Project: Specification

1DT059: Model-based Design of Embedded Software, 2022

November 18, 2022

In this project, you will develop and deploy embedded software to control a small (simulated) elevator system. The (simulated) elevator has been constructed by Magnus Lång in September 2020. It has GUI interface animation, programmed in Qt. It is controlled via an API, described in appendix A. Your tasks include

1. to design a controller in Simulink/Stateflow to control the (simulated) elevator,
2. to combine your designed controller with a Simulink model that mimics the (simulated) elevator system,
3. to show by simulation that your controller fulfills the informal and formal requirements (given below)
4. to formalize a number of requirements for your developed model of the elevator system, and show by testing that they are satisfied
5. to generate and deploy code from your controller, integrate it with the simulated elevator, and
6. test and tune your generated code together with the simulated elevator, and show that it behaves as described below.

Designing a controller for a similar (not identical) Elevator was part of Assignment 2. so you should have finished this assignment before starting with the project. You can use and adapt the solution from one of the group members as a basis for the project. Note that the assignment 2 controller needs some extensions to fit into the project. You may also want/need to improve it while carrying out the project.

This assignment is to be solved by groups of two students. Each group should hand in their solution, and also demonstrate their working design. If there are difficulties to form groups, get back to us, and we will try to resolve this. It is of course not allowed to share or copy (parts of) solutions between groups.

In the following, we describe the components of the project.

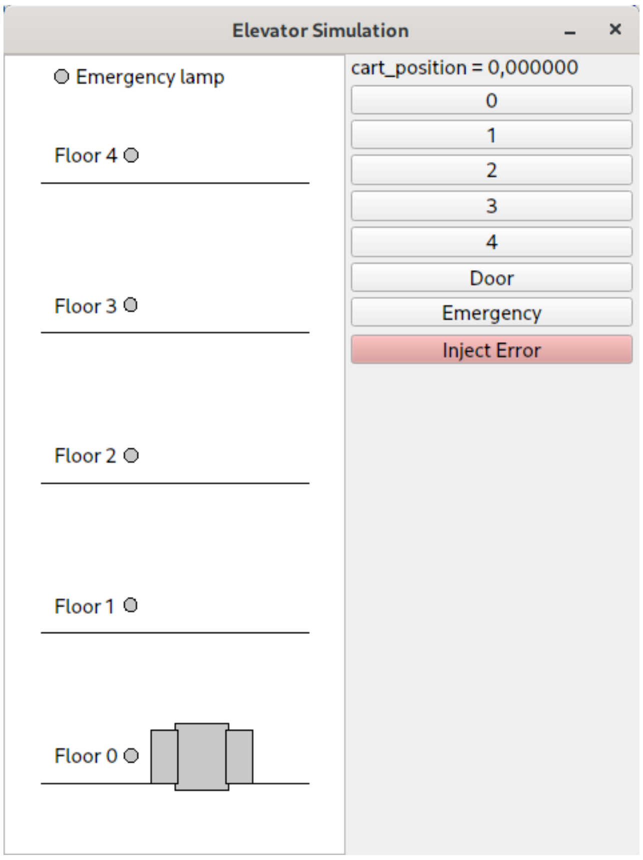

Figure 1: Simulator Graphical Interface

Interface to Simulated Elevator System The interface, shown in fig. 1, contains 7 buttons (five, numbered 0, 1, 2, 3, 4, correspond to five floors, the sixth to open the doors, and the seventh is an emergency stop button) that can be pressed by users (for simplicity, we do not distinguish between buttons at floors and buttons inside the elevator, there is just one button for each floor). The purpose of the eighth button "Inject Error" is to test the fault tolerance of the system. The (simulated) elevator has a lamp for each floor, and a red lamp connected to the emergency stop button. The movement of the (simulated) elevator cart is controlled by a motor, which you can control via the given API. The position of the elevator cart can be read by the Controller via a "sensor", which can be accessed via the supplied API. Note that this position sensor can be unreliable and can sometimes produce strange results (see further below).

Informal Requirements for Elevator System The elevator should behave in a way rather similar, but not identical, to the controller in last problem in Homework 2. Informally, the requirements include the following.

- Typical "elevator-behavior" requirements, including that: whenever the button for floor n has been pressed, the elevator must eventually arrive to floor n. When this happens, the doors open. Once the doors are open, they remain open for at least 5 seconds. The doors must close before leaving any floor. Whenever the "emergency stop" button is pressed, the elevator is stopped and the red lamp is lit. The lamp is switched off and the elevator resumes to normal when the "emergency stop" button is pressed again.

- Once a floor button is pressed, double pressing (double clicking within a short time period Δt) that floor button should undo the floor selection. In such cases, the elevator should not stop at that specific floor. You should make appropriate assumption about the value Δt and other related scenarios.

- The elevator should run smoothly, and make smooth stops and starts at floors. It should thus exhibit a suitable maximum speed and a suitable maximum absolute value of acceleration.

- The elevator should stop only at the precise locations of the floors, as specified by the API.

- The elevator should be robust to faults/disturbances, at least in the following sense: if the distance sensor provides clearly wrong data (e.g., if the **"position"** input shows strange values for some reason, then the elevator should detect that "something is wrong"), and perform suitable action (maybe this could be just stopping, and/or possibly lighting up some lamp, etc. Feel free to design this in a suitable way).

- The elevator cart in your Simulink model should move in approximately the same way (similar speed, starting and stopping behavior) as that in the (simulated) elevator.

You can reuse your floor-planning algorithm from Assignment 2, but if its performance is not good, you are encouraged to optimize it (this has lower priority than making the elevator functional).

Formal requirements that should be formalized and tested For your constructed Simulink/Stateflow model of the elevator system, you should formulate requirements (numbered R1 – R4 below). Each requirement should be formulated as a monitor, which has an output signal that can be sent to a `Proof Obligation` or an `Assertion` block, and which can be added to your elevator system. You should also generate a test suite for your elevator controller, which should be used to check requirements R1 – R4. You should use Simulink Design Verifier (SLDV) to generate a test suite which covers as much as possible of your model (including the monitors). Generating a test suite which covers *all* elements of your model may be beyond the capability of SLDV, but your test suite should come reasonably close to this. This means you should generate tests under a coverage criterion that gives the "best" result. You can experiment with different coverage criteria: let SLDV run for at least an hour on each and see which one gives the best (in your subjective judgment) test suite. Below are some suggestions that may help in the test generation:

- The test generation must use a model which includes your simple elevator model. If the elevator is modeled using a continuous-time integrator you should, when you generate the test suite,replace it by a discrete-time integrator (SLDV does not like continuous-time integrators)
- You should replace user interface blocks ("buttons" etc.) by open input blocks, for which the generated test suite will define sequences of values. You can focus the test generation by constraining the values of inputs (e.g., to 0 or 1, or to [0,1]).

Run your test suite on the controller instrumented with monitors. If some tests trigger violations of requirements you can either

- improve the controller, if the violation can be said to be a design mistake, or
- explain why it is reasonable that the requirement be violated and why the model need not be fixed.

The requirements are as follows.

R1: The doors are never open when the elevator is not at some floor.

R2: When doors are opened, they remain open for at least 5 seconds.

R3: The elevator stops at floor A only if the button for floor A has been pressed previously, after the last time that the elevator stopped at floor A, and since then has not been "unpressed". Here, you can choose a suitable value for A, such as 2 or 3.

R4: After a button-push for any floor, the elevator arrives within d time units to that floor, unless the button is "unpressed".

Overall comments In order to carry out the project, you likely need to do the following.

- You should make yourself familiar with the (simulated) elevator and its interface; this is described later in this document.
- Making a Simulink model of the correspondence between the control signals from the controller to the (simulated) elevator, controlling the speed of the elevator cart. This correspondence includes, soft starting and stopping, and approximately similar speed.

What you should hand in You should produce a report (in .pdf), where you describe your overall design effort (what design decisions where applied to the various parts), how your controller is designed (description of its main mechanisms), how your Simulink model was constructed (the main steps of this). You should also describe your formalization of the requirements, and report how much coverage has been achieved in your test suite for checking them. You should describe detected discrepancies between your designed elevator system and formal requirements, and what you made to resolve them. Include the models of controller, monitors, and simulated elevator, and report from test generation effort.

You should also present your design on the day of project presentations. This presentation should explain your controller design, and how you produced the Simulink elevator model. You should also demonstrate a usage scenario on the (simulated) elevator, and thereafter apply the same usage scenario on the Simulink model.

To produce a model that can be used both for code generation and simulation, the block *Environment Controller* (see Simulink documentation) can choose to forward either of two signals, depending on whether code is generated or the model is simulated. You may consider whether you want to use it (although it is not required). An illustration of how the block can be used is given in the file `possible_sketch.slx`.

Submission

Solutions (all files) to this assignment are to be submitted via the Student Portal by **January 25, 2023**. You should also prepare a demonstration. These demonstrations should happen on **January 12, 2023**. A schedule will be put up later, please plan to attend demonstrations of other groups also.

A The (Simulated) Elevator

The (simulated) elevator is a program that you run on your computer. You program it by writing a program that interfaces with the API, very much like if you were programming a micro-controller to drive a real elevator.

You will receive full source code for the (simulated) elevator, so that it may be compiled for your own computer, but you are not allowed to modify it, and are not intended to inspect it, aside from the `simulation/elevator.h` header file that declares the methods of the API, shown below:

```c
/* Elevator Simulator, version 1 */
#ifndef ELEVATOR_H
#define ELEVATOR_H

#ifdef __cplusplus
extern "C" {
#endif

#include <stdbool.h>

/* Opaque handle type to a simulation */
typedef struct elevator *elevator_h;

/* Bitfield of which buttons are pressed down. */
typedef unsigned button_state;
#define BUTTON_STATE_FLOOR0    0x01
#define BUTTON_STATE_FLOOR1    0x02
#define BUTTON_STATE_FLOOR2    0x04
#define BUTTON_STATE_FLOOR3    0x08
#define BUTTON_STATE_FLOOR4    0x10
#define BUTTON_STATE_DOOR      0x20
#define BUTTON_STATE_EMERGENCY 0x40

/* Bitfield of which lamps should light. */
typedef unsigned lamp_state;
#define LAMP_STATE_FLOOR0    0x01
#define LAMP_STATE_FLOOR1    0x02
#define LAMP_STATE_FLOOR2    0x04
#define LAMP_STATE_FLOOR3    0x08
#define LAMP_STATE_FLOOR4    0x10
#define LAMP_STATE_EMERGENCY 0x40

/* Starts the simulation. Writes a valid handle to *handle_p and returns 0 on
 * success, returns nonzero on failure. */
int start_elevator(elevator_h *handle_p);
/* Stops the simulation */
void stop_elevator(elevator_h);
/* Speeds are between -1 (full speed down) and 1 (full speed up). */
void set_motor_speed(elevator_h, float);
/* Height of the elevator in floors above ground level. Values range between
```

```c
 * 0.0 (ground floor) and 4.0 (4th floor). */
float get_cart_position(elevator_h);
/* Get the buttons that are currently pressed down */
button_state get_button_state(elevator_h);
/* Opens or closes the doors. */
void set_doors_open(elevator_h, bool);
/* Sets the lamps specified in lamp_state to on, the rest off
    . */
void set_lamp_state(elevator_h, lamp_state);

#ifdef __cplusplus
}
#endif

#endif // ELEVATOR_H
```

You will also receive a file `main.c` that demonstrates the use of this API, which you should replace with your own file.

```c
/* Don't include anything else from simulation/ */
#include "simulation/elevator.h"

#include <unistd.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
    elevator_h e;

    /* Before we start, we must initialise the simulation */
    int ret;
    if ((ret = start_elevator(&e))) {
        fprintf(stderr, "Failed to initialise elevator
  simulator: %d\n", ret);
        return 1;
    }

    /* This is an example that runs a fixed program. Replace
  this by hooking up
     * to your own elevator controller. */
    set_doors_open(e, true);
    sleep(2);
    set_doors_open(e, false);
    set_lamp_state(e, LAMP_STATE_FLOOR4);
    set_motor_speed(e, 0.5);
    /* Note, this is a terrible way to control an elevator.
```

```
   */
    while (get_cart_position(e) < 4) usleep(1000);
    set_motor_speed(e, 0);
    usleep(500000);
    set_lamp_state(e, LAMP_STATE_FLOOR0 | LAMP_STATE_FLOOR2);
    set_doors_open(e, true);
    while (!(get_button_state(e) & BUTTON_STATE_FLOOR0))
   usleep(1000);
    set_doors_open(e, false);
    set_motor_speed(e, -0.5);
    sleep(5);
    set_motor_speed(e, 0);
    sleep(1);

    /* If we decide to quit, for whatever reason, we should
   stop the
     * simulation. */
    stop_elevator(e);
    return 0;
}
```

B Compiling the (Simulated) Elevator

Please use a Linux environment to work with the (simulated) elevator. In order to build the (simulated) elevator, you also need Qt and CMake. On Ubuntu, you can install these by issuing following command:

```
$ sudo apt install qtbase5-dev cmake
```

Then, assuming you have the (simulated) elevator in a directory called `elevator_sim` under the current, you can make a build directory `build` like this:

```
$ mkdir build && cd build
$ cmake ../elevator_sim
```

Then, in the build directory, you can just type `make` to build the simulator. This produces both a dynamic library `libSimulation.so` with the simulator, and an executable `elevator_sim` (from `main.c`). You can then run it in the `build` directory with

```
$ ./elevator_sim
```

You should see the interface (fig. 1). When you're ready, add your generated code and replace `main.c` to hook the simulator up to your controller. You can add extra `.c` files to the build in `CMakeFiles.txt`. Good luck!