CONTROL SYSTEMS TECHONOLOGIES

AUTOMATION ENGINEERING

ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

ACADEMIC YEAR 2019 / 2020

DESIGN OF AN ELEVATOR USING CODESYS

LORENZO BABINI

IVAN ALBERICO

HAZEM MOHAMED HASSAN MOSTAFA

17 / 02 / 2020

CONTENTS

1. ABSTRACT

Logic control is nowadays a predominant part of industrial automation, and as it improves day by day more, it inevitably spreads in our daily life and in everything that surrounds us.

It is has become almost quite common to use logic control when it is up to design systems such as, for example, electronic appliances for the house or industrial machinery for factories, whether it is for work or just for didactic purposes.

Our project is, in fact, based on the design of the model of an elevator using both **SFC** and **ST** languages in **CoDeSys** (Controller Development System), which is a development environment aimed at implementing the logic control and at programming controller applications according to the international industrial standard IEC 61131-3.

Elevators are part of our daily life and you can find them almost everywhere: from the building where you live, to the places you usually visit like shopping centers, your university or the place where you work.

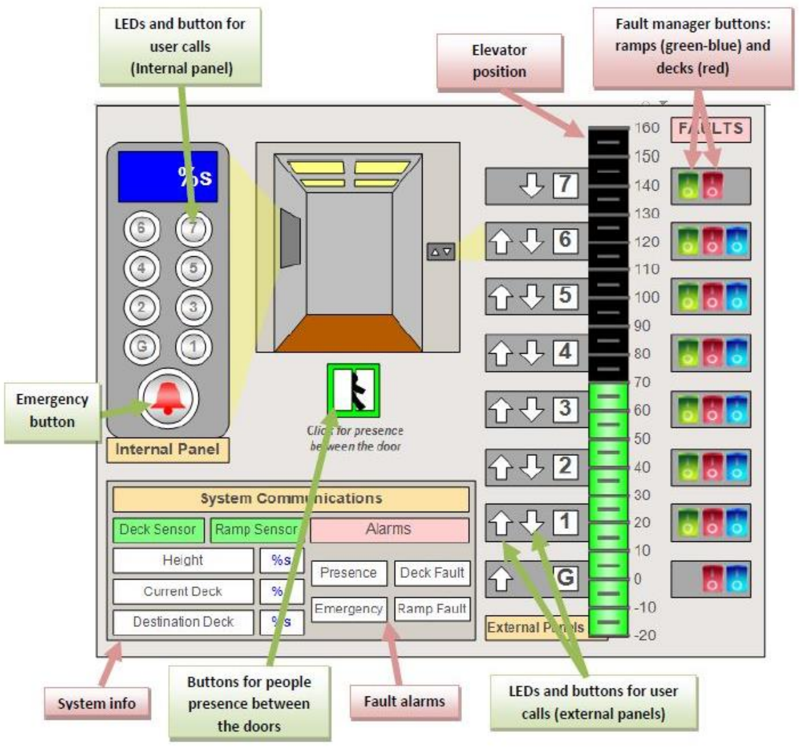
Of course there could have been several ways to implement the model, however the path we chose seemed to us to be the most accurate in satisfying the overall characteristics of the mechanism in the most practical and intuitive way.

The main goal of the project was to put into practice all the theoretical knowledge we learned in class. The whole implementation is mostly based on the frequent use of steps, transitions, synchronizations, parallelisms and IF instructions, which are the basic elements of the SFC and ST languages.

2. INTRODUCTION

First of all, the design is based on the “Generalized Actuator” approach, which uses Function Blocks to guarantee **modularity** and **reusability** and to organize the control program in a **readable** way.

This approach also relies on the definition of a **Policy**, the code that defines the sequence of actions of the mechanism. The Policy first sends an event to trigger the GA, and then the GA executes the action. Later on, a GA sends an acknowledge to communicate to the Policy that the action is completed.


LEDs and button for user calls (Internal panel)
Elevator position
Fault manager buttons: ramps (green-blue) and decks (red)
FAULTS
Emergency button
Internal Panel
Click for presence between the door
System Communications
Deck Sensor
Ramp Sensor
Alarms
Height
Current Deck
Destination Deck
Presence
Deck Fault
Emergency
Ramp Fault
External Panels
System info
Buttons for people presence between the doors
Fault alarms
LEDs and buttons for user calls (external panels)

The variables used to control the elevator are divided into input variables, both sensors and HMI variables, and output variables, the actuators. In total, there are 7 sensors and 5 actuators, and to control the overall system it was enough to use **3 GAs** in total, taking into account that the following two constraints had to be respected:

(1) *“The union of the actuators belonging to the whole set of GAs must be equal to the whole actuators set of the system (A)”*

$$a\_GA\_1 \cup \ldots \cup a\_GA\_i \cup \ldots \cup a\_GA\_n = A$$

(2) *“The sets of sensors and actuators belonging to different GAs must be disjoint”*

$$a\_GA\_i \cap a\_GA\_j = 0 \quad \forall\ i,j \text{ with } i \neq j$$

$$s\_GA\_i \cap s\_GA\_j = 0 \quad \forall\ i,j \text{ with } i \neq j$$

In order to satisfy these previous properties, the variables were grouped in the following way:

GAs	ACTUATORS	SENSORS
Door	Closing, Opening	Open, Closed, Presence
Motor	Motor_on, Vel, Up	LimitDown, LimitUp, DeckSensor, RampSensor
Led	/	/

Each GA has a specific task to fulfill, but this will be seen much more in detail later on. Generally speaking, the **Door GA** manages the opening and the closing of the door, the **Motor GA** regulates the upward and downward movement of the cab, while the **Led GA** manages instead the LED graphical interface (when a button is pushed the corresponding LED has to turn on) and the ordering sequence of the calls.

With respect to this latter part, an effort was made to improve the overall algorithm in the most productive way possible. The algorithm tries to optimize the timing among the calls and the sequence pattern of the inputs: in this way the elevator does not just fulfill the tasks in the mere order they are stored, but tries in an intelligent way to enhance the performances.

Another important building block of the project were **counters**, used in order to track the position of the cab with respect to the deck and ramp sensors.

However, in order to implement counters it was necessary to recall the intrinsic working principle of PLCs, on which CoDeSys is based. Since PLCs work in a cyclical way, the “traditional” design of counters that relies on FOR loops, was not totally accurate: in fact, in logic control, this approach can be risky because it can easily lead the system into an infinite loop. Instead, in the project, counters were designed taking advantage of the rising edge detections, along with some auxiliary variables used to store the information.

EXAMPLE:

```
IF Deck AND NOT AuxDeck THEN
            AuxDeck := TRUE;
            CounterDeck := CounterDeck + 1;
      END_IF
      IF NOT Deck THEN
            AuxDeck := FALSE;
      END_IF
```

3. POLICY

If the GAs are the organs of the program, the Policy would be the heart of it. The policy is like the main that coordinates the calls and organizes the general tasks of the GAs.

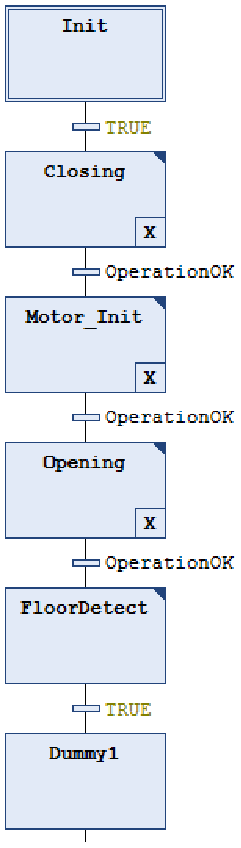
As stated above in the **Introduction**, the **Policy** first sends an event to trigger the GA, and then the GA executes the action. This happens by means of some strings, namely the string "**Ready**" that changes the state of the GA from FALSE to TRUE, and the string expressing the function the GA has to perform. Later on, a GA sends an acknowledge to communicate to the **Policy** that the action is completed.

In the first part of the Policy, we have **Closing**, **Motor_Init**, **Opening** and **FloorDetect**, which are the steps related to the Initialization. **Closing** and **Opening** are both connected to the **Door GA** and they regulate the door behavior.

Between the previous two steps, there is **Motor_Init** which triggers the **Motor GA** to perform the main part of the Initialization, and after that, once the Initialization is done, there is **FloorDetect** that calls the **Led GA** which is a START/STOP GA that continuously tracks the position of the cab and waits for the calls.

The following step is **Dummy1,** which is a reference step, a sort of breakpoint from where the system begins each time a button is pushed, only after the Initialization has been done.

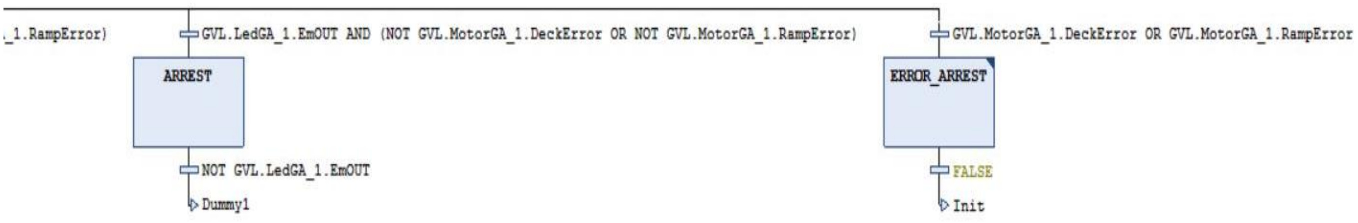
The more detailed steps of the Initialization will be seen in the **Motor GA**, where it is handled.



At this point there are three different branches. The aim is to separate the normal operating conditions from the cases in which some problem occurs, for example when the Emergency button is pressed or a sensor fault (both deck or ramp) is detected. In these two cases, the elevator cannot just continue working as in normal conditions, but it has to behave differently.

In the case of the **Emergency** button being pressed, the cab must stop at the nearest floor possible in order to allow people to come out, and, as long as the Emergency boolean variable is TRUE, we should prevent the elevator from moving to other floors. Once the **Emergency** button is pressed a second time (so Emergency = FALSE), the system is sent again to the step **Dummy1**, ready for the next call.

On the other hand, in the case in which a **sensor fault** occurs, we should behave in a similar way. As soon as the system detects that at a certain floor we have a problem in either a deck sensor or a ramp sensor, the cab must be sent to the next floor. Also in this case we should prevent the elevator to move, but here to restore the normal working condition of the elevator, we have to start the simulation over.

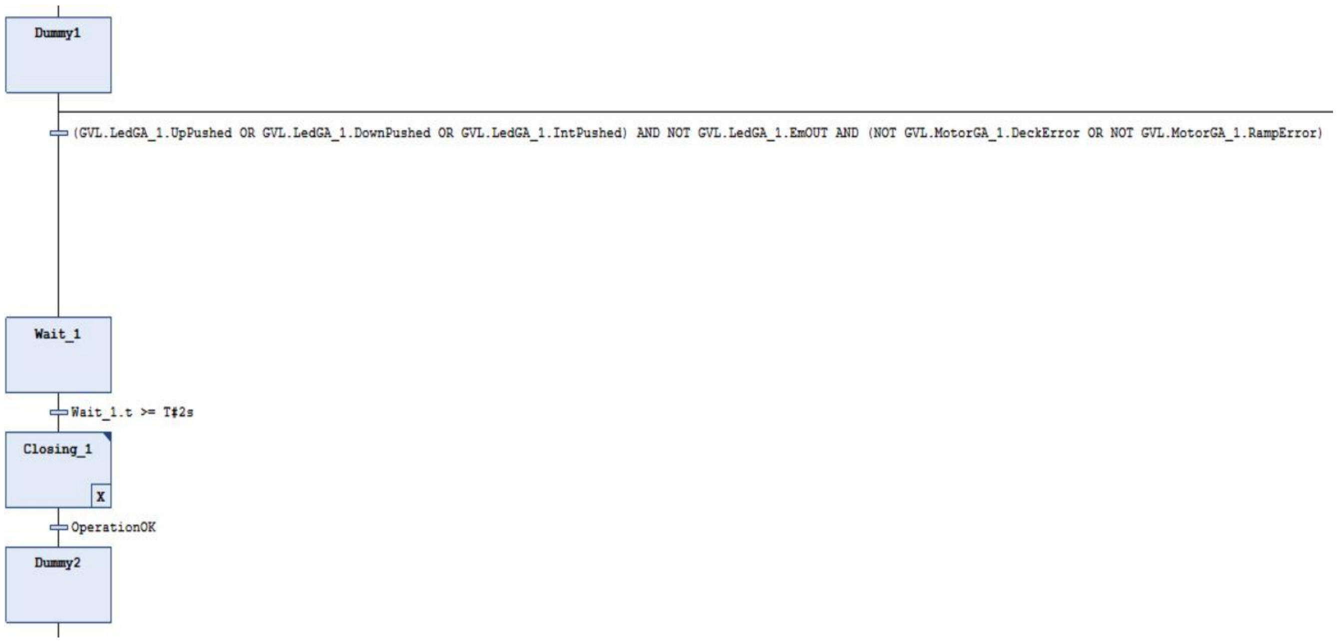
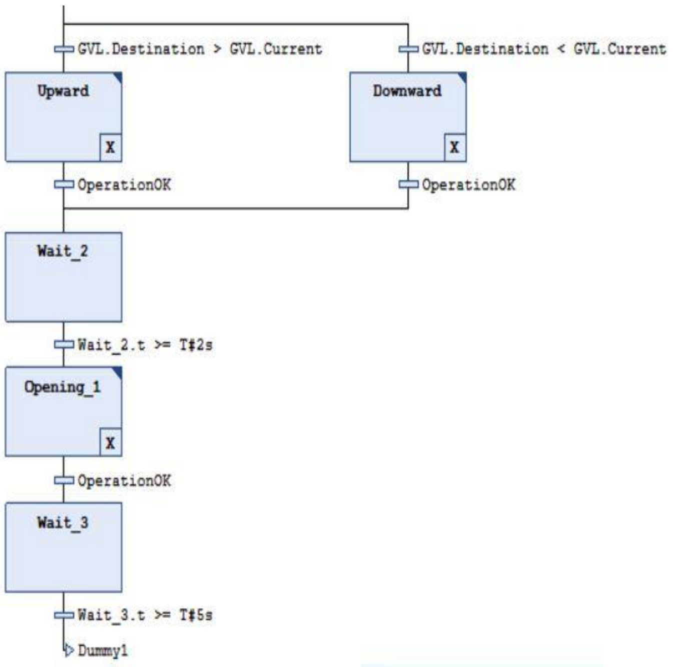


_1.RampError)
GVL.LedGA_1.EmOUT AND (NOT GVL.MotorGA_1.DeckError OR NOT GVL.MotorGA_1.RampError)
GVL.MotorGA_1.DeckError OR GVL.MotorGA_1.RampError
ARREST
ERROR_ARREST
NOT GVL.LedGA_1.EmOUT
Dummy1
FALSE
Init

Now, provided that there is not any fault in the system anymore, when a button is pressed (no matter if it is internal or external) the elevator must go to the desired floor.

In the last part of the **Policy**, in fact, once a button is pressed, the elevator first waits a time interval of 2 seconds, then it closes the door and moves the cab according to the desired floor. At this point of the **Policy** there are two parallel branches, relatively with the steps **Upward** and **Downward** that both trigger the **Motor GA** related to the movement of the cab.

The system is designed in way such that if the destination floor is greater than the current floor, the Upward step becomes active and it tells the **Motor GA** to move the cab upward until the floor is reached, and vice versa for the other case. Once the desired floor has been reached, the system waits another time interval of 2 seconds before opening the doors, then it waits other 5 seconds and goes back to the **Dummy1** step previously met, ready for another call.

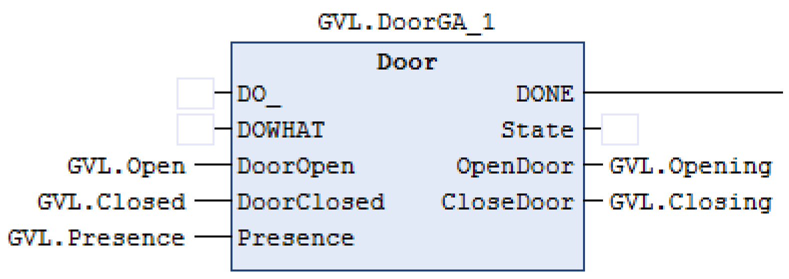

Dummy1
(GVL.LedGA_1.UpPushed OR GVL.LedGA_1.DownPushed OR GVL.LedGA_1.IntPushed) AND NOT GVL.LedGA_1.EmOUT AND (NOT GVL.MotorGA_1.DeckError OR NOT GVL.MotorGA_1.RampError)
Wait_1
Wait_1.t >= T#2s
Closing_1
X
OperationOK
Dummy2

GVL.Destination > GVL.Current
GVL.Destination < GVL.Current
Upward
X
Downward
X
OperationOK
OperationOK
Wait_2
Wait_2.t >= T#2s
Opening_1
X
OperationOK
Wait_3
Wait_3.t >= T#5s
Dummy1

4. DOOR GA

The **Door GA** is the one related to the behavior of the elevator's door. Overall, it is aimed at performing three main tasks:

1) Opening
2) Closing
3) Presence detection

The first two steps of the **Door GA** are the ones that are common to all the GAs, namely **Ready** and **Busy**, which are triggered by the **Policy**. When the state of the GA is 'Ready' and the Policy sets the variable **DO_** to true and expresses in **DOWHAT** the functionality he wants the GA to perform, then **Busy** step is active and the state of the GA goes to 'Busy', meaning that it is starting to work.



Now, depending on what has been expressed in the variable DOWHAT we have two different parallel branches: one for the opening and the other for the closing.

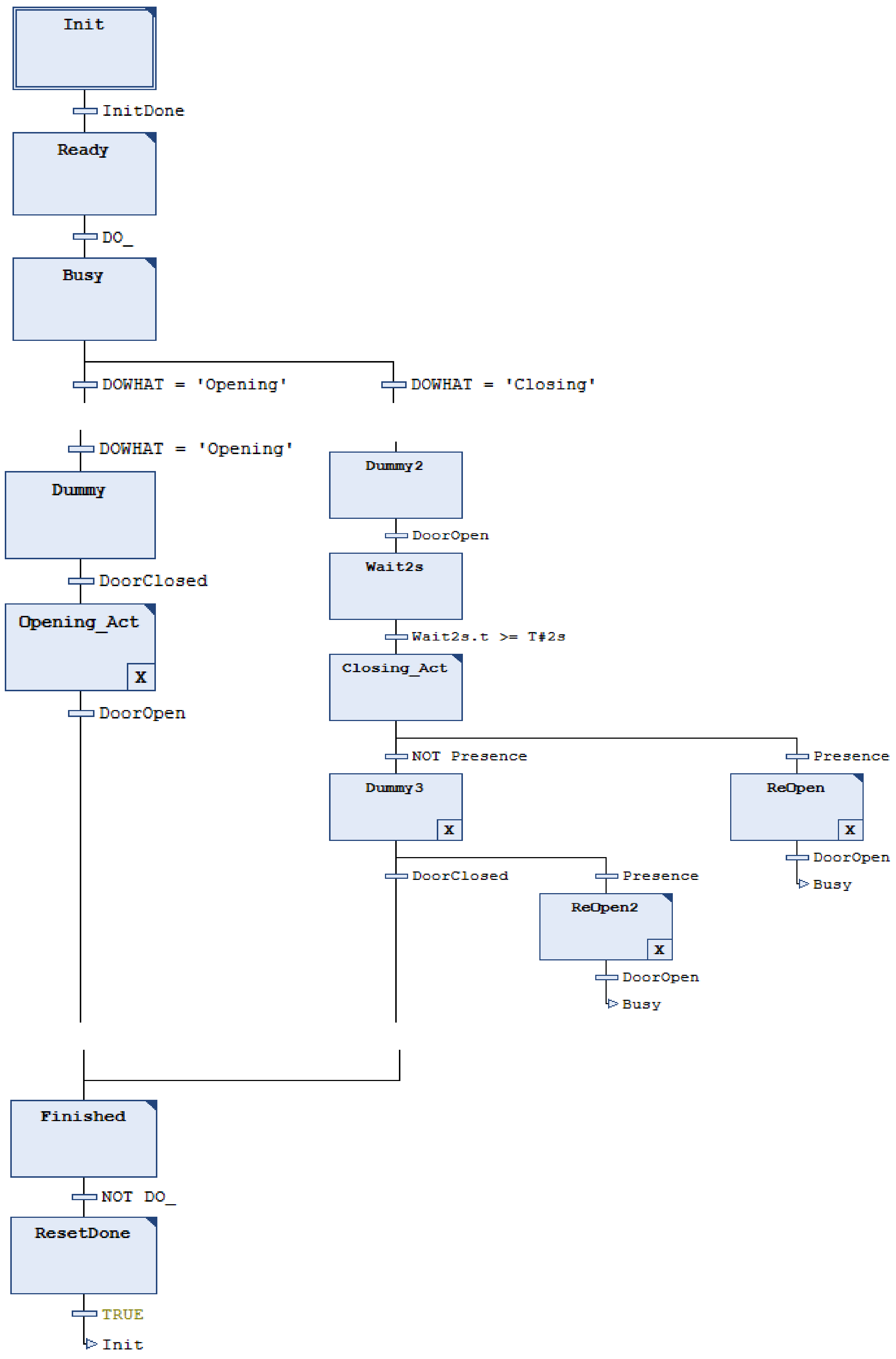
If DOWHAT = 'Opening' and the door is completely closed (**DoorClosed** = TRUE), then the system proceeds to the opening of the door: **OpenDoor**, which is the variable associated to the actuator **Opening**, is set to TRUE and once it is completely open, it is set to FALSE again as an exit action.

In a similar way, if DOWHAT = 'Closing' and the door is completely opened (**DoorOpen** = TRUE), the system first waits 2 seconds and then proceeds to the closing of the door, setting **CloseDoor**, the variable associated to the actuator **Closing**, to TRUE.

What is different now, is that it must be checked if there is something in-between the door, which is hindering the door photocell. If something is detected by the sensor, the variable **Presence** becomes TRUE and the system goes in a parallel branch activating the step **ReOpen**, which is aimed at reopening the door, setting **OpenDoor** = TRUE again.

Once the door is completely open, the system jumps to the **Busy** state again, reiterating the same passages. It is needed to place the parallel branch of the **ReOpen** step twice because it can happen that **Presence** becomes TRUE while the door is already closing. If nothing hinders the door, **CloseDoor** is equal to TRUE until the door is fully closed (**DoorClosed** = TRUE), and then it is set to FALSE again as an exit action.

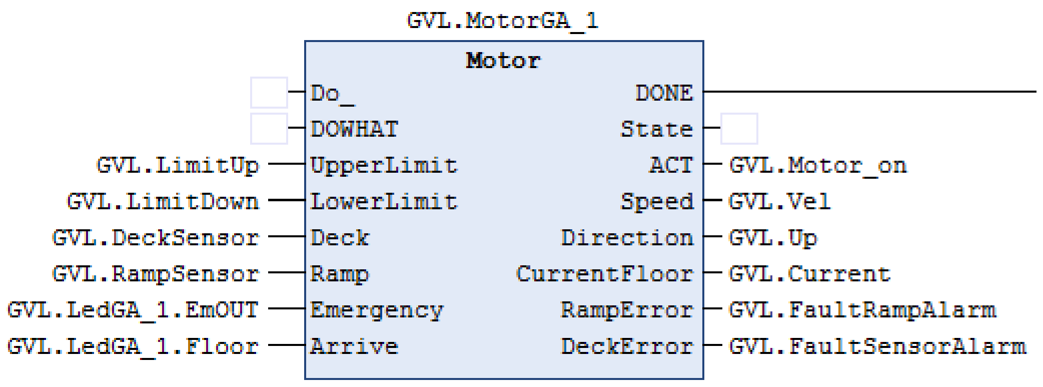
In both cases of opening and closing, once the task has been fulfilled, the step **Finished** is active, in which the variable **DONE** is set to TRUE, meaning that the action is finished. When NOT DO_ = TRUE, the step **ResetDone** becomes active, in which **DONE** is set to FALSE again, and the GA is reset and ready to work again from the beginning.


Init
InitDone
Ready
DO_
Busy
DOWHAT = 'Opening'
DOWHAT = 'Closing'
DOWHAT = 'Opening'
Dummy
DoorClosed
Opening_Act
X
DoorOpen
Dummy2
DoorOpen
Wait2s
Wait2s.t >= T#2s
Closing_Act
NOT Presence
Presence
Dummy3
X
ReOpen
X
DoorOpen
Busy
DoorClosed
Presence
ReOpen2
X
DoorOpen
Busy
Finished
NOT DO_
ResetDone
TRUE
Init

5. MOTOR GA

Both **Motor GA** and **Led GA** are a little bit more complex than the previous one.

The **Motor GA** represents the core of the elevator's control because it handles the movement of the cab. The first part of the GA is the one related to the **Initialization**, which has to be executed only once, at the beginning of each simulation. To do that, a variable **Initia** is used, that is set equal to TRUE only once the Initialization has been done, and that allows to skip this part each time the **Motor GA** is again called by the **Policy**.



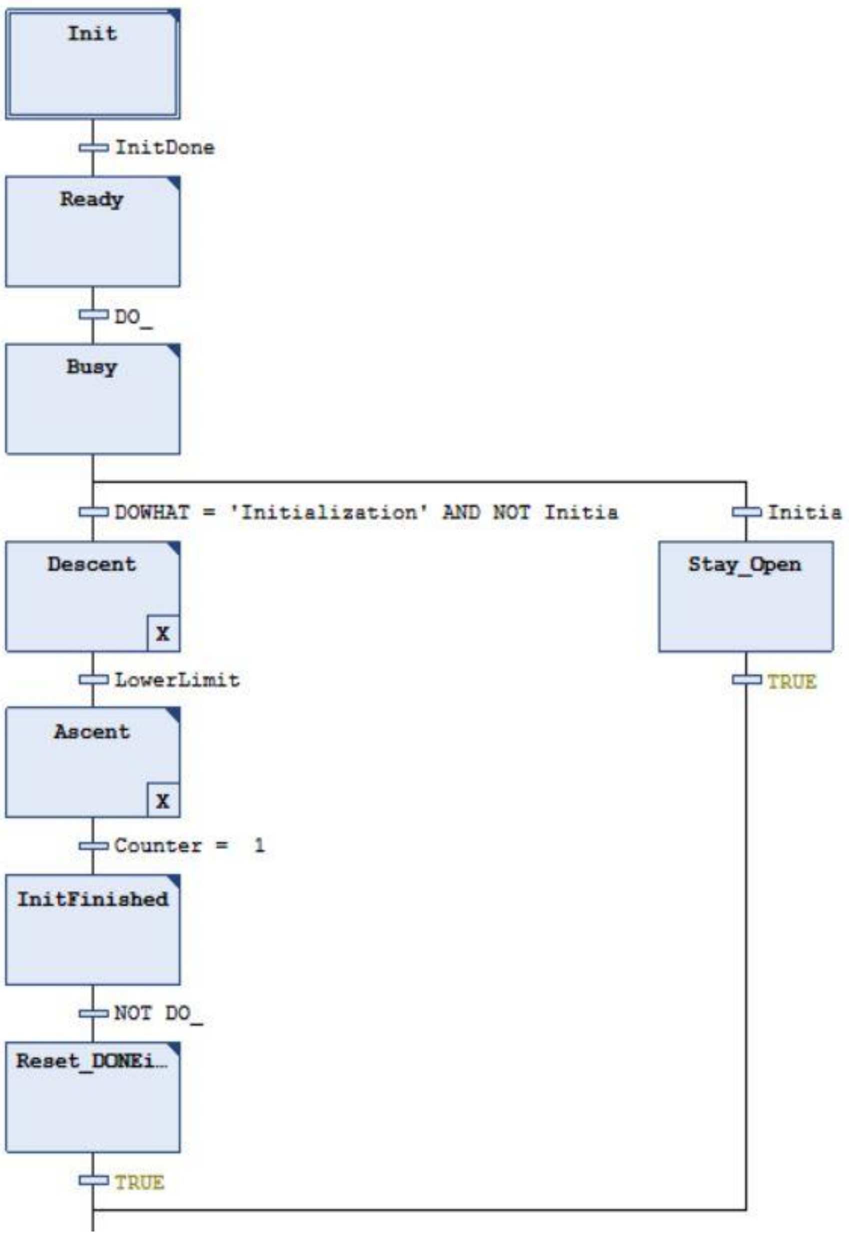
The **Initialization** consists in 5 main steps:

1. Closing of the doors (checking the presence sensor);
2. Descent of the cab to the lower limit sensor with low speed;
3. Ascent of the cab to the first floor with low speed;
4. Opening of the doors;
5. Waiting for calls.

At the very beginning of the simulation DOWHAT = 'Initialization' and Initia = FALSE, therefore the step **Descent** is active and here it is set that **ACT** = TRUE (Motor actuator), **Direction** = FALSE (Up actuator) and **Speed** = FALSE (Vel actuator). These instructions allow the elevator to start moving downward, and it keeps doing so until it reaches the lower limit sensor (**LowerLimit** = TRUE). Now the step Ascent is active, **Direction** is switched to TRUE and so the cab changes its direction and moves upward.

It is right now that counters come into play. A counter is introduced, which increases each time a deck sensor is met, so each time the cab is moved upward of one floor. In our case, since it is necessary to stop the cab at the ground floor, it is enough for the counter to reach the value 1.

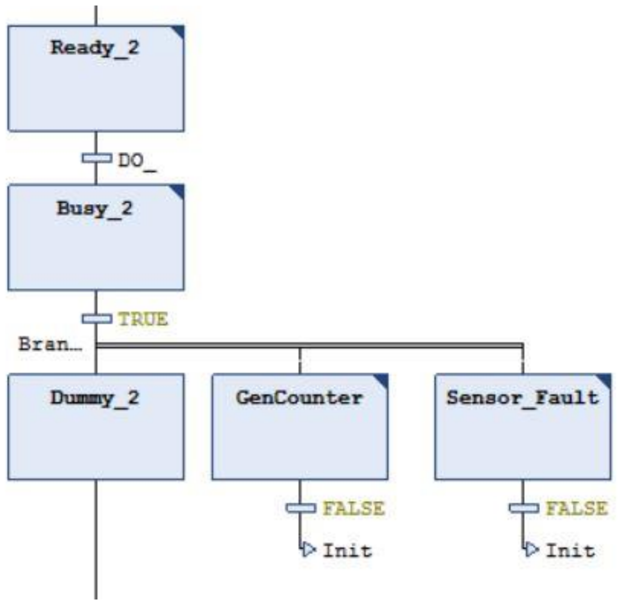
In fact, as soon as **Counter** = 1, the actuator **ACT** is set to FALSE, **Initia** is set to TRUE, and so the elevator stops at the ground floor and the initialization is over.



Following in the **Motor GA**, there is a parallelism with 3 branches, one for the subsequent upward and downward movements of the elevator, one for the counting of the deck and ramp sensors as the cab changes floor, and one for the sensor fault detection, which will be analyzed much more in detail later on.

In the step **GenCounter** we defined the two main counters of the whole mechanism, **CounterDeck** and **CounterRamp**. This step is active for all the execution of the program, and this is possible by setting to FALSE the transition that comes after.

However, since the two counters must be able to continuously register and update the correct number of deck and ramp sensors, as the cab moves up and down, two cases were considered. When DOWHAT = ‘Upward’ both **CounterDeck** and **CounterRamp** are increased by 1, each time a deck sensor or a ramp sensor is encountered. Instead, when DOWHAT = ‘Downward’, they are decreased by 1.



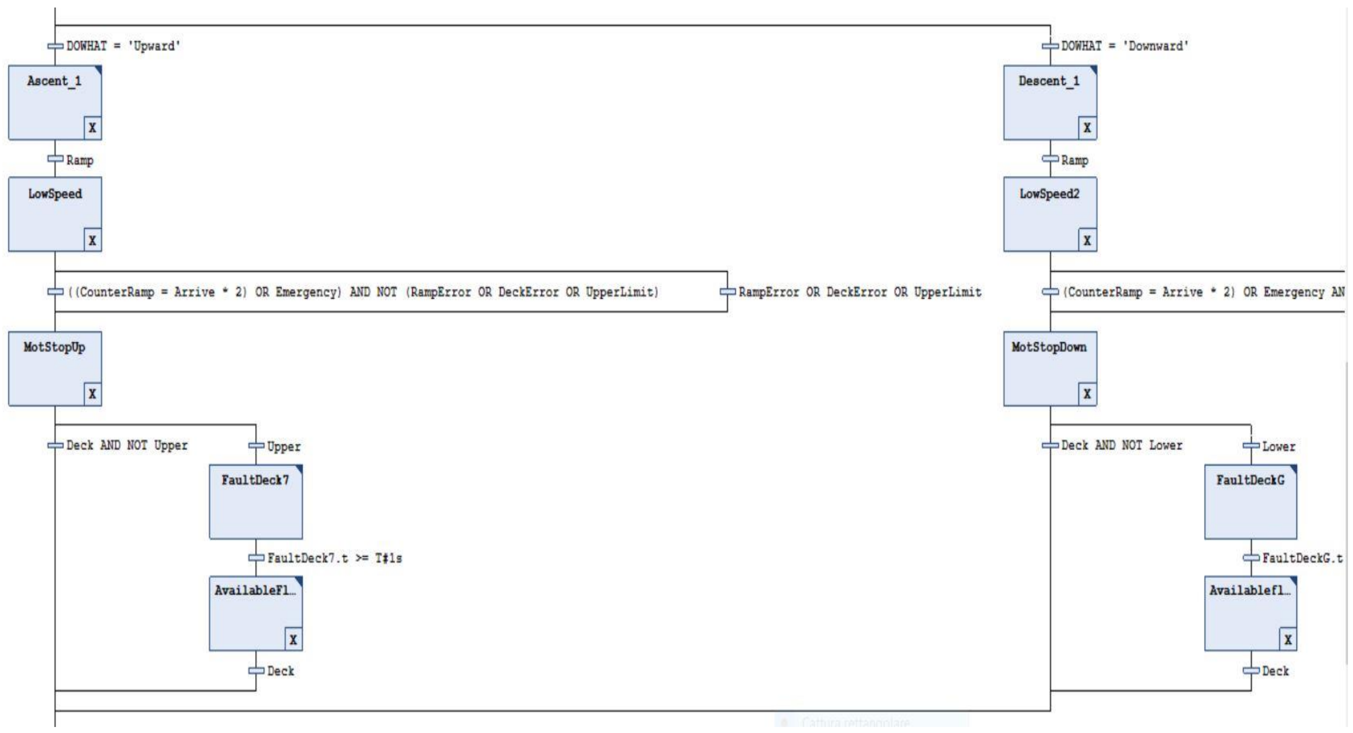
Back to the **Motor GA** main structure, after the parallelism, the main branch of the GA is then divided into other two branches, one for the upward movement and the other for the downward.

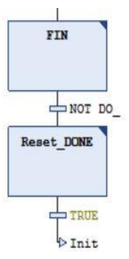
If DOWHAT = 'Upward', step **Ascent_1** is active and so **ACT** = TRUE, **Direction** = TRUE and **Speed** = FALSE. In this way, the cab moves upward with low speed until the first ramp sensor is met. In fact, when **Ramp** = TRUE, we set **Speed** = TRUE in the exit action and so we start moving the cab with high speed.

The cab keeps moving at high speed until **CounterRamp = Arrive * 2** (or until the Emergency button is pressed or a deck/ramp fault has occurred), where **Arrive** is an INT variable that represents the destination floor. When **CounterRamp = Arrive * 2**, it means that the ramp immediately before our destination floor has been encountered, so **Speed** is reset to FALSE in order that the cab reaches the floor with low speed.

After that, once a deck sensor is met (**Deck** = TRUE) it means we have reached the desired floor, so **ACT** is set to FALSE, hence the elevator does not move anymore, or until further indications are expressed.

If DOWHAT = 'Downward', the exact same steps are implemented, but in the opposite direction.





5.1 SENSOR FAULT

The sensor fault detection is handled in one of the 3 parallel branches of the **Motor GA**. The relative step, **Sensor_Fault**, is maintained active throughout the whole execution of the program, since its next transition is set to FALSE.

In order to detect faults in the deck/ramp sensors, two counters are used, namely **n** for ramps and **m** for decks. Counter **n** increases each time a ramp sensor is detected, while m increases each time a deck sensor is detected.

If no fault is present, we can deduce that each two ramps detected, a deck is found. So when **n** = 2 and a deck sensor is detected, n is reset.

When a deck sensor is detected, the system checks two conditions: that no other deck sensor has been encountered and that no more than two ramp sensors have been detected (the condition used are **m = 1 AND n = 0**, and n is chosen equal to zero because the condition to reset it, is placed before this one).

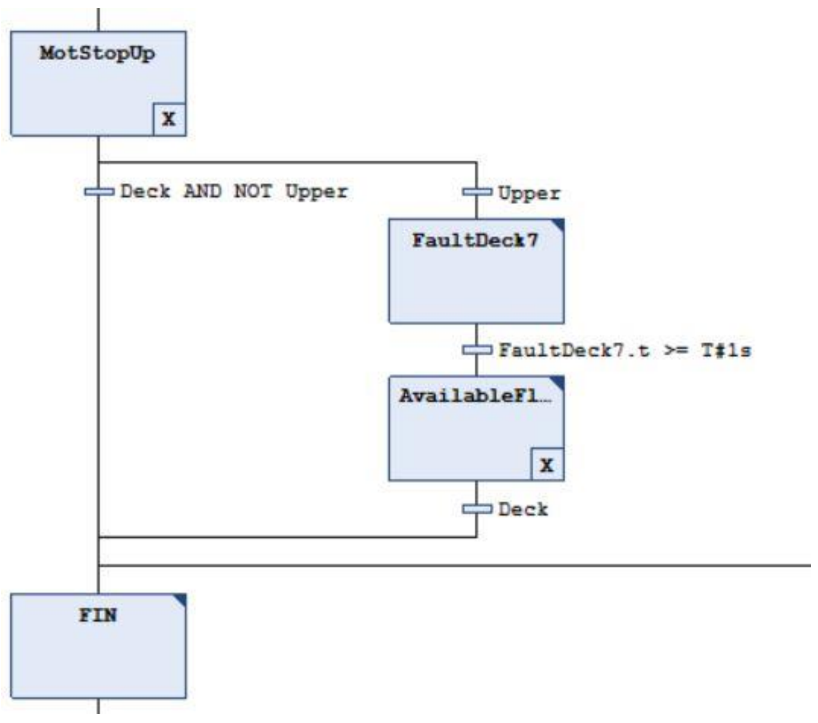
When a deck sensor is detected and n is not equal to zero, it means that a ramp sensor has been skipped or it does not work properly, then **m** is not reset to 0 and so ramp error is shown.

When **n** is larger than 2 means that a deck sensor has not been encountered due to the fact that **n** has not been reset, so deck error is shown.

The variable **Begin** is used as sometimes the system increases m when the program starts working. In this way, m is reset at the first ramp sensor and then **Begin** is not used anymore.

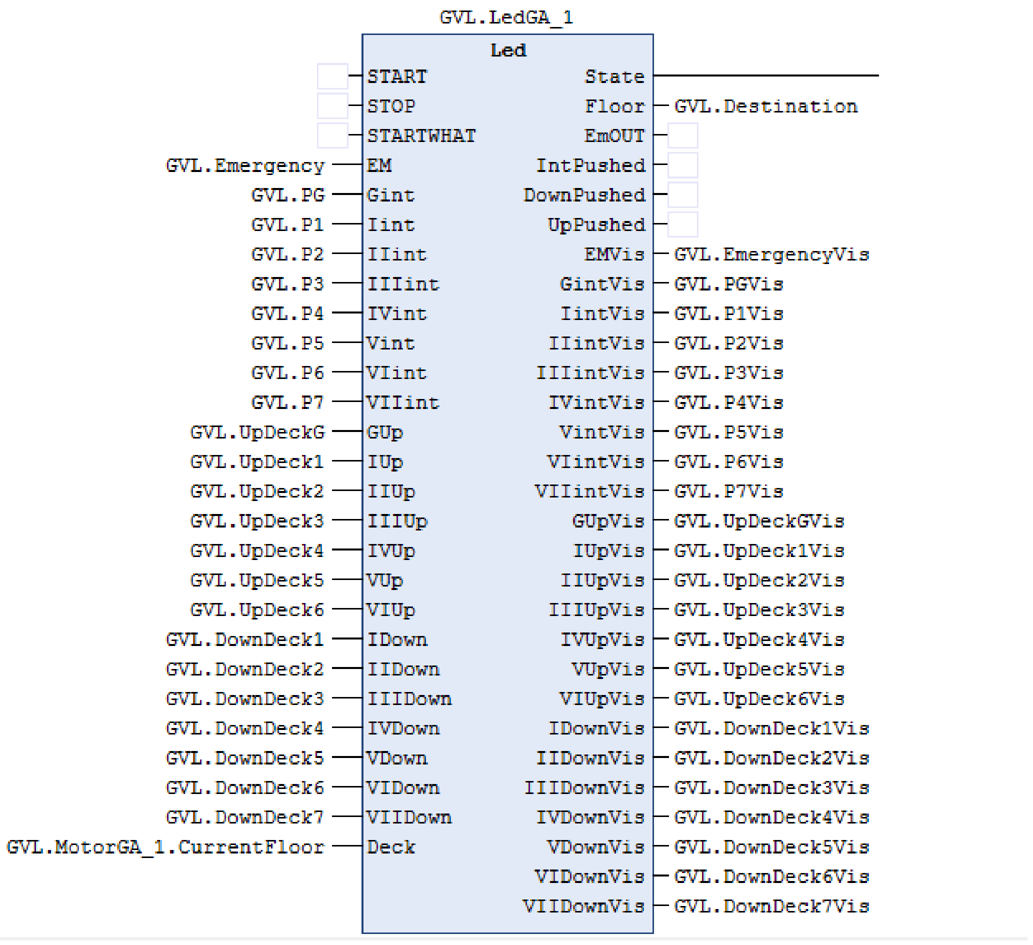
Particular care should be taken if faults happen at 7th or ground floor. If the fault is due to a ramp sensor, the speed is immediately reduced in order to reduce the risk for people inside the car. The cabin reaches the upper or lower limit and autonomously goes to the first safe floor. If the fault happen at the deck sensor, there is no possibility to reduce the speed of the cabin before it reaches the upper or lower limit. As before the cabin will reach the first safe floor on its own.

This latter part is implemented outside **Sensor_Fault** and it is present in both the '**Upward'** and '**Downward'** branches of the **Motor GA**. In fact, if Upper = TRUE, which means that the **UpperLimit** sensor is met, step **FaultDeck7** becomes active (the same is done in the 'Downward' branch with Lower = TRUE and **FaultDeckG** becoming active), the system waits 1 second and then it switches to the low speed until the first possible floor is reached, which is the 7th.



6. LED GA

The **Led GA** is the one that manages the LED graphical interface (when a button is pushed the corresponding LED has to turn on), but also sets the priorities among the calls and handles the **Emergency** fault. **Led GA** is a START/STOP GA, so from the moment it is called by the **Policy** in the **FloorDetect**, it continues working for the whole program execution (until otherwise declared in the **Policy**).



The GA basically consists in a parallelism of 11 different branches: 8 of them are related to the floors involved, one is for the **Emergency**, one for the **Ordering** and one for the **Reset**.

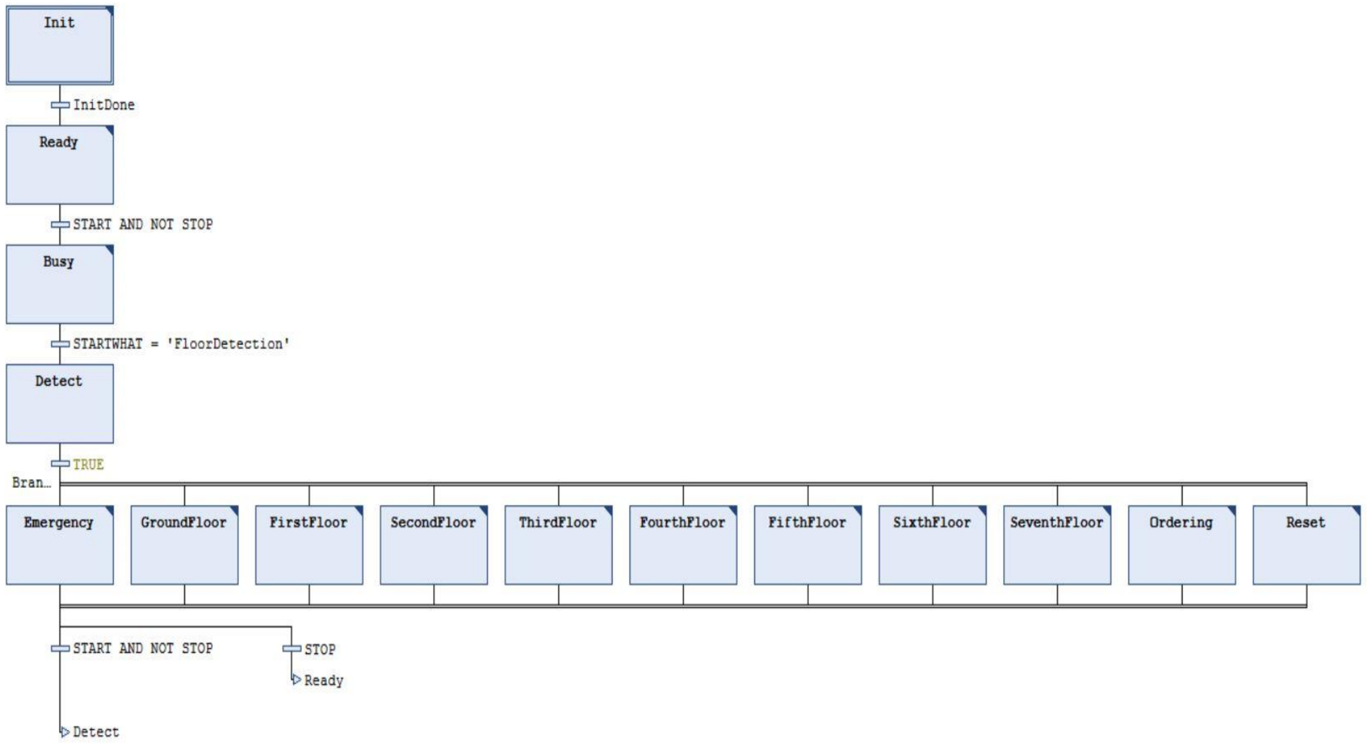
In each step related to the specific floor, a connection between the pushed button and the equivalent LED is made. For example, in **GroundFloor**, to establish a connection between the internal button of the cab and the equivalent LED in the visualization, the following code has been used:

```
IF Gint AND NOT AuxG1 THEN
        AuxG1 := TRUE;
END_IF
IF AuxG1 THEN
        GintVis := TRUE;
        IntPushed := TRUE;
END_IF
IF Floor = 0 AND Deck = 0 THEN
        GintVis := FALSE;
        AuxG1 := FALSE;
END_IF
```

This is repeated also for the external buttons (both **Up** and **Down**) and for each floor involved.

In the step **Emergency**, a connection between the emergency button and the relative LED is created. Once the emergency button is pressed, the **Led GA** sends as an output the variable **EmOut** to the **Policy**, which makes the elevator stop at the possible next floor. When the **Emergency** button is pressed a second time, **EmOut** goes to FALSE again, and the elevator restarts working as before.

In **Reset**, instead, **IntPushed** is reset to FALSE when all the cells of the array **Intern[ ]** are FALSE again (all the destinations had been reached), and this is done for both internal and external buttons



However, it is in **Ordering** that all the priorities of the calls are handled. When a button is pushed, the corresponding array cell is filled and it is reset only when the destination is reached. Three arrays are present, one related to the cabin buttons, the other two related to the floor buttons divided in upward and downward direction.

The priority is given according to the direction. The first button that is pushed is served first, and it is the one setting the initial direction priority. At the same time, the system checks if other buttons in other floors are pushed, before we get to the first destination, and this is done using a counter. At the beginning, the counter starts counting from 0 to 7 because no other button is pushed. When a call occurs, the system sets the priority, making a comparison between the current floor and the destination one. The system assigns as current value for the counter the current floor number. In this way, checking from the current floor to the maximum or minimum one, and depending on whether the priority is upward or downward, the counter is able to understand if it can serve other floors before the destination one.

For example, if the fourth floor button is pushed and the cabin is at the ground floor, priority is given to the upward movement. If the buttons inside the cabin are pressed, having a destination floor closer with respect to the first destination, or if external upward buttons are pressed, which are at lower floors with respect to the destination one, the cabin first stops at those. Downward direction buttons are not considered (if pushed, the relative array cells are filled but they will not be served until the priority is changed).

Until no button is pushed, the system continuously checks whether the priority is respected and changes it if no other button in the same direction is pushed.

If no button is pushed, the system does not check the priority but remain idle waiting for a call.

7. CONCLUSIONS

After designing what it seemed to be a ‘simple’ elevator, a new sensibility for the effort that goes into dealing with logic control was obtained.

There are numerous methods and approaches that can be taken to achieve the same goal, some simpler than others, and this was a valuable lesson.

Designing even what it seems to be simple may take you a lot of time and might not be very straightforward.

We hope this was only a beginning and that someday we will be able to design also way more complicated things.