UML LAB 2 MODELING A RAILROAD CROSSING SYSTEM

1 OBJECTIVE

Figure 1- A typical railroad crossing

We want to develop a railroad crossing system.

The railroad crossing system manages three train sensors, a 3-color light (green, amber, red), and a barrier. This system manages a one-way track, with only one lane.

The three train sensors are:

- **Approach**: Signals that a train is getting towards the crossing. The light is set to amber for 2 seconds, then to red. At that moment, barriers are lowered. This process takes 5 seconds.
- **In:** Signals that a train is about to enter the crossing. If the barriers are not yet at their most down position, an error message is sent to the crossing's maintenance headquarters, and a special blinking red light is lit on a panel visible from the train.
- **Leave:** Signals that a train has just left the crossing. The barriers are opened (this process also takes 5 seconds) and the 3-color light is set to green.

Moreover, the system should take the safest decision in all circumstances. This means that if a decision needs to be taken, we assume the system selects the one with maximal safety.

2 REQUIREMENTS

The requirements diagram is as follows.

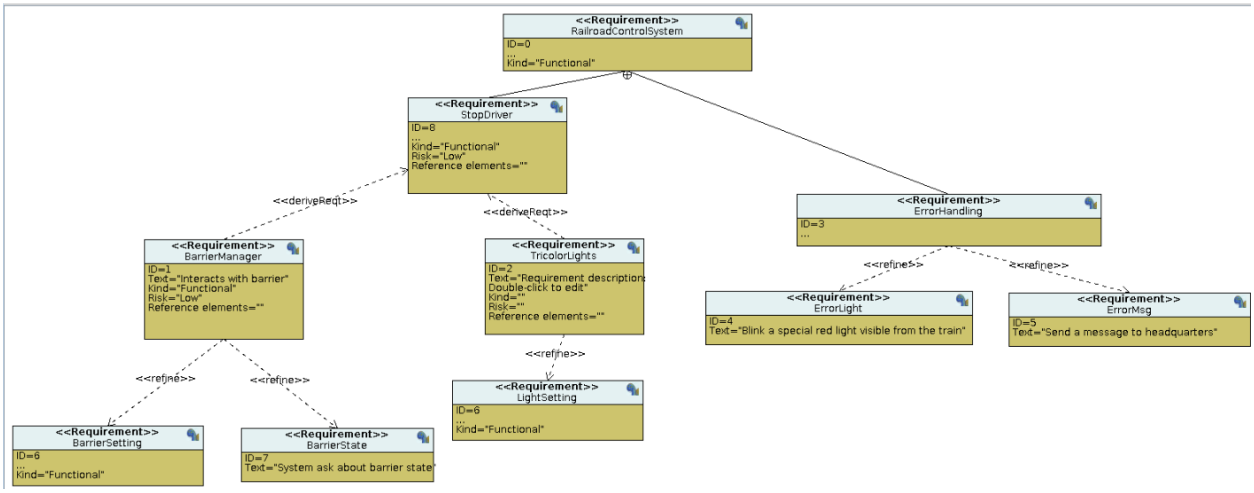

Figure 2- Requirements diagram

In the requirement diagram, we represented the application as a subset of three main requirements. We determined those main requirements using the specifications given above. The barriers and lights are the main actors being driven by the application during a typical usage, therefore they both have their own subset of requirements. Moreover, the specification mentions one specific error case that must be handled, we represented it as a third requirement.

3 USE CASE

The use case diagram is as follows.

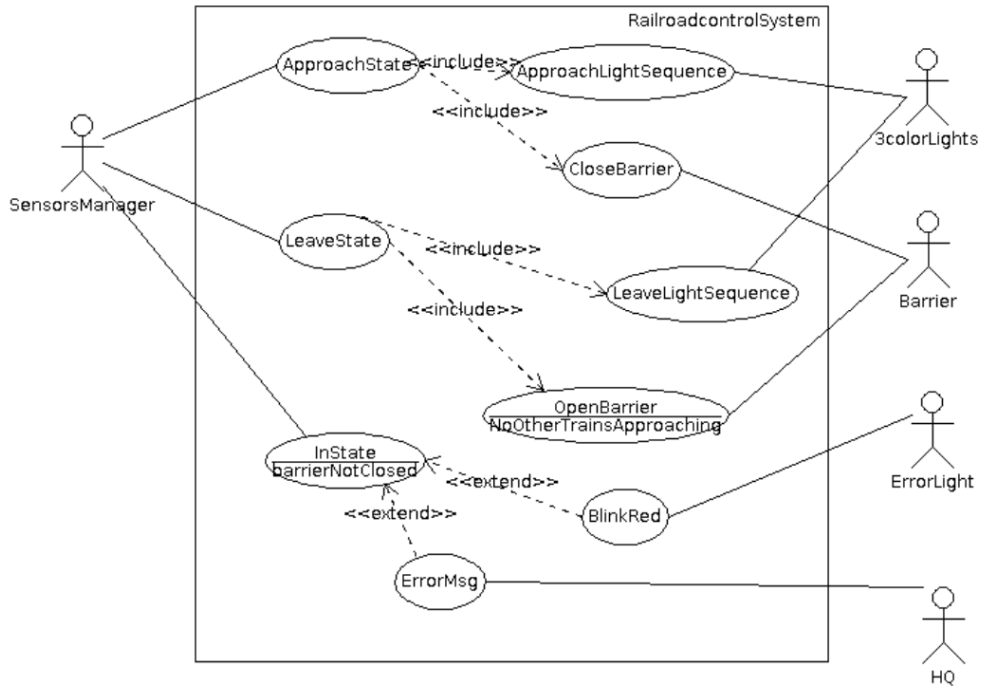

Figure 3- Use case diagram

We decided to represent all the input sensors in one actor "SensorsManager" for better readability. The actors on the left represent the outputs of the system. "ErrorLight" and "HQ" are only used in case of an error represented by the condition "barrierNotClosed". The light sequences and barrier orders are always used, regardless of an error, hence they are "includes", while the "blinkRed" and "errorMsg" cases are only required when an error occurs and are "extends".

4 ACTIVITY DIAGRAM

The following activity diagram is not complete (it's not a problem because it will not be compiled), but it contains the main processes of the application.

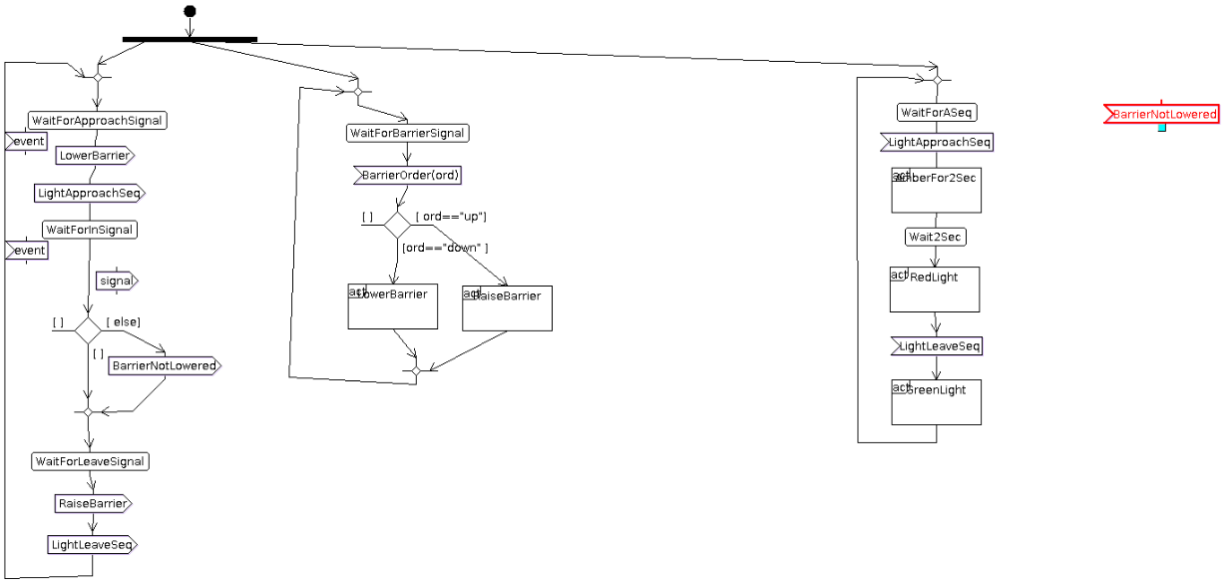

Figure 4- Activity diagram

Our draft activity diagram emphasizes the main processes of the system. The left branch represents the activity of the main controller, it waits for the input signal and starts the corresponding barrier and light processes using internal signals. The center and right branches are processes for the barrier and lights respectively.

Some unconnected signals are present, they indicate parts of the diagram that we could not implement due to a lack of time. This diagram is a draft, but we used it as a base for the block diagram.

5 BLOCK DIAGRAM

For the block diagram, we can take inspiration from the activity diagram and work things from here.

To simulate the train, we add a testbench sending the sensors values after a given time, in the correct order.

The barrier error can be raised randomly by the *s* variable representing the barrier status.

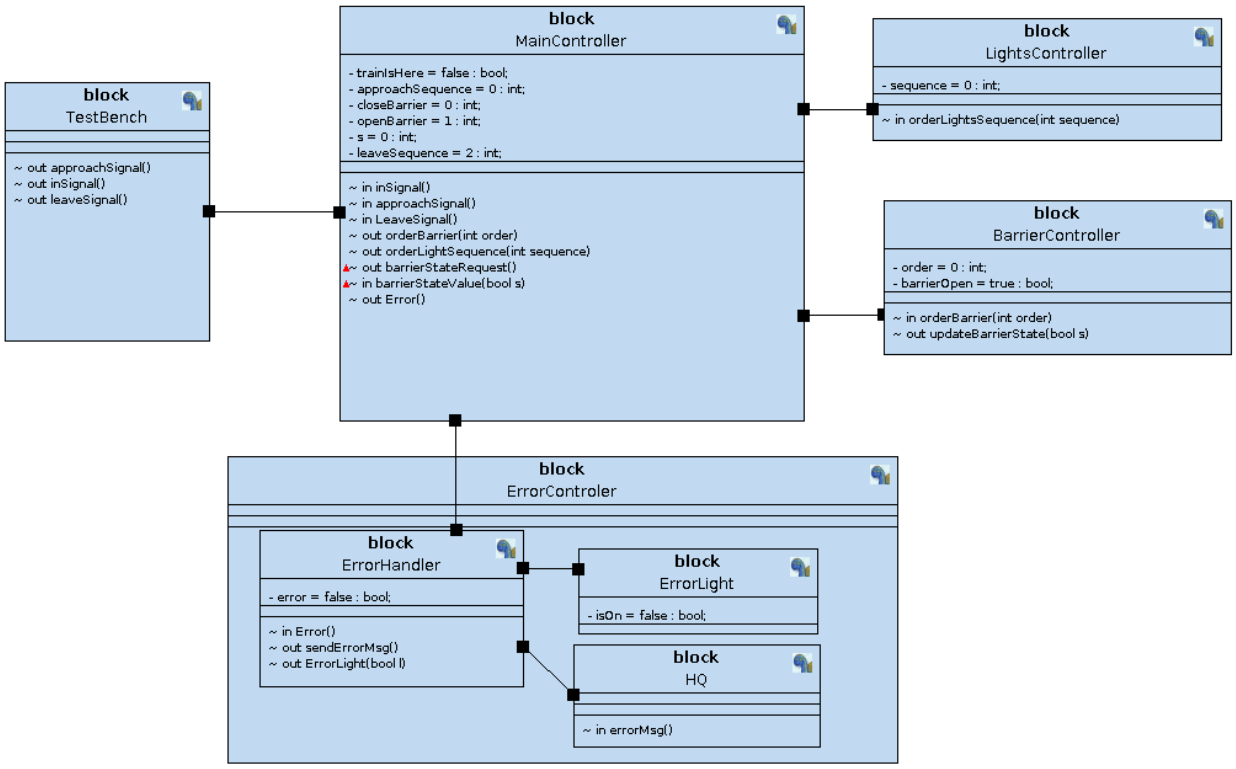

Figure 5- Top level block diagram

The center block is the main controller and acts as a scheduler to drive the three side blocks "LightsController", "BarrierController", and "ErrorController". The three side blocks correspond to the three main parts of the system defined in the requirement diagram.

The test bench represents an expected sequence for the input signals (approach, in, and leave).

The "LightsController" and "BarrierController" blocks receive orders from the main controller and execute a sequence of activities accordingly. However, they do not send any feedback about their state to the main controller. A more refined system would implement a state checking system to check for possible errors in the execution of the orders. For now, our system simulates an error in the state of the barrier using a random variable in the main controller.

The "ErrorController" block triggers an error sequence when it receives the error signal. This block is called by default whenever an error is handled by the system.

The main controller schedules the operations depending on the signals received. Following the specifications, the main controller handled the case when an "in" signal is received but the barriers are not completely lowered. This error case is represented by the "s" random variable. Another error is handled, one that is not mentioned is the specifications. It is the case where a second train arrives while the previous one is still near the barrier. In that case the error sequence is called (blinking red light and a message to HQ) and the barrier is lowered, we considered this sequence to be "the safest" decision to take in that case.

We choose not to set any time limit for the error state, as we don't want to make any assumptions concerning the way the error is handled by the Headquarter.

Below are the diagrams for each of these blocks.

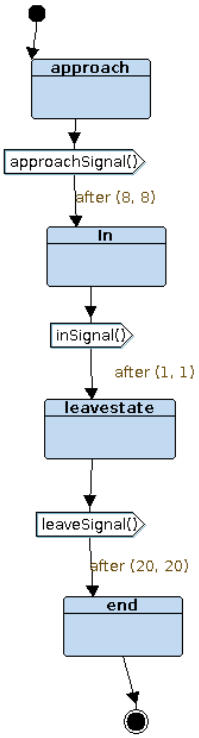

Figure 6- Test bench

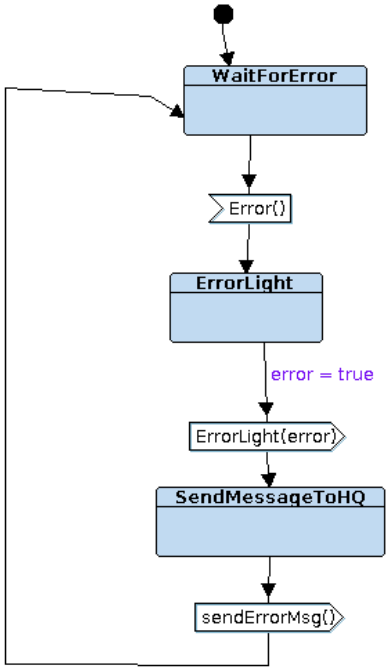

Figure 7- Error handling

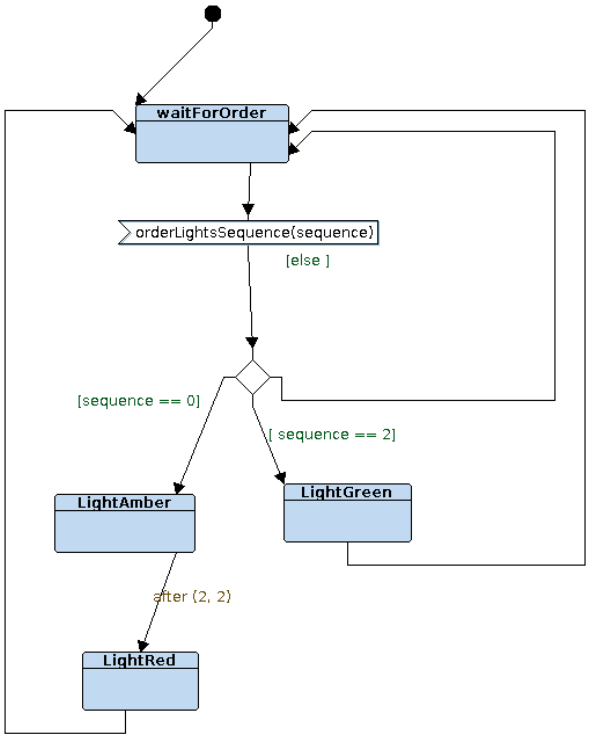

Figure 8- Lights controller diagram

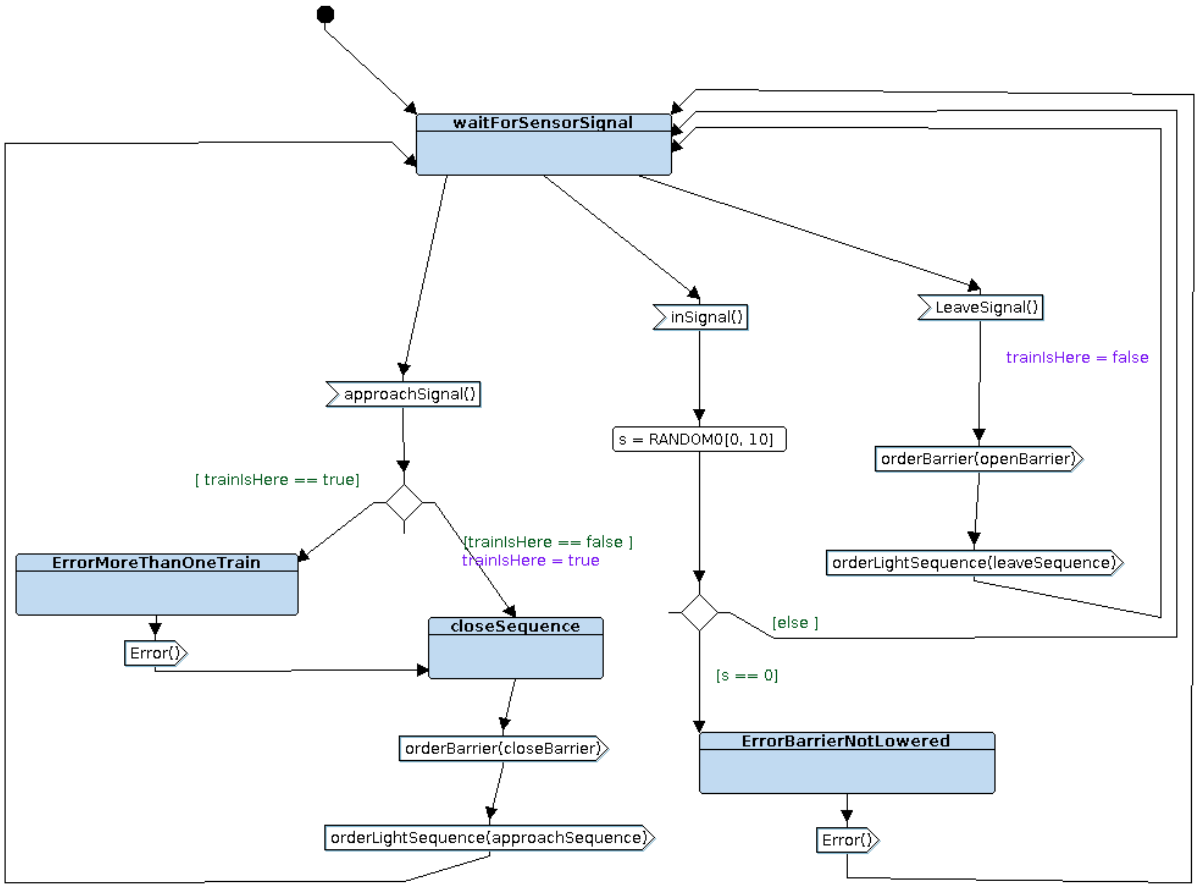

Figure 9- Main controller with random S variable

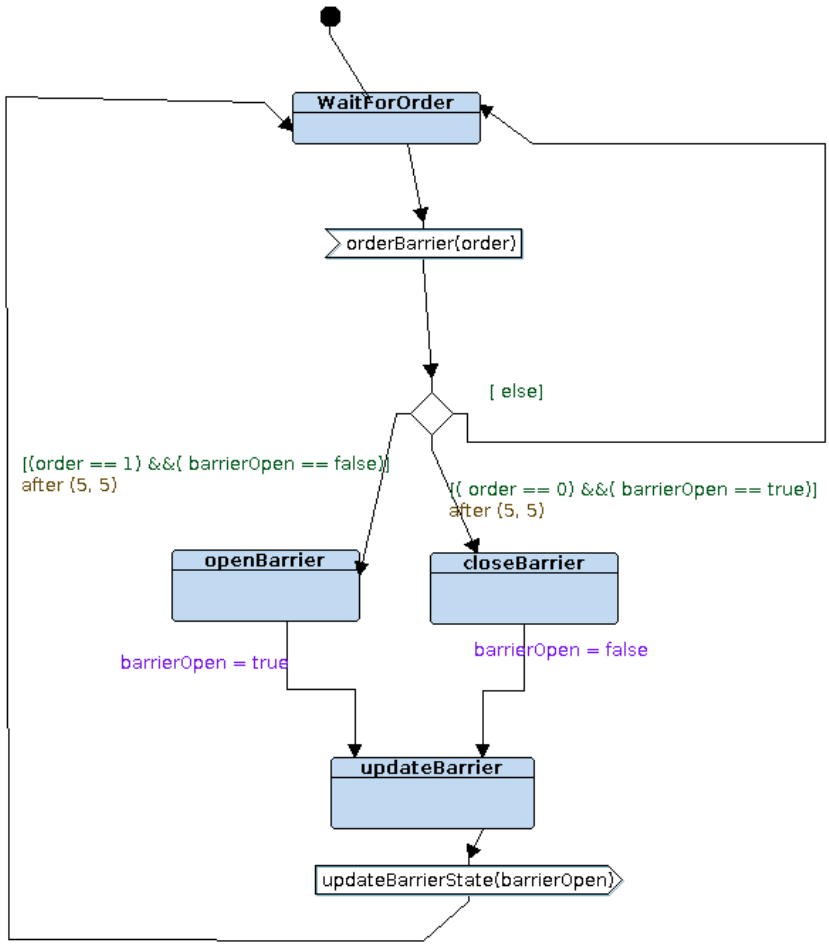

Figure 10- Barrier controller

When simulating with TTool, we get the following testcase depending on the random s value. We can also change the testbench to simulate the arrival of another train before the first one leaves.

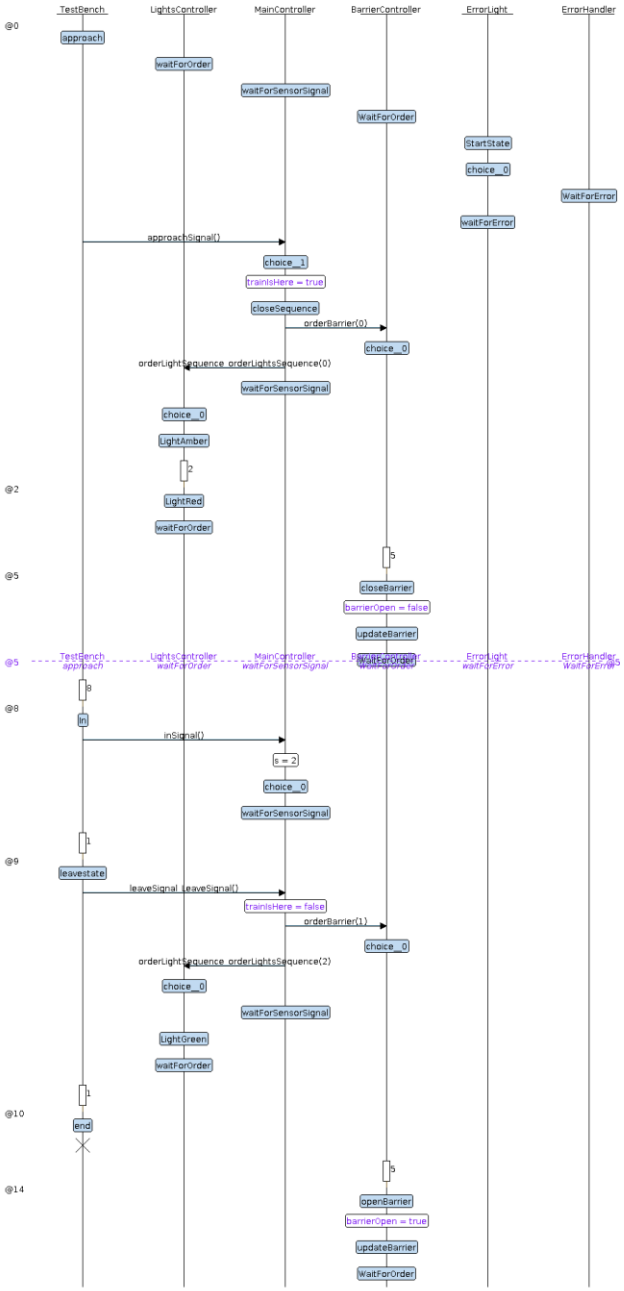

Figure 11- simulation result- no error

This simulation shows the expected case in normal function. Following the "approach" signal, the lights turn amber then red and the barriers are lowered. When receiving the "in" signal, s equals 2 which represents that the barrier have been lowered correctly, so no error is triggered. Finally following the "leave" signal, the light is turned green and the barrier opens.

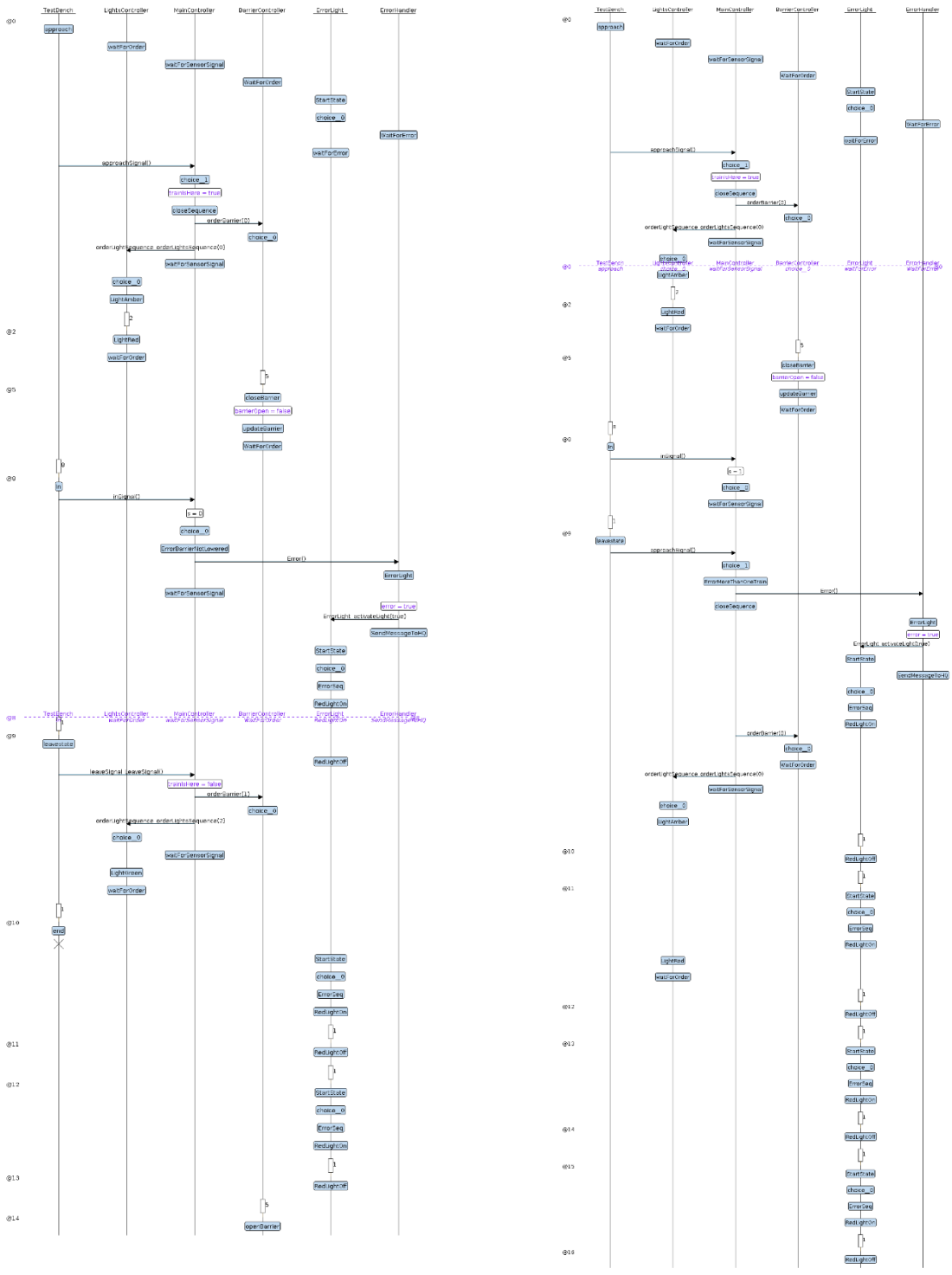

Figure 12- simulation result- error (right)

This simulation represents a case where the barriers are not completely close after receiving the "in" signal. S has been generated as 0, so the main controller triggers the error sequence by sending an error signal at time 8. A message is sent to HQ and the special error light switches value every tick, it is effectively blinking. However, our

system still allows the barrier to open after the "leave" signal has been triggered. This can be considered safe, as the train is supposedly away after that signal is received. But another input could be considered to manually "reset" the error and go back to normal.

Figure 13- second train (left)

Finally, this simulation shows the case where an "approach" signal is received while a train is already here (here it happens at time 9). The same error sequence as the previous case is triggered, and the barrier is kept closed.

6 CONCLUSION

The solution presented in this report gives satisfying results. It effectively handles three scenarios with two of them being error cases. However, some work could still be done on the design. For instance, the lights and barrier could both have an interface that stores the states of the lights/barriers and sends it to the main controller when requested. This way we could simulate different cases where the time the barriers take to lower is randomized, or where the lights receive an order while a sequence is already running. This would be our first change to build a more robust system.