Traffic Light Control

INFO-F-412 – Formal Verification

Jamal Ben Azouze, Marien Bourguignon, Nicolas De Groote,
Simon Picard, Arnaud Rosette, Gabriel Ekanga

May 29, 2015

Contents

1 Introduction

1.1 First Idea

The goal of this project is to model a system composed of traffic lights and a crossroad with four roads and four crosswalks, and then applying formal verification techniques to check its compliance to some properties. Each road would have three traffic lanes: one to turn left, one to turn right and one to continue right ahead. Each traffic lane would have his own traffic light that can turn red, orange or green. The system would detect the number of cars queuing on each lane. The crosswalks would also have their own traffic light that can turn green or red and push buttons for pedestrians to notify the controller. The controller would then have to respect properties, to avoid accident and traffic jam.

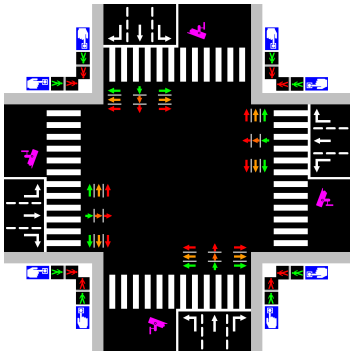


Figure 1: Visual representation of the first draft

1.2 Refined Idea

The system was made too complex, and so impractical upon checking properties. The number of states was too high. To simplify the system and decrease the number of possibilities, each road were changed to only have one lane and one traffic light. The cars will still be able to turn and the system will still be able to know where every car wants to go. This limited the number of states while still keeping all aspects from the initial idea. Nevertheless this modification was not sufficient. The model was therefore simplified again. The actual model now represents a crossroads in a T shape with three roads and one crosswalk.The orange color was also removed. The system have push buttons for pedestrians to notify that they want to cross. There also are detectors to inform the controller on how many cars are waiting at the red lights and where they want to go. This allows to assign some kind of priorities, while still taking into account the pedestrians.

Assumptions made during the modeling phase:

- The traffic lights do not have orange lights and the cars can stop instantaneously
- Every entity respect the highway code (no car or pedestrian will cross a red traffic light)
- At any time a car can arrive and join a queue of cars at a traffic light

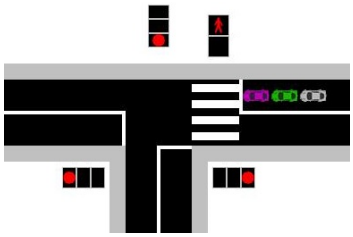


Figure 2: Visual representation of the final idea

- At any time a pedestrian can push the button to cross the street (more pedestrians would only join the first one and cross the street together as a group)
- A pedestrian will always cross the street in less than a fixed time
- A car will always cross the crossroads in less than a fixed time

1.3 Connection With Embedded System

This project from the Formal Verification course is combined with the project from Embedded System Design. These two projects are complementary. From the embedded point of view, we had to modelize an environment, and generate a controller based on that model to avoid unwanted situations through winning strategies, computed with timed games. For the formal verification course, a more formal approach was taken. We also had to modelize the system, but also to verify that the model was compliant to some properties. The two models being based on the same problematic that is managing crossroads with traffic lights, parts of one can be seen in the other, thus the connection.

1.4 Benefits of Formal Verification

The most widespread technique to verify if a system behaves the way it should is testing. One will test a system by running it with changing inputs and observe the outcomes. If a system fails to do its task, a flaw is found. Unfortunately, this process is not always practical. For example, no sane person will try to debug a space rocket by pushing the launch button. Indeed, just economically speaking, it could be a huge disaster if some parts of it get damaged. Moreover, it is usually impossible to test a system by feeding it with all possible inputs. Formal verification brings (partial) solutions to these problems, by allowing to check if the model of a system is compliant to some properties.

As explained in section 1.2, the main goal is to manage a crossroad with traffic lights while avoiding accidents. Lives being at stakes, the conventional testing approach is impractical. Formal verification, and more precisely model checking will allows us to make sure no bad states (such as traffic collision or car/pedestrian collision) can be reached. This will be done thanks to timed automatons and temporal logic, that are used to check if temporal properties are respected.

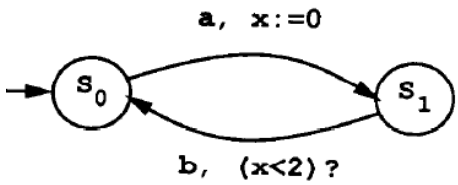
2 Prerequisite

In this section, important subjects that need to be understood upfront will be discussed. First, timed automaton will be explained, followed by the notion of temporal logic and temporal properties, and finally the tools used during the implementation of our project

2.1 Timed Automaton

Timed automata are finite state automata that run on infinite words as input, accepting timed languages. Time languages are composed of timed words, themselves composed of symbol coupled with time. Compared to other formalisms, such as ω-automaton, real time constrains can be taken into consideration.

An example of timed automaton can be seen on figure 3. S_0 is the initial state and x is a clock. When the symbol 'a' is read, the transition to S_1 is taken, the clock is reset to 0. From there, it can go back to S_0 by reading the symbol 'b', only if the clock is below 2. Of course, the clock is monotonically increasing. For more information on the subject, see [2].



$$\{((ab)^\omega, \tau) \mid \forall i.\ (\tau_{2i} < \tau_{2i-1} + 2)\}.$$

Figure 3: A timed automaton with its language. Image from [2]

2.2 Temporal Logic and Properties

Temporal logic is obtained when the notion of time and logic are combined together. It allows statements such as "Tomorrow, it must be true" to be made, which was not possible with first order logic for example. In our situation, two kinds of temporal logic are of interest: linear temporal logic (LTL) and computation tree logic (CTL).

These two logics are used to specify properties that a system has to comply to, meaning that given a model and a formula from CTL or LTL, we can check if the properties hold. LTL and CTL do not have the same expressivity, so one must choose the right logic according to its need.

The main difference between LTL and CTL is that LTL is evaluated on infinite traces of a model, and CTL allows, from a state, to check if a property holds for all branches from that state of the computation tree, or just for one (using $\forall$ and $\exists$). Also, it is usually faster to check if a CTL formula hold within a system than a LTL one. For more information on the subject, see [5].

The following temporal properties can be expressed using either CTL or LTL:

- **Safety**: the system never reaches an unwanted state.
- **Liveness**: the system makes progress while avoiding deadlocks, which is a situation where two or more objects are waiting infinitely for the other to finish.
- **Persistence**: From a certain point, the system never leaves a set of given states.
- **Fairness**: Everyone is at some point satisfied (cannot be expressed using LTL).

2.3 Game Theory and Timed Games

Definition 1. *Game is loosely defined to be a competition in which opposing parties attempt to achieve some goal according to prespecified rules.[6]*

Game theory is the study of how competitors interact in a situation that can be seen as a game, with the different outcome according to who win. In our situation, one could say the controller of the traffic lights is the first player, and the cars/pedestrians are the second player (environment). The goal of the controller will be to find a winning strategy to that game, being a sequence of moves that will inevitably lead to the wanted outcome.

Timed games are games where time is part of the rules, bringing a real time dimension. A game of chess is a timed game, where each player has to make a move before a time limit. Similarly, a crossroad is a game where the controller has to keep the cars and pedestrians safe, while taking into account time constraint on the traffic lights.

2.4 Tools

Checking that a model is compliant to a property manually is a cumbersome process, borderline impossible if the size of the model is too high. Tools are here to help us do that verification in a reasonable time.

2.4.1 Spin

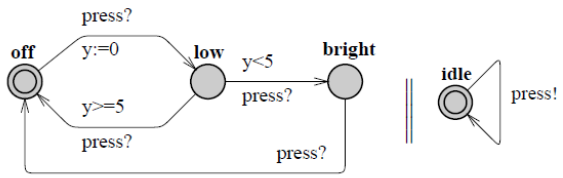
Definition 2. *Spin is a popular open-source software verification tool [...]. The tool can be used for the formal verification of multi-threaded software applications.[1]*

Spin is really efficient for growing and shrinking numbers of processes, using the advantages of multi-core computers. The tool is designed to scale well, and to handle large problem sizes efficiently.

This could be handy for our project to be able to use a great computational power. Nevertheless, Spin does not handle timers nor does it can generate a winning strategy. For these reasons, Spin was not used for this project.

2.4.2 Uppaal

Uppal is a tool used for verification of real time systems that can be modeled as network of timed automata. It further enhances them by allowing use of bounded variables, structured data types, user defined functions and channel synchronization. A system is composed of one ore more timed automata, synchronized through channel. The modellisation is done using the graphical interface, and the elements stated before.



(a) Lamp. (b) User.

Figure 4: Timed automata used by Uppaal. Image from [4]

An example of a simple model can be seen on picture 4. In this model, two timed automata cohexist in the same model. The right part represents a user potentially pressing a button (and so sending a signal), on the left part one can see the modelization of a lamp, that can be turned on or off upon receiving signals. If the user presses the button twice fast enough, he/she can control the brightness of the lamp. For more information on the subject, see [4].

2.4.3 Uppaal Tiga

Tiga is an extension of Uppaal, allowing timed games (hence TiGa). The main difference with Uppaal is the possibility of having transitions taken nondeterministically by the environment. It also allows computing a winning strategy and a controller synthesis. For more information on the subject, see [3][4].

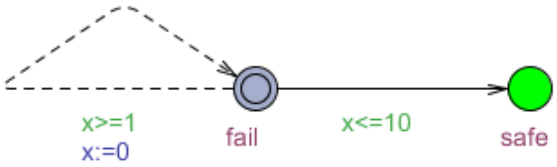


Figure 5: Timed automata used by Uppaal TiGa. Image from demo folder of Uppaal Tiga

One can see on picture 5 a system made of a single timed automaton. The dashed transition is controlled by the environment.

3 Modeling

Modeling is an important part of the process. This part expresses the real world in terms of automatons. So the real world which is not formal is translated into a formal model which can be used to verify properties. We have to be careful when modeling the system because we have to make compromises between having a model with a lot of details which is close to the reality but might be too complex to understand and verify and a less detailed more simple model that would not reflect the reality precisely.

3.1 Actors

In this section, we will describe the different actors who intervene in our system.

Traffic Lights : Traffic lights are the main actors in this project. They give the authorization to cars and pedestrians to cross the roads at given instants. Their role is to schedule the events. Lights have three possible states in real world but we assume that they only have two states (green and red) in our model because it reduces the complexity of the system and the orange state does not bring interesting actions. Indeed, an orange light has the same meaning as that of the red one in the real world : a car has to wait when a light is orange.

Pedestrians : Pedestrians come to a traffic light, they can push a button and wait that the corresponding traffic light passes from red to green in order to cross the road. When a traffic light is green, it stays green during a certain amount of time.

Cars : Cars are the last actors of our system. They arrive to traffic lights and they wait until they are allowed to enter the crossroad. There are several possible scenarios for them. When their traffic light turn green, each car can follow different directions. We assume that a car cannot enter the crossroad and exiting it by the same road. For example, a car coming from the east road cannot leave the crossroad by the east road. We also assume that we have a way to detect car arrivals and car destination.

3.2 Timed Automaton

In order to formally model the system, we use several automatons. More precisely, we use timed automatons from the Uppaal tool. Some of them are part of the environment, the others are part of the controller.
The environment is composed of automatons which are not controllable by the system. For example, car arrivals are unpredictable and uncontrollable, so they belong to the environment.
The actors which implement the solution for avoiding crashes belong to the controller. They interact (control) with the environment in order to avoid crashes.

3.2.1 Environment

Before implementing a strategy for avoiding crashes between different cars or between cars and pedestrians, the environment has to be formally modeled.
The environment is composed of pedestrians, cars and lights. More precisely, the environment

contains generators and lights. The generators are used to randomly generate cars or pedestrians. There is a generator for each road and each crosswalk taking place in the crossing. Concretely, there are three car generators and one pedestrian generator.

Pedestrian generator : This automaton is presented in figure 6. There is one instance of the pedestrian generator automaton in the system : one for the crosswalk which take place on the east road.
It is made of four states and the initial one is the state where there is no pedestrian requesting to cross the crosswalk. Once a pedestrian pushes on the button situated on the crossing light, the automaton enters the PushButton state and notifies the controller. It stays in that state until the pedestrian light turns green. If the light goes green, the pedestrian will cross the road during an undetermined number of time which cannot exceed a constant called PEDESTRIAN_CROSSING_TIME. Since the crossing time for a pedestrian has an upper bound, the automaton will stay in the Cross state exactly during this maximum number of time before returning the initial state.
Once the automaton is waiting in the PushButton state, it cannot receive other requests from other pedestrians. It is due to the fact that we consider pedestrians as a group of pedestrians because all the pedestrians cross the road in the same time unlike cars. So the automaton only have to know if there is at least one pedestrian waiting to cross the road. It does not care if there is one or more pedestrians. The same reasoning leads us to only consider one pedestrian generator for the two sides of the crosswalk. This is done because if the light is green for one side of the crosswalk, it is also green for the other side.
This generator can only generate a fixed number of pedestrian request because the number of possible states has to be reduce for the model checking. Indeed, if the number of generations is not limited, the number of states is infinite and checking properties on the model take an infinite number of time.

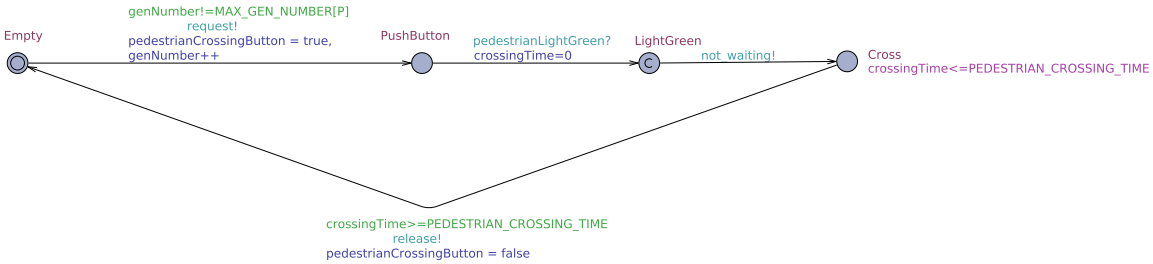


Figure 6: Timed automaton of pedestrian generator

Car generator : A second automaton that belongs to the environment is the car generator automaton described in figure 7. This automaton simulates car arrivals at a particular crossing light. There are three instances of this automaton in the system : one for each road (east, south and west).
Each car generator has a queue storing cars waiting to cross the crossroad because, unlike the generator for pedestrians, this generator have to distinguish each car waiting because cars do not cross together at the same time. The queue is not infinite, it is bounded. The size of this queue is two because the time to check properties is too important with more than two locations in the

queue and the model is too simple with less locations.
In the initial state, there is no cars waiting to cross the crossroad. In this state, a car can arrive at any moment if the queue is not full. Once the light becomes green, the car which is at the first position in the queue enters the crossroad and leaves it by one of the other roads. When the car finishes crossing, the other cars in the queue move forward. If the light is still green, another car enters the crossroad. Once the light becomes red, the automaton returns in the initial state. If some cars are still in the queue, they wait until the light turns green, otherwise the generator wait for car arrivals.
The direction (left, up or right) of each car is stored in the queue in order to provide information to the controller of the crossing lights.
In this model, we assume that drivers are smart, so a car does not wait if the light is green. We also assume that the crossing time for cars is upper bounded, so the crossing of a car take at most a constant time.
Like pedestrian generator, this generator is only allowed to generate a fixed number of cars because the model is too complex in terms of possible states if the number of generated cars is unbounded. In the model, the instance of the car generator for the east road can generate two cars, the one for the south road is allowed to generate one car and the other one for the west road is also allowed to generate one car.

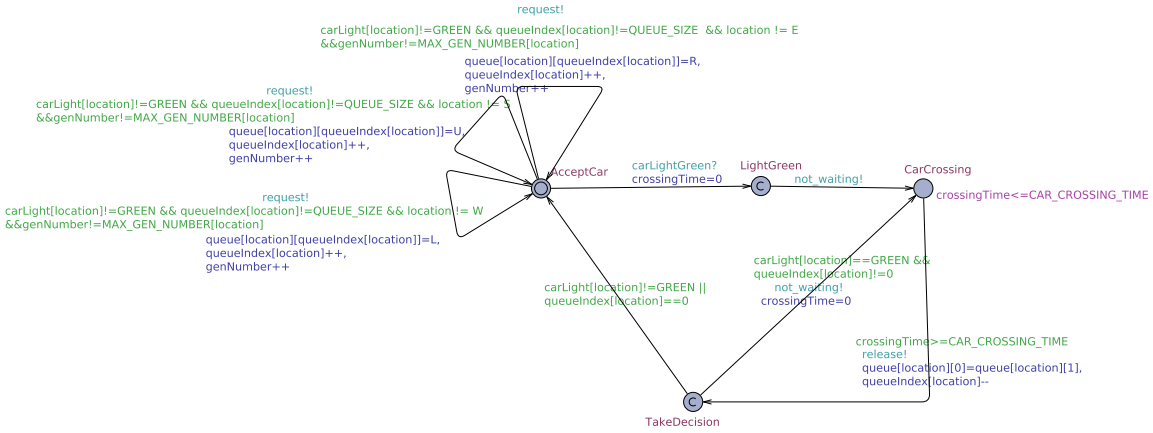


Figure 7: Timed automaton of car generator

3.2.2 Controller / Strategy

In this section, the automatons which implement a strategy to avoid crashes will be described. These automatons take decisions according to the state of the environment and interact with this environment in order to control it and avoid crashes. The controller is composed of two automatons : one handling requests from the generators and controlling the waiting time of each pedestrian or car, the other one implementing a strategy to avoid crashes.

Timer : This automaton is described in figure 8. The goal of this automaton is to wait for requests from the environment. There are four instances of such an automaton, one for each generator.

Each generator interacts with its own timer in order to notify the controller about changes in the environment. Some of the notifications are done through channels which require synchronization between automatons. So timer automatons are sub-processes of the main controller (light controller) that are useful for handling changes from the environment without overwhelming the main controller, they are also useful to not freeze the environment because some of the notifications are done through channels which require the controller always being able to receive a notification.
The other goal of this part of the controller is to measure the waiting time of each car or pedestrian in a generator in order to not let them wait too much time. Once the waiting time for a generator exceed a maximum constant, this generator becomes priority and the light for this generator has to turn green immediately. This is done to avoid a car or a pedestrian waiting infinitely for crossing.
At the beginning, the automaton is in the NotWaiting state, it means that there is no car or pedestrian waiting in its associated generator. Once it receives a notification from the environment (a car has arrived or a pedestrian has pushed a button), it goes to a waiting state and starts a clock to measure the waiting time. If the waiting time exceeds the maximum waiting time, it goes to another state and notify the main controller through a variable.
In order to notify the light controller about cars or pedestrians waiting too much, the timer automaton manages a queue of generators which exceed the max waiting time.
The timer automaton can only receive a maximum of two requests because queues in generators have a length of two. There is no queue for pedestrian generator but it is not a problem for the timer, it simply means that the timer associated with the pedestrian generator will never reach states where there are two items waiting.

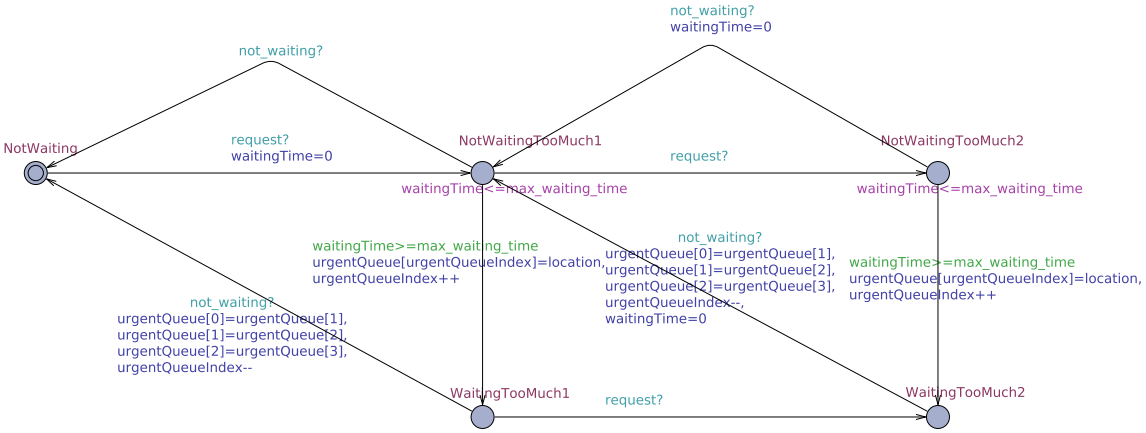


Figure 8: Timed automaton of timer

Light controller : Light controller is the main automaton of the controller. It implements a strategy to avoid crashes between pedestrians and cars or between several cars. It also tries to maximize the crossing rate of the crossroad and minimize the waiting time. There is a single instance of this automaton in the system. The automaton is graphically described in figure 9.
Implementing a strategy to let pedestrians cross the road and cars enter the crossroad without crashes is an example of the critical section problem. In this case, the critical section is crossing the crossroad. There are several changes from the initial critical section problem : for example, it

is possible to have several car crossing at the same time if some conditions are respected. The goal here is to find a scheduler which is able to avoid crashes, let all the cars or pedestrians wanting to cross crossing without waiting indefinitely, minimize the waiting time and maximize the crossing rate.
The strategy is made of two main policies : a policy when there is no generator exceeding the maximum waiting time (called the regular strategy or regular policy) and another when there is at least one generator which exceed the maximum waiting time (called the urgent strategy or urgent policy).
The regular strategy acts like this : if there is no request for crossing, all the lights stay red. If there is only one request for crossing, the light turns green for the location where the request come from. If there are several requests at the same time, there are priorities assigned to each road and to the crosswalk. Pedestrian requests have the highest priority, then cars coming from east, then cars coming from south, cars coming from west have the lowest priority. So, for example, if a car is waiting at the east road and a car is waiting at the west road and they cannot cross at the same time because a crash is coming, the car from the east will cross first, then the car from west. The regular policy allows multiple crossing at the same time if it is not possible to have a crash. For example, if a car is coming from east and want to go to west and a car is coming from west and want to go to south, the controller will let the two cars crossing at the same time because no crash is coming. The strategy also allows several cars from the same location crossing in a row. For example, if two cars are waiting at the east light, they can cross in a row.
The other policy is the urgent one. The controller enters this policy only if there is at least one generator which exceeds its maximum waiting time. Generators that exceeds their maximum waiting time are stored in a queue by their location (east, south, west or pedestrian). The first generator which reaches its maximum waiting time is stored at the first location in the queue, the second one at the second location, etc. Then, the controller empties the queue in a FIFO (first in first out) order. This mechanism is useful to avoid pedestrians or cars waiting indefinitely to cross.

4 Model checking

The purpose of creating formal models using automatons is that it allows to formally check if the system complies with given properties. This is normally achieved by creating temporal formulas that have to be verified by the system but Uppaal only uses a subset of TCTL. The different properties are therefore using the BNF-grammar for the requirement specification language used in the verifier of Uppaal which is really similar to CTL.

4.1 Tested properties

In Uppaal, to operator leads to :

```
p --> q
```

is equivalent to :

```
A[] (p imply A<> q)
```

4.1.1 Liveness properties

1. A pedestrian will never wait indefinitely to cross the road

   ```
   PedestrianGeneratorEast.PushButton --> PedestrianGeneratorEast.Cross
   ```

   A pedestrian that pushes the button will always finally obtain the possibility to cross. He has the guarantee that he will receive the permission to cross.

2. A car will never wait indefinitely to enter the crossroad

   (a) A car wanting to enter the crossroad from the east road will never wait indefinitely

   ```
   (CarGeneratorEast.AcceptCar&&queueIndex[E]!=0) -->
           CarGeneratorEast.CarCrossing
   ```

   (b) A car wanting to enter the crossroad from the south road will never wait indefinitely

   ```
   (CarGeneratorSouth.AcceptCar&&queueIndex[S]!=0) -->
           CarGeneratorSouth.CarCrossing
   ```

   (c) A car wanting to enter the crossroad from the west road will never wait indefinitely

   ```
   (CarGeneratorWest.AcceptCar&&queueIndex[W]!=0) -->
           CarGeneratorWest.CarCrossing
   ```

   For each car accepted in the queue of any direction, the car will finally cross the roads. (This is similar to the pedestrian pushing the button and that will eventually cross the road.)

3. No Deadlock :

   ```
   A[] not deadlock
   ```

   This property expresses the fact that the system will never freeze.

4.1.2 Safety properties

1. The pedestrian light will never be green at a wrong time :

   ```
   A[] not
   (pedestrianLight == GREEN &&
       (carLight[E] == GREEN ||
           (carLight[W] == GREEN && queue[W][0] == U) ||
           (carLight[S] == GREEN && queue[S][0] == R)
       )
   )
   ```

   For all possibilities, we never have the pedestrian light green and the car light at the same point (east) green. And there never is the pedestrian light green and a car that wants to cross the crosswalk with his traffic light green. That is to say, while the crosswalk light is green, there never is :

- the east light green
- the west light green with the first car in the queue there that wants to go right ahead
- the south light green with the first car in the queue there that wants to turn right

2. No pedestrian will be hit by a car :

```
A[] not
(PedestrianGeneratorEast.Cross &&
    (CarGeneratorEast.CarCrossing ||
        CarGeneratorWest.CarCrossing &&
        queue[W][0] == U ||
        CarGeneratorSouth.CarCrossing &&
        queue[S][0]==R
    )
)
```

   There never is a pedestrian crossing and a car in motion going where the crosswalk stands (east). Meaning that we never have the sitution shown in figure 10 where a pedestrian is crossing and:

   - A car from the east is crossing
   - A car from the west is crossing that goes right ahead
   - A car from the south is crossing that turn right

3. No combination of lights allowing car collisions :

```
A[] not
(
    (carLight[S] == GREEN && queue[S][0] == L &&
        ((carLight[W] == GREEN && queue[W][0] == U) ||
            (carLight[E] == GREEN && (queue[E][0] == L || queue[E][0] == U))
        )
    ) ||

    (carLight[S] == GREEN && queue[S][0] == R &&
    carLight[W] == GREEN && queue[W][0] == U) ||

    (carLight[W] == GREEN && queue[W][0] == U &&
        ((carLight[E] == GREEN && queue[E][0] == L) ||
            (carLight[S] == GREEN && (queue[S][0] == L || queue[S][0] == R))
        )
    ) ||

    (carLight[W] == GREEN && queue[W][0] == R &&
    carLight[E] == GREEN && queue[E][0] == L) ||
```

```
    (carLight[E] == GREEN && queue[E][0] == L &&
        ((carLight[S] == GREEN && queue[S][0] == L) ||
            (carLight[W] == GREEN && (queue[W][0] == U || queue[W][0] == R))
        )
    ) ||

    (carLight[E] == GREEN && queue[E][0] == U &&
    carLight[S] == GREEN && queue[S][0] == L)
)
```

For every possible path followed by a car, the light will never be green if it allows cars from different directions to cross their paths (therefore avoiding possible collisions).

4. No car collision :

   Different cars in motion from different origins cannot have paths that cross.

   The following situations cannot happen:

   - A car crossing from the east (whether it turns or not) with a car crossing from the south turning left
   - A car crossing from the east and going left with a car crossing from the west whether it turns or not. (See example in figure 11)
   - A car crossing from the west and a car crossing from the south (whatever direction they go to)

```
A[] not (
    (
        CarGeneratorSouth.CarCrossing && queue[S][0] == L &&
        (
            (
                CarGeneratorWest.CarCrossing && queue[W][0] == U
            ) || (
                CarGeneratorEast.CarCrossing &&
                (
                    queue[E][0] == L || queue[E][0] == U
                )
            )
        )
    ) || (
        CarGeneratorSouth.CarCrossing && queue[S][0] == R &&
        CarGeneratorWest.CarCrossing && queue[W][0] == U
    ) || (
        CarGeneratorWest.CarCrossing && queue[W][0] == U &&
        (
            (
                CarGeneratorEast.CarCrossing && queue[E][0] == L
            ) || (
```

```
                    CarGeneratorSouth.CarCrossing &&
                    (
                        queue[S][0] == L || queue[S][0] == R
                    )
                )
            )
        ) || (
            CarGeneratorWest.CarCrossing && queue[W][0] == R &&
            CarGeneratorEast.CarCrossing && queue[E][0] == L
        ) || (
            CarGeneratorEast.CarCrossing && queue[E][0] == L &&
            (
                (
                    CarGeneratorSouth.CarCrossing && queue[S][0] == L
                ) || (
                    CarGeneratorWest.CarCrossing &&
                    (
                        queue[W][0] == U || queue[W][0] == R
                    )
                )
            )
        ) || (
            CarGeneratorEast.CarCrossing && queue[E][0] == U &&
            CarGeneratorSouth.CarCrossing && queue[S][0] == L
        )
    )
```

5 Encountered problems

A major problem encountered in this project is the explosion of the number of time needed to verify properties when the number of possible states in the system increase. It is due to the fact that the time needed to assert a property is exponential in the number of possible states in the system. So when we are modelling the real world with automatons we always have to be careful with the number of states in the automaton. We always have to try to reduce this number as far as possible without making the model trivial and loosing too much information about the real world.
In order to address this issue, we reduced the number of states by removing non-interesting ones. For example, we removed the orange state in the lights because this state is only useful to model the fact that a light takes time to turn from green to red which is not really the most interesting thing to model.
We also had to reduce the number of possible generations for each generator. At the beginning, a generator was able to generate an infinite number of cars or pedestrians. With an infinite number of generation, the time to check properties on the model was too high. So, we had to bound the number of generations for each generator. The maximum number of generations is now : two for the east car generator, one for the south car generator, one for the west car generator and two for the pedestrian generator.

6 Conclusion

By doing this project, we now have a better understanding of formal verification methods especially model checking technique. We experimented the development of a system driven by model checking. We now see the differences between the different steps of such a process and the benefits offered by model checking.
When we develop a system driven by model checking techniques, we have to understand the real world and exhibit what are the specifications of this world at first. We also have to decide what are the properties that we want to verify on the real world. Once this first step is done we can start to model the system in a more formal way with automatons and to express properties with logic formulas. Thanks to the properties express in logic, the real world express in automatons and a tool to verify properties on the model, we can assert whether our system is correct or not. If our system is correct, model checking helps us to have the sureness that our system is correct. If it is not correct, model checking helps us to highlight what is wrong in the system. Model checking tools often provide a way to show counter-example if a formula is not satisfied by the model which help us to correct the model.

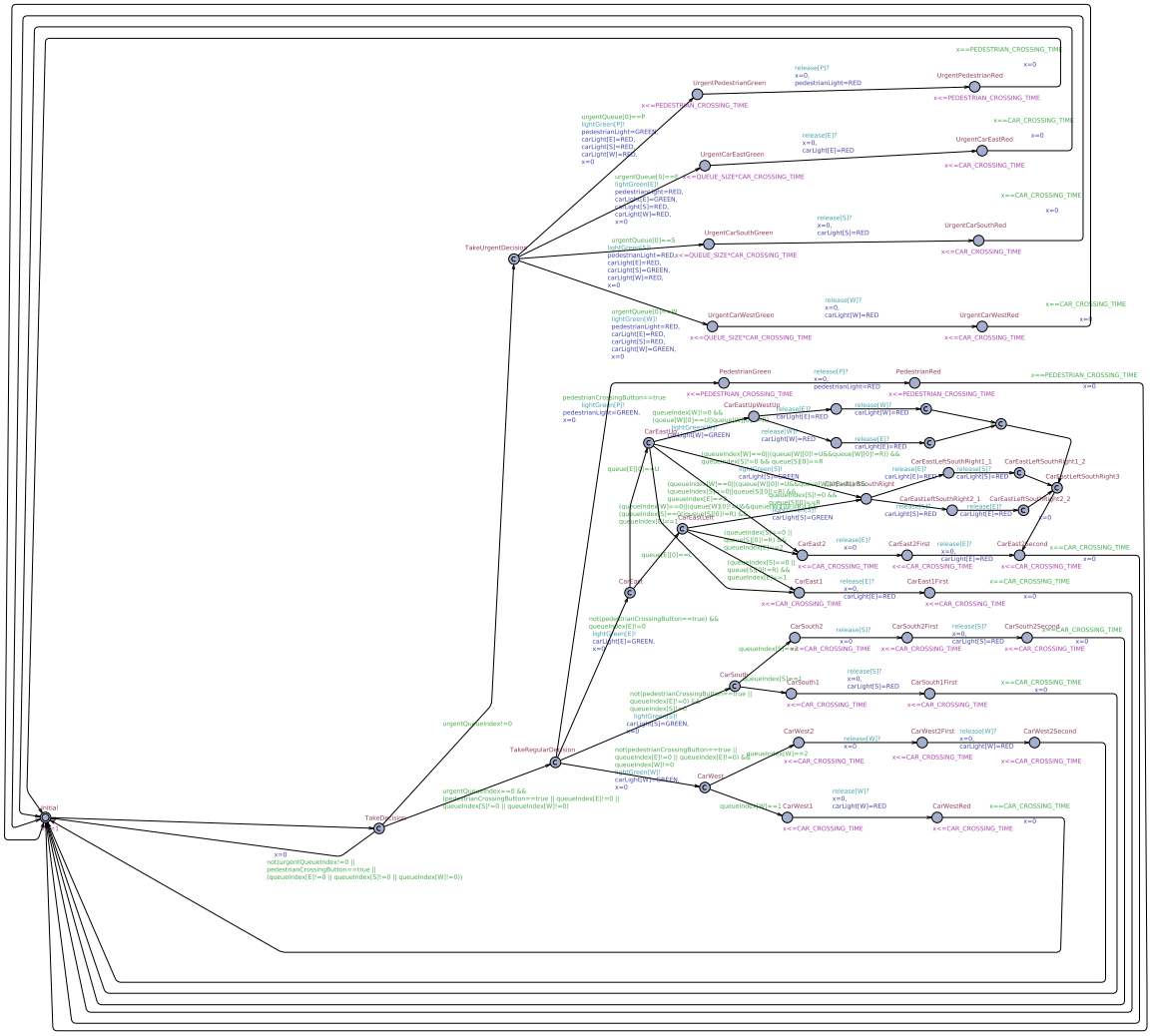


Figure 9: Timed automaton of light controller

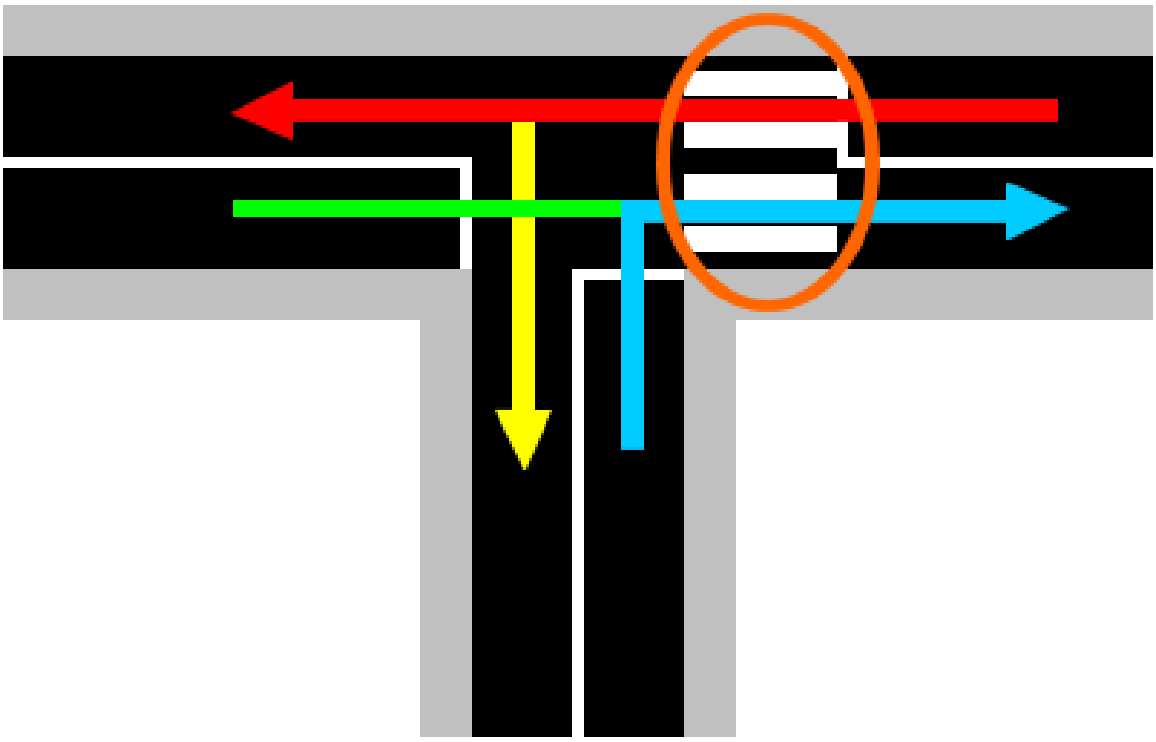


Figure 10: All possible collisions for pedestrians

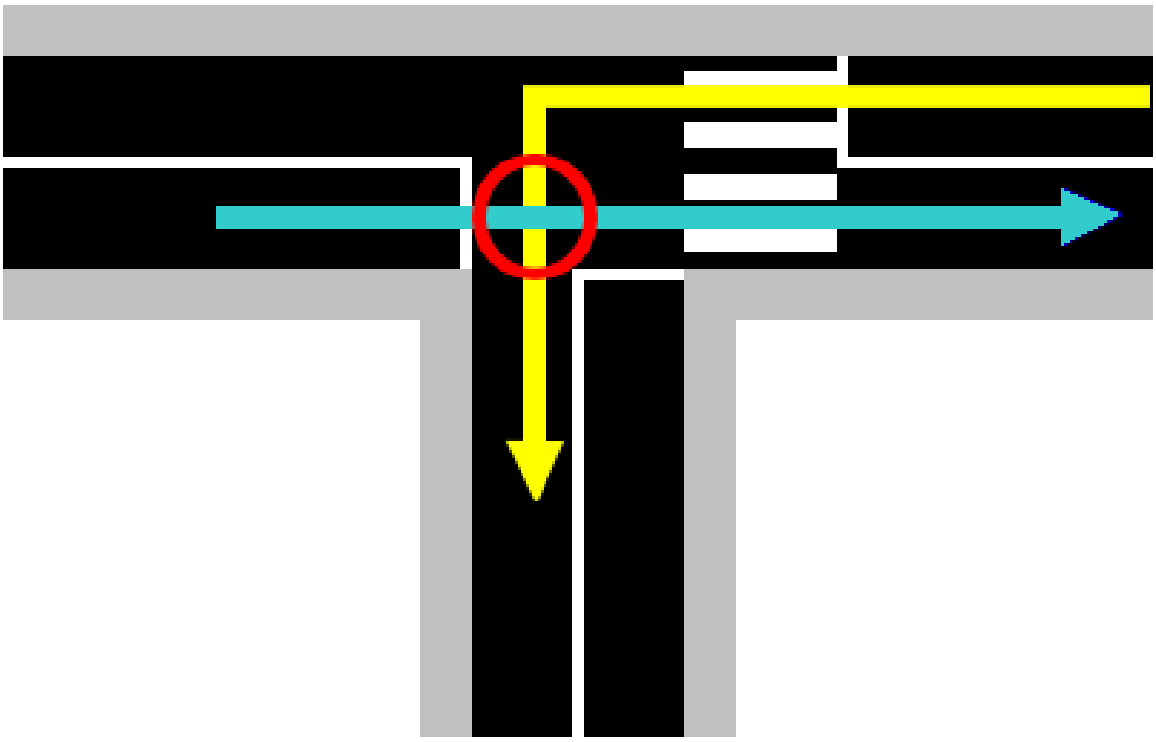


Figure 11: Example of collision for cars

References

[1] Spin - formal verification. `http://spinroot.com/spin/what.html`. Accessed: 2015-05-13.

[2] Rajeev Alur and David L Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994.

[3] Gerd Behrmann, Agnes Cougnard, Alexandre David, Emmanuel Fleury, Kim G Larsen, and Didier Lime. Uppaal tiga user-manual, 2007.

[4] Gerd Behrmann, Re David, and Kim G. Larsen. A tutorial on uppaal 4.0, 2006.

[5] Thierry Massart. Formal verification of computer systems, February 2013.

[6] Michael Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, 1st edition, 1996.