EXAMINATION

Formal Methods and Tools
Faculty of EEMCS
University of Twente

CONCURRENT & DISTRIBUTED PROGRAMMING

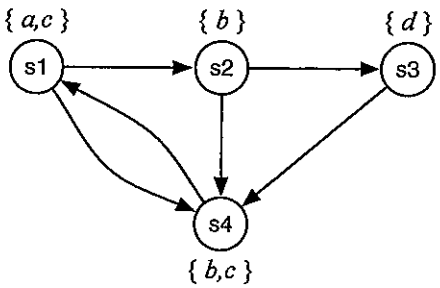
code: **213530**
date: **14 April 2009**
time: **13.30–17.00**

- This is an 'open book' examination. You are allowed to have a copy of [Ben-Ari 2006], a copy of the *slides*, and any additional *printed* papers or tutorials on the subject. 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).

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. As we are confronted with a non-deterministic state diagram – in the sense that, e.g., from state s1 there is a possibility to either move to state s2 or to state s4 – an LTL-formula is valid in a state if and only if *all* paths starting in that state satisfy the formula.

a. $\bigcirc\bigcirc a$

b. $\neg a \,\mathsf{U}\, b$

c. $((a \,\mathsf{U}\, b) \vee d) \Rightarrow \bigcirc b$

d. $\Box\neg d \Rightarrow \Box(a \,\mathsf{U}\, b)$

e. $\Box(b \vee c) \Rightarrow \Box(a \vee \bigcirc a)$

Question 2 (10 points)

For each natural language property below, give an LTL formula expressing the property. We assume p, q, and r to be atomic propositions. These atomic propositions can be seen as events: 'event p has happened/occurred' means that there has been a state where p was true.

a. After event p has happened, event q will never happen.

b. Event p always precedes q.

c. Event p occurs at most twice, i.e. there are most two states in the path where p is true.

d. The events p and q come in pairs: after each event p there will be an event q before a new p appears. Furthermore between each pair of p and q, event r never occurs.

e. Property p is true in each 'odd' state but false in each 'even' state, i.e. p is true in the 1^{st}, 3^{rd}, 5^{th}, etc. state, but false in the 2^{nd}, 4^{th}, 6^{th}, etc. state.

Question 3 (10 points)

Consider the following critical section algorithm, which is Algorithm 5.4 of [Ben-Ari 2006].

integer gate1 ← 0, gate2 ← 0

p	q
loop forever	loop forever
p1: *non-critical section*	q1: *non-critical section*
p2: gate1 ← p	q2: gate1 ← q
p3: if gate2 ≠ 0 goto p2	q3: if gate2 ≠ 0 goto q2
p4: gate2 ← p	q4: gate2 ← q
p5: if gate1 ≠ p	q5: if gate1 ≠ q
p6: if gate2 ≠ p goto p2	q6: if gate2 ≠ q goto q2
p7: *critical section*	q7: *critical section*
p8: gate2 ← 0	q8: gate2 ← 0

Question: Prove or disprove that this algorithm ensures *mutual exclusion.*

Question 4 (10 points)

Consider the following algorithm for the Dining Philosophers problem, which uses a *monitor*. The algorithm is Algorithm 7.5 of [Ben-Ari 2006].

```
monitor ForkMonitor
    integer array[0..4] fork ← [2, ..., 2]
    condition array[0..4] OKtoEat
    operation releaseForks(integer i)
        fork[i+1] ← fork[i+1]+1
        fork[i-1] ← fork[i-1]+1
        if fork[i+1] = 2
            signalC(OKtoEat[i+1])
        if fork[i-1] = 2
            signalC(OKtoEat[i-1])
    operation takeForks(integer i)
        if fork[i] ≠ 2
            waitC(OKtoEat[i])
        fork[i+1] ← fork[i+1]-1
        fork[i-1] ← fork[i-1]-1
```

philosopher i

```
    loop forever
p1:     think
p2:     takeForks(i)
p3:     eat
p4:     releaseForks(i)
```

Question: Proof that the following formula is invariant:

$$\neg empty(OKtoEat[i]) \Rightarrow (fork[i] < 2)$$

Question 5 (10 points)

Consider the following Java program.

```
class IntVal {
    private int val;
    public  IntVal(int val)     { this.val = val;   }
    public  void set(int val)   { this.val = val;   }
    public  int  get()          { return val;       }
}

public class TwoThreads {
    public static void main(String[] args) {
        final IntVal ival = new IntVal(1);

        Thread t1 = new Thread(new Runnable() {
            public void run()   { int n = ival.get(); ival.set(n+3); }
        });

        Thread t2 = new Thread(new Runnable() {
            public void run()   { int n = ival.get(); ival.set(n*5); }
        });

        t1.start(); t2.start();
        System.out.println(ival.get());
    }
}
```

a. *(6 pts.)* Model the Java program `TwoThreads` in PROMELA. Make sure that the PROMELA model can print the same values as the Java program.

b. *(4 pts.)* What values can be printed by the Java program (and the PROMELA model)?

Question 6 (10 points)

Consider the following *consensus* algorithm, which is Algorithm 12.2 from [Ben-Ari 2006]: the *Byzantine Generals* algorithm.

```
                planType finalPlan
                planType array[generals] plan
                planType array[generals, generals] reportedPlan
                planType array[generals] majorityPlan
p1:   plan[myID] ← chooseAttackOrRetreat

p2:   for all other generals G
p3:      send(G, myID, plan[myID])
p4:   for all other generals G
p5:      receive(G, plan[G])

p6:   for all other generals G
p7:      for all other generals G' except G
p8:         send(G', myID, G, plan[G])
p9:   for all other generals G
p10:     for all other generals G' except G
p11:        receive(G, G', reportedPlan[G, G'])

p12:  for all other generals G
p13:     majorityPlan[G] ← majority(plan[G] ∪ reportedPlan[*, G])

p14:  majorityPlan[myID] ← plan[myID]
p15:  finalPlan ← majority(majorityPlan)
```

Suppose now that there are four generals: X, Y, Z and T. The generals X, Y and Z are *loyal* generals. General T is a *traitor* general. The four generals behave as follows:

- X wants to attack ('A'),
- Y wants to retreat ('R'),
- Z wants to attack ('A'), and
- T tries to obstruct the consensus of X, Y and Z.
  T echoes the messages of its neighbours , so:
  - T always sends attack ('A') messages to X.
  - T always sends retreat ('R') messages to Y.
  - T always sends attack ('A') messages to Z.

Question: draw the *knowledge trees* for general Y and general T.

Question 7 (10 points)

Indicate for each of the statements below whether the statement is *true* or *false*. You do *not* have to motivate your answer.[1]

a. Semaphores are not contained in Java directly, but can be implemented with the primitives that Java offers.

b. The methods `wait` and `notify` of Java's `Object` class can only be called within a `synchronized` block or within a `synchronized` method.

c. Each constructor of a subclass of the Java class `Thread` must be declared `synchronized`.

d. It is not allowed in Java to call the method `run` of a `Thread` directly. Instead, the method `start` must be called. This method takes care that the method `run` is called by the JVM (by callback).

e. Java supports sockets as communication mechanism between threads. A disadvantage of sockets is that a protocol has to be defined for the exchange of messages via the sockets. Javas RMI library is implemented using sockets.

f. RMI is implemented in Java using a library of classes. There are no extra features added to Java to support RMI.

g. The RMI registry keeps references to both the *server* object as well as the *client* objects within an RMI Client/Server application.

h. An `atomic` block in PROMELA can be implemented in Java using a `synchronized` block or a `synchronized` method.

i. An assignment statement in PROMELA is always atomic. An assignment statement in Java enclosed in a `synchronized` block is always atomic.

j. Rendez-vous communication in a PROMELA model can be implemented in Java using RMI. Buffered communication in a PROMELA model can be implemented in Java using sockets.

[CDP 2008/2009 kw3 – 8 April 2009]

[1] For each wrong answer, 2 points are subtracted from the maximum of 10 points for this exercise. Consequently, in case you are not sure, it might be better to leave the question open than to answer it.