1a

i).

FLIGHT = ( sit->FLIGHT

| seatBeltSignOff -> SAFE

| {seatBeltSignOn, walk} -> ERROR ),

SAFE = ( {sit, walk} -> SAFE

| seatBeltSignOn -> FLIGHT

| seatBeltSignOff->ERROR ).

alternative:

property FLIGHT = UNSAFE,

UNSAFE = ( sit -> UNSAFE

| seatBeltSignOff -> SAFE ),

SAFE = ( {sit, walk} -> SAFE

| seatBeltSignOn -> UNSAFE ).

ii). Ruhi has a nice explanation on [Piazza](https://piazza.com/class/itv4vb3e5wm8i?cid=28).

MILES = MILE[0],

MILE[m:0..3] = ( mileage[m] -> MILE[m]

| when (m<3) fly[f:1..(3-m)] -> MILE[m+f]

| when (m>0) redeem[r:1..m] -> MILE[m-r]).

iii).

CABIN = ( buyEconomy -> ECO ),

ECO = ( requestUpgrade -> flyEconomy -> CABIN

| requestUpgrade -> flyBusiness -> CABIN).

not sure what is an acceptable solution for deterministic FSP, possibly:

ECO = ( when (1) requestUpgrade -> flyEconomy -> CABIN

| when (0) requestUpgrade -> flyBusiness -> CABIN).

For deterministic i instead think:

CABIN = (buyEconomy -> requestUpgrade -> UPGRADE),

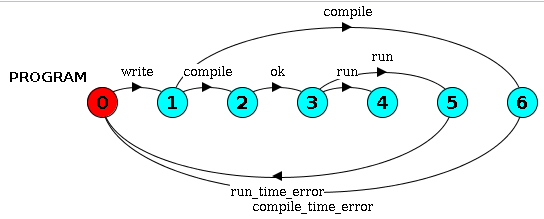
UPGRADE = (flyEconomy -> CABIN | flyBusiness -> CABIN).

the above answer effectively ignores the possibility of upgrading to business

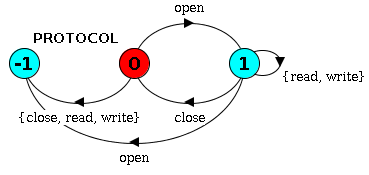
“(when false)”

1b

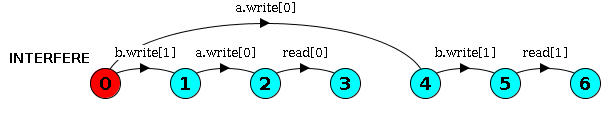
i).



ii).



iii).



Can be minimised by merging states 3 and 6.

1c

ONLY\_FOUR = ONLY\_FOUR[0],

ONLY\_FOUR[c:0..N] = ( when (c<4) login -> ONLY\_FOUR[c+1]

| when (c>0) logout -> ONLY\_FOUR[c-1] ).

Assuming, that admins can’t disable logged in users:

USER = USER[False],

USER[enabled:Bool]

= ( when (!enabled) enable -> login

-> logout -> USER[True]

| when (enabled) disable -> USER[False] ).

||SYSTEM = ( user[U]:USER || user[U]::ONLY\_FOUR ).

------

A bit more complicated, but users can be disabled while logged in:

USER = USER[False][False],

USER[enabled:Bool][logged:Bool]

= ( when (!enabled) enable -> USER[True][logged]

| when (enabled && !logged) login -> USER[enabled][True]

| when (logged) logout -> USER[enabled][False]

| when (enabled) disable -> USER[False][logged] ).

-------

const MAX\_USER = 4

const N = 5

range U = 1..N

const False = 0

const True = 1

range Bool = False..True

property ONLY\_FOUR = ONLY\_FOUR[0],

ONLY\_FOUR[i:0..MAX\_USER] =

(when (i < MAX\_USER) login[u:U]->ONLY\_FOUR[i + 1]

|when (i > 0) logout[u:U]->ONLY\_FOUR[i - 1]).

property LOGIN\_WHEN\_ENABLED(ID=0) = DISABLED\_USER,

DISABLED\_USER = (enable[ID]->ENABLED\_USER),

ENABLED\_USER =

(login[ID]->LOGGED\_IN\_USER | disable[ID]->DISABLED\_USER),

LOGGED\_IN\_USER =

(logout[ID]->ENABLED\_USER |disable[ID]->logout[ID]->DISABLED\_USER).

||SYSTEM = (ONLY\_FOUR || forall [u:U] LOGIN\_WHEN\_ENABLED(u)).

The following answer can be tested in LTSA:

const N = 5

range U = 1..N

property ONLY\_FOUR = ONLY\_FOUR[0],

ONLY\_FOUR[c:0..4] = (when (c<4) login -> ONLY\_FOUR[c+1]

|when (c>0) logout -> ONLY\_FOUR[c-1]).

USER\_PROTOCOL = (enable -> ENABLED),

ENABLED = (login -> logout -> ENABLED

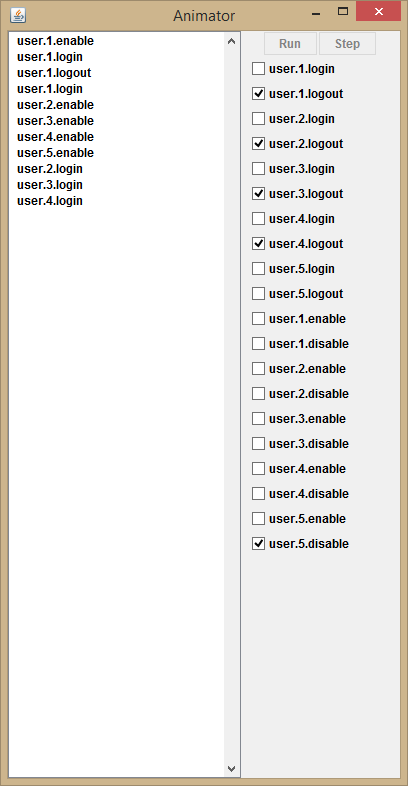
|disable -> USER\_PROTOCOL).

property ||LOGIN\_WHEN\_ENABLED = (user[u:U]:USER\_PROTOCOL).

||SYSTEM = ({user[u:U]}::ONLY\_FOUR || LOGIN\_WHEN\_ENABLED).

The users can now login more than more while enabled, and the question asks “Specify the two requirements above using safety properties” so we use the property keyword for the “ONLY\_FOUR” and “LOGIN\_WHEN\_ENABLED” systems.

If you want to see it actually working in LTSA then you can remove the “property” keywords, compose to create the SYSTEM then run any trace. Leaving “property” in will let you perform any action but will give you ERROR in the trace when you perform an action not allowed by the properties.



Can’t login with user.5.

2a

i).

Monitor is passive synchronisation primitive which responds to (input) actions and provides condition synchronisation.

In Java, monitors are implemented as objects with synchronized methods which use a while loop and wait() which puts a thread to the object’s wait set and changes in the state of the monitor are signalled to the waiting threads using notify() and notifyAll() which remove threads from the object’s wait set.

ii).

Wait set of a Java object is a set of all Threads that called wait() while executing that object’s method. These Threads cannot be run until removed from this set by another thread calling notify() or notifyAll() while executing a method of that object.

iii.)

Notify() chooses randomly (non-deterministically) which thread it removes from the wait set. Therefore, if a thread waiting for a specific condition is awakened, it will just call wait() again, possibly causing a deadlock. However, if notifyAll() is used instead, all the threads in the wait set test whether the condition they have been waiting for is satisfied or not and add themselves back to the wait set or continue their execution.

2b

i).

Kinda works, if you do tick before run (could be enforced by prioritisation (or not))

TIMER = TIMER[0],

TIMER[t:TIME] = ( when (t<MAX-1) tick -> TIMER[t+1]

| when (t==MAX-1) tick -> TIMER[0]

| at[t] -> TIMER[t]).

ii).

tick -> tick -> a.at.2 -> tick -> tick -> tick -> tick -> tick

-> b.at.7 -> tick -> tick -> tick -> b.run -> a.run

iii).

progress RunA = {a.run}

progress RunB = {b.run}