Solution
Assignment 02 — 25.09.2019 – v1.2
Concurrency and Java

Exercise 1 (2 Points)

Answer the following questions (0.5 point each):

a) What states can a Java thread be in? **Answer:**

From a very simplified perspective, a thread can be either in the runnable *(thread is executing) or the* not runnable *(everything else) state. However, there exist six states in the Java runtime environment such as* NEW *(thread not yet started),* RUNNABLE, BLOCKED *(thread waiting for a monitor lock),* WAITING *(waiting indefinitely for another thread to perform a particular action),* TIMED_WAITING *(WAITING with a timeout,* i.e., *upper time limit for the wait) and* TERMINATED *(thread exited). For more details, please have a look at the Java 10 API documentation on* https://docs.oracle.com/javase/10/docs/api/java/lang/Thread.State.html

b) How can you turn a Java class into a monitor? **Answer:**

It is as simple as declaring all public methods synchronized. *You will hear more on that in the fourth lecture titled "safety patterns".*

c) What is the *Runnable* interface good for? **Answer:**

An object that implements Runnable *can be set into the active state,* i.e., *you can create a running thread based on a* Runnable *object. Hence,* Runnable *serves the same purpose as the ordinary* Thread *class. However, there is one major difference: Because Java does not support the concept of multiple inheritance, inheriting from the* Thread *class (to implement concurrent code routines) prevents inheriting from any another class. In such cases where you need to inherit from another class your only option is to use the* Runnable *interface.*

d) Specify an FSP that repeatedly performs hello, but may stop at any time. **Answer:**

```
HELLO = (hello -> HELLO | hello -> STOP).
```

Exercise 2 (2 Points)

Consider the following Java implementation of a Singleton within a single-threaded application:

```
public class Singleton {
   private static Singleton instance = null;
   private Singleton() {}
   public static Singleton getInstance() {
      if(instance == null) {
         instance = new Singleton();
      }
      return instance;
   }
}
```

a) What could happen if the application is multithreaded? **Answer:**

There exists the possibility that many instances of the Singleton class are created, because multiple threads could concurrently verify the condition `(instance == null)`*, succeed, and then continue with the instantiation of the class* Singleton *potentially overwriting previously assigned variable values.*

b) How can you transform the existing implementation into a thread-safe singleton by just adding one keyword? Please provide your code! **Answer:**

The `getInstance()` *method needs to be synchronized as shown below:*

```
public static synchronized Singleton getInstance() {
   if(instance == null) {
      instance = new Singleton();
   }
   return instance;
}
```

c) Suppose there arrive 1000 requests/second from different threads at the singleton you just made thread-safe. Does your implementation introduce a bottleneck? If yes, how can you improve it? **Answer:**

Making the `getInstance()` *method synchronized will lock the entire class every time a thread calls this method and the lock won't be released before the active thread exits the method. This is not efficient. A more efficient solution with a tighter synchronization scope is as follows:*

```
private static volatile Singleton instance;

public static Singleton getInstance() {
   if(instance == null) {
      synchronized (Singleton.class) {
         if(instance == null) {
            instance = new Singleton();
         }
      }
   }
   return instance;
}
```

*Please be aware that this double-checked locking solution frequently causes much complexity in real-life scenarios (*e.g.*, when the same object references get used in- and outside of the synchronized block) and could lead to concurrency bugs which are much harder to track than with the simpler generic approach. You should also be aware of the keyword "volatile": It ensures in-order accesses for the variable (otherwise some compilers might optimize code which could break thread-safety). Please note that some older Java releases do not properly support this feature.*

Exercise 3 (3.5 Points)

Download LTSA from http://www.doc.ic.ac.uk/~jnm/book/ltsa/download.html. For each of the following processes shown in Figure 1, provide the Finite State Process (FSP) description of the corresponding

Labeled Transition System (LTS) graph. You may verify the FSP descriptions by generating the state machines and using the "draw" functionality of the tool. **Answer:**

```
APPOINTMENT = (hello -> converse -> goodbye -> STOP).

HOLIDAY = (arrive->relax->leave->HOLIDAY).

SPEED = (on->DRIVING),
DRIVING = (speed->DRIVING | off->SPEED).

LEFTONCE = (ahead-> (left->STOP |right->LEFTONCE)).

TREBLE = (in[i:1..3] -> out[i*3] -> TREBLE).

FIVETICK (N=5) = FIVETICK[1],
FIVETICK[i:1..N] = ( when(i<N) tick -> FIVETICK[i+1]
                   | when(i==N) tick -> STOP).

PERSON = (
  workday -> sleep -> work -> PERSON
  | holiday -> sleep -> (
                          play -> PERSON
                          | shop -> PERSON
                        )
).
```

Exercise 4 (2.5 Points)

Consider the full Race5K FSP from the lecture.

a) How many states and how many possible traces occur if the number of steps is 5 (as in the lecture)? **Answer:**

36 states (cartesian product of the individual state spaces) fostering 252 traces from the start state to the end (= deadlock) state.

b) What is the number of states and traces for the generalized case (*i.e.*, for n steps)? **Answer:**

We assume 2 processes, n denotes the number of steps.

- *Number of states for each process:* $(n + 1)$
- *Number of states in total:* $(n + 1)^2$
- *Number of traces:* $\frac{(2 \cdot n)!}{n! \cdot n!}$ *(also known as* $binomial(2n, n)$*) is the number of possible paths in an* $n \times n$ *regular, directed grid graph from node* $(0, 0)$ *to node* (n, n)*,*

c) Check your solution using the LTSA tool. **Answer:**

You can only check the number of states and the number of the transitions, but not the number of traces. You can gather those numbers by using the composition feature you can find in the toolbar. Example output for 5 steps:

```
Compiled:  COMPETITOR
Composition:
Race5K = tortoise:COMPETITOR || hare:COMPETITOR
State Space:
6 * 6 = 2 ** 6
Analysing...
Depth 11 -- States:  36 Transitions:  60 Memory used:  9298K
Trace to DEADLOCK:
tortoise.1
tortoise.2
tortoise.3
tortoise.4
tortoise.done
hare.1
hare.2
hare.3
hare.4
hare.done
Analysed in:  0ms
```

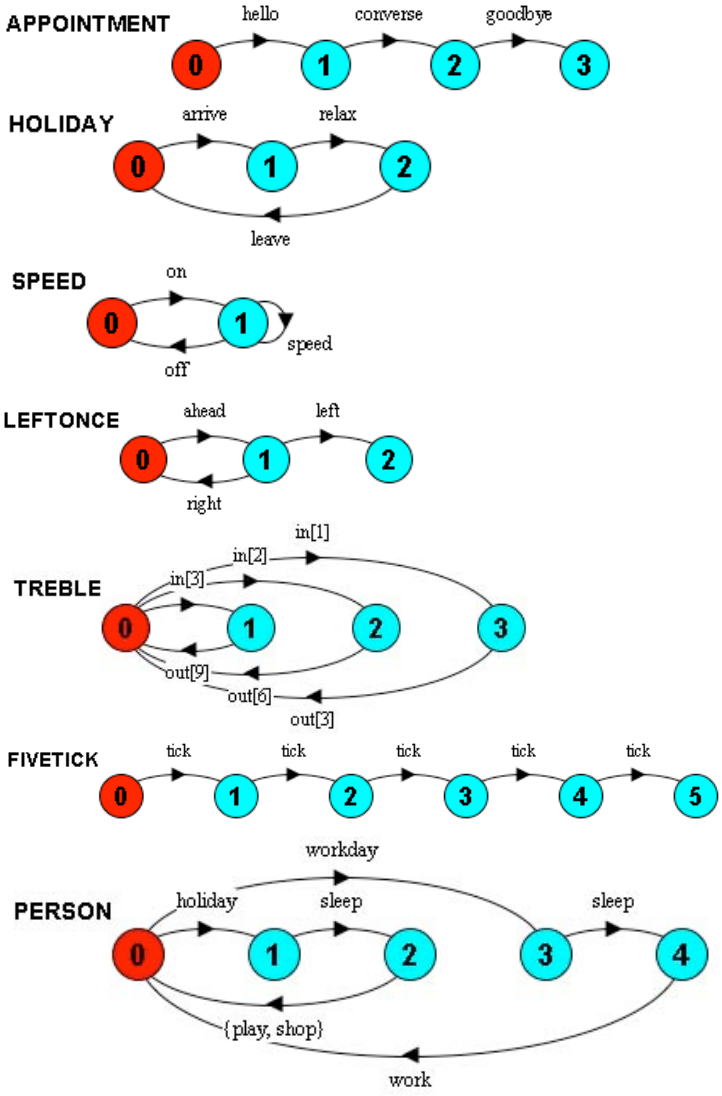

Figure 1: LTSA graphs.