Single-Elevator Report

**Filippo Bosi - Tommaso Ceccarelli**

#### Objective

The project’s purpose was to implement the logic control of a single elevator. The elevator analysed displays two user interfaces (internal, external panel) and two technical interfaces (System Communications and Faults).

The macro-steps our group discussed and implemented could be summarised as:

* How the elevator should manage the different kinds of floor calls; develop a priority hierarchy that is univocal and logically simple, in order to have predictability.
* How the elevator should manage an emergency call; realise an alternative call management policy where emergency handling has the highest priority.
* How the elevator should manage a generic fault; implement a system control that works in the best way possible when lacking a sensor, warning the user when normal functioning is compromised.

We used the “Generalized Actuator” approach, which led to the “GAs” definition first; the “Policy” definition and implementation was then carried abreast with the GAs implementation to face the increasing complexity of control.

#### Generalized Actuators

From the beginning we already knew we would have needed more than one GA. We opted for a two GAs implementation, where each GA is given a specific task to fulfill.

Our idea is based on the following table:

|  |  |  |
| --- | --- | --- |
| Actions | Sensors | Actuators |
| ‘OpenDoors’ | Open | Opening |
| Presence |  |
| ‘CloseDoors’ | Close | Closing |
| Presence |  |
| ‘Nominal’ and ‘Initial’ | LimitDown | Motor\_on |
| LimitUp | Up |
| DeckSensor | Vel |
| Ramp Sensor |  |

Both our GAs are classified as Do/Done GAs, which means that every time a specific action is given to them as input by the Policy, a Done output is sent back once the action is completed.

Our first GA is called *Doors\_GA* and it has to manage doors related tasks. The *‘Presence’* button adds a certain level of complexity to the overall structure. Based on our personal experience of elevators, we decided to implement this function such that each time a presence is detected in between doors, we want them to completely re-open and close them only once this presence disappears. We made the choice not to wait directly inside the GA after the opening/closing actions, but to set waiting times in the Policy.

Another important aspect of this GA is the alternative choice that is not only composed by the two main requests sent by the Policy (*‘CloseDoors’* and *‘OpenDoors’*), but also includes the emergency possibility. Thanks to this choice, everytime there is an emergency and the elevator cab is stopped at a floor, doors will always be opened.

The second GA, called *Motoring\_GA*, gathers together all the remaining sensors and actuators available in the project. This GA controls every aspect of the cabin motion: direction, speed and whether it has to start the motor or not. Basically, we wanted this GA to move the elevator to the correct destination requested by the Policy. The basic idea and implementation behind this GA is fairly simple, but its complexity rises once exceptions (Emergency and, in particular, Faults) need to be handled.

In the *Motoring\_GA* there is immediately a distinction between two cases: *‘Initial’* and *‘Nominal’* functioning. This was necessary due to the fact that during initialization a slightly different procedure is required since we don’t know on which floor the elevator is and it always has to move with low speed.

The destination can often vary, depending on users' calls, even during the motion of the elevator but only if there is enough space to stop safely at low speed. This last situation was a bit challenging for us, but we managed to implement a suitable solution that consists in verifying if the new possible destination is feasible or not, depending on the ramp sensor counter value. This is performed in the GA that later informs the Policy if this particular mid-call is reachable or it is too late to change target in order to stop with a low speed.

#### Policy

The policy definition was performed right after the GAs’ one. Three macro-steps had to be initially considered:

##### Initialization phase

##### Call management control

##### Nominal functioning phase

1. Once the GAs were accomplished, the initialization phase was created as a straightforward sequence of SFC steps. The proper phase consists of three steps: *CloseDoors*, *FirstFloor* and *OpenDoors*, where the two Do/Done GAs are used with no particular control addictions.

*CloseDoors* uses the *Doors\_GA* and is designated for doors closing. In *FirstFloor* the Policy sends an *‘Initial’* request to the *Motoring\_GA*; as soon as the elevator cab has reached the First floor, a *‘Done’* message is sent back to the Policy. Before proceeding the “memory” variables are initialized; these variables are used to store pieces of information that we don’t want to lose every “nominal functioning cycle”.

*OpenDoors* is finally related to doors opening. After this, the “call management” is executed abreast with the “nominal functioning phase”.

1. The call management is divided into two macro-actions in our project:

* Store users’ calls and manage the corresponding LEDs
* Perform the “priority algorithm”

All calls are saved in one block throughout the whole elevator functioning: *GetCalls*. This block is placed in a dedicated branch, parallel to the main branch, so that calls are stored seamlessly. It stores users’ calls by overwriting the designated arrays: if a call button is pressed, the corresponding boolean variable is set to ‘TRUE’ (and changed back when the request is satisfied). Corresponding LEDs are turned ON by overwriting their current value with the one of the corresponding button previously stored.

The “priority algorithm” used is a personal choice, made with our elevator-functioning knowledge and experience. In the destination decision-making three macro-cases are distinguished:

* “Highest priority” case: there are stored calls that do not require a direction change of the elevator cab; among them, the nearest to the current floor is chosen.
* “Medium priority” case: (the case above cannot be performed) there are stored calls that require a future direction change of the elevator cab; among them, the further from the current floor is chosen.
* “Lowest priority” case: (the cases above cannot be performed) all calls require a direction change of the elevator cab; the value of the variable that holds the current direction is changed and the algorithm is cycled again.

This algorithm is written in three different SFC blocks (with few necessary corrections): *WaitForCalls*, *PreCalls* and *MidCalls*. This division, as well as being useful, enhances the readability of this Policy section.

*WaitForCalls* is the first block and so the one that sets the best destination, transitioning to the “nominal functioning branch”. If all calls are satisfied, the algorithm is not performed until any call is made.

*PreCalls* is placed in a branch parallel to the “nominal functioning” one. This block looks for late calls (made before the motor starting) and compares them to the saved destination. If a more suitable call is found (nearer/highest priority), destination is changed.

Lastly, *MidCalls* compares calls made when the elevator is already moving. Changing direction here is no longer possible despite priorities; nearer or highest-priority floors can be considered. However, before changing destination, *Motoring\_GA* has to grant the possibility to stop there with no abrupt deceleration.

1. The nominal functioning phase (carried out entirely on the homonym branch) is similar to the initialization phase, but with waiting times added to simulate a realistic behavior. *CloseDoors2* and *OpenDoors2* are the equivalent blocks of the ones seen above; differences are anyhow present in *Destination*. Here the cab destination has to be sent to *Motoring\_GA*, and eventually updated continuously. Furthermore, it’s *Destination* duty to reset the calls of the floor reached, potentially changing the future direction.

#### Emergency Handling

Emergency handling is the procedure that describes the behaviour of the elevator from when the emergency button is pressed the first time to when it is unpressed. In case of emergency, the cabin has to stop to the nearest safe floor and stay still until resolution. To do so, we firstly implemented an exception in our *Door* GA that prevents doors from closing and opens them if an emergency is present.

The worst case is when an emergency is called while the elevator is trying to reach its destination. In this situation the *Motoring\_GA*, after becoming aware of the emergency, selects the nearest possible floor in which it is possible to stop at and sends it to the Policy. Then, the Policy set the new destination of the cabin to that specific floor and, once arrived, the doors will stay open. We also managed to keep gathering users' calls during the emergency phase, without any anomalous repercussion in storing calls and future paths.

As it will be better explained later, in our project, the emergency button is also exploited, adding a useful feature to its intentional use, in order to reset any occurred faults by pressing it two times.

#### Fault Management

Our purpose, when it comes to fault management, was to realise a logic control capable of working with as few restrictions as possible in case of fault. To be able to achieve this, we decided to gather and store as much datas as possible from sensors.

The first step taken was to create an array to be compiled in *Motoring\_GA* with all the sensors’ readings; boolean variables ‘FALSE’ and ‘TRUE’ are assigned respectively to *Ramps* and *Decks*.

The second step was to create a reading algorithm able to identify a possible fault type and its position; if the sequence changes or shifts, a fault has been detected.

Then a fault management policy was implemented; when a Ramp fault is detected, its position is stored in a specific array in the Policy. When nominal functioning of the elevator is performed again, the array is sent back to *Motoring\_GA*; the Ramp fault is so expected and compensated in the GA’s array overwriting. This correction allows to store a new fault.

When a Deck fault is detected, the position is still sent to the designated array in the Policy, but the related floor becomes, as the simulation design implies, unreachable. While the GA’s array can still be compensated, permitting to reach all other floors flawlessly, we thought users should be warned too, using all the available LEDs.

In an additional branch, parallel to the main one, LEDs related to Deck-faulted floors are switched ON and OFF alternatively (*LEDActive*, *LEDnotActive*) to warn the user about the inability to reach the indicated floor.

Finally, once the Fault is solved, pressing the Emergency button two times is enough to clear the Policy arrays storing faults; we thought this would be an easy procedure for the virtual “technician” to inform of the occurred fixing.

#### Conclusions

We believe that our project is an effective and efficient logic control of an elevator; capable of satisfying all the actions usually performed by a realistic elevator: from managing priorities between users’ calls to moving the elevator in case of faults, without causing irreparable damages. Moreover, the GAs we realized satisfy the requests of readability (SFC easy to understand), parts reusability (encapsulation) and independent faults detection.

With our implementation we are able to manage up to two different kinds of faults simultaneously (if they are not adjacent), our code can be expanded to handle more faults. We decided to use arrays (instead of simpler integer variables) to allow more than one faulty floor to be stored at a time. After much debugging and tests, needed to track down the most insidious issues in our code, we can affirm that we are absolutely satisfied with the level of robustness and reliability reached by our logic control implementation.

*A special thanks goes to our siblings that helped us with the hard debugging work.*