EXAMINATION

Formal Methods and Tools
Faculty of EEMCS
University of Twente

CONCURRENT & DISTRIBUTED PROGRAMMING

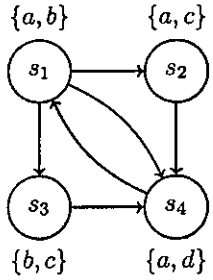
code: **192135300**
date: **3 February 2012**
time: **13.45–17.15**

- This is an 'open book' examination. You are allowed to have a copy of [Ben-Ari 2006] and [TAMP 2008], and a copy of the (unannotated) lecture *slides*. You are *not* allowed to take personal notes and (answers to) previous examinations with you.
- You can earn 70 points with the following 7 questions.
  The final mark for Concurrent & Distributed Programming is the sum of the marks obtained for this examination (70 points) and the three take home assignments (30 points). In addition, you can earn 10 bonus points in question 4 and 7, to make up for lost points elsewhere.

VEEL SUCCES!

Question 1 (10 points)

Consider the following state diagram that consists of four states. Four atomic propositions are used: a, b, c and d. Next to each state, the set of the atomic prositions is indicated that hold in that state.



You are requested to indicate for each of the following LTL-formulae the *set of states* for which these formulae are valid. Recall that an LTL-formula is valid in a state if and only if *all* paths starting in that state satisfy the formula.

a. *(2 pts)* $\bigcirc\bigcirc a$

b. *(3 pts)* $\Box\Diamond b$

c. *(2 pts)* $a \,\mathsf{U}\, d$

d. *(3 pts)* $\Box(a \Rightarrow (a \,\mathsf{U}\, (a \wedge c)))$

Question 2 (10 points)

Suppose we have two kinds of vehicles, bike and cars, that keep crossing a bridge. Since the bridge is rather narrow, only one vehicle can cross it at the same time. Suppose we have the following atomic propositions for a *Bike* at our disposal:

- *Bike.arrive* – indicates that a *Bike* arrives at the bridge;
- *Bike.onbridge* – indicates that a *Bike* has entered the narrow bridge and occupies it;
- *Bike.hasleft* – indicates that a *Bike* has left the bridge and continues its journey;

For a *Car* three similar predicates are defined.

Specify in Linear Temporal Logic the following properties, only using the six atomic predicates above.

a. *(2 pts) Mutual exclusion:* only one vehicle at a time can be on the bridge.

b. *(2 pts) Absence of starvation (for cars):* if a car arrives at the bridge, it will be able to proceed.

c. *(3 pts) Precedence (of bikes over cars):* a bike takes always precedence over a car.

d. *(3 pts)* Are absence of starvation and precedence compatible, or are these conflicting requirements? Explain your answer.

Question 3 (10 points)

Consider the following adaptation of Dekker's algorithm for the critical section algorithm. In this algorithm, the while-statement is replaced by an if- and an await-statement (l3,8).

boolean wantp ← false, wantq ← false, integer turn ← 1	
p	**q**
loop forever	loop forever
p1: *non-critical section*	q1: *non-critical section*
p2: wantp ← true	q2: wantq ← true
p3: if wantq	q3: if wantp
p4: if turn = 2	q4: if turn = 1
p5: wantp ← false	q5: wantq ← false
p6: await turn = 1	q6: await turn = 2
p7: wantp ← true	q7: wantq ← true
p8: await wantq = false	q8: await wantp = false
p9: *critical section*	q9: *critical section*
p10: wantp ← false	q10: wantq ← false
p11: turn ← 2	q11: turn ← 1

Question: Does this algorithm still ensure *mutual exclusion*?
If so, provide a complete set of invariants with which mutual exclusion can be proved; (you are not required to prove each invariant in detail).
If not, show a concrete trace that violates mutual exlcusion; (indicate which line is executed and the changes to the program variables).

Question 4 (10+5 points)

a. *(10 pts)* Consider the following MPI program for three processors, where x, y, z are program variables:

$\mathbf{P_0}$	$\mathbf{P_1}$	$\mathbf{P_2}$
Send(to:P_1, 42)	Recv(from:P_0, x)	Send(to:P_1, 27)
Send(to:P_1, 43)	Recv(from:$*$, y)	
	Recv(from:P_0, z)	

Which scenarios are possible for this example program? For each scenario, indicate:

- whether it deadlocks, or terminates successfully.
- what are the final value of x, y, z.

b. *(BONUS +5 pts)* Safra's algorithm for termination detection in a ring (as described by Dijkstra) uses a token with two fields: a global balance count and a status flag. Draw a concrete scenario in which the balance count can become a negative number.

Question 5 (10 points)

Consider *register* r with the following operations:

- $r.\mathrm{W}(x)$ *writes* value x into register r
- $r.\mathrm{R}(y)$ *reads* value y from register r

Assume that the usual consistency properties for registers hold.
Question: Draw histories of at most three processes (A, B, C), using one or more registers, such that certain consistency properties (*sequential consistency* (SC), *quiescently consistency* (QC), *linearizability* (LIN)) are satisfied:

a. *(3 pts)* Draw a history which is QC, but not SC and not LIN.

b. *(3 pts)* Draw a history which is SC, but not QC and not LIN.

c. *(4 pts)* Draw a history which is SC and QC, but not LIN.

In each case, explain why the properties hold (or do not hold), for example, by marking linearization points.

Question 6 (10 points)

Consider class BoundedStack in Figure 1. It implements a bounded stack of integers, using an array as internal representation. The implementation has several (different) concurrency errors. Discuss *three* different errors, explain why they cause a problem, and what could be done to fix them.

Each error can give three points, and if you correctly identify all three, you get an extra point.

```
import java.util.concurrent.locks.*;

class BoundedStack {

    private int count = 0;
    private int max;
    private int[] contents = new int[max];
    private Lock l = new ReentrantLock();

    BoundedStack(int m) {
        max = m;
    }

    int pop() throws EmptyException {
    int res;
    if (count > 0) {
        l.lock();
        res = contents[count--];
        l.unlock();
    }
    else { throw new EmptyException(); }
    return res;
    }

    int size() {
    return count;
    }

    void push(int e) throws FullException {
    if (count < max) {
        l.lock();
        contents[count++] = e;
        l.unlock();
    }
    else { throw new FullException(); }
    }
}

class EmptyException extends Exception {}
class FullException extends Exception {}
```

Figure 1: BoundedStack implementation

```
public class ImmutablePoint {

   private final int x;
   private final int y;

   ImmutablePoint(int x, int y) {
    this.x = x;
    this.y = y;
   }

   int x() {
    return x;
   }

   int y() {
    return y;
   }
}
```

Figure 2: Class `ImmutablePoint`

```
class Dot {

   protected ImmutablePoint loc;

   public Dot(int x, int y) {
     loc = new ImmutablePoint(x, y);
   }

   public synchronized ImmutablePoint location() {return loc;}

   public synchronized void shiftX(int delta) {
     ImmutablePoint old = location();
     loc = new ImmutablePoint(old.x() + delta, old.y());
   }
}
```

Figure 3: Class `Dot`

Question 7 (10+5 points)

Consider classes `ImmutablePoint` in Figure 2 and `Dot` in Figure 3.

Class `Dot` only uses synchronized methods. If the computation to compute the new location is difficult and time-consuming, this may affect performance.

a. *(10 pnts)* A possible solution is to limit synchronization to brief intervals, only to access and later update the state. This is called *optimistic update*. Methods changing the state then get the form:

   (1) get a copy of the current state representation (while holding a lock)

   (2) construct a new state representation (without holding any locks)

   (3) commit to the new state only if the old state has not changed since

   If the 3rd step fails, the whole process starts again.

   Question: Adapt the implementation of class `Dot` to a class `OptimisticDot`, supporting this optimistic update protocol.

b. *(BONUS +5 pnts)* Rewrite class `Dot` as a class `LockFreeDot` that is completely lock free.