EXAMINATION

Formal Methods and Tools
Faculty of EEMCS
University of Twente

CONCURRENT & DISTRIBUTED PROGRAMMING

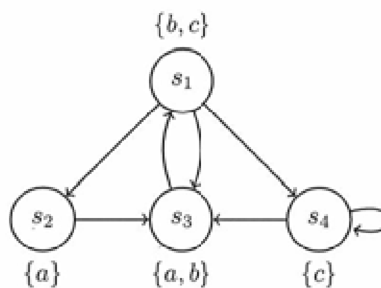
code: **192135300**
date: **1 February 2013**
time: **13.45–17.15**

- 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.
- This exam consists of 5 pages. 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 3 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. Three atomic propositions are used: a, b and c. Next to each state, the set of the atomic positions 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)* $a \mathsf{U} c$

b. *(2 pts)* $c \Rightarrow \Diamond b$

c. *(2 pts)* $\Box\Diamond b$

Express the following properties in LTL as well:

d. *(2 pts)* Whenever a holds, then from some later point in time, c will always hold

e. *(2 pts)* Whenever a holds, b must have been true at some earlier point in time

Question 2 (10 points)

Below is an algorithm for mutal exclusion as presented by Manna and Pnueli.

Manna-Pnueli Algorithm	
integer wantp ← 0, wantq ← 0	
p	q
loop forever *non-critical section* p1: if wantq = −1 then wantp ← −1 else wantp ← +1 p2: await wantq ≠ wantp *critical section* p3: wantp ← 0	loop forever *non-critical section* q1: if wantp = −1 then wantq ← +1 else wantq ← −1 q2: await wantp ≠ −wantq *critical section* q3: wantq ← 0

Note that the whole *if-then-else* statement is considered to be atomic. If the *then* and *else* branches were separate statements, then the algorithm would be incorrect.

a. *(3 pts)* Which methods can be used to (dis)prove the correctness of the mutual exclusion property of the algorithm? What are their advantages and disadvantages?

b. *(7 pts)* Prove the mutual exclusion property of the algorithm.

Question 3 (10+5 points)

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

$\mathbf{P_0}$	$\mathbf{P_1}$	$\mathbf{P_2}$
ISend(to:P_1, 42, h_0)	IRecv(from:$*$, x, h_1)	Barrier()
Barrier()	Barrier()	ISend(to:P_1,43,h_2)
Wait(h_0)	Wait(h_1)	Wait(h_2)
	Recv(from:P_2, y)	

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

- whether it deadlocks, or terminates successfully.
- what are the final values of x and y?

b. *(5 BONUS pts)* Consider an initial situation, in which a root process (0) has a special value x, and all workers $i \in W$ have their private value y_i. Write a parallel algorithm to compute $\sum_{i \in W} (x * y_i)$, using MPI's collective operations. You don't have to care about buffer allocation in memory.

Question 4 (10 points)

Safra's algorithm for termination detection in a ring network uses a token with two fields: a global balance count and a status flag.

a. *(4 pts)* Draw a concrete scenario in which the balance count can become a negative number.

b. *(3 pts)* Explain why the alternative Dijkstra/Scholten algorithm can be used in a more general setting.

c. *(3 pts)* Why can Safra's algorithm be more efficient than Dijkstra/Scholten's basic algorithm?

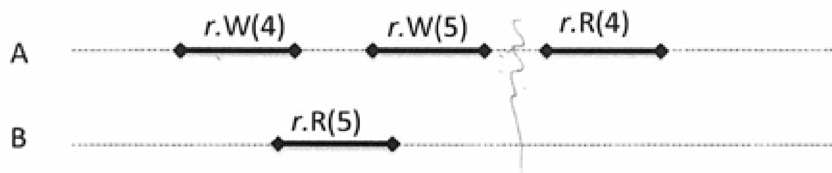
Question 5 (10 points)

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.

a. *(3 pts)* Consider the following history.

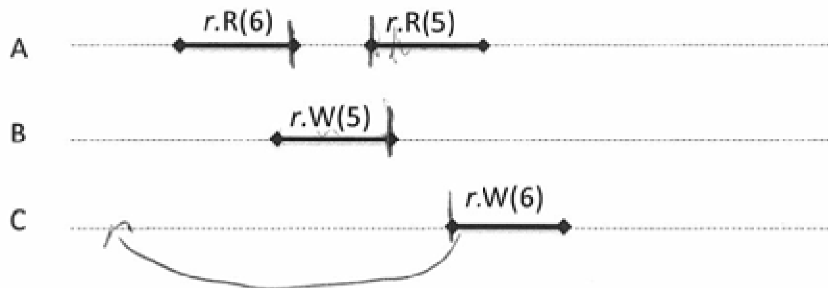


Is this history

- quiescently consistent (QC),
- sequentially consistent (SC),
- linearisable (LIN)?

Motivate your answer.

b. *(3 pts)* Consider the following history.



Is this history

- quiescently consistent (QC),
- sequentially consistent (SC),
- linearisable (LIN)?

Motivate your answer.

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

Question 6 (10 points)

Consider class Hotel in Figure 1. It implements some simple hotel functionality, maintaining an array of rooms and a list of people waiting to check in. For simplicity, it does not model that guests can check out; you can assume that a separate thread will take care of this.

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

class Person {
}

class Hotel extends Thread {

    int nr_rooms = 10;
    Person [] rooms = new Person [nr_rooms];
    List<Person> queue = new ArrayList<Person>();
    Lock queueLock = new ReentrantLock();

    boolean occupied(int i) {
        return (room[i] != null);
    }

    int checkIn(Person p) {
        int i = 0;
        while(occupied(i)) {i = (i + 1) % nr_rooms;}
        rooms[i] = p;
        return i;
    }

    void enter(Persoon p) {
        queueLock.lock();
        queue.add(p);
        queueLock.unlock();
    }

    // every desk employee is a thread
    public void run() {
    while (true) {
        if (!queue.isEmpty())
        {
            queueLock.lock();
            Person guest = queue.remove(0);
            queueLock.unlock();
            checkIn(guest);
        };
    }
    }
}
```

Figure 1: Hotel implementation

a. *(6 pts)* The implementation has several different (concurrency) errors. Discuss *three* different errors, explain why they cause a problem, and what should be done to fix them. Each error is worth 2 points.

b. *(4 pts)* Write a lock-free implementation of the checkIn method that can be safely used in a concurrent context. Also describe the changes necessary to the rest of the class.

Question 7 (10 + 5 points)

In this question, you are asked to write some OpenCL kernels. No points will be subtracted for OpenCL syntax errors.

a. *(5 pts)* Write a kernel foo(int* a, int a_size) such that every thread does the following:

   - It compares a[tid] with a[$tid - 1$].
   - If a[tid] is larger than a[$tid - 1$], a[tid] is added to a[$tid - 1$].
   - This process repeats until no assignments are taking place anymore.

   Make sure your kernel is free of data races. You may assume the existence of a function ALLZEROS that can test whether all values in an array are 0.

b. *(5 pts)* Write a kernel sum(int* a, int a_size) that computes the sum of all values in input array a in $\mathcal{O}(\log n)$. You are allowed to assume that the length of a is a power of 2.

c. *(5 BONUS pts)* Provide a loop invariant for your sum kernel.