**University of Camerino**

Master Degree in Computer Science

Course of System Verification Lab

# Promela Lift Elevator

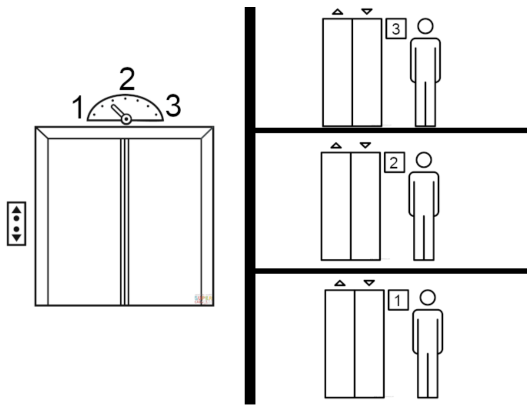
Author

**Davide Parente 119984**

A.A. 2021/2022

## 1 What is about?

The project consists on the simulation of a lift system. To carry out this project there are some concepts to identify. First of all we have an *"elevator"* that consists a lift which would be the variable that identifies the floor where the cabin is located at a specific moment of the execution, it has also the behaviour of the cabin so it can move in the different floors. Also the system is developed for the management of 3 different floors.
Below of this short description is possible to see an example of this.



How we can see there are three specific floor with the respective button and the lift that in this specific moment is at floor 1.

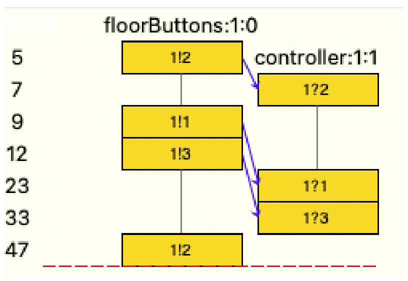
## 2 Description of code

A brief introduction of the field used for the correct functionality of this program.

- The first variable we can see when we open the project is *chan c*. It is the channel that permits our controller to communicate with the respective button floor. Practically when a button is pressed the request for the plane is added on this channel, that have the functionality of a Queue.

  ```
  chan c = [FLOOR] of {byte};
  ```

  A simple interactive run show us how this channel works.



- The variable state declared in this way :

  ```
  bool state[FLOOR];
  ```

It is an array used to define when a button is pressed or not. Basically if the state[0] is set to true someone at floor 1 pressed the button and sooner or later the lift will go to the floor 1.

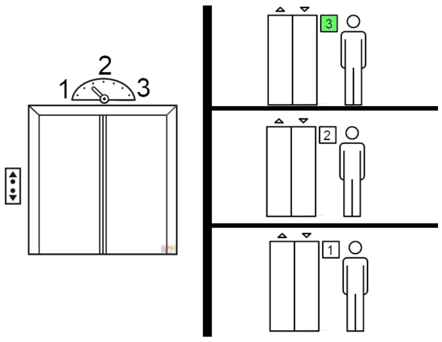


Figura 1: Button 3 pressed. state[0],state[1]= false; state[2]=true;

In this specific case someone pressed the button at floor Three and the elevator will go to this specific floor.
Another example would be with all buttons pressed. In this case the lift must respect the queue order.

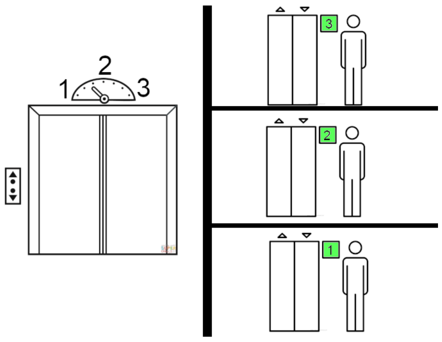
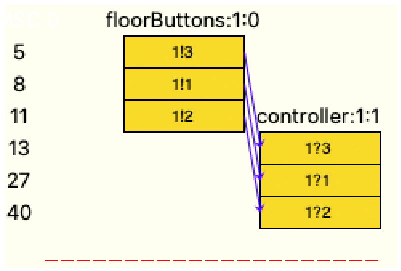


Figura 2: state[0],state[1],state[2]=true; c=[2,1,3]

In this case the run will be:



- The boolean opendoor is setted at true when the port of the cabin are opened otherwise it is close and the value is false. Initially the port are close.

```
bool opendoor=false; // door close
```

- The short variable elevator indicates the current position of the lift. Initially it is setted at 1 that indicates the first floor.

```
short elevator=1;
```

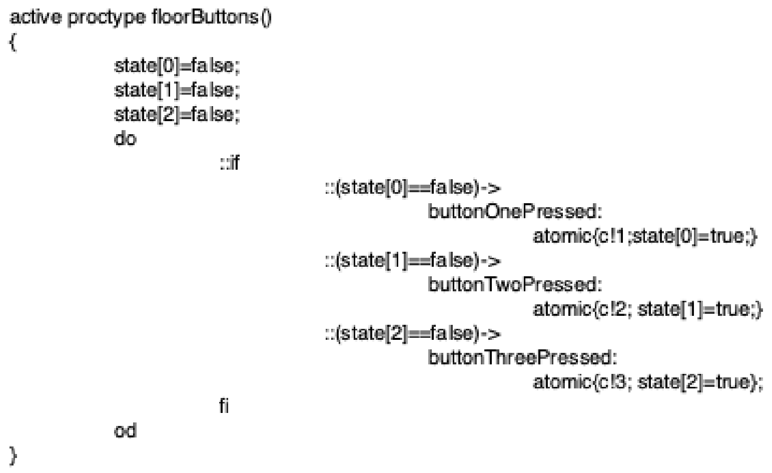
- Standing is a boolean variable that indicates that the elevator is in the correct floor level.

```
bool standing=false;
```

- The variable short piano indicates the current value read from the channel c.

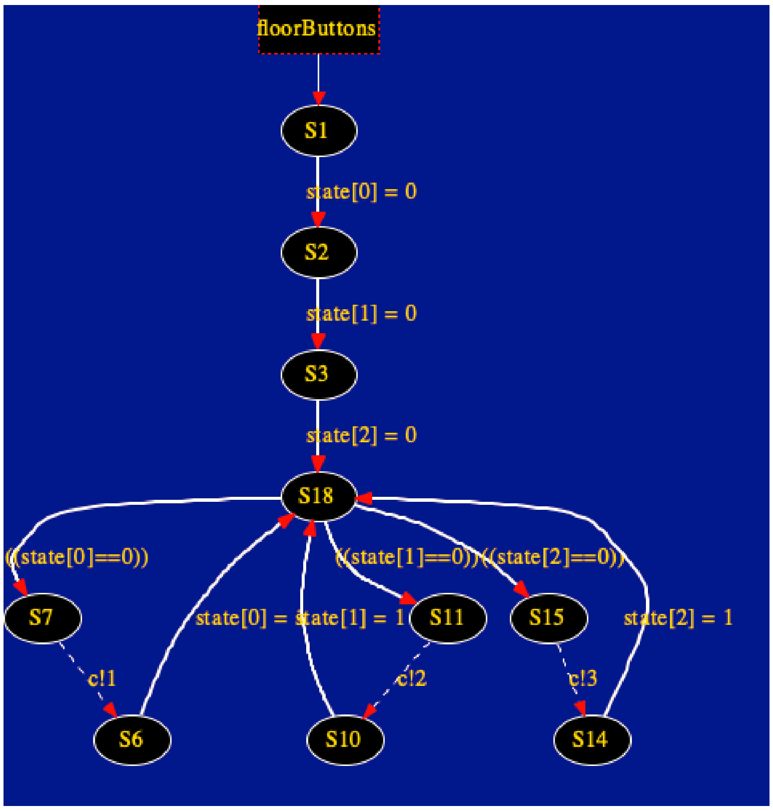
```
short piano;
```

After having indicated the variables, it is necessary to understand the two proctype.
We have the class floorButtons that indicates the behaviour of the button that could be chosen in a non-deterministic way.



```
active proctype floorButtons()
{
        state[0]=false;
        state[1]=false;
        state[2]=false;
        do
                ::if
                        ::(state[0]==false)->
                                buttonOnePressed:
                                        atomic{c!1;state[0]=true;}
                        ::(state[1]==false)->
                                buttonTwoPressed:
                                        atomic{c!2; state[1]=true;}
                        ::(state[2]==false)->
                                buttonThreePressed:
                                        atomic{c!3; state[2]=true};
                fi
        od
}
```

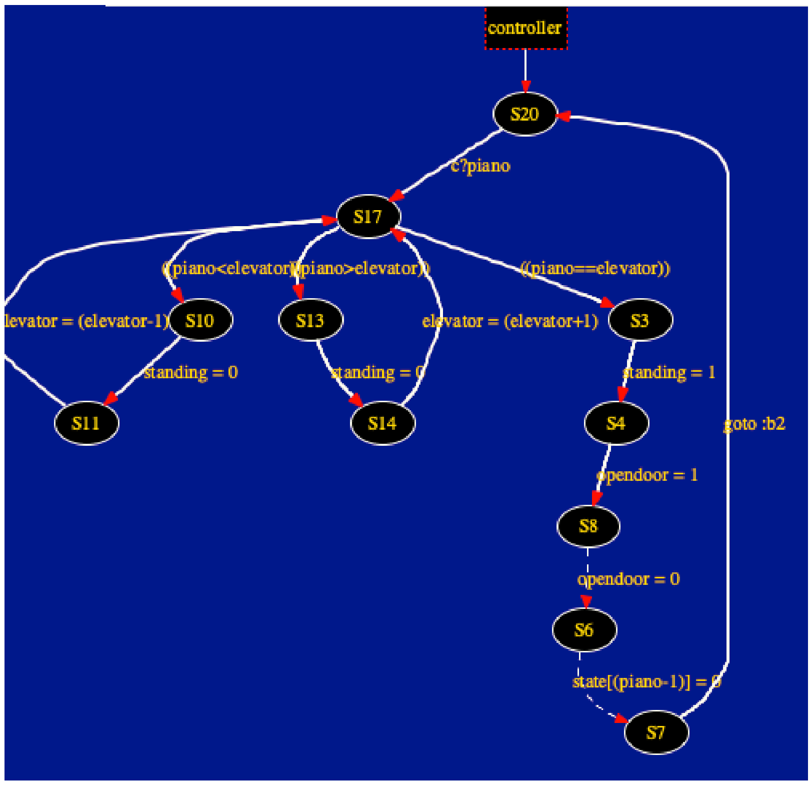
First of all it puts all the button like unpressed next it enter in a loop where in a non-deterministic way choice the button to press. If a button is already pressed it can't be pressed again, in fact before entering in the label *buttonXPressed* there is an *if* condition that check if the button is already pressed.
The action in all the label are setted like atomic option because the operations must be done atomically and in the same time.
When the button is pressed, the piano selected by the button is added to channel c.
The automa generated is:



Furthermore, a controller has been developed which has the purpose of reading from channel c the floor where the lift must go. Then you will enter a loop until the elevator reaches the floor read by channel c. If you need to enter and change the floor, the variable elevator will be modified so that it can reach the established floor.

```
active proctype controller()
{
          do
          ::c?piano;
                    movelevator:
                    do
                              ::if
                              ::(piano==elevator)->
                                        standing=true;
                                        dooropened:
                                                  opendoor=true;
                                        doorclosed:
                                                  atomic{ opendoor=false;
                                                  state[piano-1]=false;
                                                  break;} //here open and close door
                              ::(piano<elevator)->
                                        standing=false;
                                        down:
                                                  elevator--;
                              ::(piano>elevator)->
                                        standing=false;
                                        up:
                                                  elevator++;
                              fi
                    od
          od
}
```

The automa generated is:



The state *S20* have the scope of read from the channel c the value of the floor where to go.
The state *S17* check where the elevator has to go, if it is already on the correct floor or if it has to go up.

## 3 LTL formalization

- Whenever the door is open the cabin must be standing.

```
ltl p1 { [] (opendoor -> standing)}
```

- Whenever the cabin is moving the door must be closed.

```
ltl p2 {[]((controller@down || controller@up) -> !opendoor)}
```

- A button cannot remain pressed forever.

```
ltl p3 {[](state[0]-> <> !state[0]) &&[](state[1]-> <> !state[1])
```

- The door cannot remain open forever.

```
ltl p4 {[](opendoor -> <> !opendoor)}
```

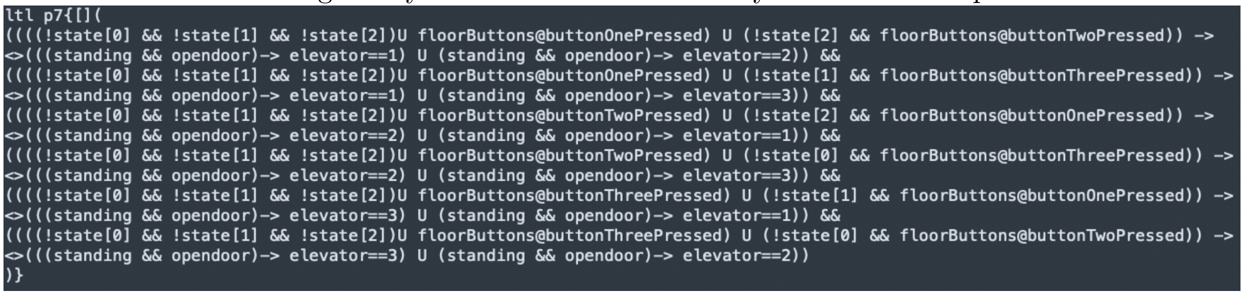
- The door cannot remain closed forever.

```
ltl p5 {[](!opendoor -> <> opendoor)}
```

- Whenever the button at floor x (x=1,2,3) becomes pressed then the cabin will eventually be at the fllor x with the door open

```
ltl p6 {[](state[piano-1] -> <>(elevator==piano && opendoor))}
```

- Whenever no button is currently pressed and the button at floor x (x = 1, 2, 3) becomes pressed and, afterwards, also the button at floor y (y =! x and y = 1, 2, 3) becomes pressed and, in the meanwhile, no other button becomes pressed then the cabin will be standing at floor x with the door open and, afterwards, the cabin will be standing at floor y with the door open and in the meanwhile the cabin will not be standing at any other floor different from y with the door open.

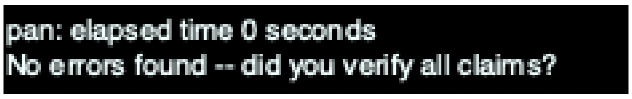


```
ltl p7{[](
((((!state[0] && !state[1] && !state[2])U floorButtons@buttonOnePressed) U (!state[2] && floorButtons@buttonTwoPressed)) ->
<>(((standing && opendoor)-> elevator==1) U (standing && opendoor)-> elevator==2)) &&
((((!state[0] && !state[1] && !state[2])U floorButtons@buttonOnePressed) U (!state[1] && floorButtons@buttonThreePressed)) ->
<>(((standing && opendoor)-> elevator==1) U (standing && opendoor)-> elevator==3)) &&
((((!state[0] && !state[1] && !state[2])U floorButtons@buttonTwoPressed) U (!state[2] && floorButtons@buttonOnePressed)) ->
<>(((standing && opendoor)-> elevator==2) U (standing && opendoor)-> elevator==1)) &&
((((!state[0] && !state[1] && !state[2])U floorButtons@buttonTwoPressed) U (!state[0] && floorButtons@buttonThreePressed)) ->
<>(((standing && opendoor)-> elevator==2) U (standing && opendoor)-> elevator==3)) &&
((((!state[0] && !state[1] && !state[2])U floorButtons@buttonThreePressed) U (!state[1] && floorButtons@buttonOnePressed)) ->
<>(((standing && opendoor)-> elevator==3) U (standing && opendoor)-> elevator==1)) &&
((((!state[0] && !state[1] && !state[2])U floorButtons@buttonThreePressed) U (!state[0] && floorButtons@buttonTwoPressed)) ->
<>(((standing && opendoor)-> elevator==3) U (standing && opendoor)-> elevator==2))
)}
```

the rule was written for every possible pair of floors. To be exact, the order of the pairs is: 1.2; 1.3; 2.1; 2.3; 3.1; 3.2.
Initially, it is checked that each button is not pressed, then it is checked that first one button is pressed and then the other. In the second part after the implication it is simply checked that the first value of the pair is executed first and then the second (that is, that the lift goes first to floor x and then to floor y).

## 4 Conclusion

Running all the properties together don't find some type of error.



```
pan: elapsed time 0 seconds
No errors found -- did you verify all claims?
```

The building of the project could require some seconds because the last ltl property is heavy. All the properties are checked in a few of seconds without errors. Finally the project respects all requirements in an easy way, separating the behavior of the buttons from the general behavior of the elevator which is managed entirely by the controller and by the three variables opendoor, standing and elevator.