Project - Elevator

Dominik Koszkul, Michal Oleszczyk, Cezary Dynak, Marek Frydrysiak

January 18, 2016

1 Introduction

The main goal of the project is to develop a controller of a lift system. It will consists of a simulator (depicting a state of a set of lifts) and a controller, which will control their activity basing on users requests and the system state.

The main concept is to create two different programs (one implementing the controller system and the second one for implementation of the elevators simulation). The programs should be independent, i.e. they should use some general protocol for communication, and work failure-free with any other external programs which also use that specified protocol. Both programs start with a set of parameters: a number of elevators and a number of floors supported by any of them. The number of floors does not need to be the same for different elevators. Our elevator controller and simulation do not support underground floors.

Assumption: at the very beginning all elevators are in the following state: closed doors on the floor 0 (ground floor). The controller remembers the current state of all the elevators anytime.

2 Project concept

2.1 Module structure

The module structure of the system is depicted in Figure 1.

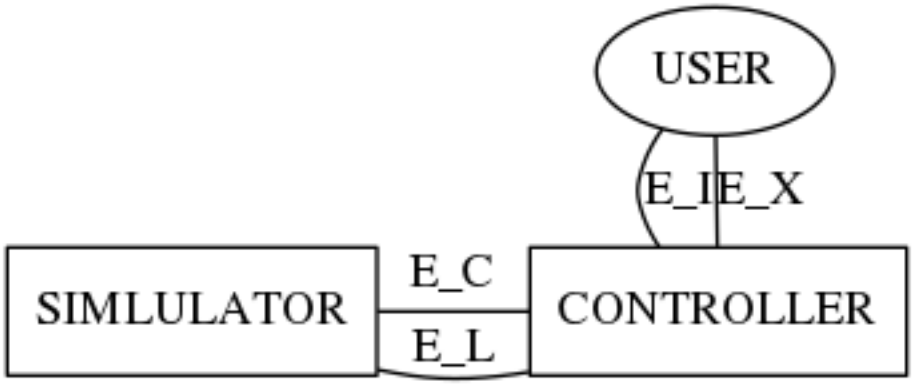


Figure 1: DES graph

The system is based on three main parts: the simulator, the controller and the user interface. Essentially, the user interface and the simulator are built within one program. It is as the simulator's part is not only responsible for visualising a state of the lifts, but also it allows a user to interact with it, for instance, by calling a lift on a particular floor choosing a destination floor.

2.2 Module communication

Since the communication protocol between the simulator and the controller is set, it is not necessary for any of them to know how the other works internally. The programs obey the rules of message format, what is sufficiently necessary for a proper communication.

The set of events E is described as follows:

$$E = E^x \cup E^i \cup E^c \cup E^l,$$

where:

E^x - events from the external buttons,

E^i - events from the internal buttons,

E^c - events from the controller,

E^l - events from the lifts.

3 System model

3.1 States

The state of the system model can be defined as follows:

$$S = [W, P, Q],$$

where:

W - lifts state,

P - external buttons state,

Q - internal buttons state.

The sets of states W, P, Q are precisely explained in the subsequent sections.

3.1.1 Lift states

$$W = [w_1, w_2, ..., w_i, ..., w_l],$$

where:

l - number of lifts.

For each lift state

$$w_i = [d_i, o_i, f_i],$$

where:

$d \in \{\text{going\_down}, \text{stopped}, \text{going\_up}\}$,

$o \in \{\text{closed}, \text{opened}\}$,

$f \in \{0, 1, ..., i, ..., n\}$.

Moreover

i - next floor to be reached or the current floor,

n - number of floors.

3.1.2 External buttons states

$$P = [p_1, p_2, ..., p_i, ..., p_n],$$

where:

n - number of floors,

$$p_i = [g_{i_d}, g_{i_u}],$$

where:
$g_{i_d} \in \{\text{not\_pushed}, \text{pushed}\}$, $g_{i_u} \in \{\text{not\_pushed}, \text{pushed}\}$.

Moreover

d - down,

u - up.

3.1.3 Internal buttons states

$$Q = [q_1, q_2, ..., q_i, ..., q_l]$$

where:

- l - number of lifts

$$q_i = [b_{i_0}, b_{i_1}, ..., b_{i_j}, ..., b_{i_n}]$$

$b_{i_j} \in \{\text{not\_pushed}, \text{pushed}\}$

3.2 Events

3.2.1 Events from the external buttons

$$E^x = [\text{floor\_nr}, \text{direction}]$$

floor_nr $\in \{0, 1, ..., n\}$,

direction $\in \{down, up\}$.

3.2.2 Events from the internal buttons

$$E^i = [\text{lift\_nr}, \text{button\_nr}]$$

button_nr $\in \{0, 1, ..., n\}$

3.2.3 Events from the lifts

$$E^l = [\text{lift\_nr}, \text{action}]$$

action $\in \{\text{going\_down}, \text{stopped}, \text{going\_up}, \text{closed}, \text{opened}\}$

3.3 Transition functions $f(s, e)$

3.3.1 Events from the external buttons

$$f(s_0, [\text{lift\_nr}, \text{button\_down}]) = [0, [[0, 0], ..., [g_{\text{lift\_nr}_d} = \text{pushed}, 0], ..., [0, 0]], 0]$$
$$f(s_0, [\text{lift\_nr}, \text{button\_up}]) = [0, [[0, 0], ..., [0, g_{\text{lift\_nr}_u} = \text{pushed}, 0], ..., [0, 0]], 0]$$

3.3.2 Events from the internal buttons

$$f(s_0, [\text{lift\_nr}, \text{button\_nr}]) = [0, [[0, ..., 0], ..., [0, ..., b_{\text{lift\_nr}_{\text{button\_nr}}} = \text{pushed}, .., 0], ..., [0, ..., 0]], 0]$$

3.4 Initial state s_0

All 0.

$$s_0 = [0, 0, 0]$$

3.5 Marked states S_M

All permitted.

4 Controller model

4.1 Events

4.1.1 Events from controller

$$E^c = [\text{lift\_nr}, \text{command}]$$

command $\in \{\text{go\_down}, \text{stop}, \text{go\_up}, \text{close\_door}, \text{open\_door}\}$

4.2 Transition function $f(s, e)$

4.2.1 Events from controller

$f(s_0, [\text{lift\_nr}, \text{go\_down}] = [[..., [d_{\text{lift\_nr}} = \text{going\_down}, 0, f_{\text{lift\_nr}} + 1], ...], 0, 0]$
$f(s_0, [\text{lift\_nr}, \text{stop}] = [[..., [d_{\text{lift\_nr}} = \text{stopped}, 0, 0], ...], 0, 0]$
$f(s_0, [\text{lift\_nr}, \text{go\_up}] = [[..., [d_{\text{lift\_nr}} = \text{going\_up}, 0, f_{\text{lift\_nr}} - 1], ...], 0, 0]$
$f(s_0, [\text{lift\_nr}, \text{close\_door}] = [[..., [0, o_{\text{lift\_nr}} = \text{closed}, 0], ...], 0, 0]$
$f(s_0, [\text{lift\_nr}, \text{open\_door}] = [[..., [0, o_{\text{lift\_nr}} = \text{opened}, 0], ...], 0, 0]$

5 Implementation

5.1 Simulation

The simulation program was written in the Python2.7 scripting language. For preparation of the user interface we used the *Tkinter* standard library. The simulation consists of 3 modules: graphics.py, elevator.py and my_threads.py. It implements 2 basic classes:

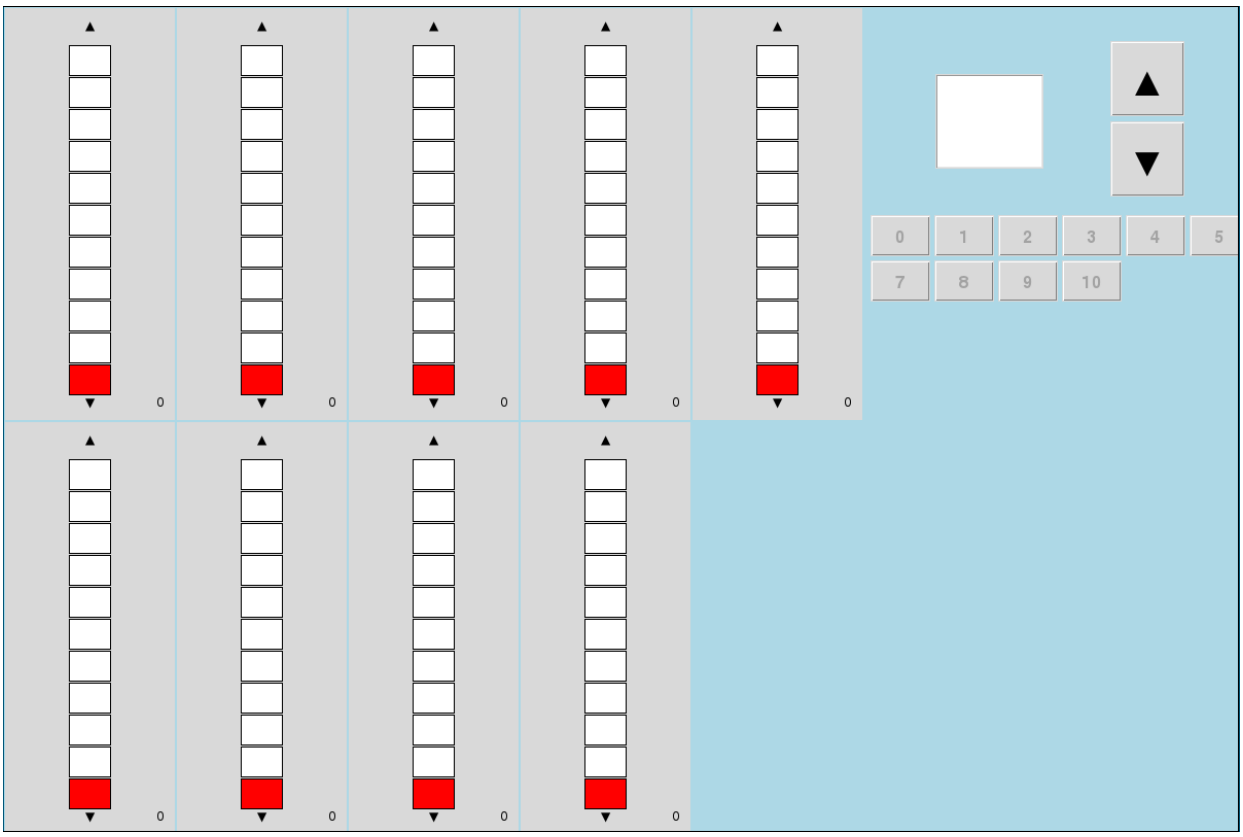
- Stage - class responsible for drawing the user interface in general (makes windows, buttons, digital panels).
- Elevator - class responsible for handling the elevator object (sets its parameters, states).

and classes inheriting from the class *Thread*:

- MoveLiftDown - class executes a command type $X : Y$.
- StopLift - class executes a command type $X : s$.
- MoveLiftUp - class executes a command type $X : Y$.
- OpenDoors - class executes a command type $X : o$.
- CloseDoors - class executes a command type $X : c$.
- ListenInstructions - class listen whether on the input port there is any incoming command or not.
- SendInstructions - class sends to the output port a command for controller.

The program is basing on 4 main threads (to quarantee the simultaneous work):

- Main thread - is responsible for the parameters configuration, start other thread and waiting for them to finish.
- Graphical thread - is responsible for refreshing the user interface (drawing the elevators and the floors) and for sending.
- Listenning thread - is responsible for listening on the input port and adding the incoming instructions to the input queue.
- Sending thread - is responsible for sending a command to the controller via the output port if there is an instruction ready to be sent to the output queue.



Furthermore, any command read from the controller (like MoveLift, StopLift, OpenDoor, CloseDoor) takes some predefined time. That is why any of that incoming commands starts another independent thread. In that thread, the commands are executed (for example during the MoveLift command the user interface shows direction of movement, highlighting a specified floor to green). After the execution that thread puts in the output queue a confirmation of the execution for the controller $(X : a)$.

5.2 Controller

The controller uses the same set of data, as the simulator provides. The exemplary controller execution is listed below

```
>> node socket_control.js
floor list: 10,10,10,10,10,10,10,10,10,7,8,20
max floor: 20
W =
[ [ 0, 0, 0 ],
  [ 0, 0, 0 ],
  [ 0, 0, 0 ],
  [ 0, 0, 0 ],
  [ 0, 0, 0 ],
  [ 0, 0, 0 ],
  [ 0, 0, 0 ],
  [ 0, 0, 0 ],
```

```
  [ 0, 0, 0 ],
  [ 0, 0, 0 ],
  [ 0, 0, 0 ],
  [ 0, 0, 0 ] ]
P =
[ [ 0, 0 ],
  [ 0, 0 ],
  [ 0, 0 ],
  [ 0, 0 ],
  [ 0, 0 ],
  [ 0, 0 ],
  [ 0, 0 ],
  [ 0, 0 ],
  [ 0, 0 ],
  [ 0, 0 ],
  [ 0, 0 ],
  [ 0, 0 ],
  [ 0, 0 ],
  [ 0, 0 ],
  [ 0, 0 ],
  [ 0, 0 ],
  [ 0, 0 ],
  [ 0, 0 ],
  [ 0, 0 ],
  [ 0, 0 ] ]
Q =
[ [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ],
  [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ],
  [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ],
  [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ],
  [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ],
  [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ],
  [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ],
  [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ],
  [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ],
  [ 0, 0, 0, 0, 0, 0, 0 ],
  [ 0, 0, 0, 0, 0, 0, 0, 0 ],
  [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ] ]
```

5.3 Connection

The connection is based on sockets. The 'incoming' port for the controller is at the same time the 'outup' port for the simulation program and vice versa.

For now:
controller ->simulation is realised on port 8089,
simulation ->controller is realised on port 8090.

5.4 Protocol

The protocol corresponds to the events defined in the previous section.

5.4.1 Controller → Simulation commands

1. $X : Y$ (where X is the number of an elevator, Y is number of a floor) - let the elevator X move to the floor Y.
2. $X : s$ (where X is the number of an elevator) - let the elevator X stops.
3. $X : o$ (where X is the number of an elevator) - let the elevator X opens the door.
4. $X : c$ (where X is the number of an elevator) - let the elevator X closes the door

5.4.2 Simulation → Controller commands

1. $X : a$ (where X is the number of an elevator) - the elevator X confirms execution of the previous controller command.
2. $Y : d$ (where Y is the number of a floor) - on the floor Y^{th} user pushes the button to go down.
3. $Y : u$ (where Y is the number of a floor) - on the floor Y^{th} user pushes the button to go up.
4. $X : Y$ (where X is the number of an elevator, Y is the number of a floor) - inside the elevator X user pushes the button to go to the Y^{th} floor.

Command $X : a$ should be send from the simulation to the controller after execution of any controller command. It provides a synchronization between both programs.

6 Effort

The task was divided among the group members in the following order:

1. Michal Oleszczyk - simulation part, documentation
2. Dominik Koszkul - simulation part, documentation
3. Cezary Dynak - controller part, documentation
4. Marek Frydrysiak - controller part, documentation