TDTS07 Lab Report 1

Nora Björklund and Christopher Hallberg

8 april 2013

1 Exercise

A<> P.s3 was not satisfied since s0 just can stay in its own state since there are no invariants, while **E<> P.s3** is satisfied since the system *can* end up in state s3. Once the system starts from the initial state it will never go back to it.

2 Fischer part I

n	Seconds to finish
8	< 1
9	4
10	16
11	60
12	240

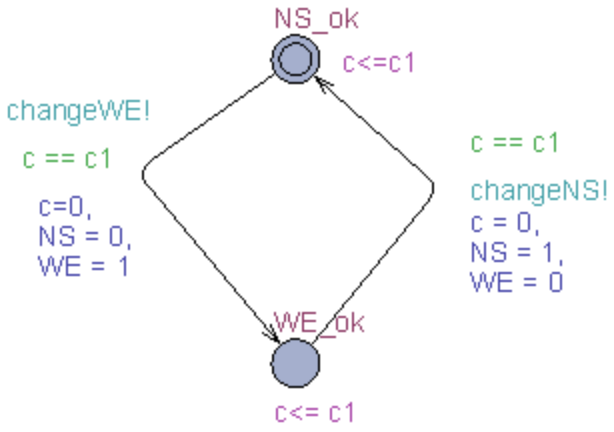
Tabell 1: Table for n instances of Fischer's mutual exclusion protocol

3 Fischer part II

4 Traffic Light Controller

The traffic light controller is implemented with a process P, and a timer. The timer have two states, one where it can let cars coming from north or

south drive and one state when cars from east and west can drive and it changes when an invariant c is more than a certain number c1. When the timer changes state a synchronisation call is set, the invariant c is set to 0 and boolean assignments that north-south or west-east can drive or not.

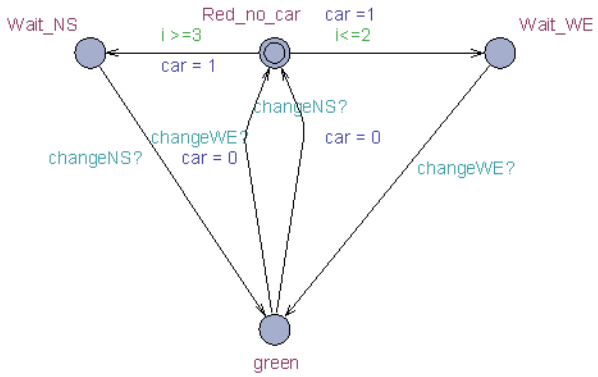


Figur 1: Timer

The process P starts in an initial state, Red_no_car, where the system have no waiting cars. Incoming cars have a signal i, and if that signal is greator than or equal to 3 then the car is coming from the north or the south. If it is less than or equal to 2 then it is a car from either west or east. The variable i is then used as an invariant placing the system in either state wait_WE (waiting car from west or east) or in wait_NS. Here the system synchronizes with the timer, so if a car from east is waiting for green then it has to wait for changeWE! to occur. When synched with the timer the system is in a state called green which represents that the car is given green and dissapears, we then once more wait to synchronize with the timer bringing the system back to the initial state. We have then created four instances of the process called N, S, W and E to represent the four directions.

We verify the following on the system:

- $A[]$ not deadlock, check that the system does not deadlock



Figur 2: Process P

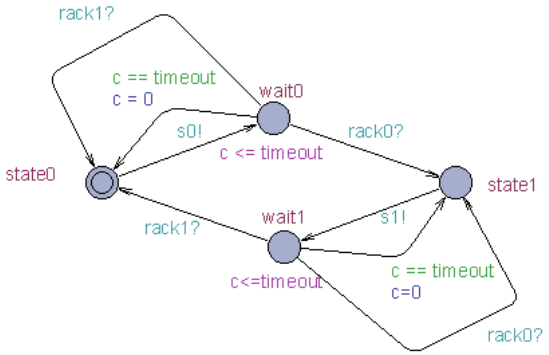
- $E <>$ X.green, check that the system eventually has the ability to end up in X.green (we do this for all directions so insert either N, S, W or E instead of X).
- $E <>$ E.Wait_WE, checks that the system eventually could end up in E.Wait_WE, we do the same for the other directions, though N and S uses Wait_NS instead of Wait_WE.
- $W.car -- > W.green$, checks that if we are in state W.car will transition to W.green (if true that means a waiting car will be given green)
- $S.car -- > S.green$
- $A[]$ not ((N.green or S.green) and (W.green or E.green)), if false then we can give a car from the north and the west green at the same time.

5 Alternating Bit Protocol

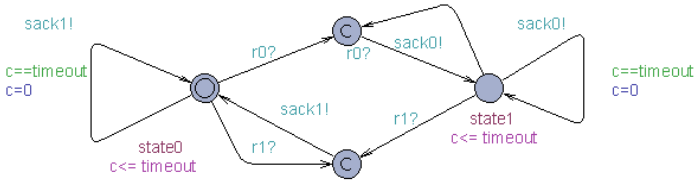
The alternating bit protocol consists of three processes, a sender, a reciever and an unreliable channel and it works as explained in the lab tutorial.

The system was verified for the following statements:

- $E <> U.from\_r\_to\_s$, the unreliable channel might eventually end up in state from_r_to_s, is a true statement.

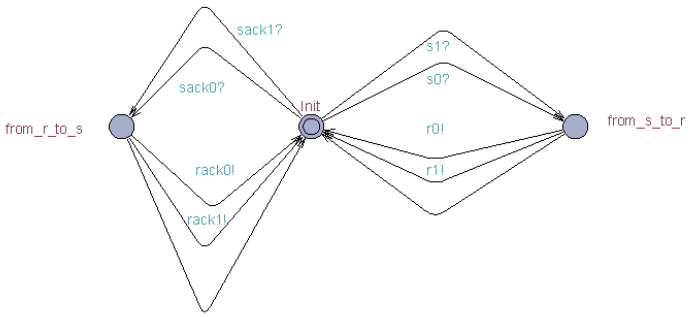


Figur 3: Sender



Figur 4: Reciever

- $S.state0 - - > R.state1$, given that the sender is in state0 the reciever will eventually go to its state1.This is a false statement since the unrelaiable channel can lose messages.
- $S.state1 - - > R.state0$, this is also a false statement (same explanation as above)
- $A[]notdeadlock$, the system will never deadlock (true in the system)



Figur 5: Unreliable channel