Project - Elevator

Dominik Koszkul, Michal Oleszczyk, Cezary Dynak, Marek Frydrysiak

December 2, 2015

1 Main concept

Main concept is to create two different programs (one for implementation of controller system and the second one for implementation of elevators simulation). Programs should be independent. That means they should use some general protocol for communication, and should work fine with any other external programs which also use that specified protocol. Both programs start with set of parameters: number of elevators and number of floors supported by any of them. Number of floor 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 state: closed door on floor 0 (ground floor). Controller remembers current state of all elevators in anytime.

2 Automaton description

$$G = (E, S, f, \Gamma, s_0, S_M)$$

2.1 States S

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

2.1.1 Lift states

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

where:

- l - number of lifts

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

where:
$d \in \{-1, 0, 1\}$

- -1 - down
- 0 - stop

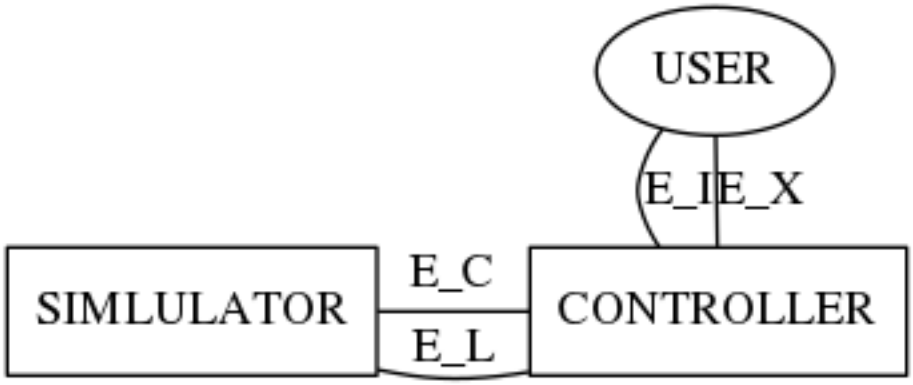


Figure 1: DES graph

- 1 - up

$o \in \{0, 1\}$

- 0 - closed
- 1 - open

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

- i - last reached floor
- n - number of floors

2.1.2 External buttons states

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

where:

- n - number of floors

$$p_i = [g_{u_i}, g_{d_i}]$$

where: $g_{u_i} \in \{0, 1\}$

- 0 - not pushed
- 1 - pushed

$g_{d_i} \in \{0, 1\}$

- 0 - not pushed
- 1 - pushed

2.1.3 Internal buttons states

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

where:

- l - number of lifts

$$q_i = [b_0, b_1, ..., b_i, ..., b_n]$$

$b_i \in \{0, 1\}$

- 0 - not pushed
- 1 - pushed

2.2 Events E

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

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

direction $\in \{0, 1\}$

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

command $\in \{0, 1, 2, 3, 4\}$

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

ACK $\in \{0, 1, 2, 3\}$

2.3 Transition function $f(s, e)$

$0 < i < n$

$$f([0, 0, i], \text{open\_doors}) = [0, 1, i]$$

$$f([0, 1, i], \text{close\_doors}) = [0, 0, i]$$

$$f([0, 0, i], \text{move\_up}) = [0, 1, i + 1]$$

$$f([0, 0, i], \text{move\_down}) = [0, 1, i - 1]$$

2.4 Active event function $\Gamma(s)$

$0 < i < n$

$$\Gamma(0, 0, i) = \{\text{close\_doors}\}$$

$$\Gamma(0, 1, i) = \{\text{move\_up}, \text{move\_down}, \text{open\_doors}\}$$

$$\Gamma(1, 0, i) = \{stop\}$$

$$\Gamma(-1, 0, i) = \{stop\}$$

2.5 Initial state s_0

All 0.

2.6 Marked states S_M

All accepted.

2.7 Protocol

Controller ->Simulation commands:

a. $X : Y$ (where X is number of elevator, Y is number of floor) - let elevator X move to floor Y.

b. $X : s$ (where X is number of elevator) - let elevator X stops.

c. $X : o$ (where X is number of elevator) - let elevator X opens the door.

d. $X : c$ (where X is number of elevator) - let elevator X closes the door

Simulation ->Controller commands:

a. $X : a$ (where X is number of elevator) - elevator X confirms execution of previous controller command.

b. $Y : d$ (where Y is number of floor) - on the floor Y^{th} user push the button to go down.

c. $Y : u$ (where Y is number of floor) - on the floor Y^{th} user push the button to go up.

d. $X : Y$ (where X is number of elevator, Y is number of floor) - inside elevator X user push the button to go to the Y^{th} floor.

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

3 Implementation

3.1 General informations

3.1.1 Controller

Has to be done.

3.1.2 Simulation

Simulation program has been written in Python2.7 scripting language. For preparation user interface we used *Tkinter* standard library. Simulation consists 3 modules: graphics.py, elevator.py and my_threads.py and implements 2 basic classes:

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

and classes inheriting from class *Thread*:

- MoveLift - class executes command type $X : Y$.
- StopLift - class executes command type $X : s$.
- OpenDoors - class executes command type $X : o$.
- CloseDoors - class executes command type $X : c$.
- ListenInstructions - class listen if in input port is any incoming command.
- SendInstructions - class sends to output port command for controller.

Program is basing on 4 main threads (to quarantee simultaneously work):

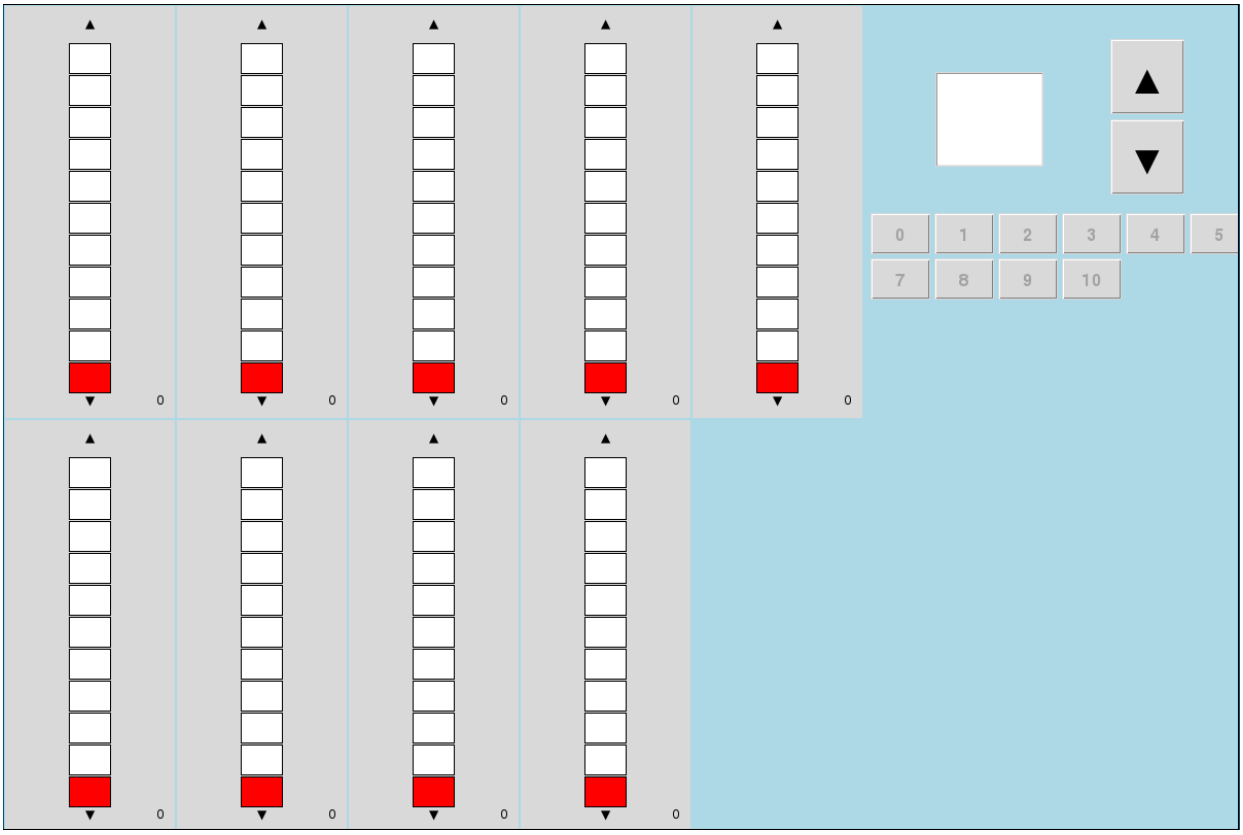
- Main thread - is responsible for configure parameters, start other thread and waiting for them to finish.
- Graphical thread - is responsible for refreshing user interface (drawing elevators and floors) and for sending.
- Listenning thread - is responsible for listenning input port and adding incoming instructions to input queue.
- Sending thread - is responsible for sending command to controller via output port if there is some instruction ready to send in output queue.

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

3.2 Connection

Connection is based on simple sockets. Incoming port for controller is a the same time outcoming port for simulation program. On the other hand outcoming port for controller is an incomming port for simulation.

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


0
0
0
0
0
0
0
0
0
0
0
1
2
3
4
5
7
8
9
10