CS 2SD3. Sample solutions to the assignment 3.

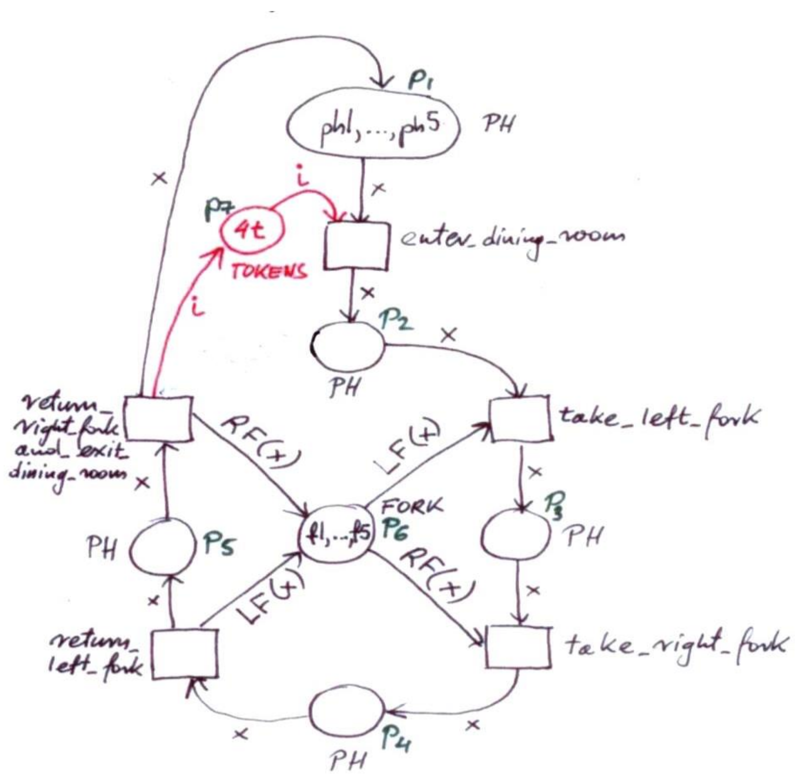
Total of this assignment is 123 pts. Each assignment is worth 10% of total. Most of solutions are not unique.

If you think your solution has been marked wrongly, write a short memo stating where marking in wrong and what you think is right, and resubmit to me during class, office hours, or just slip under the door to my office. The deadline for a complaint is 2 weeks after the assignment is marked and returned.

1.[10] Consider the Coloured Petri Net solution to Dining Philosophers with a butler, presented as a sample solution to Question 5 of Assignment 2. Prove that this solution is deadlock-free by mimicking the proof of Proposition from page 33 of Lecture Notes 12.

Solution:

colour PH = with ph1 | ph2 | ph3 | ph4 | ph
colour FORK = with f1 | f2 | f3 | f4 | f5
colour TOKENS = with t
var x : PH
var i: TOKENS
fun LF x = case of ph1 ⇒ f2 | ph2 ⇒ f3 | ph3 ⇒ f4 | ph4 ⇒ f5 | ph5 ⇒ f1
fun RF x = case of ph1 ⇒ f1 | ph2 ⇒ f2 | ph3 ⇒ f3 | ph4 ⇒ f4 | ph5 ⇒ f5



Interpretation of places:
p_1 - thinking room

p_2 - philosophers without forks in the dining room
p_3 - philosophers with left forks in the dining room
p_4 - philosophers that are eating
p_5 - philosophers that finished eating and still with right forks in the dining room
p_6 - unused forks
p_7 - butler or counter

Solution to the Question 1:

We need to find some appropriate invariants, for example:

[inv1] $m(p_1)+m(p_2)+m(p_3)+m(p_4)+m(p_5) = ph1+ph2+ph3+ph4+ph5$

[inv2] $|m(p_7)| +|m(p_2)| +|m(p_3)| +|m(p_4)| +|m(p_5)| = 4$

[inv3] $LF(m(p_4))+RF(m(p_4)) + m(p_6) = f1+ f2+ f3+ f4+ f4 + f5$

Now consider two cases:

(a) $m(p_4)+m(p_5) \neq 0$. Then either return_left_fork or returen_right_fork_and_exit_dinig_room can be fired.
(b) $m(p_4)+m(p_5) = 0$. Then from invariant [inv3] we have :

$$LF(m(p_3)) + m(p_6) = f1+ f2+ f3+ f4+ f4 +f5$$

and from invariant [inv1]:

$$m(p_1)+m(p_2)+m(p_3) = ph1+ph2+ph3+ph4+ph5.$$

From the definitions of LF(x) and RF(x) We have $LF(x) \neq RF(x)$ for all x = ph1, ph2, ph3, ph4, ph5.
Hence if $m(p_3) \neq 0$ then take_right_fork can be fired.
Similarly if $m(p_2) \neq 0$ then take_left_fork can be fired.
If $m(p_1) \neq ph1+ph2+ph3+ph4+ph5$, then either $m(p_3) \neq 0$ or $m(p_2) \neq 0$.
If $m(p_1) = ph1+ph2+ph3+ph4+ph5$ then $m(p_2) = 0$, and from invariant [inv2] $|m(p_7)|=4$, so enter_dining_room can be fired.

This is not the only solution.

2.[15] A self-service gas station has a number of pumps for delivering gas to customers for their vehicles. Customers are expected to prepay a cashier for their gas. The cashier activates the pump to deliver gas.

- a.[5] Provide a model for the gas station with *N* customers and *M* pumps. Include in the model a range for different amounts of payment and that customer is not satisfied (ERROR) if incorrect amount of gas is delivered.

- b.[5] Specify and check (with *N*=2, *M*=3) a safety property FIFO (First In First Out), which ensures that customers are served in the order in which they pay.

- c.[5] Provide a simple Java implementation for the gas station system with *N*=2, *M*=3.

a.

```
const N = 3   //number of customers
const M = 2   //number of pumps

range C = 1..N
range P = 1..M
range A = 1..2   //Amount of money or Gas

CUSTOMER = (prepay[a:A]->gas[x:A]->
             if (x==a) then CUSTOMER else ERROR).

CASHIER      = (customer[c:C].prepay[x:A]->start[P][c][x]->CASHIER).

PUMP         = (start[c:C][x:A] -> gas[c][x]->PUMP).

DELIVER = (gas[P][c:C][x:A]->customer[c].gas[x]->DELIVER).

||STATION = (CASHIER || pump[1..M]:PUMP || DELIVER)
            /{pump[i:1..M].start/start[i],
              pump[i:1..M].gas/gas[i]}.

||GASSTATION =  (customer[1..N]:CUSTOMER ||STATION).
```

b.

```
range T = 1..2
property
   FIFO          =  (customer[i:T].prepay[A] -> PAID[i]),
   PAID[i:T]     =  (customer[i].gas[A]      -> FIFO
                    |customer[j:T].prepay[A] -> PAID[i][j]),
   PAID[i:T][j:T]=  (customer[i].gas[A]      -> PAID[j]).

||CHECK_FIFO = (GASSTATION || FIFO).
```

This property is expected to be violated

c. Java solutions are not provided.

3.[15] The cheese counter in a supermarket is continuously mobbed by hungry customers. There are two sorts of customer: bold customers who push their way to the front of the mob and demand services; and meek customers who wait patiently for service. Request for service is denoted by the action *getcheese* and service completion is signalled by the action *cheese*.

(a)[5] Assuming that there is always cheese available, model the system with FSP for a fixed population of two bold customers and two meek customers.

(b)[5] Assuming that there is always cheese available, model the system with Petri nets (any kind).

(c)[5] For the FSP model, show that meek customers may never be served when their requests to get cheese have lower priority than those of bold customers.

Solutions:

a.[5] There are many simple solutions. The one is shown below

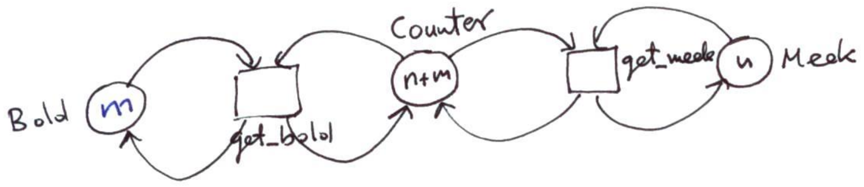
```
set Bold = {bold[1..2]}
set Meek = {meek[1..2]}
set Customers = {Bold,Meek}

CUSTOMER = (getcheese->CUSTOMER).
COUNTER =   (getcheese->COUNTER).

||CHEESE_COUNTER = (Customers:CUSTOMER || Customers::COUNTER)
```

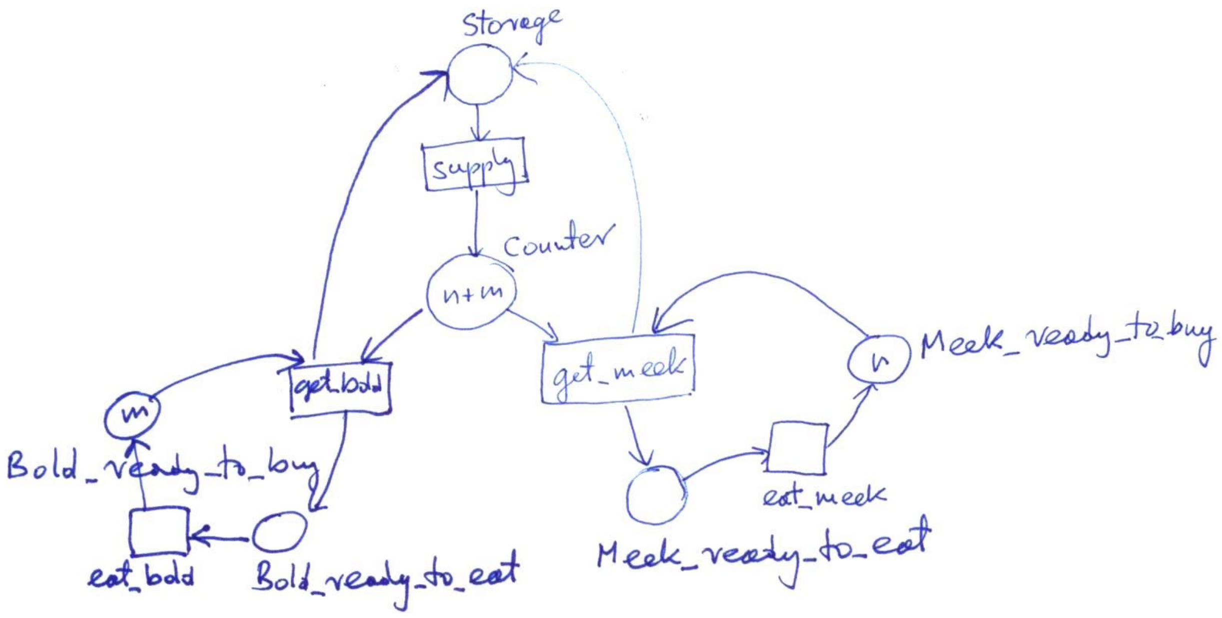
Another solution:

```
/* there is always cheese available */
CHEESE_DISPENSER = ( serve → cheese → CHEESE_DISPENSER ).

/* a customer is always trying to get cheese */
CUSTOMER = ( get_cheese → CUSTOMER ).

/* the hungry mob:
* The idea is to separate bold customers from
* meek customers by drawing a line, and then,
* prioritizing the actions of  bold customers. */

const Customers = 4 /* maximum number of customers */

range Bold = 1..Customers/2
range Meek = Customers/2+1..Customers


||HUNGRY_MOB = (
   forall[b : Bold] bold[b]:CUSTOMER
   || forall[m : Meek] meek[m]:CUSTOMER )
```

```
/* cheese counter */
||CHEESE_COUNTER = ( HUNGRY_MOB || CHEESE_DISPENSER )
  /{ {bold[Bold],meek[Meek]}.get_cheese/serve }.
```

b.[5] Two simple solutions in terms of Place/Transition nets.





c.[5] Adding priorities results in:
||CHEESE_COUNTER =
(Customers:CUSTOMER || Customers::COUNTER)>>{Meek.getcheese}.
If we use
progress BOLD = {Bold.getcheese}
progress MEEK = {Meek.getcheese}
clearly Mees.getcheese get starved, as Bold.getcheese will always get executed. The same with the second solution. A choice between meek and bold will always be solved in favour of bold. Any argument of this kind should be accepted.

4.[10] To restore order, the management installs a ticket machine that issues tickets to customers. Tickets are numbered in the range 1..*MT*. When ticket *MT* has been issued, the next ticket to be issued is ticket number 1, i.e. the management install a new ticket roll. The cheese counter has a display that indicates the ticket number of the customer currently being served. The customer with the ticket with the same number as the counter display then goes to the counter and is served. When the service is finished, the number is incremented (modulo *MT*). Model this system (with FSP) and show that, even when their requests have low priority, meek customers are now served.

Solution:

```
set Bold = {bold[1..2]}
set Meek = {meek[1..2]}
set Customers = {Bold,Meek}

const MT = 4  //maximum ticket number
range T = 1..MT

CUSTOMER = (ticket[t:T]->getcheese[t]->CUSTOMER).

TICKET = TICKET[1],
TICKET[t:T] = (ticket[t]->TICKET[t%MT+1]).

COUNTER = COUNTER[1],
COUNTER[t:T] = (getcheese[t]->COUNTER[t%MT+1]).

||CHEESE_COUNTER =
           (Customers:CUSTOMER || Customers::TICKET ||
Customers::COUNTER)>>{Meek.getcheese[T]}.

progress BOLD = {Bold.getcheese[T]}
progress MEEK = {Meek.getcheese[T]}
```

5.[10] Translate the model of the cheese counter from Question 4 into a Java program. Each customer should be implemented by a dynamically created thread that obtains a ticket, is served cheese and then terminates.

Solution:

```
/* -- Java implementation
class Ticket {
      const MT = 1000;
      private int next = 0;

      public synchronized int ticket() {
            next = next%MT + 1;
            return next;
      }
}

class Counter {
      const MT = 1000;
      private int serve = 1;

      public synchronized getcheese(int ticket)
                  throws Interruptedexception {
            while (ticket!=serve) wait();
            serve = serve%MT + 1;
            notifyAll();
      }
}
*/
```

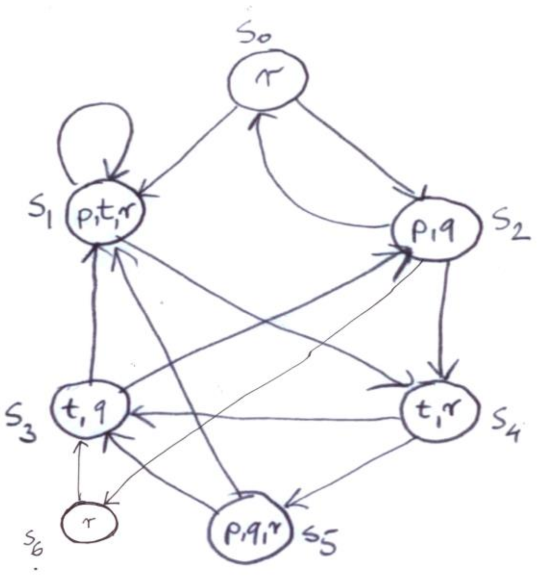
6.[10] Design (with FSP) a message-passing protocol which allows a producer process communicating with a consumer process by *asynchronous* messaging to send only a bounded number of messages, *N*, before it is blocked waiting for the consumer to receive a message. Construct a model which can be used to verify that your protocol prevents queue overflow if ports are correctly dimensioned.

Solution:

```
// the idea here is to send a set of N tokens to
// the producer. before sending the producer must get a token
// the consumer returns tokens in response to message receipt

// Asynchronous message passing port
//(turn off "Display warning messages")

const N = 3
set   M = {msg}
set   S = {[M],[M][M]}

PORT            //empty state, only send permitted
  = (send[x:M]->PORT[x]),
PORT[h:M]       //one message queued to port
  = (send[x:M]->PORT[x][h]
    |receive[h]->PORT
    ),
PORT[t:S][h:M]  //two or more  messages queued to port
   = (send[x:M]->PORT[x][t][h]
     |receive[h]->PORT[t]
     ).

PRODUCER = (empty.receive.token -> dest.send.msg -> PRODUCER).

CONSUMER = SENDBUF[N],
SENDBUF[i:1..N] = (empty.send.token -> if (i==1) then CONTINUE else
SENDBUF[i-1]),
CONTINUE = (dest.receive.msg -> empty.send.token -> CONTINUE).

||PRODCONS = (PRODUCER || CONSUMER || empty:PORT || dest: PORT)
             /{empty.[i:{send,receive}].token/empty[i].msg}.
```

7.[38] This question deals with Model Checking.

(a) Consider the system *M* defined below:



Determine whether $M, s_0 \models \varphi$ and $M, s_2 \models \varphi$ hold and justify your answer, where φ is the LTL or CTL formula:

(i)[2] $\neg p \Rightarrow r$
(ii)[2] $\neg$ EG r
(iii)[2] E(t U q)
(iv)[2] F q

b.[6] Express in LTL and CTL: 'Event *p* precedes *s* and *t* on all computational paths' (You may find it easier to code the negation of that specification first).

c.[6] Express in LTL and CTL: 'Between the events *q* and *r*, *p* is never true but *t* is always true'.

d.[6] Express in LTL and CTL: 'Φ is true infinitely often along every paths starting at s'. What about LTL for this statement?

e.[6] Express in LTL and CTL: 'Whenever p is followed by q (after some finite amount of steps), then the system enters an 'interval' in which no r occurs until t'.

f.[6] Express in LTL and CTL: 'Between the events q and r, p is never true'.

a. (i) $(\neg p \Rightarrow r)$ is equivalent to $(\neg(\neg p) \vee r) \equiv p \vee r$. We have $L(s_0)=\{r\}$ so $M, s_0 \models \varphi$. We have $L(s_2)=\{p,q\}$, so $M, s_0 \models \varphi$.

(ii) We have $r \in L(s_0)$ and $r \in L(s_1)$. Moreover there is an infinite path $s_0 \rightarrow s_1 \rightarrow s_1 \rightarrow s_1 \rightarrow \ldots$, so $M,s_0 \models \mathrm{EG}\ r$. Therefore, we infer $M,s_0 \not\models \neg\ \mathrm{EG}\ r$.
Since $r \notin L(s_2)=\{p,q\}$, so $M, s_2 \models \neg\ \mathrm{EG}\ r$ as *future includes present.*

(iii) See LN 15, page 68. Since $t \notin L(s_0)$ and $t \notin L(s_2)$, we have $M,s_0 \not\models \mathrm{E}(t\ \mathrm{U}\ q)$, and $M,s_2 \not\models \mathrm{E}(t\ \mathrm{U}\ q)$.

(vi) Since $q \in L(s_2)$ and there are infinite paths $s_0 \rightarrow s_2 \rightarrow \ldots$, we have $M, s_0 \models \mathrm{F}\ q$. And clearly since $q \in L(s_2)$ then $s_0 \models \mathrm{F}\ q$ (*future includes present* again).

b. Statement: ‘Event *p* precedes *s* and *t* on all computational paths’.
Negation: ‘There exists a path where *p* does not precede *s* or does not precede *t*’.

Ambiguities: Is the case when *p* never happens allowed? We assume it is not (which means ‘yes’ for negation). Does ‘precede’ allows *p* and *s* (or *p* and *t*) be in the same state? We assume it is not (which means ‘yes’ for negation).

LTL: $\mathrm{G}(\mathrm{F}p \wedge (p \Rightarrow \mathrm{F}s) \wedge (p \Rightarrow \mathrm{F}t))$
CTL: $\mathrm{AG}(\mathrm{AF}p \wedge \mathrm{AG}(p \Rightarrow \mathrm{AF}s) \wedge \mathrm{AG}(p \Rightarrow \mathrm{AF}t))$

c. Statement: ‘Between the events *q* and *r*, *p* is never true but *t* is always true’
Ambiguities: Is the case when r or q never happens allowed? We assume that it is not.
What exactly “between” means? We assume “between” is “closed interval” so p is false in the state that holds q and in the state that holds r.

LTL: $\mathrm{G}(\mathrm{F}\ q \wedge \mathrm{F}\ r \wedge (q \Rightarrow (\neg\ p\ \mathrm{U}\ r) \wedge (q \Rightarrow (\mathrm{F}t\ \mathrm{U}\ r)))$
CTL: $\mathrm{AG}(\mathrm{AF}\ q \wedge \mathrm{AF}\ r) \wedge \mathrm{AG}(q \Rightarrow \mathrm{A}(\neg\ p\ \mathrm{U}\ r))$

d. CTL: $s \models \mathrm{AG}(\mathrm{AF}\ \Phi)$
LTL: $s \models \mathrm{G}(\mathrm{F}\ \Phi)$

e. The process of translating informal requirements into formal specifications is subject to various pitfalls. On eof them is simply ambiguity. For example it is unclear whether “after some finite steps” means “at least one, but finitely many”, or whether zero steps are allowed as well. It may also be debatable what “then” exactly means in “... then the system enters...”. We chose to solve this problem for the case when zero steps are not admissible, mostly since “followed b” suggest a real state transition to tale place. The LTL formula is the following

$$\mathrm{G}(p \Rightarrow \mathrm{XG}(\neg\ q \vee \neg\ r\ \mathrm{U}\ t)),$$

while an equivalent CTL formula is:

$AG(p \Rightarrow AX\ AG(\neg q \ \vee A \neg r\ U\ t]))$,

It says: At any state, if p is true, then at any state which one can reach with at least one state transition from here, either q is false, or r is false until t becomes true (for all continuations of the computation path). This is evidently the property we intended to model. Various other “equivalent” solutions can be given.

f. Express in LTL and CTL: ‘Between the events q and r, p is never true’.

Ambiguities: Is the case when r or q never happens allowed? We assume that it is not. What exactly “between” means? We assume “between” is “closed interval” so p is false in the state that holds q and in the state that holds r.

LTL: $G(F\ q \wedge F\ r \wedge (\neg q \vee (\neg p\ U\ r))$

CTL: $AG(AF\ q \wedge AF\ r) \wedge AG(q \Rightarrow A(\neg p\ U\ r))$

8.[15] Consider *Readers-Writers* as described in the first part of LN12 and Chapter 7 of the textbook. Take the case of two readers and two writers and provide a model in LTL or CTL (your choice). You have to provide a state machine that defines the model as figures on pages 30 and 33 of LN15 for *Mutual Exclusion*, appropriate atomic predicates as n_1, n_2, t_1, t_2, c_1, c_2 for Mutual Exclusion, and appropriate safety and liveness properties.

Solution.

Solutions are structurally similar to Mutual Exclusion that was considered in class. Assume the following atomic predicates that characterise properties of processes:

lpr_i - local processing of *reader i*, *i*=1,2,
lpw_i - local processing of *writer i*, *i*=1,2,
tr_i - *reader i*, *i*=1,2, requests reading,
tw_i - *writer i*, *i*=1,2, requests writing,
r_i - *reader i*, *i*=1,2, is reading,
w_i - *writer i*, *i*=1,2, is writing,

Note that the solution restricted to writers only should be the same as Mutual Exclusion considered in class! Hence to avoid similar problems we have to introduce additional boolean variables (or atomic predicates): turn=w1, turn=w2 and turn=r, to indicate that worlds where writer 1 will write (turn=w1), writer 2 will write (turn=w2), or readers (one or both) will read (turn=r).

Now states can be identified by atomic predicates of the form:

(str1, str2,stw1,stw2, turn)

where: str1 ∈ {lpr_1 , tr_1 , r_1 }, str2 ∈ {lpr_2 , tr_2 , r_2 }, - status of readers;
stw1 ∈ {lpw_1 , tw_1 , w_1 }, str2 ∈ {lpw_2 , tw_2 , w_2 }, - status of readers;
turn ∈ {turn=w1, turn=w2,turn=r}, - status of turns.

Life of a reader is a simple cycle: (lpr_1 , *,*,*,*) → (tr_1 , *,*,*,*) → (r_1 , *,*,*,*) → back to beginning,
similarly for writers: (*,*,lpw_1 , *) → (*,*,tw_1 , *) → (*,*,w_1 , *) → back to beginning.

Not all combinations of atomic predicates are allowed, for example
$stw1 = w_1 \Rightarrow str1 \neq r_1 \wedge str2 \neq r_2 \wedge stw2 \neq w_2$, or
$str1 = r_1 \Rightarrow stw1 \neq w_1 \wedge stw2 \neq w_2$

Properties are also very similar to these for Mutual Exclusion:

Safety in LTL:$G\ (w_1 \Rightarrow \neg (w_2 \vee r_1 \vee r_2))$, or in CTL: $AG(w_1 \Rightarrow \neg (w_2 \vee r_1 \vee r_2))$
Liveness in LTL: $G(tr_1 \Rightarrow F\ r_1)$, or in CTL: $AG(tr_1 \Rightarrow AF\ r_1)$

The remaining analysis is also similar to Mutual Exclusion from the book.