Modelling of Software Intensive Systems

Assignment 5: State Charts

December 17, 2023

Thomas Gueutal
s0195095@ad.ua.ac.be

Felix Vernieuwe
s0196084@ad.ua.ac.be

Outline

0. Introduction

This report details our findings and solutions for assignment 5. We first attempt to work out the provided exercises and fully understanding them. Then we move on to the modelling of a statechart which satisfies the given set of requirements.

The assignment is made for the course *Modeling of Software-Intensive Systems* at the University of Antwerp, with the TA for this task being Joeri Exelmans.

0.1. Assignment Workload

Thomas Gueutal

Time spent on the assignment: 10 hours

Worked on:

- Solving exercises (pair programming)
- Implementing the Traffic Controller SC
- Writing a summary of the SC design in the report
- Writing a summary of the test design in the report

Felix Vernieuwe

Time spent on the assignment: 7 hours

Worked on:

- Solving exercises (pair programming)
- Verifying SC solution with requirements definition
- Writing and completing the report

0.2. Project Tree

Our submission archive is structured as follows:

- `/assignment-files/` : Assignement description, and provided images, files and scripts
- `/attachments/` : Exported SVG renderings of the statecharts (exercises + solution)
- `/report/` : Folder containing the report in PDF form
- `/statechart/` : Folder containing extracted `Mosis23StartingPoint.zip` with solutions

0.3. Tools & Scripts

We made use of **Itemis CREATE** to solve the exercises, per the requirements. All our scripts and solutions can be found in the `statechart` directory. This report was written in **Typst**.

0.4. Assignment Discussion

For every report, we try to equally divide the workload across the two of us, so that we both understand and practice the underlying theory and used tools. This proved a little bit more tricky for this assignment, as we found that it was not very intuitive to both work in turns on implementing the statechart. Thus we chose to let one person implement the actual statechart, and the other to write up the report. Since the course Software Engineering in Ba3 also has an assignment on yakindu-based state charts, we find that in the end we both understand the topic of this assignment.

The assignment was implemented in following order:

- solve all exercises: A to E (pair-programming)
- draft answers to exercises in report
- extend the `Statechart.ysc` SC iteratively
  - implement each requirement completely, before moving on to the next one
  - verify the validity of the extension manually using `trafficlight_gui.py`
  - run the `trafficlight_test.py` tests to ensure the validity of the extension
- write a fourth test case manually in `trafficlight_test.py`

1. Basic Exercises

1.1. Exercise A

Question: why is event 'y' never raised?

When StateA is entered as default state on simulation initialisation, the state requests a timer to be started for each of its transitions containing an *after xs* annotation, indicating that said transition should be taken after a given duration has passed. Each time StateA is **exited**, every such started timer gets **reset/cancelled**.

If we now inspect the two transitions of StateA, we see two different `after` statements: one firing after a single second having passed, the other after two seconds. Each respectively raises its own timed event: `x` and `y`.

Finally, it is obvious that the timer started by the `after 1s` transition will *always* finish sooner than the timer for the `after 2s` transition. So after the first second of the simulation has passed, a timed input event gets fired, which matches with the `after 1s` transition. This causes the state to be left, resetting both timers.

In short, the timer for the `after 2s` transition will never be able to finish uninterrupted, as it always gets reset/cancelled when a second passes. The transition with trigger `after 2s` with event `y` will thus never be able to fire.

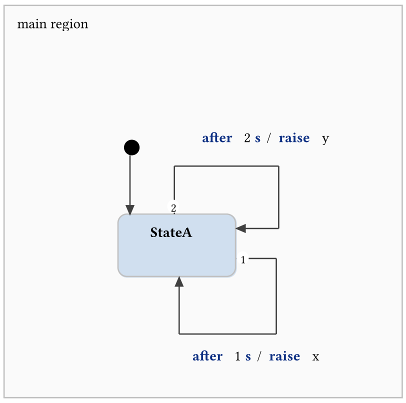

Figure 1: Exercise **A** state chart diagram

1.2. Exercise B

Question: why does the output event trace look like this:

- 2s: ***inner***
- 3s: ***outer***
- 5s: ***inner***
- 6s: ***outer***
- ...

Specifically, why do both '***inner***' and '***outer***' occur at 3-second intervals?

This exercise builds upon the ideas covered in the previous one. However, instead of dealing with a single state, we now have a composite state Outer containing the state Inner.

The core concept to understand, is that taking a transition with as source the Outer state, results in both the Outer, as well as all its contained states being exited (in this case, Inner). Our simulation initially starts on $t = 0s$, and sets up a counter for Outer and Inner. At $t = 2s$, the Inner transition gets called due to the timer event passing, resulting in a clock reset for Inner and raising the ***inner*** event.

Next, on $t = 3s$, the Outer state receives the timer input event, enacts the corresponding transition, and resets the clock for *both* Outer, as well as Inner. Generally speaking, at $t = 3 \cdot n + 2s$, the nth ***inner*** event will fire (2s after the clock initalisation/reset), and then at $t = 3 \cdot (n - 1)s$, the ***outer*** event fires, resetting both Outer and Inner.

So in general, the Inner transition gets to fire once before the Outer transition resets its timer.

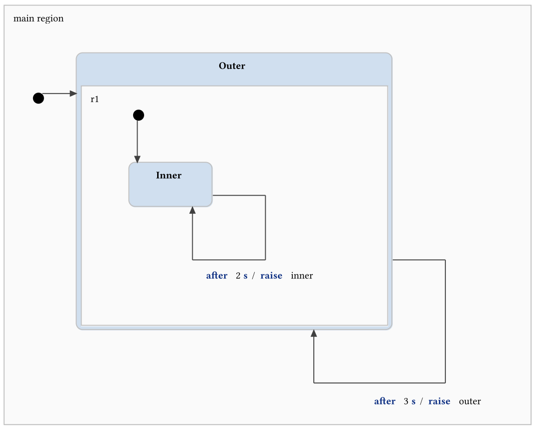

Figure 2: Exercise **B** state chart diagram

1.3. Exercise C

Question:

1. What behavior would you intuitively expect?
2. Run the simulation. What actually happens?
3. Replace the 'Temp' state by a 'Choice'-element (see Palette on the right). Does that fix it?

(1) Expected behaviour: As initially, $x = 0$, we expect that at $t = 1s$, the `after 1s` transition fires and we exit the Initial state to enter Temp. The transition action results in x being incremented by 1. Next, we *expect* that due to $x = 1$, the `[x==1]` transition is immediately satisfied, and the simulation ends up in the One state.

(2) Actual behaviour: While the first part matches with our expectations (everything up to the '*expect*'), we notice that the transition to One is **not taken** when we arrive at the Temp state, nor does the simulation ever move beyond the Temp state — despite $x = 1$.

(3) Adjustment fix: Replacing the state with a choice *does* fix this issue.

In general, when you are currently in a state, the only way to trigger a transition, is when an event is fired in the system and its conditions are fulfilled — be it events and/or guard conditions. Furthermore, if we arrive at the target state, our event is considered as consumed for that particular (orthogonal) region. For every active state, only one transition is usually taken per event.

Instead, by replacing the state Temp with a Choice, we can immediately consider all outgoing transitions of the choice[1]. Note also that the error in the adjusted diagram exists due to no default transition being specified for the Choice.

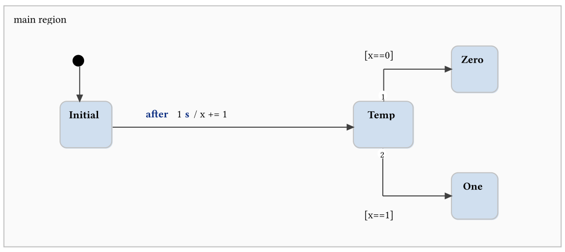

Figure 3: Exercise **C** state chart

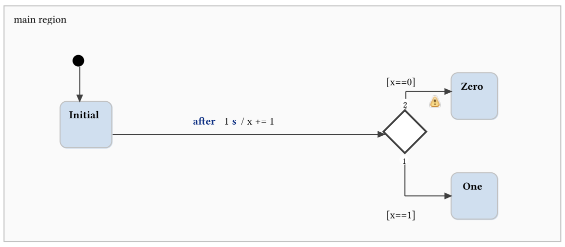

Figure 4: **Adjusted** exercise **C** state chart

[1]Itemis documentation

1.4. Exercise D

Question:

1. Run the simulation. What happens at time == 5s ?
2. Explain.

In this exercise, we combine our findings of Exercise C, with the orthogonal state design pattern. Note that we have the initial diagram of C in the left half of the Orthogonal state.

(1) As before, at $t = 1s$, we fire the `after` transition and move from Initial to Temp, with $x = 1$. For the next four seconds ($t \in]1, 5[\ s$), the left half of our Orthogonal state stays in Temp, while the right half is still in its initial state S. Finally, at $t = 5s$, the `after 5s` transition for r2/S fires, triggering the guarded transition `[x==1]` in r1/Temp.

(2) The moment that this timed event fires (internally called `S_time_event_0`), we trigger a statechart-wide RTC step, and place the timer event at the head of the FIFO event queue. Next, with this event at the front of the queue, we start a fair-step and visit every (orthogonal) region in a left-to-right, top-to-bottom fashion.[2]

As Orthogonal is currently an active state of our statechart, we visit each of its constituent regions in the manner described above, firing at most one transition, (optionally) using the timer event placed in the queue:

1. Region r1 (leftmost): as we are in Temp, we can follow one of two outgoing transitions: `[x==0]` or `[x==1]`. As $x = 1$, we can evidently take the latter, ending up in One. No other transitions may/can be taken.
2. Region r2 (rightmost): still at initial state S, we can use the timer event to follow the transition from S to T, as this fulfills its requirements.

Since no other events/effects are released, we cannot start another fairstep as the FIFO queue is empty (after consuming the event at the head of the queue), thus ending the RTC-step. Our simulation will continue indefinitely, staying invariant at states r1/One and r2/T.

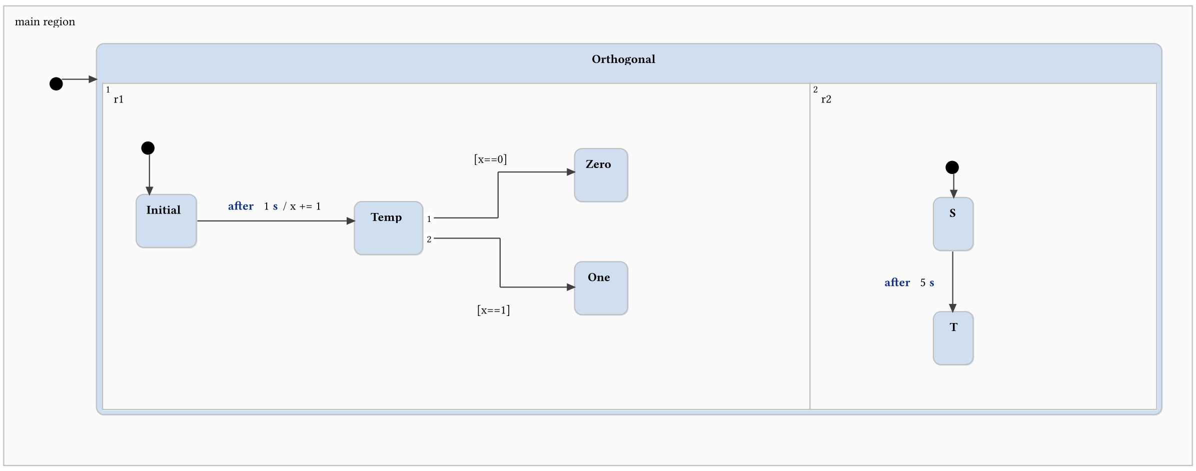

Figure 5: Exercise **D** state chart diagram

[2]ItemIS semantics as described by the project **assignment page**

1.5. Exercise E

Question: When running this model, the output trace will be as follows:

- 1s : *x*
- 1s : *y*
- 2s: *x*
- 2s: *y*
- …

Explain why:

1. Why does '*x*' only fire when '*y*' fires?
2. Why does '*x*' always occur before '*y*'?

Finally, this question builds upon the concepts discussed in Exercise D:

(1) As observed before, guarded transitions are only considered when we are in a RTC fairstep. Thus, to show that *x* only fires when *y* fires, we need to consider the following two cases:

- *Why x doesn't fire on initialisation:* the entry-state transition does *not* trigger a RTC step, as no effects were released on entering our initial state(s).
- *Why x cannot fire without y:* the only place where our system raises a RTC step, is in the `after 1s` timer event instantiated by StateB. When this RTC step triggers, there will (always) be two transitions that can be taken:
    - r1/StateA/ `[v==0]` as $v = 0$ per initialisation and stays constant
      Output: *x* *Target State*: StateA
    - r2/StateB/ `after 2s` due to the timed event that caused the RTC step
      Output: *y* *Target State*: StateB

Alltogether, since we always stay in the same states, and *y* triggering always means that *x* could have been triggered, we can safely say that there is a causal relationship where $y \overset{\text{implies}}{\Rightarrow} x$.

(2) Proving the second question is simpler: as the RTC step considers every region in a LTR/TB order, we can easily state that the transition r1/StateA/ `[v==1]` is *always* considered before r2/StateB/ `after 2s`, and their effects will also be released in that order.

Therefore, we can trivially state that *x* always get raised before *y*, or $x < y$.

As a final side-note: *x* raising does *not* cause a recursive loop for StateA as the event is marked as outgoing. If we were to instead mark it as an internal event, then each `raise x` would get added to the FIFO event queue, starting another fairstep.

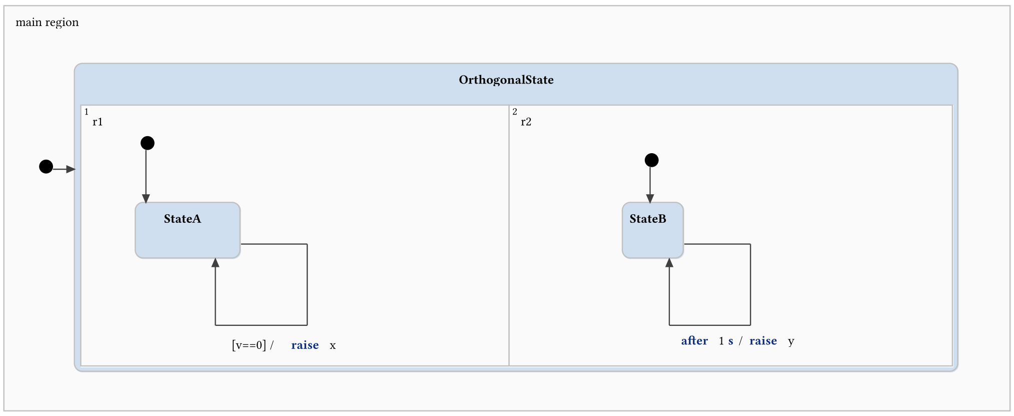

Figure 6: Exercise **E** state chart diagram

2. Traffic Light Controller

2.1. Requirements Discussion

This section briefly goes over and summarises all the design requirements for the traffic light.

- *"**Initially**, when the controller starts, the **traffic light** is set to **red**, and is in '**dumb mode**'."*
  - ⟶ Initial set of states: **red** and **dumb mode**
- *"In '**dumb mode**', the **traffic light cycles** through red, green, yellow and then red again. The light will be red for 2 seconds, green for 2 seconds, and yellow for 1 second.*
  Note: These time durations are not realistic, but they make it easier to (interactively) test your solution."
  - ⟶ Dumb mode: cycle between red, green and yellow with intervals (2s, 2s, 1s)
- *"The light can be **switched** from '**dumb mode**' to '**smart mode**' (and back) by **pressing** the **TOGGLE MODE-button**, and releasing the button quickly (pressed down for strictly **less than 2 seconds**)."*
  - ⟶ Toggle Dumb/Smart mode with pressing TOGGLE MODE button for max. < 2s
- *"In 'smart mode', the **behavior is identical** to 'dumb mode', **except** that when the **light is green**, and a **car is detected**, the **light will not turn yellow until 2 seconds have passed since the car-detection-event**. For instance, if the light has been green for 1.1 seconds, and a car is detected, then the light will stay green for a total duration of 3.1 seconds."*
  - ⟶ In Smart mode, green switches to yellow exactly 2 seconds after latest car-detection-event
- *"Even in 'smart mode', there is still a limit on how far the green period can be extended: the **total green light duration** must **never** be **greater than 5 seconds**."*
  - ⟶ In Smart mode, green *must* switch to yellow after 5 seconds (or less)
- *"There is a **status LED** that must be on if and only **if 'smart mode' is activated**."*
  - ⟶ In Smart mode enable status LED, in Dumb mode disable status LED
- *"Switching between 'dumb mode' and 'smart mode' must be seamless: the light should not be reset (e.g., to red). The only difference between these modes is whether new car-detection-events are responded to or not. For instance:"*
  - ⟶ The only difference between Smart and Dumb is accepting vs ignoring carDetected events and acting accordingly
- *"…if the light is in 'dumb mode', and has been red for 0.2 seconds, and then switches to 'smart mode', it should remain red for the remaining 1.8 seconds."*
  - ⟶ Switching dumb ⇝ smart: maintain times between light states
- *"…going from 'dumb mode' to 'smart mode', the traffic light simply starts responding to car-detection-events, possibly extending the current green period (up to a maximum of 5 seconds, even if the switch from 'dumb' to 'smart' happened at some point during those 5 seconds)"*
  - ⟶ Switching dumb ⇝ smart: car-detection-events extend the green period
- *"…going from 'smart mode' to 'dumb mode', the traffic light simply stops responding to car-detection events. If the current green period was already extended (due to car-detection-events before switching to 'dumb mode'), then the green period remains extended."*
  - ⟶ Switching smart ⇝ dumb: finish remaining green period
- *"Pressing and holding the **TOGGLE MODE-button** for **at least 2 seconds**, and then releasing it, starts or ends a '**police interrupt**'. A police interrupt completely **overrides the functionality of the traffic light**. A police interrupt **shows a blinking yellow light**. When a police interrupt starts, the **yellow light turns on immediately**, and **stays on for 500 milliseconds**, and then **off for 500 milliseconds**, then on again, etc."*
  - ⟶ Toggle Dumb/Smart mode for > 2s, toggle police interrupt
  - ⟶ Police interrupt start: override traffic light, cycle yellow light for 500ms on/off
- *"**During a 'police interrupt', 'smart mode' can still be turned on or off** in the usual manner (and the status LED should turn on and off accordingly). Of course, the chosen mode has **no impact** on the **traffic light's behavior** during a police interrupt."*
  - ⟶ Dumb/Smart mode can be toggled during police interrupt
- *"When a '**police interrupt' ends**, the **traffic light** is **reset** to a **red light** (for safety reasons), and **resumes normal behavior** ('dumb' or 'smart', depending on the active mode)."*
  - ⟶ Police interrupt end: set to red light, resume traffic light cycle

2.2. State Chart Implementation

In this section, we detail how we went about implementing the requirements into a statechart, and discuss some of the assumptions we made along the way.

2.2.1. Caveats

We enabled `@ParentFirstExecution` in the state chart code section, to prevent any related bugs. Otherwise, we might have to deal with the issue that children may be given priority in transition checking, given a specifc event.[3] Note that this made no difference for this assignment, but we keep it for the sake of completeness.

We made one **important assumption** for the system: when our traffic light controller is in 'smart' mode and the toggle mode button is pressed, 'carDetected' events are still able to influence the system. This means that, for instance, if the system is in smart mode and the toggle mode button has been pressed for 20s, but it has not been released yet, then 'carDetected' events would still be able to prolong a green light up to the 5s mark. Effectively, smart mode can only be disabled (i.e. dumb mode enabled) after a 'buttonPressed' event is raised and strictly less than 2s later a 'buttonReleased' event is raised.

2.2.2. Construction

With those caveats discussed, we can briefly move on to the actual implementation of our statechart In broad terms, our chart consists out of three (four) major components, each fulfilling specific parts of the requirements:

1. trafficLightsHandling: implements the logic of how the traffic light should change colors
    - NormalBehavior: how/when the traffic light should go red ⟶ green ⟶ yellow ⟶ … (R0, R1, R3, R4, R6, R9)
    - PoliceInterruptBehavior: how the traffic light should function when overridden (R7)
2. toggleButtonHandling: implements the logic of the dumb/smart/police mode toggling (R2, R5, R6, R7, R8)

These two handlers are enveloped by the trafficLight state, which serves as the top-most layer and interface of our controller. It defines the current status of the colors in our traffic light, as well the smart mode status (and its LED), and all the input/internal/output events used.

2.2.2.1. Normal Behavior

Our NormalBehavior, contains the logic for the red-green-yellow traffic light cycle, for both the dumb and smart modes. Each color of the traffic light corresponds to its own state, whose enter/exit actions automatically handle the updating of the traffic lights' color using output events (used with synchronization of the GUI or unit testing, for instance).

As per the requirements, the transitions from yellow and red are simply `after n seconds` timers, however, green is a bit more interesting to discuss:

- In **dumb mode**, we always transition after **two seconds**
- In **smart mode**, we transition
    - **two seconds** after the latest car being detected
    - OR after **five seconds** as a fallthrough transition

[3]For a more formal description as to why this was done, we refer to the YAKINDU tutorial (from August 2022) provided to us in the assignment. The relevant fragment starts around **01:41:50**. At **01:46:05**, it is described that we are encouraged to always use @ParentFirstExecution.

While the former is rather evident, we implement the latter by making use of a composite state, which allows us to essentialy make use of two separate timers: one timer that passes after two seconds (only being actively reset in smart mode when a new car arrives), and a second timer that fires after five seconds. This second clock *never* resets (unless green $\longrightarrow$ yellow happens), which ensures that the green-to-yellow transition is guaranteed to happen after the requisite duration has passed.

2.2.2.2. Interrupted Behavior

The PoliceInterruptBehavior is very straight forward: as long as the police overrides the traffic light behaviour, the light should cycle between yellow ON and yellow OFF every 0.5 seconds. As before, the entering/exiting of the states automatically handle updating the corresponding traffic lights' colors (and the individual light's ON or OFF states).

2.2.2.3. Toggle Button

Our manager between the two/three different behavioural patterns, is the toggleButtonHandling region. This exists separately from the trafficLightHandling area we discussed above, and merely influences the light cycling behaviour.

The main difficulty for this part came from properly differentiating between short ($< 2s$) and long ($> 2s$) presses, and then accordingly inducing the changes in traffic light behaviour.

Both behaviors related to the toggle mode button, require the button to be pressed for a set amount of time. They only differ in how long the button has to be pressed before it is released. So a buttonPressed event always results in one of the two behaviors (change mode or police interrupt).

The change mode behavior can be activated from the PressedShort state. When detecting a short press, we update the LED immediately and change the isSmartMode boolean, which accordingly disable/enables transitions within the NormalBehavior circuit.

If the button is not released for long enough, we transition to the PressedLong state instead. From here, the police interrupt behavior can be toggled. We raise an internal interrupt event, which starts/ends the police interrupt, changing from/to the PoliceInterruptBehavior state.

2.3. Unit Test Implementation

The implemented unit test checks requirements R8 and R9.

Requirement R8 concerns the toggling of the dumb/smart mode still working without issue during the police interrupt system state, and the related LED being updated properly.
We can test this by toggling the system **to smart mode during the police interrupt**. We then turn the police interrupt off again, and let the traffic light cycle to green. Next we fire some carDetected events and expect the green light to turn OFF after more than 2s. As an extension to this, the test then toggles the system to dumb mode again, cycles the traffic light to green once more, fires a carDetected event, and finally expects the light to turn off after the normal 2s mark despite the carDetected event.

Requirement R9 says that turning the interrupt off resumes the normal behavior at the red state. It also requires that the system dumb/smart mode is unaffected by the toggling of the interrupt.
In order to properly test this, we toggle the system mode to smart during the interrupt. After toggling off the interrupt, we check two things. **First**, verify that the system resumes at red (red turns ON) after stopping the interrupt. **Second**, we toggle the system mode and check that the LED gets turned off – the system gets set to dumb.

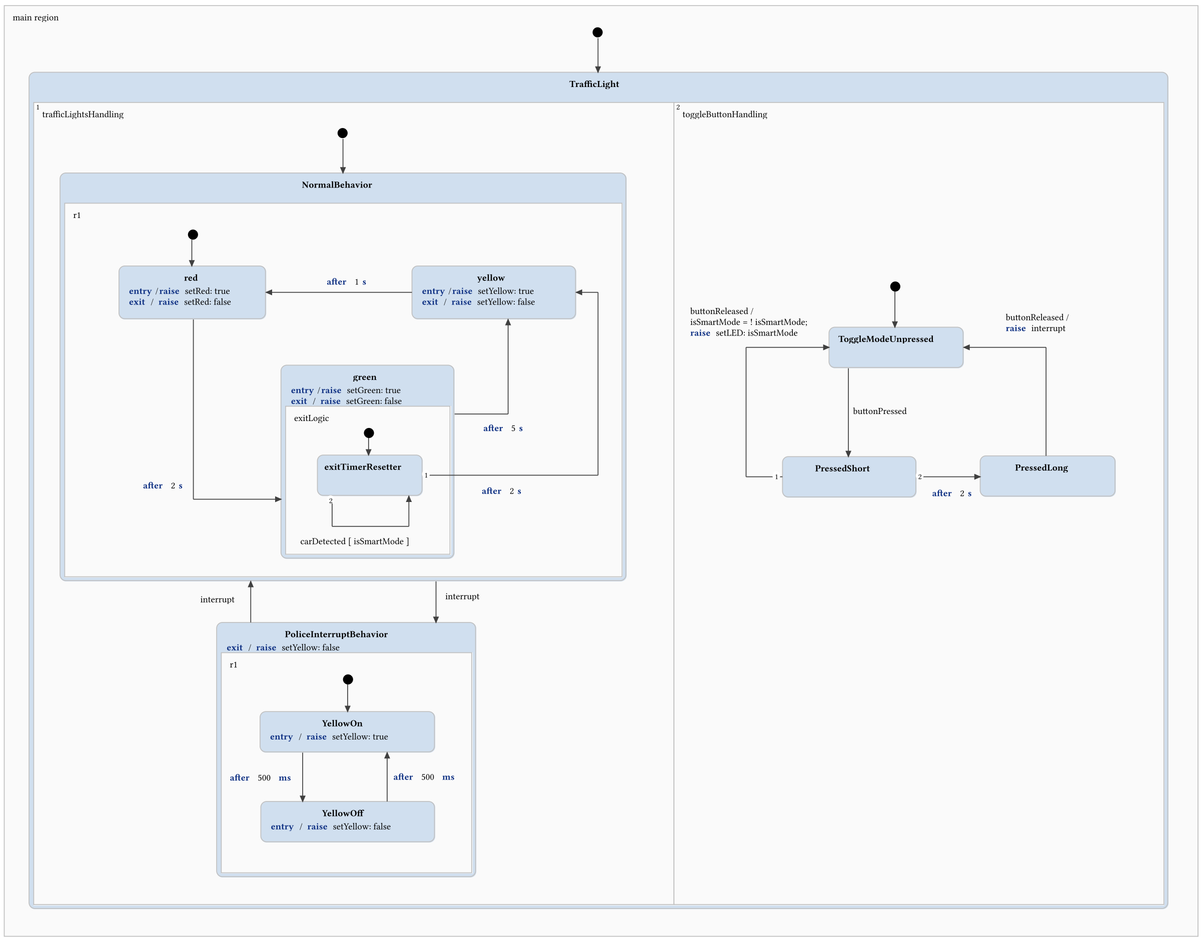

Figure 7: Full Traffic Light state chart