Traffic Light Control

INFO-F-410 – Embedded Systems Design

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

May 29, 2015

Contents

1 Introduction **3**
1.1 First Idea . . . 3
1.2 Refined Idea . . . 3
1.3 Connection With Formal Verification . . . 4
1.4 Benefits of Such Approach . . . 4

2 Prerequisite **4**
2.1 Timed Automaton . . . 5
2.2 Temporal Logic and Properties . . . 5
2.3 Game Theory and Timed Games . . . 6
2.4 Tools . . . 6
2.4.1 Uppaal . . . 6
2.4.2 Uppaal Tiga . . . 7

3 Modeling **7**
3.1 Actors . . . 7
3.1.1 Cars . . . 7
3.1.2 Pedestrians . . . 7
3.1.3 Traffic lights . . . 8
3.2 Timed Automaton . . . 8
3.2.1 Pedestrian generator . . . 8
3.2.2 Car generator . . . 8
3.2.3 Light controller . . . 9

4 Winning strategy **10**
4.1 Winning condition . . . 11

5 Simulator **15**

6 Conclusion **16**

1 Introduction

1.1 First Idea

The goal of this project is to design a controller for traffic lights at a crossroads. The original idea was to model a crossroads with four roads and four crosswalks. 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.

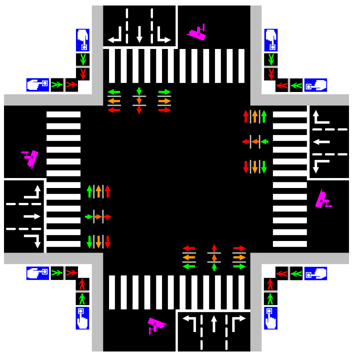


Figure 1: Visual representation of the first draft

1.2 Refined Idea

This first design was too complex to generate a winning strategy. 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

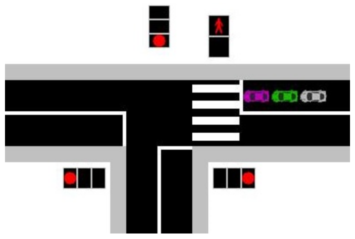


Figure 2: Visual representation of the final idea

- 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 Formal Verification

This project from the Embedded System Design course is combined with the project from Formal Verification course. 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 traffics lights, parts of one can be seen in the other, thus the connection.

1.4 Benefits of Such Approach

The traditional way of implementing a controller would be to simply write the code of that controller, test it, and finally set it up on an embedded system. This approach is not always ideal, especially when dealing with critical system as crossroads, where lives are at stake. Our method consist of modeling the system with timed automatons, and let automated tools generate winning strategies, acting as the controller. This more formal approach delegates work from the developers, and will potentially reduce the risks of reaching dangerous states such as traffic collisions or car/pedestrian collisions depending on the accuracy of the model and if winning strategies can be found.

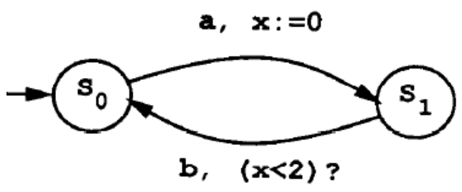
2 Prerequisite

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 w-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 [1].



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

Figure 3: A timed automaton with its language. Image from [1]

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 much 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 [4].

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.[5]*

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 Uppaal

Uppal is a tool used for verification of real time systems that can be modelled 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.

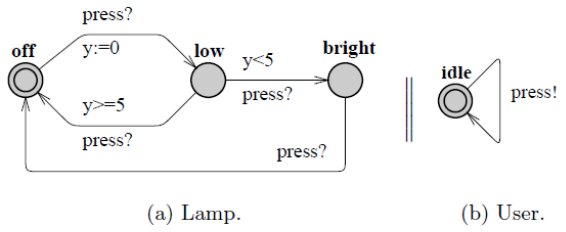


Figure 4: Timed automata used by Uppaal. Image from [3]

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 modellisation 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 [3].

2.4.2 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 of a winning strategy and controller synthesis. For more information on the subject, see [2][3].

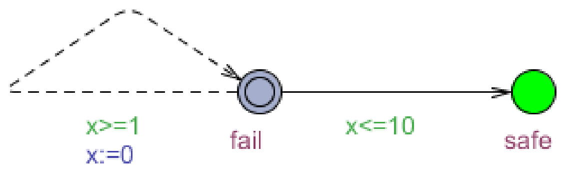


Figure 5: Timed automata used by Uppaal TiGa. Image from demo folder of Uppaal Tiga

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

3 Modeling

3.1 Actors

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

3.1.1 Cars

The cars can come from three directions, West, South and East, and can go up, right or left.
Those cars are parts of the environments, which means that they are by it and are therefore generated in a pseudo randomly way in some lane with some direction to take. There is no bound on the number of cars generated but there can be maximum two cars to limit complexity problems.
The cars begin their path at the beginning of the lane, then they arrive to the traffic light which is either green or red and the car continue is way or not accordingly with it. The time that the cars take to drive through the crossroad is constant and the cars will never be waiting more than a given time limit. If there is more than one car waiting then a queue is formed with a FIFO policy which means that the first one waiting will be the first one leaving the queue.

3.1.2 Pedestrians

There is only one crosswalk, located on the East lane, the pedestrians will always come from the top of the lane and will wait until they can cross the road.
Like the cars, the pedestrians are part of the environment, they are randomly generated in a uncontrollable way. In the modeling, one pedestrian actually represent a group of pedestrians, because

after they have waited to cross the road, once the traffic light allows them to do it, they all go through the crosswalk at once, as a group.
The pedestrians begin their path from the East, walk to the crosswalk, push the button to ask the permission to cross the road and then wait until the traffic light is green. Their crossing time is the same as the one of the cars and is constant. The traffic light will never be green before a pedestrian push the button.

3.1.3 Traffic lights

There is four traffic lights, one for each car lane and one for the pedestrians.
The traffic lights can be green or red, orange was removed because in our modeling, driver have perfect control over their car and can therefore stop instantly. Those traffic lights have sensors that can detect if a car is waiting and, moreover, know the direction that car is going to take by analyzing the blinkers.
The traffic lights are controlled by an embedded system. Their configuration are totally handled by the system, there is no prior restriction on them and could all be green at same time for example.

3.2 Timed Automaton

In this section, we describe the different models that we used.

3.2.1 Pedestrian generator

As it is shown on the figure 6, the pedestrian generator is rather straightforward, the initial state is the empty state, where no pedestrians want to cross the road, from this state there is an uncontrollable transition representing the arrival of pedestrian which leads to the waiting state. The pedestrians have signaled their willingness to cross the road by pushing a button and wait until the light turn green.
To ensure that there is no starvation for the pedestrians, that they will not wait infinitely, a timer is launched and if it goes bigger than a fixed limit, the automaton goes to a broken state through a uncontrollable transition that the system has to take due to an invariant on the waiting state.
The transition going out of the waiting state is controllable. Indeed, it is synchronized with the controller because it makes the light turn green and thus the pedestrians crossing the road. It takes time for the pedestrians to walk through the lane, in order to represent this elapsing time, the automaton goes in a Cross state for one time unit, before returning to the empty state, waiting for pedestrians to be randomly generated.

3.2.2 Car generator

The car generator is a bit more complicated, as seen in the figure 7, it contains fives states and several transitions. Excluding the broken one, the automaton can be in four states, which is actually a cartesian product between the sets of the possible light color, and the set of the presence or not of at least one car.
Initially, the automaton is in the empty red state, where there is no car in the queue and the light is red. The system has then two possibilities, either the controller makes the light turn green and the automaton goes to the empty green states or a car arrives and the system goes in the waiting state. There are actually three transitions between the empty red state and the waiting state, there can

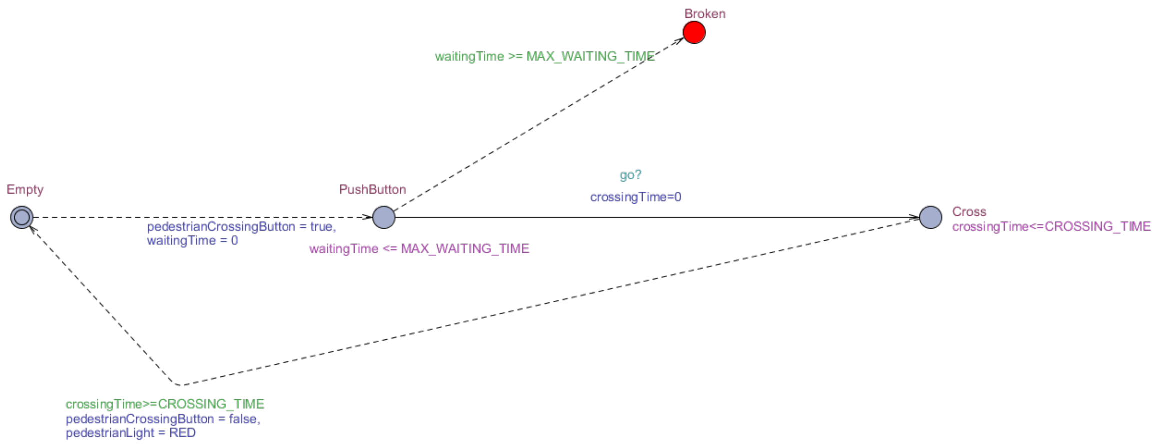

Broken
waitingTime >= MAX_WAITING_TIME
go?
crossingTime=0
Empty
PushButton
Cross
crossingTime<=CROSSING_TIME
pedestrianCrossingButton = true,
waitingTime = 0
waitingTime <= MAX_WAITING_TIME
crossingTime>=CROSSING_TIME
pedestrianCrossingButton = false,
pedestrianLight = RED

Figure 6: Pedestrian generator automaton

be two potential directions for the cars, those are compactly represented in one transition using a select statement, which allows to select non-deterministically a value for i. To make the automaton able to represent the three lanes, there is one transition for each direction even though only two directions are possible for each lane. During this transition, the direction of the car is saved in an array to be used for car collision detection.
Once the system is in the waiting time, more cars can join the queue (actually only one because the queue is limited to two cars) and, as in the pedestrian generator, a timer is launched in order to prevent the car from waiting too long under the threat of going in the broken state.
To avoid this state, the controller has to turn the light green, and let the car go, represented by the Go state. Like for the pedestrians, the cars take some time to drive through the crossroad and it is represented using a clock.
If there is more than one car, then the light stay green and if the controller does not turn it red, another car engages itself in the crossroad. On the other hand, if there is no other car, the automaton goes to the empty green state. In this last state, new car can arrive, making the automaton going back in the Go state or the controller can turn the light red, the system will then be in the empty red state.
There is three instances of this timed automaton in the system, one for each lane.

3.2.3 Light controller

Finally the figure 8 is the automaton of the controller, which just offer all the actions that can be performed, turning each light green or red. Each action takes one unit of time, to make the whole system moving forward and avoid stagnation.
This controller is left with all the possible acitons, the goal is now to find a winning strategy that correctly schedule the cars and the pedestrians over the crossroad.

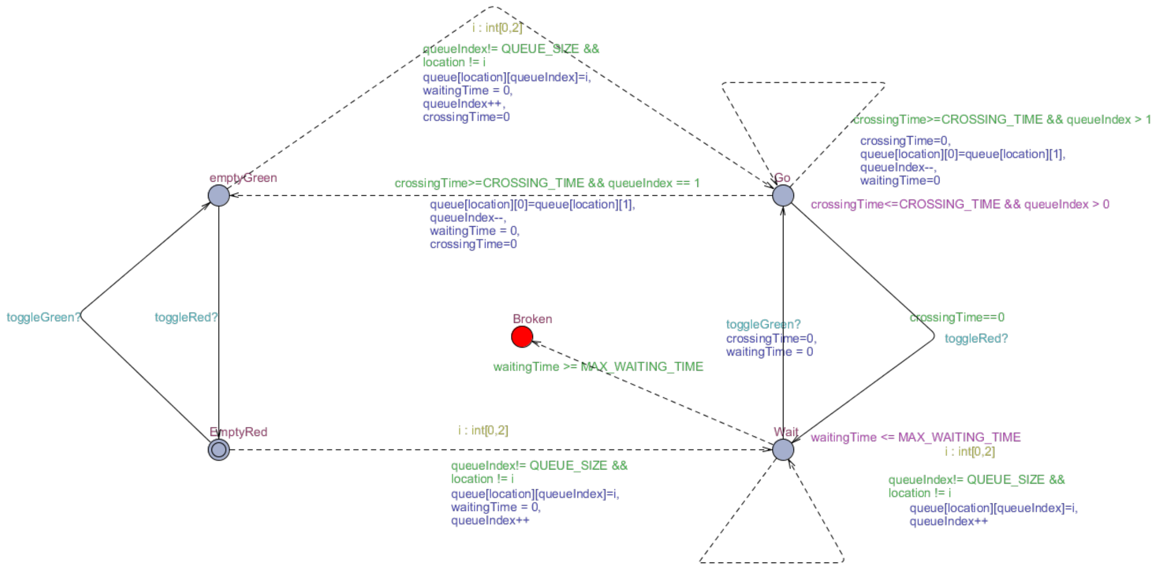

i : int[0,2]
queueIndex!= QUEUE_SIZE &&
location != i
queue[location][queueIndex]=i,
waitingTime = 0,
queueIndex++,
crossingTime=0
crossingTime>=CROSSING_TIME && queueIndex > 1
crossingTime=0,
queue[location][0]=queue[location][1],
queueIndex--,
waitingTime=0
emptyGreen
crossingTime>=CROSSING_TIME && queueIndex == 1
Go
queue[location][0]=queue[location][1],
queueIndex--,
waitingTime = 0,
crossingTime=0
crossingTime<=CROSSING_TIME && queueIndex > 0
toggleGreen?
toggleRed?
Broken
toggleGreen?
crossingTime=0,
waitingTime = 0
crossingTime==0
toggleRed?
waitingTime >= MAX_WAITING_TIME
EmptyRed
i : int[0,2]
Wait
waitingTime <= MAX_WAITING_TIME
i : int[0,2]
queueIndex!= QUEUE_SIZE &&
location != i
queue[location][queueIndex]=i,
waitingTime = 0,
queueIndex++
queueIndex!= QUEUE_SIZE &&
location != i
queue[location][queueIndex]=i,
queueIndex++

Figure 7: Car generator automaton

4 Winning strategy

The system modelized in this project can be seen as a timed game in a graph. The players are the system controller on the first hand and the environment on the other hand. The dynamic of such a game is described as follow: when the system reaches a given state (vertex), both the environment and the controller may decide to bring the "game" in a new state. depending on the conditions defined on the state and the guards set on the allowed actions (edges) from that state. When the environment decides to take a given transition, the controller cannot do anything against this decision and vice versa. The environment can do everything, even the craziest combinations of events and the job of the controller is to maintain the system in good states by taking the right decisions.

In the context of a timed game between a given controller and its environment, it is easily understandable that something has to be done to make sure that the controller always "win" the game, i.e. the system must always remain safe regardless of the environment actions. Model checking tools such as Uppaal, Spin and Prism allow to achieve this goal. As explained in the previous sections, Uppaal is used in this work to verify whether the controller design can always remain the winner of the game against the environment. When this verification succeed, it is possible to get a winning strategy for the controller from Uppaal. Sometimes the system can be too complex (too many states and variables) so that the generation of this strategy takes too many time.

A winning strategy gives a strong guarantee on the correctness of a given controller. The goal of such a strategy for a given controller consists in determining which actions this controller has to take in each state reached by the system in order to either reach a target set of states (reachability property) or either to avoid a given set of states (safety property). In this project, the aim was to

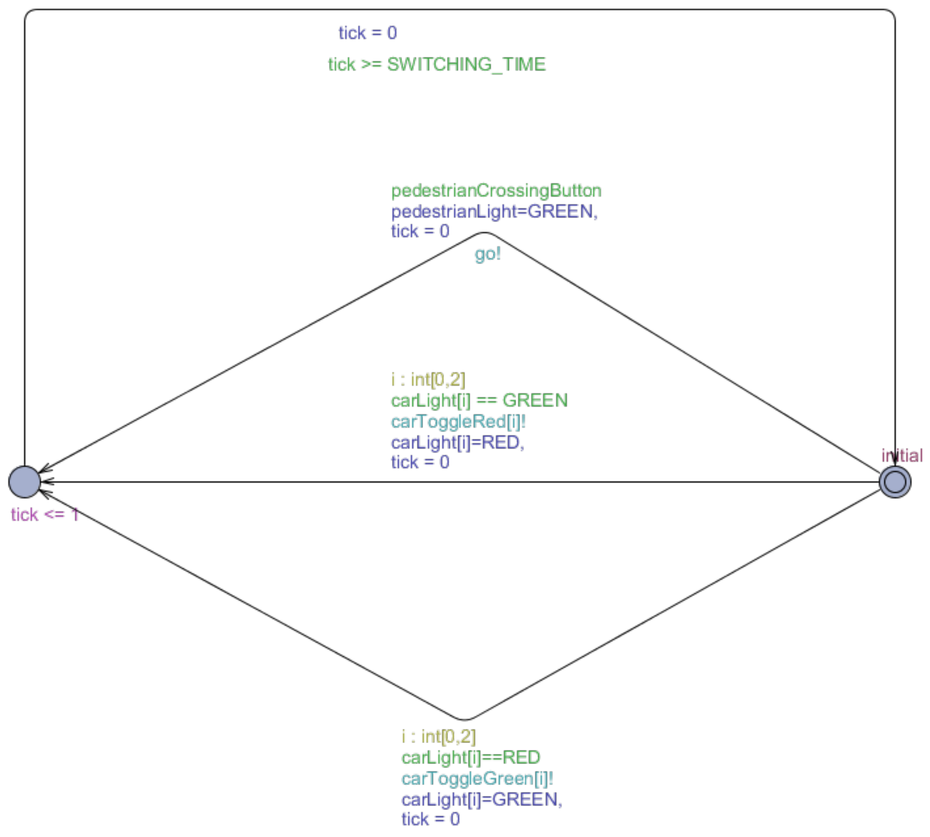


Figure 8: Light controller automaton

avoid accidents and ensure fluidity at the crossroad, i.e. cars should never collide and pedestrians should never be knocked down by cars. Moreover, the controller must avoid deadlocks and starvation. This leads to the fact that in this case the winning strategy will have to satisfy a set of safety properties. These properties have been already discussed in the previous sections of this report.

4.1 Winning condition

Basically, to get a winning strategy from Uppaal, one has to specify clearly what are the *winning conditions*. It can be fullfilled by using the timed computation tree logic (TCTL), the timed version of the well-known temporal logic CTL. For a given time gamed graph A, a set of goal states (win) and a set of bad states (lose), Uppal allows four types of winning conditions:

- *Pure reachability:* the controller **must** win.
- *Strict Reachability with Avoidance:* the controller **must** reach the goal states and avoid the bad ones.
- *Weak Reachability with Avoidance:* the controller **should** reach "win" and **must** avoid "lose"

- *Pure safety:* the controller **must** avoid the bad states.

As we have seen before, the controller designed for this project must avoid the bad states of the system (accidents, deadlocks, etc.). So one understands easily that the winning conditions will be of the type pure safety. These conditions are listed and explained bellow. Note that when the keyword *control* : appears before a condition, it indicates to Uppaal that the controller must satisfy this property having control over controllable transitions.

- Cars should never collide:

```
form_1 =

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

To understand this condition, let's look into an example :

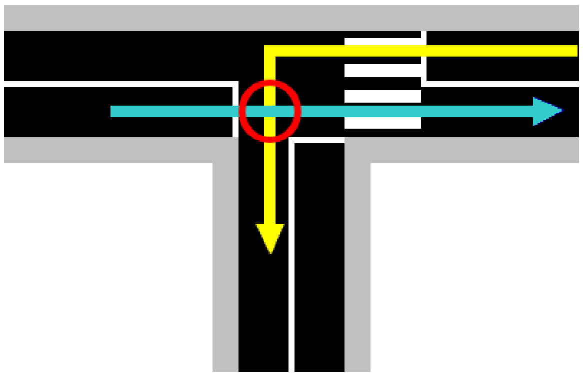


Figure 9: Example of car collision

As it can be seen in the figure 9, a car coming from the East and going left cannot cross the road at the same time than a car coming from the West and going up. Indeed a collision will occur. This property is stated in this part of the winning condition :

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

If a car coming from the West and going straight ahead is crossing, there can be no car coming from the East and turning Left, nor any car coming from the south.

- Pedestrians should never be knocked down by cars:

  ```
  form_2 =

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

  In the formula above, one can understand that *PedestrianGeneratorEast.Cross* means that a pedestrian is crossing the road and the following conditions ensure that no car cross the crosswalk at the same time.
  The restrictions are shown in figure 10, it is all the paths that go through the crosswalk.

- A pedestrian should never wait infinitively before being able to cross a road:

  ```
  form_3 =

  control: A[] not (PedestrianGeneratorEast.Broken)
  ```

  Note that the "Broken" represents a state where a given pedestrian has waited longer than a threshold time. This ensures that pedestrians never wait forever.

- A car should never wait infinitively before being able to cross the crossroad:

  ```
  form_4 =

  control: A[] not (CarGeneratorEast.Broken || CarGeneratorSouth.Broken
  || CarGeneratorWest.Broken)
  ```

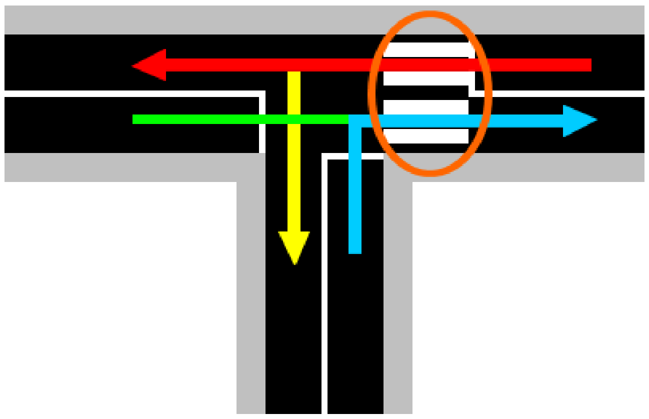


Figure 10: All the wrong car paths during a pedestrian cross

Like for pedestrians, this condition ensures that all the cars (from east, west and south) never wait forever before crossing the crossroad.

- Here is the final condition, note that this condition is a join of all the conditions above, with the or operator between them :

```
form_1 || form_2 || form_4 ||form_4
```

5 Simulator

A simulator has been implemented to illustrate how the system works. Remember that the system comprises a crossroad (in a 'T' shape) of three roads, a traffic light in each road and a crosswalk that has its own traffic light. In other words, the system controller has to manage cars, pedestrians and traffic lights in such a way that accidents never occur.

The simulator represent a trace of an execution of the system. The simulator could not directly use the strategy created by Uppaal Tiga because it was huge, giving the transition to take for every possible system configuration. Unfortunatly, in Uppaal tiga, a trace is not exportable so the simulator represent the trace of a controller with an implemented strategy using guard and invariant, modelised via Uppaal where the trace can be extracted.

The simulation of the system is made possible by the Uppaal timed automata parser library (Libutap). The Libutap library is a c++ library distributed separately from Uppaal under the LGPL license. It allows to convert a given system execution trace, generated from Uppaal, to a human readable one. Remember that an execution trace is one branch among all the possible execution branches of the execution tree of a given system. The readable trace provided by Libutap brings into light the system transitions and all other information needed by a human to describe the system at each time contained into the trace interval time. Basically, the trace generated by the Uppaal tool cannot be understood easily, due to its format, that is why the use of Libutap in this project is justified.
Libutap works as follows,

- To get the Uppaal intermediate format for a model in the .xml format one uses:

  ```
  UPPAAL_COMPILE_ONLY=1 verifyta model.xml - > model.if
  ```

- With the generated intermediate format, one can reconstruct the trace in a readable format with:

  ```
  tracer model.if trace.xtr
  ```

Now that we have seen the Libutap library, lets go back to the simulator. In order to do a simulation, one has to do the following steps:

- First, one generates an execution trace of the system in Uppaal GUI program by using the model files that describe each part of the system;
- Once the first trace has been generated, one uses the Libutap tool to translate the "bad" trace to a more readable one;
- Eventually, the readable trace is transmitted to the simulator that parses it and does the simulation.

Note that in this project, some execution traces are generated in order to show different execution scenarios of the system with the simulator. Figure 11 shows a step of a given simulation where all the cars are stopped in order to allow the pedestrian (see yellow arrows) to cross the road. Note that since the car from the left is stopped, one can conclude that this car will go straight, because otherwise it would be allowed to cross the crossroad. Figure 12 shows a step where only the car from the right is allowed to cross the crossroad (yellow arrow). In this case, one can understand easily why the car from the left is stopped. Basically, this car can not cross the crossroad because probably the car from the right will turn to the left. So the car from the left can not move, regardless of the direction it will go. Otherwise, it would collide with the other car.

6 Conclusion

This project taught us a new way of developing controllers that behave accordingly to their associated external environments. One would usually write its code, and somehow test it (being through classical testing, simulation, or model checking). Using synthesis to generate a correct controller from a modeled environment removes that error prone approach inherently linked with the human factor, ultimately yielding safer code in timely fashion.

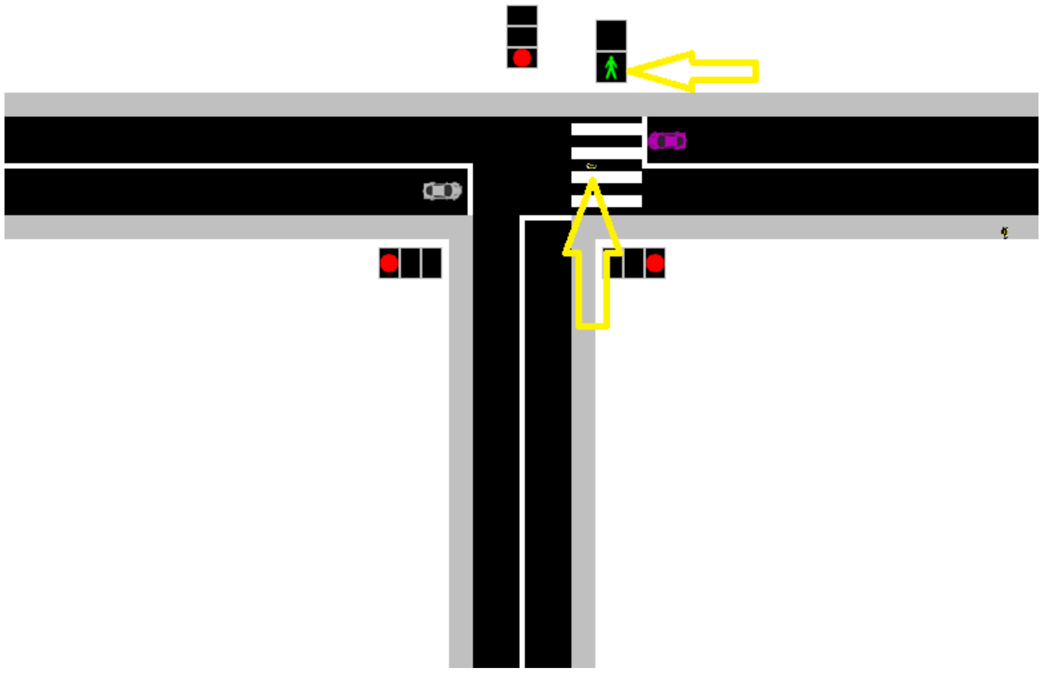


Figure 11: An example of a simulation step

The complexity being pushed up to the modeling phase, it is obvious that the model has to match the reality, which is not always possible. Indeed, one cannot always have a grasp of all the parameters influencing the system under examination, or the system is simply too big for the tools to do the synthesis. It means that errors or lack of details at model level will inevitably be reflected onto the synthesis, potentially leading to unpredictable situations. As always, nothing comes for free, and so the best solution has to be chosen while taking into account the need, but also the feasibility.

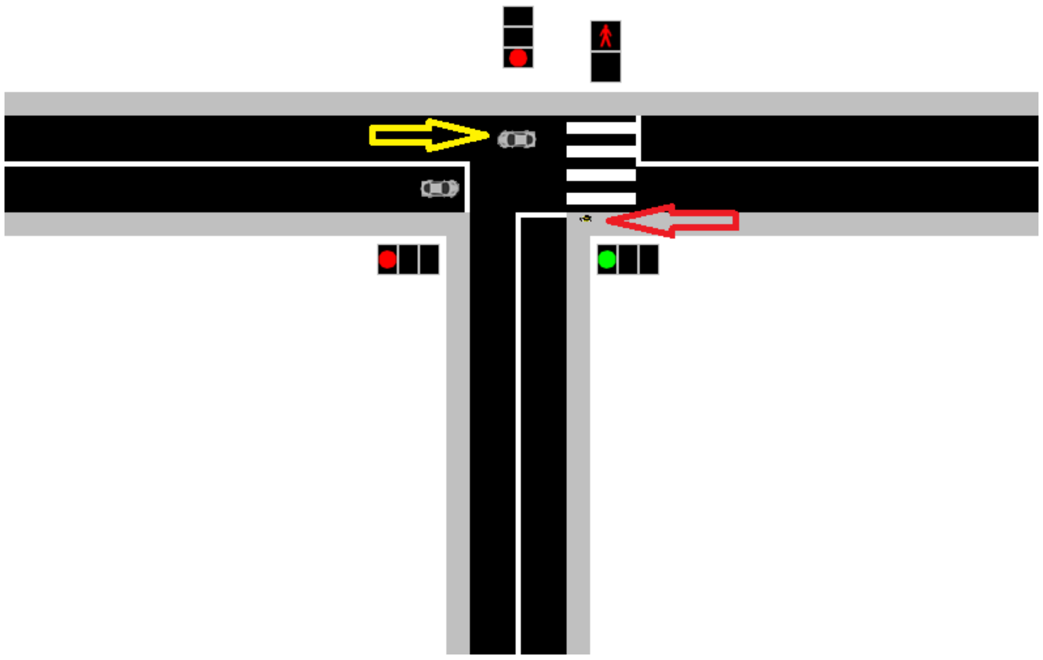


Figure 12: Another example of a simulation step

References

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

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

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

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

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