LICS(CS F214) Assignment

Akshay Shukla 2022A7PS0087P

Gobind Singh 2022A7PS0083P

Siddhartha Gotur 2022A7PS0070P

Sriram Hebbale 2022A7PS0147P

Granth Jain 2022A7PS0172P

Assumptions:

1. The part of the model ensuring liveliness does not have any synchronous statements involved and hence UPPAAL does not display anything for the initial channels in the Symbolic Simulator. It can be understood as a situation where the user is waiting to enter the critical state. The symbolic simulator can be followed to see the change in states for each of the users during this period.
2. The library and the bank combined is a shared resource that is used by a single user at a time.
3. There are only two users that can run simultaneously in this model.
4. The users initially have a bank balance of 1000.
5. We redirect the user to sign out after the user views his membership status, tries to buy a card but has insufficient balance, or has a successful transaction.
6. We embedded membership to be '1' for a gold card and '2' for a platinum card. This has been done to make the system autonomous.

7. If the user already has a membership, they cannot buy the same membership again but they have the liberty to buy the other one. E.g: if a user has a gold membership, he cannot buy another gold card but can buy a platinum card and the membership will update accordingly.
8. The bank sends the transaction_completed action which needs to be clicked on in the model before proceeding to the next critical state (it won't cause a deadlock though)
9. The price of platinum is 200 and gold is 100. They can be given any arbitrary value with the constraint that the price of platinum is higher than gold.
10. The critical section is called CS and comprises the library and the bank.
11. The initial user_id is 1 which is anyways updated as either user enters the critical section.
12. In the first iteration of the model, whichever user reaches the 'wait' state first, is permitted to pass through the critical state. Whereas the other user is made to wait at the 'wait' state till the other user has 'signed_out'.
13. If one user has either reached the wait state or has proceeded further while the other user is in the 'want' state, then the transition of the second user from 'want' to 'wait' state is given higher priority.

Our Model:

Our model comprises three systems namely: *user, library*, and *bank*.
The library and bank are critical systems in which only one user can access it at a time. In the model, we have created the scenario for two users by instantiating the User in the System Declarations as 'user1' and 'user2'. The library is named 'lib' and the bank is named 'bank'. We have now declared memberships and amounts as arrays with user_id as a global variable controlling the two users. The user_id =1 corresponds to user1 and user_id = 2 corresponds to user2 and this is updated right when the user enters the critical section.

Basic skeleton for the model:

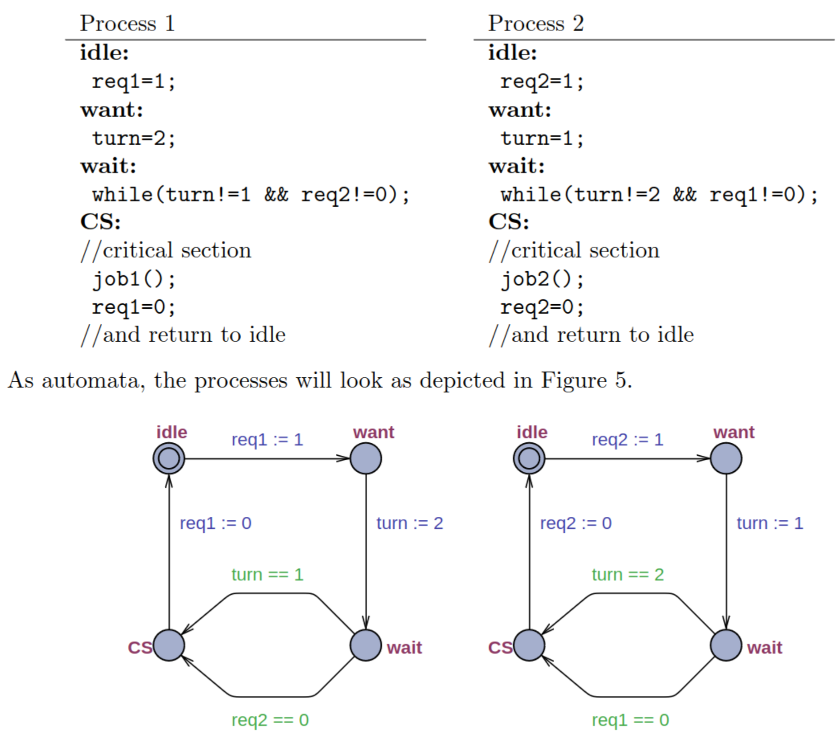
Initially for a user, the model starts with the state 'signed_out' where either of the users can attempt to enter the critical section. When a user enters the critical section, he enters his email ID which gets authorized by the library template. If the email ID is not authorized, the user is redirected to the signed_out state. If authorized, the user enters the signed_in state and the library enters its active state (initial state). As mentioned in the question, the user has two options:

- To view his membership details:
  The library enters the library_view state after which the user is redirected to the signed_out state.

- To buy a new membership: to buy a new membership, the user sends a request to the bank system to buy either a Gold card or a Platinum Card. Now, based on the user's bank balance which is stored in an array of size two (for two users), and his current membership status, he is shown options of buying a gold card, platinum card, or an insufficient balance option which redirects him back to the signed_out state. After a successful transaction, the user is also redirected back to the signed_out state, and his bank balance, and membership status (which are part of a global array) are modified accordingly.

How do we handle the synchronization?

We referred to the UPPAAL small tutorial documentation on how to use mutexes and ensure mutual exclusion. Analogous to the algorithm given below, our signed_out state maps to the idle state, and the entire 'User' template maps to the critical sections of the code. This ensures that only one user can access the critical section of the code. Right before the user enters the 'want' state, the request of that user (req_user) is set to '1' and right before going to the 'wait' state, the 'turn' variable is set opposite to the user which calls a request to ensure liveliness. The user will enter the critical section if either the value of its turn is 0 or the request of the other user is 0.



```
Process 1
idle:
 req1=1;
want:
 turn=2;
wait:
 while(turn!=1 && req2!=0);
CS:
//critical section
 job1();
 req1=0;
//and return to idle
```

```
Process 2
idle:
 req2=1;
want:
 turn=1;
wait:
 while(turn!=2 && req1!=0);
CS:
//critical section
 job2();
 req2=0;
//and return to idle
```

As automata, the processes will look as depicted in Figure 5.

Fig 1. The model as shown in the documentation

Our model handles the safety property as only 1 user is allowed to pass through the critical section at a time. If two users try to pass through the critical section at the same time, one user is made to wait at the '**wait**' state until the other user enters the **'signed_out'** state.

A[] not (user1.CS and user2.CS)

Our model handles the liveness property as every time a user is waiting at the 'wait' state it eventually gains access to pass through the critical section. This property is ensured by the condition update between the 'want' and 'wait' state: ***turn: = (me==1?2:1),***
This means that if one user has the chance currently then it passes its chance to the other user using this ternary statement.

A<> ((user1.wait imply (user1.CS)) && (user2.wait imply (user2.CS)))

We also use a command in the verifier to ensure that the model never enters a state of deadlock.

A[] not deadlock
