Buriani Gioele 874477
Galletti Vittorio 882862

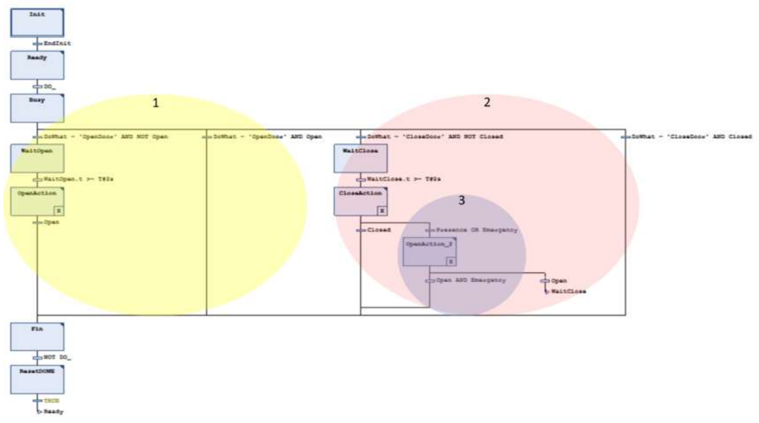
SINGLE ELEVATOR REPORT

INTRODUCTION

The project consists in the development of a control system for a single elevator. In this project three different coding languages have been used: SFC (Sequential Function Chart), ST (Structural Text) and FBD (Function Block Diagram). SFC was used to define the sequential logic of the elevator, defining different stages and actions order, ST language was used to explain the actions inside of the SFC blocks and FBD was used to define the GAs.
The method used for the project is the Generalized Actuators (GA) approach: this also helped to better detect and handle faults in the system.
The intent was to simulate an elevator with an algorithm as realistic as possible that aims at having as few people as possible inside the elevator at the same time. Also, the order the calls were made was prioritized with some optimization that allowed to satisfy each call in the shortest possible time. Moreover, the HMI interface utilization has been optimized as much as possible (only the correct LEDs turn on and off). For safety reasons we assumed that the doors of the elevator should stay closed when the system is at rest.
Based on the sensors and actuators to be handled, two GAs have been defined: one for the opening and closing of doors and the other one for the movement of the elevator body.

DOORS GA

This GA has the aim to control the opening and closing of the door of the elevator taking into account the possible presence of something between the doors. It controls the *Presence*, *Open* and *Closed* sensors and the *Opening* and *Closing* actuators.



1 Opening branch
When the GA is in the state ready and the *DoWhat* input becomes 'OpenDoors', this process starts. If the door is not already open (resulting from some possible fault), after 2 seconds (from guideline) the door starts opening. As soon as the *Open* sensor becomes true (or if it was true from beginning) the GA is finished and goes back to the *Ready* state.

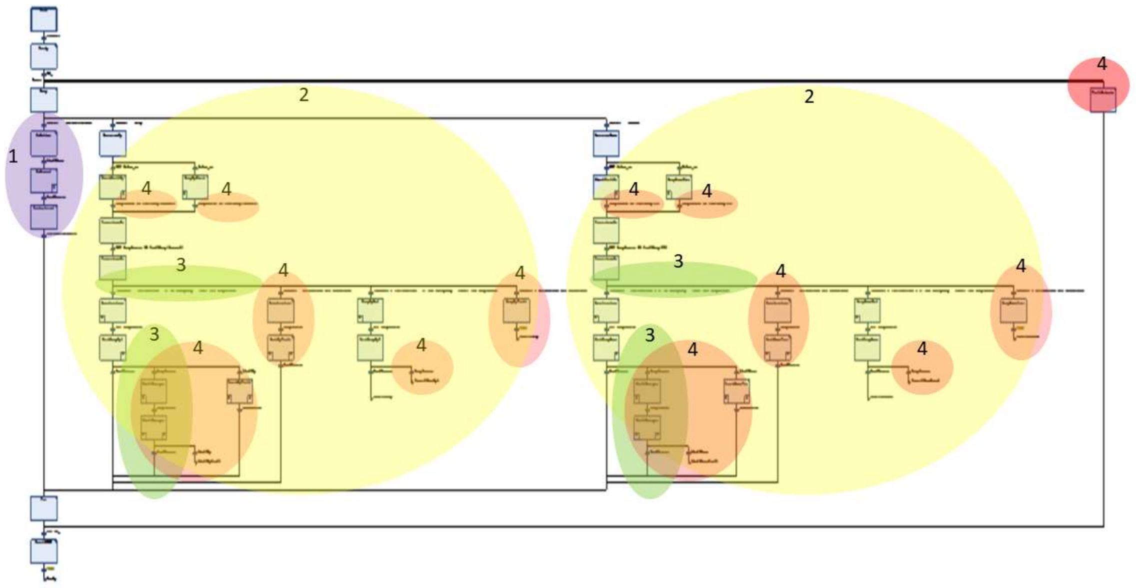
2 Closing branch
The principle behind this branch is very similar to the Opening branch, the main difference is the fact that the *Presence* sensor must be taken into account. Also the handling of emergency differs from the first case, but this will be better analysed in the Emergency section.
If the presence sensor becomes true during the closing action, a transition activates an opening step that, when completely fulfilled, takes the active step back to *WaitClose*. As long as the *Presence* sensor remains true (something is standing between the doors) the GA will be stuck in this loop and the doors will remain open.

3 Emergency management
Part dedicated to the management of the GA behavior in case of *Emergency* button activation. Better described in the Emergency section.

Buriani Gioele 874477
Galletti Vittorio 882862

MOVE GA

This GA manages the actual moving of the elevator, controlling the motor of the system on the base of the information given by several sensors displaced on the elevator path. It controls the *DeckSensor*, *RampSensor*, *LimitUp* and *LimitDown* sensors and the *Motor_on*, *Up* and *Vel* actuators.



1 Initialization branch
This branch has the only goal of controlling the action of the elevator during the initialization phase. It is therefore called by the policy only once during the whole “life” of the system (if everything works properly).
The operation consists in moving the elevator down at low speed until it reaches the *LimitDown* sensor, then it reverts its direction and goes upwards until it reaches the first *DeckSensor* and stops. At this point, the *Current* variable is set to 0 and the initialization is finished.

2 GoUp and GoDown branches
These two branches form the main body of the GA and are devoted to ensure the nominal functioning of the GA. The two branches are perfectly symmetric, so only the *GoUp* branch will be analyzed. Also, the parts of the branch devoted to emergency and fault behavior will be skipped in order to be better described in the respective sections.
The branch has a cyclic behavior that is meant to restart each time the elevator reaches a *DeckSensor*, unless it reaches the destination deck: in this case the GA action is finished. The destination is known to the GA as soon as the policy has finished computing it (*GVL.Current* is set as an input of the GA). In this way the elevator will be able to adapt its behavior in real-time.
At the top, the first alternative is to detect whether the elevator is at standstill or it is already moving (meaning that it has just passed through a deck at which it did not have to stop): in the first case the system will start the motor for upward movement at slow speed, until it reaches the first *RampSensor* after which it increases the speed.
After a transition phase, there is another alternative choice (only the first and third branch will now be considered). This time the system detect whether the elevator is meant to stop at the next deck (*Current* = *Destination* - 1) or not: in the first case the elevator decelerates and updates the current floor (the reason will be explained in the Policy section), in the second case it will keep going and updates *Current*. Lastly, as soon as the elevator reaches a *DeckSensor*, in the first case it stops and the GA is finished, in the second case there will be a jump to the beginning of the branch and the process starts again.

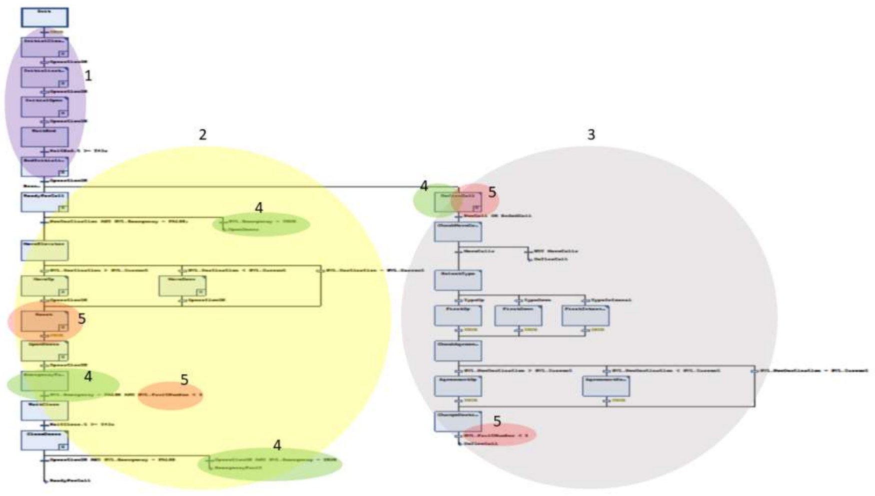
3 Emergency management
Part dedicated to the management of the GA behavior in case of *Emergency* button activation. Better described in the Emergency section.

4 Fault management
Part dedicated to the management of the GA behavior in case of fault buttons activation. Better described in the Faults section.

Buriani Gioele 874477
Galletti Vittorio 882862

POLICY

It works as the “brain” of the system: manages the initialization phase, the sequence of actions for nominal behavior (activating the proper GAs) and implements the priority algorithm.



1 Initialization phase
It is the first phase of the program, allows the system to start from a known fixed position.
It starts by closing the doors (thanks to the *DoorsGA*), then calls the *MoveGA* to move the elevator in the correct way (defined in the GA body) and lastly opens the doors again (once again using the *DoorsGA*). A further closing of the door has been added to the initialization phase in order to guarantee the safety condition described in the Introduction section.

2 Action Sequence branch
This branch dictates the sequence of actions that have to be performed in order to satisfy each call. The parts of the branch dedicated to emergency behavior will be skipped in order to be better described in the Emergency section.
The branch starts from the *ReadyForCall* step, that remains active as long as the elevator is “at rest”. As soon as the *NewDestination* variable becomes true (the Priority Algorithm branch has elaborated a new destination), the system activates.
Depending on the position of the destination deck with respect to the current one, the policy activates the adequate branch of the *MoveGA* determining the movement direction of the elevator. If the destination coincides with the current deck (the elevator is called to open the doors), the *MoveGA* is not called.
The *Reset* step is activated as soon as the *MoveGA* has finished its operation (the elevator has reached a deck). Now the system needs to adjourn the calls received: based on the call that has just been satisfied, the system will turn off the corresponding LED of the HMI interface, lower the priority of all the successive calls and the *NextPriority* value (priority will be better described in the next paragraph) and cross out the call from the call list. Also, at the end of the process the variable *EndedCall* is set to true in order to trigger the next computation on the Priority Algorithm branch. After this, the elevator will open its doors, wait 5 seconds and then close the doors again (again, for safety reasons). The branch will then jump back to the first step ready to satisfy the next call (or remain at rest).

3 Priority Algorithm branch
This branch remains permanently active as it is put in parallelism with the Action Sequence branch: this structure allows the policy to detect and arrange the incoming calls while running the various actions in the meantime. In fact the function of this branch is to read the inputs of the HMI interface and, based on the calls the have been made, compute the next destination to be reached by the elevator.
The first step is *DefineCall* and is used to detect the inputs: as soon as a call is made, it is inserted on the call list depending on its type (*CallUp*, *CallDown*, *CallInternal*), is given a priority depending on *NextPriority* value (starts from 1, each time a call is made it increases by 1, each time a call is fulfilled it decrease by 1) and the corresponding LED is activated. Each time the *NewCall* value is set to true. The transition happens either because of this *NewCall* variable, or because of the *EndedCall* variable (that was mentioned before): the algorithm is run either if a new call is received, or if an old call has been satisfied. This step is almost always active as all further operations take place in few cycles. The next passage simply determines if there are other calls to be computed or the system can go back to rest.

Buriani Gioele   874477
Galletti Vittorio   882862

The real algorithm starts now: at first the call with priority 1 is set as *NextDestination* (actually the deck corresponding to the call). Then, depending on the type of the first call (Up, Down or Internal) the system verifies if there is another call of the same type that would allow the elevator to optimize its behavior: if a call from deck 2 has been made to go up and then another call from ground floor has been made to go up as well, it would be convenient to first go to ground floor (even if the other call should have the priority) and then stop at deck to while going upwards. This step (one for the Up calls and one for the Down calls) allows the system to implement this behavior by changing *NextDestination* into the said call (if present).
The next step allows the elevator to further optimize its behavior by stopping at any deck that is "in the way" of the first call: any internal call and any external call with the direction agreeing with the direction of movement. This allows the elevator to satisfy more calls without moving any further distance. Once again said calls (if present) are stored in the *NextDestination* variable.
At last, the *Destination* variable is made equal to *NextDestination* (updating it only at the end makes the destination clearer for the HMI interface). The destination is immediately updated in the GA so that a call for the next deck available can be immediately executed. The GA however will only consider the call if it is made before the second ramp sensor (the current will be updated at that point): in this way we avoid abrupt decelerations. The branch then jumps back to the *DefineCall* step.

3 Emergency management
Part dedicated to the management of the Policy behavior in case of *Emergency* button activation. Better described in the Emergency section.

4 Fault management
Part dedicated to the management of the Policy behavior in case of fault buttons activation. Better described in the Faults section.

EMERGENCY

This event consists in someone from the inside the elevator pushing the *Emergency* button. This should result in the elevator stopping at the nearest deck and the subsequent opening of the doors (that should remain open until the button is pressed again).
The handling of this event has been subdivided in the 3 main parts of the program.

1 Doors GA
This part has been inserted to cover the case in which the button is pressed during the closing of the doors.
In principle it works similarly to the event of the *Presence* button, but this case is a little different: if the emergency is deactivated before the doors are completely open, the system jumps back to *WaitClose* as in the *Presence* case. If instead *Emergency* is still true when the doors are completely open, the GA terminates.

2 Move GA
his part is used to ensure the elevator stops at the first possible deck in case of *Emergency* activation during elevator movement.
The main part consists in an OR condition on the first branch in case the *Emergency* condition is true. This ensures that, even if the following deck would not be the destination, the elevator stops there anyway and stops the GA. A further part is inserted in the bottom part to cover the case of *Emergency* called before a deck with a broken sensor: the elevator would keep going until it reaches the next deck.

3 Policy
The emergency management in the policy handles the activation of the LED (in the *DefineCall* step), the case of *Emergency* activation while the system is at rest (but the doors are closed for safety reasons) and the stall of the system until *Emergency* deactivation.
The first task has already been covered, the second one is made through a transition after *ReadyForCall* step that results in a jump to *OpenDoors* step. The third task is implemented through a step *EmergencyFault* placed after the *OpenDoors* step in order to stall the system after the doors have been opened. In case of *Emergency* during the last closing of the doors, a jump (activated only if *Emergency* is true) has been placed to bring the active step to *EmergencyFault*.
It is important to notice that the emergency does not impair the call detection system.

Buriani Gioele 874477
Galletti Vittorio 882862

FAULTS

Faults consist in the malfunctioning of one of the sensors displaced along the elevator path. They can be both deck faults and ramp faults, depending on the type of sensor missing. The fault detection is handled entirely by the *MoveGA* as part of the low level diagnostic. The behavior after the fault detection is partly managed by the policy too. The fault management is divided in three phases: fault detection, behavior on detection and nominal behavior after detection.

1 Move GA

The detection part is handled entirely in this GA thanks to the *FaultDetection* step. This step analyzes the sensors met in relation to the last sensor registered: if the sequence of deck – ramp – ramp – deck… is interrupted, the system detects a fault (deck fault for 3 consecutive ramps, ramp fault for only one ramp between 2 decks). This step also detects the limit case of deck faults consisting of the elevator reaching the *LimitDown* of *LimitUp* sensors (in case of deck faults at decks G and 7). After the detection of a fault, the system memorizes it in the two vectors *FaultDeck* and *FaultRamp* depending on the type of fault. When the elevator moves this step is always active thanks to parallelism structure with the *Busy* step (only after initialization phase is finished).
For the behavior on detection, we should differentiate between deck and ramp fault. For the first case, two structure were implemented in the first and third branch in the ending part of the GA: if instead of a deck sensor (nominal behavior) the elevator encounters another ramp sensor, a series of transitions and jumps ensures an acceptable behavior both in case of fault on an intermediate deck (the elevator will simply keep on its course) and in case of fault on the destination deck (the elevator will stop on the following deck, in case of decks G or 7 it will go to the corresponding limit an then back to deck before the malfunctioning one). For the ramp fault case, two additional branches have been added in the ending part the handle the two possible cases. The fourth branch is dedicated to the case of ramp fault between two intermediate decks: in this case the elevator will detect the fault and will keep on its course. The second branch, instead, regards the situation in which the ramp fault is immediately before the destination deck: in that case the elevator detects the fault, starts decelerating when it reaches the deck, then, after a while, goes back to the destination deck with low speed.
For the nominal behavior after detection, the deck faults are handled in the policy, while in that GA has been added a system to dedicated to ramp faults. At the beginning of the Action Sequence branch, if the system knows the existence of a ramp fault between the two decks it is passing through, the GA skips a the first part so that at the first (and only) ramp sensor met it immediately goes to the final branches. The result is that the nominal behavior is preserved and the elevator will always move at low speed between those two decks.

2 Policy

The policy part is only dedicated to the nominal behavior after detection of deck faults. In particular in the *Reset* step the program eliminates all calls made for the malfunctioning deck, turns off the corresponding LEDs and lowers the priority of all the other calls. Also, in the *DefineCall* step, the system avoids further calls to be made for that deck. In this way the malfunctioning deck will be cut off from the elevator functioning.
The last consideration about fault is a limit of faults after which the elevator stops working altogether. In this case this limit has been set to 2: after two faults detected the elevator will open the doors correctly and then stop working.

CONCLUSION

For the conclusion, there are some final considerations.
Algorithms have been implemented using arrays in order to make the project independent from the number of floors and to simplify the implementation of the concept of priority.
The use of time measurement to help in case of faults was taken into account: time taken by the elevator to do several activities could have been calculated during the initialization phase and then these information could have been used to recognize the fault ramps (understand which ramp sensor is malfunctioning) or to ensure nominal behavior to malfunctioning decks. However, not knowing how motor is controlled (speed control, torque control,…) and due the possible presence of delay due to friction or usage, these methods to face the low-level diagnostic has been avoided.
Talking about the high-level diagnostic, for this project it is hard to have problems with communication between policy and GAs. The project was based on a simulation and there was no reason for the occurrence of external perturbations. In case of need, an implementation that compares counters for both the GAs and the policy could be used.