Lab 6: Elevator Controller

Due: 11:59:59pm, Sunday May 28th, 2017

1 Introduction

Feel Good Inc. is constructing its new 3-floor headquarter complex at La Jolla, CA. Per California Building Code (CBC), an elevator is required for accessibility of the building. They asked students from UC San Diego to help them design the elevator controller in hardware.

1.1 System Composition

As shown in Figure 1, the complex has three floors. At each floor, there are request buttons on the wall indicating if the passengers waiting at this floor want to go up or go down. (there are no going down button on the first floor and no going up button on the third floor.) Inside the elevator cabin, there is a panel with three buttons, indicating the destinations of the passengers inside the cabin. The cabin is driven by the motor in the control room. The motor is controlled by a motor controller, which controls whether the motor is rotating or not, as well as the direction controller, which controls the moving directions of the elevator. We assume when the elevator stops moving, the door will keep open, and when elevator keeps moving, the door will be closed.

As shown in Figure 2, the elevator control system is composed of elevator controller, and a floor sensor and motor controller. In this lab, you will implement the elevator controller.

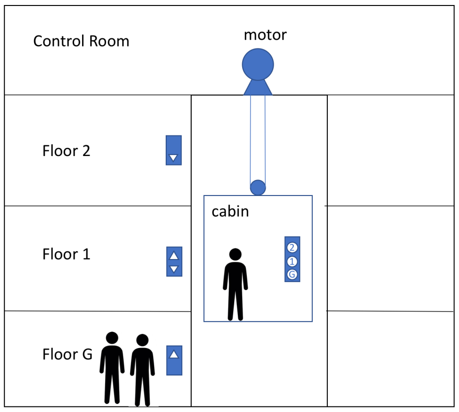

Figure 1: 3-storey headquarter complex with an elevator inside.

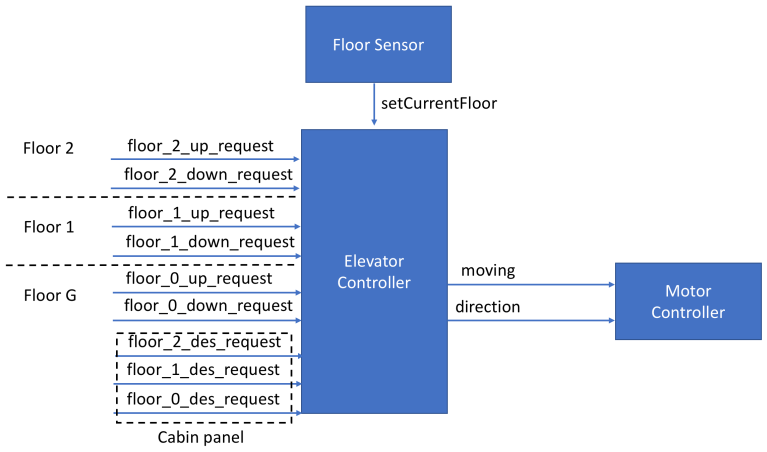

Figure 2: Schematic of the elevator control system.

1.2 Elevator Scheduling

The elevator works as follows: based on the current state of the elevator (which buttons are pressed, the current floor of the cabin and the moving direction of the cabin), the next floor to stop is uniquely decided. (In other words, the next floor to stop is a combinational function of the buttons that are pressed, the current floor of the cabin and the moving direction of the cabin).

We assume the elevator has unlimited capacity. We adapt the following general policy to decide the next floor to stop, assuming the elevator is moving up

1. if there is no request, stop at the current floor.
2. if the elevator is currently moving up and there are going up requests from the floors above or destinations to the floors above, stop at the closest floor from above that has either request or is destination.
3. except for the cases stated above, if the elevator is currently moving up and there are going down requests from the floors above, stop at the farthest floor from above that has that request.
4. except for the cases stated above, if the elevator is currently moving up and there are no requests from the floors above or destinations to the floors above, change the moving directions and then check if it matches the cases stated above.

For the cases when the elevator is moving down, the policy should change respectively. If multiple buttons are pressed at exactly at the same time, the elevator will prefer keep its current moving direction, unless no destinations or requests are from current moving direction.

Once the next floor to stop is decided, we are able to decide the moving/stoping and moving directions of the cabin to get to the next floor to stop. We represent the state of the elevator control system using a tuple {`moving, direction`}. `moving` is an enumerate type variable with value `START` and `STOP`. `direction` is an enumerate type variable with value `UP` and `DOWN` representing the moving direction of the elevator. Applying the above policy, the elevator controller forms a finite state machine (FSM), as shown in Figure 3.

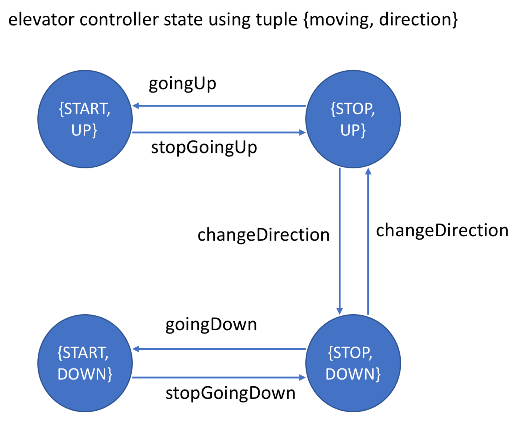

Figure 3: FSM representation of the elevator controller.

2 Lab Assignment

First, make a copy of the source code using the following command

```
$ cp -r  /home/linux/ieng6/cs140g/public/lab6 .
```

In `Elevator.bsv`, we define the following enum type to represent the moving direction of the elevator.

```
typedef enum {UP, DOWN} Direction
typedef enum {START, STOP} Moving
```

Figure 2 shows the schematic of this elevator control system.

In `Elevator.bsv`, we give the interface for implementing the elevator controller.

```
interface Elevator;
        method Action floor_0_up_request;
        method Action floor_1_up_request;
        method Action floor_1_down_request;
        method Action floor_2_down_request;

        method Action floor_0_des_request;
        method Action floor_1_des_request;
        method Action floor_2_des_request;

        method Action setCurrentFloor(int floor);

        method Moving moving;
        method Direction direction;
endinterface
```

`floor_{0,1,2}_{up,down}_request` represents the buttons outside the cabin on each floor (Floor G = Floor 0) to request going up or going down. `floor_{0,1,2}_des_request` represents the buttons inside the cabin indicating the destinations of the elevator. `setCurrentFloor(int floor)` indicates the elevator controller that the cabin is currently at floor `int floor`.

`moving` indicates the signal the elevator controller sending to the motor controller whether to turn on the rotation or not, and `direction` indicates the signal the elevator controller sending to the motor controller of the moving direction. The motor controller checks these signals continously, and the changes of the value of the signals are immediately received by the motor controller, resulting in the change of stop/start moving and moving direction change of the cabin instaneously.

In order to implement the interface `Elevator.bsv`, we predefined several registers inside the `mkElevator`.

```
module mkElevator(Elevator);
        Reg#(Vector#(NUM_FLOOR, Bool)) floor_up_pressed <− mkReg(replicate(False));
        Reg#(Vector#(NUM_FLOOR, Bool)) floor_down_pressed <− mkReg(replicate(False));
        Reg#(Vector#(NUM_FLOOR, Bool)) floor_des <− mkReg(replicate(False));

        Reg#(int) currentFloor <−mkReg(0);
        Reg#(Direction) direction_reg <−mkReg(UP);
        Reg#(Moving) moving_reg <−mkReg(STOP);

   ...
endmodule
```

`floor_{up,down}_pressed[i]` are set `True` if the respective requests from method `floor_i_{up,down}_request` are made, and they will remain `True` until the the respective request has been resolved. (=The elevator stops at that floor due to that request. You can think of them as indicating lights on the buttons of the elevator.) Because the moving down request at bottom floor and moving up request at top floor are meaningless, we always assume `floor_up_pressed[NUM_FLOOR-1]` and `floor_down_pressed[0]` are `False`.

Similarly, `floor_des[i]` are set `True` if the respective requests from method `floor_i_des_request` are made, and they will remain `True` until the respective request has been resolved.

Register `currentFloor` indicates the current floor of the cabin. Register `direction_reg` represents the moving direction of the elevator and register `moving_reg` represents the the moving/stoping of the elevator.

Exercise 1: 30 points

Assume the elevator is currently moving up due to some request from above or current floor or the destination to the floors above or the current floor, we define the following function:

```
function int nextFloorUp(Vector#(NUM_FLOOR, Bool) floor_up_pressed, Vector#(NUM_FLOOR, Bool) floor_down_pressed, Vector
    #(NUM_FLOOR, Bool) floor_des, int currentFloor);
```

to determine the next floor to stop. `nextFloorUp` takes the `floor_up_pressed`, `floor_down_pressed`, `floor_des`, and `currentFloor` as defined in the `mkElevator` as input and returns the next floor to stop as an integer.

Similarly, assume the elevator is currently moving down due to some request from below or current floor or the destination to the floors below or the current floor, we define the following function to determine the next floor to stop.

```
function int nextFloorDown(Vector#(NUM_FLOOR, Bool) floor_up_pressed, Vector#(NUM_FLOOR, Bool) floor_down_pressed,
    Vector#(NUM_FLOOR, Bool) floor_des, int currentFloor);
```

In `Elevator.bsv`, complete the bodies `nextFloorUp` and `nextFloorDown`. You can test your implementations using the following testbenches.

```
$ make nextFloorSingle
$ ./simNextFloorUp
$ ./simNextFloorDown
```

You can compare your simulation output with the the reference simulation provided in `refSimNextFloorUp.txt` and `refSimNextFloorDown.txt` for `simNextFloorUp` and `simNextFloorDown` respectively.

You can use vimdiff or diff to compare the output

```
$ vimdiff refSimNextFloorUp.txt <(./simNextFloorUp)
$ vimdiff refSimNextFloorDown.txt <(./simNextFloorDown)
```

Check grading section [3] for the grading details.

Exercise 2: 20 points

Now assume the elevator is at arbitrary state (not necessarily moving up or moving down due to request from above or below.)

We define the following function to determine the next floor to stop.

```
nextFloor(Vector#(NUM_FLOOR, Bool) floor_up_pressed, Vector#(NUM_FLOOR, Bool) floor_down_pressed, Vector#(
    NUM_FLOOR, Bool) floor_des, int currentFloor, Direction direction);
```

In `Elevator.bsv`, complete the bodies `nextFloor`. You can test your implementations using the following testbenches.

```
$ make nextFloor
$ ./simNextFloor
```

You can compare your simulation output with the the reference simulation provided in `refSimNextFloor.txt`.

```
$ vimdiff refSimNextFloor.txt <(./simNextFloor)
```

Check grading section [3] for the grading details.

Exercise 3: 40 points

Now that we know the next floor to stop as function of the elevator state, we can decide the moving direction and moving/stopping of the motor based on the next floor to stop and the current floor. We define the following rules in the mkElevator module:

```
module mkElevator(Elevator):
    rule goingUp;
    rule goingDown;
    rule changeDirection;
    rule stopGoingUp;
    rule stopGoingDown;
endmodule
```

Complete these rules based on the information given in Figure 3. Please also notice that you have to reset the respective registers once the elevator stops at certain floor (i.e., if the stop at the current floor is due to destination button, going up button or going down button being pressed and the respective register for the button is set, reset this register to `False`).

You have to test your implementations using the following testbench.

```
$ make rules
$ ./simRules
```

The testbench generates the trace of the cabin due to several requests, please compare it with `refSimRules.txt` to see if it makes sense.

```
$ vimdiff refSimRules.txt <(./simRules)
```

Check grading section [3] for the grading details.

Exercise 4: 10 points

Feel Good Inc. gets more funding from National Science Foundation (NSF), and instead of building a 3-floor headquarter complex, they plan to build an 10-floor headquarter complex. We provide interface for 10-floor elevator in the following:

```
interface ElevatorNF;
        method Action floor_up_request(int i);
        method Action floor_down_request(int i);
        method Action floor_des_request(int i);
        method Action setCurrentFloor(int floor);
        method Moving moving;
        method Direction direction;
endinterface
```

In `Elevator10f.bsv`, complete the rules inside `mkElevatorNF` module.

You have to test your implementations using the following testbenches.

```
$ make 10floor
$./sim10Floors
```

The testbench generates the trace of the cabin due to several requests, please compare it with `refSim10Floors.txt` to see if it makes sense.

```
$ vimdiff refSim10Floors.txt <(./sim10Floors)
```

Check grading section [3] for the grading details.

3 Grading

You are expected to test your functions and module for additional test cases. Matching your simulation output with the given reference simulation is not sufficient to get full credits. The provided Testbenches only covers basic test cases. You can modify the Testbench files to add new test cases or modify the given one. Note that none of the testbench files would be submitted forgrading.

File `TbNextFloor.bsv` contains modules `mkNextFloorUp`, `mkNextFloorDown` and `mkNextFloor` for **Exercise 1** and **Exercise 2**. You can modify rule `r_test` for each of them.

File `TbRules.bsv` contains module `mkRules` for **Exercise 3**. and file `Tb10f.bsv` contains module `mk10Floors` for **Exercise 4**. You can modify rule `r_step` in both the files for additional test cases.

4 Submission

For Exercise 1, 2 and 3, please write your answers in `Elevator.bsv`. For Exercise 4, write your answers in `Elevator10f.bsv`. Write down your name, PID and e-mail address in `Lab6.txt` located in `lab6/` directory. If you worked in a group, write down your partner's information as well. While your solutions can be identical, each group member must make their own submission. You will be scored based on your own submission. To submit your assignment, simply run the following command from `lab6/` directory:

To submit your assignment, simply run the following command from `lab6/` directory:

```
$ bundleP6 <your-email>
```

Submission verification

Refer to Lab 1.