ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

DEPARTMENT OF ELECTRICAL, ELECTRONIC, AND INFORMATION ENGINEERING (DEI)

Project Report

Elevator Control Project

Student:	Abess Ouardi
Course:	Control System Technologies
Professors:	C. Conficoni, M. Ccciari, A. Testa
Date:	January 2025

Bologna, Italy

Contents

1. System Overview and General Architecture

1.1. Introduction

This project focuses on the implementation and simulation of an automated elevator system using the CoDeSys development environment. The goal is to design, test, and validate the complete control logic of a single–cabin elevator governed by a programmable logic controller (PLC). The system is realized entirely in software and operates through two coordinated programs:

1. a control program implementing the decision logic, and
2. a simulation program emulating the physical plant, sensors, and actuators.

This approach enables full testing of the elevator control sequence without physical hardware while maintaining timing, feedback, and signal dependencies close to real operating conditions.

1.2. System Architecture

The architecture reproduces a typical industrial automation structure composed of a high–level policy layer and a low–level actuation layer. The control program, developed using both the Sequential Function Chart (SFC) and Structured Text (ST) languages, manages the logical behavior of the elevator: floor requests, movement sequencing, door handling, and safety reactions. The simulation program, written in ST, emulates the mechanical and electrical dynamics of the real plant, responding to control outputs with suitable delays and feedback signals. This separation allows complete closed–loop validation of the control policy under realistic timing and fault conditions.

1.2.1. Control and Simulation Interaction

At each PLC cycle, the control program outputs commands such as `DO_MoveUp`, `DO_OpenDoor`, or `DO_CloseDoor`. The simulation program interprets these commands, reproducing the corresponding physical action through internal counters, delay elements, and state variables. These quantities simulate realistic mechanical and sensor response times, for instance, the duration of door movement or the travel time between consecutive decks.

All these internal simulation values are not only used for the dynamic behavior of the control loop but are also directly linked to the HMI visualization. The same variables governing cab height, door displacement, and sensor activations drive the graphical components of the interface, ensuring full synchronization between the simulated physics and the visual representation. Thus, when a door is closing, the counter managing its delay simultaneously updates the animated door on the HMI, while the virtual position of the cab and the deck indicators are refreshed according to the simulated motion profile.

This bidirectional coupling establishes a closed loop that replicates the logical and physical behavior of a real PLC-controlled elevator, allowing both realistic visual feedback and accurate validation of control logic.

1.3. Sensors and Actuators

1.3.1. Sensors

The simulated elevator is equipped with a complete set of logical sensors, each reproducing the behavior of physical devices typically installed in real elevator systems. These sensors continuously provide feedback to the controller, allowing it to determine the cab position, regulate velocity profiles, and guarantee operational safety.

- Deck sensors are placed at every floor level and detect when the cabin is perfectly aligned with a deck. Their activation corresponds to the arrival of the cab at the target floor, enabling the transition to door–opening operations.
- Ramp sensors are located immediately *before* and *after* each deck sensor, resulting in two ramp sensors per floor. These sensors serve as intermediate checkpoints used to regulate the cab's velocity: when the elevator starts moving, it accelerates only after leaving the first ramp sensor, and when approaching a target floor, it decelerates upon detecting the ramp sensor before the deck. This ensures smooth motion and precise stopping at the deck level.
- Limit sensors (`LimitUp` and `LimitDown`) are positioned beyond the highest and lowest decks, outside the normal travel range of the cabin. They act as mechanical end–stop sensors and are used exclusively to prevent the elevator from exceeding its allowable travel limits. These sensors are not associated with any real floor and are normally inactive during regular operation.
- Door sensors (`Open`, `Closed`) provide binary feedback indicating the actual door position. A value of `TRUE` (or 1) means the condition is active.For example, `Closed = TRUE` when the doors are fully closed, and `FALSE` (or 0) when the condition is not met.

- Presence sensor monitors the area between the doors. When an obstacle is detected, its signal becomes `TRUE`, preventing any door–closing command until the obstruction is removed.
- Emergency button acts as a safety input that immediately interrupts normal operation and initiates the emergency stop sequence described in the corresponding section.

Each of these sensors follows a Boolean logic pattern (`TRUE`/`FALSE`), reflecting the discrete nature of industrial PLC inputs. Their values are continuously updated by the simulation logic and also used to drive the corresponding HMI indicators, ensuring that visual feedback always mirrors the real–time system state.

Fault–injection switches for the deck and ramp sensors are included within the simulation environment. Although they are not part of the operational control policy, they enable fault–tolerance testing and diagnostic validation under simulated failure conditions.

1.3.2. Actuators

The actuation subsystem represents all the output signals generated by the control logic to drive the simulated components of the elevator. Each actuator translates a logical command from the controller into a physical action within the simulation environment, ensuring realistic interaction between software and plant model.

- *Motor actuation* — The elevator motor is commanded through three main signals: direction (`Up`/`Down`), enable (`Motor_on`), and velocity selection (`Vel`). The combination of these signals determines whether the cab moves upward or downward and whether the movement occurs at normal or low speed. The low–speed mode is automatically engaged during approach and departure phases, allowing precise alignment with the deck sensors.
- *Door actuation* — Two control signals manage the door mechanism: one for opening and one for closing. Each operation follows a predefined time delay, simulated through internal counters to reproduce the physical inertia of the doors. Safety interlocks prevent closing if the presence sensor is active or if a fault is detected in the door actuator.
- *LED indicators and visual actuators* — Several output variables correspond to the visual feedback elements on the HMI. They include confirmation LEDs for internal and external calls, indicators of movement direction, and an emergency status light. These signals are updated in real time according to the control logic and the simulated plant state, ensuring consistent visual representation.

All actuators follow a unified control structure: when the policy issues a command, the simulation program reproduces the corresponding mechanical behavior through state transitions and internal timers. This abstraction allows each actuator—motor, doors, or lights—to be managed as a generalized block, simplifying coordination and improving modularity in the overall control design.

1.4. Emergency and Fault Management

The elevator is equipped with a single **emergency button**. When pressed, the controller immediately halts motion and commands the cabin to reach the nearest accessible floor. Once at rest, the doors open automatically to allow evacuation. The system remains in a safe, disabled state until the emergency signal is released.

Faults such as missing ramp or deck signals can be simulated to verify the reaction of the controller. These faults are handled by the simulation layer, allowing the validation of error–handling routines and the verification of diagnostic feedbacks to the operator interface.

1.5. Human–Machine Interface and Simulation Environment

The graphical HMI provided with the CoDeSys environment acts as an intelligent simulator that visualizes, in real time, the state of the entire system. The interface integrates both control and simulation aspects:

- Position bar: displays the current position of the elevator cabin, dynamically filling upward or downward during motion.
- External call indicators: show active calls and the requested travel direction.
- Internal panel: dynamically represents pressed floor buttons and door state.
- Door visualization: opens and closes in synchrony with the actual door status signals.
- Fault switches: allow the user to inject simulated sensor malfunctions.
- System communication panel: displays real–time variables such as height, presence, limits, and emergency signals.

The simulation environment does not merely visualize variables—it emulates the time behavior of the mechanical components. For instance, a door–closing command initiates a five–second timer; if no fault is detected during this interval, the corresponding “door closed” sensor becomes active. In this way, the complete dynamic response of the system is reproduced without physical hardware.

1.6. Generalized Actuators Methodology

The entire control strategy has been developed according to the Generalized Actuator (GA) methodology. This approach provides a structured framework for decomposing the automation logic into two clearly separated layers:

- a high–level policy managing the overall system sequence;
- a collection of Generalized Actuators implementing individual mechanical actions.

1.6.1. Principle of Operation

The Generalized Actuator (GA) approach defines a structured and modular method for designing logic controllers in industrial automation. Rather than mixing the decision logic (*what to do*) with the execution logic (*how to do it*), this methodology clearly separates the two. Each GA is responsible for executing a family of low–level actions, while the higher–level policy coordinates these actions according to the desired operating sequence.

Conceptual Structure

A Generalized Actuator is a self–contained software entity that encapsulates the sensors, actuators, and control logic required to perform one or more related actions. In the elevator project, this concept was implemented by defining three GA instances, each dedicated to a specific family of actions:

- door opening;
- door closing.
- cabin motion (upward or downward displacement);

Each GA instance is implemented as an Instance in CoDeSys, continuously active throughout the system operation. Even when not executing a specific task, the GA monitors its associated sensors and actuators, providing real–time feedback and fault detection capabilities. This continuous activity allows low–level diagnostics to be handled directly inside the GA, without affecting the policy logic.

Standard Interface and Communication

The communication between the policy and each GA follows a standardized interface based on the `DO-DONE` convention. This ensures that every actuator block behaves consistently, regardless of its internal structure or the specific action performed.

- `DO` is a command input used to start a specific action (for example, `DO = TRUE`, `DO_WHAT = CLOSE_DOOR`);
- the GA executes the required sequence of operations, typically described in SFC form with ST code blocks in each state;

- once the action is completed, the GA raises the `DONE` flag and provides feedback on its `STATE`;
- if any anomaly occurs, the GA sets the `FAULTS` signal to `TRUE`.

The `STATE` variable can assume three main values:

- `READY` — the GA is idle and ready to accept a new command;
- `BUSY` — the GA is executing a commanded action;
- `FAULTS` — an error condition was detected during execution.

This standard interface simplifies synchronization between the control policy and the mechanism layer. For example, the policy can issue a "close door" command by setting `DO := TRUE` and monitoring `DONE`. Once the GA reports completion (`DONE = TRUE`), the policy resets the command and proceeds to the next state.

Internal Logic and Modularity

Internally, each GA is structured as an SFC (Sequential Function Chart), representing the finite–state automaton that executes and supervises the requested action. Typical macro–states include:

- *Initialization* — sets the internal variables and ensures a consistent starting condition;
- *Ready* — waiting for a valid command from the policy;
- *Busy* — executing the commanded action, controlling actuators, and monitoring sensor feedback;
- *Fault* — handling abnormal situations or timeout errors.

Each SFC step can contain Structured Text (ST) code that performs logical checks, timing functions, and variable assignments. This hierarchical and encapsulated design ensures that changes in sensor behavior or timing do not affect the overall system coordination.

Advantages of the GA Approach

The main advantages of this methodology are:

- *Modularity* — each GA acts as an independent software module that can be reused or replaced without altering the control policy;
- *Readability* — the separation between "mechanisms" (executing actions) and "policy" (deciding actions) improves clarity and maintainability;

- *Reusability* — the same GA can be applied to different projects or replicated across similar systems;
- *Diagnostics* — continuous sensor monitoring and built–in fault signaling (`FAULTS`) allow immediate fault detection at the component level.

Application to the Elevator System

In this project, three GA instances were created and used across all operational modes. Whenever an action was needed—such as closing the doors, moving the cabin, or opening the doors—the control policy invoked the appropriate GA instance, providing the action identifier via the `DO_WHAT` variable. The GA autonomously executed the action sequence, verified its success, and notified completion through the `DONE` flag. If a fault occurred (for example, a sensor not responding within the allowed delay), the `FAULTS` flag was raised, and the policy immediately triggered a recovery or diagnostic routine.

2. Initialization Procedure

2.1. Why an Initialization is Necessary

When the system is powered up after installation or maintenance, the elevator car is at an *unknown* position with respect to the building floors and the door state may not be coherent with the internal variables. Before the standard operating cycle can start, the controller must:

1. re–align software states with the physical (simulated) plant;
2. determine a known reference floor and direction limits;
3. place the doors in a safe, verified condition.

This *one–time* procedure runs only at first commissioning or after service interventions; once completed, the system switches to the normal call–handling policy.

The initialization implements the following ordered steps (all at low risk and low speed):

1. Close doors while monitoring the presence sensor (safety interlock).
2. Descend slowly until the `LimitDown` sensor activates (lower mechanical limit).
3. Ascend slowly to the `Deck` of the first floor and latch “current floor = 1”.
4. Open doors to establish a safe initial condition.
5. Wait for calls and hand over to the nominal control policy.

The controller knows a priori the number of levels; only the current deck must be discovered.

2.2. SFC Structure and Entry State

The Sequential Function Chart (SFC) begins with the `Init` state, which serves as a dedicated setup phase for initializing all key data structures and preparing the system for deterministic execution. This step is essential to ensure that the control logic, timers, and Generalized

Actuators (GAs) start from a known and consistent condition, preventing undefined behavior during the first execution cycles.

The initialization routine implemented in Structured Text (ST) is reported below:

```
FOR i := 0 TO 8 DO
    GVL.SEQUENCE[i] := -1;
END_FOR

GVL.INITIALIZATION_VAR := TRUE;

EMERGENCY_STOP := 8;
```

Listing 2.1: Initialization procedure in the `Init` state.

Here, `GVL` stands for Global Variable List, a centralized structure in CoDeSys used to store all system-wide variables that must be accessible from multiple programs and function blocks. It provides a shared memory context, allowing seamless communication among the main control logic, the Generalized Actuator instances, and the simulation model.

The array `GVL.SEQUENCE` acts as the memory of the elevator controller, storing the list of pending floor requests. Each element represents a target floor index; a value of `-1` indicates an empty slot. At startup, all positions are cleared (set to `-1`) to ensure that no residual calls from previous executions persist. This array is later managed dynamically by the logic program, which inserts new calls, reorders them according to direction (ascending or descending), and removes completed requests.

The variable `GVL.INITIALIZATION_VAR` is set to `TRUE` to flag the system as being in the initialization phase, while the constant `EMERGENCY_STOP` is initialized with a default value of 8, corresponding to the code used to identify an emergency condition in the call–management logic.

After all parameters are sanitized and initialized, a *TRUE transition* (always active) dispatches control to the next operative state in the sequence, beginning the physical initialization procedure (door closure and cabin positioning).

2.3. Step 1 — Closing of the Doors via `CLOSE_GA`

The first operative phase of the initialization procedure concerns the closure of the elevator doors. This action must be performed before any cabin movement to ensure that no passengers are exposed to mechanical risks and that all sensors are properly aligned for subsequent motion commands.

The operation is handled by the `CLOSE_GA` instance of the Generalized Actuator family. Rather than acting directly on outputs, the Sequential Function Chart simply requests the door–closing service by setting the actuator's command interface. Once called, the GA autonomously manages timing, supervision of the door sensors, and presence verification. The relevant Structured Text logic executed within the `CLOSING_INIT` state is:

```
IF GVL.CLOSE_GA_1.STATE = STATE_GA.S_READY THEN
    GVL.CLOSE_GA_1.DO_ := TRUE;
    GVL.CLOSE_GA_1.DO_WHAT := BASIC_ACTIONS.CLOSE_DOOR;
END_IF

IF GVL.CLOSE_GA_1.STATE = STATE_GA.S_BUSY AND GVL.CLOSE_GA_1.DONE
   THEN
    GVL.CLOSE_GA_1.DO_ := FALSE;
    SYNCHRONIZATION := TRUE;
END_IF

OPERATION_OK := SYNCHRONIZATION AND NOT GVL.CLOSE_GA_1.DONE;
```

Listing 2.2: Structured Text code for the door–closing phase.

As soon as the GA reports that it is ready, the control logic issues a command to perform the closing sequence. While the actuator executes, it internally supervises the door position and presence sensors. When the movement is complete, it sets the `DONE` flag, signalling to the main chart that the operation has successfully concluded. The state then disables the command and raises the local synchronization flag, confirming completion.

An exit action is executed when leaving the `CLOSING_INIT` state to restore the initial condition of the variables and to prevent residual activation signals from persisting across transitions:

```
OPERATION_OK := FALSE;
SYNCHRONIZATION := FALSE;
GVL.CLOSE_GA_1.DO_ := FALSE;
```

Listing 2.3: Exit action from the door–closing state.

Internally, the actuator follows its own finite–state logic, progressing through the phases of *ready*, *closing*, and *finalization*. When the closed–door sensor becomes active within the allowed time window and no obstacle is detected, the GA asserts `DONE = TRUE` and `FAULT = FALSE`, allowing the SFC to proceed toward the next step, `DOWN_INIT`. If instead the closure is obstructed or the maximum closing time is exceeded, the actuator raises its `FAULT` flag while keeping `DONE = TRUE`. In that case, the control sequence branches to the `OPENING_INIT` state to re-open the doors and ensure a safe condition before retrying the procedure.

Through this mechanism, the initialization always starts from a verified and consistent door state. The SFC remains focused on the progression of phases, while the actuator handles the low-level coordination, timing, and fault supervision internally.

2.4. Step 2 — Descent to Lower Limit via MOVING_GA

Once the doors are fully closed and the system is in a safe state, the initialization procedure continues with the controlled descent of the elevator cabin toward the lower mechanical limit. This step establishes a physical reference position from which the controller can synchronize its internal position counters, ensuring that all subsequent movements are based on a verified zero level.

The vertical motion is executed by the `MOVING_GA` instance, a Generalized Actuator dedicated to motor control and vertical positioning. Through a simple `DO-DONE` interaction, the SFC requests a downward movement while the actuator autonomously handles all timing, sensor supervision, and safety interlocks. During initialization, the movement is deliberately carried out at slow speed to guarantee accuracy and avoid shocks at the lower end of the hoistway.

The Structured Text code embedded in the `DOWN_INIT` state is the following:

```
IF GVL.MOVING_GA_1.STATE = STATE_GA.S_READY THEN
    GVL.MOVING_GA_1.DO_ := TRUE;
    GVL.MOVING_GA_1.DO_WHAT := BASIC_ACTIONS.MOTOR_DOWN_SLOW;
END_IF

IF GVL.MOVING_GA_1.STATE = STATE_GA.S_BUSY AND GVL.MOVING_GA_1.DONE
   THEN
    GVL.MOVING_GA_1.DO_ := FALSE;
    SYNCHRONIZATION := TRUE;
END_IF

OPERATION_OK := SYNCHRONIZATION AND NOT GVL.MOVING_GA_1.DONE;
```

Listing 2.4: Structured Text code executed during the downward initialization.

When the actuator reports readiness, the command is issued to initiate a slow downward movement. From this point onward, `MOVING_GA` takes control of the motor drive and supervises the relevant sensors, particularly the ramp and deck detectors as well as the `LIMIT_DOWN` switch. The actuator continuously updates the internal counters associated with decks and ramps, allowing the controller to maintain a precise record of its current position throughout the descent.

Internally, the GA regulates speed transitions and ensures a progressive deceleration as the cabin approaches the terminal limit. If the lower limit sensor is reached, the actuator automatically stops the motor, resets the motion command, and signals completion by setting `DONE = TRUE`. This event triggers the SFC transition to the subsequent phase, `UP_INIT`, marking the end of the downward calibration procedure.

By the end of this step, the elevator is positioned at its mechanical reference point, all counters are synchronized, and the system is ready to begin its upward alignment toward the first floor at the same controlled velocity.

2.5. Step 3 — Ascent to the First Floor via `MOVING_GA`

After reaching the lower mechanical limit, the initialization sequence continues with the ascent toward the first floor. This motion finalizes the calibration process, ensuring that the cabin begins normal operation from a precise and verified position.

The movement is again performed by the `MOVING_GA` actuator, now instructed to drive the elevator upward at low speed. Unlike standard operation, where the actuator may alternate between fast and slow velocities, the initialization procedure enforces exclusively slow motion to maximize positioning accuracy and mechanical safety.

```
IF GVL.MOVING_GA_1.STATE = STATE_GA.S_READY THEN
    GVL.MOVING_GA_1.DO_ := TRUE;
    GVL.MOVING_GA_1.DO_WHAT := BASIC_ACTIONS.MOTOR_UP_SLOW;
END_IF

IF GVL.MOVING_GA_1.STATE = STATE_GA.S_BUSY AND GVL.MOVING_GA_1.DONE
   THEN
    GVL.MOVING_GA_1.DO_ := FALSE;
    SYNCHRONIZATION := TRUE;
END_IF

OPERATION_OK := SYNCHRONIZATION AND NOT GVL.MOVING_GA_1.DONE;
```

Listing 2.5: Structured Text code for the upward initialization phase.

The actuator autonomously supervises the sensors during the ascent, updating the internal position counters until the deck sensor corresponding to the first floor is detected. Upon reaching this point, the motor is stopped automatically, and the `DONE` flag is raised. The SFC recognizes this condition as the successful completion of the upward calibration and resets the synchronization variables before transitioning to the next phase.

At this stage, the elevator is correctly aligned with the first floor and fully synchronized with the reference sensors. The system can now proceed to the final initialization step—opening the doors and entering standby mode for normal operation.

2.6. Step 4 — Opening of the Doors via `OPEN_GA`

The last action of the initialization sequence brings the elevator, now positioned at the first floor, into a safe and accessible condition by opening the doors. This task is handled by the third instance of the Generalized Actuator family, `OPEN_GA`, which manages all phases of the door–opening motion using the same `DO-DONE` convention adopted in previous steps. By invoking this actuator, the main logic delegates the low–level handling of timing, sensor supervision, and safety interlocks, ensuring a reliable and standardized behavior.

2.6.1. Door–Opening Control Logic

When the SFC enters the OPENING_INIT state, the controller activates the actuator through a simple command handshake. The relevant Structured Text code is:

```
IF GVL.OPEN_GA_1.STATE = STATE_GA.S_READY THEN
    GVL.OPEN_GA_1.DO_ := TRUE;
    GVL.OPEN_GA_1.DO_WHAT := BASIC_ACTIONS.OPEN_DOOR;
END_IF

IF GVL.OPEN_GA_1.STATE = STATE_GA.S_BUSY AND GVL.OPEN_GA_1.DONE THEN
    GVL.OPEN_GA_1.DO_ := FALSE;
    SYNCHRONIZATION := TRUE;
END_IF

OPERATION_OK := SYNCHRONIZATION AND NOT GVL.OPEN_GA_1.DONE;
```

Listing 2.6: Structured Text code executed in state OPENING_INIT

Once the actuator receives the command, it takes control of the door mechanism, managing the motion and verifying the state of the door–open sensor. The process is fully autonomous: when the door reaches its open position, the actuator sets DONE = TRUE, signaling completion to the SFC. If any anomaly is detected—such as a timeout or conflicting signal—the actuator would raise its internal FAULT flag while maintaining the same standardized output interface.

2.6.2. Transition to the WAIT State

When the door–opening action completes successfully, the initialization process concludes and the controller transitions into the WAIT state. This marks the moment when the elevator becomes fully operational: the cabin is correctly aligned with the first floor, the doors are open, and the system is ready to accept user commands.

The WAIT state represents both the end of the initialization phase and the fundamental idle condition of the elevator. In this state, the cabin remains stationary with doors open, and the logic continuously monitors the memory array containing pending calls. If a new or stored request is detected, the SFC triggers the appropriate sequence to serve it; otherwise, the elevator remains in a low–power standby mode, maintaining readiness for immediate activation.

2.6.3. Exit Action toward WAIT

Before entering the WAIT state, a short exit routine is executed to clear synchronization flags and restore base variables. This ensures that all counters and internal indicators are properly reset before the system switches to normal operation:

```
OPERATION_OK := FALSE;
```

```
SYNCHRONIZATION := FALSE;
GVL.Current := 0;
COUNT_RAMP := 0;
COUNT_DECK := 0;

GVL.INITIALIZATION_VAR := FALSE;
```

Listing 2.7: Exit action executed when leaving `OPENING_INIT`

This routine eliminates residual control data from the initialization process, restoring the reference conditions required for standard functioning. From this point onward, the elevator operates under its normal policy, cyclically returning to the `WAIT` state whenever no pending calls are present.

3. Normal Operation

During standard operation, the elevator follows a structured policy designed to manage calls efficiently and safely. This logic is continuously executed by a dedicated SFC step named `LOGIC`, which remains active at all times, even during initialization, movement, or door operations. By running cyclically, the `LOGIC` step ensures that new calls can be received, validated, and stored in real time, without waiting for the completion of other procedures.

3.1. Service Policy

The service policy governs how the elevator accepts, organizes, and executes user calls depending on its operational state. Each request can originate from either the internal cabin panel (buttons G–7) or from the external hall buttons (UP/DOWN). All accepted calls are stored in a fixed-size array `SEQUENCE[0..8]`, which serves as the scheduling memory of the system. The first position, `SEQUENCE[0]`, always holds the active destination, while the subsequent entries store pending calls. Empty slots are represented by the value `-1`. When a destination is reached, it is removed, and the array elements are shifted forward, ensuring that the next valid call becomes the current target.

The policy distinguishes between two main operating modes:

- Static mode: the elevator is stationary, with doors either open or in transition.
- Dynamic mode: the elevator is moving upward or downward, at fast or slow speed.

3.1.1. Static Mode Behavior

When a call is made while the elevator is static, the controller first checks whether the array is empty:

- Array empty: the new call is automatically accepted and becomes the next destination. The system determines its travel direction (`GOING_UP`) by comparing the current floor with the call's target.
- Array not empty: the call is accepted only if it follows the current direction of travel. Internal

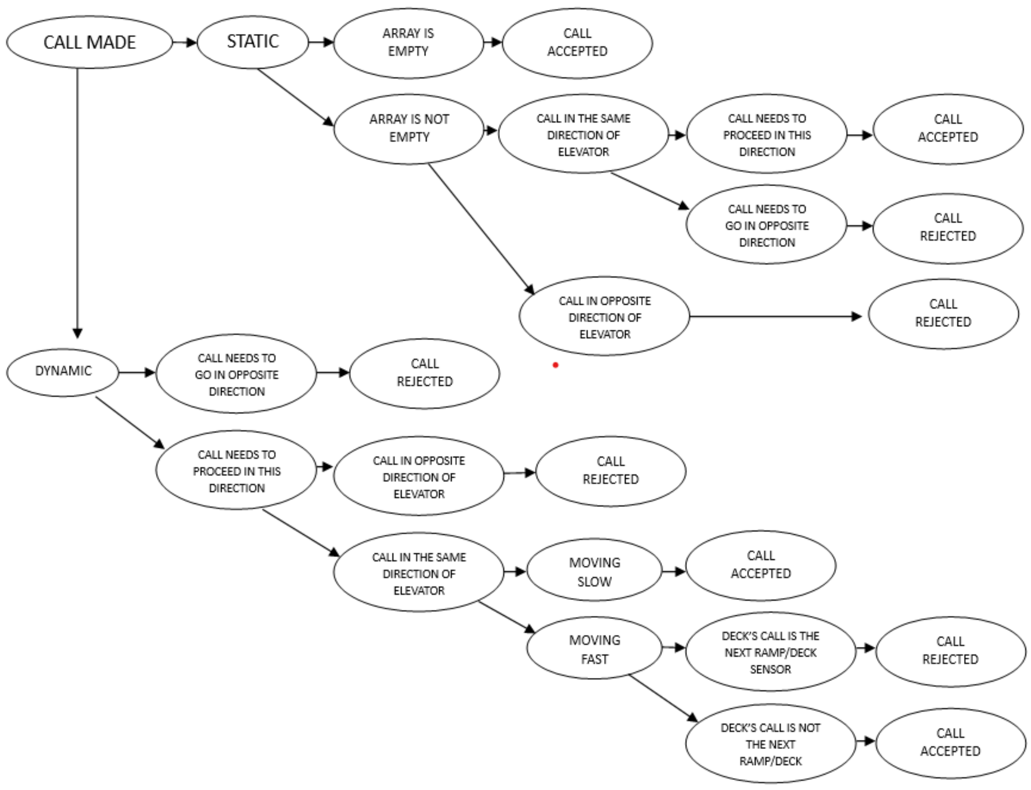
CALL MADE
STATIC
ARRAY IS EMPTY
CALL ACCEPTED
ARRAY IS NOT EMPTY
CALL IN THE SAME DIRECTION OF ELEVATOR
CALL NEEDS TO PROCEED IN THIS DIRECTION
CALL ACCEPTED
CALL NEEDS TO GO IN OPPOSITE DIRECTION
CALL REJECTED
CALL IN OPPOSITE DIRECTION OF ELEVATOR
CALL REJECTED
DYNAMIC
CALL NEEDS TO GO IN OPPOSITE DIRECTION
CALL REJECTED
CALL NEEDS TO PROCEED IN THIS DIRECTION
CALL IN OPPOSITE DIRECTION OF ELEVATOR
CALL REJECTED
CALL IN THE SAME DIRECTION OF ELEVATOR
MOVING SLOW
CALL ACCEPTED
MOVING FAST
DECK'S CALL IS THE NEXT RAMP/DECK SENSOR
CALL REJECTED
DECK'S CALL IS NOT THE NEXT RAMP/DECK
CALL ACCEPTED

Figure 3.1: Flowchart representing the elevator's call acceptance logic.

calls (from inside the cabin) are always accepted but inserted in order. Calls in the opposite direction are rejected to avoid unnecessary direction changes.

3.1.2. Dynamic Mode Behavior

When the elevator is in motion, calls are evaluated with respect to both direction and speed:

- Opposite direction: the call is rejected and deferred until the elevator stops.
- Same direction: the controller checks the current velocity and position:
- At slow speed, the call is accepted normally.
- At fast speed, the call is accepted only if it does not correspond to the next ramp or deck sensor. This prevents unsafe stops too close to the elevator's current position.

3.1.3. Ordering and Memory Management

Accepted calls are automatically sorted:
- in ascending order when traveling upward,
- in descending order when traveling downward.

This guarantees that the elevator serves calls efficiently along its current direction of motion. After each stop, the served deck is removed from the array, and all entries are shifted upward to maintain continuity. If no more destinations remain, the system returns to the `WAIT` state, ready to accept new requests.//

Overall, the service policy (Fig. 3.1) ensures that:

- calls in the same direction of travel are prioritized;
- direction changes occur only when all calls in the current direction have been served;
- new requests are dynamically filtered based on motion speed and position;
- the system behaves predictably and safely under both static and dynamic conditions.

This structured yet flexible strategy replicates the behavior of real multi-call elevator controllers, combining safety, responsiveness, and efficient scheduling.

3.2. Implementation in Structured Text

The `LOGIC` step is permanently active in the background. Its role is to manage the memory array `SEQUENCE`, handle user inputs, and decide whether to insert, reject, or reorder new calls. This design ensures that the system remains reactive at all times — for example, it can register new requests while the elevator is moving or while door actuators are operating.

3.2.1. Initialization and Destination Handling

At each PLC cycle, internal flags are reset:

```
TRAP := FALSE;
INTERN_CALL := FALSE;
THERE := FALSE;
```

For simulation purposes, the destination variable mirrors the head of the sequence:

```
GVL.Destination := GVL.SEQUENCE[0];
```

When the cab reaches the current destination (`GVL.Current = GVL.SEQUENCE[0]`), and is in one of the waiting or opening states, the sequence is updated as follows:

- The reached floor is cleared (`SEQUENCE[0] := -1`);
- The array is compacted, pushing all `-1` entries to the end.

3.2.2. Call Detection and Encoding

A single integer, `POSSIBLE_CALL`, encodes the currently detected request:

- Values 0-7 correspond to floors G–7 (from both internal and external buttons);
- Special identifiers are used for emergency stops or safety-related actions;
- Value 8 indicates that no call was detected in the current cycle.

If no call is detected (POSSIBLE_CALL = 8), the logic skips further computation to optimize processing time.

Before adding a new call:

- The system scans the sequence to verify if the deck already exists, setting THERE := TRUE if found;
- The caller's intention is inferred from the pressed button:
  - CALL_HEADING_UP := TRUE for upward hall calls;
  - CALL_HEADING_UP := FALSE for downward hall calls.
- For internal cabin calls, INTERN_CALL := TRUE.
- The relative position of the cab and the call determines NEED_GO_UP.

If the array is empty (TRAP = FALSE), the elevator immediately assigns:

```
GOING_UP := NEED_GO_UP;
GVL.SEQUENCE[0] := POSSIBLE_CALL;
```

When a sequence is already active (TRAP = TRUE), the elevator evaluates whether the new call can be integrated into the current motion queue:

- If the elevator is heading upward:
  - The call is accepted if it corresponds to an upward hall request or an internal call;
  - While moving upward fast, the next ramp call is ignored to avoid redundant stops.
  - Accepted calls are appended to the end of the array and the sequence is re-sorted in ascending order.
- If the elevator is heading downward:
  - The call is accepted if it is a downward hall request or an internal call;
  - While moving downward fast, the next ramp call is ignored;
  - Accepted calls are appended and the sequence is sorted in descending order.

3.2.3. Determinism and Performance

The array has a fixed and small size (`9 elements`), allowing deterministic performance even with multiple calls. Insertion sorting and packing operations are executed each PLC cycle, guaranteeing that calls can be managed concurrently with motion and actuator tasks.

The continuous execution of `LOGIC` thus ensures that the system:

- Remains responsive to new user inputs;
- Maintains the correct service direction and queue order;
- Dynamically updates destinations without interfering with door or motion subsystems.

3.3. Actuation of the Control Policy

Once the call management logic (`LOGIC`) determines the current destination stored in `SEQUENCE[0]`, the system executes the corresponding motion and door operations according to the state chart illustrated in Fig. 3.2. This part of the program translates the high-level service policy into concrete actions by coordinating the generalized actuators (`CLOSE_GA`, `MOVING_GA`, `OPEN_GA`).

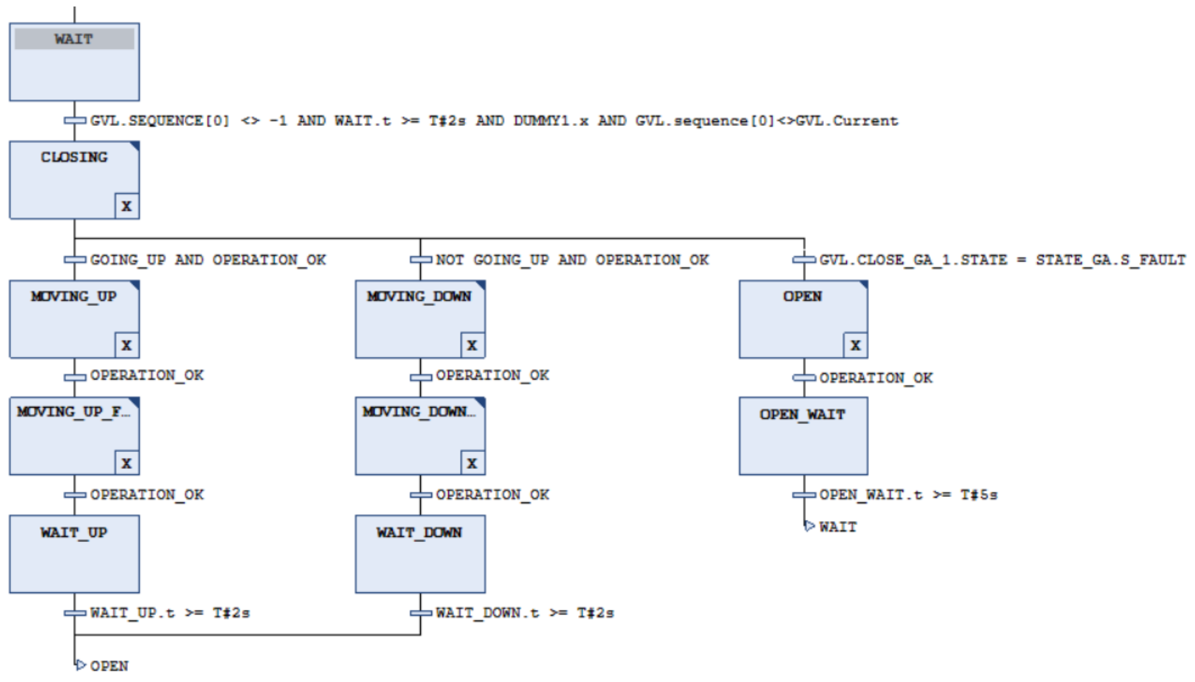
WAIT
GVL.SEQUENCE[0] <> -1 AND WAIT.t >= T#2s AND DUMMY1.x AND GVL.sequence[0]<>GVL.Current
CLOSING
X
GOING_UP AND OPERATION_OK
NOT GOING_UP AND OPERATION_OK
GVL.CLOSE_GA_1.STATE = STATE_GA.S_FAULT
MOVING_UP
MOVING_DOWN
OPEN
OPERATION_OK
MOVING_UP_F...
MOVING_DOWN...
OPEN_WAIT
OPEN_WAIT.t >= T#5s
WAIT
WAIT_UP
WAIT_DOWN
WAIT_UP.t >= T#2s
WAIT_DOWN.t >= T#2s
OPEN

Figure 3.2: State chart of the normal operation logic implementing the service policy.

3.3.1. Normal Operation Sequence

After the initialization phase, the elevator enters its regular operating cycle, alternating between idle periods and motion phases. The system remains in the `WAIT` state whenever there are no active calls or after completing a service. A transition from WAIT to motion occurs only when a specific set of conditions is satisfied:

- At least one valid destination exists in the memory array (GVL.SEQUENCE[0] <> -1);
- The minimum waiting time has elapsed (WAIT.t >= T#2s), ensuring a pause between consecutive actions;
- The elevator's current deck differs from the destination (GVL.Current <> GVL.SEQUENCE[0]);
- The system is in state DUMMY1, meaning that the emergency button is not pressed.

When all these requirements are met, the controller commands the CLOSE_GA actuator to close the doors, ensuring that every movement begins under verified safe conditions. During this process, the presence sensor is continuously monitored. If an obstacle is detected or an emergency signal is triggered, the closing sequence is interrupted, and the system immediately transitions to the OPEN state. The OPEN_GA actuator reopens the doors to restore safety, after which the elevator returns to the WAIT state to reassess pending calls once normal conditions are restored.

If the doors close successfully without obstruction, the motion phase begins. The direction of travel is defined by the service policy computed in the LOGIC block, based on the position of the current deck relative to the target in the SEQUENCE array. The displacement is handled by the MOVING_GA actuator, which autonomously controls motor activation, speed, and synchronization flags.

Two possible motion branches exist:

- **Upward motion:** the elevator travels upward through the states MOVING_UP and MOVING_UP_FAST, switching to slow speed as it approaches the destination deck to ensure smooth stopping.
- **Downward motion:** the system follows the states MOVING_DOWN and MOVING_DOWN_FAST with an identical speed management strategy, but in the opposite direction.

In both cases, the synchronization flag OPERATION_OK guarantees that every actuator completes its assigned task before the next one begins, maintaining a consistent execution flow.

When the elevator reaches the target deck, the MOVING_GA actuator sets its DONE flag, signaling completion of the travel phase. The system then transitions to the OPEN state, where the OPEN_GA actuator commands the doors to open. Once the doors are fully opened, a waiting period of four seconds (OPEN_WAIT.t >= T#4s) is enforced to allow passengers to exit and enter safely.

After this interval, the elevator returns to the WAIT state. If additional calls remain in the sequence array, a new cycle begins; otherwise, the system remains idle until a new request is made. This cyclic structure—WAIT → CLOSING → MOVING → OPENING → WAIT—represents the normal operational behavior of the elevator, combining efficiency, safety, and seamless synchronization between control logic and actuator management.

4. Emergency Management

4.1. Overview

The elevator control system integrates a dedicated emergency management routine to guarantee passenger safety under any operating condition. When the emergency button is pressed, the elevator immediately interrupts its current operation and stops at the first available deck, allowing passengers to exit safely. If the elevator is already stationary, it remains in place until the emergency condition is cleared. During this phase, normal motion commands are inhibited, but the internal call management logic continues to run, maintaining full awareness of active or newly registered requests.

The system remains in the emergency state until the button is pressed again, signaling that normal operation can resume. This design ensures both safe interruption of ongoing actions and smooth recovery of the control process once the emergency is deactivated, without loss of queued call information or synchronization between subsystems.

4.2. Concurrent Task Structure

As soon as the program is launched, the following parallel tasks are activated via simultaneous divergence:

- Initialization and normal operation;
- Logic management for user calls and sequence definition;
- Emergency management routine;
- Rising/falling edge detection;
- LED management and fault signaling.

This concurrent structure guarantees that even during an emergency, the elevator continues monitoring calls, inputs, and LED indicators, thus satisfying the requirement for continuous user interface functionality.

4.3. Emergency State Handling

The emergency routine is managed by a set of dedicated states, active from the very start of system operation. Immediately after initialization, the system enters the `DUMMY1` state, which remains active as long as the emergency button is not pressed. This ensures that the emergency routine is always ready — even during the initialization phase — and can intervene at any moment.

When the emergency button is pressed, the system evaluates the current motion condition and transitions to the corresponding emergency management state. The logic is as follows:

- **If the elevator is static:** The system transitions to the `DUMMY2` state, in which no code is executed. This state effectively freezes all motion and disables actuation until the emergency button is pressed again.
- **If the elevator is in motion:** Depending on the motion direction and velocity, the system transitions to one of the following states:
  - `MOVING_UP_EM` or `MOVING_UP_FAST_EM`;
  - `MOVING_DOWN_EM` or `MOVING_DOWN_FAST_EM`.

When in motion and the emergency button is pressed, the elevator computes the closest safe stopping deck by modifying the call sequence array (`SEQUENCE`). The update procedure depends on the elevator's current speed and position memory.

4.3.1. Fast Upward Motion

If the elevator is moving upward at fast speed and the emergency is triggered, the following Structured Text code is executed:

```
IF EM_HELP_3 THEN
    (* Checking the position of elevator to stop safely *)
    IF (GVL.POSITION_MEMORY[0]=0 AND GVL.POSITION_MEMORY[1]=1)
       OR (GVL.POSITION_MEMORY[0]=1 AND GVL.POSITION_MEMORY[1]=0)
          THEN
        GVL.sequence[8] := INT_TO_SINT(GVL.Current) + 1;
    ELSIF (GVL.POSITION_MEMORY[0]=1 AND GVL.POSITION_MEMORY[1]=1)
       THEN
        GVL.sequence[8] := INT_TO_SINT(GVL.Current) + 2;
    END_IF

    (* Compact and sort the sequence *)
    N := 8;
    FOR J:=0 TO N DO
```

```
        FOR I:=0 TO J-1 DO
            IF GVL.SEQUENCE[I] = -1 THEN
                GVL.SEQUENCE[I] := GVL.SEQUENCE[N];
                GVL.SEQUENCE[N] := -1;
                N:= N-1;
                J:=0;
            END_IF
        END_FOR
    END_FOR

    FOR I:=0 TO 8 DO
        FOR J:=0 TO I DO
            IF GVL.SEQUENCE[I] < GVL.SEQUENCE[J]
               AND GVL.SEQUENCE[I] <> -1 THEN
                SWITCH_HELP := GVL.SEQUENCE[I];
                GVL.SEQUENCE[I] := GVL.SEQUENCE[J];
                GVL.SEQUENCE[J] := SWITCH_HELP;
            END_IF
        END_FOR
    END_FOR
    EM_HELP_3 := FALSE;
END_IF
```

In this configuration, the elevator evaluates the last two ramp sensors (`GVL.POSITION_MEMORY`) to determine the next possible stopping point. If the elevator is approaching a deck sensor directly, the target becomes the next deck (`Current + 1`); otherwise, it stops two decks ahead (`Current + 2`). The call sequence array is then compacted and sorted to make this target the only valid destination.

4.3.2. Slow Upward Motion

If the elevator is moving upward slowly, the following logic applies:

```
IF EM_HELP_3 THEN
    GVL.SEQUENCE[8] := INT_TO_SINT(GVL.Current) + 1;

    (* Compact and sort the sequence *)
    N := 8;
    FOR J:=0 TO N DO
        FOR I:=0 TO J-1 DO
            IF GVL.SEQUENCE[I] = -1 THEN
                GVL.SEQUENCE[I] := GVL.SEQUENCE[N];
                GVL.SEQUENCE[N] := -1;
                N:= N-1;
                J:=0;
            END_IF
        END_FOR
    END_FOR

    FOR I:=0 TO 8 DO
        FOR J:=0 TO I DO
            IF GVL.SEQUENCE[I] < GVL.SEQUENCE[J]
               AND GVL.SEQUENCE[I] <> -1 THEN
                SWITCH_HELP := GVL.SEQUENCE[I];
                GVL.SEQUENCE[I] := GVL.SEQUENCE[J];
                GVL.SEQUENCE[J] := SWITCH_HELP;
            END_IF
        END_FOR
    END_FOR
    EM_HELP_3 := FALSE;
END_IF
```

Here, the emergency stop deck is simply the next one above the current position (`Current + 1`), ensuring the elevator stops as soon as physically possible.

Downward Motion

For downward motion (fast or slow), a symmetric logic applies, using the same approach but with decremented deck indices (`Current - 1` or `Current - 2`) based on the sensor configuration and motion speed.

4.4. Static Emergency Condition

If the elevator is stationary when the emergency button is pressed, the controller enters the `DUMMY2` state. In this condition, no movement or actuation occurs; the system remains idle and waits for the emergency button to be pressed again. This ensures that the elevator does not attempt to perform any action while the emergency mode is active.

4.5. System Recovery and Exit from Emergency

Once the elevator has stopped safely at the computed floor, the system behaves as during a normal arrival: doors are opened through the `OPEN_GA` actuator, and passengers may exit. The control remains in a locked condition, continuously checking the emergency input.

The elevator resumes normal operation only when the emergency button is pressed a second time. At that point:

- The `DUMMY1` state is re-entered;
- Normal initialization and motion logic are restored;
- The standard policy for call management and motion actuation resumes.

During the entire emergency phase, the user call manager remains active but all incoming calls are discarded — each one results in an automatic substitution by `-1` within the sequence array, ensuring that no new destination interferes with the emergency stop procedure.

5. Human–Machine Interface (HMI)

5.1. Overview

The Human–Machine Interface (HMI) represents the graphical and functional layer through which the user interacts with the elevator system. It serves both as a control and diagnostic panel, emulating the real physical components of the installation. The developed interface allows full visualization of the elevator state, manual triggering of inputs, and monitoring of sensors, actuators, and system faults. A simulation module integrated into the environment also reproduces realistic timing and delays of the mechanical components, enabling testing and debugging of the control logic without physical hardware.

5.2. Interface Description

Figure 5.1 illustrates the HMI layout designed in the CODESYS environment. The interface emulates the full behavior of the elevator system, combining real–time visual indicators, user interaction buttons, and diagnostic panels.

5.2.1. Internal Panel

On the left side of the interface, the internal panel reproduces the typical button set found inside an elevator cabin:

- Numeric keys (`G-7`) correspond to the available decks and allow passengers to select their desired floor.
- A red alarm button represents the *Emergency Button*, activating the dedicated emergency routine described in Chapter 4.
- The numeric display labeled "`%s`" dynamically shows system information such as the current deck or state of motion.

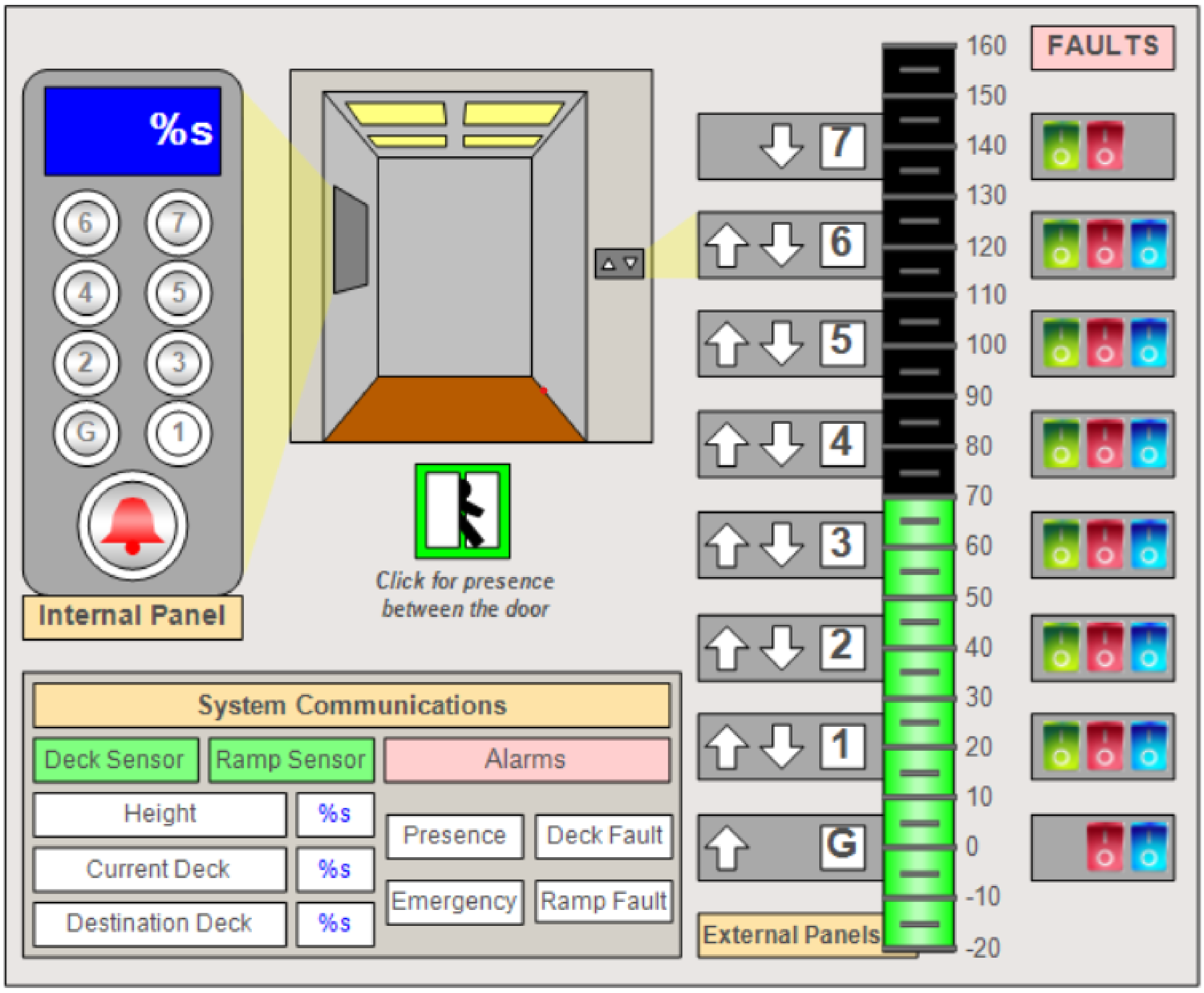

Figure 5.1: Developed HMI for the elevator control system.

5.2.2. Cab Visualization and Door Sensors

At the center of the interface, a graphical representation of the elevator cab is shown. The doors open and close dynamically in correspondence with the real door actuator (`OPEN_GA` and `CLOSE_GA`) status. Below the cab image, a presence sensor area can be toggled by clicking on the symbol representing a person. This sensor is used to simulate the detection of an obstacle between the doors, any activation of this input will prevent door closure and trigger a re–opening sequence for safety.

5.2.3. External Panels and Shaft Visualization

On the right side, a column of floors and arrows represents the external call panels located on each deck:

- Each floor features UP and/or DOWN call arrows depending on its position (intermediate floors include both, while the top and bottom floors include only one).

- When a user issues a call from a floor, the corresponding LED indicator illuminates, signaling that the request has been accepted and stored in the `SEQUENCE` array.
- A vertical bar to the left of the arrows provides a real–time visualization of the elevator's vertical position. The bar fills or empties as the cab moves along the shaft.

5.2.4. Sensor and Fault Indicators

Next to each floor label, three color–coded LEDs represent the state of the deck and ramp sensors:

- Green: ramp sensor before the current deck;
- Red: deck sensor corresponding to this floor;
- Blue: ramp sensor after this floor.

These sensors are simulated and dynamically updated by the internal logic according to the elevator position. In addition, the top–right Faults panel allows manual activation of fault conditions for testing (e.g., ramp or deck sensor malfunction).

5.2.5. System Communication Section

At the bottom of the interface, the System Communications area summarizes the real–time values of key variables:

- `Height`, `Current Deck`, and `Destination Deck` display system variables in textual form;
- The `Presence`, `Emergency`, and fault fields (`Deck Fault`, `Ramp Fault`) indicate sensor and alarm conditions;
- This section also mirrors internal process variables, enabling detailed monitoring during simulation or debugging.

5.3. LED Management Logic

The illumination of the buttons, arrows, and indicators in the HMI is handled by a dedicated routine implemented in the `LED_MANAGEMENT` state. Its purpose is to update all visual elements based on the current content of the `SEQUENCE` array (which stores active calls), the current deck position, and the system direction (`GOING_UP` flag).

The management logic executes continuously and operates according to the following rules:

1. For each element of the SEQUENCE array, the corresponding floor button and directional call indicators (UP/DOWN arrows) are turned on. This ensures that every accepted request is visually acknowledged on the interface.
2. When the elevator reaches a given floor (GVL.Current), all visual indicators corresponding to that deck are turned off — both in the cabin and on the external panels — confirming that the call has been served.
3. The emergency indicator is driven by the state of the variable em_help_3, which reflects the activation of the emergency button. When the emergency routine is active, all normal call indicators remain visible but no new inputs are accepted.

5.3.1. Implementation in Structured Text

The core of the LED management is shown below:

```
FOR i := 0 TO 8 DO
    IF GVL.SEQUENCE[i] = 0 THEN
        GVL.PGVis := TRUE;
        IF GOING_UP THEN GVL.UpDeckGVis := TRUE; END_IF
    ELSIF GVL.SEQUENCE[i] = 1 THEN
        GVL.P1Vis := TRUE;
        IF GOING_UP THEN GVL.UpDeck1Vis := TRUE;
        ELSE GVL.DownDeck1Vis := TRUE; END_IF
    ...
END_FOR

(* Turning off LEDs of the current deck *)
IF GVL.Current = 0 THEN
    GVL.PGVis := FALSE; GVL.UpDeckGVis := FALSE;
ELSIF GVL.Current = 1 THEN
    GVL.P1Vis := FALSE;
    GVL.UpDeck1Vis := FALSE; GVL.DownDeck1Vis := FALSE;
...
END_IF

GVL.EmergencyVis := NOT em_help_3;
```

This logic ensures full synchronization between the simulated hardware and the interface. Each time a new call is inserted into the SEQUENCE array by the logic controller, the corresponding floor LEDs and call arrows are illuminated. As soon as the elevator reaches that deck and the call is cleared, all indicators associated with that floor are turned off automatically.

5.4. Integration with System Logic

The `LED_MANAGEMENT` state operates continuously and independently from the main control sequence. This design guarantees that the HMI remains responsive at all times — even during emergency phases or motion transitions — providing accurate feedback to the user. By reflecting the internal logical states in real time, the interface acts as both an operational control panel and a visualization tool, completing the digital twin representation of the elevator system.

6. Simulation and System Validation

6.1. Purpose of the Simulation Environment

To verify the correctness and robustness of the elevator control logic, a dedicated simulation program was developed in CODESYS. The simulation replicates the physical dynamics of the elevator system, including cab motion, door actuation, sensor activation, and safety limits, allowing full validation of the logic without physical hardware.

This environment also provides a direct visual link between the control variables and the HMI interface, ensuring that the virtual indicators, lights, and buttons accurately reflect the system's real-time behavior.

6.2. Initialization and Global Variables

The simulation begins by defining the initial position of the elevator cab inside the hoistway. This initialization procedure is executed only once, during the first cycle of the program:

```
IF NOT GVL.InitCycle THEN
    GVL.InitCycle := TRUE;
    GVL.Height := 20 * GVL.InitialDeck;
    GVL.Door := 0;
    GVL.Current := GVL.Height / 20;
END_IF;
```

The variable `GVL.Height` represents the cab's vertical position in arbitrary simulation units (corresponding to centimeters or decimeters in a real shaft). Each deck is spaced by 20 units, providing sufficient granularity to simulate ramps and decks between floors. The `GVL.Door` variable controls the degree of door opening (from 0 = fully open to 60 = fully closed), while `GVL.Current` identifies the current deck index derived from the height.

6.3. Dynamic Behavior of the Simulation

After initialization, the program continuously updates the elevator's state at each cycle to emulate physical motion, door dynamics, and sensor transitions. This allows real-time correspondence with the HMI visualization and controller logic.

6.3.1. Door Simulation

The door mechanism is modeled as a linear displacement between 0 and 60 units:

- When the control variable `GVL.Closing` is TRUE, the door closes progressively (`GVL.Door := GVL.Door + 1`).
- When `GVL.Opening` is TRUE, the door opens (`GVL.Door := GVL.Door - 1`).

Logical flags `GVL.Open` and `GVL.Closed` are set once the corresponding limits are reached, ensuring synchronization with the real control logic and preventing overextension:

```
IF GVL.Door >= 60 THEN GVL.Closed := TRUE; END_IF
IF GVL.Door <= 0  THEN GVL.Open   := TRUE; END_IF
```

This mechanism accurately reproduces door behavior, allowing the HMI to visualize both motion and safety interlocks.

6.3.2. Cab Movement and Position Control

The cab motion is controlled by the `GVL.Motor_on` and `GVL.Up` variables:

- If both are active, the cab moves upward;
- If `GVL.Motor_on` is TRUE and `GVL.Up` is FALSE, it moves downward.

The displacement speed depends on the velocity flag `GVL.Vel`: a slow speed corresponds to one unit per cycle, and a fast speed corresponds to two units. The corresponding logic is:

```
IF GVL.Vel THEN
    GVL.Displacement := 2;
ELSE
    GVL.Displacement := 1;
END_IF
```

Height is then incremented or decremented according to direction, respecting boundary limits between -20 and 160 simulation units.

6.3.3. Limit and Safety Sensors

Two virtual limit switches (`GVL.LimitUp` and `GVL.LimitDown`) are defined to prevent motion beyond the shaft extremes:

```
IF GVL.Height <= -10 THEN GVL.LimitDown := TRUE; END_IF
IF GVL.Height >= 150 THEN GVL.LimitUp := TRUE; END_IF
```

These conditions correspond to the top and bottom mechanical stops of the hoistway.

6.4. Deck and Ramp Sensor Emulation

The most significant aspect of the simulation is the virtual replication of the deck and ramp sensors. Each floor features one deck sensor and two ramp sensors (one before and one after the deck), each associated with a specific height interval.

6.4.1. Deck Sensors

Deck sensors are active when the cab height lies within a ±2 unit window around the nominal floor height. The following code illustrates the logic for deck 3 (60 units high):

```
IF ((GVL.Height >= 60-2) AND (GVL.Height <= 60+2)) THEN
    GVL.Sensor3OK := TRUE;
ELSE
    GVL.Sensor3OK := FALSE;
END_IF
```

Similar conditions are defined for all decks (G to 7), ensuring precise deck identification throughout the elevator's motion.

6.4.2. Ramp Sensors

Ramp sensors act as intermediate triggers between decks, providing early detection before and after reaching a floor. These are used by the motion logic to decelerate the cab before arriving at a deck, enabling smooth and realistic stopping behavior.

Each ramp sensor is activated when the cab height is within a ±2 unit range around its nominal value. For example, the ramp sensor before Deck 3 is defined as:

```
IF ((GVL.Height >= 55-2) AND (GVL.Height <= 55+2)) THEN
    GVL.Ramp3DownOK := TRUE;
ELSE
    GVL.Ramp3DownOK := FALSE;
END_IF
```

Ramp sensors are logically combined in `GVL.RampSensor`, providing a single Boolean output to the controller for simplified interpretation.

6.5. Fault Simulation

The simulation also includes optional fault generation logic. Users can activate *Deck Fault* or *Ramp Fault* conditions through the HMI to test the controller's reaction to abnormal sensor behavior. This mechanism is implemented by inverting or disabling sensor Boolean values and verifying whether the logic correctly transitions to safe states or prevents movement.

6.6. Integration with the HMI

All variables computed by the simulation program are directly linked to the HMI components (see Chapter 5). The door animation, cab position bar, and sensor indicators are updated in real time, allowing visual verification of every control phase:

- During normal operation, the HMI reflects the continuous motion of the cab and door transitions;
- During emergency activation, the simulation accurately reproduces the stopping and door-opening behavior at the nearest safe deck;
- During initialization, both the graphical indicators and sensors follow the step-by-step sequence described in the logic charts.

6.7. Validation Results

Through the simulation, all control functions (initialization, normal operation, emergency management, and LED synchronization) were successfully validated. The simulation environment proved essential for verifying the consistency between logical design and expected physical response, offering a controlled and repeatable testing platform.

This approach ensured that every aspect of the elevator's digital control, from call handling to actuator management, operates safely, deterministically, and in full alignment with the specified requirements.