Implementing Intersection Behavior in STL
CS 81b - Research Final Report [*]

Nikhil Gupta[†]

June 8, 2018

[*]Report for end of term 3.
[†]Supervised by CDS Professor Richard Murray.

1 Introduction

1.1 Linear Temporal Logic

Linear Temporal Logic (LTL) specifies semantics governing the behavior of discrete system, using discrete time. For all sequences of states allowed in the world, one can specify an LTL formula φ which chooses a subset of these sequences which satisfy this formula. Some of the operators that comprise this formula include always ($\Box$), eventually ($\Diamond$), next ($\bigcirc$). In this manner, φ can be used to specify constraints for some problem one is trying to solve. After specifying this, the formula then goes through the a model checking phase which checks what sequences of the world satisfy this formula. Once this phase is complete, the controller then contains all information needed about the execution of the model. The controller can then be used to solve for a specific execution in all (valid) environments. A valid environment is an environment that does not violate assumptions specified in φ.

1.2 Signal Temporal Logic

LTL has limited applications due to its discrete nature. As such, there is an extension of LTL called Signal Temporal Logic (STL), which is used for this project. STL extends LTL by permitting real values in the environment and real valued time. Given the exponential increase in the size of the environment, model checking has become a much more difficult problem. Specific details can be found in the attached resources[1][2]. One significant strategy to combat the exponential increase in the state space is using a receding horizon strategy where the model checker solves for a given horizon into the future instead of to the end of the problem. After the system moves one step into the future, it solves the problem again for this new horizon.

2 Objective

The objective of this project was to synthesize an STL formula φ which could govern the operation of a car through an intersection. The difficulty of this problem comes in when you start to include all the dynamic components at an intersection. These include

- Other cars
- Pedestrians
- Red lights
- Yellow lights
- Yield
- Oncoming traffic
- Speed limits
- Traffic control signs
- Irregular situations

Once you start to include these dynamic elements into the model, the model starts to increase in size and complexity. As such, tackling all these components at once would be a significant task. Due to the structure of how STL formulas are made (with conjunctions), we are able to break this problem up into individual pieces and combine different components without too much effort, as long as they do not actively conflict. Another goal of this project was to find some element of this problem which could not be implemented in the STL framework, and as such, would require some extension to STL. Two examples of these would be the ability to online monitor STL properties and to do reactive synthesis operation in uncertain environments.

[1]Synthesis of Control Protocols for Autonomous Systems (Wongpiromsarn, Topcu, Murray, 2012)
[2]Model Predictive Control for Signal TEmporal Logic Specifications (Raman, et al., 2016)

3 Progress and Results

This project is being implemented on the BluSTL platform[3], written in Matlab[4]. The first direction this project took was a reimplementation attempt in Python due to the pervasiveness of Python. However, after a little while, this direction was deemed unrewarding due to the amount of time it would take to reimplement the project, with relatively little insight gained on the goal of the project. As such, the project was going to built on Matlab[5].

3.1 Avoid an Obstacle

The first piece of the intersection problem that was implemented was the ability for the car to avoid an obstacle[6]. This is a safety condition that will be heavily utilized in the following examples; in particular, we would not want the car to hit any obstacles, whether they be people, static objects, or other cars. As such, this is a core piece of the full intersection problem. Furthermore, in isolation, it is a relatively simple piece to implement so it was a good first attempt at using BluSTL. The formula used to implement this was

$$\square(x_2(t) < 2 \wedge x_2(t) > -2 \wedge (|x_1(t)| > 0.4 \vee |x_2(t)| > 0.4)) \wedge \Diamond(x_1(t) > 0)$$

As a break down of what each component means, x_1 is the vertical distance away from the obstacle location. x_2 is the horizontal distance from the obstacle. We have that the road extends 2 units on either side of the obstacle (first 2 terms). We then have the location of the obstacle, which is 0.4 units in any direction around the origin. All these predicates can be classified as safety conditions. As such, we now need the progress condition, which is the last term. This ensures that something actually happens in the model. In particular, that the car moves past the obstacle. An image representing the execution can be found in Appendix A.

3.2 Simple Signal

The next example implemented was a simple interaction with a signal[7]. The added complexity of this problem was the integration of a dynamic environment element, which was the light changing states. In this example, the car is on one side of the intersection with a green light, the light then turns red before the car can move through it so it has to wait, and then move through the intersection once the light turns green. The formula governing this operation is

$$\square(w_1(t) > 0 \rightarrow (|x_1(t)| > 1 \wedge \Diamond(w_1(t) < 0))) \wedge \square(w_1(t) < 0 \rightarrow \Diamond(x_1(t) > 1))$$

In this formula, we are only moving in 1 dimension so there is no need to restrict x_2. w_1 is the state of the signal with -1 being green and 1 being red. The first predicate governs some restriction about what we expect from the environment. In particular, if the signal is red, then we are not in the intersection (which spans from -1 to 1), and it will eventually turn green. The next predicate governs our progress condition: if the signal is green, then we eventually get past the intersection. Since this is a dynamic model that moves through time, representative screenshots can be found in Appendix B. A GIF can also be found on the GitHub[8].

[3]BluSTL: A Model Predictive Control Toolbox with Signal Temporal Logic Constraints (Raman and Donze, 2016)
[4]`https://github.com/BluSTL/BluSTL`
[5]`https://github.com/nkgupta1/BluSTL`
[6]`https://github.com/nkgupta1/BluSTL/blob/master/examples/car.m`
[7]`https://github.com/nkgupta1/BluSTL/blob/master/examples/car_signal.m`
[8]`https://github.com/nkgupta1/BluSTL/blob/master/images/car_signal1.gif`

3.3 Penalizing Late Inputs

The previous example had a less than ideal feature in that the car would be delayed in reacting to the environment. Representative screenshots of this issue can be found in Appendix B. In particular, in Figure 2c, we can see that the car has stopped short of the intersection. This is further highlighted in Appendix C where the car is not at the intersection yet when the light turns red. However, when we penalize late control inputs, the car tries to achieve its goal as soon as possible so that it does not have to do late inputs. As such, we get Figure 3b where the car moves all the way up to the intersection.

The specific way to penalize inputs can be found in the repository [9]. The mixed integer linear programming solver that solves for the controller does so by trying to minimize an objective function while satisfying a set of constraints. This objective function was extended to include a term penalizing late inputs. In particular, given some horizon of control inputs, the objective function has a term that is the time of the last non-zero input.

While this penalty term improved the performance of the controller, which can be found in Appendix C, it still was not ideal. In particular, the car would still pause at various points. A GIF of the improved execution can be found on the GitHub[10]. Note that this new execution only changed the objective function, everything else remained the same.

3.4 Intersection

The next example implemented was a car making a left turn on yield with oncoming traffic. This code can be found in the GitHub[11] and a GIF of the execution can also be found in the GitHub[12]. The STL formula governing the operation of this is

$$\begin{aligned}
&\Diamond(\Box(x_1(t) < -2)) && (1)\\
&\wedge \Box((x_1(t) > 0 \wedge x_1(t) < 1) \vee (x_2(t) > 0 \wedge x_2(t) < 1)) && (2)\\
&\wedge \Box(|x_1(t) - w_1(t)| > 0.4 \vee |x_2(t) - w_2(t)| > 0.7) && (3)
\end{aligned}$$

x_1 is the x coordinate of vehicle and x_2 is the y coordinate of the vehicle. w_1 is the x coordinate of the traffic and w_2 is the y coordinate of the traffic. The vehicle starts at (0.5, -3) and tries to move to $x_1 < -2$, which corresponds to a left turn. This condition is specified in (1). (2) specifies a safety condition on the lanes; the car will always stay in its lane. The vertical lane is bound between $x \in (0, 1)$ and the horizontal lane is bound between $y \in (0, 1)$. (3) specifies that we will not hit the oncoming traffic. In particular, the vehicle stays at least 0.4 units away from it in x and 0.7 units away from it in the y. This banned region is designated by the shaded box in Appendix D.

One interesting issue that initially occurred in this example is that the controller would violate our STL formula. This seemed odd because this specification is usually taken to be a hard requirement. However, the car was clearly violating it, as can be found in Appendix D. In debugging this issue, I found out that the specification is not a hard requirement but instead occurs a penalty for violating it. As such, I increased this penalty[13] which resolved the issue, as can found in the GIF on the GitHub[14]. In finding an appropriate value for this penalty term, I increased it until the execution did as I expected it to but if I increased it too much, the solver found the program unsolvable. As such, this was not a motivated approach for resolving this issue and requires further exploration.

[9] https://github.com/nkgupta1/BluSTL/blob/master/examples/car_signal_class.m
[10] https://github.com/nkgupta1/BluSTL/blob/master/images/car_signal2.gif
[11] https://github.com/nkgupta1/BluSTL/blob/master/examples/car_intersection.m
[12] https://github.com/nkgupta1/BluSTL/blob/master/images/intersection1.gif
[13] https://github.com/nkgupta1/BluSTL/blob/master/examples/car_intersection.m#L12
[14] https://github.com/nkgupta1/BluSTL/blob/master/images/intersection2.gif

4 Next Steps

Given the current examples that are implemented, it would be prudent to outline the direction the project will take. While there was significant progress in making the car move through the signal in a timely fashion, the car still pauses at certain points in execution. I suspect this has to do with the controller predicting no inputs are necessary for the horizon so it does not move. However, when time is running out, the controller then realizes it has to move again to achieve the goal condition so it moves again.

The next goal would be implement further components of the full model in a piecewise manner. In particular, this would allow us to implement a rudimentary rotation matrix so that we could define the car's motion in terms of a velocity and angle. This would a more general model of movement for the car and increase the number of scenarios the car could enter. There is a failed attempt at this on the GitHub[15]. This attempt centered around the use of `implies` which acts like an if-else for the optimizer. However, the optimizer would always return one of the following two errors:

- `Warning: You have unbounded variables in an implication leading to a lousy big-M relaxation.`
- `Warning: You have unbounded variables in IFF leading to a lousy big-M relaxation.`

This suggests that the if part of the imply statements do not cover the whole space (are under-constrained) but there did not seem to be such an error. The specific simple example was a constant dynamic that would exponentially increase x and then once x crossed some threshold, the piecewise dynamics would stop applying the old dynamics, keeping it at a constant level. However, whenever the system reached this threshold, it would return an error saying `Yalmip error (disturbance too bad ?): Either infeasible or unbounded`. This error probably relates to the under-constrained errors earlier.

Another topic to pursue is the use of a penalty term for violating the specification instead of outright failure. In particular, something to look into is whether one can design a motivated approach such that the specification is always satisfied (a hard constraint). Alternatively, is there a way one can bypass this relaxation of a penalty term.

Another easy step to take this project from here is to make compound examples and extended examples. For example, the car can try to take a left turn on yield with a signal. Additionally, the car can try to navigate through multiple intersections in a street grid with the overall goal of getting to a final goal on the other side of the grid.

[15]`https://github.com/nkgupta1/BluSTL/blob/piecewise/src/STLC_get_controller.m#L140`

Appendices

A Execution of car avoiding obstacle

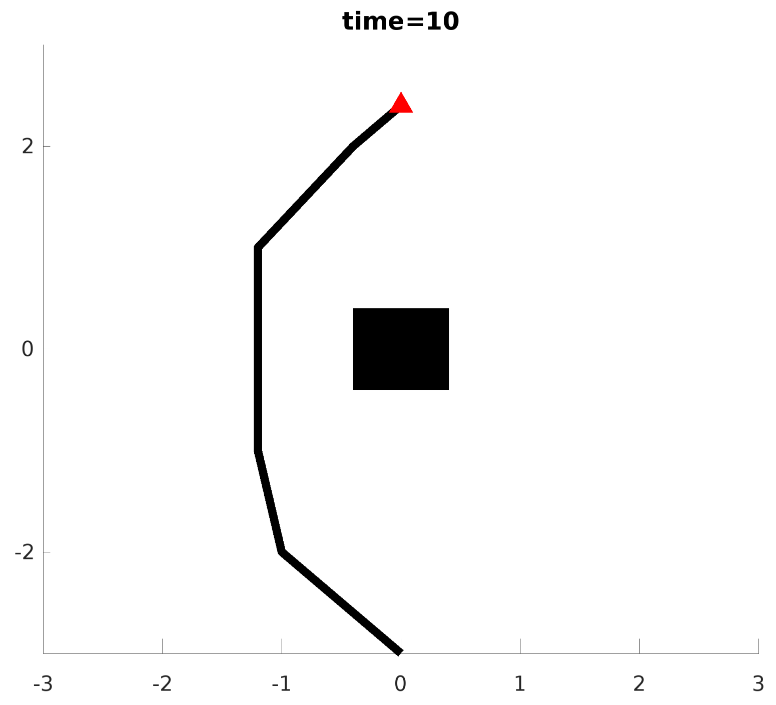
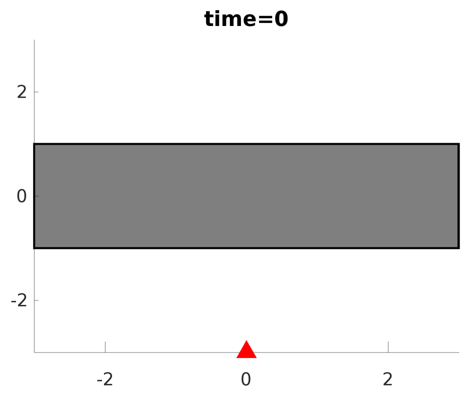
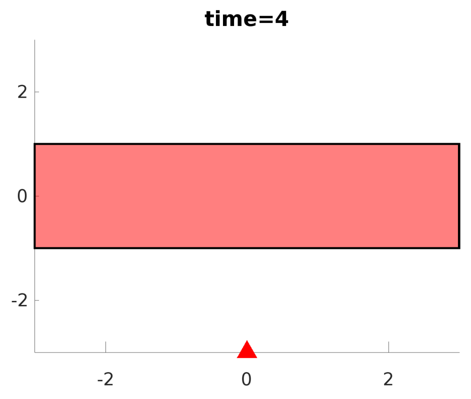


Figure 1: Execution of car (red triangle) past obstacle (black box) via the black path

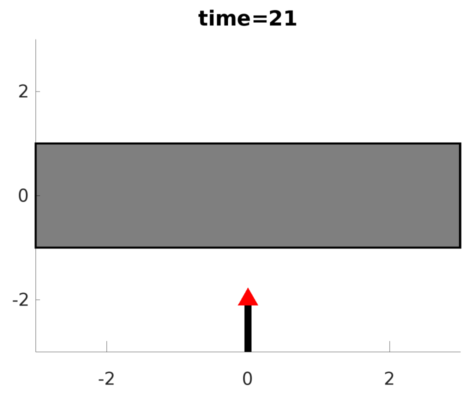
B Execution of car moving through simple intersection



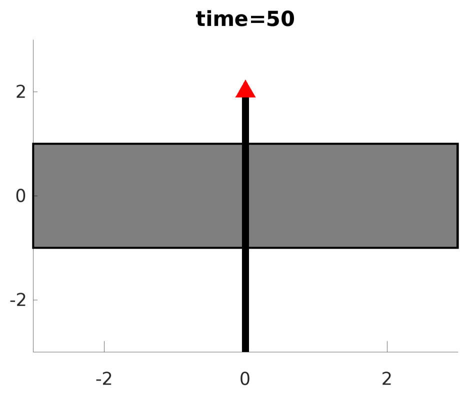
(a) Beginning of model



(b) Signal just turned red



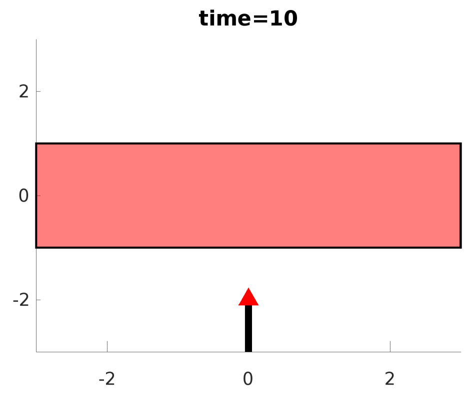
(c) Car has moved to intersection and signal just turned green



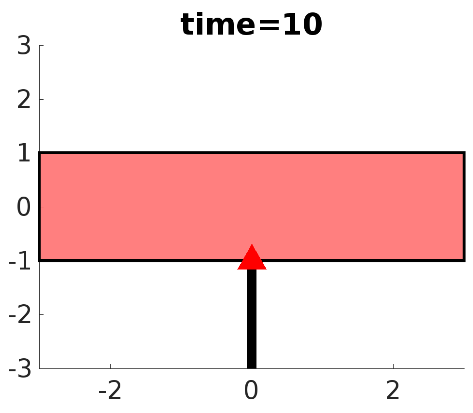
(d) Model is completed

Figure 2: Operation of a car (red triangle) through an intersection (rectangle) through which passage is valid when gray and is not when red

C Penalizing late inputs



(a) Original model without penalty for late control inputs



(b) Updated model with penalty for late control inputs

D Intersection crash

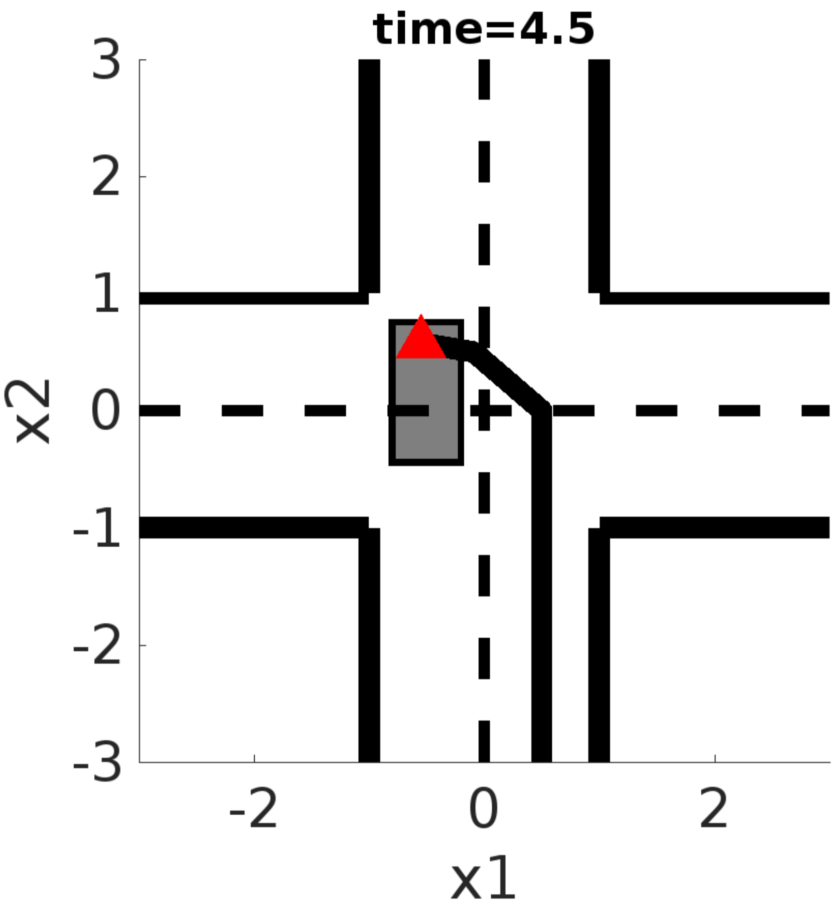


Figure 4: Default penalty for violating the constraints was not high enough so the car crashes into oncoming traffic.