CS 3SD3. Sample solutions to the assignment 3.

Total of this assignment is 166 pts. Each assignment is worth 11% 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 via e-mail (as pdf). The deadline for a complaint is 2 weeks after the assignment is marked and returned.

1.[10] A lift has a maximum property of ten people. In the model of the lift control system, passengers entering a lift are signalled by an `enter` action and passengers leaving the lift are signalled by an `exit` action.

a.[5] Provide a model of the lift in FSP.

b.[5] Specify a safety property in FSP which when composed with the lift check that the system never allows the lift it controls to have more than ten occupants.

Solution

a.

```
const N = 10

LIFT = LIFT[0],
LIFT[i:0..10] = (when(i<10) enter -> LIFT[i+1]
                |when(i>0) exit  -> LIFT[i-1]
                |when(i==10)go  -> LIFT[i]  //lift is full
                |when(i>0) go  -> LIFT[i]   //lift no longer waits
                ).
```

b.

```
const N = 10

property LIFTCAPACITY = LIFTC[0],
LIFTC[i:0..10] = (enter -> LIFTC[i+1]
                 |when(i>0) exit  -> LIFTC[i-1]
                 |when(i==0)exit  -> LIFTC[0]
                 ).
```

2.[20] The net model considered in Lecture Notes 12 assumed that each process can either read or write. Suppose that we have *three* kind of processes, *readers* that can only read, *writers* that can only write, and *rw-processes*, that can both read and write. Assume we have *m readers*, *k writers* and *n rw-processes*.

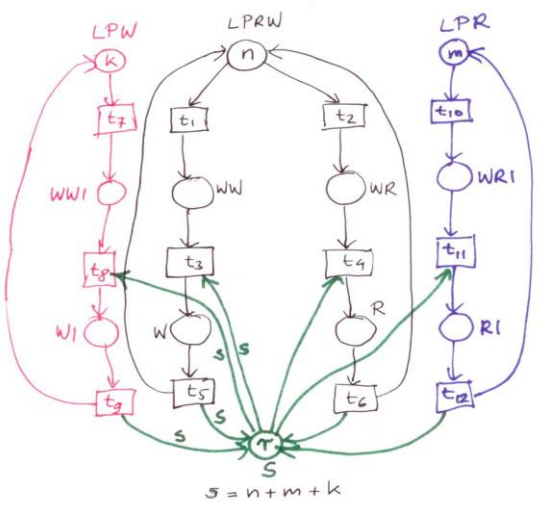
a.[10] Provide a Petri nets solution in the style of Lecture Notes 12 for this version of Readers and Writers problem. The solution should be for an **arbitrary** *n*, *m*, and *k*, **not** for any specific values of *n*, *m* and *k* (as for example *n*=3, *m*=2, *k*=3).

b.[10] Prove that your solution is deadlock-free by mimicking the proof of Proposition from page 27 of Lecture Notes 12.

Solution:

a. We assume s=n+m+k. Black part: rw-processes, red: writers, blue: readers, green: tickets
WW - waiting to write for rw-processes
WR - waiting to read for rw-processes
W - writing for rw-processes
R - reading for rw-processes
WW1 - waiting to write for writers
WR1 - waiting to read for readers
W1 - writing for writers
R - reading for readers
S - tickets (synchronization)

Note that we *cannot* join WW and WW1 into one place and similarly for WR and WR1, W and W1, R and R1.



b.

There are several obvious invariants. Let s=n+m+k.

in1: m(LPWR)+m(WW)+m(WR)+m(W)+m(R)=n
in2: m(LPW)+m(WW1)+m(W1)=k
in3: m(LPR)+m(WR1)+m(R1)=m
in4: m(LPW)+m(LPWR)+m(LPR)+m(WW1)+m(WW)+m(WR)+m(WR1)+
m(W1)+m(W)+m(R)+m(R1)=s

The invariant in4 follows from in1, in2 and in3 but it is very useful in explicit form.

in5: $m(R)+s\cdot m(W)+m(R1)+s\cdot m(W1)+m(s)= s$

If $m(LPW)+m(LPWR)+m(LPR)+m(W1)+m(W)+m(R)+m(R1)>0$, then at least one of t_1, t_2, t_7, t_{10}, t_9, t_5, t_6 or t_{12} has concession.
If $m(LPW)+m(LPWR)+m(LPR)+m(W1)+m(W)+m(R)+m(R1)=0$, from in4 we have:

$$m(WW1)+m(WW)+m(WR)+m(WR1)=s,$$

and from in5:

$$m(WR)+m(WR1)+m(WW)+m(WW1)=s$$
$$m(S)=s,$$

so either t_8 or t_3, or at least one of t_4, t_{11} has concession.

3.[30] Consider an scenario in which a number of producers P communicate with a number of consumers C via a buffer of messages, or asynchronous channel, of capacity B. Each consumer has associated with it a particular id and it must consume only those messages addressed to it. On the other hand, producers produce messages for all of the consumers. Messages produced by producers ought not to be lost. As a particularity, there exists a process MIX that nondeterministicaly engages in either a consumption or a production cycle.

a.[15] Your task is to model the above scenario in *FSP,*
b.[15] and later on provide a Java implementation.

Moreover, you are asked to specify some properties your model is desired to have, such as:

(a) The model is deadlock free.

(b) Producers produce infinite messages for each consumer.

(c) No producer is blocked indefinitely.

(d) No consumer is blocked indefinitely.
(e) Each consumer only consumes messages addressed to it.

and to guarantee that those properties (and others you may want) are satisfied by both the model and the implementation.

Modelling hints

The overall structure of the system can be modeled in terms of the following processes:

(a) BUFFER: this process can be used to model the buffer of messages. You can use the characterization provided in Chapter 10 of the textbook (or Lecture Notes 14) of an asynchronous buffer of communication. Note however that in FSP the size of a queue must be of a fixed size, hence some considerations should be given to avoiding overflow (or underflow) conditions.

(b) PRODUCER: this process can be regarded as modelling the generation of the messages to every consumer and sending them to the buffer.

(c) CONSUMER: this process can be regarded modelling the receiving of messages from the buffer and consuming them.

(d) MIX: this process can be regarded as modelling the behaviour of the "mix" process.

(e) AMP: this process is just the composition of all of the aforementioned processes (together with the proper synchronization of actions) modelling the behavior of an asynchronous message passing communication.

Moreover, you must carefully document any design decision you assume and/or take using proper comments in the source code and structure diagrams whenever necessary.

Implementation hints

The overall implementation details that you may consider include:

(a) Create a makefile and a run script, further details can be asked during the tutorials.

(b) Use Java library classes whenever possible.

(c) Design, document and test your code - Provide a short report detailing this point.

(d) Carefully document any assumptions and design decisions you make.

(e) Some messages should be displayed. This enables testing of each component. Include a description of the intended meaning of each one of them. Describe the design of your test plan and describe the outcome of the testing process.

Solution.
a.[15] Model in FSP

```
const P = 1   /* number of producers */
const C = 1   /* number of consumers */
const M = C+1 /* id for the mix process */

range Msgs = 0..M  /* messages ids */
range Prods = 0..P   /* producers ids */
range Cons = 0..C  /* consumers ids */

const B = 4 /* buffer bound */

/* buffer msgs queue */
set Queue = {[Msgs], [Msgs][Msgs], [Msgs][Msgs][Msgs]}

/*************************************************************************/

/* asynchronous communication buffer. It can store up to 4
msgs without overflowing. */

BUFFER = (
    enqueue[m: Msgs] → BUFFER[m] ),

    BUFFER[m: Msgs] =  (
        enqueue[n: Msgs] → BUFFER[m][n]
        |dequeue[m] → BUFFER ),

    BUFFER[m: Msgs][ms: Queue] = (
        enqueue[n: Msgs] → BUFFER[m][ms][n]
        | dequeue[m] → BUFFER[ms] ).

BOUND = BOUND[0],

    BOUND[n: 0..3] = (
        when n < 3 increment → BOUND[n+1]
        | when 0 < n decrement → BOUND[n-1] ).

||BOUNDED_BUFFER = ( BOUND || BUFFER )


   /{ enqueue[Msgs]/increment, dequeue[Msgs]/decrement }.

/************************************************************************/
```

```
/* producer */

PRODUCER = (
    produce[msg:Msgs] → send[msg] → PRODUCER ).

||PRODUCERS = ( forall[i:Prods] p[i]:PRODUCER ).

/*************************************************************************/

/* consumer */

CONSUMER(Id=0) =
 ( receive[Id] → consume[Id] → CONSUMER ).

||CONSUMERS = ( forall[i:Cons] c[i]:CONSUMER(i) ).

/*************************************************************************/

/* mix process, i.e.: non deterministicaly it behaves as a
producer or as a consumer. */

MIX(Id=0) = (
    switch → produce[msg:Msgs] → STUTTER[msg]
    | switch → receive[Id] → consume[Id] → MIX ),

    STUTTER[msg: Msgs] = (
        send[msg] → MIX
        | receive[Id] → consume[Id] → STUTTER[msg] ).

/*************************************************************************/

/* system properties. */

/* There are infinite messages produced for each consumer. */

progress MSGS00 = { p[0].produce[0] }
progress MSGS01 = { p[0].produce[1] }
progress MSGS02 = { p[0].produce[2] }
progress MSGS10 = { p[1].produce[0] }
progress MSGS11 = { p[1].produce[1] }
progress MSGS12 = { p[1].produce[2] }

/* No producer is blocked indefinitely. */

progress SENDS0 = { p[0].send[Msgs] }
progress SENDS1 = { p[1].send[Msgs] }

/* No consumer is blocked indefinitely. */

progress C0 = { c[0].consume[0] }
progress C1 = { c[1].consume[1] }
```

```
progress C2 = { mix.consume[2] }

/* Each consumer only consumes messages addressed to it */

property ADDRESSED = (
    c[i: Cons].consume[i] → ADDRESSED
    | mix.consume[M] → ADDRESSED ).

/*****************************************************************************/

/* Asynchronous message passing. */

||AMP = ( PRODUCERS || BOUNDED_BUFFER || mix:MIX(M) || CONSUMERS || ADDRESSED )
    /{ p[Prods].send/enqueue,

       mix.send/enqueue,
       mix.receive[M]/dequeue[M],

       c[i: Cons].receive[i]/dequeue[i] }.
```

b.[10] Java implementation is not presented.

4.[20] Two warring neighbours are separated by a field with wild berries. They agree to permit each other to enter the field to pick berries, but also need to ensure that only one of them is ever in the field at a time. After negotiation, they agree to the following protocol. When a one neighbour wants to enter the field, he raises a flag. If he sees his neighbour's flag, he does not enter but lowers his flag and tries again. If he does not see his neighbour's flag, he enters the field and picks berries. He lowers his flag after leaving the field.

a.[10] Model this algorithm for two neighbours `n1` and `n2`. Specified the required *safety* properties for the field and check that it does indeed ensure mutually exclusive access. Specify the required *progress* properties for the neighbours such that they both get to pick berries given a fair scheduling strategy. Are any adverse circumstances in which neighbours would not make progress? What if the neighbours are greedy?

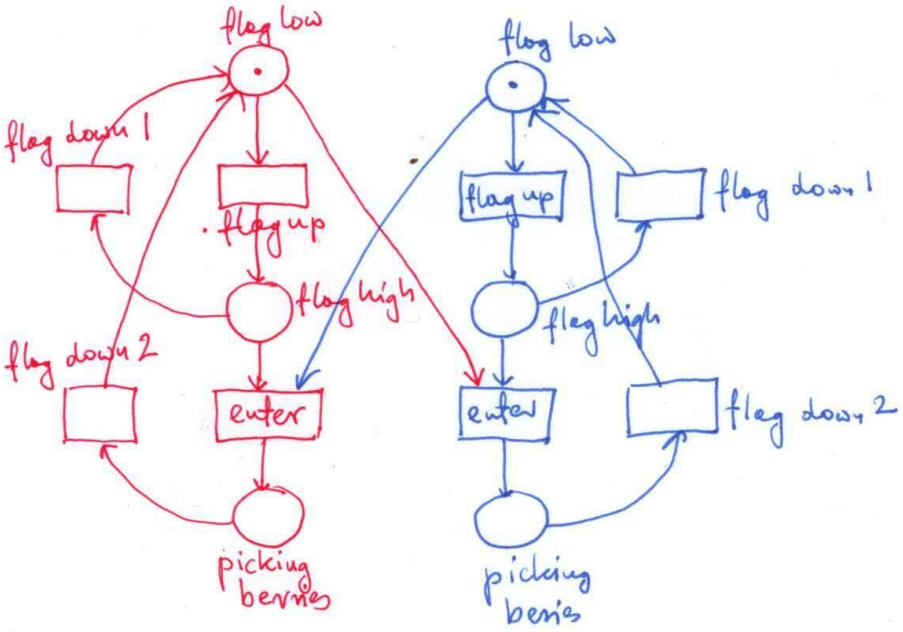
b.[10] Model this algorithm for two neighbours using Petri nets (any kind)

Solution:

(a)[10]

```
const False = 0
const True  = 1
range Bool  = False..True
set   BoolActions = {setTrue, setFalse, [False], [True]}

BOOLVAR = VAL[False],
VAL[v:Bool] = (setTrue  -> VAL[True]
              |setFalse -> VAL[False]
              |[v]      -> VAL[v]
              ).

||FLAGS = (flag1:BOOLVAR || flag2:BOOLVAR).

NEIGHBOUR1 = (flag1.setTrue -> TEST),
TEST       = (flag2[b:Bool] ->
                if(b) then
                   (flag1.setFalse -> NEIGHBOUR1)
                else
                   (enter -> exit -> flag1.setFalse -> NEIGHBOUR1)
             )+{{flag1,flag2}.BoolActions}.

NEIGHBOUR2 = (flag2.setTrue -> TEST),
TEST       = (flag1[b:Bool] ->
                if(b) then
                   (flag2.setFalse -> NEIGHBOUR2)
                else
                   (enter -> exit-> flag2.setFalse -> NEIGHBOUR2)
             )+{{flag1,flag2}.BoolActions}.

property SAFETY = (n1.enter -> n1.exit -> SAFETY | n2.enter -> n2.exit
-> SAFETY).

||FIELD = (n1:NEIGHBOUR1 || n2:NEIGHBOUR2 || {n1,n2}::FLAGS ||
SAFETY).

progress ENTER1  = {n1.enter}  //NEIGBOUR 1 always gets to enter
progress ENTER2  = {n2.enter}  //NEIGHBOUR 2 always gets to enter

/* greedy neighbours - make setting the flags high priority -
eagerness to enter*/
||GREEDY = FIELD << {{n1,n2}.{flag1,flag2}.setTrue}.

/* progress violations show situation where neither neighbour enters
*  each continually retests the lock
*/
```

(b)[10]



5.[10] *Simplified Multidimensional Semaphores* are defined as follows:

(i) The extended primitives *edown* and *eup* are atomic (indivisible) and each operates on a set of semaphore variables which must be initiated with non-negative integer value.

(ii) *edown*(S_1,...,S_n):
if for all i, $1 \le i \le n$, $S_i > 0$ then for all i, $1 \le i \le n$, $S_i := S_i - 1$
else block execution of calling processes

(iii) *eup*(S_1,...,S_n):
if processes blocked on (S_1,...,S_n) then awaken one of them
else for all i, $1 \le i \le n$, $S_i := S_i + 1$

Model the *Simplified Multidimensional Semaphores* by FSPs for n=2 and maximal value of S_1, S_2 equal to 3.

Hint. *edown*(S_1,S_2) could be interpreted as just a synchronized execution of *down*(S_1) and *down*(S_2). Similarly for *eup*(S_1,S_2

Solution:

Generic semaphore is defined in FSP's as:

```
SEM(N=INITIAL_VALUE) = SEMA[N]
SEMA[v:Int] = (when(v≤Max) up -> SEMA[v+1] |
               when(v>0) down -> SEMA[v-1])
```

Then a Simplified Multidimensional Semaphores over variables S_1, S_2 with the and maximal values of S_1, S_2 equal to 3, can be modelled by FSPs as follows:

```
SEMS1S2(INITIAL1=3,INITIAL2=3)  =
(S1:SEM(3) || S2:SEM(3))
/{S1.S2.up/S1.up, S1.S2.up/S2.up, S1.S2.down/S1.down,
     S1.S2.down/S1.down}
```

6.[20] 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.[10] 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.[10] 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.

Solution: (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).
```

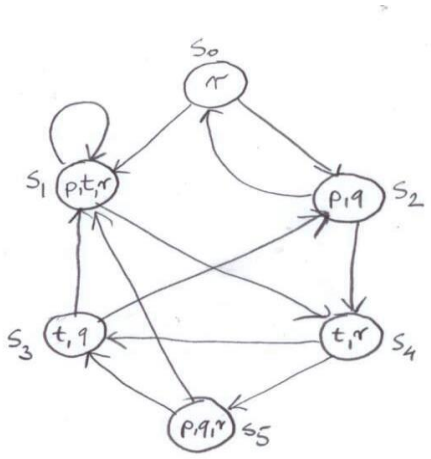
(b)

```
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

7.[36] This question deals with Model Checking.

(a)[12] 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)[3] $\neg p \Rightarrow r$

(ii)[3] $\neg$ EG r

(iii)[3] E(t U q)

(iv)[3] F q

(b)[8] 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)[8] Express in LTL and CTL: 'Between the events *q* and *r*, *p* is never true but *t* is always true'.

(d)[8] Express in CTL: 'Φ is true infinitely often along every paths starting at s'. What about LTL for this statement?

Solutions:

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 $M, s_2 \models \varphi$.

(ii) We have $r \in L(s_0)$ and $r \in L(s_1)$. Moreover there is an infinite path $s_0 \rightarrow s_1 \rightarrow s_1 \rightarrow s_1 \rightarrow$ ..., so $M,s_0 \models$ EG r. Therefore, we infer $M,s_0 \not\models \neg$ EG r.
Since $r \notin L(s_2)=\{p,q\}$, so $M, s_2 \models \neg$ 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 $M,s_0 \not\models$ E(t U q), and $M,s_2 \not\models$ E(t U q).

(vi) Since $q \in L(s_2)$ and there are infinite paths $s_0 \rightarrow s_2 \rightarrow$ ..., we have $M, s_0 \models$ F q. And clearly since $q \in L(s_2)$ then $s_0 \models$ 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: G(Fp ∧ (p ⇒ Fs) ∧ (p ⇒ Ft))
CTL: AG(Fp ∧ AG(p ⇒ AFs) ∧ AG(p ⇒ AFt))

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: G(F q ∧ F r ∧ (q ⇒ (¬ p U r) ∧ (q ⇒ (Ft U r)))
CTL: AG(F q ∧ F r) ∧ AG(q ⇒ A(¬ p U r))

d. CTL: s |= AG(AF Φ)
LTL: s |= G(F Φ)

8.[20] Consider *Readers-Writers* as described on page 14 of Lecture Notes 12 and analysed in Lecture Notes 12 after page 14. Take the case of three processes and provide a model in LTL or CTL. You have to provide a state machine that defines the model as figures on pages 30 and 33 of Lecture Notes 15 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*=1,2,3
lpw_i - local processing of *writer i*, *i*=1,2,3
tr_i - *reader i*, *i*=1,2,3 requests reading,
tw_i - *writer i*, *i*=1,2,3 requests writing,
r_i - *reader i*, *i*=1,2,3 is reading,
w_i - *writer i*, *i*=1,2,3 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 introduced additional boolean variables (or atomic predicates): turn=w1, turn=w2, turn=w3 and turn=r, to indicate that worlds where writer 1 will write (turn=w1), writer 2 will write (turn=w2), writer 3 will write (turn=w3) or readers (one or both) will read (turn=r).

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

(str1, str2,str3,stw1,stw2, stw3,turn)

where: str1 ∈ {lpr_1 , tr_1 , r_1 }, str2 ∈ {lpr_2 , tr_2 , r_2 }, str3 ∈ {lpr_3 , tr_3 , r_3 }- status of readers;
stw1 ∈ {lpw_1 , tw_1 , w_1 }, str2 ∈ {lpw_2 , tw_2 , w_2 }, str3 ∈ {lpw_3 , tw_3 , w_3 } - status of writerss;
turn ∈ {turn=w1, turn=w2, turn=w3, 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 ⇒ str1 ≠ r_1 ∧ str2 ≠ r_2 ∧ str3 ≠ r_3 ∧ stw2 ≠ w_2 ∧ stw3 ≠ w_3 , or
str1 = r_1 ⇒ stw1 ≠ w_1 ∧ stw2 ≠ w_2 ∧ stw3 ≠ w_3.

Properties are also very similar to these for Mutual Exclusion:

Safety in LTL: G (w_1 ⇒ ¬ (w_2 ∨ w_3 ∨ r_1 ∨ r_2 ∨ r_3)), or in CTL: AG(w_1 ⇒ ¬ (w_2 ∨ w_3 ∨ r_1 ∨ r_2 ∨ r_3)), etc.
Liveness in LTL: G(tr_1 ⇒ F r_1), or in CTL: AG(tr_1 ⇒ AF r_1), etc.

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