Assignment 1: Modelling

INTRODUCTION

The objective of this project is to model, specify, verify, and test control software for elevators. The entire project assignment consists of three parts:

- **Part I: Modeling** The first part aims at producing models of the control software and its environment. This is related to Block I of the course.
- **Part II: Verification** The second part aims at expressing correctness of the system as a set of temporal formulas and to prove that the models satisfy these properties. This is related to Block II of the course.
- **Part III: Testing** The last part aims at verifying that implementations of the control software conform to the specification models. This is related to Block III of the course.

Each part is consists of several tasks. Each part ends with a description of what you should hand in for that part. Each of the tasks contains several actions. Actions are labelled as follows:

▷ Read this assignment.

Each action is followed by further explanation.

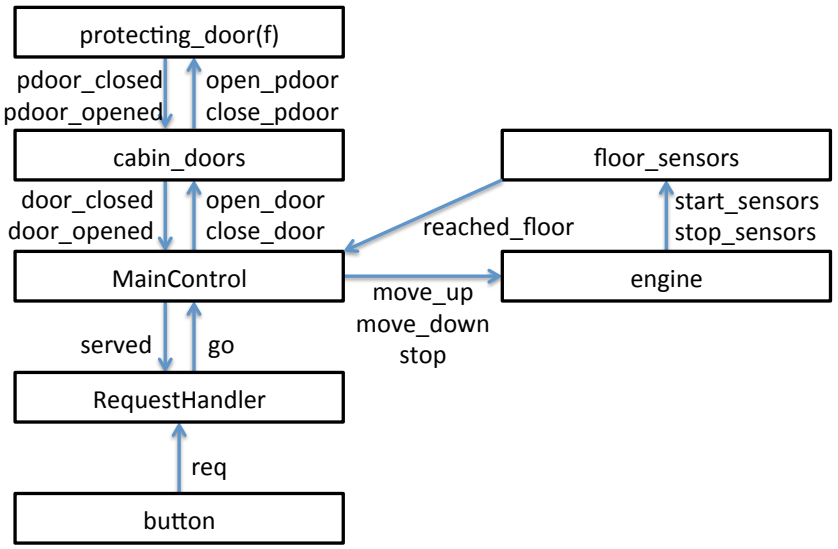

FIGURE 1.1 A simple elevator system

Figure 1.1 presents an overview of an elevator system. A process *button* sends out `req`s (requests) to the *RequestHandler*. A `req` contains a natural number representing the requested floor. The RequestHandler receives `req`s and decides when and to which floor the elevator should move. By transmitting a `go`, it tells the control software to move to a certain floor. A `go` contains the target floor, to which the elevator is to move. The RequestHandler then has to wait for a `served` before it can send another `go`. The choice of the destination floor is taken according to an allocation policy. In the assignment, we will consider a specific policy, explained below.
The control software, i.e., *MainControl*, will move the elevator to the target floor and send out a `served` once the elevator has reached that floor. MainControl controls the *engine* and the *cabin_doors*.

The engine can be instructed to move up or down, or to stop. Once the engine starts moving, it will start sensors. Each time the elevator reaches a floor, these sensors will send a `reached_floor` to the MainControl. When the elevator is moving up or down, the floor sensors indicate when a new floor is reached. Note that the sensors do not tell the floor number. It is the task of MainControl to keep track of its current position. When the destination floor is reached, MainControl should stop the engine and open the door.
The cabin_doors can be instructed to open or close. Each floor has a *protecting_door* that is closed by default, and that should be open only when the elevator is standing still at that floor. When the cabin_door is given the signal to close, they will first close the protecting doors of the current floor. Once these are closed, the cabin_door will be closed. For opening, it is similar. Note that the elevator cabin always waits for passengers in a state where its cabin door and the protecting door are open.

Our objective is to model, verify, and test systems with multiple elevators for various numbers of floors. Also, a core aspect of elevator control is the algorithm used to determine which request(s) should be served next. We will model a specific algorithm (explained below) and prove properties over it.

1 Tasks for Part I (20 hours)

The objective of Part I is to create a model of the elevator control software, more specifically the control unit and the request handler. Part I consists of four tasks. The first two tasks constitute a tutorial to get started with UPPAAL and the elevator model. The remaining tasks consist of creating UPPAAL models.

TASK 1.1. UPPAAL TUTORIAL (2 HOURS)

▷ Watch the movie on starting with UPPAAL on YouLearn ("Cursus", then "Project").
▷ Download and install UPPAAL 4.1.19 (see YouLearn, "Bronnen", "Externe Links").
▷ Browse through the tutorial (see YouLearn, "Bronnen", "Externe Links").
Do not read the entire tutorial. Just browse through it, so that you know what kind of information can be found in the tutorial and so that you can look it up as needed. Some interesting parts are:

- the last part of Section 2.1 enumerating additional features wrt. timed automata (starting on page 4, below Definition 3).
- Sections 2.2 and 2.3.
- Section 7 about modelling patterns. The objective is to know that there exist four interesting patterns (see Fig. 22) so that you can get back to them and get more details when you feel you need them.

TASK 1.2. FIRST IN FIRST OUT REQUEST HANDLER (2 HOURS)

The second task is to model a control unit implementing a First In First Out (FIFO) scheduling policy. Floor requests are queued and then served in the order in which they are received by the controller. The remainder of this task provides instructions how to do this. The purpose of Task 1.2 is to familiarize yourself with both UPPAAL and the elevator model.

Your starting point is a UPPAAL system with all the automata presented in Figure 1.1.
▷ Download the UPPAAL elevator model from YouLearn ("Cursus", then "Project")
You will find two files. File "elevator.xml" contains UPPAAL models and file "elevator.q" contains the no deadlock property. Section 2 provides you with explanation of the system.

▷ Read Section 2 of this document and then continue here.

Dummy Request Handler.
The current model contains a dummy request handler. You are going to replace this by a FIFO request handler.

▷ Open the UPPAAL model and inspect the template “dummy_RequestHandler”.
This request handler accepts a request and then instructs the main control unit to go the requested floor. After this request has been served, it can receive a new request. The communications are modelled in two different ways. The first communication is between the button and the dummy request handler. This communication uses a parametrized channel *req*. Basically, for each floor *i* there is a channel *req[i]*. When the dummy request handler receives *req[i]* it stores *i* in the local variable *current_req*. The second communication is between dummy request handler and the main control unit. This communication makes use of a global variable (target_floor, see "Declarations" of the UPPAAL project). When sending a *go*, the request handler puts a value into this global variable. When receiving a *go*, the main control reads the global variable and immediately sets it to zero. In UPPAAL, this is a very efficient[1] way of encoding a *synchronous communication*.

[1] It is a way to mitigate the state space explosion problem, on which more in LU 6

▷ Verify that the current model is deadlock-free.
Go to the Verifier and check the property "A[] not deadlock". This means: "for all executions of the system at any given time, the system is not in a deadlock".

FIFO Request Handler.
The dummy request handler can only deal with one request at a time. Figure 1.2 provides a model of a FIFO request handler. To understand its behavior, take a look at the "Declarations" of template "dummy_RequestHandler". It provides operations *enqueue* and *dequeue* to store requests into a queue. These operations can be called in the update part of a transition. An enqueue operation only makes sense when the queue is not full. Similarly, a dequeue operation can only happen when the queue is not empty.

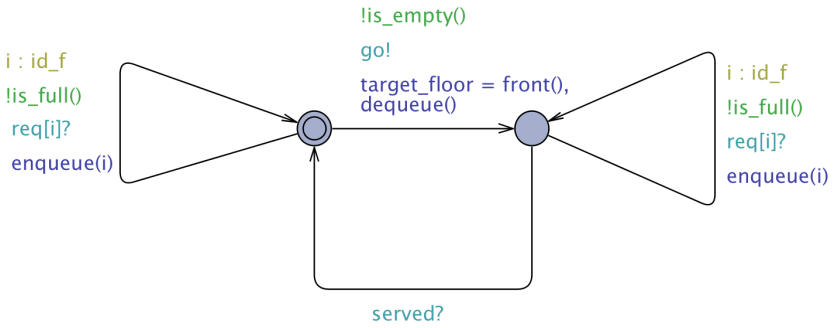

FIGURE 1.2 The FIFO request automaton

▷ Replace the dummy request handler by the automaton in Figure 1.2.
You can rename the request handler in the field "Name:". If you do this, you should also rename it in the "System Declarations".
▷ Verify that the current model is deadlock-free.
When you click "Check" in the Verifier, UPPAAL will ask you if you want to reload the model. You will always want to answer "Yes". Also, when reloading the model, UPPAAL might find syntax errors. When that happens, you can go back to the Editor where all syntax errors will be red and underlined.

The final part of Task 1.2 introduces you to the Simulator.
▷ Go to the dummy main control and remove the edge labelled *served!*.
Convince yourself that the system now contains a deadlock. If the system is in this deadlock, in what state is the FIFO Request Handler? And the Main Control? What will be the value of global variable *current_floor*? And of the local variable *dest_floor* of Main Control?
▷ Verify that the current model is indeed *not* deadlock-free.
▷ Go to the Simulator and click "Random".
The Simulator runs your system for some steps. It will end in a deadlocked state. In this state, there are no enabled transitions. Check whether the deadlock corresponds to the deadlock you envisioned.
▷ Go to dummy main control and redraw the edge that you removed.

TASK 1.3. MAIN CONTROL (6 HOURS)

Open the UPPAAL model and take a look at the template "dummy_MainControl". It opens the elevator door. When receiving a *go*, it stores the contents of the global variable *target_floor* into the local variable *dest_floor*. It now knows which floor it is destined to go to. The main control should now do its task (see the introduction). However, the dummy does nothing and immediately sends a *served*.

▷ Replace the dummy main control by a model of a realistic main control.

It is a good habit to work in small steps. The goal is to slowly increase the complexity of a model. Regularly check if your model deadlocks and use the simulator to see if your extensions behave as you expected.

TASK 1.4. DESTINATION GROUP CONTROL (10 HOURS)

In this task, you will (1) extend your model to deal with several elevators and (2) model a Destination Group Control (DGC) mechanism.

▷ Model multiple elevators. This can elegantly be achieved by adding the number of elevators as a new variable of your model, in a similar way to the number of floors. You can parametrize automata in the “Parameters:” field in the Editor.

Next, the current simple RequestHandler is to be replaced by a smart one, called a DGCRequestHandler (for: Destination Group Control). We consider a simple DGC policy for a so-called "up-peak" situation. We describe the policy for a configuration of three elevators and nine floors (excluding the ground floor). Elevator 1 is assigned to serve floors 1, 2, and 3. Elevator 2 is assigned to serve floors 4, 5, and 6. Elevator 3 is assigned to serve floors 7, 8, and 9. All elevators start at the ground floor (0) with their door open. An elevator leaves *as soon as* it has 5 requests, that is, the sum of the requests to the floors served by the elevator equals 5. When all requests have been served, the elevator comes back to the ground floor. To prevent from waiting for passengers forever, whenever there is at least one request, the elevator will also leave after at most `max_wait` time units.

▷ Model this DGC, taking into account the following helpful, *obligatory* constraints.
Helpful constraints:

- Other than parametrization, do not change the templates of the doors, engine, sensors or the button.
- Use the existing code for the queue, instead of writing a lot of code yourself.
- First, you have to make sure that requests initiated by the button are sent to the ReqHandler of the corresponding elevator. To do this, formulate a function `is_req_for_lift` that takes as parameter a floor request and returns a Bool. This function should be independent of the number of elevators and floors. Communication between the button and the ReqHandler can then be achieved using this function and *conditional synchronous communication*, one of the modelling patterns.

Do I have a good model?

After completing Task 1.4, you can assess yourself whether your model is sufficient to be handed in. It should satisfy the following criteria:

- It should be easy to play with different configurations, for instance, different numbers of floors or elevators. This means that – for instance – going from a model with 3 floors to a model with 4 floors should not be more work than changing the value of one parameter.
- Your model should be understandable. This means that states, channels and functions have clear and descriptive names. Any code that is used should be commented. It also means that the model is small and elegant.
- Your model should satisfy the "no deadlock" property and this should be verifiable for a setting with 2 elevators and 3 floors (including the ground floor).

For Part 1, you should deliver an UPPAAL .xml file and its associated .q file.

2 Model of the environment

To help you, we give you the models of the environment of the system. The environment is composed of the following components:

- The elevator cabin door.
- The door protecting users at a floor.
- The main engine.
- The buttons of the control panel inside the cabin and the request buttons at each floor.
- The floor sensors indicating when a new floor is reached.

Note that in all models variable x denotes a clock variable.

2.1 THE ELEVATOR CABIN DOOR

The cabin door is modelled using the UPPAAL automaton in Figure 1.3. A closed door receives an `open_door` command and starts by opening the protective door from the current floor. This must take less than `door_time` time units. When the protective door has opened, the cabin door is opened as well and a `door_open` is emitted to notify the MainControl that the doors are open. Similarly for closing the door when its open.

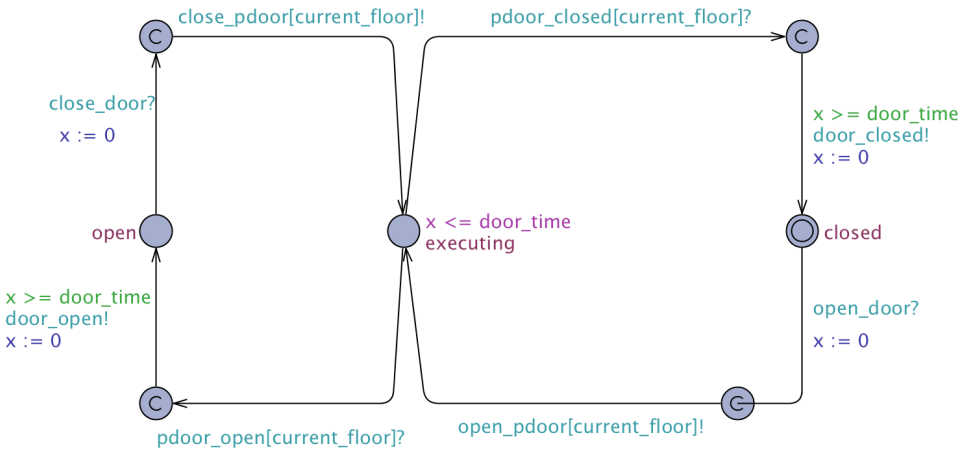

FIGURE 1.3 The cabin door automaton

2.2 THE PROTECTING DOOR

At each floor, there are doors protecting users from falling in the elevator shaft. These doors are modelled using the UPPAAL automaton in Figure 1.4. A door receives an `open_pdoor` or `close_pdoor` command and starts executing this command. It takes `door_time` time units to close or open the door. Note that to ensure that the protecting door will always be open or closed, two intermediate states are needed. Indeed, after receiving a `close_pdoor` command, the protecting door does not accept an `open_pdoor` command before actually being closed. The protecting door will also not emit an `pdoor_open` message if the door is not open. The same holds when the door is closed.

2.3 A GENERIC BUTTON MODEL

Figure 1.5 shows a UPPAAL automaton for a generic button. There is at least 1 time unit between two button actions. The action of pressing the button is represented by

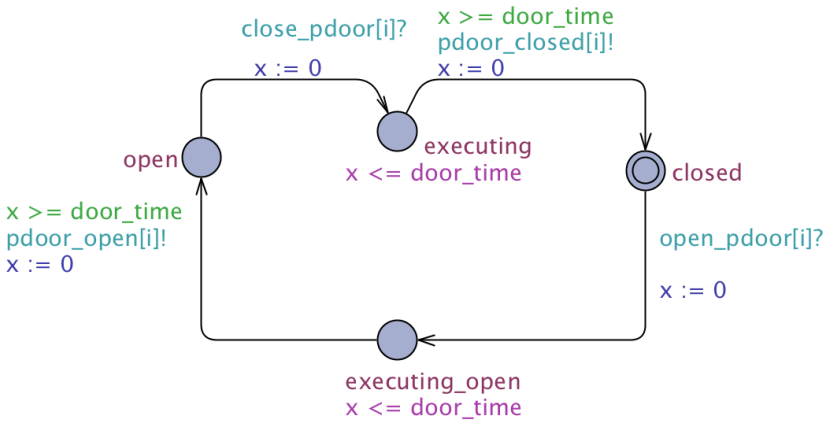

FIGURE 1.4 The protecting door automaton

output action `req[i]`, where *i* denotes the number of the requested floor. Note that here no difference is made between a request from the cabin and a request from a floor. Also, this button non-deterministically chooses which floor has been pressed. This button is also the only button of the UPPAAL model. This makes the model simpler to verify as only one clock is needed for modelling all the buttons. In a model with one button per floor, there would be one clock variable per floor. For verification efficiency, it is important to minimise the number of clock variables.

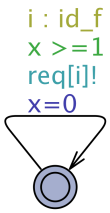

FIGURE 1.5 A parametric button.

Remark. One could imagine a simpler model where clock x is removed. The issue with this simpler model is that it takes zero time to press the button several times in a row. This allows for *zeno* behaviours, that is, the button can be pressed infinitely often in zero time. In practice, such a button can prevent the progress of time.

2.4 THE MAIN ENGINE AND THE FLOOR SENSORS

The engine and sensors models are given in Figure 1.6 and Figure 1.7. The engine initially starts in an idle location and waits for the commands `move_up` and `move_down`. When such a command arrives, the engine moves to a state where it will immediately start the sensors. After that, it moves to state where it is actually moving up or down until the command `stop` arrives. The engine then moves back to the idle location after having stopped the sensors.
The sensors receive the `start_sensors` and `stop_sensors` commands. When such a start command arrives, the time needed to reach the next floor is simulated by spending `floor_time` time units in the state named moving. Reaching the next floor is indicated by output action `reach_floor`. Note that the floor sensors will keep emitting `reach_floor` signals as long as they do not receive a stop command. This models the fact that if the engine is not stopped, then the elevator cabin continues moving even after the ground floor or the roof of the building have been reached.

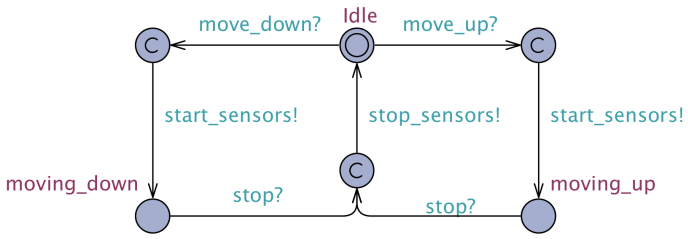

FIGURE 1.6 Model of the engine.

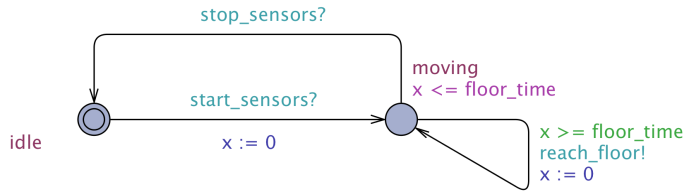

FIGURE 1.7 Model of the floor sensors.