Info-F-410 Embedded Systems Design
Project report
Group A

Maazouz Mehdi, Boucher Nicolas

22 June 2019

1 Introduction

The goal of this report is to present and explain the conception, modelling and verification of a traffic light control system we built for the project of the class Embedded Systems Design. Furthermore, we also explain how we translated our model into a working python application.

2 Conception

The crossroad we modelled (presented in Fig1) include three roads for the cars and one road for firemen only, which can't be entered from the crossroad.
There are in total seven traffic lights (one being reserved for the firemen), three pedestrian traffic lights and three push-buttons. Each push-button corresponds to one pedestrian traffic light.
We assume that all traffic laws are respected for our crossroad model to be correct.

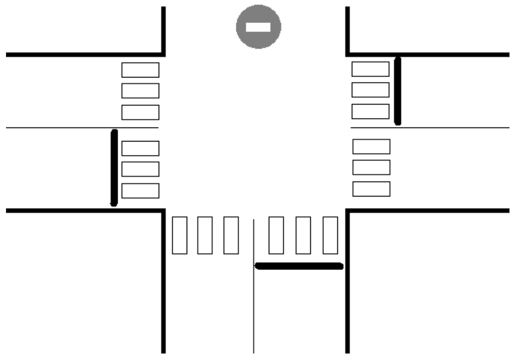


Figure 1: The crossroad

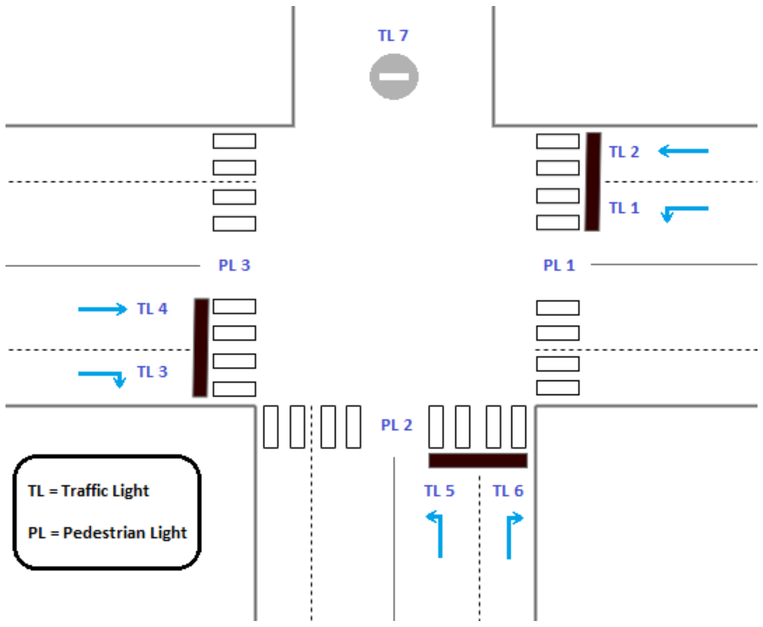


Figure 2: Traffic lights and pedestrian lights numerical ids

2.1 Traffic lights

The traffic lights determine whether a lane may let cars drive through the crossroad or not, and cycle between different states: red, green, and orange.
Seven traffic lights are built into our system, their distribution can be seen in Fig2.

2.2 Push-buttons and pedestrian lights

All push-buttons are activated in a probabilistic manner (probability of .2 at each time step), so as to simulate random pedestrian wishing to cross the street. The push-buttons are in a default state of deactivated, and get activated when a pedestrian presses it. Once a push-button is activated, it will have an impact on the future states of the traffic lights due to a concept of phases depending on the states of the push-buttons.
If two push-buttons or more are in an activated state at the same time, then once the current phase is finished, all pedestrian lights will go green and all traffic lights will be red. We made it this way because not allowing the third non-requested pedestrian crossing to go green wouldn't allow any cars to pass anyway.

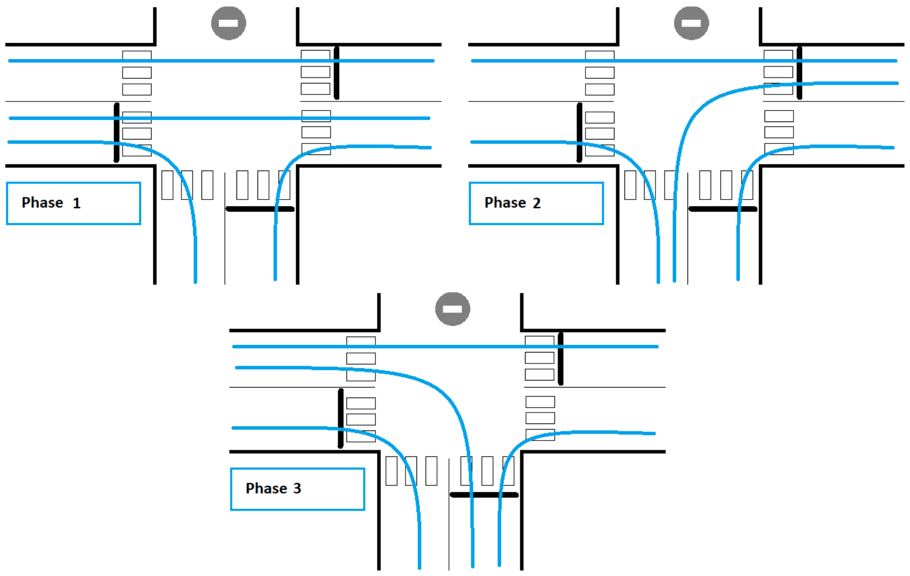


Figure 3: Phases 1, 2, 3

2.3 Phases

The phases are the main states of the whole crossroad. They represent all the possible and safe combinations of traffic lights states, pedestrian light states, push button states and firemen states. Here, by safe we mean that the cross-road doesn't allow car crashes or pedestrians risking getting ran over ($\neq$ safety properties). This is done by reducing the amount of possible phases to those that don't make car flows intersect with each other or go through currently used pedestrian crossings.
By following this logic, the effective number of phases of our model amounts to twelve.
The phases depicted in Fig3 represent the cars-only phases.
Some of the traffic lights depend on the current phase. For example TL4 (see Fig2) is green when the current phase is phase 1. As a consequence, the TL1 is set to red in order to avoid accidents. When the current phase is the phase 2, TL1 is set to green and TL4 is set to red.
Initially the system is set to the phase 1.

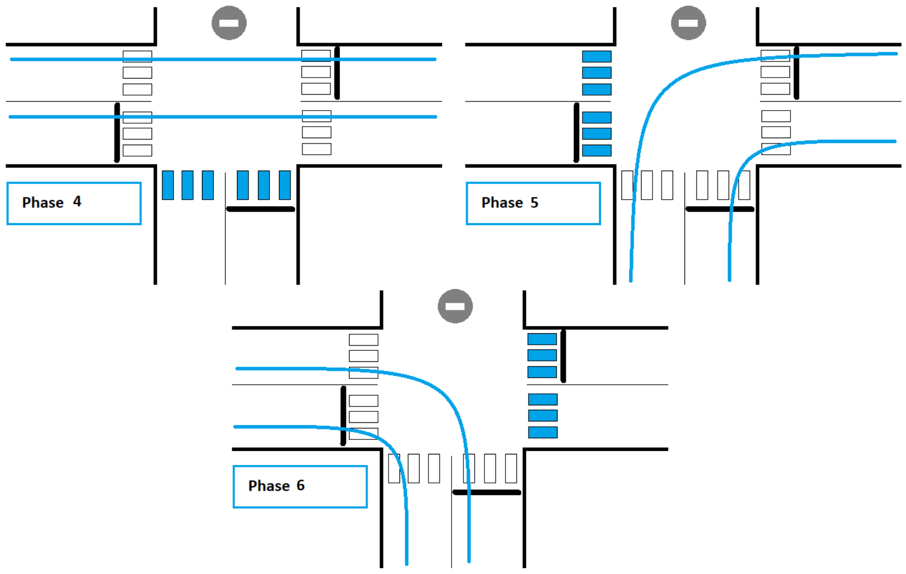


Figure 4: Phases 4, 5, 6

The phases 4 to 6 (see Fig4) are a variation of phases 1 to 3, each containing one green pedestrian light. We state they're a variation of the first three because they always include the car flow that goes through the middle of the crossroad that defines the phase they're based on.
As they're a variation, we do not allow the system to go from a phase to its own pedestrian variation, the same being true the other way around (e.g., phase 1 can't go to phase 4 directly as it would be redundant: the TL 3 and 6 that are missing from phase 4 appear both in phase 2, and each of them appear in phase 5 and 6).
Fig 5 demonstrates how the phases 1 to 6 can succeed one another.

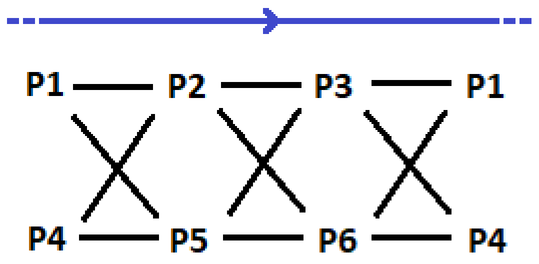


Figure 5: Phases 1 to 6 main cycle

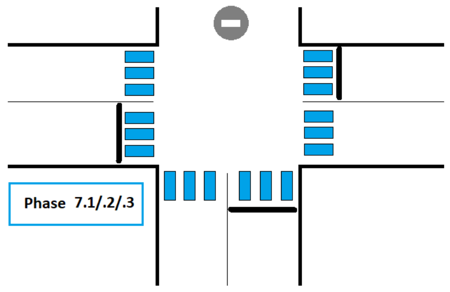


Figure 6: Phases 7.1, 7.2, 7.3

2.4 Pedestrian-only phase

Whenever two push-buttons are pressed at the same time, or more specifically end up both activated at some point, we state that in the following phase all traffic lights should turn to red and all pedestrian lights should turn to green.
We agreed on doing so because only allowing the two pedestrian lights corresponding to the activated push buttons to turn green and leaving the last one red didn't make sense to us as no car could drive through the crossroad with two of its entrances being blocked by the pedestrians.
Of course, this phase is also used when all of the three push-buttons are in the activated state.
The phase we use for modeling this is the abstract phase 7, subdivided in the three phases 7.1, 7.2 and 7.3 (Fig 6). We use three phases instead of one so we can simulate a memory in our system: we need to know in which phase the system was before allowing the pedestrian-only phase to activate, so we can then go back to the correct phase once phase 7 is done. We used the same kind of logic for the next concept we'll introduce to the system: firemen.

2.5 Firemen

In order to add complexity to our model, we have added a firemen exit that enters the crossroad. In concrete terms, at each state of our system, there is a small probability (would be fixed at .02 for a realistic simulation, but was fixed at .1 for the sake of the demo) that the firemen need to exit their fire station through the crossroad.
As a consequence of this, at the next phase following an incoming firemen signal all of the traffic lights and the pedestrian lights have to turn red except for the TL7 that allows the firemen to enter the crossroad. The newly added abstract phase that expresses this behaviour is Phase 8 and is subdivided in 3 concrete phases, which we named respectively **phase_8_1**, **phase_8_2** and **phase_8_3** (see fig 7).
Splitting the phase in three allows us to know which phase was in effect prior to phase 8. We need to know this so we can then go back to a *correct* next phase

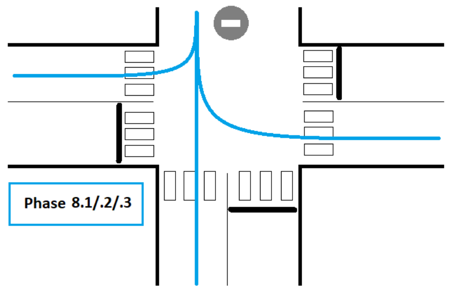


Figure 7: Phases 8.1, 8.2, 8.3

once the firemen phase is done. If we only had one firemen phase, we would have no way of knowing which phase to go next, and could potentially induce erroneous behaviours such as skipping some phases entirely (like looping from 1 to 8 to 1 to 8 forever).

2.6 States of the phases

All our phases go through the exact same cycle of states (see fig 8). Once a phase state transition from deactivated to orange, it must complete its entire cycle before any another phase is able to transition from deactivated to orange as well.
Please note that although the states orange, red and green of this diagram have a relation with the states orange, red and green of the traffic lights, these concepts are different.
The relation between the phases, traffic lights, pedestrian lights, push buttons and firemen are as follow:

- Transitions between phases depend on the state of the push-buttons and the state of the firemen. These transitions happen when the current phase's state transitions from finish to deactivated.
- Transitions between traffic light states (red, orange, green) depend on the state of the current phase only. When a phase goes to orange, it sets all the traffic lights and pedestrian lights that need to be orange in this specific phase to orange. Same goes for the red state, then for the green state. The diagrams in fig3, fig4, fig6 and fig7 represent the complete state of the system when the phases are in their green state (when all traffic and pedestrian lights are in their own correct state in respect to the current phase).
- Transitions between push-button states (activated, deactivated) happen probabilistically for the transition deactivated -> activated, but the transition activated -> deactivated only happens when their designated pedestrian light's state is set to green.

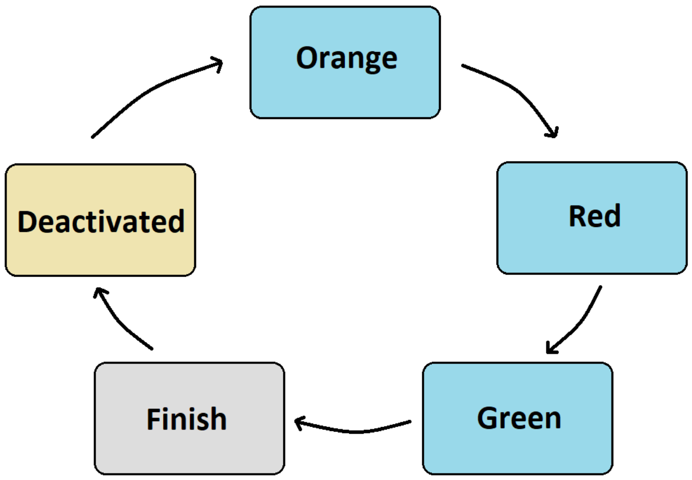


Figure 8: Phase states cycle

- Transitions between the firemen states happen probabilistically as well when waiting for a firemen signal, but the state goes back to no signal once the firemen traffic light (TL 7) is set to green.

2.7 Transitions between the phases

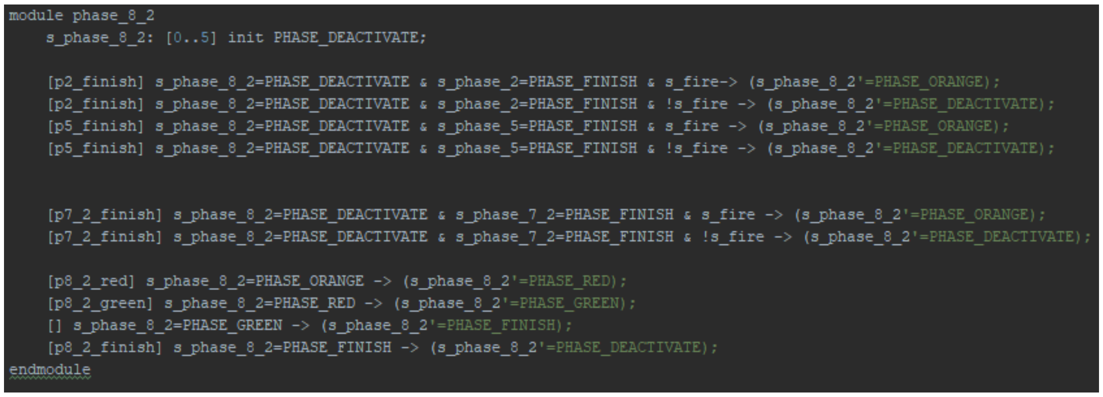
Hereafter are all the transition conditions between the phases of the system, which we used to create our model with the Prism model checking tool.

- Current phase = phase 1
  - No firemen and no pedestrian button pushed
    - Next phase = phase 2
  - No firemen and only the pedestrian button 3 pushed
    - Next phase = phase 5
  - No firemen and more than one pedestrian button pushed
    - Next phase = phase 7_1
  - firemen
    - Next phase = phase 8_1
- Current phase = phase 2
  - No firemen and no pedestrian button pushed
    - Next phase = phase 3
  - No firemen and only the pedestrian button 1 pushed

        - Next phase = phase 6
    - No firemen and more than one pedestrian button pushed
        - Next phase = phase 7_2
    - firemen
        - Next phase = phase 8_2
- Current phase = phase 3
    - No firemen and no pedestrian button pushed
        - Next phase = phase 1
    - No firemen and only the pedestrian button 2 pushed
        - Next phase = phase 4
    - No firemen and more than one pedestrian button pushed
        - Next phase = phase 7_3
    - firemen
        - Next phase = phase 8_3
- Current phase = phase 4
    - No firemen and no pedestrian button pushed
        - Next phase = phase 2
    - No firemen and only the pedestrian button 3 pushed
        - Next phase = phase 5
    - No firemen and more than one pedestrian button pushed
        - Next phase = phase 7_1
    - firemen
        - Next phase = phase 8_1
- Current phase = phase 5
    - No firemen and no pedestrian button pushed
        - Next phase = phase 3
    - No firemen and only the pedestrian button 1 pushed
        - Next phase = phase 6
    - No firemen and more than one pedestrian button pushed
        - Next phase = phase 7_2
    - firemen
        - Next phase = phase 8_2
- Current phase = phase 6

  - No firemen and no pedestrian button pushed
    - Next phase = phase 1
  - No firemen and only the pedestrian button 2 pushed
    - Next phase = phase 4
  - No firemen and more than one pedestrian button pushed
    - Next phase = phase 7_3
  - firemen
    - Next phase = phase 8_3

- Current phase = phase 7_1
  - No firemen
    - Next phase = phase 2
  - firemen
    - Next phase = phase 8_1
- Current phase = phase 7_2
  - No firemen
    - Next phase = phase 3
  - firemen
    - Next phase = phase 8_2
- Current phase = phase 7_3
  - No firemen
    - Next phase = phase 1
  - firemen
    - Next phase = phase 8_3
- Current phase = phase 8_1
  - Pedestrian button 3 not pushed
    - Next phase = phase 2
  - Pedestrian button 3 pushed
    - Next phase = phase 5
- Current phase = phase 8_2
  - Pedestrian button 1 not pushed
    - Next phase = phase 3
  - Pedestrian button 1 pushed
    - Next phase = phase 6



```
module phase_8_2
    s_phase_8_2: [0..5] init PHASE_DEACTIVATE;

    [p2_finish] s_phase_8_2=PHASE_DEACTIVATE & s_phase_2=PHASE_FINISH & s_fire-> (s_phase_8_2'=PHASE_ORANGE);
    [p2_finish] s_phase_8_2=PHASE_DEACTIVATE & s_phase_2=PHASE_FINISH & !s_fire -> (s_phase_8_2'=PHASE_DEACTIVATE);
    [p5_finish] s_phase_8_2=PHASE_DEACTIVATE & s_phase_5=PHASE_FINISH & s_fire -> (s_phase_8_2'=PHASE_ORANGE);
    [p5_finish] s_phase_8_2=PHASE_DEACTIVATE & s_phase_5=PHASE_FINISH & !s_fire -> (s_phase_8_2'=PHASE_DEACTIVATE);


    [p7_2_finish] s_phase_8_2=PHASE_DEACTIVATE & s_phase_7_2=PHASE_FINISH & s_fire -> (s_phase_8_2'=PHASE_ORANGE);
    [p7_2_finish] s_phase_8_2=PHASE_DEACTIVATE & s_phase_7_2=PHASE_FINISH & !s_fire -> (s_phase_8_2'=PHASE_DEACTIVATE);

    [p8_2_red] s_phase_8_2=PHASE_ORANGE -> (s_phase_8_2'=PHASE_RED);
    [p8_2_green] s_phase_8_2=PHASE_RED -> (s_phase_8_2'=PHASE_GREEN);
    [] s_phase_8_2=PHASE_GREEN -> (s_phase_8_2'=PHASE_FINISH);
    [p8_2_finish] s_phase_8_2=PHASE_FINISH -> (s_phase_8_2'=PHASE_DEACTIVATE);
endmodule
```

Figure 9: Prism module representing phase 8.2

- Current phase = phase 8_3
  - Pedestrian button 2 not pushed
    - Next phase = phase 1
  - Pedestrian button 2 pushed
    - Next phase = phase 4

3 Modelling

3.1 Module creation

Every concept we talked about so far pretty much translate into Prism modules. Fig9 shows an example of such a module. The first line of a module determines the states in which the module can be in. In the case of a phase, those states are in deactivated, orange, red, green, finished, while in the case of a push-button it would be only activated, deactivated.
All the other lines are either conditional or probabilistic conditional transitions (our model is deterministic), which are the prism equivalent of the conditions enunciated in the previous point ”Transitions between the phases”. However, in Prism, when we write a condition to transition from one state to another, we also have to add a condition that allows the state to stay the same. This effectively doubles the amount of conditions needed. However, we only considered the conditions that did cause a change of state for our Python implementation.
The labels in brackets on the left of most conditions are synchronisation labels. All conditions (and their respective transitions) having the same label need to be executed synchronously. This is how Prism allowed us to model the dependencies between our modules.

4 Properties

In order to check the correctness of our model, we have verified a lot of properties. It is important to notice that our properties will be true because of the construction of our model. Below you will see some of properties we have verified, however a complete list containing all properties will be added at the end of this report. Remark we have used CTL and PCTL to describe our properties.

4.1 Safety

First of all, we have verified that our system cannot go into a bad state whatever the path followed.

- Here we have checked that incompatible traffic lights are not green simultaneously with the traffic light 1
  - A [G (tl1=GREEN =¿ (tl4=RED & tl5=RED & pl1=RED & pl2=RED & tl7=RED))]
- An another example of this kind of property. When the firemen have their traffic light at green, no cars or pedestrians may go through the crossroad
  - A[G (tl7=GREEN =¿ (pl1=RED & pl2=RED & pl3=RED & tl1=RED & tl2=RED & tl3=RED & tl4=RED & tl5=RED & tl6=RED))]
- With Prism, we can easily check the likelihood to get a deadlock is equal to zero
  - $\texttt{P}_{<=0}$ `[F "deadlock"]`

4.2 Liveness

Here we have verified the good behaviour of our model.

- First we wanted to be sure that all of our traffic lights could be green infinitely often. Here are some examples of the verified properties.
  - A [G F tl1 = GREEN]
  - A [G F tl2 = GREEN]
  - A [G F tl3 = GREEN]
  - A [G F tl4 = GREEN]
  - A [G F tl5 = GREEN]
  - A [G F tl6 = GREEN]
- However, the property below is not verified.
  - A [G F tl7 = GREEN]

- Indeed, there exists a path where no fireman have to enter the crossroad. As a consequence the traffic light 7 never gets to the green state.
  - E [F G tl7 = RED]
  - More specifically
    - E [G tl7 = RED]
  - But the probability to stay in this path infinitely tends to 0. So we can use PCTL language to check the good behaviour of our system
    - P¿=1 [G F tl7 = GREEN]
- Some of our traffic lights can be red infinitely often. This property is verified for the traffic light 1, 4, 5 and 7.
  - A [G F tl1 = RED]
  - A [G F tl4 = RED]
  - A [G F tl5 = RED]
  - A [G F tl7 = RED]
- For the others traffic lights, there exist a path where these are always green. An example of this path could be **phase 1** followed by **phase 2** followed by **phase 3** followed by **phase 1** followed by ....
  Please note that their initial values is red. So removing the finally (F) does not work.
  - E [F G tl2 = GREEN]
  - E [F G tl3 = GREEN]
  - E [F G tl6 = GREEN]
- In addition, we have verified properties about the pedestrian lights. We verified that there exists a path where no pedestrian pushes their button, so the pedestrian lights still have the red value.
  - E [G pl1 = RED]
  - E [G pl2 = RED]
  - E [G pl3 = RED]
- Our pedestrian lights could be red infinitely often. Either these still are red or these get the red value after one or several phases.
  - A [G F pl1=RED]
  - A [G F pl2=RED]
  - A [G F pl3=RED]

4.3 Fairness

In this section, we have checked our model is fair.

- First, we have tested that if pedestrians push the pedestrian button 1, then they will eventually go into the crossroad.
  - A [G ((pb1=PB_ACTIVATE) =¿ (F pl1=GREEN))]
- Below, you can see that if the firemen want to go into the crossroad, the next phase will be a firemen phase. Regardless of we were in a cars and/or a pedestrians phase before.
  - A [G (fire=PB_ACTIVATE & (phase_1 = PHASE_FINISH | phase_4 = PHASE_FINISH | phase_7_1 = PHASE_FINISH) =¿ (X (phase_8_1=PHASE_ORANGE)))]
- In addition we wanted to be sure that whatever the path followed, if the current phase is a firemen phase then the next phase will give the hand to a car and/or pedestrian phase.
  - A [G (phase_8_1=PHASE_FINISH =¿ (X (phase_8_1=PHASE_DEACTIVATE & phase_8_2=PHASE_DEACTIVATE & phase_8_3=PHASE_DEACTIVATE)))]

4.4 Miscellaneous properties

Here we have added some other properties we thought were interesting to verify.

- First we have checked the absolute priority of the firemen. Even if one or several push buttons are enabled, the next phase will be a firemen phase.
  - P$_{>=1}$ [G (((fire = PB_ACTIVATE) & (pb1 = PB_ACTIVATE | pb2 = PB_ACTIVATE | pb3 = PB_ACTIVATE) & (phase_1 = PHASE_FINISH)) => (X (phase_8_1 = PHASE_ORANGE)))]
- As a reminder, by construction we have prevented that a phase following a firemen phase would be a pedestrian phase. Here you can see the property that verifies this.
  - A [G ((phase_8_1 = PHASE_FINISH) => (X (phase_7_1 = PHASE_DEACTIVATE)) & (X phase_7_2 = PHASE_DEACTIVATE] & (X phase_7_3 = PHASE_DEACTIVATE]))))]

  We also wanted to know what was the maximal probability to have the traffic light 5 to be red for at least 4 phases.
  - P*max* =? [tl5=RED U>= 20 tl5=GREEN] = 0.28404...
- Here we have the same property but for phase 3
  - P*max* =? [phase_3=PHASE_DEACTIVATE U>= 20 phase_3=PHASE_ORANGE] = 0.18445...

5 Translation of the model in Python

5.1 Main direction

Prism, although being a great tool for creating and verifying probabilistic models, does not provide a code generation functionality such as what some non-probabilistic model checkers like Lustre do.
We had limited options to translate our model into another language such as Python. What we agreed on doing was to create the Python code to be as close to the Prism code in its architecture as we possibly could. We wanted to write the code so as to emulate how it would have been potentially written had it been generated either by Prism (assuming such a feature existed) or by a theoretical external tool.
This approach resulted in a python code that was rather voluminous, and that seemed to have a repetitive structure resembling the original Prism code.

5.2 Module translation

Each Prism module translates to one Python class. Each of those classes have a state which will contain an integer value referring to the right state value and an *update_self(self)* method.
The phase classes will additionally include update methods that update all the objects of the system (traffic lights, push-buttons, pedestrian lights and firemen) as well as a method that will check whether the phase should stop and let another phase start.
This way, we emulate the synchronisation labels of Prism: every time a phase updates, we call the *update_self(self)* method of all other objects to check whether the update of the phase should have an impact on those objects. If it does, those impacted objects' states are updated accordingly. In Prism, this is done by using the synchronisation feature.
To allow for more readability, we split those classes into different files, regrouping all the traffic lights in a file, all the push buttons, and so on. These files are:

- phases.py
- trafficlights.py
- pedestrianlights.py
- pushbuttons.py
- firemen.py

5.3 Module manager

The main method of the program can be found in *implementation.py*.
This file features a few classes such as the ones allowing the drawing of the

crossroad to be displayed onto the console, but also the IntersectionManager class.
This class has the crucial job of providing an access from all objects to all the other objects, effectively making the update of any "impacted objects" possible (as seen in the last point). Indeed, as explained earlier, the different modules have dependencies on other modules. To model these dependencies in Python, we needed a way to access the references of the objects representing those modules.
The IntersectionManager contains all the references to the instantiated objects of the modules classes. Once the manager is set up, it is then passed as an attribute of all those objects as well, making it very easy to access any object from any other object.
In fact, we could have renamed this class to "ModuleInstanceReferences". But as it also fulfills the job of returning the current phase, we decided to call it manager.

5.4 Loss of mathematical integrity

Of course, we didn't prove that our python model was mathematically the same as our Prism model. This is the price to pay for creating our own translated model instead of having it generated by a mathematically proven tool. But as no such tool existed, we carried on with our own implementation.
It is also why we tried to mimic Prism's way of implementing models: the closer we were from Prism's architectural style, the lower were our chances to introduce big differences between the two models. As much as we tried to find some, we haven't found any dissimilarities between the two model executions. This however is no proof of their equality, although making such a proof would be way out of the scope of this project.

5.5 Main loop logic

Once the program has instantiated all the module classes and set up the manager, it starts its main loop. The only activated phase is the first one, all others are in the deactivated phase. This means the *current phase* returned by the manager will be phase 1 at the start of the loop until there's a phase change.
The logic is the following (illustrated in fig10):
The current phase first updates itself. This means it will test all the conditions that could lead to a state change, then apply the state change when it finds a true condition. These conditions are the translated conditions that can be found in the Prism module of that phase.
If the current phase changes to the *PHASE_FINISH* state, it also finds all the potential phases that can succeed it and calls the *update_self()* function on them. We decided to update all potential successor to mimic the way Prism does it: when the phase changes its state, it also checks all the other modules that are synchronized on that specific state change. Another way we could have done this would have been to directly find which phase should succeed the current

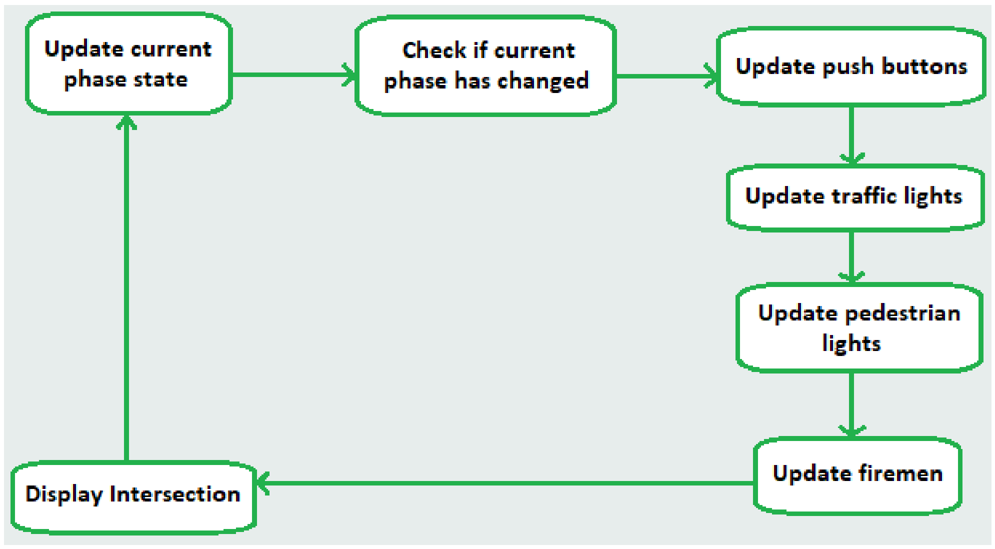
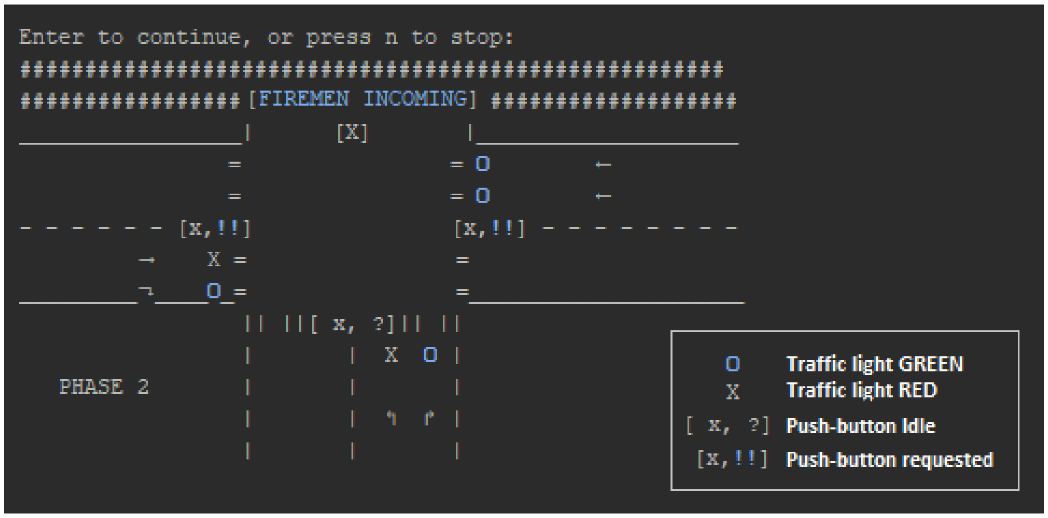


Figure 10: Main loop logic

phase based on the current state of the system, but this didn't satisfy our goal of keeping our logic as close as Prism's.
Once the current phase has updated its state and potentially started the new current phase, we update all the push-buttons, traffic lights, pedestrian lights, and firemen objects. This is done by calling the several *update_related_object* methods found in the current phase instance. As you may observe, these methods are the same in all our phase classes. We should've either put those only once in the manager class, or filtered the update_related methods calls to only call the *truly* related objects. This is something we wanted to change, but forgot to do so. It stills works, because the non-related instances will never update themselves as none of their update conditions will be true. But it is however wasting computations.
The whole system has now been updated and is printed in the console. Once that is done, the loop starts over and continues for every press of the enter key, until the user enters the character ”n” to stop the program.

5.6 Visual output of the program

Please note that the result shown in fig11 (without the legend in the right corner) was obtained by running the code in the Pycharm console - works on both windows and ubuntu. When we tried to run it in our distribution's consoles, it would break the output and make it practically unreadable.
No parameters are needed for running the code. The states of all modules in the last 10 timesteps is shown below the crossroad display.
We used Python 3.7 and numpy 1.16.4



```
Enter to continue, or press n to stop:
######################################################
#################[FIREMEN INCOMING] ###################
_________________|      [X]       |_____________________
                =                = O        ←
                =                = O        ←
- - - - - - [x,!!]               [x,!!] - - - - - - - -
          →    X =               =
_________¬____O_=                =______________________
                 || ||[ x, ?]|| ||
                 |       |  X  O |
   PHASE 2       |       |       |
                 |       |  ↰  ↱ |
                 |       |       |
```

```
    O     Traffic light GREEN
    X     Traffic light RED
[ x, ?]   Push-button Idle
 [x,!!]   Push-button requested
```

Figure 11: Result legend

5.7 Annexes

5.7.1 Python code

The entire Python source code and the pism model can be found in the src folder joined to this report.

5.7.2 Complete list of verified properties

- Safety
  - A [G tl1=GREEN => (tl4=RED & tl5=RED & pl1=RED & pl2=RED & tl7=RED)]
  - A [G tl2=GREEN => (pl3=RED & tl7=RED)]
  - A [G tl3=GREEN => (pl3=RED & pl2=RED & tl7=RED)]
  - A [G tl4=GREEN = (tl1=RED & tl5=RED & pl1=RED & pl3=RED & tl7=RED)]
  - A [G tl5=GREEN => (tl1=RED & tl4=RED & pl2=RED & pl3=RED & tl7=RED)]
  - A [G tl6=GREEN => (pl1=RED & pl2=RED & tl7=RED)]
  - A [G tl7=GREEN => (pl1=RED & pl2=RED & pl3=RED & tl1=RED & tl2=RED & tl3=RED & tl4=RED & tl5=RED & tl6=RED)]
  - P<= 0 [F "deadlock"]
- Fairness

  - A [G ((pb1=ACTIVATE) => (F pl1=GREEN))]
  - A [G ((pb2=ACTIVATE) => (F pl2=GREEN))]
  - A [G ((pb3=ACTIVATE) => (F pl3=GREEN))]
  - A [G ((fire=ACTIVATE) => (F tl7=GREEN))]
  - P>= 1 [G phase_8_1=PHASE_FINISH => (X (phase_8_1=PHASE_DEACTIVATE & phase_8_2=PHASE_DEACTIVATE & phase_8_3=PHASE_DEACTIVATE))
    ]
  - P>= 1 [G phase_8_2=PHASE_FINISH => (X (phase_8_1=PHASE_DEACTIVATE & phase_8_2=PHASE_DEACTIVATE & phase_8_3=PHASE_DEACTIVATE))
    ]
  - P>= 1 [G phase_8_3=PHASE_FINISH => (X (phase_8_1=PHASE_DEACTIVATE & phase_8_2=PHASE_DEACTIVATE & phase_8_3=PHASE_DEACTIVATE))
    ]
  - P>= 1 [G phase_7_1=PHASE_FINISH => (X (phase_7_1=PHASE_DEACTIVATE & phase_7_2=PHASE_DEACTIVATE & phase_7_3=PHASE_DEACTIVATE))
    ]
  - P>= 1 [G phase_7_2=PHASE_FINISH => (X (phase_7_1=PHASE_DEACTIVATE & phase_7_2=PHASE_DEACTIVATE & phase_7_3=PHASE_DEACTIVATE))
    ]
  - P>= 1 [G phase_7_3=PHASE_FINISH => (X (phase_7_1=PHASE_DEACTIVATE & phase_7_2=PHASE_DEACTIVATE & phase_7_3=PHASE_DEACTIVATE))
    ]
  - P>= 1 [G F (fire=ACTIVATE & (phase_1=PHASE_FINISH | phase_4=PHASE_FINISH | phase_7_1=PHASE_FINISH) & (X (phase_8_1=PHASE_ORANGE)))
    ]
  - P>= 1 [G F (fire=ACTIVATE & (phase_2=PHASE_FINISH | phase_5=PHASE_FINISH | phase_7_2=PHASE_FINISH) & (X (phase_8_2=PHASE_ORANGE)))
    ]
  - P>= 1 [G F (fire=ACTIVATE & (phase_3=PHASE_FINISH | phase_6=PHASE_FINISH | phase_7_3=PHASE_FINISH) & (X (phase_8_3=PHASE_ORANGE)))
    ]

- Liveness
  - A [G F tl1 = GREEN]
  - A [G F tl2 = GREEN]
  - A [G F tl3 = GREEN]
  - A [G F tl4 = GREEN]
  - A [G F tl5 = GREEN]
  - A [G F tl6 = GREEN]
  - E [F G tl7 = RED]

- E [G tl7 = RED]
- A [G F tl1 = RED]
- A [G F tl4 = RED]
- A [G F tl5 = RED]
- A [G F tl7 = RED]
- E [F G tl2 = GREEN]
- E [F G tl3 = GREEN]
- E [F G tl6 = GREEN]
- E [G pl1 = RED]
- E [G pl2 = RED]
- E [G pl3 = RED]
- A [G F pl1=RED]
- A [G F pl2=RED]
- A [G F pl3=RED]
- P¿=1 [G F tl7 = GREEN]
- P>= 1 [G F phase_1=PHASE_ORANGE]
- P>= 1 [G F phase_2=PHASE_ORANGE]
- P>= 1 [G F phase_3=PHASE_ORANGE]
- P>= 1 [G F phase_4=PHASE_ORANGE]
- P>= 1 [G F phase_5=PHASE_ORANGE]
- P>= 1 [G F phase_6=PHASE_ORANGE]
- P>= 1 [G F phase_7_1=PHASE_ORANGE]
- P>= 1 [G F phase_7_2=PHASE_ORANGE]
- P>= 1 [G F phase_7_3=PHASE_ORANGE]
- P>= 1 [G F phase_8_1=PHASE_ORANGE]
- P>= 1 [G F phase_8_2=PHASE_ORANGE]
- P>= 1 [G F phase_8_3=PHASE_ORANGE]

• Miscellaneous

  - P$_{>=1}$ [G (((fire = PB_ACTIVATE) & (pb1 = PB_ACTIVATE | pb2 = PB_ACTIVATE | pb3 = PB_ACTIVATE) & (phase_1 = PHASE_FINISH)) => (X (phase_8_1 = PHASE_ORANGE)))]
  - A [G ((phase_8_1 = PHASE_FINISH) => (X (phase_7_1 = PHASE_DEACTIVATE)) & (X phase_7_2 = PHASE_DEACTIVATE] & (X phase_7_3 = PHASE_DEACTIVATE]))))]
  - Pmax =? [tl5=RED U>= 20 tl5=GREEN] = 0.28404...
  - Pmax =? [phase_3=PHASE_DEACTIVATE U>= 20 phase_3=PHASE_ORANGE] = 0.18445...