Question 1

Part 2

Assumption: "propagate the latest weather update" means that all connected clients will not only be notified of the latest weather, but **will also receive the latest weather**.

Let us define

- `isLatestSent` – whether the WCP has sent the update message to the CM.
  `#define isLatestSent (cm_chan ? <USER_UPDATED_WEATHER, DUMMY_VAL, DUMMY_VAL>)`
- `are_all_new_successful`- whether all the connected clients reported success for using the new weather.

Then, the LTL property used is:

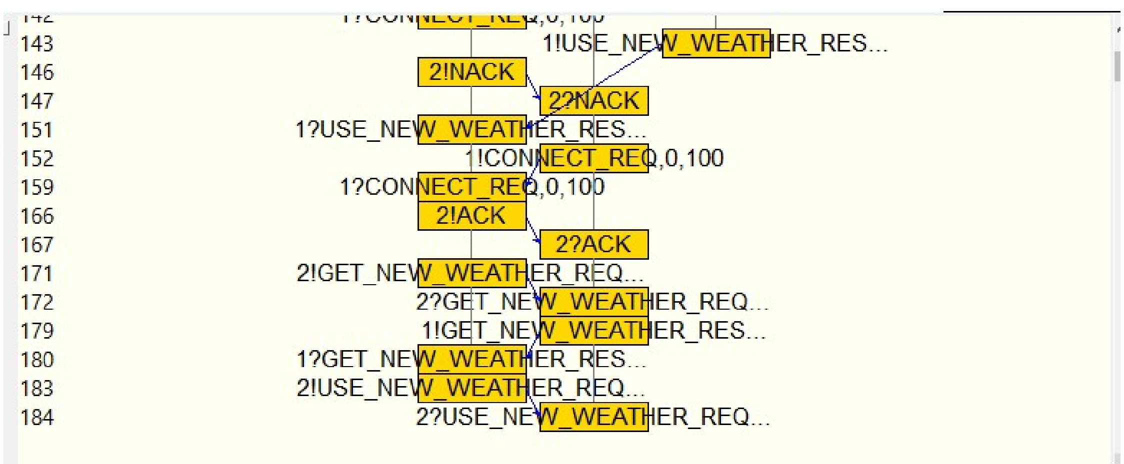
```
[] (isLatestSent -> <> are_all_new_successful )
```

It means:

Globally, if the Communication Manager is sent a weather update message, then eventually, all connected clients will successfully use the new weather (and send the corresponding message to the CM).

Part 3

The exact nature of the deadlock seems to depend on the implementation of the model. In my final implementation, states are written by the CM without using message passing, but can be read by clients and WCP. The resultant counter example encountered is:



The property that should hold is "each receive operation should eventually get a message from the corresponding send operation". This does not hold for the CM.

In the MSC above, the CM sends the `USE_NEW_WEATHER` message to the client (step 183). The client receives the message (step 184) but it is in the `INITIALIZING` state. So, it does not process it, and line 88 of the code below is executed instead of line 95.

The CM is in the `POST_INITIALIZING` state and is waiting for a message from the client, but no response is sent from the client to the CM regarding the success status of using the new weather information (since the client did not process it). This results in a deadlock.

```
  :: client_chan[id] ? req ->
    if
    /* Step A4a. B4a. Client response to Get New Weather info */
    :: (client_status[id] == INITIALIZING  || client_status[id] == UPDATING) -> {
      if :: (req == GET_NEW_WEATHER_REQ) -> {
        set_is_successful();
        cm_chan ! GET_NEW_WEATHER_RESP, id, is_successful;
      }

      :: else ->
        printf("===Error: req=%d (should be 8- GET_NEW_WEATHER_REQ)\n", req);
      fi
    }

    /* Step A5a. B5a. Client response to Use New Weather info */
    :: (client_status[id] == POST_INITIALIZING || client_status[id] == POST_UPDATING) ->
      if :: (req == USE_NEW_WEATHER_REQ) -> {
          set_is_successful();
          cm_chan ! USE_NEW_WEATHER_RESP, id, is_successful;
      }

      :: else
      fi
```

Other Issues with the Protocol

1. In Step 4 of *Client Initialization*, if the client reports failure, the WCP is not re-enabled.
2. The key property mentioned in Part 2 does not hold. During the *Weather Update*, if a connected client reports failure to get the new weather or to use the new weather, they are asked to use the old weather or are disconnected. Hence, there can be scenarios in which the information from the weather update doesn't reach the connected clients.

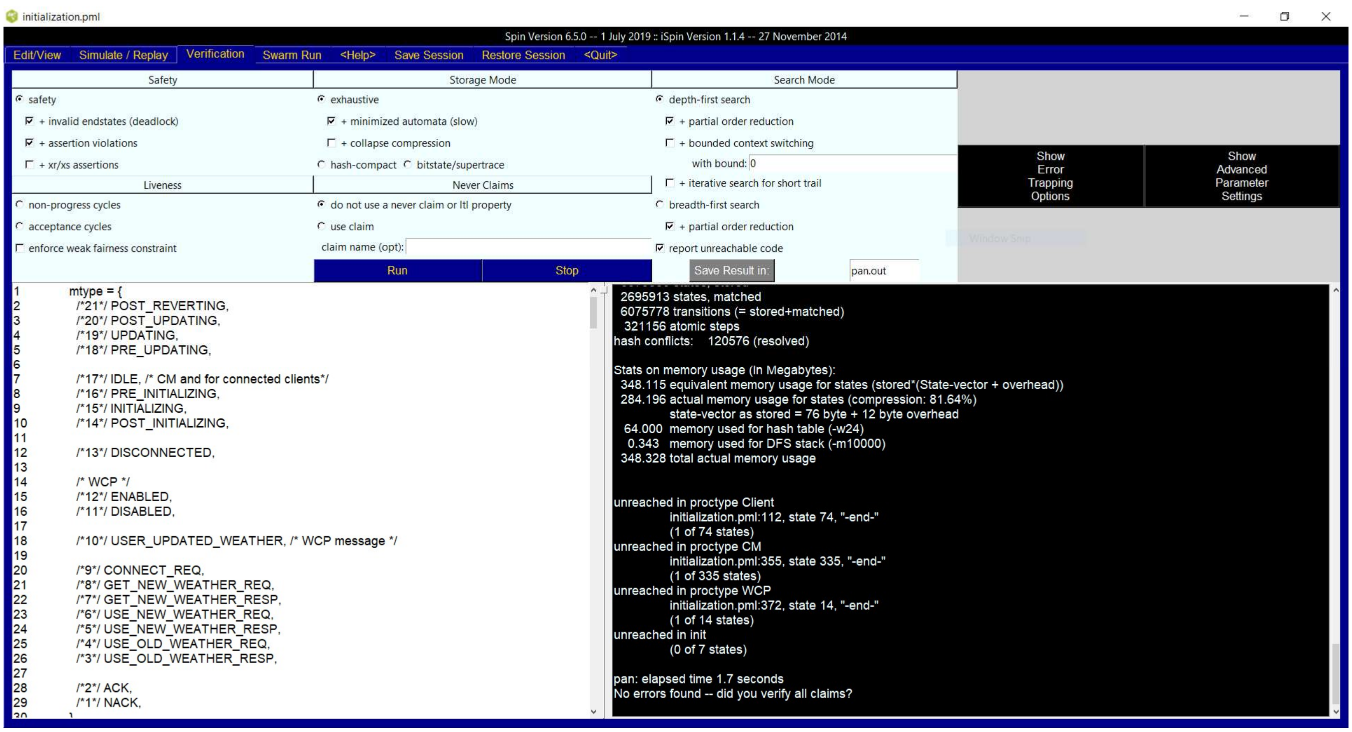
Part 4

The deadlock can be fixed by putting the statements in an atomic block (see lines 217 to 221 below). Now, the client is guaranteed to be in the correct state (`POST_INITIALIZING`) to receive the `USE_NEW_WEATHER_REQ` message.

```
/* Step A4b. CM action to client response for Get New Weather info */
:: (cm_status == INITIALIZING && id == client_id && req == GET_NEW_WEATHER_RESP) ->
  if :: (val == 1) ->
    atomic {
      client_chan[client_id] ! USE_NEW_WEATHER_REQ;
      cm_status = POST_INITIALIZING;
      client_status[client_id] = POST_INITIALIZING;
    }

  :: else ->
    client_status[client_id] = DISCONNECTED;
    cm_status = IDLE;
    client_id = DUMMY_VAL;
    // FIXME: Enable WCP?
  fi
```

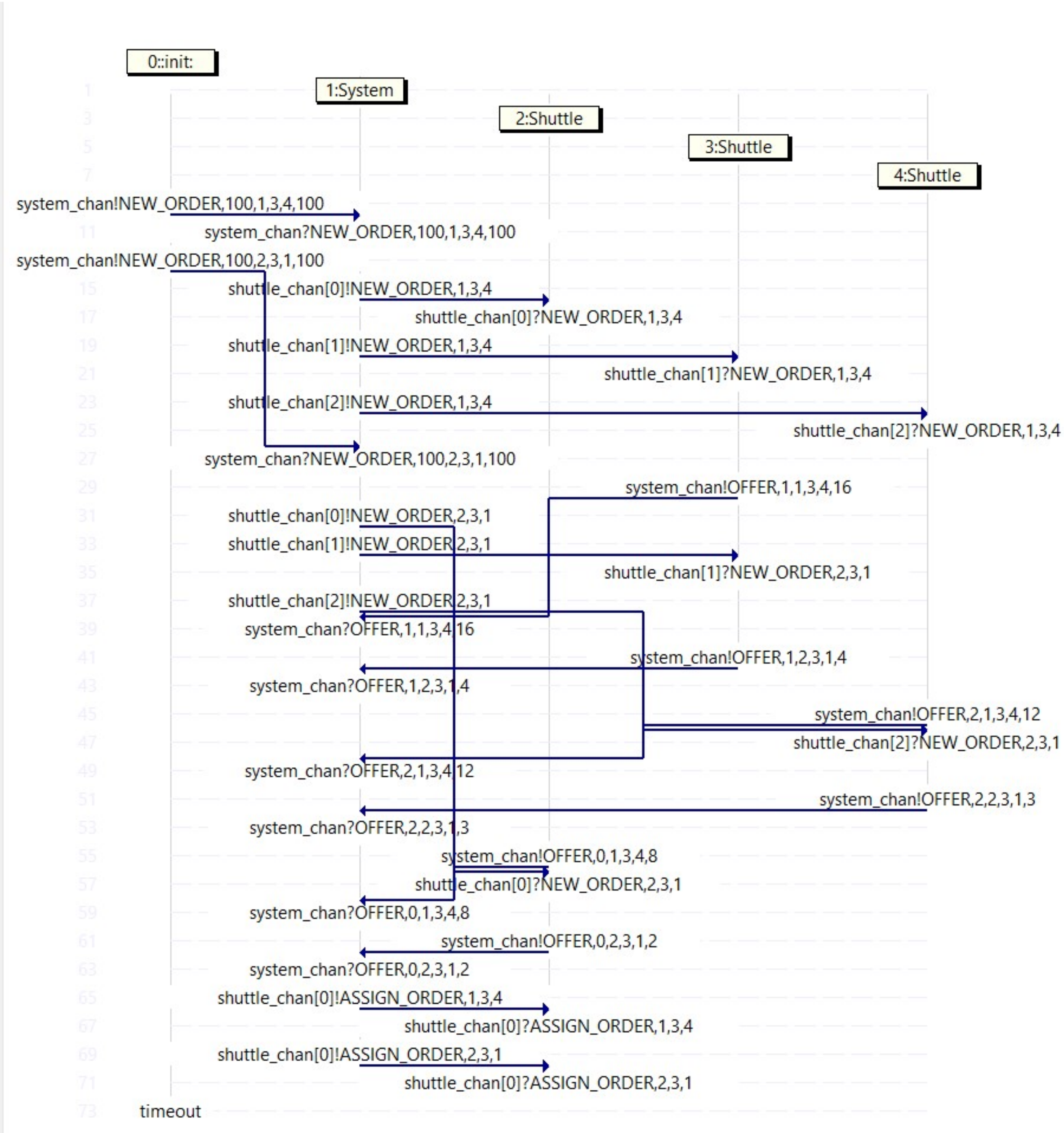
To prove that this fix is sufficient, use the “safety” option in “Verification” tool in iSpin. When we check for deadlocks, the output displays “No errors found”, while previously (in part 3) it would display “To replay the error-trail, goto Simulate/Replay and select "Run"”.



Question 2

Part 2

Using SPIN and running `spin -M railway.pml` gives this Message Sequence Chart that shows one sequence of events that leads to the desired results.



Sequence of events

Assignment of orders:

- Management System receives a `NEW_ORDER` with `start=1, dest=3, size=4` (order 1).
- Management System sends order 1 to the 3 shuttles using the `shuttle_chan[idx]` buffer.
- Management System receives a `NEW_ORDER` with `start=2, dest=3, size=1` (order 2).
- Shuttle 2 sends an offer for order 1 to Management System with `payment=16`.
- Management System sends order 2 to the 3 shuttles using the `shuttle_chan[idx]` buffer.
- Shuttle 2 sends an offer for order 2 to Management System with `payment=4`.
- Shuttle 3 sends an offer for order 1 to Management System with `payment=12`.
- Shuttle 3 sends an offer for order 2 to Management System with `payment=3`.

- Shuttle 1 sends an offer for order 1 to Management System with `payment=8`.
- Shuttle 1 sends an offer for order 2 to Management System with `payment=2`.
- Management System sends an `ASSIGN_ORDER` message to Shuttle 1 for order 1.
- Management System sends an `ASSIGN_ORDER` message to Shuttle 1 for order 2.

Processing of orders by Shuttle 1:

- Shuttle 1 processes the orders one at a time, starting from the earliest one.
- It chooses the initial direction to travel in based on the distance of the route to the start station.
- It then travels in the chosen direction, while checking that the track is used by only one shuttle.
- Once the start station (Station 1 and then Station 2) is reached, it loads the passengers.
  It chooses the direction to travel in based on the distance of the route to the destination station.
- It then travels in the chosen direction, while checking that the track is used by only one shuttle.
- Once the destination station (Station 3 and then Station 3) is reached, it unloads the passengers.

At the end of the execution, the shuttles are:

- Shuttle 1: Stationary at Station 3 with no load
- Shuttle 2: Stationary at Station 1 with no load
- Shuttle 3: Stationary at Station 2 with no load

This can be seen from the Data Window in iSpin after the run terminates. For example, the screenshot below shows the values of the variables in the process corresponding to Shuttle 1 (note the values of `curr_station` , `load` and `status`) after running a simulation with `seed = 123` on iSpin. Similar results are observed for Shuttle 2 and Shuttle 3.

```
:init:(0):o1.start  =  1
:init:(0):o2.dest  =  3
:init:(0):o2.size  =  1
:init:(0):o2.start  =  2
Shuttle(2):are_passengers_loaded  =  0
Shuttle(2):can_travel  =  1
Shuttle(2):capacity  =  5
Shuttle(2):charge  =  2
Shuttle(2):curr_order  =  2
Shuttle(2):curr_station  =  3
Shuttle(2):distance  =  1
Shuttle(2):i  =  3
Shuttle(2):id  =  0
Shuttle(2):idx  =  1
Shuttle(2):j  =  1
Shuttle(2):load  =  0
Shuttle(2):msg  =  ASSIGN_ORDER
Shuttle(2):num_orders  =  2
Shuttle(2):o.dest  =  3
Shuttle(2):o.size  =  1
Shuttle(2):o.start  =  2
Shuttle(2):orders[0].dest  =  3
Shuttle(2):orders[0].size  =  4
Shuttle(2):orders[0].start  =  1
Shuttle(2):orders[1].dest  =  3
Shuttle(2):orders[1].size  =  1
Shuttle(2):orders[1].start  =  2
Shuttle(2):payment  =  2
Shuttle(2):start_station  =  1
Shuttle(2):status  =  STATIONARY
Shuttle(3):are_passengers_loaded  =  0
Shuttle(3):can_travel  =  1
Shuttle(3):capacity  =  8
```