Universiteit Antwerpen
Faculteit Wetenschappen

Modelling Of Software Intensive Systems

Assignment 4: Statecharts

1st Master computer science
2024-2025

Liam Leirs 20210395
Robbe Teughels 20211127

Exercises:

Exercise A

Event y never gets raised because the model never stays in stateA for 2 seconds without an event occurring. Instead, each second event x gets raised and the model moves back to stateA, resetting the timer.

Exercise B

The model starts in the composite state Outer, within this state it starts in the Inner state. After 2 seconds, the inner event gets raised and within the Outer state we move back to Inner. 1 second later (at 3s) the outer transition occurs, raising event outer and the model moves back to Outer, re-initializing the timers of both Outer and Inner. This process repeats: at 5s, inner gets raised and moves back to Inner, 1 second later (at 6s), outer gets raised and timers are reset again.

Exercise C

1) Intuitively we would expect that after 1 second, the model moves from Initial state to Temp state, incrementing x with 1. Then the transition with guard clause [x==1] triggers and we move to state One.

2) The model gets stuck in state Temp, unable to move to any subsequent state because no more events get triggered so the guard clauses aren't checked.

3) Replacing Temp by a choice element fixes the issue

Exercise D

At 5 seconds, the model is in state One, T. Initially, the model is in state Initial, S. After 1 second, the model moves to state Temp, S and x gets incremented by 1. The statechart stays in state Temp, S until time gets to 5 seconds, which triggers the transition from S to T. Since this event gets processed, the guard clause [x==1] gets evaluated which moves the model from state Temp to state One in r1. So finally at t = 5 seconds, the model is in state One, T.

Exercise E

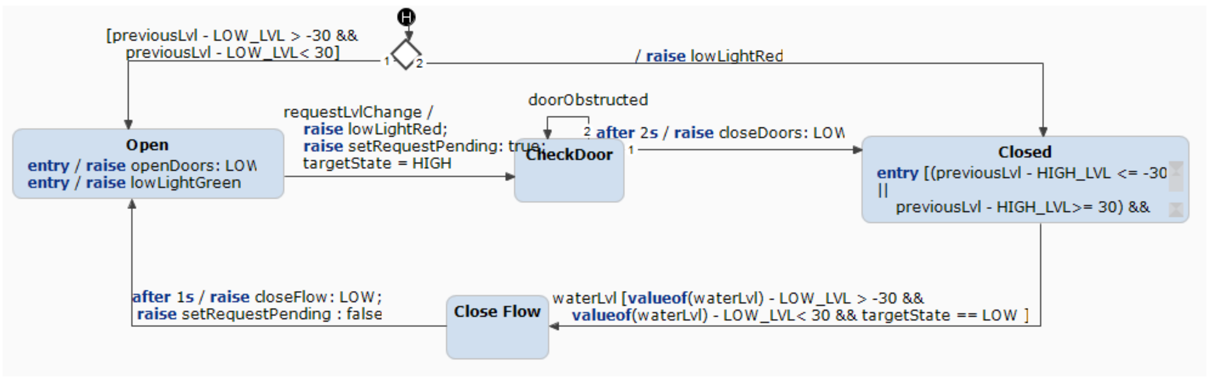
1) Similar to previous exercise, guard clause [v==0] only gets checked when an event triggers, so after 1 second y gets raised and consequently also x

2) In orthogonal states, events get raised from left to right. So after 1 second, the transition out of stateA gets fired and x gets raised, afterwards y gets raised.

Water Lock:

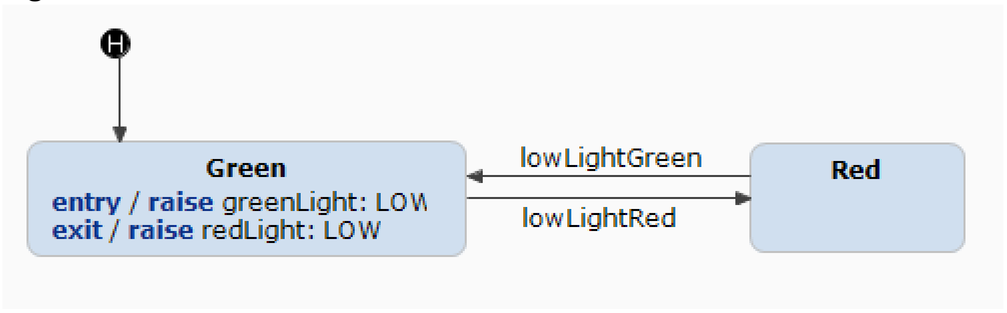
The state is properly divided into multiple components. The system is split into 2 distinct (almost identical) parts, the LOW part and the HIGH part. They are further separated into the door and light components.

Door:


[previousLvl - LOW_LVL > -30 &&
previousLvl - LOW_LVL< 30]
/ raise lowLightRed
requestLvlChange /
raise lowLightRed;
raise setRequestPending: true;
targetState = HIGH
doorObstructed
after 2s / raise closeDoors: LOW
Open
entry / raise openDoors: LOW
entry / raise lowLightGreen
CheckDoor
Closed
entry [(previousLvl - HIGH_LVL <= -30
||
previousLvl - HIGH_LVL>= 30) &&
after 1s / raise closeFlow: LOW;
raise setRequestPending : false
Close Flow
waterLvl [valueof(waterLvl) - LOW_LVL > -30 &&
valueof(waterLvl) - LOW_LVL< 30 && targetState == LOW]

This a general implementation. Initializations is abstracted out by determine starting state. This requires a proper initiation of the global system to work. The output request are mostly handled by entry statements. This ensures proper output when leaving and reentering the component using the previous state. The door is also a controller for its corresponding light.

Light:



The same as for the previous state. Updating the light happens when entering/leaving the state. This makes the light always red. Making it easier to integrate with higher levels of behaviour (interrupts as breakSensor).

Additional components:

The doors needs an approximation of the water levels. We have another orthogonal component responsible for updating the value. The lock the system when the sensor breaks, we detect this and leave change behavior to emergency mode. Here all the functionality of the Doors and lights are disabled. Meaning that we left there state and updating all required values to keep the system valid (all lights on red, doors closed, …). Because of the implementation of previous components, we con focus on only the core features (trying to resume).

Additional testcase:

The simulation must be robust and consistent but most of all safe. To check this, we combined the normal behavior with the exceptional behavior. Concrete, we looked at first proper initialization of the system. Then executing two request. During both requests, we both had a sensor broken and recovered. Then looking at the system during the simulation, we made sure no unsafe situation occurred.

With this test, we cover most of the requirements, R0: initialization, R1: workflow, R2: waterChangeRequest, R3: detection of sensorDefects, R4: response to a defect, R5: emergency Mode, R7: continue working after RESUME.

This testcase focuses on the global working of the system, leaving out the possible edge cases for each requirement.

Workflow:

We started this assignment by making the exercises together. Liam Leirs build a proper prototype of the WaterLock. Because of bug (mosly history related), Robbe Teughels created a new version of the WaterLock using the key componenst of the previous version.

Time Spend:

Liam Leirs: 6h

Robbe Teughels: 5,30h