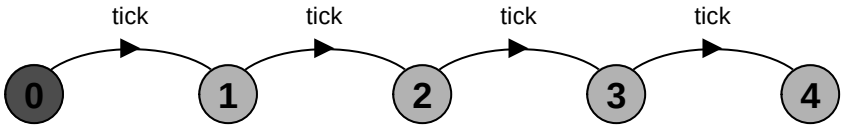

IV. MOVE:



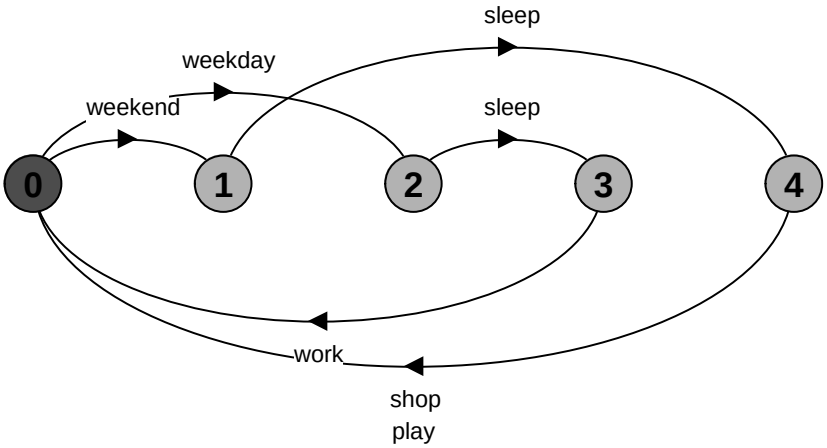

V. DOUBLE




VI. FOURTICK:




VII. PERSON:




For each of the following exercises 2.2 to 2.6, draw the state machine diagram that corresponds to your *FSP* specification and check that it can perform the required actions. The state machines may be drawn manually or generated using the analysis tool, *LTSA*. *LTSA* may also be used to animate (run) the specification to produce a trace.

2.2 A variable stores values in the range 0..*N* and supports the actions *read* and *write*. Model the variable as a process, `VARIABLE`, using *FSP*.

For N=2, check that it can perform the actions given by the trace:

write.2 → read.2 → read.2 → write.1 → write.0 → read.0

2.3 A bistable digital circuit receives a sequence of *trigger* inputs and alternately outputs 0 and 1. Model the process `BISTABLE` using *FSP*, and check that it produces the required output i.e. it should perform the actions given by the trace:

trigger → 1 → trigger → 0 → trigger → 1 → trigger → 0 …

(Hint: the alphabet of `BISTABLE` is *[0],[1],trigger* ).

2.4 A sensor measures the water *level* of a tank. The level (initially 5) is measured in units 0..9. The sensor outputs a *low* signal if the level is less than 2 and a *high* signal if the level is greater than 8 otherwise it outputs *normal*. Model the sensor as an *FSP* process, `SENSOR`.

(Hint: the alphabet of `SENSOR` is *level[0..9], high, low, normal*).

2.5 A drinks dispensing machine charges 15p for a can of Sugarola. The machine accepts coins with denominations 5p, 10p and 20p and gives change. Model the machine as an *FSP* process, `DRINKS`.

2.6 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 (108 MHz). When *scan* is pressed, the radio scans towards the bottom of the band (88 MHz). 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 (*on, off, scan, reset, lock, end*) model the FM radio as an *FSP* process, `RADIO`.

2.7 Program the radio of 2.6 in Java, complete with graphic display.

## Chapter 3 - exercises

3.1 Show that `S1` and `S2` describe the same behavior:

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

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

3.2 `ELEMENT=(up->down->ELEMENT)` accepts an `up` action and then a `down` action. Using parallel composition and process `ELEMENT` describe a model, with interface actions `up` and `down`, that can accept up to four consecutive `up` actions before a `down` action. Draw a Structure Diagram for your solution. (Hint – see TWOBUFF)

3.3 Extend the model of the client-server system `CLIENT_SERVER` such that there can be more than one client using the server.

3.4 Modify the model of the client-server system described in question 3.3 such that the call may terminate with a timeout action rather than a response from the server. What happens to the server in this situation?

3.5 A roller coaster control system only permits its car to depart when it is 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. The car then goes around the roller coaster track and then waits for another M passengers. A maximum of M passengers may occupy the platform. 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 an arrival and `CONTROL` and `CAR` interact by the shared action `depart` signaling car departure. Draw the Structure Diagram for the system and provide FSP descriptions for each process and the overall composition.

3.6 Modify the example Java program `ThreadDemo` such that it consists of three rotating displays.

# Chapter 4 - exercises

## 4.1 Recursive Locking in Java

Once a thread has acquired the lock on an object by executing a synchronized method, that method may itself call another synchronized method from the same object (directly or indirectly) without having to wait to acquire the lock again. The lock counts how many times it has been acquired by the same thread and does not allow another thread to access the object until there has been an equivalent number of releases. This locking strategy is sometimes termed *recursive* locking since it permits recursive synchronized methods. For example:

```
public synchronized void increment(int n) {
   if (n>0) {
       ++value;
       increment(n-1);
   } else return;
}
```

This is a rather unlikely recursive version of a method which increments `value` by `n`. If locking in Java was not recursive, it would cause a calling thread to block resulting in a deadlock.

Given the following declarations:

```
const N = 3
range P = 1..2  //thread identities
range C = 0..N  //counter range for lock
```

Model a Java recursive lock as the *FSP* process `RECURSIVE_LOCK` with the alphabet `{acquire[p:P],release[p:P]}`. `acquire[p]` acquires the lock for thread `p`.

## Chapter 5 - exercises

5.1 A single slot buffer may be modeled by:

```
ONEBUF = (put->get->ONEBUF).
```

Program a Java class `OneBuf` that implements this one slot buffer as a monitor.

5.2 Replace the condition synchronization in your implementation of the one slot buffer by using semaphores. Given that Java defines assignment to scalar types (with the exception of long and double) and references types to be atomic, does your revised implementation require the use of the monitor's mutual exclusion lock?

5.3 In the museum example (assessed coursework), identify which of the processes, `EAST`, `WEST`, `CONTROL` and `DIRECTOR`, should be threads and which should be monitors. Provide an implementation of the monitor(s).

5.4 FSP allows multiple processes to synchronize on a single action. A set of processes with the action *sync* in their alphabets must all perform this action before any of them can proceed. Implement a monitor called `Barrier` in Java with a method `sync` that ensures that all of *N* threads must call `sync` before any of them can proceed.

5.5 *The Savings Account Problem:* A savings account is shared by several people. Each person may deposit or withdraw funds from the account subject to the constraint that the balance of the account must never become negative. Develop a model for the problem and from the model derive a Java implementation of a monitor for the savings account.