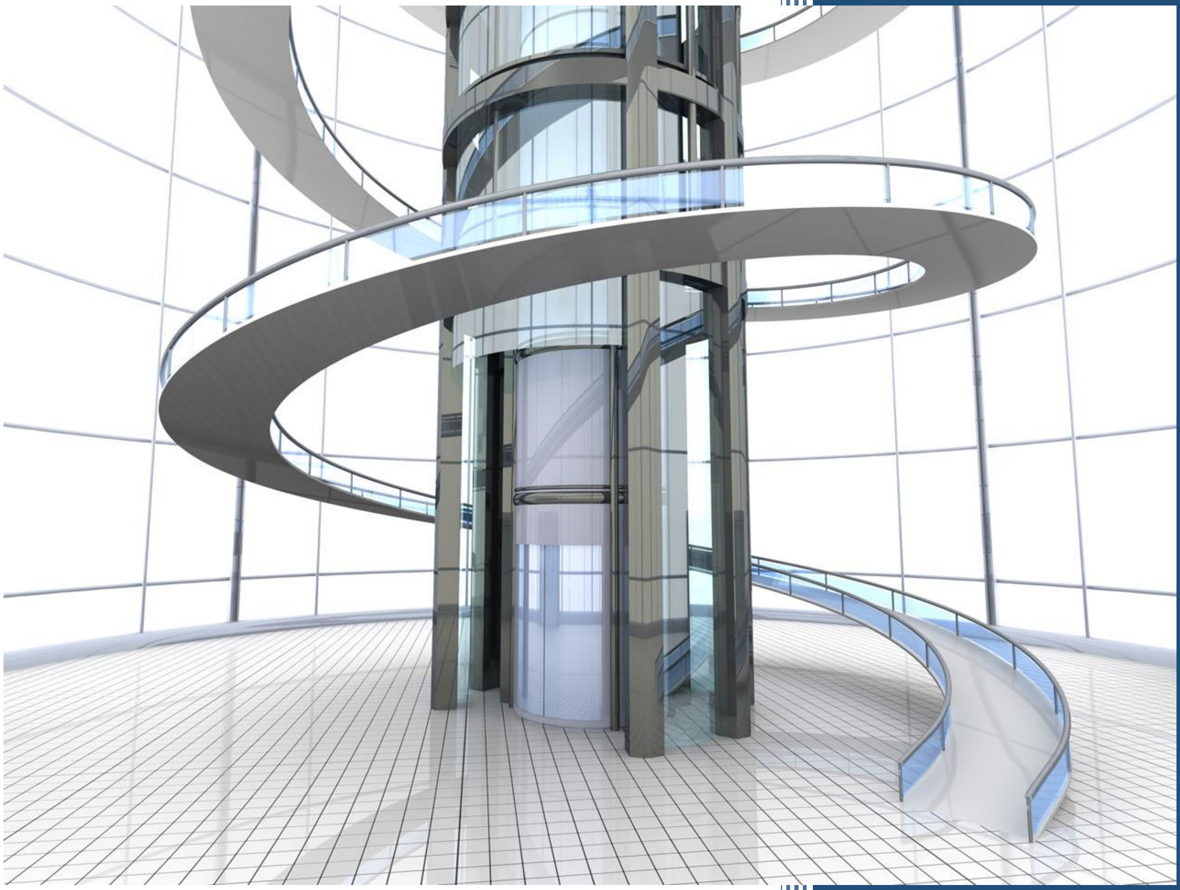
POLITECNICO
MILANO 1863

Automation and Control Laboratory

Automation of an elevator plant



Andrea Verzaglia 823568

Matteo Valenti 823695

Ramtin Kamali 840748

Roberto Ruggiero 836521

Table of contents

Introduction

The purpose of this project is the automation of a small-scale model of an elevator with 4 floors.
The work was organized in different phases, related to the modelling of the discrete event system and the implementation of the control system on a PLC.
The general requirements of an elevator plant are:

- ✓ Provide even service to each floor
- ✓ Minimize how long passengers wait for the elevator to arrive
- ✓ Minimize how long passengers spend to get to their destination floor
- ✓ Serve as many passengers as possible

First of all, an accurate analysis of the system architecture has been made, in particular the characterisation of the output/input signals of the PLC. It was interesting to understand the function of each component of the elevator plant, starting from a predefined PLC I/O configuration. Beside of the basic commands, a series of experiment has allowed to evaluate the behaviour of the plant in safety condition, i.e. limit switches working mechanism.
Subsequently, as it is used for the development of a software, a sequence of requirements has been defined starting from the basic logic (no memory for requests but first) up to the logic with storage (store all request at any time), increasing step by step the complexity level.
The modelling of the system, based on the requirements, was the key point; the modelling strategy, selected among the possible available in the discrete system literature, was the *Statechart*.
The next step was the implementation, that is the coding. It is straightforward to think that the modelling and implementation stages cannot be considered completely separated. In this context, in fact, the main difficulty in the design of control system is to touch all possible situations, which are not intuitive and can be analysed only through experimental results. The simulation of different realistic cases (testing) has allowed to avoid state of the system, which are not controlled in a suitable way or have not been analysed theoretically in modelling phase yet. In doing so, an important role was played by the graphical simulation of the elevator plant, defined in the visualization section of the control software.
Subsequently, some additional features were taken into account, like the presence of an obstacle, during the closing doors phase, the emergency STOP, the blackout, the fire, the people loading and unloading management.
The final operation was the analysis of results.

1. Description of the system

This section presents a description of the system architecture, which is schemed in the block diagram below. The main components are: the elevator plant, the Programmable Logic Controller and the related software.

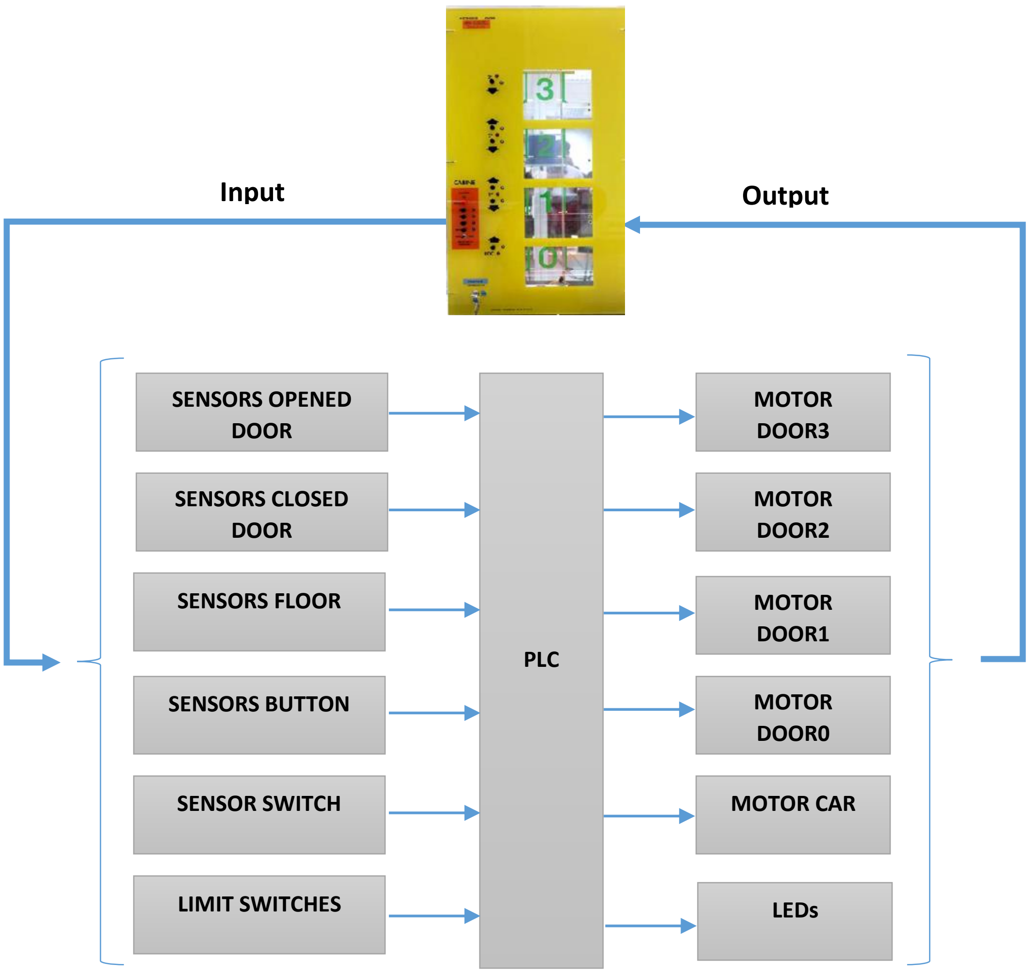


Figure 1

1.1 Structure of the Elevator Plant

The main elements of the elevator plant are:

1. Buttons:
   - P0, P1, P2, P3 (calling buttons inside the cabin)
   - PD1, PD2, PD3 (calling buttons outside of the cabin for going down)
   - PS0, PS1, PS2 (calling buttons outside of the cabin for going up)
   - STOP (stop button in order to stop the elevator procedure)

2. LEDs
   - LPC0, LPC1, LPC2, LPC3 (red light signs of car presence in specific floor)
   - LPD0, LPD1, LPD2, LPD3 (yellow light signs of descending calls)
   - LPS0, LPS1, LPS2, LPS3 (yellow light signs of ascending calls)
   - LL (yellow light representing the light inside the cabin)

3. Manual Switch
   - SWITCH (simulating an obstacle to door closure)

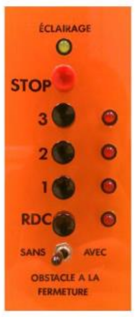


Figure 2: LEDS and buttons inside the cabin

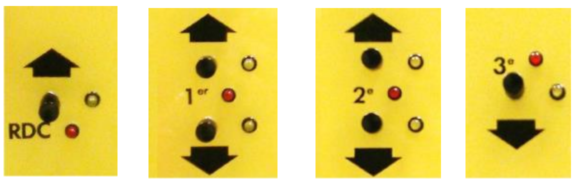


Figure 3: LEDs and buttons outside the cabin

4. Sensors
   - PA0, PA1, PA2, PA3 (open-door optical sensor)
   - PC0, PC1, PC2, PC3 (closed-door optical sensor)
   - RPC0, RPC1, RPC2, RPC3 (car presence optical sensor)

Figure 4: Optical sensor

5. Electromechanical actuators (one for the cabin and four for each door)

Figure 5: DC motor

6. Motor Signal Commands
   - AP0, AP1, AP2, AP3 (opening door signal)
   - CP0, CP1, CP2, CP3 (closing door signal)
   - RPC0, RPC1, RPC2, RPC3 (car presence detector)
   - SALI, SCENDI (car movement signal)

7. Safety Key and limit switches

Figure 6: Limit switch

1.2 PLC

Figure 7: PLC

A PLC is a programmable controller, so a digital device, that can perform various control action depending on the running application and the hardware configuration. Thanks to its robustness it's mainly used in control applications where environment is noisy.
The device used in our application is ABB PM-571 version of the AC500 series, and can be joint with additional modules like digital or analogic I/O, timers or communication interfaces. However, because our plant has a totally digital interface only digital I/O modules are used. It supports the IEE standard, so it can be programmed in all its five languages. Moreover, it provides an LCD display, an operator keypad, an SD card slot, and two integrated serial interfaces.

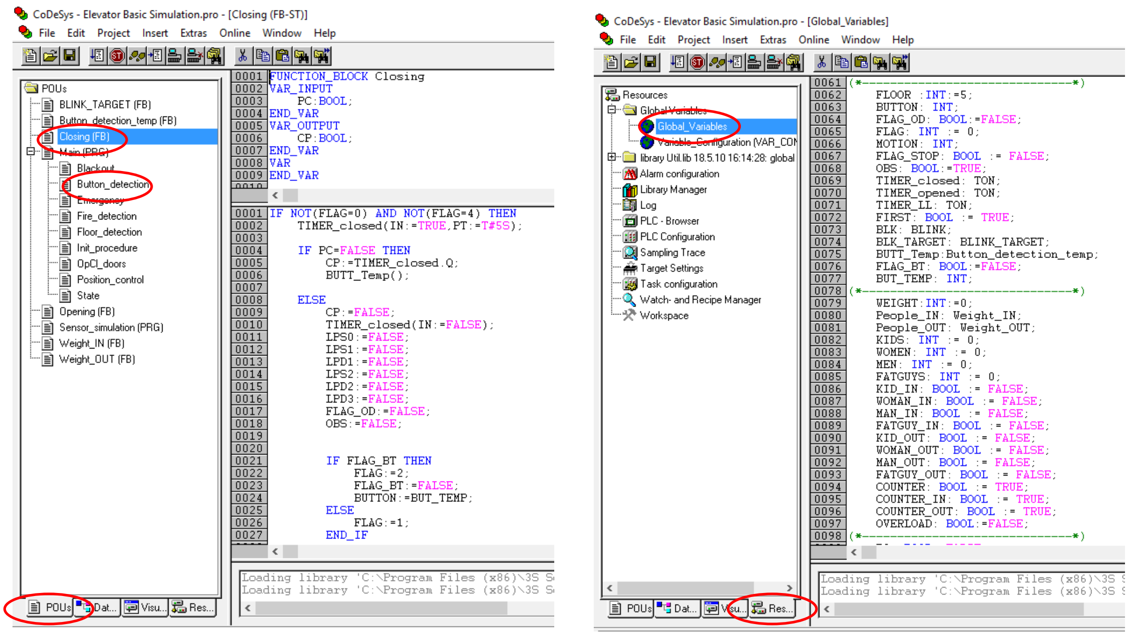
1.3 CoDeSys

Code development has been carried out with the environment PS501 CONTROL BUILDER PLUS 2.2.0 Based on CoDeSys Version 2.3.9.34; it supports IEC 61131-3 standard allowing to write code in Sequential Function Chart, Ladder Diagram, Structured Text and Function Block Diagram languages. We have chosen the STRUCTURED TEXT. The project is structured in more sections:

- ✓ "Program Organization Units" (POUs) which collects all code instructions like Function Blocks, Actions and Programs. Each POU consists of a declaration part and a body. The body is written in one of the IEC programming languages.
- ✓ "Visualization" which allows to create a graphical interface of the system;
- ✓ "Resources" that manages mainly the technical specifications of the PLC (task configuration) and the global variables declaration and monitoring.

The PLC_PRG (Main) is a special predefined POU (called exactly once per control cycle); each project must contain this special program.
The code is divided in different function blocks (FB) and actions. This approach lets to recall more times the same function and to make the code more understandable. An action can be called only inside the function, where is defined, while the function block can be called everywhere, provided that it is declared as function type.

In the following figure, examples of action and FB are respectively *Button_detection* and *Closing* (FB). Another significant difference between an action and a function block is that the declaration of variables is defined only for the latter one; the action inherits the variables of the function, which belongs to.



```
FUNCTION_BLOCK Closing
VAR_INPUT
    PC:BOOL;
END_VAR
VAR_OUTPUT
    CP:BOOL;
END_VAR
VAR
END_VAR
```

```
IF NOT(FLAG=0) AND NOT(FLAG=4) THEN
    TIMER_closed(IN:=TRUE,PT:=T#5S);

    IF PC=FALSE THEN
        CP:=TIMER_closed.Q;
        BUTT_Temp();

    ELSE
        CP:=FALSE;
        TIMER_closed(IN:=FALSE);
        LPS0:=FALSE;
        LPS1:=FALSE;
        LPD1:=FALSE;
        LPS2:=FALSE;
        LPD2:=FALSE;
        LPD3:=FALSE;
        FLAG_OD:=FALSE;
        OBS:=FALSE;


        IF FLAG_BT THEN
            FLAG:=2;
            FLAG_BT:=FALSE;
            BUTTON:=BUT_TEMP;
        ELSE
            FLAG:=1;
        END_IF
```

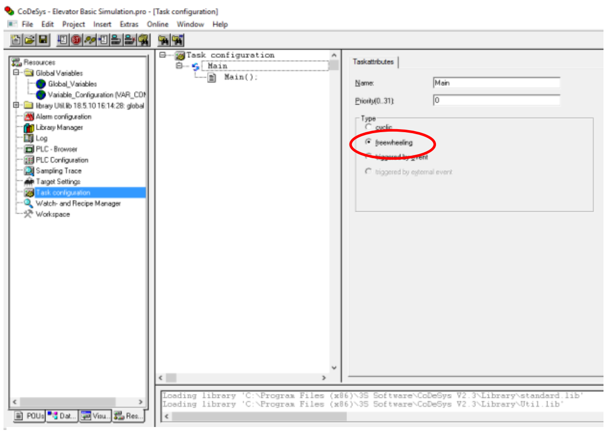
```
Loading library 'C:\Program Files (x86)\3S S
Loading library 'C:\Program Files (x86)\3S S
```

```
(*-----------------------------------*)
    FLOOR :INT:=5;
    BUTTON: INT;
    FLAG_OD: BOOL:=FALSE;
    FLAG: INT := 0;
    MOTION: INT;
    FLAG_STOP: BOOL := FALSE;
    OBS: BOOL:=TRUE;
    TIMER_closed: TON;
    TIMER_opened: TON;
    TIMER_LL: TON;
    FIRST: BOOL := TRUE;
    BLK: BLINK;
    BLK_TARGET: BLINK_TARGET;
    BUTT_Temp:Button_detection_temp;
    FLAG_BT: BOOL:=FALSE;
    BUT_TEMP: INT;
(*-----------------------------------*)
    WEIGHT:INT:=0;
    People_IN: Weight_IN;
    People_OUT: Weight_OUT;
    KIDS: INT := 0;
    WOMEN: INT := 0;
    MEN: INT := 0;
    FATGUYS: INT := 0;
    KID_IN: BOOL := FALSE;
    WOMAN_IN: BOOL := FALSE;
    MAN_IN: BOOL := FALSE;
    FATGUY_IN: BOOL := FALSE;
    KID_OUT: BOOL := FALSE;
    WOMAN_OUT: BOOL := FALSE;
    MAN_OUT: BOOL := FALSE;
    FATGUY_OUT: BOOL := FALSE;
    COUNTER: BOOL := TRUE;
    COUNTER_IN: BOOL := TRUE;
    COUNTER_OUT: BOOL := TRUE;
    OVERLOAD: BOOL:=FALSE;
(*-----------------------------------*)
```

```
Loading library 'C:\Program Files (x86)\3S S
Loading library 'C:\Program Files (x86)\3S S
```

Figure 8: Codesys environment

In the upper part of each function block there is the declaration of local variables or I/O variables. The declaration of global variables is in "Resources" section on the bottom (*Figure 8*: Codesys environment).
It is important to underline that the execution of the program (Task configuration) is set to freewheeling (*Figure 9*: Task configuration), which allows to start a new execution cycle, as soon the previous one is finished.


CoDeSys - Elevator Basic Simulation.pro - [Task configuration]
File Edit Project Insert Extras Online Window Help
Resources
Global Variables
Global_Variables
Variable_Configuration (VAR_CON
library Util.lib 18.5.10 16:14:28: global
Alarm configuration
Library Manager
Log
PLC - Browser
PLC Configuration
Sampling Trace
Target Settings
Task configuration
Watch- and Recipe Manager
Workspace
POUs
Dat...
Visu...
Res...

```
Task configuration
    Main
        Main();
```

Taskattributes

Name: Main

Priority (0..31): 0

Type: cyclic [✓ freewheeling] triggered by event triggered by external event

```
Loading library 'C:\Program Files (x86)\3S Software\CoDeSys V2.3\Library\standard.lib'
Loading library 'C:\Program Files (x86)\3S Software\CoDeSys V2.3\Library\Util.lib'
```

Figure 9: Task configuration

2. Basic logic

2.1 Requirements

In this section we present the specifications which the system has to satisfy.
The initialization is the first procedure which has to be executed, as the program is downloaded on PLC, in both versions of the software logic. It allows to bring the plant to a normal working condition, that is the cabin in a floor and all doors closed. To obtain this result, a floor detection and a check of the doors state has to be performed. Notice that this first operation can be viewed as a recovery operation after an out of service condition.
The basic idea of the first version of the software, modelled based on the logic without storage, is the following: the elevator should continue working regardless the pushing of buttons, different from the first one, that is related to the actual destination floor. As consequence, there is no pushed buttons memorization while the elevator is busy. For this goal, a button detection process has to be defined, which detects a pushed button either internal or external. This operation is deactivated while the cabin is moving, but it works if the plant is in idle condition or during the opening/closing process to allow the user to select the requested floor.
Notice that a floor detection procedure has to be executed in every working conditions, to know the cabin position. Moreover, it is important to underline that in this logic, the external buttons of each floor (for instance 1↑ and 1↓) have the same function.
The LEDs should be managed to indicate whether the elevator is free or not. To do so, all external yellow LEDs are turned on when the elevator is busy. In this way the user finds out when it is possible to call the elevator itself.
The red ones instead are used to indicate the presence of the car in a floor. Moreover, as soon as a button inside the cabin is pushed during the opening/closing procedure, the corresponding red LED is switched on and a blinking of the same LED occurs, when the cabin starts to move to show the current destination floor.
In the opening and closing processes, a timer is used in order to make a delay before the execution of opening and closing. It means as the cabin reaches the target, the corresponding door starts to open after 2 seconds to simulate the arrest of the cabin, and closes after 5 seconds to allow the people entering/exiting.
There are two particular cases which have to be taken into account: the obstacle presence and the stop button pushing.
Let us consider the presence of obstacle first. If an obstacle is placed during the doors closing process, doors have to stop to close and start to open immediately (timer is deactivated); they cannot be closed until the obstacle is removed (in this condition the timer of closing is reset).
Regarding the stop button, if it is pushed while the elevator is moving up or down, the car has to stop and the system waits for another button call (button detection). It is straightforward that the current requested call is reset. This situation is declared turning ON all external yellow LEDs.
The reopening process is performed not only in presence of an obstacle, but also when the STOP button or the buttons related to the current floor are pushed.
Finally, the LL LED, representing the light inside the cabin is switched off, if the elevator remains in an idle condition for a certain time, to save energy consumption.

2.2 Software design: Modelling and code description

Discrete event systems' behaviour can be modelled through different formal languages from the UML standard (Unified Modelling Language), each one of them puts in evidence different aspects of a system or is suitable for particular kinds of problems.
Because of the high number of states, which characterize our plant to take into account all possible situations, we opted for the Statechart models.
Statecharts allow us to think about a "state" of a system at a particular point in time and characterize the behaviour of the system based on that state.
We can use this technique to model and design software systems (in our case the software for PLC) by identifying what states the system can be in, what inputs or events trigger state transitions, and how the system will behave in each state. In this model, we view the execution of the software as a sequence of transitions that moves the system through its various states.
The software used to define Statecharts is the Matlab-Simulink Stateflow tool. It is an environment for modelling and simulating combinatorial and sequential decision logic based on state machines and flow charts. Stateflow lets you combine graphical and tabular representations, including state transition diagrams, flow charts, state transition tables, and truth tables, to model how your system reacts to events, time-based conditions, and external input signals. It also includes state machines animation and static and run-time checks for testing design consistency and completeness before implementation.
Formally a Statechart is a hierarchical model of a system and introduces the concept of a composite state (also called nested state). This permits to overcome the state explosion problem, which could occur using a finite state machine formalism.
From the graphical point of view, the basic entities of the Statechart diagram are:

- ✓ States: represented by rectangles with rounded corners. As said before states may be composite and can include a group of states or even another "sub-level" Statechart (sub-chart). Formally a default start state is required at every hierarchical level.
- ✓ Transitions: shown as an arrow between two states. The guard, that is a boolean logic condition, can also be assigned to a transition, so it can take place only if the guard is TRUE. The guard is expressed in square brackets alongside the arrow. Eventually some assignments or actions (denoted in curly brackets) can be performed when the branch is passed through.
- ✓ Junction: which enables representation of different transition paths for a single transition.

In order to make the design easier, a modular approach is considered, which allows to analysed each operation (module) separately, and at the same to describe the system from the external point of view (high level) to the internal ones (sub-charts).
In the following for each module a representation and a description of the Statechart are given. In order to make the report clearer, some internal sub-charts are not inserted. However, they are provided in the Simulink file attached **HERE**.

Module 1: Elevator plant (*Main*)

The basic idea of this program is that the value of FLAG variable defines the action to be called. To do that, a CASE structure is used. The possibles values assigned to FLAG are:

0 → Init_procedure();
1 → Button_detection();
2 → Position_control();
3 → OpCl_doors();
4 → Emergency();

For time being, it is not important how the FLAG variable changes inside each action (a detailed description is given in the following sections). As it is mentioned in the description of the Statechart, the *Floor_detection* function is called at each iteration, regardless of the FLAG value. The *Blackout* and *Fire_detection* actions are also executed in every condition. The way in which they work is described in the 5. Additional features.

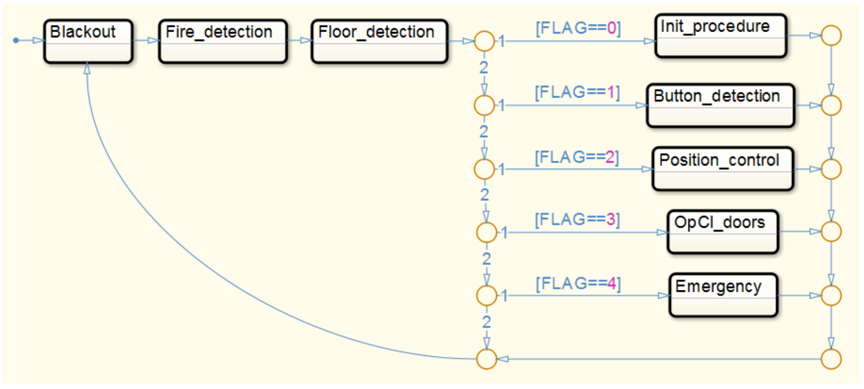


Figure 10: Elevator plant Statechart

Module 2: Floor detection (*Floor_detection*)

This function is based on the check of sensors (RPC0, RPC1, RPC2, RPC3), which detects the presence of the cabin in the corresponding floor. As the cabin is at a floor, the sensor variable becomes TRUE, the integer (0, 1, 2, 3), representing the number of the floor is assigned to the variable FLOOR and the LED (LPC0, LPC1, LPC2, LPC3) corresponding to such floor is switched on (its value is set to TRUE). It is obvious that there is no chance to have more than one sensor variable equal to TRUE. The switching off of the LED is not performed if the corresponding internal button, identified by the BUT_TEMP variable, has been pushed, to show the target floor during the opening/closing process.

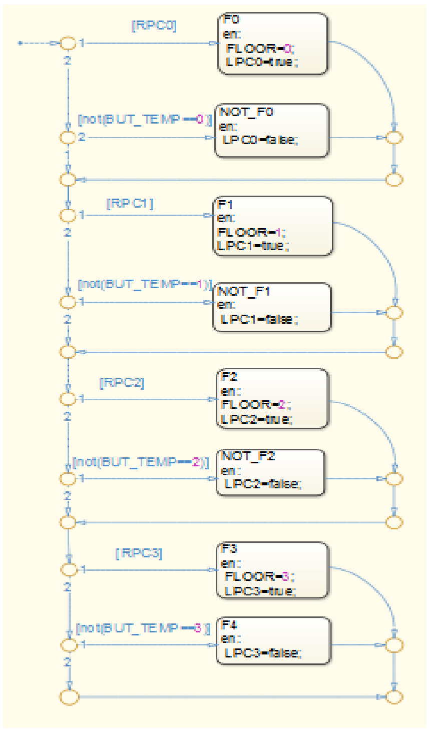


Figure 11: Floor_detection

Module 3: Initialization (*Init_procedure*)

In the initial procedure, we call the blinking function (BLK type, defined in the Util.lib/signals library) which causes all yellow lights outside of the cabin starting blinking and it lasts until the end of this procedure. To do so, we assign boolean value TRUE or FALSE to the blinking function's enabling input, by checking if the main FLAG is still zero, i.e. if the program still in the initial procedure.

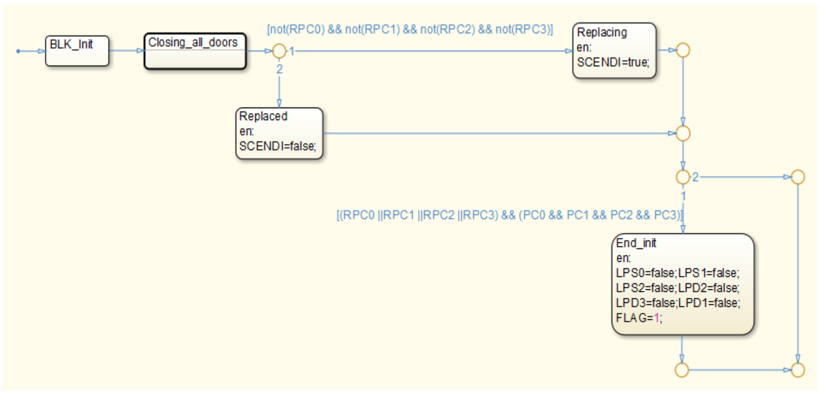


Figure 12: Init_procedure Statechart

As this procedure starts working the closing of all doors is performed.
The description of the closing function is given in the following.
Furthermore, it is verified if the cabin is at any floor by checking RPC sensors, which declare the presence of the cabin in a specific floor and in the case that none of them is TRUE, the cabin starts going down until it reaches a floor. The going down states is executed by assigning TRUE to SCENDI variable.
As the cabin is at one of the floors and all doors are closed, the condition of the next test is satisfied and it is executed. By this, the integer 1 is assigned to FLAG, which means in the next cycle of program execution the initial procedure is skipped and button detection function is called.
Finally, all yellow lights are turned off. It is worth that FLAG will not be 0 anymore.

Module 4: Button detection

Button_detection

The key idea of this function is: as a button is pushed, the corresponding sensor variable becomes TRUE and the integer (0, 1, 2, 3), representing the number of the target floor is assigned to the variable BUTTON. Once a button is pushed (upper/lower external button or internal one), the main FLAG variable is set to 2, which means in the next cycle of program execution the button detection procedure is skipped and the *Position_control* function is called.
Notice that the BUT_TEMP variable is set equal to BUTTON. Actually this command is not necessary since this variable is used only in the button detection operation performing during the closing/opening process but it is done just to assign an initialization value.
The main FLAG remains 1 in only two cases: no button is pushed or STOP button is pushed.
In these conditions, the system waits for the pressure of whatever button, different from STOP.
A particular condition occurs when the STOP button is kept pushed for 3 seconds: the FLAG_EMG is set to TRUE and the FLAG is set to 4 so that in the next iteration the *Emergency* function is called. A detailed description of this condition is given the Additional features.
Moreover, in order to save energy a timer is used to switch off the “eclairage” LED (LL) in case no buttons are pushed in 10 seconds and the stop button has not been pushed.

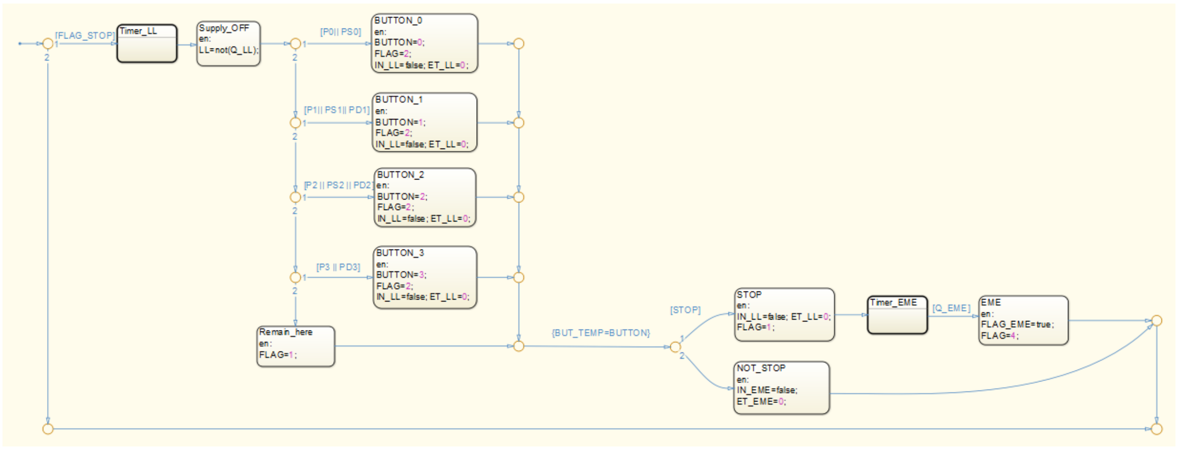

[FLAG_STOP]
Timer_LL
Supply_OFF en: LL=not(Q_LL);
[P0|| PS0]
BUTTON_0 en: BUTTON=0; FLAG=2; IN_LL=false; ET_LL=0;
[P1|| PS1|| PD1]
BUTTON_1 en: BUTTON=1; FLAG=2; IN_LL=false; ET_LL=0;
[P2 || PS2 || PD2]
BUTTON_2 en: BUTTON=2; FLAG=2; IN_LL=false; ET_LL=0;
[P3 || PD3]
BUTTON_3 en: BUTTON=3; FLAG=2; IN_LL=false; ET_LL=0;
Remain_here en: FLAG=1;
{BUT_TEMP=BUTTON}
[STOP]
STOP en: IN_LL=false; ET_LL=0; FLAG=1;
Timer_EME
[Q_EME]
EME en: FLAG_EME=true; FLAG=4;
NOT_STOP en: IN_EME=false; ET_EME=0;

Figure 13: Button_detection Statechart

Button_detection_temp

The *Button_detection_temp* function block, called in the opening/closing process through the BUTT_Temp command, can be considered a small version of the *Button_detection*. It is composed by a check associated only to cabin buttons, which is executed if no button has been already pushed. In this last case FLAG_BT is set to TRUE.

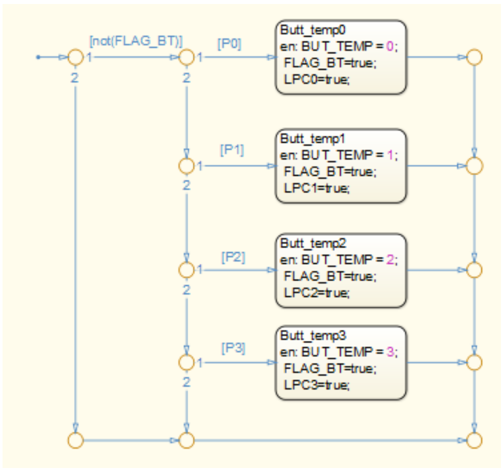

[not(FLAG_BT)]
[P0]
Butt_temp0 en: BUT_TEMP = 0; FLAG_BT=true; LPC0=true;
[P1]
Butt_temp1 en: BUT_TEMP = 1; FLAG_BT=true; LPC1=true;
[P2]
Butt_temp2 en: BUT_TEMP = 2; FLAG_BT=true; LPC2=true;
[P3]
Butt_temp3 en: BUT_TEMP = 3; FLAG_BT=true; LPC3=true;

Figure 14: Button_detection_temp Statechart

Module 5: Position control (*Position_control*)

This action is related to the movement of the cabin. As it is called, all external LEDs are turned on to declare that the elevator is busy. Subsequently, we check if the target floor represented by the BUTTON variable is reached or not. Depending on the relation between the BUTTON and the FLOOR the motor is driven in a specific direction by assigning TRUE value to the output variables SALI or SCENDI. In movement conditions, i.e. going up and going down, the function *BLK_TARGET* is called to show the destination, and the variable MOTION is set respectively to 0 or 1. When the requested floor is reached or STOP is pushed, the cabin is stopped setting the variables SALI and SCENDI to FALSE and red LEDs are turned off. Subsequently, FLAG is set to 3 in normal working condition (target reached) or set to 1 in "STOP condition". This last one is identified by setting the FLAG_STOP variable to TRUE; as soon as the cabin starts to move again the same variable is reset to FALSE.
Depending on the FLAG value, in the next iteration, the *Button_detection* or the Opening/Closing operation will be performed.
The third conditional instruction (**AFTER_STOP** state in the figure below) is associated to the specific case in which the button pushed after the stop corresponds to the last floor which the elevator is passed through (FLOOR). The reason for which this case is managed separately is that, when BUTTON=FLOOR the cabin would not move. Knowing the direction of movement before the pressure of stop, the car can be placed in the previous floor, forcing the moving.

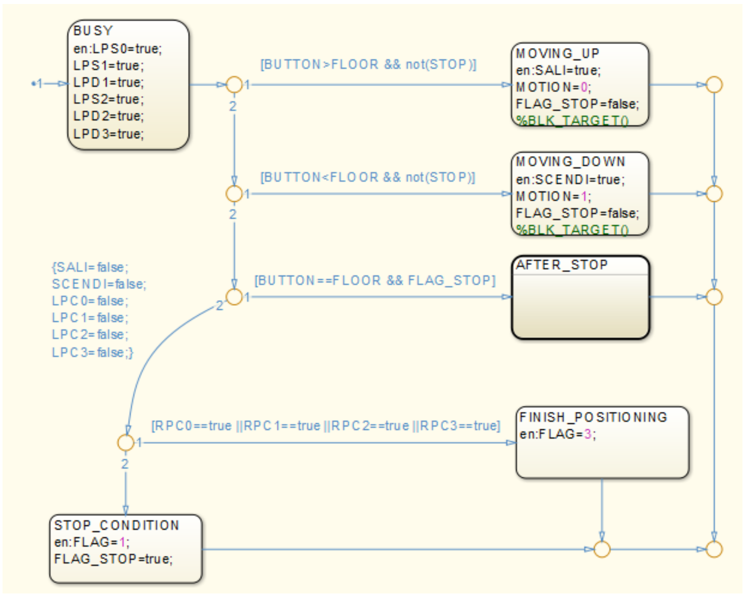


Figure 15: Position Control Statechart

Module 5: Opening/Closing Process (*OpCl_doors*)

This procedure starts working as the main FLAG becomes 3.
A CASE structure is in charge of the execution of this function block which considers the variable FLOOR and executes opening and closing process in the corresponding floor. Let's take a better look at one case, assuming that the FLOOR variable is 0 (in the diagram denoted as OpCl_0).
As first FLAG_OD is checked to detect whether the door is open: if FALSE, hence door is closed, the opening procedure is performed (calling the Opening function). If TRUE, a further condition is evaluated, so as to guarantee that the door closes only in a "safety condition": there is not an obstacle (SWITCH=FALSE), no stop condition (STOP=FALSE), the overall load's weight (WEIGHT

variable) is less or equal to 320 Kg, and any button related to the floor (either inside or outside the cabin) is not pushed.
If above conditions are all verified the Closing function is called, if not so closing action is stopped by forcing CP0 and FLAG_OD to FALSE.
Note that the forcing assignment on FLAG_OD is made for opening the door when the closing procedure is not finished due to an obstacle, a stop or a pushing button. Without that action normally FLAG_OD would be set to FALSE only at door completely closed, so entering OpCl, at the first test on FLAG_OD would be TRUE and the opening not be executed leading to a not proper behaviour.

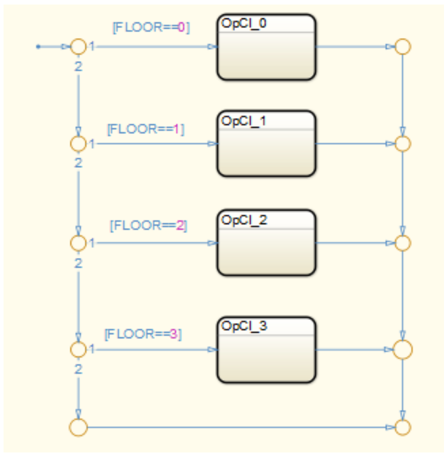


Figure 16: OpCl_doors Statechart

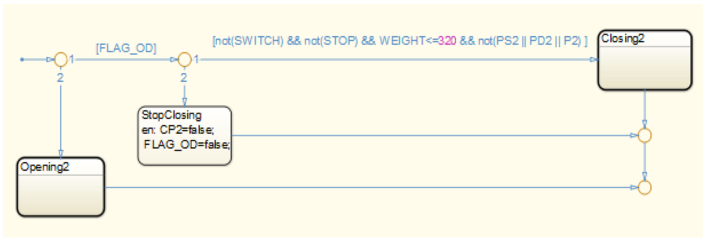

[FLAG_OD]
[not(SWITCH) && not(STOP) && WEIGHT<=320 && not(PS2 || PD2 || P2)]
Closing2
StopClosing
en: CP2=false;
FLAG_OD=false;
Opening2
1
2

Figure 17: OpCl_0 Statechart

Module 6: Closing

The function is structured in such way that depending on the working condition, a different closing process is performed. It can distinguish two cases:

- normal mode FLAG=3
- initialization/emergency→FLAG=0/FLAG=4

In normal mode, the closing starts after 5 seconds due to the action of Timer_closed function (TON type, defined in standard.lib/Timer library). It works as following: as the input changes to TRUE, it takes PT seconds for the output to get TRUE and changes to FALSE as the input changes its value to FALSE.
When this function block is called, if the corresponding closed-door sensor (PC) indicates that the door is not closed, as the timer's output gets TRUE the CP command for closing door motor gets on and closing starts.
During this process a button detection function *BUTT_Temp* (*Button_detection_temp* type) is called. This represent the realistic situation in which the user pushes the target button before the door is completely closed. In this specific case the FLAG_BT allows to go directly to the position control in the next iteration, since a button is already pushed (BUTTON=BUT_TEMP).
Otherwise, i.e. the corresponding door is closed, the closing command is switched off (CP=FALSE) as well as the timer's input. Furthermore, only in normal mode we set to FALSE both FLAG_OD and OBS to activate in the right way the opening process as the next floor requested is visited, and also turn off all lights (free elevator). Finally, the integer 1 is assigned to the main FLAG, which means in the next cycle of program execution, the button detection procedure will be performed.

During the initialization procedure or the emergency condition the closing process works in the same way but no timer is used because it is not related to the people entering and exiting. The closing door of the ground floor (0) Statechart is represented in the following figure.

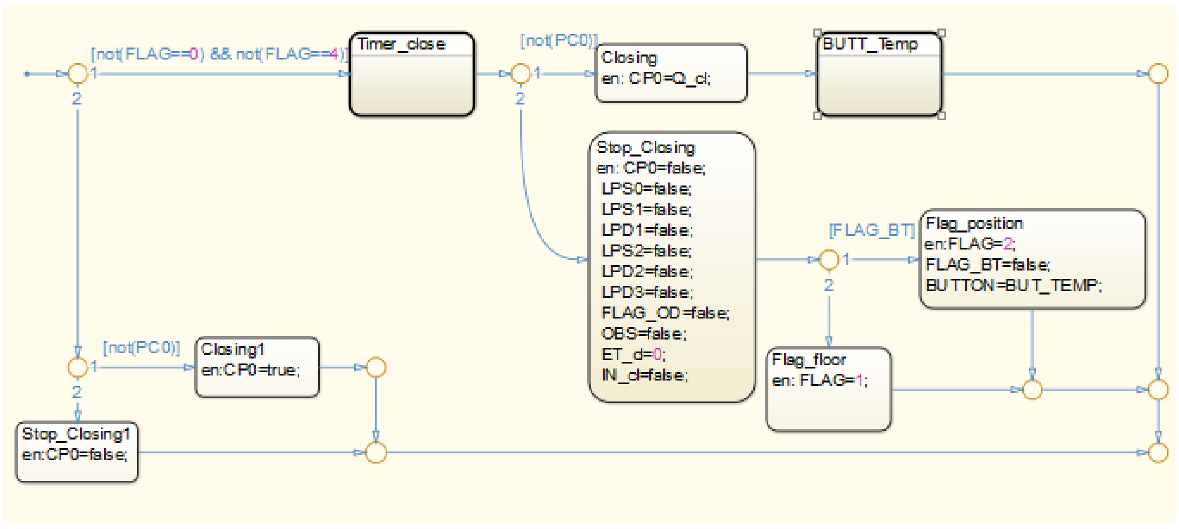


Figure 18:Closing Statechart

Module 7: Opening

At first the two functions *People_IN* (*Weight_IN* type) and *People_OUT* (*Weight _OUT* type) are called. The idea is to simulate the entering of people with different weight to activate the closing process and, as consequence the movement of the car, only if we are under the boundary condition. This one is represented trough a warning message (see the 5.1 People loading and unloading requirements).
In the following part we check whether the OBS flag is TRUE or not. The OBS variable is used to distinguish the first standard opening process (OBS=FALSE) and the reopening due to the insert of an obstacle, the stop button or any button related to the floor (either inside or outside the cabin) pushing (OBS=TRUE). In these last cases we want to open the door without waiting for timer (the cabin is already at that floor so we do not need to simulate the arrest of the cabin itself).
Let's analyze in details each case. In standard opening the TIMER_opened is called and TRUE is assigned to its input. Moreover, if PA is FALSE, i.e. the door has not been opened yet, opening timer's output is assigned to AP which is the command for turning on the motor in order to open the door. While when the PA is TRUE, which means the door is opened, the motor is stopped by assigning FALSE to AP variable, the closing and opening timer are reset, and both FLAG_OD and OBS are set to TRUE.
Otherwise, in the case that OBS flag is TRUE the same actions are performed, without using any timer. On the other hand, if the door is open, we turn off the opening motor, we make FLAG_OD equal to TRUE and closing timer is reset.
As in the closing procedure, a button detection is made during the opening, apart from the emergency situation (FLAG=4).
The Statechart of the opening door at the ground floor is shown in the figure below.

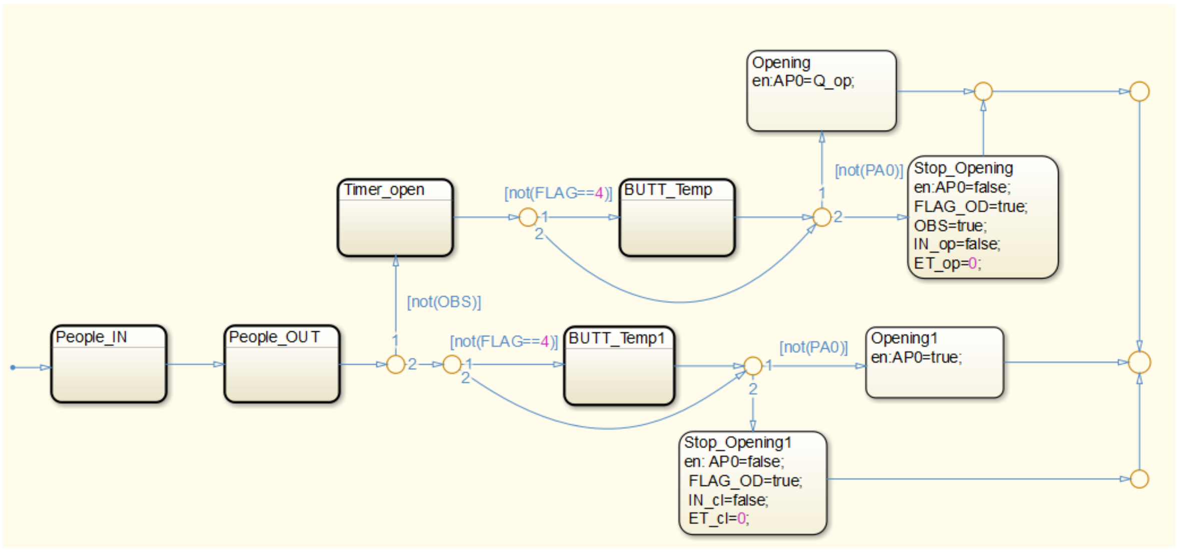


Figure 19: Opening_0 Statechart

Module 8: Blinking Target (*BLK_TARGET*)

This simple function block is used to declare the destination related to the car movement, i.e. the requested floor. Depending on the BUTTON value a blinking of the corresponding floor red LED is activated. Notice that this function is not disabled explicitly but it is simple forced the value TRUE or FALSE to the LEDs according to the working condition.

Notice that all PLC code related to this logic is furnished **HERE**.

3. Logic with Storage A

3.1 Requirements

This section is about the logic with storage. As already mentioned in the previous section the first procedure, which has to be executed as the program is downloaded on PLC (also when it is reset) is the initialization.
Also in this case, a description of the main specifications, which the system has to satisfy, once the initialization is finished, is given in the following.
The significant difference of this second version of the software, modelled based on the logic with storage, is that while the elevator is working (during the car replacement and the opening/closing of the door), the pushed buttons, different from the first one, have to be memorised.
The mechanism is based on the SCAN algorithm. When a new request arrives while the system is idle, the initial movement will be in a direction (up or down), which depends on the current floor and the first pushed button. As additional requests arrive, they are serviced only in the current direction until the car reaches the farthest requested floor. When this happens, the direction of the car reverses, and the requests that were remaining in the opposite direction are serviced. The system keeps to work until all demands are satisfied.
In this logic, external buttons are taken into account in different way, depending on the direction of the car movement. The request is satisfied only if the "direction" of the button is the same with respect to the current one. That is why, despite of the previous logic, the external LEDs are switched on only when the corresponding button is pushed and they are switched off when the request is serviced. As the yellow LEDs, the red ones are used to declare either an outstanding request, corresponding to a button inside the cabin and the presence of the cabin at a floor.
Obviously the floor detection procedure has to be actuated in every working conditions.
The behaviour of the system in presence of an obstacle and the stop button pushed remains the same. In the latter case all memorised calls have to be reset.

3.2 Software design: Modelling and code description

The code and the modelling of this logic present similar features to the previous version, for this reason we will focus on the description of the main differences (the elevator plant, the initialization procedure are not taken into account).
Also for this logic the complete Simulink Statechart is given **HERE**.

Module 1: Floor detection (*Floor_detection*)

The basic idea of this action remains the same, i.e. turning on and off the red LEDs and to assign the suitable value to the FLOOR variable, to declare the presence of the cabin at a floor. However, the switching off of a LED is not performed if the corresponding button inside the cabin has been pushed, regardless the sensor value. As it will be described in the Button detection section, if a button inside the cabin is pushed, the corresponding element of a proper boolean array is set to TRUE.

Module 2: Button detection (*Button_detection*)

Roughly speaking, the role of this function is to store any pushed button in any state and also to choose in which order requests should be satisfied (priority algorithm) by assigning a proper value to BUTTON variable. This value corresponds to the maximum or minimum, i.e. the highest or lowest floor requested, depending if the cabin has to go up or down. The updating of maximum or minimum is executed in different way, according to the specific condition.
In order to store buttons, three different boolean arrays are defined, which are associated respectively to:

- ✓ Buttons inside the cabin (BUTTON_CAR[..])
- ✓ Buttons outside the cabin declaring the going up (BUTTON_UP[…])
- ✓ Buttons outside the cabin declaring the going down (BUTTON_DOWN[…])

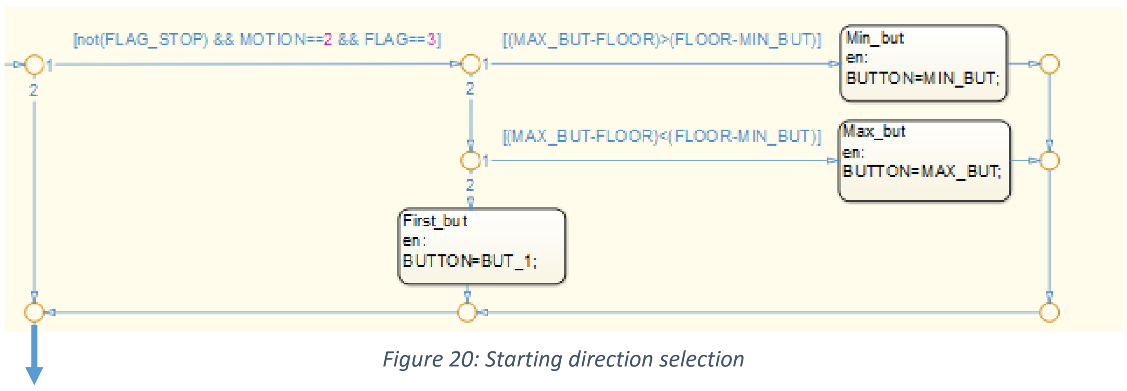
Moreover, it is worth to mention that we design this function block such that it can be called in any other actions and function blocks without interfering the running part.
Let's start to describe in detail how the function works.
Before anything, as in the previous logic, in order to save energy, the "éclairage" LED (LL) is switched off if the following conditions are satisfied: there are no outstanding request and so we are waiting for a button request (FLAG=1), no buttons are pushed in 10 seconds and the stop button has not been pushed (NOT(FLAG_STOP)). For this goal, we define a timer which is enabled after 10 seconds to turn the LED off.
Beside, we initialize three variables BUT_1, BUTTON and BUT_TEMP with current recognized floor value (FLOOR).
Subsequently, we check if one or more button are pushed, in particular internal buttons (Px=TRUE for x=0,…,3), going up declaration buttons (PSx=TRUE for x=0 ,...,3) and going down declaration buttons (PDx=TRUE for x=0 ,…,3). If the "pushing condition" is satisfied, we assign TRUE value to the corresponding boolean array element. Moreover, a BUT_TEMP variable is defined in order to declare the corresponding floor to the last pushed button. We also turn on the corresponding LED, which is red or yellow if an internal or external respectively button is pushed, by assigning TRUE to LPCx or LPSx/LPDx for x=0,..,3. Let's consider an example: let us assume the button 2 inside the cabin, is pushed. As consequence BUTTON_CAR[2] is set to TRUE, BUT_TEMP is set to 2 and LPC2 is set to TRUE.

CODE VARIABLES	REAL BUTTONS
BUTTON_CAR[0]	P0
BUTTON_CAR[1]	P1
BUTTON_CAR[2]	P2
BUTTON_CAR[3]	P3
BUTTON_UP[0]	PS0
BUTTON_UP[1]	PS1
BUTTON_UP[2]	PS2
BUTTON_DOWN[1]	PD1
BUTTON_DOWN[2]	PD2
BUTTON_DOWN[3]	PD3

Table 1: Variables-buttons correspondence

Notice that, in Matlab Statechart, the index starts from 1 and not from 0.
After that, the PUSHED flag, which declares the occurrence of pushing at least one button is set to TRUE to further usage. The next assignments are related to the "starting direction selection", which is described in the following. The last statement is devoted to a variable called UPDATE. The aim of this variable is to avoid the updating of the maximum or minimum when, respectively, the maximum or minimum (going up or going down) requested floor are reached (BUTTON=FLOOR).

The next part is about the selection of starting direction algorithm, executed when the variable MOTION is 2, i.e. when the cabin has not started to move yet, the stop button has not been pushed and the opening/closing process is happening. We based on the SJF (Shortest Job First) algorithm which selects, as first call to be serviced, the nearest requested floor. It works as follows: we calculate the difference between the maximum floor request and minimum floor requested, that is the difference between MAX_BUT and MIN_BUT, and current floor.
If the lower floor is closer ((MAX_BUT-FLOOR)>(FLOOR-MIN_BUT)), we assign MIN_BUT to the BUTTON variable, otherwise, i.e. the upper floor is closer ((MAX_BUT-FLOOR)>(FLOOR-MIN_BUT)), we assign MAX_BUT to the BUTTON variable; when the distance from the current floor is the same, the first floor requested to be serviced is the first button pushed (FIFO). This happens by assigning BUT_1 value to the BUTTON variable. Notice that the first button is memorized into BUT_1 only once, by checking the FIRST_BUT variable. Let's provide an example: assume that the cabin is at the 1st floor, two users push the external buttons of the same floor to open the doors; once entered, one pushes 3, the other pushes 0. The request which is serviced immediately is the second one, regardless of the order with which the two buttons are pushed. In case the users pushed first 2 and after 0, the cabin will go to the 2nd floor first.


[not(FLAG_STOP) && MOTION==2 && FLAG==3]
[(MAX_BUT-FLOOR)>(FLOOR-MIN_BUT)]
[(MAX_BUT-FLOOR)<(FLOOR-MIN_BUT)]
Min_but
en:
BUTTON=MIN_BUT;
Max_but
en:
BUTTON=MAX_BUT;
First_but
en:
BUTTON=BUT_1;
1
2
Figure 20: Starting direction selection

Now we focus on the updating maximum (MAX_BUT) and minimum (MIN_BUT) variables when we are NOT in an idle condition (MOTION≠2). There are two different cases which should be considered: the situation in which we need to update the maximum or minimum with calls, related to the current direction and already memorized but not satisfied yet, and the case concerning the updating with the last button pushed. Let's analyze in details the two conditions.

1. First of all, assume that all requests in ONE direction are satisfied i.e. BUTTON=FLOOR. In this case a change of direction is forced by assigning the value 1 or 0 (down or up) to the MOTION variable depending on the current direction (0 or 1 respectively). The UPDATE variable is set to TRUE. Moreover, if there are any requests out of range in the current direction the MAX_BUT or the MIN_BUT are updated to the proper value, that is the floor corresponding to the calls which have not been serviced yet.

For instance, let's consider that the calls to be serviced are 3 and 0 and the cabin is going up. If the button 2↑ (BUTTON_UP[2]) is pushed, when the cabin is between the 2nd and the 3rd floor, at the end of the closing doors at the 3rd floor the MAX_BUT is set to 2. In this way the sequence with which the calls are serviced are 3-0-2.
Otherwise, that is there is no outstanding request, we simply reset the maximum or minimum respectively to 0 and 3.
All assignments just described can be performed only if the variable EXTEND is TRUE, that is once the door is closed. This allow to keeping going in the same direction, if the MAX_BUT or MIN_BUT is changed during the opening/closing procedure.
For instance, let's consider that the cabin is at the 1st floor in idle condition. One call comes from the 2nd floor to go up (2 ↑). After a while (when the cabin is between the first and second floor) a user at first floor pushes 1 button to go to the GF. Once the first user has entered, he pushes 3. In so doing, the MAX_BUT is "extended" from 2 to 3. The path of the elevator is 1-2-3-1-0.

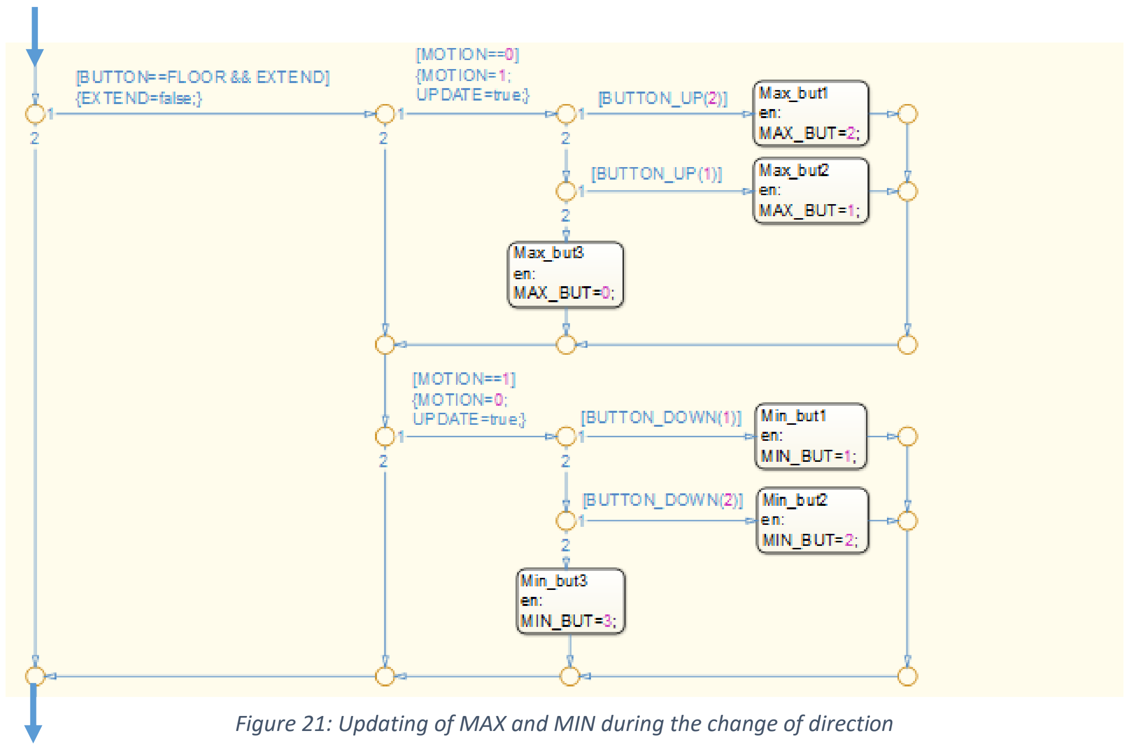


Figure 21: Updating of MAX and MIN during the change of direction

2. The second case is about the situation in which the elevator has not finished to satisfy all request yet (MOTION=0 or MOTION=1). Whenever, at least one button has been pushed (PUSHED=TRUE), if the UPDATE variable is not TRUE, that is the minimum or maximum floor requested are not serviced yet, we update MAX_BUT, MIN_BUT depending on the current value of the BUT_TEMP variable, which stores, as explained before, the last button pressed. This is executed only if the BUT_TEMP is different from the current floor, because in that case the request is considered already serviced. As it is described in the Module 5: Opening/Closing Process of the previous logic, the pressure of the same button corresponding to the actual floor works as an obstacle.

The idea is the following: if the maximum variable is smaller than BUT_TEMP (MAX_BUT<BUT_TEMP), MAX_BUT is updated by the value BUT_TEMP, while if the minimum variable is greater than BUT_TEMP (MIN_BUT>BUT_TEMP), MIN_BUT is updated by the value BUT_TEMP.

The next step after updating the minimum and maximum value, is the assignment to the BUTTON variable. So, if we are moving up i.e. MOTION=0, BUTTON is updated by assigning MAX_BUT to, otherwise, in moving down state, the MIN_BUT is assigned to BUTTON variable. This is implemented by a switch-case structure. Summarizing, the goal is to "extend" the last call to be serviced in the current direction.

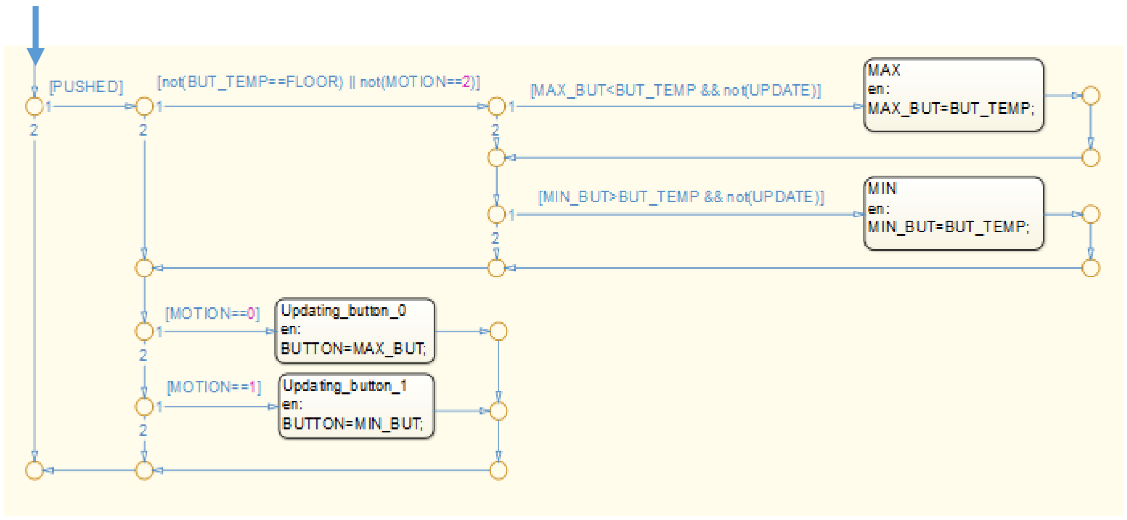


Figure 22: Extension of MAX or MIN

The three Statecharts above (Figure 20, Figure 21, Figure 22) form the **Max_Min_assignment** sub-chart that appears below (Figure 23).

In conclusion, as all request are satisfied or a STOP button is pushed (FLAG=1), it is obvious that we should waiting for a button request and the main FLAG is kept being 1 to call *Button_detection* function block in every cycle. In this case, if any button is pushed which is declared by PUSHED variable, BUT_TEMP variable is assigned to BUTTON and main FLAG becomes 2 in order to execute position control in the next cycle of the program execution.
In addition, there is a check for STOP button and if it has been pushed previously, we keep the cabin LED (LL) ON in order to declare that the cabin is working, but in STOP condition.
Moreover, if the STOP button has been pushed for 3 seconds detected by TIMER_EME, it means that a person is in the emergency situation and may need help (see Module 3: Emergency).
If so, the emergency flag declaration (FLAG_EMG) becomes TRUE and the FLAG is changed to 4. It allows the program to execute the *Emergency* action in the next cycle of program execution.
Otherwise, if the button has been pushed for less than 3 seconds, we reset the TIMER_EME timer to prevent saturation and to be able to use it later. The last condition verified regards the yellow LEDs, which are switched off to declare a reset of all calls.

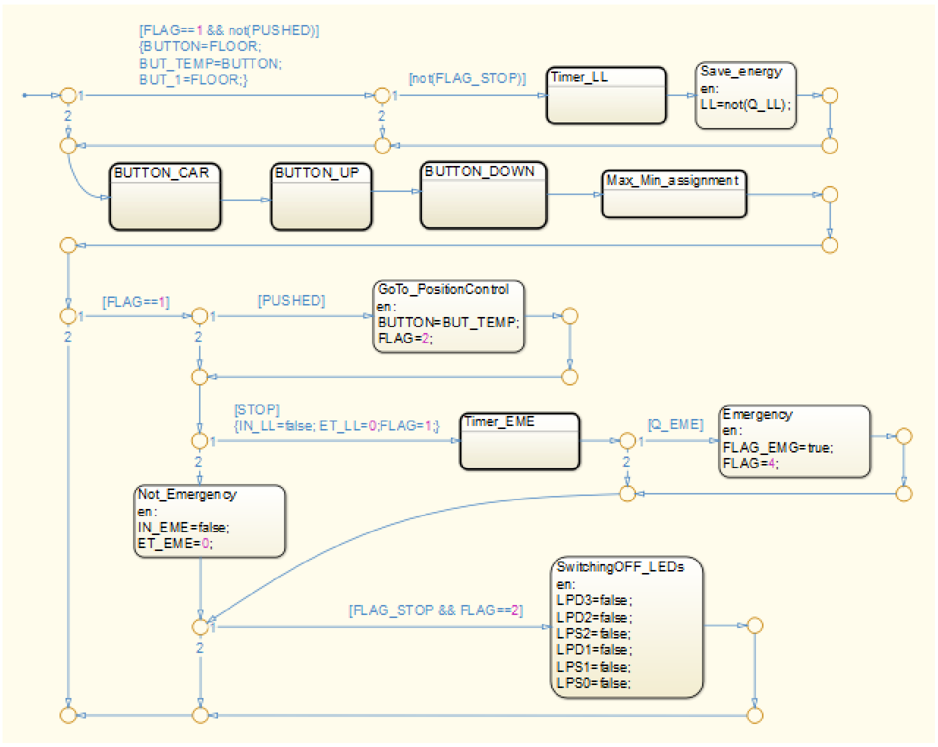


Figure 23: Button_detection Statechart

Module 3: Position control

The first operation executed when this action is called is the *Button_detection*, according to the idea that also when the cabin is moving whatever button is pushed, it has to be memorized. Regarding the car placement, the mechanism remains the same, apart from the addition of some instructions to do the intermediate stop.

We check if the target floor represented by the BUTTON variable is reached or not. Depending on the relation between the BUTTON and the FLOOR and the pressure of the STOP button, the motor is driven in a specific direction by assigning TRUE value to the output variables SALI or SCENDI, or it is stopped setting both of them to FALSE. In movement conditions, i.e. going up and going down, the function *INT_Stop* is called, to stop the cabin at intermediate level as it is mentioned in the Module 7: Intermediate Stop. Moreover, the variable MOTION is set respectively to 0 (up) or 1 (down). If the requested floor is reached or the stop is pushed, the final instructions are executed and the FLAG is set to 3 in normal working condition and 1 in STOP condition. In this last case the FLAG_STOP is initialized to TRUE, all variables associated to a normal working condition are set to the initialization value, button array elements are set to FALSE, yellow LEDs are turned on and the MOTION value is memorized trough the MOTION_TEMP variable and then set to 2 (idle condition). As consequence, in the next iteration, the Button detection (FLAG=1) or the Opening/Closing (FLAG=3) will be performed.

Notice that in the third conditional instruction (**AFTER_STOP**), which has the same functionality of the previous version, the direction of movement before the pressure of stop depends on the MOTION_TEMP value and not on MOTION, which is has been set to 2.

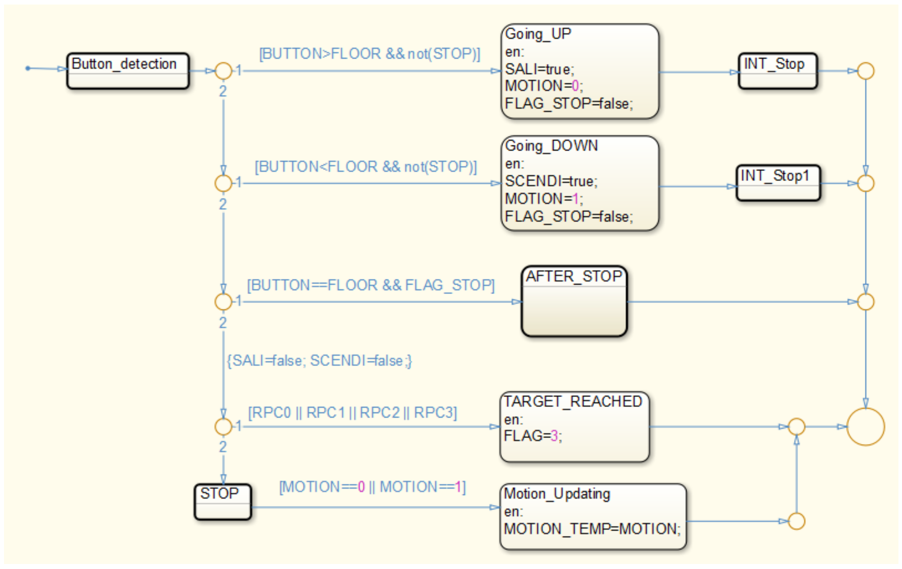


Figure 24: Position_control Statechart

Module 4: Opening/Closing Process

The *OpCl_doors* procedure is almost identical to the previous logic. The only difference is that at the very beginning the *Button_detection* function is called. The reason for which we call this function more times is that we want to be sure that, each time a button is pushed, it is memorised correctly. It is obvious that, in principle, the number of execution cycle of the software is very high and, as consequence, the frequency is higher than the time in which a button remains pushed. However, in this way we guarantee that also in the rare case in which the button is pushed instantaneously, the check is fast enough to catch the input signal.

Module 5: Closing

The mechanism related to the closing of doors, as well as the timer management, have not been changed. What are noteworthy, are the Reset function, the list of assignments that is executed once the doors have been closed and the call of the *Button_detection*.
The *Reset* function (*Req_serviced* type) is used to turn off LEDs and to set to FALSE the button array elements corresponding to the request which is just satisfied, i.e. the floor in which the opening/closing procedure is happening. To do that, it is used a CASE structure, based on the FLOOR variable.
Notice that when FLOOR is equal to 1 or 2 (intermediate levels) both external LEDs and corresponding button array elements are set to FALSE. This is justified by assuming that, regardless of the desired direction, any person enters in the elevator, as the door open. Of course this happens when both external buttons are pushed. Otherwise, i.e. only one of the external button is pushed, the reset is performed only if the cabin is stopped in that floor, namely if the "button direction" is equal to the current one, so the opening/closing process is executed.
In order to obtain an updating of the maximum or the minimum floor requested, also during the opening and the closing process, the variable EXTEND is used, as it is anticipated in the Module 2: Button detection (*Button_detection*). It is set to TRUE when the maximum or the minimum memorized is reached at the end of the closing process.

Subsequently, if all requests are satisfied, i.e. all elements of buttons array are FALSE, the MOTION, MAX_BUT and MIN_BUT are set to the initialization values respectively 2, 0 and 3, the value FALSE is assigned to the PUSHED variable and FLAG is set to 1. Moreover, the FIRST_BUT is set to TRUE, to be ready to memorize the first button pushed and execute in the correct way the FIFO algorithm, described in the Module 2: Button detection. Finally, the UPDATE variable is reset. Otherwise, i.e. there are still calls to satisfy, FLAG is set to 2. In the next iteration depending on the FLAG value the elevator will be stop, waiting for a call or it will move on with the positon control to service the outstanding request.

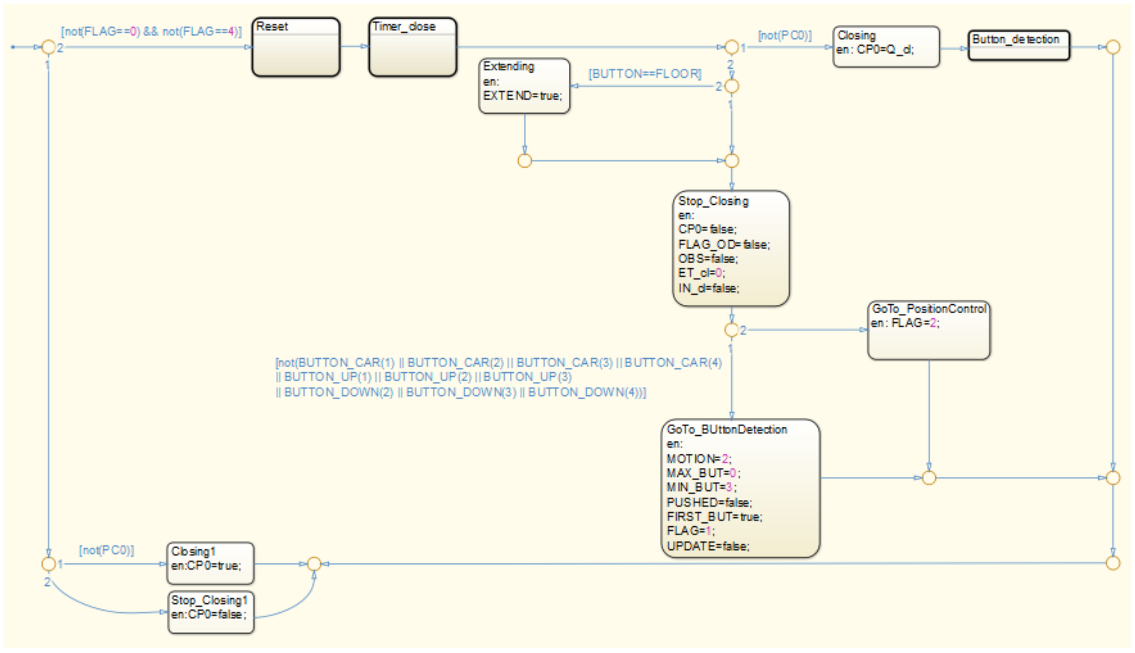


Figure 25: Closing_0 Statechart

Module 6: Opening

The opening function has not been modified. The unique adding instructions are the Reset and the *Button_detection*, which are executed only in normal working condition, that is when FLAG is different from 4 (Emergency situation).

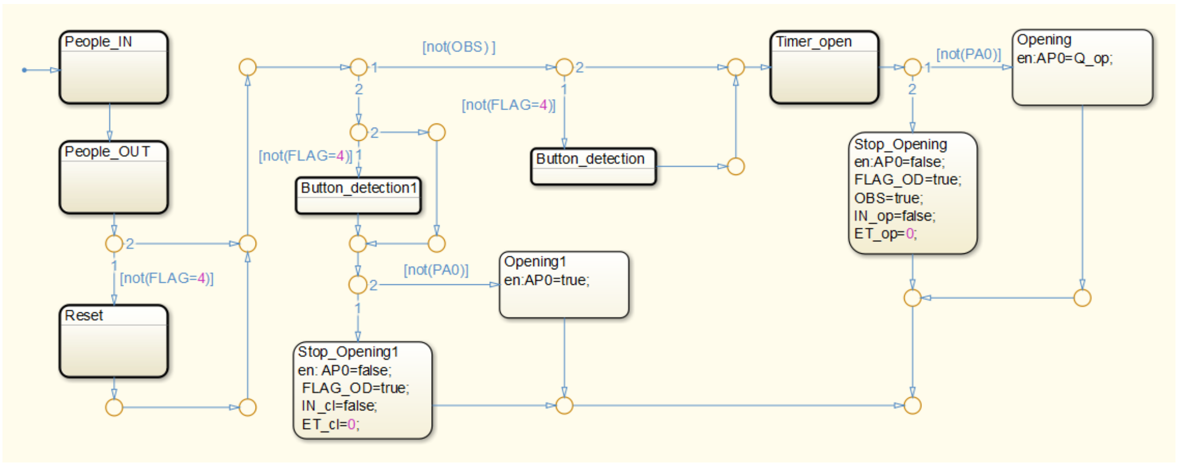


Figure 26: Opening_0 Statechart

Module 7: Intermediate Stop

Intermediate stop is designed for managing floors requests along the car's path “planned” from previous calls. To stop the cabin at an intermediate floor (in this case only 1 or 2), calls from outside must be done in the car's moving direction. For instance, if car is moving to floor 3 from 1 or 0, pushing the button 2↑, the car will stop at the second floor (so request is satisfied), while pushing button 2↓ it will pass it. To this end, if an intermediate floor is reached (RPC1 or RPC2 TRUE) cabin is stopped only if that floor has been called from the car (BUTTON_CAR[…] TRUE) or from outside accordingly to the MOTION value (MOTION=1 check on BUTTON_UP[…] , MOTION=0 check on BUTTON_DOWN[…]).

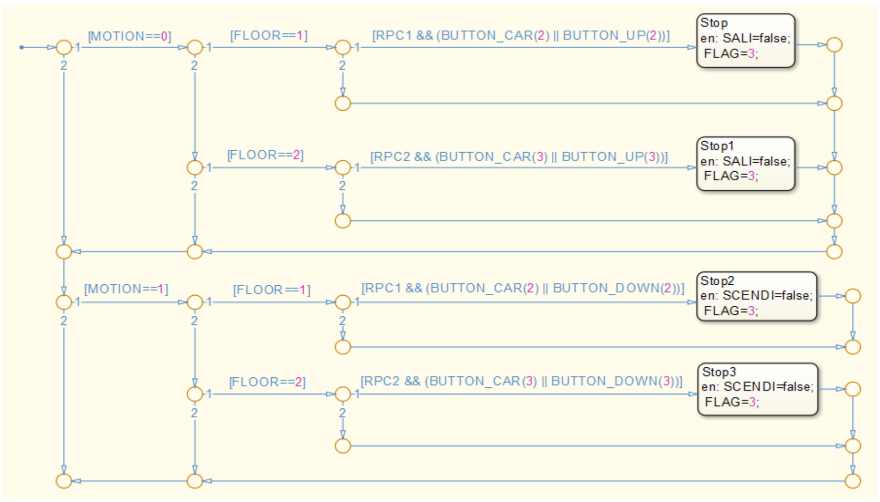


Figure 27: Intermediate_stop Statechart

Notice that all PLC code related to this logic is furnished **HERE**.

4. Logic with Storage B

4.1 Requirements

The main idea of this second version of the logic with storage is to take into account a priority of the calls inside the cabin with respect to external ones, in order to guarantee the smallest travelling time. Thus, as soon as a user enters inside the cabin, he can decide the moving direction.
All the other specifications remain the same.

4.2 Software design: Modelling and code description

From the design point of view, the main differences are in the button detection process.

Button detection (*Button_detection*)

We have introduced the variables BUT_1_CAR, FIRST_BUT_CAR in order to detect the first button pushed inside the cabin. In particular, if there are no pushed internal buttons, the former is set to FLOOR while the second one to FALSE. The main modification is related to the button assignment depending on the current direction (MOTION) and the START variable, which permits to perform the priority only once (for the first user). It can distinguish two cases:

- MOTION=0: while the cabin is going up if a button inside the cabin is pushed(BUT_1_CAR), we check the relation between the BUT_1, which has allowed to bring the cabin at the current floor, and the BUT_1_CAR. In case BUT_1_CAR<BUT_1 we force the change of direction (MOTION=1) even if there are calls coming from upper floors.
- MOTION=1: the same logic is applied, but in the opposite direction.

Let's consider an example. The cabin is at the GF and a user calls the elevator from the 1st floor to go down (1↓). Once the cabin has reached the 1st floor, the user enters and pushes 0. At the same time another call comes from the 3rd floor. According to our logic the cabin moves down to service the request inside the cabin first, although not all request in the up direction have been serviced.
The path is the following: 0-1-0-3.
It is worth to underline that in case the change of direction is performed with this priority algorithm and there are no further requests in the upper/downer floor (MOTION=0 or MOTION=1) the reset of the maximum or minimum has to be executed, because the BUTTON variable changes immediately, so the instructions of 16 of the previous logic are not processed.
A further modify is applied to the starting direction algorithm associated to the idle condition (MOTION=2, Module 2: Button detection (*Button_detection*)). Instead of forcing as first floor to be service always the nearest floor, it is given priority to the internal buttons. In case only internal buttons are pushed the first button pushed is satisfied.
Notice that the reset or setting to initialization value of the new variables is performed also at the very end of the *Emergency* and *Closing* function blocks.

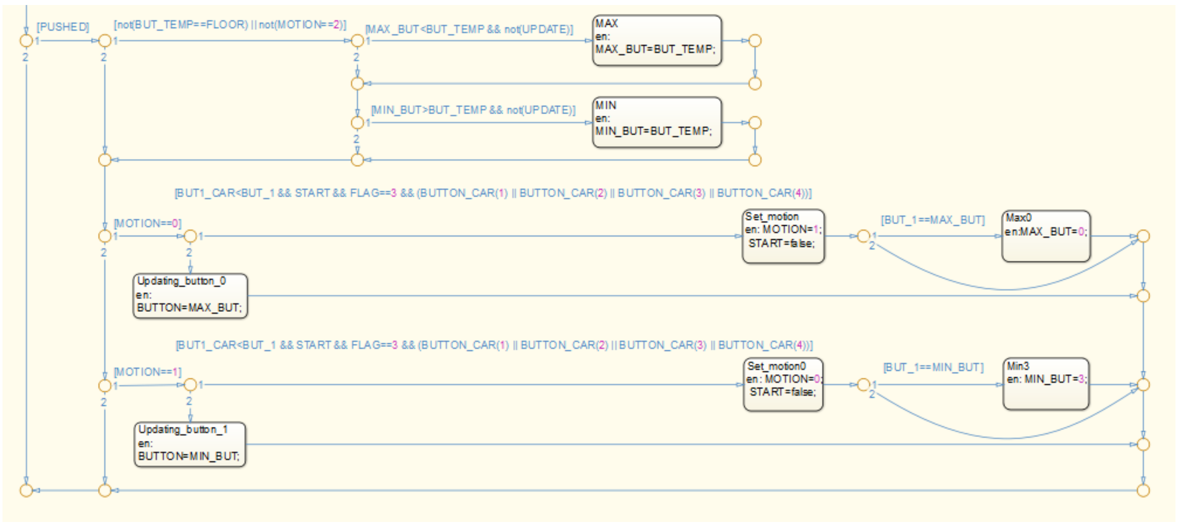

[PUSHED]
[not(BUT_TEMP==FLOOR) || not(MOTION==2)]
[MAX_BUT<BUT_TEMP && not(UPDATE)]
MAX
en:
MAX_BUT=BUT_TEMP;
[MIN_BUT>BUT_TEMP && not(UPDATE)]
MIN
en:
MIN_BUT=BUT_TEMP;
[BUT1_CAR<BUT_1 && START && FLAG==3 && (BUTTON_CAR(1) || BUTTON_CAR(2) || BUTTON_CAR(3) || BUTTON_CAR(4))]
[MOTION==0]
Set_motion
en: MOTION=1;
START=false;
[BUT_1==MAX_BUT]
Max0
en:MAX_BUT=0;
Updating_button_0
en:
BUTTON=MAX_BUT;
[BUT1_CAR<BUT_1 && START && FLAG==3 && (BUTTON_CAR(1) || BUTTON_CAR(2) || BUTTON_CAR(3) || BUTTON_CAR(4))]
[MOTION==1]
Set_motion0
en: MOTION=0;
START=false;
[BUT_1==MIN_BUT]
Min3
en: MIN_BUT=3;
Updating_button_1
en:
BUTTON=MIN_BUT;

Figure 28: Button detection (with cabin priority)

Notice that all PLC code related to this logic is furnished **HERE**.

5. Additional features

5.1 People loading and unloading requirements

The management of the people loading/unloading allow us to forbid the movement of the cabin when the weight is higher than a threshold, set to 320 kg. To this end, the door is kept opened and an alert message exhorts people inside the cabin to exit till the weight becomes lower than the maximum one allowed. A control graphical panel has been developed to simulate the people entering/exiting (see **4** Figure 33: Visualization Basic logic and Figure 34: Visualization logic with storage).

Module 1: Weitgh_IN/Weight_OUT

These function blocks are used to simulate the people entering/exiting. Let's focus on the entering, that is the *Weight_IN* function. The basic idea is to increase the WEIGTH variable and the number of people (KIDS, WOMEN, MEN, FATGUYS) depending on the pressure of a specific button in the visualization (KID_IN, WOMAN_IN, MAN_IN, FATGUY_IN). In order to obtain a correct loading (one person at a time), we introduce the COUNTER_IN variable which allows to increase the WEIGTH variable and the counter corresponding to the actual button pushed, only once, regardless of the pushing time. This permits to avoid also the simulation problem, consisting of obtaining a random increasing of the mentioned variables, depending on the pushing time. The same mechanism is associated to the *Weight_OUT*, but it works in the opposite way: when a "people button" is pushed, the WEIGHT and the corresponding counter are decreased. Moreover, it is also avoided the decreasing of the counter (as consequence also of the WEIGHT) when the counter itself is 0. The figure below shows the *Weight_IN* Statechart. The *Weight_OUT* is not presented for sake of simplicity.

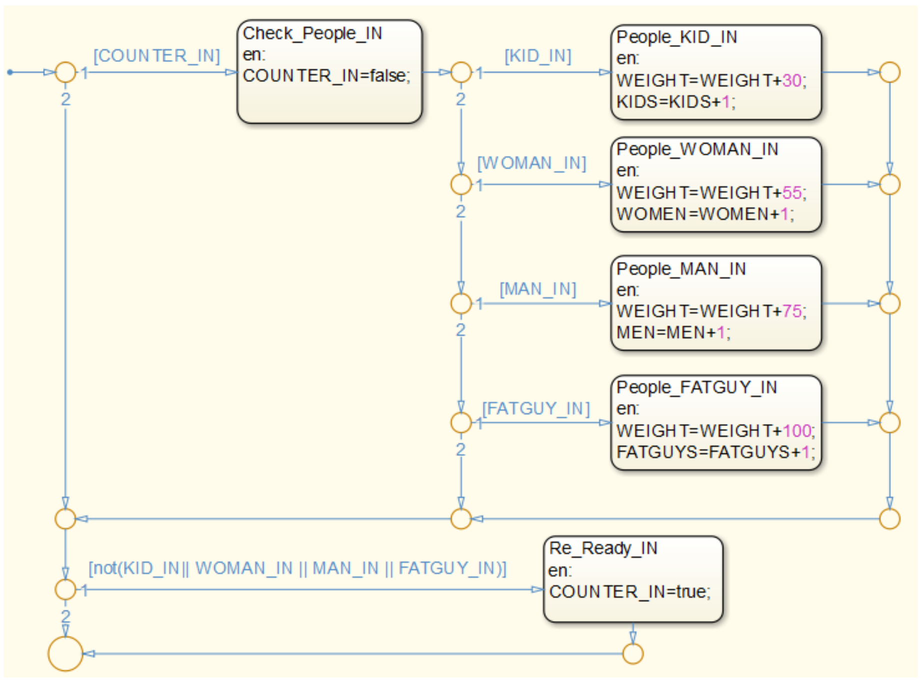


Figure 29: Weight IN Statechart

5.2 Exceptional conditions requirements

In this section we present some comments about the automatic behaviour of the plant in exceptional situations, i.e. the presence of fire in one or more floors, the blackout and an emergency condition, in which one or more people, which are inside the cabin, needs help. Also in this case we used the graphical control panel, which allows to simulate these special conditions and the related solutions. Regardless of the type of exceptional situation, all memorised calls have to be deleted.
First of all, during a blackout, the elevator is stopped and all LEDs are turned off. We can assume that the elevator is linked to an emergency generator which allows the cabin to be forced to go down to the nearest floor and opening the door. Door remains opened and the LL LED keeps blinking to declare the ALERT STATE and prevents people to use the elevator, until the power supply comes back.
Regarding the presence of the Fire, the behaviour of the plant is the same, except from the doors closing which is activated when the STOP is pushed for 3 secs. Notice that only a variable has been selected to declare the presence of the FIRE. An alternative choice could be to consider a variable for each floor and bring the cabin to the nearest floor which is not affected to the FIRE. However, this solution has not been chosen because, the best idea is to bring people out from the elevator as soon as possible to decrease the possibility of elevator wire breaking during the replacement.
Finally, the Emergency condition, due to the presence of a person in the cabin which needs medical help is managed manually (let us suppose that there is an “Instruction message” inside the cabin). Also in this case, as a user inside the cabin keeps pushing the STOP button for 3 secs, the cabin is forced to go down to the nearest floor. The recovery to normal working condition is managed as it was in the Fire case.
In the last two cases (fire and emergency) all yellow LEDs blink to declare the bad condition.

Module 1: Blackout

This function block is used to simulate a blackout condition and at the same time the automatic reaction of the plant. At first a check is actuated on the following variables: BO, FLAG_BO and FLAG_RECOVERY. Let’s define the meaning of them.

- BO becomes TRUE as a blackout condition occurs (the visualization button is pushed)
- FLAG_BO, which is initialized to FALSE (in normal working condition), is also linked to the blackout condition but not directly connected to the visualization button
- FLAG_RECOVERY is TRUE when the recovery procedure related to the blackout is happening.

If the values of these variables are respectively TRUE, FALSE and FALSE, all output signals (LEDs, cabin and doors motors) are set to FALSE, the timer of the LL LED is reset, the FLAG_BO is set to TRUE and the main FLAG to 5. This last assignment allows to remains in an idle condition until the recovery is activated with the corresponding visualization button.

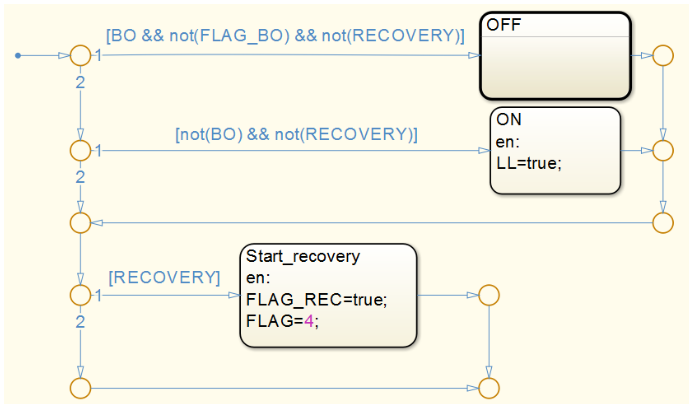


Figure 30: Blackout Statechart

Comparing the versions of the two logics, the only difference is that in the logic with storage there is also a reset of all buttons memorised until that moment, corresponding to request which have not been satisfied yet.
Otherwise, in normal mode (RECOVERY=FALSE and BO=FALSE) the LL LED, which indicates that the power supply is ON, is set to TRUE.
The final check allows to set the FLAG to 4 and the FLAG_RECOVERY to TRUE, as soon the RECOVERY button is pushed. This allows to execute in the next cycle the *Emergency* function block.

Module 2: Fire detection

The purpose of this function is to simulate the behaviour of the plant, in presence of a fire. It consists of a list of statements executed as the visualization button FIRE is pushed. Firstly, FLAG is set to 4, so in the next iteration the *Emergency* function is called. After that the value TRUE is assigned to the FLAG_EMG variable and the cabin is stopped if it were going down (SALI =FALSE). Notice that the assignment SCENDI=FALSE is not executed because the requirement states that the cabin has to force to go down to the nearest floor, so if the cabin was going down, it continues to do it until a floor is reached. Also in this case the reset of buttons memorised is executed only in the logic with storage.

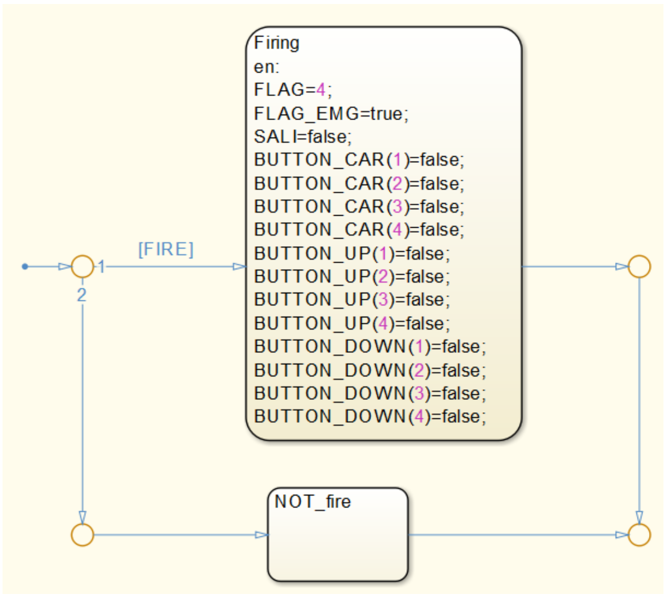


Figure 31: Fire detection Statechart (Storage)

Module 3: Emergency

First of all, it is worthy to define the conditions in which the Emergency state occurs, namely:

- Blackout, i.e. we are out of power, simulated by the BO variable
- one pushes stop button for 3 seconds in order to declare an emergency situation
- Fire presence, simulated by the FIRE variable

If any of this condition holds the main FLAG changes to 4 which means, in the next iteration, the *Emergency* function block is called. Let's explain how the function works.
At very first we check if the FLAG_EMG=TRUE, namely we are in presence of fire or emergency condition; in this case a blinking of all yellow LED is activated to declare the warning state.
Subsequently, if there is the blackout and the Recovery operation has started the LL led on the plant starts blinking to declare that the recovery state is happening. Notice that in the visualization, instead of the blinking, during the recovery a fictitious emergency light is turned on (LL_EME). Moreover, it is noteworthy that in a real plant the Recovery should be activated automatically. In this project we decide to start the recovery operation, during a blackout, manually, through the related button in the GUI only to make the presentation more understandable.
After that, all red lights are switched off to declare that all previous memorized CAR buttons are reset.

As it is described in the requirements, it is verified if the cabin is at any floor by checking RPC sensors, which declare the presence of the cabin in a specific floor and in the case that none of them is TRUE, the cabin starts going down until it reaches a floor. The going down states is executed by assigning TRUE to SCENDI variable.

In the following part, we check:

- The presence of any obstacle (SWITCH)
- Emergency flag declaration (FLAG_EMG=TRUE), which happens in emergency situation caused by pushing stop button for 3 seconds or in presence of fire
- Blackout flag declaration (FLAG_BO)

All these conditions are reasons for keeping the door open in a floor. If any of those condition has been satisfied, with a CASE structure we open the door corresponding to the floor that is declared by the *Floor_detection* function block. As expected, we also put the closing motor command of the corresponding floor's door FALSE to make sure the door is kept opened. It is straightforward that the opening, as consequence the closing, is performed only in the case in which the cabin is at a floor.
In the opposite case, that is there is no more an exceptional situation, the door closing process is activated, using as before a CASE structure. It is taken into account also the rare situation (which is of course likely to happen) in which emergency function is called because of the emergency situation by pushing stop button for more than 3 seconds and in the meanwhile blackout also happens.
So, to return to normal working condition we check if:

- the STOP is pushed for at least 3 seconds (TIMER_EME_OFF) when the door is completely open (checking PAx) and the emergency/fire condition is gone (FIRE=FALSE)
- the blackout is over when the door is completely open

It is worthy to make the FLAG_BO FALSE after the blackout state is finished (BO=FALSE) and at least one of the doors is open. This permits to complete the recovery procedure (especially the opening process), regardless the BO becomes FALSE (there is not Blackout anymore).
Finally, as the cabin is in one of the floors and all doors are closed, a list of assignments is executed. By this, the integer 1 is assigned to the main FLAG, which means in the next cycle of program execution the emergency procedure is skipped and *Button detection* function is called. Moreover, all yellow lights are turned off and Boolean value FALSE is assigned to OBS flag and FLAG_OD flag, to have a correct opening/closing procedure in the next normal working condition.
After that, we assign FALSE to the all timers' inputs in order to make them deactivated completely and ready for the next emergency happening. In addition, flag FIRST gets TRUE again and to just emphasis that we are sure that we are not in the recovery state, we assign FALSE to the FLAG_REC. In conclusion the MAX_BUT and MIN_BUT are set to their initialization value.

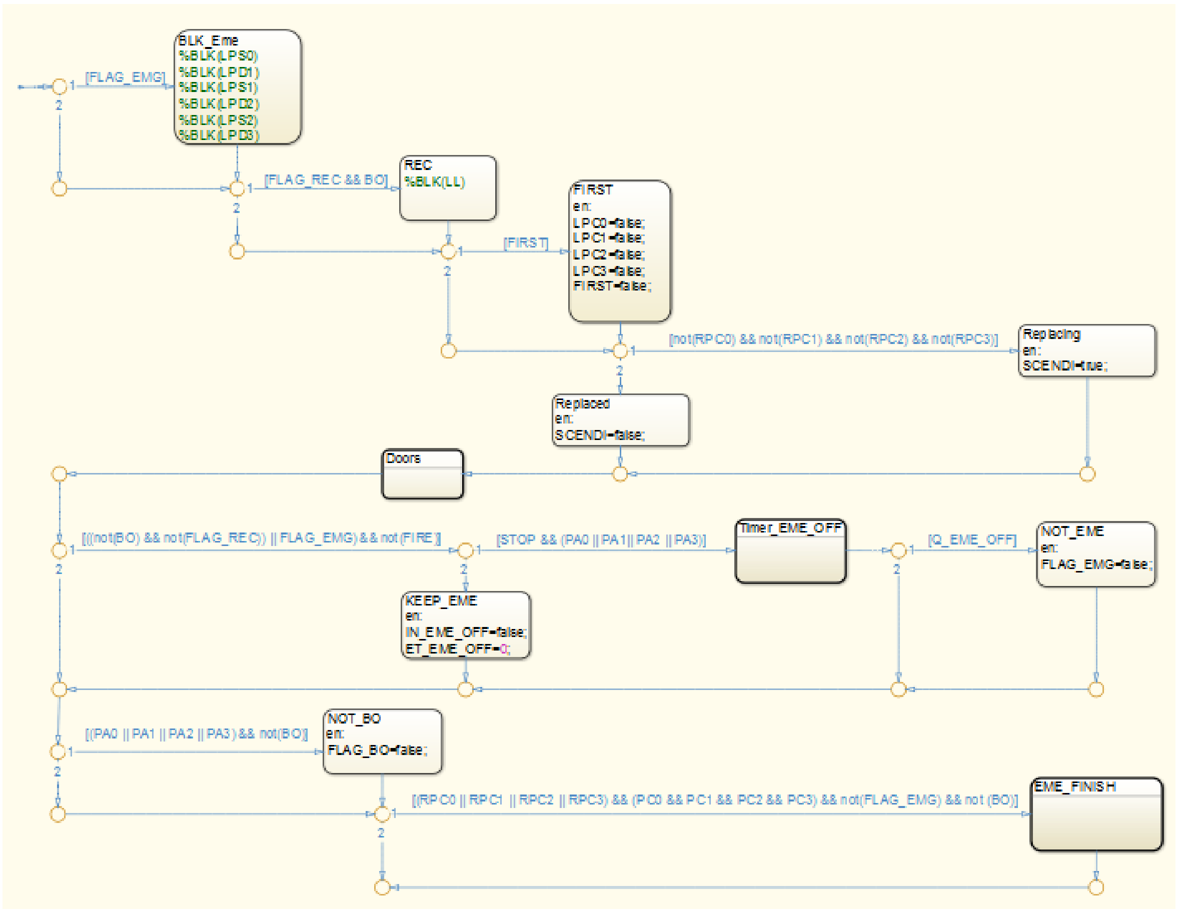


Figure 32: Emergency Statechart

6. Visualization and Testing

6.1 Visualization

In CoDeSys software, there is a feature that provide us the opportunity to create a GUI, which allows to visualize our system with the same exact facilities of the real plant. The main advantage of having visualization is saving time. It may happen that the main plant has some hardware problems like broken buttons or broken motors which procrastinate our implementation and testing procedure. Therefore, by having a visualization, we can implement and test modifications or added elements faster. To this end, we add some boxes (1.a - 1.b) which show the actual values of main variables in order to recognize eventual errors. These variables are BUT_TEMP, BUTTON, MOTION, FLAG in basic logic and MIN_BUT, MAX_BUT, BUT_1, BUT_TEMP, BUTTON, MOTION, FLAG in storage logic. Beside this, some additional features are added, to simulate exceptional situations, which can be shown only in visualization and not in the real plant. For instance, it is impossible to have a real fire and to see how the elevator react to the fire happening, so we simulate it by adding a toggle button which, whenever is pushed, simulates the on-going fire; by just pushing the button again, the normal state is recovered.
The same feature is used to simulate Blackout and Recovery operation (2).
There is also a display which indicates the current floor and a meter which shows the real-time position of the cabin (3).
Moreover, as it is mentioned in the previous section, a control panel is used to simulate the entering/exiting of people, composed by some tap buttons to add and subtract people and a meter which displays the current total weight (4).
Finally, a text box is added to declare the actual state or eventually a warning message (5).

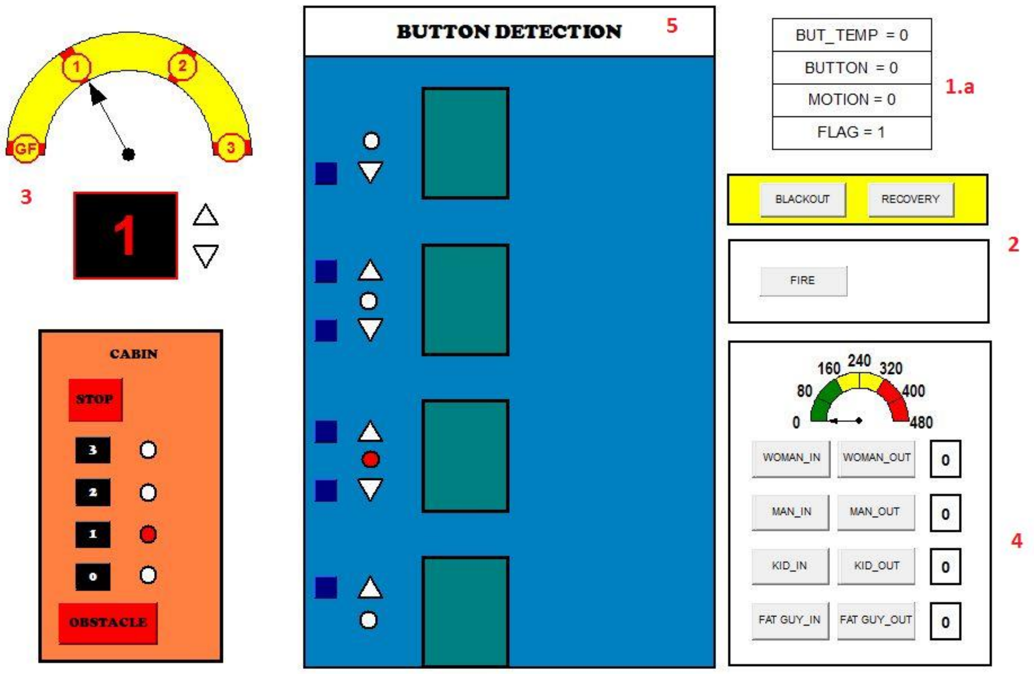


Figure 33: Visualization Basic logic

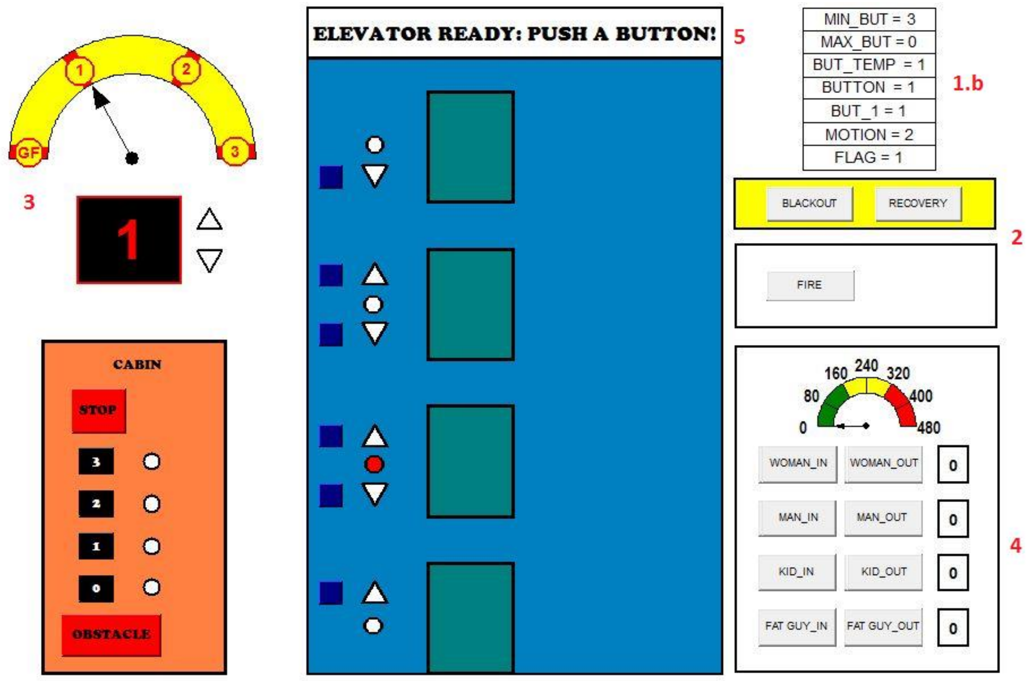


Figure 34: Visualization logic with storage

6.2 Testing

One of the most important part of any project is Testing. Before explaining how we use and implement testing in our project, it is worth mentioning two common types of testing approaches: Black box and White Box Testing. In the following we explain how these methods of testing work and how they are applied in our case.

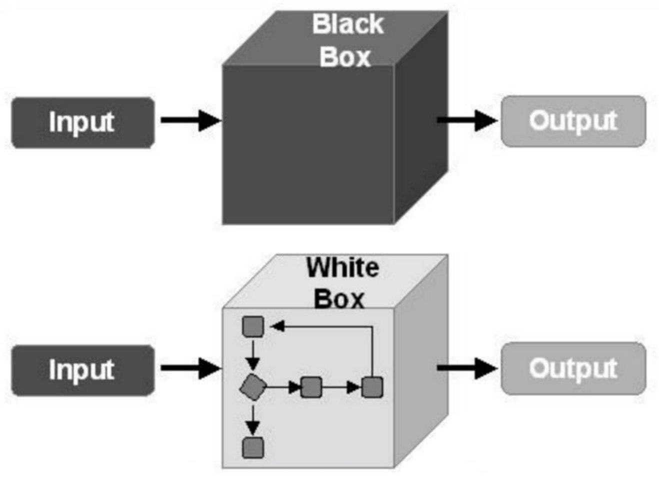


Figure 35: Testing approaches

Black-box testing

Black-box testing (also known as functional testing) treats software under test as a black-box without knowing its internals. Tests use software interfaces and try to ensure that they work as expected. As long as functionality of interfaces remains unchanged, tests should pass even if internals are changed. Tester is aware of what the program should do but does not have the knowledge of how it does it. Black-box testing is most commonly used type of testing in traditional organizations that have testers as a separate department, especially when they are not proficient in coding and have difficulties to understand the code. It provides external perspective of the software under test.

Some of the advantages of black-box testing are:

- Efficient for large segments of code
- Code access is not required
- Separation between user's and developer's perspectives

Some of the disadvantages of black-box testing are:

- Limited coverage since only a fraction of test scenarios is performed
- Inefficient testing due to tester's luck of knowledge about software internals
- Blind coverage since tester has limited knowledge about the application

The way we use black box testing approach is that during the implementation, every time we added a function or an action to our code, we implemented and ran it on the PLC which is connected to our plant. In this case, we start defining different probable happening for the elevator as a regular user. So, we assume the elevator as a black box and we enter inputs (pushing buttons) without caring about how the code works and focusing only on how the elevator should react to those inputs. Of course, as mentioned before, we cannot cover all the possible scenarios, however, we solve this problem by focusing on tricky ones and those tricky states which may cause problems. Then, if during this type of testing, we faced any problem, we record it to further testing option which is white box testing approach. It is worth mentioning that, in order to make this procedure faster, visualization helps a lot in terms of saving time as explained before.

White-box testing

White-box testing (also known as clear box testing, glass box testing, transparent box testing, and structural testing) looks inside the software that is being tested and uses that knowledge as part of the testing process. If, for example, exception is thrown under certain conditions, test might want to reproduce those conditions. White-box testing requires internal knowledge of the system and programming skills. It provides internal perspective of the software under test.

Some of the advantages of white-box testing are:

- Efficient in finding errors (hidden) and problems
- Helps optimizing the code
- Due to required internal knowledge of the software, maximum coverage is obtained

Some of the disadvantages of white-box testing are:

- Might not find unimplemented or missing features
- Requires high level knowledge of internals of the software under test
- Requires code access

It is evident that, accessing to the code, having enough know-how about both the code and the way it works, provide us the opportunity to use white box testing. To this end, we put break point in any part we had a problem during black box testing procedure or those parts that are complicated to trace. In this way, we verify that our code works in any scenario. As mentioned in previous part, visualization helps us to recognize problems faster, especially in tracking easily our vital variables.

7. Analysis of results

This chapter presents a comparison between logics, based on the following performance indicators:

- ✓ Waiting time outside the cabin ($T_{WAITING}$)
- ✓ Travelling time inside the cabin($T_{TRAVELLING}$)
- ✓ Total time spent using the plant(T_{TOTAL})

In order to compute these indicators, we have considered the service time, composed by:

- Moving time, that is the time needed to move the cabin between two near floors.
- Stopping time, that is the time spent during the arrest of the cabin at a floor, i.e. the time spent to open and close the door.

The loading and unloading time is considered null (the user enters or exists instantaneously).
The moving time is computed automatically, through a simulation run in CoDeSys with the different logics, considering the same sequence of calls.
The other service time are set to a fixed value, in particular the opening time is 3,5s and the closing time 6,5 s. These, in turn, take into account the timer and the time need to open and close the door effectively. Notice that re-opening operations due to the presence of an obstacle or the pressure of the stop are neglected.
It is important to underline that the magnitude order of results is not real and depend on the computer which is used to execute the simulation.

7.1 Basic Logic VS Logic with storage

We assumed two different scenarios:

- ✓ Scenario A: Morning in a residential building during working day
- ✓ Scenario B: Lunch time in a commercial building

The starting point for the selection of a reasonable sequence of calls, depending on the Scenario, was the probability matrices definition (Table 2, Table 3).
These matrices model the probability distribution of the destination floor selected by elevator pending passengers at different floors. The probability that a user wants to go to a specific floor is thus different depending on the departure floor and the Scenario considered for the simulation. This allows us to model a series of situations, supposing:

- improbable/impossible the case in which source floor and destination floor are coincident;
- less probable the usage of the elevator for going to adjacent floors (people use also stairs);
- the most probable target floor changes according to the Scenario (for instance the percentage of people which decide to go to the last floor with restaurant in a commercial building at lunch time is high, as well as the number of users that want to reach the ground floor to go out in a residential building in the morning). They are underlined in the tables below.

		TARGET FLOOR			
		GF	1	2	3
STARTING FLOOR	GF	0%	10%	45%	45%
	1	60%	0%	20%	20%
	2	90%	5%	0%	5%
	3	90%	5%	5%	0%

Table 2: Residential building in the morning

		TARGET FLOOR			
		GF	1	2	3
STARTING FLOOR	GF	0%	5%	5%	90%
	1	30%	0%	10%	60%
	2	40%	10%	0%	50%
	3	65%	35%	5%	0%

Table3: Commercial building at lunch

Scenario A

As it is already mentioned, this Scenario regards the residential building.
Let's assume that the cabin is at the ground floor (GF), supposing that the last user left the building; a person calls the elevator from the 3rd floor to reach the GF (3↓). After a while, another user performs a call from the 2nd floor to go to the GF too (2↓).
With the basic logic the second user cannot calls the elevator because it remains in the busy state (all yellow LEDs on) until first request is serviced. Notice that the first user pushes the 0 button during the opening/closing door procedure.
Considering the same list of calls with the algorithm with storage, the second user request can be stored immediately so that he/she can be loaded during the going down and serviced together with the first user.

The sequences of floors in the two logics are respectively:

- 0-3-0-2-0 (BASIC)
- 0-3-2-0 (STORAGE)

Scenario B

This second Scenario is about a commercial building, where the first three floors (GF-1-2) are devoted to offices, while at the highest one (3) there is a restaurant.
Let's assume that the cabin is at the 3rd floor, supposing that the last user has gone to eat. There are three users: the first one calls the elevator from the 3rd floor to reach the GF (3↓, he has finished to have lunch); the second one performs a call from the 2nd floor to go to the GF too (2↓, he wants to eat something outside the building); finally, the last one from GF wants to reach the third floor to have lunch (0↑). The calls are more or less simultaneous but the order is the following: 3-2-0.
As in the previous case (Scenario A), with the basic logic the second and third users cannot call the elevator because it remains in the busy state (all yellow LEDs on) until first request is serviced. Notice that also in this simulation, we assume that all internal call (buttons inside the cabin) are performed during the opening/closing door procedure.
For this reason, as soon as the first request is serviced the third user can enter inside the cabin and pushes the desired button. As consequence the 2nd user is satisfied lastly.

Considering the same list of calls with the algorithm with storage, the second and the third user requests can be stored immediate. In this way the second user is loaded during going down associated to the first user and they reach together the GF. The final operation allows to bring the third user to his/her desired destination.
The sequences of floors in the two logics are respectively:

- 3-0-3-2-0 (BASIC)
- 3-2-0-3 (STORAGE)

SCENARIO A		BASIC	STORAGE
1ST USER	$T_{WAITING}$ [s]	14,606	14,625
	$T_{TRAVELLING}$ [s]	21,950	31,063
	T_{TOTAL} [s]	36,556	45,688
2ND USER	$T_{WAITING}$ [s]	47,826	28,064
	$T_{TRAVELLING}$ [s]	17,361	17,624
	T_{TOTAL} [s]	65,187	45,688
	T_{USAGE} [s]	71,687	52,188

Table 4: Scenario A indicators

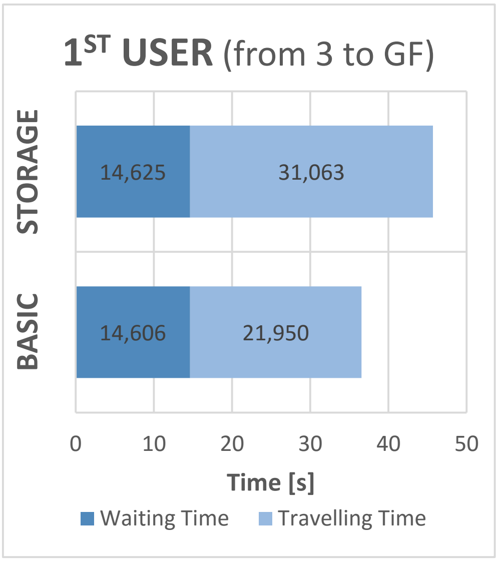


Figure 36

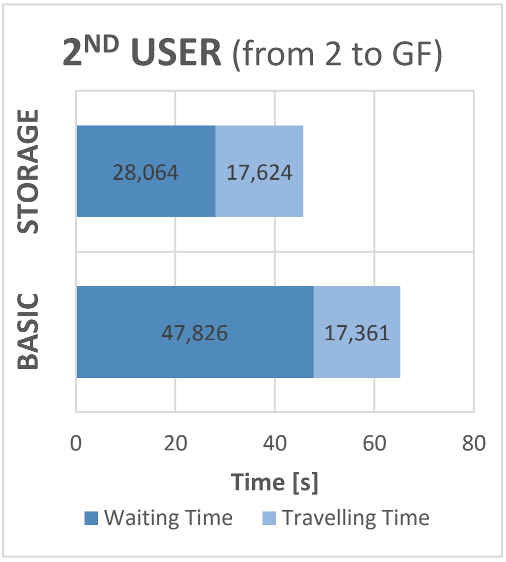


Figure 37

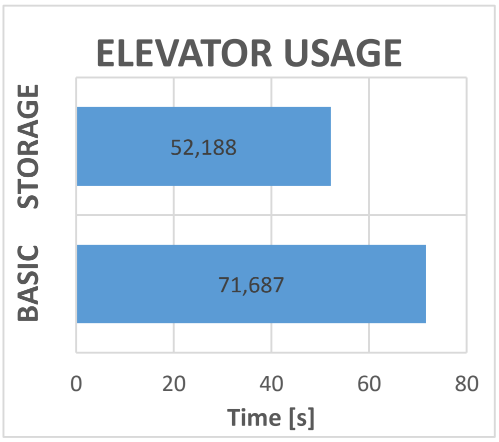


Figure 38

SCENARIO B		BASIC	STORAGE
1ST USER	$T_{WAITING}$ [s]	3,500	3,500
	$T_{TRAVELLING}$ [s]	21,149	31,107
	T_{TOTAL} [s]	24,649	34,607
2ND USER	$T_{WAITING}$ [s]	59,417	16,978
	$T_{TRAVELLING}$ [s]	17,648	17,629
	T_{TOTAL} [s]	77,065	34,607
3RD USER	$T_{WAITING}$ [s]	24,649	34,607
	$T_{TRAVELLING}$ [s]	21,348	21,170
	T_{TOTAL} [s]	45,997	55,777
	T_{USAGE} [s]	83,565	62,277

Table 5: Scenario B indicators

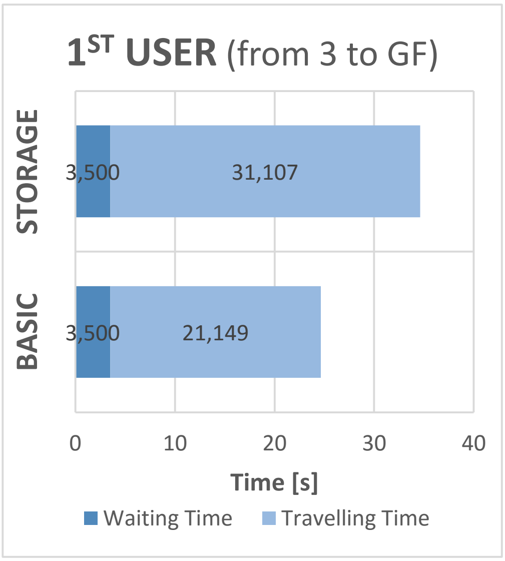


Figure 39

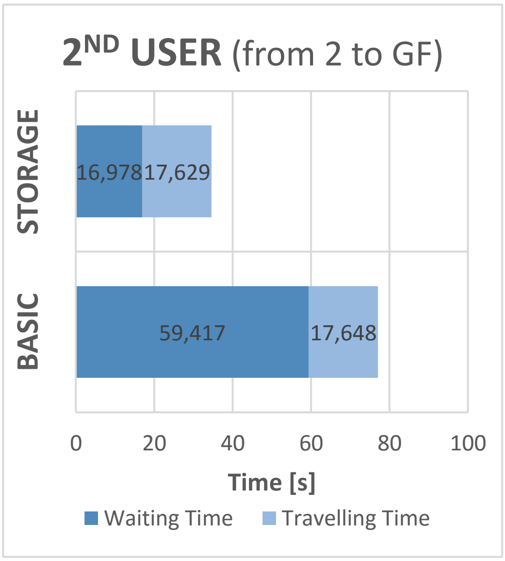


Figure 40

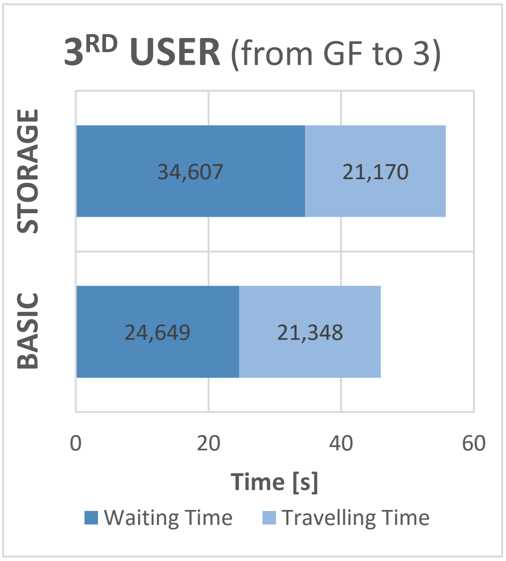


Figure 41

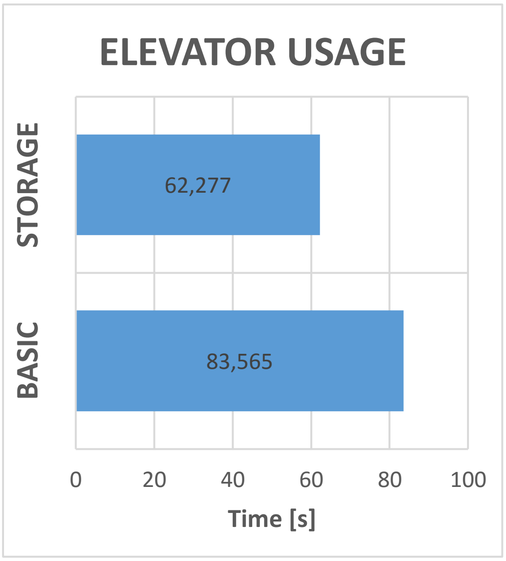


Figure 42

Let's describe the main results. First of all, as it is shown in the *Figure 38* and *Figure 42*, in the logic with storage, the total usage of the elevator, regardless of the Scenario and the user considered, is smaller than the basic one. It means that there is a calls management optimization and, as consequence, an improvement from the energy consumption point of view. In the Scenario A, the decreasing is of 27%, while in the Scenario B it is of 25%.
Looking at the tables above, going from the basic to the storage, it can be noticed that there is an improvement in the waiting time for the users which are at the intermediate floors (user 2 in both Scenarios) and a worsening in the travelling time for the users at the edges (user 1 in Scenario A, users 1 and 3 in Scenario B). This is due to the cabin stopping to service the intermediate requests. Overall the amount of improvement is higher than the worsening one.
For instance, in the second Scenario it can be seen that the waiting time of the second user decreases significantly (71%), while the travelling time of the first one increases in a lighter way (32 %).
This effect rises, considering a building with a higher number of floors and a greater number of calls. Obviously, as already said, this results are related to a simulation; however, considering a real plant although the specific time values change, the relation between the two logics remains the same.

7.2 Logic with storage A VS Logic with storage B

In this section we consider the following scenario: the cabin is in idle condition at the GF; the first user calls the elevator from the 1st floor to go down (1↓). As the cabin reaches the first floor and the user enters, he pushes the button 0. After a while the second user performs a further call from the 2nd floor to go up (2↑).
With the logic A once the cabin has stopped at the first floor, although there is an internal request, the cabin keeps to go up, because of the second request. This is a consequence of the SCAN algorithm (change direction only if all request in the current direction are satisfied).
In the logic B, due to the priority of the cabin the internal request is satisfied first.
The sequence of floors in the two logics are respectively:

- 0-1-2-3-0
- 0-1-0-2-3

		STORAGE_A	STORAGE_B
1ST USER	$T_{WAITING}$ [s]	6,964	6,985
	$T_{TRAVELLING}$ [s]	48,724	13,441
	T_{TOTAL} [s]	55,688	20,426
2ND USER	$T_{WAITING}$ [s]	20,775	37,753
	$T_{TRAVELLING}$ [s]	13,749	13,750
	T_{TOTAL} [s]	34,524	51,503
	T_{USAGE} [s]	62,188	78,429

Table 6: Scenario Indicator

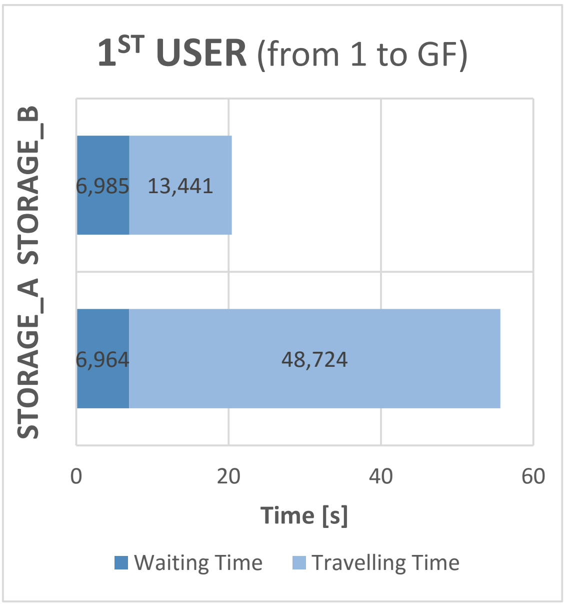


Figure 44

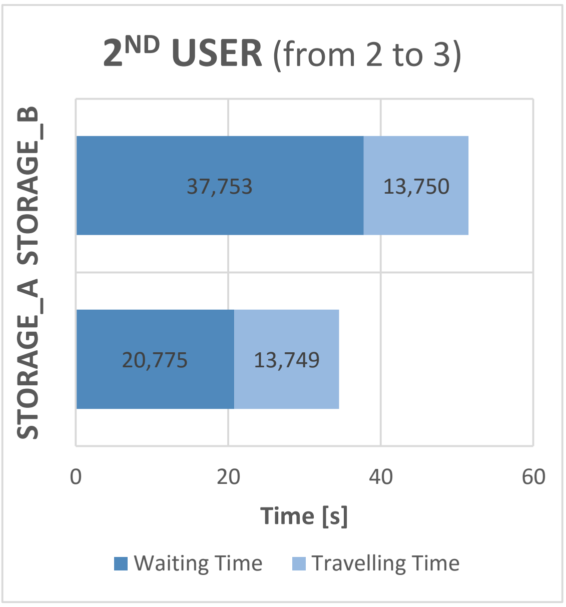


Figure 43

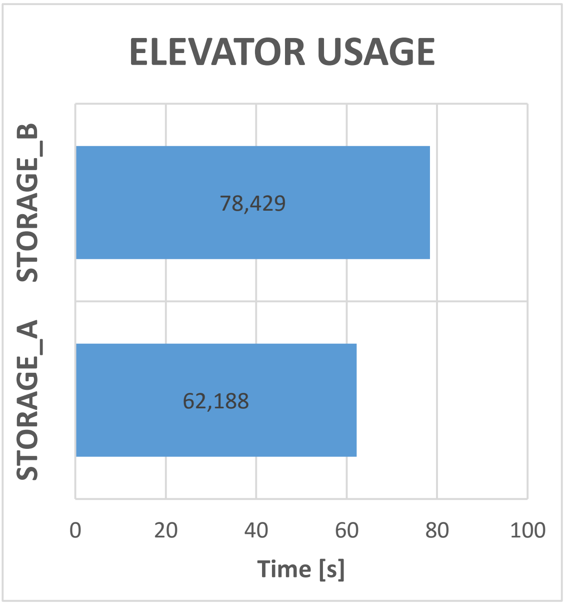


Figure 45

As it can be seen from the graphs, passing from the logic A to the logic B, there is a significant improvement of the travelling time (decreasing) associated to the first user. On the other hand, there is a worsening from the waiting time point of view (increasing) for the second user. Comparing the two logics, we can say that the second one is better: it is more reasonable that when a user enters inside the cabin, he/she takes control of it, reaching the requested floor in the shortest time as possible; while an increasing of the time spent outside the cabin is not so bad, because the user has no knowledge about the actual number of users. Moreover, with the logic A, the probability of cabin fullness is higher, as consequence of the growing time spent inside the cabin.

8. Conclusions and future developments

This conclusive section is devoted to summarize the software development procedure, used to control the elevator plant with a PLC.
A waterfall approach has been followed, starting from developing the basic logic.
In particular, as shown in the figure below, after the modelling phase, each module is implemented with the structured text language and tested. The choice of using this language derives from the previous knowledge of similar programming language, like C. After that, all modules are integrated suitably and tested again. In the testing phase, as it is already mentioned a key point was played by the visualization.
The same procedure has been repeated for the logic with storage, adapting the existing modules and adding new features.
Following this procedure, the effort distribution was:

- 15% Requirements and specifications
- 25% Modelling
- 30% Coding
- 30% Testing

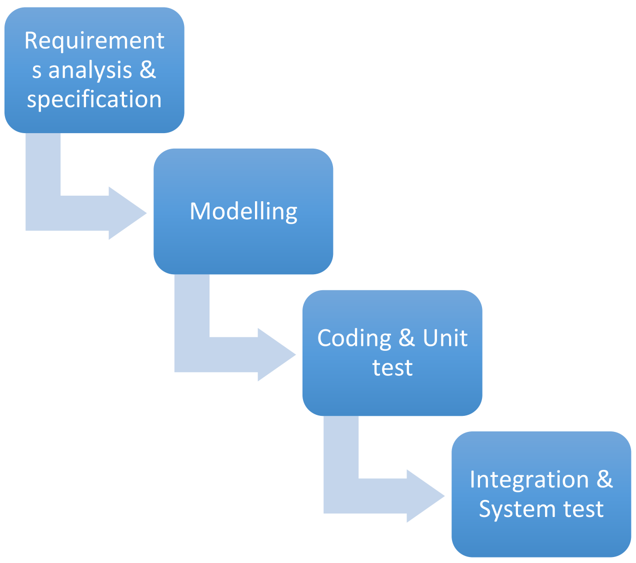
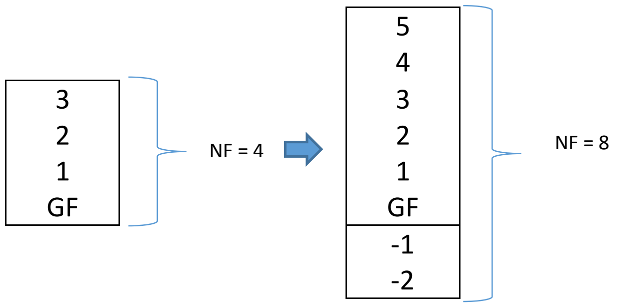

Requirements analysis & specification
Modelling
Coding & Unit test
Integration & System test

Figure 46: Waterfall approach

The main problem in the design, beside of the algorithm definition itself, was to make the code as general and simple as possible. In other words, the goal was to take into account all possible cases, considering specific situations that need to be analysed separately and integrated without causing ambiguities. For instance, the pressure of the button corresponding to the last floor which the elevator is passed through after the stop button is pushed (Basic logic,Module 5: Position control (*Position_control*)) or the maximum/minimum updating when a change of direction is needed considering outstanding request (Logic with storage, Module 2: Button detection (*Button_detection*)). As consequence, focusing on the resulting algorithm in the logic with storage, we can say that it can be used also with a higher floors number building, making trivial modifications:

- The initialization value of the MIN_BUT variable has to set to the new number of floors subtracted by 1 (NF - 1) and the MAX_BUT one remains 0. Notice that in presence of underground floors, the 0 floor is the lowest. An alternative can be using the negative values corresponding to the underground floors. In this case the MAX_BUT has to be initialized to the lowest underground floor (in the figure below -2) and the MIN_BUT to the number of floors above ground, included the GF, subtracted by 1 (in the figure below 5).



- The buttons array has to be extended, taking into account the new number of total floors. Notice that using the second approach described in the previous point, there will not be a perfect correspondence between the index of each element and the number of floor. For instance, the index of the first element of the BUTTON_CAR array will be 0 although it is referred to the -2 floor.
- The checks in the button detection has to be extended, considering all the new buttons.
- The checks in the floor detection has to be extended, considering all floors.

It is important to underline that increasing the number of floors and the potential amount of users, more than one elevator should be used. In so doing, an algorithm which manages the link between the different elevator has to be added.

In conclusion making a comparison between developed logics, we can say that there are pros and cons for each of them.
Focusing on basic logic, the advantages are that the logic provides the cabin empty each time it satisfies a request and ensures the user reaches the target floor with the shortest travelling time (time spent inside the cabin). The disadvantage is that it may happen that a user waits for the cabin for a long time. Furthermore, as soon the plant is free the user has to push the button as fast as possible, before other potential user.
Talking about the logic with storage the pro is that overall there is an optimization of the total usage of elevator and also a reduction of the waiting time, especially for users at intermediate floors, in the majority of cases. Making a comparison between the logic A and B we can say that:

- In the logic A there is no possibility to not serve a floor request, maintaining the cabin between a range of floors (it can happen in the logic B); all floors, even in a long time are reached, because the change of direction is linked **only** to the max and min concept. Without any priority between internal and external buttons the cabin keeps to move in the same direction until all requests are satisfied in that direction. This can be considered suitable especially in industrial applications or in commercial buildings.

- In the logic B, it is sure that the user which enters inside the cabin is served first, although there are outstanding requests in the current direction. Notice that the priority works only with the first button pushed inside the cabin. This choice can be justified considering that further calls can be intermediate stop or “extended target” if they are in the same direction; otherwise, they will service as soon the change of direction is performed.
  It is straightforward to think that in presence of more users inside the cabin which want to go in opposite directions, some of them could be not satisfied immediately. This is less probable in a residential building because it is difficult to have many internal calls in different directions at the same time, so this logic can be considered adapt.

On the other hand, the chronological order with which buttons are pushed is not respected, in other words it is not obvious that the first user which calls the elevator is serviced first. Moreover, when the cabin is full of people, although it stops to service further requests, it cannot serve those users and it just increases the travelling time without any advantages.
For these reasons nowadays, both logics (basic or with storage) are used depending on the type of building. Generally, the basic algorithm is applied in residential building with a small number of floors, while the storage one in public buildings (hospital, universities, commercial) with more than one elevator.

Appendix A

PLC I/O configuration

Variable	Type	Address *
AP0	BOOL	%QX22.4
AP1	BOOL	%QX22.1
AP2	BOOL	%QX1.5
AP3	BOOL	%QX1.1
CP0	BOOL	%QX22.3
CP1	BOOL	%QX22.0
CP2	BOOL	%QX1.4
CP3	BOOL	%QX1.0
LL	BOOL	%QX0.2
LPC0	BOOL	%QX0.7
LPC1	BOOL	%QX0.5
LPC2	BOOL	%QX0.4
LPC3	BOOL	%QX0.3
LPD1	BOOL	%QX1.6
LPD2	BOOL	%QX1.3
LPD3	BOOL	%QX0.6
LPS0	BOOL	%QX22.2
LPS1	BOOL	%QX1.7
LPS2	BOOL	%QX1.2
P0	BOOL	%IX0.4
P2	BOOL	%IX0.2
P2	BOOL	%IX0.3
P3	BOOL	%IX0.1
PA0	BOOL	%IX14.6
PA1	BOOL	%IX14.2
PA2	BOOL	%IX1.5
PA3	BOOL	%IX1.0
PC0	BOOL	%IX14.5
PC1	BOOL	%IX14.1
PC2	BOOL	%IX1.4
PC3	BOOL	%IX0.7
PD1	BOOL	%IX14.0
PD2	BOOL	%IX1.3
PD3	BOOL	%IX0.6
PS0	BOOL	%IX14.4
PS1	BOOL	%IX1.7
PS2	BOOL	%IX1.2
RPC0	BOOL	%IX14.7
RPC1	BOOL	%IX14.3
RPC2	BOOL	%IX1.6
RPC3	BOOL	%IX1.1
SALI	BOOL	%QX0.0
SCENDI	BOOL	%QX0.1
STOP	BOOL	%IX0.0
SWITCH	BOOL	%IX0.5

*IX →INPUT , QX→ OUTPUT

Appendix B

Global variables - Basic Logic

Variable	Type	Init value	Description
FLAG	INT	0	*Current working condition*
BUTTON	INT	-	*Current requested floor*
MOTION	INT	-	*Cabin's moving direction*
FLOOR	INT	5	*Last recognized floor*
FLAG_BT	BOOL	FALSE	*Car button pushed during OpCl_doors declaration*
BUT_TEMP	INT	-	*Car button pushed during OpCl_doors*
OBS	BOOL	FALSE	*Fast re-opening indicator*
FLAG_OD	BOOL	FALSE	*Open door declaration*
FLAG_STOP	BOOL	FALSE	*Stop condition declaration*
TIMER_closed	TON	-	*Closing activation timer*
TIMER_opened	TON	-	*Opening activation timer*
TIMER_LL	TON	-	*Timer object for LL led*
BLK	BLINK	-	*Blinking variable*
WEIGHT	INT	0	*Total cabin's load*
KIDS	INT	0	*Number of kids in the cabin*
WOMEN	INT	0	*Number of women in the cabin*
MEN	INT	0	*Number of men in the cabin*
FATGUYS	INT	0	*Number of fat guys in the cabin*
KID_IN	BOOL	FALSE	*Kid entering simulation*
WOMAN_IN	BOOL	FALSE	*Woman entering simulation*
MAN_IN	BOOL	FALSE	*Man entering simulation*
FATGUY_IN	BOOL	FALSE	*Fat guy entering simulation*
KID_OUT	BOOL	FALSE	*Kid exiting simulation*
WOMAN_OUT	BOOL	FALSE	*Woman exiting simulation*
MAN_OUT	BOOL	FALSE	*Man exiting simulation*
FATGUY_OUT	BOOL	FALSE	*Fat guy exiting simulation*
COUNTER_IN	BOOL	TRUE	*People entering enabling*
COUNTER_OUT	BOOL	TRUE	*People exiting enabling*
OVERLOAD	BOOL	FALSE	*Overload declaration*
FIRST	BOOL	TRUE	*Car LEDs turning off declaration (emergency)*
BO	BOOL	FALSE	*Blackout simulation*
RECOVERY	BOOL	FALSE	*Recovery operation simulation from blackout*
LL_EME	BOOL	FALSE	*Emergency cabin lamp*
FLAG_BO	BOOL	FALSE	*Blackout condition declaration*
FLAG_REC	BOOL	FALSE	*Recovery condition declaration*
FIRE	BOOL	FALSE	*Fire simulation*
TIMER_EME	TON	-	*Emergency activation timer*
TIMER_EME_OFF	TON	-	*Emergency deactivation timer*
FLAG_EMG	BOOL	FALSE	*Emergency condition declaration*

Appendix C

Global variables – Logic with storage

Variable	Type	Init value	Description
FLAG	INT	0	*Current working condition*
BUTTON	INT	-	*Current highest/lowest requested floor*
MOTION	INT	2	*Cabin's moving direction (2=idle)*
MOTION_TEMP	INT	-	*Last direction before STOP pushing*
FLOOR	INT	5	*Last recognized floor*
PUSHED	BOOL	FALSE	*At least one button pushed declaration*
BUT_TEMP	INT	-	*Last requested floor*
BUTTON_CAR	BOOL ARRAY	FALSE	*Car buttons memorization array*
BUTTON_UP	BOOL ARRAY	FALSE	*Going up buttons memorization array*
BUTTON_DOWN	BOOL ARRAY	FALSE	*Going down buttons memorization array*
MAX_BUT	INT	0	*Maximum requested floor*
MIN_BUT	INT	3	*Minimum requested floor*
FIRST_BUT	BOOL	TRUE	*First button pushed declaration*
BUT_1	INT	-	*First button pushed*
FIRST_BUT_CAR*	BOOL	TRUE	*First car button pushed declaration*
BUT_1_CAR*	INT	-	*First car button pushed*
EXTEND	BOOL	FALSE	*Change of direction activation*
UPDATE	BOOL	FALSE	*Max/Min updating declaration when BUTTON=FLOOR*
START*	BOOL	TRUE	*Change of direction activation due to cabin priority*
OBS	BOOL	FALSE	*Fast re-opening indicator*
FLAG_OD	BOOL	FALSE	*Open door declaration*
FLAG_STOP	BOOL	FALSE	*Stop condition declaration*
TIMER_closed	TON	-	*Closing activation timer*
TIMER_opened	TON	-	*Opening activation timer*
TIMER_LL	TON	-	*Timer object for LL led*
BLK	BLINK	-	*Blinking variable*
WEIGHT	INT	0	*Total cabin's load*
KIDS	INT	0	*Number of kids in the cabin*
WOMEN	INT	0	*Number of women in the cabin*
MEN	INT	0	*Number of men in the cabin*
FATGUYS	INT	0	*Number of fat guys in the cabin*
KID_IN	BOOL	FALSE	*Kid entering simulation*
WOMAN_IN	BOOL	FALSE	*Woman entering simulation*
MAN_IN	BOOL	FALSE	*Man entering simulation*
FATGUY_IN	BOOL	FALSE	*Fat guy entering simulation*
KID_OUT	BOOL	FALSE	*Kid exiting simulation*
WOMAN_OUT	BOOL	FALSE	*Woman exiting simulation*
MAN_OUT	BOOL	FALSE	*Man exiting simulation*
FATGUY_OUT	BOOL	FALSE	*Fat guy exiting simulation*
COUNTER_IN	BOOL	TRUE	*People entering enabling*
COUNTER_OUT	BOOL	TRUE	*People exiting enabling*
OVERLOAD	BOOL	FALSE	*Overload declaration*
FIRST	BOOL	TRUE	*Car LEDs turning off declaration (emergency)*
BO	BOOL	FALSE	*Blackout simulation*
RECOVERY	BOOL	FALSE	*Recovery operation simulation from blackout*
LL_EME	BOOL	FALSE	*Emergency cabin lamp*
FLAG_BO	BOOL	FALSE	*Blackout condition declaration*

FLAG_REC	BOOL	FALSE	*Recovery condition declaration*
FIRE	BOOL	FALSE	*Fire simulation*
TIMER_EME	TON	-	*Emergency activation timer*
TIMER_EME_OFF	TON	-	*Emergency deactivation timer*
FLAG_EMG	BOOL	FALSE	*Emergency condition declaration*

*Variables defined only in Storage B