Construction and Verification of Software – 2019/2020

Second Self Assessment Test 16 June, 2020

Notes: This is the self-assessment test of the course. It is designed to be closed book and for a duration of 1h30m. There are 4 open answer questions.

Version: A

Name:_____Number:_____

Q-1 Consider the following Java class annotated with separation logic assertions (using verifast syntax), and complete the code with the weakest preconditions, and strongest post-conditions that completes the code below, so that verifast checks it without errors.

```
//@ predicate ClockInv(Clock c; int s) = c.seconds |-> s &*& s >= 0;

class Clock {
  int seconds;

  void sync(Clock other)
```


```
  { other.seconds = this.seconds; }

  int to(Clock other)
```


```
  { return other.seconds - this.seconds; }

  public static void main(String[] args)
  //@ requires [_]System_out(?s) &*& s != null;
  //@ ensures true;
  {
    Clock c1 = new Clock();
    Clock c2 = new Clock();
    c1.sync(c2);
    System.out.println(c1.to(c1));
  }
}
```

Q-2 Select the **incorrect** Separation Logic triple represented by the method definitions below. Assume defined a class `Clock` with a field named `seconds`, and class `Timer` with a method `toSeconds`.

A -
```
public void m(Clock c1, Timer t)
//@ requires c1 != null &*& c1.seconds |-> _ &*& t != null;
//@ ensures c1.seconds |-> _ ;
{ c1.seconds = t.toSeconds(); }
```

B -
```
public void m(Clock c1, int amount)
//@ requires c1 != null &*& c1.seconds |-> _;
//@ ensures c1.seconds |-> amount;
{ c1.seconds = amount; }
```

C -
```
public void m(Clock c1, Timer t)
//@ requires c1 |-> ?seconds &*& t != null;
//@ ensures c1 |-> seconds &*& t != null;
{ c1.seconds = t.toSeconds(); }
```

D -
```
public void m(Clock c1, Timer t)
//@ requires c1 != null &*& c1.seconds |-> _ &*& t != null;
//@ ensures true;
{ c1.seconds = t.toSeconds(); }
```

Q-3 Choose a set of predicates that is suitable to represent a binary tree using objects of class BTNode.

```
class BTNode {
  int value; BTNode left, right;

  BTNode(int v)
  //@ requires true;
  //@ ensures Tree(this);
  { value = v; left = right = null; }
  ...
}
```

Q-4 Consider an ADT that representing a calendar object with the following interface.

```
public interface Calendar {
  /*@
  predicate CalendarInv(boolean rw);
  @*/

  Appointment addAppointment(String description, Date date, int minutes, User[] attendees);
  //@ requires CalendarInv(true) &*& description != null &*& date != null &*& attendees != null;
  //@ ensures CalendarInv(true) &*& result != null;

  void removeAppointment(Appointment a);
  //@ requires CalendarInv(true) &*& a != null &*& a.isValid();
  //@ ensures CalendarInv(true);

  boolean isFree(Date date, int minutes);
  //@ requires CalendarInv(_) &*& date != null;
  //@ ensures CalendarInv(_);

  Appointment[] listAppointments(Date startDate, Date endDate);
  //@ requires CalendarInv(_) &*& startDate != null &*& endDate != null &*& lessOrEqual(startDate,endDate)
  //@ ensures CalendarInv(_);

  void LockCalendar();
  //@ requires CalendarInv(true);
  //@ ensures CalendarInv(false);

  void UnlockCalendar();
  //@ requires CalendarInv(false);
  //@ ensures CalendarInv(true);

  boolean isReadOnly();
  //@ requires CalendarInv(?rw);
  //@ ensures CalendarInv(rw) &*& result == rw;
}
```

Notice that predicate `CalendarInv` must be defined by all classes that implement the interface Calendar. **Implement a concurrent wrapper ADT** that uses an instance of the (sequential) ADT interface `Calendar` and uses a monitor and related conditions to control the exclusive acces to the object and establish the preconditions of the operations above. *Note: You do not need to write all verifast* close *and* open *operations, but you should state what is the shared state and the predicates ensured by each condition.*

Q-5 Consider the following implementation for function `linearOrderedLookup`

```
public static int linearOrderedLookup(int[] a, int x) throws IllegalArgumentException {
    if (a == null || a.length == 0) throw new IllegalArgumentException();
    int i = 0;
    for(; i < a.length && a[i] < x; i++);
    if( i == a.length ) return -1;
    return i;
}
```

1. **Present the control flow graph** of the function to support the design of glass-box tests (with unrolled loops).
2. **Produce a test** for each path in the graph (identify the path that corresponds to each path)