Concurrency and Distributed Systems (CDS) Assignment

27 November - 1 December 2023

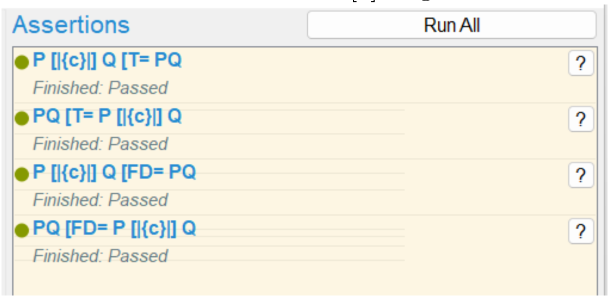
Total Number of Pages: 15

Question 1

(a)

```
PQ =
    let
        P1Q1 =
            (a -> P2Q1)
            []
            (STOP
            |~|
            (b -> P1Q2))
        P1Q2 =
            (a -> P2Q2)
            []
            (d -> P1Q1)
        P2Q1 =
            ((b -> P2Q2)
            |~|
            (c -> P2Q2))
            []
            (e -> P1Q1)
        P2Q2 =
            (e -> P1Q2)
            []
            (d -> P2Q1)
    within
        P1Q1

assert P [| {c} |] Q [T= PQ
assert PQ [T= P [| {c} |] Q
assert P [| {c} |] Q [FD= PQ
assert PQ [FD= P [| {c} |] Q
```

We check the above code in FDR[1] to get

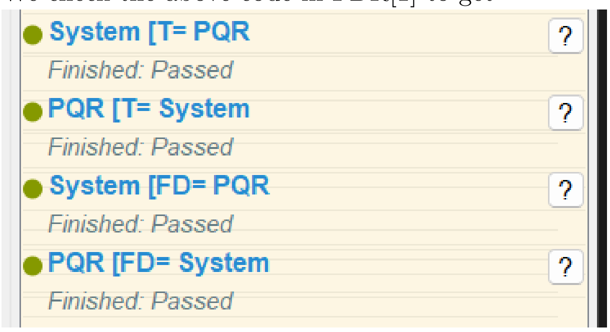


(b)

```
PQR =
    let
        P1Q1R1 =
            (a -> P2Q1R1)
            []
            (STOP
            |~|
            (b -> P1Q2R1))
        P1Q1R2 =
            (a -> P2Q1R2)
```

```
                []
                (STOP
                |~|
                (b -> P1Q2R2))
                []
                (f -> P1Q1R1)
            P1Q2R1 =
                (a -> P2Q2R1)
                []
                (d -> P1Q1R2)
            P1Q2R2 =
                (a -> P2Q2R2)
                []
                (f -> P1Q2R1)
            P2Q1R1 =
                ((b -> P2Q2R1)
                |~|
                (c -> P2Q2R1))
                []
                (e -> P1Q1R2)
            P2Q1R2 =
                ((b -> P2Q2R2)
                |~|
                (c -> P2Q2R2))
                []
                (f -> P2Q1R1)
            P2Q2R1 =
                (e -> P1Q2R2)
                []
                (d -> P2Q1R2)
            P2Q2R2 =
                (f -> P2Q2R1)
        within
            P1Q1R1

assert System [T= PQR
assert PQR [T= System
assert System [FD= PQR
assert PQR [FD= System
```

We check the above code in FDR[1] to get



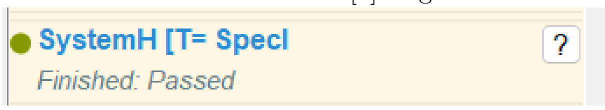
(c)

(i)

```
SpecI =
```

```
    let
        AB =
            (a -> F)
            []
            (b -> F)
        F =
            (f -> AB)
    within
        AB

assert SystemH [T= SpecI
```

We check the above code in FDR[1] to get



(ii)

```
SpecII =
    let
        AB =
            (a -> (f -> AB) |~| AB)
            |~|
            (b -> (f -> AB) |~| AB)
    within
        AB

assert SpecII [FD= SystemH
```

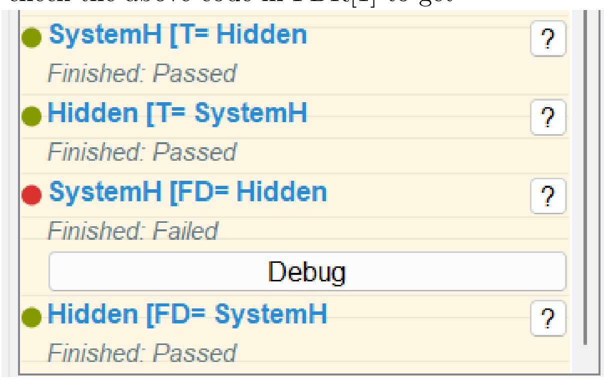
We check the above code in FDR[1] to get

(iii)

```
Hidden =
    let
        P1Q1R1 =
            (a -> P2Q1R1)
            []
            (STOP |~| (b -> P1Q2R1))
        P1Q1R2 =
            (a -> P2Q1R2)
            |~|
            (b -> P1Q2R2)
            []
            (f -> P1Q1R1)
        P1Q2R1 =
            (a -> P2Q2R1)
            []
            (STOP |~| P1Q1R2)
        P1Q2R2 =
            (a -> P2Q2R2)
            []
            (f -> P1Q2R1)
```

```
        P2Q1R1 =
            ((b -> P2Q2R1)
            |~|
            P2Q2R1)
            []
            (STOP |~| P1Q1R2)
        P2Q1R2 =
            ((b -> P2Q2R2)
            |~|
            P2Q2R2)
            []
            (f -> P2Q1R1)
        P2Q2R1 =
            (a -> P2Q2R2)
            []
            (f -> P1Q2R1)
            |~|
            ((b -> P2Q2R2)
            |~|
            P2Q2R2)
            []
            (f -> P2Q1R1)
        P2Q2R2 =
            (f -> P2Q2R1)
    within
        P1Q1R1

assert SystemH [T= Hidden
assert Hidden [T= SystemH
assert SystemH [FD= Hidden
assert Hidden [FD= SystemH
```

We check the above code in FDR[1] to get



where unfortunately although the traces match, the failure divergences do not in the case that SystemH [FD= Hidden. This means that the Hidden implementation has some failures that are not specified in SystemH[2]. I did try deriving bottom up from the definition of hiding for each component P, Q and R and then from their combining them but I found that this does not compile as there where too many hidden transitions (see attached q1.csp)

Question 2

(a)

The role of the parameters

- p = This is the proposed Place a or b

- S = This is the set of friends who you are waiting to get their votes from. This initially is set to all friends excluding oneself and then as each friend votes they are removed from the set preventing double-counting.
- A = This is a rolling count of the number of votes for place a
- B = This is a rolling count of the number of votes for place b

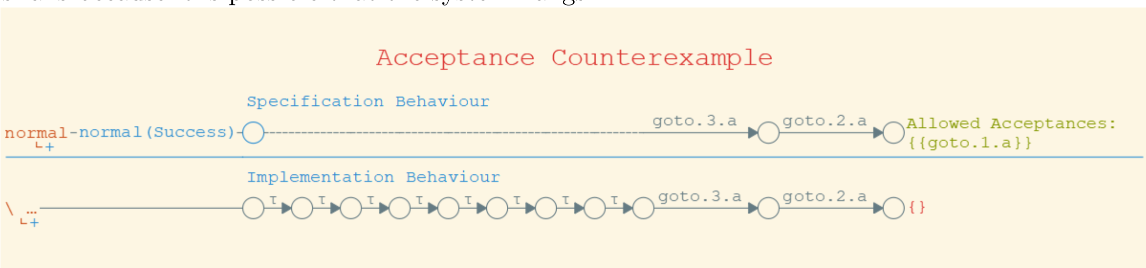
The reason for adding the A + B <= 3 constraint is just to prevent the combinatorial explosion in states that need to be considered. Currently if there are 3 votes (i.e. A + B = 3) then we still allow an additional vote which would result in 4 votes which in our current 3 friend setup does not make sense as each person can only vote once. So actually the same effect could be achieved with A + B <= 2 condition.

No it does not matter that propose.i.i is included in the alphabet of Good(i) because we only call Propose for Others which we have defined such that is never includes itself.

We define success such that

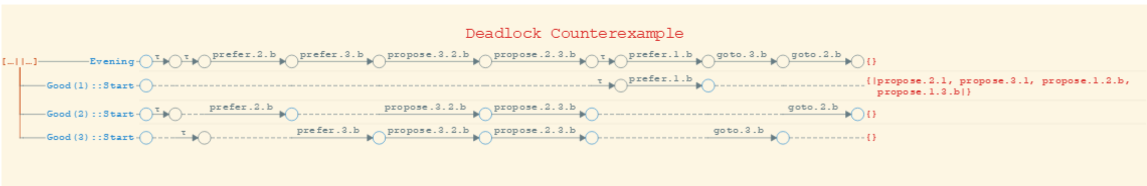
```
success(p) =
    ||| i : Three @ goto.i.p -> STOP

Success =
    success(a)
    |~|
    success(b)
```

This fails because it is possible that the system hangs



And in fact we can verify that we can get deadlock with the assertion

```
assert Evening :[deadlock free]
```

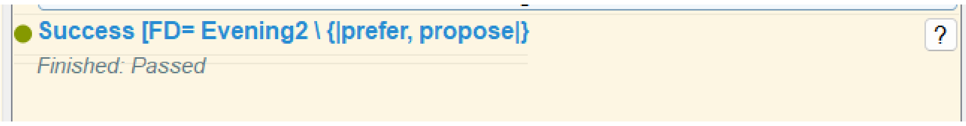


We have three separate states to consider

- After the first two prefers we have 3 with A = 1 and 1 with A = 1
- 1 proposes A to 3, which updates 3 so that it has A = 2 and 3 goes to A
- Concurrently 2 prefers A and 1 proposes A to 2 as well so A = 2 and 2 goes to A
- But this means that as neither 2 or 3 proposed A to 1, 1 is stuck in deadlock with A = 1 still.

Where 3 and 1 prefer a , 1 proposes to 3 and 2 with A = 1 and B = 0. In between, 2 also prefers A and so its state becomes A = 2 which means A stops.

(b)



The problem we had in the previous section is that it was possible for

- Some processes to be waiting to send their proposal
- And for some processes to be waiting to receive other proposals

We resolve this issue with the following guards where the empty(S) condition requires each process to have sent their proposal to all other processes and the A + B == 3 condition requires each process to have received a proposal from all other processes before they can go to a destination.

```
empty(S) & A + B == 3 & A >= 2 & goto.i.a -> STOP
[]
empty(S) & A + B == 3 & B >= 2 & goto.i.b -> STOP
```

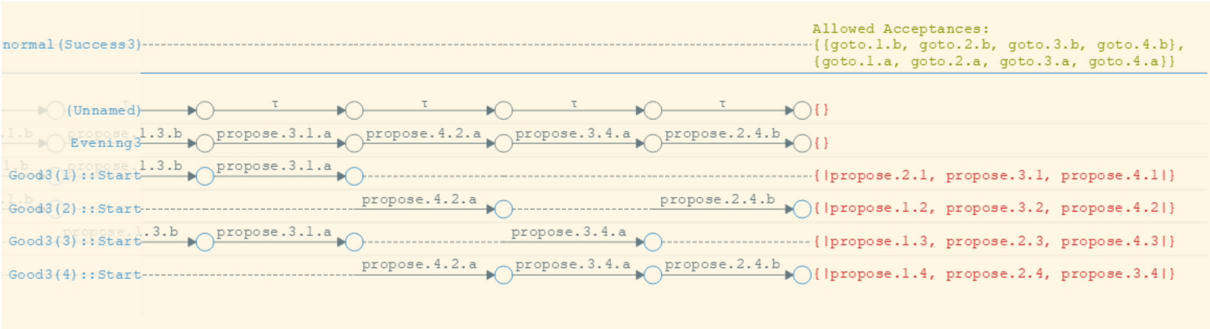
(c)

The updated version Good3 adjust for four friends and that a majority would require 3 of them to vote for the same place.

```
Good3(i) =
    let
        Others = diff(Friend,{i})

        Start =
            prefer.i.a -> Propose(a,Others,1,0)
            |~|
            prefer.i.b -> Propose(b,Others,0,1)

        Propose(p,S,A,B) =
            ( [] j : S @ propose.i.j.p ->
            Propose(p,diff(S,{j}),A,B) )
            []
            ( A + B <= 4 & [] j : Others @ propose.j.i.a ->
            Propose(p,S,A+1,B) )
            []
            ( A + B <= 4 & [] j : Others @ propose.j.i.b ->
            Propose(p,S,A,B+1) )
            []
            empty(S) & A + B == 4 & A >= 3 & goto.i.a -> STOP
            []
            empty(S) & A + B == 4 & B >= 3 & goto.i.b -> STOP
    within
        Start

Evening3 =
    || i : Friend @ [ Alpha(i) ] Good3(i)

success3(p) =
    ||| i : Friend @ goto.i.p -> STOP

Success3 =
    success3(a)
    |~|
```

```
    success3(b)

assert Success3 [FD= Evening3 \ {| prefer, propose |}
```

However, the above refinement check fails because you could have a situation where two friends prefer A and the other two friends prefer B, resulting in deadlock. This can be seen in the counter-example below where 1 and 2 prefer A and 3 and 4 prefer B.



(d)

The updated code is as follows

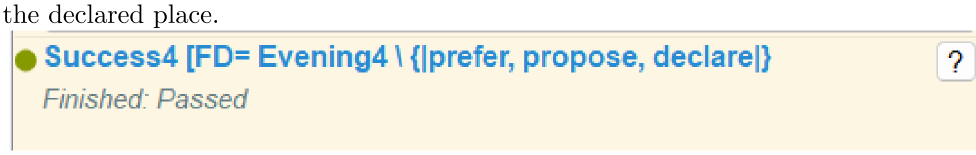
```
channel declare : Friend . Friend . Place

Good4(i) =
    let
        Others = diff(Friend,{i})
        Start =
            prefer.i.a -> Propose(a,Others,1,0,Others)
            |~|
            prefer.i.b -> Propose(b,Others,0,1,Others)

        Propose(p,S,A,B,D) =
            -- send proposals
            ( [] j : S @ propose.i.j.p ->
            Propose(p, diff(S,{j}),A,B,D) )
            []
            -- receive proposals
            ( A + B < 4 & [] j : Others @ propose.j.i.a ->
            Propose(p,S,A+1,B,D) )
            []
            ( A + B < 4 & [] j : Others @ propose.j.i.b ->
            Propose(p,S,A,B+1,D) )
            []
            -- go to destination
            (empty(S) & A + B == 4 & A >= 3 & goto.i.a -> STOP)
            []
            (empty(S) & A + B == 4 & B >= 3 & goto.i.b -> STOP)
            []
            -- send declaration
            ( i == 1 & A == 2 & B == 2 & [] j : D @ declare.i.j.p ->
            Propose(p,S,A,B, diff(D,{j})))
            []
            ( i == 1 & empty(D) & goto.i.p -> STOP)
            []
            -- receive declarations
            ( i != 1 & declare.1.i.a -> goto.i.a -> STOP)
            []
```

```
            ( i != 1 & declare.1.i.b -> goto.i.b -> STOP)
    within
        Start

proposals4(i) =
    union ( {| propose.i |}, {| propose.j.i | j <- Friend |} )
declarations(i) =
    union ( {| declare.i |}, {| declare.j.i | j <- Friend |} )
Alpha4(i) =
    union ( union( proposals4(i), declarations(i)) , {| prefer.i, goto.i |} )
Evening4 =
    || i : Friend @ [ Alpha4(i) ] Good4(i)

success4(p) =
    ||| i : Friend @ goto.i.p -> STOP
Success4 =
    success4(a)
    |~|
    success4(b)

assert Success4 [FD= Evening4 \ {| prefer, propose, declare |}
```

This now passes where the key bit of the new logic is the send and receive declarations parts.

We update the definition of Propose(p,S,A,B,D) to include a new argument D which much like S acts as a way to track whether all other friends have been notified about the decision to declare by friend 1 preventing them from going to the destination until they have done this. We only trigger this scenario in player 1 if we have the draw scenario where A = 2 and B = 2.

On the receiving side, even if not all the proposals have been sent yet (although note friend 1 must have received proposals from everyone for the A = 2 and B = 2 condition to be triggered) we still advance to the declared place.



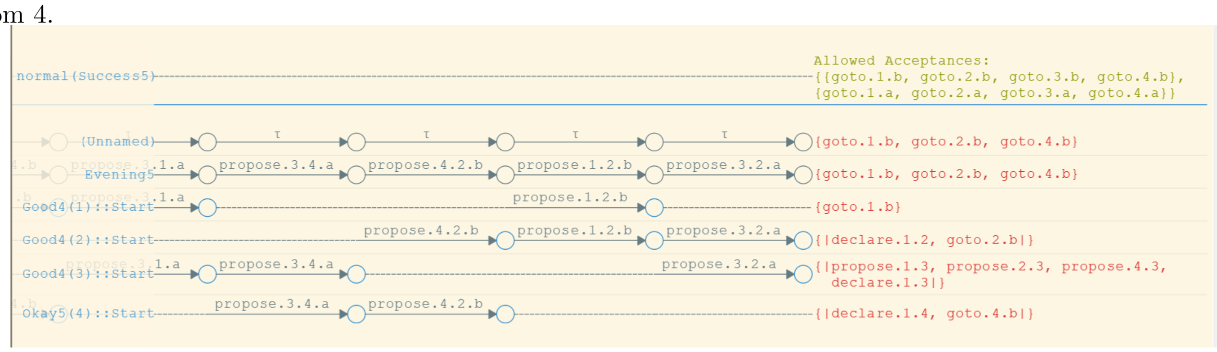
(e)

I tried updating the choice to but there were two many choices to consider so it would not compile and my computer crashes

```
( [] j : S @ (Propose(p, diff(S,{j}),A,B,D)) |~|
(propose.i.j.p -> Propose(p, diff(S,{j}),A,B,D)) )
```

Thus as an alternative I determinstically consider the case where the okay friend does not update player 3 by defining Others = {1, 2} for Okay5

This unsurprisingly generates the failure where 1, 2 and 4 go to a but 3 is still waiting for a proposal from 4.



(f)

As suggested in the question we add a new Time6 process which updates such that when all friends have been updated about the time we then increment the time and again loop through all the friends updating them until we get to 7.

```
Time = {5..7}

channel time : Friend . Time
...
Time6(start) =
    let
        Time(S, t) =
            ([] j : S @ time.j.t -> Time(diff(S,{j}), t))
            []
            (empty(S) & t < 7 & time.1.(t+1)-> Time(diff(Friend,{1}), t+1))
            []
            (empty(S) & t == 7 & STOP)
    within
        Time(Friend, start)
```

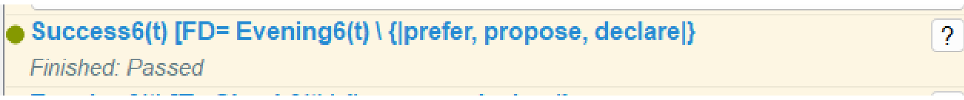
In order to receive these time messages we update Propose) such that we can receive the times as they come in and also

```
 Propose(p,S,A,B,D,T) =
        -- send proposals
        ( T < 6 & [] j : S @ propose.i.j.p ->
        Propose(p,diff(S,{j}),A,B,D,T) )
        []
        -- receive proposals
        ( T < 6 & A + B < 4 & [] j : Others @ propose.j.i.a ->
        Propose(p,S,A+1,B,D,T) )
        []
        ( T < 6 & A + B < 4 & [] j : Others @ propose.j.i.b ->
        Propose(p,S,A,B+1,D,T) )
        []
        -- go to destination
        (T == 7 & empty(S) & A + B == 4 & A >= 3 & goto.i.a -> STOP)
        []
        (T == 7 & empty(S) & A + B == 4 & B >= 3 & goto.i.b -> STOP)
        []
        ( T == 7 & A >= 3 & goto.i.a -> STOP)
        []
        ( T == 7 & B >= 3 & goto.i.b -> STOP)
        []
        -- send declaration
        ( i == 1 & A == 2 & B == 2 & [] j : D @ declare.i.j.p ->
        Propose(p,S,A,B,diff(D,{j}),T))
        []
        ( i == 1 & T == 6 & [] j : D @ declare.i.j.p ->
        Propose(p,S,A,B,diff(D,{j}),T))
        []
        (T == 7 & i == 1 & empty(D) & goto.i.p -> STOP)
        []
        -- receive declarations
        (T == 7 & i != 1 & declare.1.i.a -> goto.i.a -> STOP)
        []
        (T == 7 & i != 1 & declare.1.i.b -> goto.i.b -> STOP)
        []
        -- receive time
```

```
        (T == 5 & time.i.6 -> Propose(p, S, A, B, D, 6))
        []
        (T == 6 & time.i.7 -> Propose(p, S, A, B, D, 7))
```

In order, to check whether the correct behaviour occurs we update our Success definition such that it includes taking time updates

```
Success6P(i, p, T) =
    ( T == 7 & goto.i.p -> STOP)
    []
    ( T == 5 & time.i.6 -> Success6P(i, p, 6))
    []
    ( T == 6 & time.i.7 -> Success6P(i, p, 7))
success6(p, T) =
    ||| i : Friend @ Success6P(i, p, T)
Success6(t) =
    (success6(a,t)
    |~|
    success6(b,t))
    [|timeAlpha|] Time6(t)

t = 6
assert Success6(t) [FD= Evening6(t) \ {| prefer, propose, declare |}
```



In order to confirm that our above success is not just defaulting to the four friends go to wherever friend 1 prefers we create a new process which is aware of the friends preferences

```
Check6P(i, A, B, p, T) =
    ( T == 5 & time.i.6 -> Check6P(i, A, B, p, 6))
    []
    ( T == 6 & time.i.7 -> Check6P(i, A, B, p, 7))
    []
    ( T == 7 & A >= 3 & goto.i.a -> STOP)
    []
    ( T == 7 & B >= 3 & goto.i.b -> STOP)
    []
    ( T == 7 & A < 3 & B < 3 & goto.i.p -> STOP)
    []
    -- receive preferences
    ( T < 7 & [] j : Friend @ prefer.j.a ->
    Check6P(i, A + 1, B, p, T))
    []
    ( T < 7 & [] j : Friend @ prefer.j.b ->
    Check6P(i, A, B + 1, p, T))
check6(p, T) =
    ||| i : Friend @ Check6P(i, 0, 0, p, T)
Check6(t) =
    (check6(a,t)
    |~|
    check6(b,t))
    [|timeAlpha|] Time6(t)
```

In particular, it receives messages about A and B and requires that if the votes for either are above 3 then we should go to that destination i.e. friend 1's preferences cannot override the majority. This restricts the possible successful outcomes where we want the implementation's good outcomes to be a subset of the specifications good outcomes thus we check

```
assert Check6(t) [T= Evening6(t) \ {| propose, declare |}
```

However, unfortunately again my computer cannot handle running this many processes so I cannot confirm this part of the code.

Question 3

(a)

There are a few possible candidates that we could choose as the value for maxnum

- ¡ 3 = We could choose a number less than 3 but this would mean that of our three customers two may have the same number. Consider both getting the number 0 which would mean that one of the customers would go to the clerk and be served but the other would have to wait as the number would increment to 1. Given that presumably we want our queue to retain the ordering of the tickets and that it would not be fair for the third customer to get a 0 and then be served before this is not a good choice. It is worth noting however, that given we have two desks it is possible that customer number 2 is served after customer number 3 because 1 could be processed very quickly followed by 3 whilst 2 has a slower process
- 3 = If we choose 3 as the maxnum then by the definition of Number 0, 1 and 2 are possible numbers which means that each of the three customers can have their own number fulfilling the uniqueness criteria
- ¿ 3 = We could also have more than three tickets but given that we have a maximum of 3 customers there is no need for more to preserve uniqueness. From a modelling stand-point we would also have to consider more possibilities.

(b)

This is my display code

```
Display =
    let
        Blank =
            next.A -> Screen(0, A)
            []
            next.B -> Screen(0, B)
        Screen(n, l) =
            see.n.l ->
                if n < maxnum - 1 then
                    next.A -> Screen(n + 1, A)
                    []
                    next.B -> Screen(n + 1, B)
                else
                    next.A -> Screen(0, A)
                    []
                    next.B -> Screen(0, B)
    within
        Blank
```

The core of the logic involves updating the display screen to show either desk A or B depending upon which is called in the channel next. This is coupled with logic to increment the number n based upon what is seen on the see channel, making sure to loop back round to 0 again if we are at the maximum.

(c)

The desk logic is

```
let
    PressButton =
        next.l -> SeeDisplay
    SeeDisplay =
        see?n.l -> ReadyCustomer(n)
    ReadyCustomer(n) =
        present.l?c.n -> ServeCustomer(c)
        []
        return.l?c.n -> PressButton
    ServeCustomer(c) =
        serve.l.c -> PressButton
within
    PressButton
```

Clerks start by pressing the next button, then they see the display to know which customer to serve. Then if the ticket is valid they serve that customer or if it is not they return and press the button again.

(d)

```
Customer(c) =
    let
        Enter =
            enter.c -> GetTicket
        GetTicket =
            ticket?n -> WaitSee(n)
        WaitSee(n) =
            see.n?l -> PresentTicket(l, n)
        PresentTicket(l, n) =
            present.l.c.n -> WaitServe
            []
            return.l.c.n -> WaitSee(n)
        WaitServe =
            serve?l.c -> Leave
        Leave =
            leave.c -> STOP
    within
        Enter
```

The customer cycles through a series of states from entering, to getting a ticket, to waiting to see and to presenting a ticket. The only variation from this linear path occurs when they either present the ticket correctly or have it returned, in the former case they are served and can leave, in the latter case they return to waiting again.

(e)

```
aSystem = {| next, see, present, return, serve |}

System =
    (Ticket ||| Display ||| Desk(A) ||| Desk(B) )
    [| aSystem |]
    (Customer(P) ||| Customer(Q) ||| Customer(R))
```

We create a shared alphabet aSystem which both the visa centre and the customers are aware of. By implication only enter and leave are not shared which are pure customer actions. As suggested in the question we used a shared parallel [|||] to share these components and an interleaving ||| for the independent components

(f)

In order to check that a customer is guaranteed able to leave we create two assertions

```
CustomerService(c) =
    enter.c -> serve?l.c -> leave.c -> CustomerService(c)
AllCustomersServed =
    CustomerService(P) ||| CustomerService(Q) ||| CustomerService(R)

assert System :[deadlock free]
assert AllCustomersServed [T= System \ {| ticket, next, see, present,
return |}
```

The first checks that our System is deadlock free and the second that all customers will be served. We find that both of these cases pass suggesting that we have designed our customer process correctly.

References

[1] University of Oxford. Fdr4 csp refinement checker.

[2] University of Oxford. Concurrency and distributed systems lectures slides nov 2023.