CPDS-Conc Lab 3
Safety & Progres (I).

Authors: Jorge Castro and Joaquim Gabarro

Goal: Be familiar with safety and progress.

Basic material:

- Course slides.
- Basic reading: Chapters 7 of the book: *Concurrency, State Models & Java Programs*, Jeff Magee and Jeff Kramer, Wiley, Second Edition, 2006 (M&K for short).

3.1 Training Exercises

1. (M& K 7.3) Consider the car park problem of Chapter 5:

```
CARPARKCONTROL(N=4) = SPACES[N],
SPACES[i:0..N] = (when(i>0) arrive->SPACES[i-1] |when(i<N) depart->SPACES[i+1]).

ARRIVALS   = (arrive->ARRIVALS).
DEPARTURES = (depart->DEPARTURES).

||CARPARK = (ARRIVALS||CARPARKCONTROL(4)||DEPARTURES).
```

   - Specify a safety property which asserts that the car park does not overflow.

```
property OVERFLOW(N=4) = OVERFLOW[0],
OVERFLOW[i:0..N] = (arrive -> .... |...-> ... ).

||CHECK_CARPARK = (OVERFLOW(4) || ....).
```

   - Try safety check with `OVERFLOW(3)`
   - Specify a progress property which asserts that cars eventually enter the car park:

```
progress ENTER = {...}
```

   - Model the case where car departure is lower priority than car arrival, does starvation occurs?

```
||LIVE_CARPARK = CARPARK >>{...}.
```

2. *Gurp Airport.* The Gurb airport has several runways that are identified by a number in the range R. When a flight wants to take off, first it has to ask for a runway to the air traffic control tower, then it waits until the control tower allocates a runway. Afterwards, the flight takes off. The following FSP sketch models the Gurb airport

```
const False = 0
const True = 1

const RMAX = some number
const FMAX = some other number

range Bool = False..True
range R = 1..RMAX
range F = 1..FMAX

FLIGHT = (request.runway -> WAIT),
WAIT = (free.runway[i:R] -> take.off.done[i] -> FLIGHT).
........

||AIRPORT = (f[F]:FLIGHT || .......
```

- Assuming the airport has two runways (`RMAX = 2`) complete the `FSP` sketch above. *Hint:* complete the following snipped code for a `CONTROL` process:

  ```
  CONTROL = CONTROL[True][True],
  CONTROL[free_1: Bool][free_2:Bool] =.....
  ```

- Give a general description of `AIRPORT` valid for any number of runways (`RMAX >= 1`). *Hint:* complete the following snnipet code for `CONTROL` and `AIRPORT`:

  ```
  CONTROL_RUNWAY = ....

  ||CONTROL = ....

  ||AIRPORT = (........|| .......) / {.... /...., ..../....}.
  ```

- Write a safety property to show mutual exclusion on runways. As in the previous question, it is easier to start by considering an airport having two runways.
- Write a `AIRPORT_STRESS` process that models the airport behavior when there is a lot of runway request for taking off. Discuss the progress properties of the resulting process.

3.2 Homework

3.2.1 Alphabet extension

The *alphabet* of a process is the set of actions in which it can engage. How to deal with situations in which th set of actions in the alphabet is larger than the set of actions referenced in its definition? `Alphabet extension` can be used to extend the implicit alphabet of a process:

```
WRITER = (write[1]->write[3]->WRITER) + {write[0..3]}.
```

Alphabet of `WRITER` is the set {`write`[0..3]}. For more details, look at the section 2,2,6 of the M&K.

3.2.2 Exercises

1. *(M&K 7.6: The Warring Neighbors)* Two warring neighbors are separated by a field with wild berries. They agree to permit each other to enter the field to pick berries, but also need to ensure that only one of them is ever in the field at a time. After negotiation, they agree to the following protocol.

   When one neighbor wants to enter the field, he raises a flag. If he sees his neighbor's flag, he does not enter but lowers his flag and tries again. If he does not see his neighbor's flag, he enters the field and picks berries. He lowers his flag after leaving the field. Model this algorithm for two neighbors, `n1` and `n2` .

   - The following schema can be used to model the flags. Please complete the snipped code.

     ```
     const False = 0
     const True  = 1
     range Bool  = False..True
     set   BoolActions = {setTrue, setFalse, [False], [True]}

     BOOLVAR = VAL[False],
     VAL[v:Bool] = (setTrue  -> VAL[True]
                   |setFalse -> VAL[False]
                   |[v]      -> VAL[v]
                   ).

     ||FLAGS = (flag1:BOOLVAR || flag2:BOOLVAR).

     NEIGHBOR1 = (flag1.setTrue -> TEST),
     TEST      = (flag2[raised:Bool] ->
                     if (raised) then (flag1.setFalse -> NEIGHBOR1)
                     else (enter ... -> NEIGHBOR1)
                   ) + {{flag1,flag2}.BoolActions}.

     NEIGHBOR2 = (flag2.setTrue -> TEST),
     TEST      = (flag1[raised:Bool] ->...  ) + {{flag1,flag2}.BoolActions}.
     ```

   - Specify the required *safety property* `MUTEX` for the field and check that it does indeed ensure mutually exclusive access. In order to do the check, define a `FIELD` process by composing processes `FLAGS`, `NEIGHBOR`s and `MUTEX` and do the test with the analyser.

     ```
     property MUTEX = ....

     ||FIELD = (n1:NEIGHBOR1 ||n2:  ... || {n1,n2}:: ... || MUTEX).
     ```

   - Specify *progress properties* for the neighbors in order to check that, under fair scheduling policies, they eventually enter to the field to pick berries.

```
progress ENTER1  = {...}  //NEIGHBOR 1 always gets to enter
progress ENTER2  = {...}  //NEIGHBOR 2 always gets to enter
```

- Are there any adverse circumstances in which neighbors would not make progress? What if the neighbors are greedy? Argue your answers.

  Hint: Greedy neighbors - make setting the flags high priority - eagerness to enter. Provide the FSP description of the greedy neighbors and use the analyser to check progress violations to enter.

  ```
  ||GREEDY = FIELD << {...}.
  ```

2. *Field Program* We ask to develop a `Field` Java program corresponding to the *Warring Neighbors* exercise. As usual neighbors are `alice` denoted as `a` and `bob` denoted as `b`.

- Complete the following snipped code:

  ```
  public class Field {

      public static void main(String args[]) {
          Flags flags = new Flags();
          Thread a = new Neighbor(flags);
          a.setName("alice");
          ...
      }
  }
  ```

- Following a snipped for `Flag`

  ```
  public class Flags {

      private boolean flag_alice;
      private boolean flag_bob;

      public Flags() {
          flag_alice = false;
          ...
      }

      public synchronized boolean query_flag(String s) {
        //no condition synchronization is needed
        if (s.equals("alice")) return flag_bob;
        return ... ;
      }

      public synchronized void set_true(String s) {
          //no condition synchronization is needed
          if (s.equals("alice")) { flag_alice = true;}
          else { flag_bob = true; }
      }

      public synchronized void set_false(String s) {
          //no condition synchronization is needed
          if (s.equals("alice")) { ... }
          else { ... }
      }
  }
  ```

- Finally the neighbor (with no stress).

```
public class Neighbor extends Thread {

   private Flags flags;

   public Neighbor(Flags flags) {
      this.flags = flags;
   }

   public void run() {
      while (true) {
         try {
            String name = ... ;
            System.out.println("try again, my name is: "+ name);
            Thread.sleep((int)(200*Math.random()));
            flags. ...;
            if ( ... ) {
               System.out.println(name + " enter");
               Thread.sleep(400);
               System.out.println(name + " exits");
             }
             Thread.sleep((int)(200*Math.random()));
              flags ... ;
          }
          catch (InterruptedException e) {};
      }
   }
}
```

A possible printout could be

```
try again, my name is: alice
try again, my name is: bob
bob enter
try again, my name is: alice
bob exits
try again, my name is: bob
try again, my name is: alice
bob enter
try again, my name is: alice
try again, my name is: alice
try again, my name is: alice
bob exits
try again, my name is: alice
try again, my name is: bob
alice enter
try again, my name is: bob
try again, my name is: bob
alice exits
try again, my name is: bob
try again, my name is: alice
try again, my name is: alice
try again, my name is: bob
try again, my name is: alice
try again, my name is: bob
alice enter
```

```
try again, my name is: bob
alice exits
try again, my name is: bob
try again, my name is: alice
alice enter
...
```

- In order to model stress (or greedy), just change the sleep

```
public void run() {
   while (true) {
      try {
         String name = ...;
         System.out.println("try again, my name is: "+ name);
         flags. ...;
         //To model greedy write the sleep as follows
         Thread.sleep((int)(200*Math.random()));
          if (...) {
             ...
          }
          Thread.sleep((int)(200*Math.random()));
           flags. ...;
         }
         catch (InterruptedException e) {};
      }
   }
}
```

and in this case the printout should be:

```
try again, my name is: alice
try again, my name is: bob
try again, my name is: bob
try again, my name is: bob
try again, my name is: alice
try again, my name is: bob
try again, my name is: alice
try again, my name is: bob
try again, my name is: bob
try again, my name is: alice
try again, my name is: alice
try again, my name is: bob
try again, my name is: alice
try again, my name is: bob
try again, my name is: alice
...
```

3.2.3 How to deliver homework

Use the *Raco deliver feature* in order hand in these exercises. Each group has to submit only one file containing solutions for all exercises. The name of this file has to be composed by the subgroup number and the lastname of the participants. Moreover, names of all students in the group have to appear at the beginning of the document. *Groups of two people are mandatory.*

Important. The folder should contain two other folders.

- The first folder contains just a `txt` file with the solution to *The Warring Neighbors* exercise.
- The second folder constains the Java classes corresponding to the *Field Program*.