A Control and Time Abstract Synthesis of a Lane and Speed Controller for an Autonomous Car

Richard Moon*, Nicholas W. Renninger*

When doing motion planning for dynamical systems, we often do not deal with correct-by-construction controllers with any sort of guaranteed performance. Specifically, in the field of autonomous vehicles, much research has been done using control schemes with less than stellar notions of provability and safety. This work focuses on using the intersection of classical control control techniques with the field of formal methods, applied specifically to the topic of an autonomous vehicle. We contract a model of a road-car system with other drivers, and create a full control synthesis framework to generate lane-change and velocity controllers that have guaranteed safety properties. Our Python 2 implementation of these algorithms demonstrated a controller that chose a series of lane changes and legal velocity choices that resulted in the car reaching the goal state without hitting other vehicles, without breaking per-lane speed limits we imposed, all while minimizing the time-to-goal.

I. Introduction

Self-driving cars are becoming more relevant in today's world. As we progress closer to a world where there exists fully autonomous cars on the everyday streets, it is important to build up the algorithms behind the car's autonomy with a foundation of strong logic, and verify it rigorously, lest we have black box algorithms with no certainty of avoiding certain unwanted actions nor any understanding of what choices a black box will take in face of a new situation.

Although the current research in self driving cars is already progressing at an impressive speed, it is important for those who wish to enter the field to start off from the more ideal direction; focusing on synthesis and motion planning based on methods that are verified every step of the way, to make sure that the design is always within the full understanding of the developers, everything always expected and planned for every step of the way.

For this reason, we chose the topic of our project on a hypothetical situation, and use formal methods to attempt to synthesize the problem. Imagine someone who entered the highway, but had high motivation to get off at a specific exit as fast as possible, but not motivated enough to break the laws of the land, nor willing to crash into another car. This type of situation is relatable to most drivers today, many of us have been in a situation where we wish to get somewhere quickly, safely, but not enough to break the law.

For the scope of the project, the obstacle cars (the cars that exist on the highway that is not the main car) are deterministic in nature, and go at a constant velocity that is known the the driver, and do not change lanes, and time is discretized to constant time steps, with the transitions of all obstacle cars and the main car synchronized to occur at each time step.

Lane changes and changes in velocity are all treated as instant and occur in a single time step, synchronized with the other transitions, in the scope of this project; further work may go into relaxing any one of these assumptions to bring this problem closer to a more realistic model in the future.

II. Methodology

A. Physical Occupancy Set

To allow for better visualization and intuitive interpretation of the model, we set up a Physical Occupancy Set (POS), which represents the road and all obstacle cars without the main car in the system. It is represented as a grid with two axis representing the lane width and depth of the high, discretized into finite spaces, and

*Graduate Student, Ann & H. J. Smead Department of Aerospace Engineering, 1111 Engineering Drive, Boulder, CO 80309

a third axis to represent time, which is discretized into timesteps. The value of each element in the POS is either 1 or 0, where 1 means that there exists an obstacle car at that location and time. The obstacle cars are all initialized and propagated throughout the model before involving the car, as the obstacle cars are deterministic and single action obstacles, and are independent of the car motion.

The obstacle cars themselves were initialized as consecutive uniform distributions with a mean of around the three times the distance an obstacle car at a given lane should travel in one time step. The velocity of the obstacle cars are defined by the slowest allowed velocity for their specific lane.

The POS will be used to help calculate some of the observations explained later, and is used for plotting the solution trajectory over a visual representation of the full road and obstacle care system.

1. Transition System

The transition system can be defined as:

$$T = (X, \Sigma\delta, O, o)$$

$$x \in X,\ x = (car_X, car_Y, car_T, prev_{lane}, prev_{vel})$$

The state is defined by car_X which represents which lane the state is in, car_Y which represents the how far the state is, and car_T which represents the current timestep the state is in. The previous lane and previous velocity exist in the set of input symbols from the previous state, but are part of the current state definition. They are grouped such that a state includes the information of the velocity and lane inputs that were taken which resulted in the current state.

These are defined with the state in order to have our observations be functions solely of the current state itself. if the previous velocity and lane inputs taken were not part of the state, then any subsequent product taken using this transition system would lead to a memory dependent control scheme, rather then a memoryless one, which is more ideal for general use cases.

$$\sigma \in \Sigma,\ \sigma = (lane, vel)$$

The set of symbols in the transition function are the tuple of lane and velocity inputs, which dictates all states that the current state can transition into.

The inputs at a high level can be interpreted as the car wants to be at a specific lane in the following step of time and wants to travel at a specific velocity while moving into said lane.

$$x = (car_X, car_Y, car_T) = \delta(x_{prev}, \sigma)$$
$$x_{prev} = (prev_{lane}, car_Y - prev_{vel}, car_T - 1)$$

Due to the approach on modeling the transition system to try matching the physical system, the transition function δ transitions a state with given inputs into a new state that would represents the discretized location that would capture the location of the transition in a continuous system, along with capturing specific set of inputs.

$$o \in O\ o = (goal, speed, crash)$$

The set of observations at each given state are a tuple of propositional statements, is the state a goal state, is the state reached by going out of the legal speed limit, and is the state reached through a collision. The observations are calculated using the following functions described below, the goal function, the speed function, and the crash function.

2. Observation Functions

$$goal(car_X, car_Y, goalstate) =$$
$$\begin{cases} 1 & \text{if} \quad (car_X, car_Y) \in goalstate \\ 0 & \text{if} \quad otherwise \end{cases}$$

The goal function compares the current state to the set of states the comprise of the goal states, and returns true if the current state is part of the goal states (the car reached a goal state), otherwise returns false (the car is not at a goal state).

$$speed(car_X, prev_{lane}, prev_{vel}, vel_{prevLane}, vel_{carX}) =$$
$$\begin{cases} 0 & \text{if} \quad prev_{vel} \in vel_{prevLane} \quad \&\& \quad prev_{vel} \in vel_{carX} \\ 1 & \text{if} \quad otherwise \end{cases}$$
$$vel_{prevLane} = f(prev_{lane})$$
$$vel_{carX} = f(car_X)$$

The speed function checks the immediately previous velocity and lane input taken against the set of legal velocities allowed in the input lane ($vel_{prevLane}$), and also compares the velocity against the set of legal velocities allowed in the current lane (vel_{carX}). If the velocity exists in both of the sets, the function returns false (the car transitioned to the state within legal speed bounds), otherwise returns true (the car transitioned into this state illegally). Both ($vel_{prevLane}$) and (vel_{carX}) have their information embedded in the state's own information of their $prev_{lane}$ and car_X, through accessing a global look up table, which we will address as function f.

$$crash(car_X, car_Y, car_T, prev_{lane}, prev_{vel}, vel_{minPrevLane}, vel_{minCarX}) =$$
$$\begin{cases} 1 & \text{if} \quad sumPrevLane + sumCarX > 0 \\ 0 & \text{if} \quad otherwise \end{cases}$$
$$vel_{minPrevLane} = g(prev_{lane}) = min\ f(prev_{lane})$$
$$vel_{minCarX} = g(car_X) = min\ f(car_X)$$

The crash function checks if during a timestep if the car collides into another car; during a same lane transition, only one lane is checked, during a lane switch two lanes are checked.

$$sumPrevLane = \sum_{i=0}^{prev_{vel}-vel_{minPrevLane}} [POS(prev_{lane}, car_Y - prev_{vel} + i, car_T - 1)]$$
$$sumCarX = \sum_{i=0}^{prev_{vel}-vel_{minCarX}} [POS(car_X, car_Y - prev_{vel} + i, car_T - 1)]$$

These sums represents the the number of collisions that the car would go through while making the given transition.

Because the cars are treated as constant linear velocity between time steps, a collision would only occur if the relative position between the car and an obstacle car in the same lane or in the target transition lane changes within an increment of time (if an obstacle car in either same lane or target transition lane is front of the car at a given time, the obstacle car must be in front of the car in the next time step or else there is a collision).

Since the the obstacle cars all travel at the slowest legal velocity for their corresponding lanes(represented by the terms $vel_{minPrevLane}$ and $vel_{minCarX}$, for the previous lane $prev_lane$ and current lane $carX$, respectively), the car cannot be slower then the obstacle cars in the same lanes during a transition, as long as the car is following a legal speed.

Both $vel_{minPrevLane}$ and $vel_{minCarX}$ information are embedded into the state's $prev_{lane}$ and car_X values, which can be accessed by a lookup table called function g, which returns the minimum value returned from the function f given a lane. So it returns the slowest velocity of the set of legal velocities for a given lane.

So with this, the car can check use its relative velocity with respects to the minimum legal velocities of the lanes it is transitioning through, and in a single timestep if no obstacles exists in the relative distance (relative velocity multiplied by one unit of time) in front of the car in the lanes of transition, this means that there are no collisions in that time step and the car does not crash.

The crash function returns true if the state was reached with a collision occurring during the transition, else returns false, the state was transitioned into without an immediate collision.

3. Specification and LTL Automata

Our specification for this project was the following:

Given a car on a highway with a specific starting location and specified goal, get the goal as fast as possible while always following the laws of the road and without crashing.

The laws of the road for this problem are that each lane has a set of legal velocities, with the leftmost lane having the fastest set of velocities, and the right most lane having the slowest set of velocities, and the intermediate lanes having velocities that linearly follow this trend. There are also velocities that exist in multiple lanes to allow legal transitions between two lanes. The fastest lane location relative to physical highway would most easily be interpreted as the standard convention of the fast lane on a highway for regions where drivers drive on the right hand side of the road (such as the USA).

The Linear Temporal Logic Formula that represents the specification is written as:

G (¬ Crash & ¬ Speed) & F (Goal)

The optimization part was not included in the LTL specification, and will be accounted for when solving for an optimal solution, due to the product being deterministic and the solution method being a reachability problem.

When the LTL specification is plotted using a LTL model checker, SPOT, the following Fig. 1 shows a visual representation of the LTL automata.

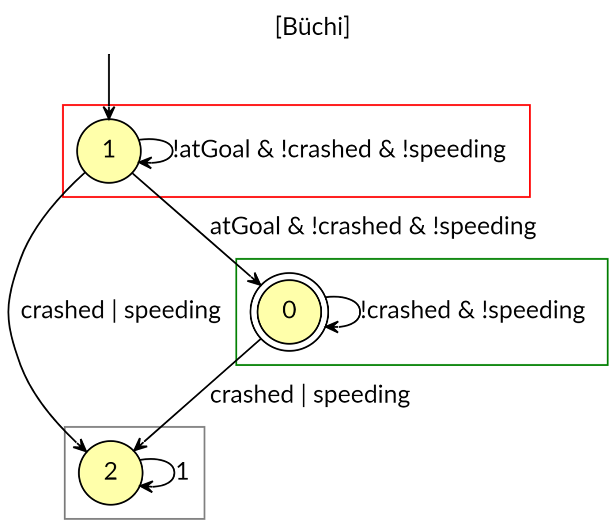

Fig. 1 Deterministic Büchi Automata for our LTL Specification $\phi = G(!crashed \wedge !speeding) \wedge F(atGoal)$

The LTL automata is defined as such:

$$A = (Q, q_0, \Sigma, F)$$

Q is the set of states, q_0, q_1, q_2. From Fig. 1 Node 1 would be q_0, Node 0 would be q_2, and Node 2 would be q_1.

q_0 is the starting state for the automata, and is the state where the goal states have not been reached but the car has not crashed or broken the law yet. q_1 is the state where the car has either crashed or broken the law, and can no longer achieve the specification. q_2 is the accepting that and means the specification has been achieved, where F is the state q_2.

$$(goal,\ speed,\ crash) = \sigma \in \Sigma$$

The input symbols for the LTL automata are the three observations seen at every state in the transition system with their corresponding input lane and velocity symbols.

4. Product Automata

To find the an optimal control strategy for the problem, the product automata must first be found. This is done by taking the product of the transition system and the LTL automata.

$$P = T \bigotimes A$$

$$P = (Q_P, Q_{0P}, \delta_P, F_p)$$

$Q_P = X \times Q$ is a finite set of states.
$Q_0 P = X \times Q_0$ is a set of initial states.
$\delta_p : X \times Q \rightarrow 2^{X \times Q}$ is the transition function where, for a state $(x, q) \in Q_P$:

$$\delta_p((x, q)) = \{(x', q') \in Q_p | x' \in \delta(x) \ \& \ q \ \in R(q, o(x'))\}$$

$F_P = X \times F$ is a set of final states.
$Q_P = (car_X, car_Y, car_T, prev_{lane}, prev_{vel} q)$ in the product automata, which is a tuple containing the product of the x state and the q state.

III. Results

Although we know what the solution would look like, given that the product is a directed acyclic graph, and it is a reachability problem, we ran into nasty implementation issues, and could not reach the goal states from the initial position.

When running our full synthesis code, during our product creation and optimization step, the code would consistently return a system exception with no debugging information:

```
Traceback (most recent call last):
File ".\main.py", line 71, in <module>
    main()
    File ".\main.py", line 63, in main
        acceptingGoalNode = DFA.formAndSolveProduct(TS=TS, LDBA=LDBAObj)

SystemError: error return without exception set
```

Even with profiling with `pdb` and extensive research online, most of the time this sort of bug has to do with a buggy library implementation outside of our control. We believe our algorithms were largely correct, but their inherent exponential blowup may have strained system-level libraries and caused unexpected, unhandled faults that give little information to debug with.

Had the implementation worked, the result section would be a simple discussion of single source single destination shortest path algorithms and the paths it would take in order to teach the goal, and confirmation of the approach. You can find the full implementation included in Appendix IV.

Once the product automata is made, we use a simple shortest path algorithm to get a solution for this problem, as the solution is just a reachability problem. Because the product is deterministic and acyclic, we can treat the product automata as a directed acyclic graph with unweighted edge weights, and use one of the many shortest path algorithms available, in our case we used a modified Breadth First Search algorithm to return the solution path.

Table 1 Simulation Parameters for the Simulation in Figure 2

Number of Lanes	All allowed Velocities	Velocities Allowed in Each Lane	Maximum Time Steps	Distance to Goal
3	(20, 25, 30, 50)	0: (20, 25) 1: (25, 30) 2: (30, 50)	6	130

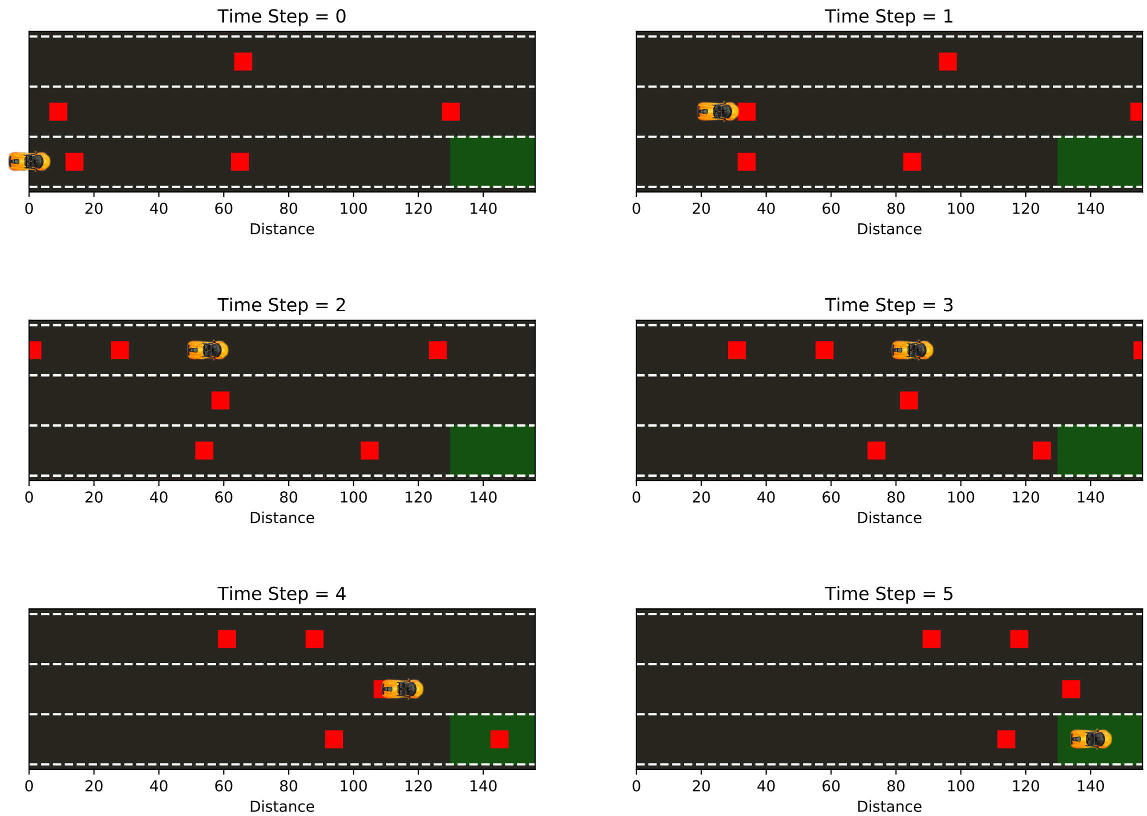

Fig. 2 Visualization of the Simulation and the Optimal Car Path Over 6 Time Steps. The orange McClaren super car represents an average car being controlled by our simulation. The red squares are other cars driving on the road, and the shaded green region in the bottom right corner of the road is the goal region in lane 0 that the orange McClaren super car is trying to reach (this is basically the highway exit).

A sample accepting, optimal path through the transition system found by optimization on the product and outputted by our code:

```
defined simulation properties
built the occupancy set
building the transition system...
built the transition system
built the LDBA
calculating the product automata
found the final solution node in the product
found an optimal accepting path:
Lane: 0 Distance: 0 Time: 0 Velocity: 20
Lane: 1 Distance: 25 Time: 1 Velocity: 25
Lane: 2 Distance: 55 Time: 2 Velocity: 30
Lane: 2 Distance: 85 Time: 3 Velocity: 30
Lane: 1 Distance: 115 Time: 4 Velocity: 30
Lane: 0 Distance: 140 Time: 5 Velocity: 25
```

As we can see in Fig. 2 and in the sample code, the car does not necessarily take the fastest available velocity when given the option. This is due to our modeling scheme, as our transitions are mapped to synchronized time steps, so to the product automata, there is no difference between different paths as long as they correctly reached the goal in the optimal number of steps.

We can also see how the car in Fig. 2 switches to the fastest lane only to move around the car in the middle lane. The velocity it travels at in the top lane (fastest lane) is legal in the middle lane as well, but because there was an obstacle car in front of it, it moved to the faster lane, and kept going at the same velocity, before returning back to the middle lane.

IV. Conclusion

The results worked as expected, meaning that our implementation and logic may not have been the most efficient way to abstract out the topic, but it definitely was a correct way, as the result of implementing our logic did not lead to a contradiction to our expectations.

On the process of trying to have the correct implementation, we tried addressing the relation with our LTL specification and how it would relate to the observations of our transition system by adding velocity and lane choice inputs as part of the transition model, increasing our dimensionality from 4 to 6, which allowed us to have all our atomic propositions in our LTL formula as observations at each state.

This allowed us to have our LTL automata to work with our transition system, but due to increasing the dimensionality of our problem, which already had large space usage already due to representing the physical grid of a highway and a discretized set amount of time, increasing the transition system led to the state explosion issue; our transition system took a decent amount of time to connect the states and initialize the model overall. Another reason that may have assisted in the state explosion was we used a breadth first approach for all our graph traversal methods, rather than depth first search. Though they have same time complexity on directed acyclic graphs, BFS is generally avoided due to the amount of extra space required in BFS to store results, rather then a DFS approach, which would store and return results recursively. So using a set of methods that is prone to take a redundant amount of extra memory on a system that exploded in the amount of states and memory used for those states would only lead to a bigger memory problem.

From watching the other presentations from the class, if given a chance to optimize this model, we would most likely have the lanes be our transition model, instead of the position of the road on top of the lanes, and we would capture the distance using an STL specification instead, to reduce the number of states in the product automata. It would also allow our optimization problem (minimize time taken) to be solvable within the specification aspect of the product model, which would be a more intuitive place to have it in, as the shortest time aspect is more of a specification.

Our final conclusion is more of a general topic, which is to know what a programming language can and cannot do, as the difference between some languages can result in very unexpected bugs when implementing logic. And sometimes even with this precaution, sometimes there are issues that exist that are beyond preparation, such as a bug in a language, which thankfully only occurred in python 3 and not in python 2, allowing us to run our code in python 2.

Acknowledgments

The authors would like to acknowledge the contributions of Prof. Morteza Lahijanian in guiding our initial work on the project and the influence Jin I. Ge et. al. had on our ideas during the development of our methodology.

A. Appendix: Python Code

A. main.py

.

```
from __future__ import print_function
import OS_Calls
import initialize
import POS
import TransitionSystem
```

```python
import LDBA
import DFA


def main():

    OS_Calls.clear_screen()

    ########################################################
    # defining simulation properties
    ########################################################

    (allLanes, allVelocities,
     allowedLaneVelocites, maxDist,
     maxTime, goalStates, initCarX,
     initCarY, initCarT, initCarVel,
     savePath) = initialize.getSimSettings()

    print('defined simulation properties')

    ########################################################
    # Defining the Occupancy Set
    ########################################################

    POSMat = POS.makePOS(allLanes, allowedLaneVelocites,
                         maxDist, maxTime,
                         initCarX, initCarY)

    print('built the occupancy set')

    ########################################################
    # Defining the Transition System
    ########################################################

    print('building the transition system...')

    TS = TransitionSystem.TransitionSystem(initCarX, initCarY, initCarT,
                                           initCarVel, maxTime, allLanes,
                                           allVelocities, allowedLaneVelocites,
                                           goalStates, POSMat)

    print('built the transition system')

    ########################################################
    # Defining the LTL Deterministic Buchi Automata (LDBA)
    #######################################################

    LTLFormula = 'G(!crashed & !speeding) & F(atGoal)'
    LDBAObj = LDBA.LDBA(LTLFormula)

    print('built the LDBA')

    ########################################################
    # Creating the Product Automata, using Topological Sort,
```

```python
    # then backtracking for the solution
    ########################################################

    print('calculating the product automata')

    acceptingGoalNode = DFA.formAndSolveProduct(TS=TS, LDBA=LDBAObj)

    if acceptingGoalNode is not None:
        print('found the final solution node in the product:')
    else:
        print('found no accepting path through product')

    optimalPath = DFA.getPathToRootFromLeaf(acceptingGoalNode)

    ########################################################
    # Print Results
    ########################################################

    if acceptingGoalNode is not None:
        print('plotting results...')
        POS.plotCarAndPOS(POSMat, optimalPath, savePath,
                          initCarY, allLanes, goalStates, maxTime)
        print('done. Have a swagtastic day!')


if __name__ == "__main__":
    main()
```

B. initialize.py

```
import os


def getSimSettings():

    savePath = os.path.join('Figures', 'results.pdf')

    ########################################################
    # defining road simulation properties
    ########################################################
    allLanes = (0, 1, 2)
    # allVelocities = (40, 50, 60, 100)
    # allVelocities = (80, 100, 120, 200)
    allVelocities = (20, 25, 30, 50)
    # allowedLaneVelocites = [(40, 50), (50, 60), (60, 100)]
    allowedLaneVelocites = [(20, 25), (25, 30), (30, 50)]

    ########################################################
    # defining the end of the simulation
    ########################################################

    # pY ranges from 0 - 4000
    # maxDist = 800
    # maxDist = 1600
    maxDist = 400

    # simulate maxTime time steps
    maxTime = 6

    goalX = 0
    # goalYMin = 550
    # goalYMin = 1100
    goalYMin = 230
    # goalYMax = 700
    # goalYMax = 1400
    goalYMax = 350

    goalStates = makeGoalStates(goalX, goalYMin, goalYMax)

    ########################################################
    # defining the car's initial state
    ########################################################

    # lane number
    initCarX = 0

    # distance from the start of the highway where car starts
    # CANT be 0
    # initCarY = 200
    initCarY = 100

    # start the simulation at a time step of 0
    initCarT = 0
```

```python
    # need to be safe and set some sort of min velocity
    initCarVel = min(allowedLaneVelocites[initCarX])

    return (allLanes, allVelocities, allowedLaneVelocites, maxDist,
            maxTime, goalStates, initCarX, initCarY, initCarT, initCarVel,
            savePath)


def makeGoalStates(goalX, goalYMin, goalYMax):

    goalStates = []
    for goalY in range(goalYMin, goalYMax + 1):
        goal = (goalX, goalY)
        goalStates.append(goal)

    return goalStates
```

C. POS.py

```
from __future__ import print_function
import numpy as np
import random
import matplotlib.pyplot as plt
from matplotlib.offsetbox import OffsetImage, AnnotationBbox


#
# @brief      Makes a Physical Occupancy Set Matrix. This matrix is basically a
#             binary, 3D matrix that indicates where is the road discretization
#             other vehicles (obstacles) exists
#
# The other cars' locations are drawn from uniform distributions and then
# transformed to distribute them properly
#
# @param      allLanes               A list of all possible lane numbers on the
#                                    highway.
# @param      allowedLaneVelocites   The allowed lane velocities tuple for a
#                                    certain lane number
# @param      maxDist                The maximum simulation distance
# @param      maxTime                The maximum simulation time
# @param      initCarX               The initial carX state. CarX ~ lane on
#                                    highway
# @param      initCarY               The initial carY state. CarY ~ distance
#                                    down highway
#
# @return     POS matrix given the simulation conditions
#
def makePOS(allLanes, allowedLaneVelocites, maxDist, maxTime,
            initCarX, initCarY):

    spaceFactor = 6

    numLanes = max(allLanes) + 1
    POS = np.zeros((numLanes, maxDist, maxTime))

    for ii in range(0, numLanes):

        # generating the spacing for the random other cars
        currLaneVelocityRange = allowedLaneVelocites[ii]
        minAllowedVelInLane = min(currLaneVelocityRange)
        dist = random.randint(0, minAllowedVelInLane * spaceFactor)

        # generating the obstacles for one time slice
        while dist < maxDist:
            POS[ii, dist, 0] = 1
            dist = dist + random.randint(0, minAllowedVelInLane * spaceFactor)

        # make sure to delete any other car that happens to be at the initial
        # state of our car
        POS[initCarX, initCarY, 0] = 0

        # propagating them forward through the time dimension of the occupancy
```

```
        # set
        for time in range(0, maxTime - 1):
            vbuf = (time + 1) * minAllowedVelInLane
            POS[ii, (vbuf + 1):, time + 1] = POS[ii, 1:-vbuf, 0]

    return POS


#
# @brief      Plots the POS matrix as well as the optimal path the car takes
#             through the road over successive time steps
#
# @param      POSMat       The POS matrix object
# @param      optimalPath  The optimal path, which is a list of Node objects
# @param      saveTitle    The save path string
# @param      initCarY     The initial downrange distance of the car
# @param      allLanes     All possible lane indices in a tuple
# @param      goalStates   The goal states - a list of (x, y)
# @param      maxTime      The maximum simulation time
#
# @return     a plot
#
def plotCarAndPOS(POSMat, optimalPath, saveTitle,
                  initCarY, allLanes, goalStates, maxTime):

    fig = plt.figure()
    image = plt.imread('car.png')
    shouldSavePlot = True

    # make a subplot for the state of the road for each time step
    for t in range(0, maxTime):

        ax = fig.add_subplot(3, 2, t + 1)

        # making the axis look like asphalt
        ax.set_facecolor((0.156, 0.149, 0.129))

        # Load images

        dat = POSMat[:, :, t]

        ########################################################
        # plotting the other cars on the road
        ########################################################

        nonZerosOrig = np.nonzero(dat)
        nonZerosFilt = []
        nonZerosFilt.append([])
        nonZerosFilt.append([])

        # car can't start at the very beginning of sim track, so trim off the
        # irrelevant beginning obstacle car states
        for ii in range(0, len(nonZerosOrig[0])):
            if nonZerosOrig[1][ii] >= initCarY:
```

```
        nonZerosFilt[0].append(nonZerosOrig[0][ii])
        nonZerosFilt[1].append(nonZerosOrig[1][ii])

otherCarPos = np.array([nonZerosFilt[0], nonZerosFilt[1]])
otherCarlanes = otherCarPos[0]

# re-normalize all trimmed distances to be distances from the car's
# starting location
otherCarDistances = otherCarPos[1] - initCarY
ax.plot(otherCarDistances, otherCarlanes, 'rs',
        markersize=11, label='Other Cars')

########################################################
# plotting the lanes of the road
########################################################

xCoordsOfOtherCars = np.array(range(len(otherCarDistances))) *\
    max(otherCarDistances) / len(otherCarDistances)

for lanePos in range(min(allLanes) - 1, max(allLanes) + 2):
    lane = np.repeat(lanePos + 0.5, len(otherCarDistances))
    ax.plot(xCoordsOfOtherCars, lane, 'w--',
            label=None)

########################################################
# plotting the goal states
########################################################

xGoal = []
yGoal = []
for state in goalStates:
    # re-normalize all trimmed distances to be distances from the car's
    # starting location
    xGoal.append(state[1] - initCarY)
    yGoal.append(state[0])

minYGoal = min(yGoal) - 0.5
maxYGoal = max(yGoal) + 0.5

ax.fill_between(xGoal, minYGoal, maxYGoal,
                alpha=0.5, color='green', label='Goal Region')

########################################################
# plotting the car's position at time t
########################################################

currCarState = optimalPath[t].state

# re-normalize all trimmed distances to be distances from the car's
# starting location
carY = currCarState.carY - initCarY
carX = currCarState.carX

imscatter(carY, carX, image, zoom=0.04, ax=ax)
```

```python
        ######################################################
        # plot labeling
        ######################################################

        # never describe your figures
        # ax.legend(fancybox=True, framealpha=0.5, loc='upper right')
        ax.set_title('Time Step = %d' % t)
        ax.set_xlabel('Distance')

        ######################################################
        # plot formatting
        ######################################################

        minY = min(allLanes) - 0.6
        maxY = max(allLanes) + 0.6
        maxX = min(xGoal) * 1.2
        minX = 0

        plt.xlim((minX, maxX))
        plt.ylim((minY, maxY))
        ax.axes.get_yaxis().set_visible(False)

    plt.tight_layout()
    plt.subplots_adjust(wspace=0.2, hspace=0.8)
    if shouldSavePlot:
        fig = plt.gcf()
        fig.canvas.manager.full_screen_toggle()
        fig.show()
        fig.set_size_inches((11, 8.5), forward=False)
        plt.savefig(saveTitle, dpi=500)
        print('wrote figure to ', saveTitle)
    plt.show()


#
# @brief      This is black magic
#
# @param      x       x plot location
# @param      y       y plot location
# @param      image   The image path
# @param      ax      the axes artist object
# @param      zoom    The zoom level for the image
#
# @return     list of artist objects for all of da (x,y) pairs
#
def imscatter(x, y, image, ax=None, zoom=1):
    if ax is None:
        ax = plt.gca()
    im = OffsetImage(image, zoom=zoom)
    x, y = np.atleast_1d(x, y)
    artists = []
    for x0, y0 in zip(x, y):
        ab = AnnotationBbox(im, (x0, y0), xycoords='data', frameon=False)
```

```
    artists.append(ax.add_artist(ab))
ax.update_datalim(np.column_stack([x, y]))
ax.autoscale()
return artists
```

D. Node.py

.

```
class NodeState:
    # @brief      This class (struct) contains all of the info needed to hold
    #             the state of a node in the DFAs used for this project

    #
    # @brief      Constructs the NodeState object.
    #
    # @param      self      The NodeState object instance
    # @param      carX      The current carX value ~ lane on highway
    # @param      carY      The current carY value ~ distance down highway
    # @param      carT      The current carT value ~ current time step
    # @param      q         The LDBA state - pretty much meaningless
    # @param      prevLane  The previous lane for the car during the last time
    #                       step
    # @param      prevVel   The previous velocity for the car during the last
    #                       time step
    #
    def __init__(self, carX=None, carY=None, carT=None, q=None,
                 prevLane=None, prevVel=None):

        self.carX = carX
        self.carY = carY
        self.carT = carT
        self.q = q
        self.prevLane = prevLane
        self.prevVel = prevVel


class Observation:
    # @brief this is just an enum / struct for the three different observations

    #
    # @brief      Constructs the Observation object.
    #
    # @param      self      The Observation object instance
    # @param      atGoal    @bool for inidicating the NodeState is in the goal
    #                       state
    # @param      crashed   @bool for inidicating the NodeState is in a state
    #                       that has collided with an obstacle in the POS
    # @param      speeding  @bool for inidicating the NodeState is such that
    #                       the current velocity exceeds the defined maximum
    #                       allowable velocities
    #
    def __init__(self, atGoal, crashed, speeding):

        self.atGoal = atGoal
        self.crashed = crashed
        self.speeding = speeding


class Node:
    #
```

```python
# @brief      Class for a full Node in an automata for this project
#

#
# @brief      Constructs the Node object.
#
# @param      self         The Node object instance
# @param      state        A reference to the NodeState object, containing
#                          the state info for the node
# @param      index        The node index (numerical, unique id)
# @param      obs          An Observation object holding the three
#                          observations a Node in this project can have:
#                          1) 'atGoal': @bool
#                          2) 'crashed': @bool
#                          3) 'speeding': @bool
# @param      adjList      The Node's adjacency list, containing the
#                          connected Nodes to this Node instance
# @param      isAccepting  Indicates if this Node is accepting in a DBA
# @param      isVisited    Indicates if this Node has been visited during a
#                          graph search
# @param      parent       The parent node
#
def __init__(self, state, index=None, obs=None, adjList=[],
             isAccepting=False, isVisited=False, parent=None):

    self.state = state
    self.index = index
    self.obs = obs
    self.adjList = adjList
    self.isAccepting = isAccepting
    self.isVisited = isVisited
    self.parent = parent
```

E. DFA.py

.

```
from __future__ import print_function
import Node
from collections import deque


class DFA:
    # @brief      A class representing a Deterministic Finite Automata

    # @brief      Constructs the DFA object.
    #
    # @param      self       The object instance reference
    # @param      nodes      A list of Node objects making up the DFA
    # @param      startNode  The initializing Node object
    # @param      transFcn   The transaction function for the DFA
    # @param      accepts    A list of indices into nodes for Nodes that accept
    #
    def __init__(self, nodes, startNode, transFcn=None, accepts=None):

        self.nodes = nodes
        self.startNode = startNode
        self.transFcn = transFcn
        self.accepts = accepts


#
# @brief      This functions forms the product automata between a deterministic
#             Buchi automata and a deterministic finite transition
#             system.transition
#
#             As forming the product involves using a graph search (here the
#             graph search is based on BFS), we simply instrument this BFS
#             search to build up a path through the product, and return once it
#             finds the first accepting state. As this graph is a DAG, BFS
#             produces the shortest path through the graph and thus this trace
#             in the product is actually the optimal controller.
#
# @param      TS    A TransistionSystem to product with the LBDA, containing
#                   the physical modeling transitions
# @param      LDBA  The LDBA (LTL Deterministic Buchi Automata) encoding the
#                   LTL specification on TS
#
# @return     the first Node object in the product to accept
#
def formAndSolveProduct(TS, LDBA):

    startTSNode = TS.DFA.startNode
    startLDBANode = LDBA.DFA.startNode

    startTSState = startTSNode.state
    carX = startTSState.carX
    carY = startTSState.carY
    carT = startTSState.carT
```

```python
prevLane = startTSState.prevLane
prevVel = startTSState.prevVel
obs = startTSNode.obs

q = startLDBANode.state.q

startProdState = Node.NodeState(carX, carY, carT, q, prevLane, prevVel)

index = 0
prevProdNode = None
prevTSNode = None
startProdNode = Node.Node(state=startProdState, index=index,
                          obs=obs, adjList=[],
                          isAccepting=False, isVisited=False,
                          parent=prevProdNode)

Nodes = []
Nodes.append(startProdNode)
index += 1

nodeQueue = deque()
nodeQueue.append((startProdNode, prevTSNode, startTSNode))
accepts = []

while nodeQueue:

    prevProdNode,\
        prevTSNode, currTSNode = nodeQueue.popleft()
    newProdNode = prevProdNode

    if not currTSNode.isVisited:

        currTSNode.isVisited = True
        keepSearching = True

        if (prevTSNode is not None) and (prevProdNode is not None):
            currObsv = currTSNode.obs

            currState = currTSNode.state

            carX = currState.carX
            carY = currState.carY
            carT = currState.carT
            prevLane = currState.prevLane
            prevVel = currState.prevVel

            qNew = LDBA.DFA.transFcn(prevProdNode.state.q, currObsv)
            qNewAccepts = (qNew in LDBA.DFA.accepts)

            newProdState = Node.NodeState(carX, carY, carT, qNew,
                                          prevLane, prevVel)
            newProdNode = Node.Node(state=newProdState, index=index,
                                    isAccepting=qNewAccepts,
                                    parent=prevProdNode)
```

```python
                # anything Node after reaching state 1 will not work
                keepSearching = (qNew != 1)
                if keepSearching:
                    accepts.append(index)
                    prevProdNode.adjList.append(newProdNode)

                # turn on for debug :)
                # print('X:', carX,
                #       'Y:', carY,
                #       'T:', carT,
                #       'index:', newProdNode.index,
                #       'parentIdx:', newProdNode.parent.index,
                #       'prevLane:', prevLane,
                #       'q:', qNew,
                #       'atGoal:', currObsv.atGoal,
                #       'crashed:', currObsv.crashed,
                #       'speeding:', currObsv.speeding,
                #       'keepSearching:', keepSearching)

                # goal state is defined in LDBA as q = 2
                atGoal = (qNew == 2)
                if atGoal:
                    return newProdNode

                index += 1

            if keepSearching:
                # now need after we have relaxed some of da edges its time to
                # do the BFS queuing
                for neighbor in currTSNode.adjList:
                    if not neighbor.isVisited:
                        nodeQueue.append((newProdNode, currTSNode, neighbor))

    # if you get here things have gone horribly wrong
    return None


#
# @brief      Gets the path to root from leaf of the DFA
#
# @param      leaf  The leaf Node object
#
# @return     A list of Node objects with the root at index = 0 and the leaf at
#             the last index
#
def getPathToRootFromLeaf(leaf):

    currNode = leaf
    nodeQueue = deque()
    Nodes = []

    # putting nodes in a stack so we can reverse their order
    while currNode is not None:
```

```
    nodeQueue.append(currNode)
    currNode = currNode.parent

# popping the stack into an array of Node, which containt the ordered path
# from the root to the leaf Node.
while nodeQueue:
    currNode = nodeQueue.pop()
    Nodes.append(currNode)

    state = currNode.state

    print('Lane:', state.carX,
          'Distance:', state.carY,
          'Time:', state.carT,
          'Velocity:', state.prevVel)

return Nodes
```

F. TransitionSystem.py

.

```python
import Node
import DFA
from collections import deque


class TransitionSystem:
    # @brief    A class representing a transition system DFA abstraction
    #
    # For our purposes, this class represents the car-road transition system,
    # with all possible lane changes and choices of velocity allowed

    #
    # @brief      Constructs the TransitionSystem object.
    #
    # @param      self              The TransitionSystem object instance
    # @param      initCarX          The initial carX state. CarX ~ lane on
    #                               highway
    # @param      initCarY          The initial carY state. CarY ~ distance down
    #                               highway
    # @param      initCarT          The initial carT state. CarT ~ time step
    # @param      initCarVel        The initial velocity of the car
    # @param      maxTime           The maximum time step allowed
    # @param      allLanes          A list of all possible lane numbers on the
    #                               highway.
    # @param      allVelocities     A list of all possible car velocities for
    #                               ALL lanes
    # @param      allowedLaneVels   The allowed lane velocities tuple for a
    #                               certain lane number
    # @param      goalStates        The goal states for the car
    # @param      POS               The POS (physical occupancy set) object
    #                               which contains the 3D projection of a Node
    #                               state onto the x, y, and time grid for the
    #                               empty road.
    #
    def __init__(self, initCarX, initCarY, initCarT, initCarVel,
                 maxTime, allLanes, allVelocities, allowedLaneVels,
                 goalStates, POS):

        newNodeState = Node.NodeState(initCarX, initCarY, carT=0,
                                      prevLane=initCarX, prevVel=initCarVel)

        newNodeObs = Node.Observation(atGoal=False, crashed=False,
                                      speeding=False)

        initNode = Node.Node(state=newNodeState, obs=newNodeObs,
                             isVisited=False)
        Nodes = []
        Nodes.append(initNode)

        nodeQueue = deque()
        nodeQueue.append(initNode)
        atGoal = False
```

```python
while True:

    currNode = nodeQueue.popleft()
    allowedLanes = self.getAdjLanes(currNode.state.carX, allLanes)

    if (currNode.state.carT != maxTime):
        for lane in allowedLanes:
            for vel in allVelocities:

                # populate a new node at this state
                currState = currNode.state
                carX = lane
                carY = currState.carY + vel
                carT = currState.carT + 1
                prevLane = currState.carX
                prevVel = vel

                nextState = Node.NodeState(carX=carX,
                                           carY=carY,
                                           carT=carT,
                                           prevLane=prevLane,
                                           prevVel=prevVel)

                nextNode = Node.Node(state=nextState,
                                     isVisited=False,
                                     adjList=[])

                allowedVelsPrevLane = allowedLaneVels[prevLane]
                allowedVelsCarXLane = allowedLaneVels[carX]

                # determining if there is crashing
                minSpeedInPrevLane = min(allowedVelsPrevLane)
                minSpeedInCarXLane = min(allowedVelsCarXLane)

                crashed = self.crashed(prevLane, prevVel, carX, carY,
                                       carT, minSpeedInPrevLane,
                                       minSpeedInCarXLane, POS)
                if not crashed:

                    # determining if there is speeding
                    speeding = self.speeding(prevLane, prevVel, carX,
                                             allowedVelsPrevLane,
                                             allowedVelsCarXLane)

                    # determining if the new state is in the goal state
                    atGoal = self.inGoalStates(carX, carY, goalStates)

                    # adding these observations to the new node
                    obs = Node.Observation(atGoal=atGoal,
                                           crashed=crashed,
                                           speeding=speeding)
                    nextNode.obs = obs
```

```
                    # need to add nextNode to the adj list of the node
                    # that reached nextNode (currNode), then get ready
                    # to build up nextNode
                    currNode.adjList.append(nextNode)
                    nodeQueue.append(nextNode)

        else:
            self.DFA = DFA.DFA(nodes=Nodes, startNode=initNode)
            return

#
# @brief      Calculates a boolean for whether the car is speeding
#
#             This boolean flag is used as one of the observations of the
#             TransitionSystem itself.
#
# @param      self                 The TransitionSystem object instance
# @param      prevLane             The lane the car was in during the
#                                  previous time step
# @param      prevVel              The previous velocity the car was
#                                  traveling at during the previous time
#                                  step
# @param      carX                 The current carX value ~ lane on highway
# @param      allowedVelsPrevLane  The set of legal velocities allowed in
#                                  the prevLane lane
# @param      allowedVelsCarXLane  The set of legal velocities allowed in
#                                  the carX lane
#
# @return     @bool indicating whether or not the car was / is speeding in
#             the prior or current time step
#
def speeding(self, prevLane, prevVel, carX, allowedVelsPrevLane,
             allowedVelsCarXLane):

    if (prevVel in allowedVelsPrevLane) and \
       (prevVel in allowedVelsCarXLane):
        return False
    else:
        return True

#
# @brief      Calculates a boolean for whether the car is in one of the
#             goal states along the road.
#
#             A goal state is a lane (x) and horizontal distance down the
#             highway from the starting location (Y). These correspond to
#             an exit on the highway the driver would like to use. This
#             boolean flag is used as one of the observations of the
#             TransitionSystem itself.
#
# @param      self        The TransitionSystem object instance
# @param      carX        The current carX value ~ lane on highway
# @param      carY        The current carY value ~ distance down highway
# @param      goalStates  The set of goal states in x and y
```

```python
#
# @return     @bool indicating whether or not the car is in one of the set
#             of goalStates during current time step
#
def inGoalStates(self, carX, carY, goalStates):

    minYGoalState = goalStates[0][1]
    xGoalState = goalStates[0][0]
    if (carX == xGoalState) and (carY > minYGoalState):
        return True
    else:
        return False

#
# @brief      Calculates a boolean for whether the car has crashed into
#             another one of the obstacle cars along the highway
#
#             This boolean flag is used as one of the observations of the
#             TransitionSystem itself.
#
# @param      self                The TransitionSystem object instance
# @param      prevLane            The previous lane for the car during the
#                                 last time step
# @param      prevVel             The previous velocity for the car during
#                                 the last time step
# @param      carX                The current carX value ~ lane on highway
# @param      carY                The current carY value ~ distance down
#                                 highway
# @param      carT                The current carT value ~ current time
#                                 step
# @param      minSpeedInPrevLane  The minimum legal speed in the previous
#                                 lane
# @param      minSpeedInCarXLane  The minimum legal speed in the carX lane
# @param      POS                 The POS (physical occupancy set) object
#                                 which contains the 3D projection of a
#                                 Node state onto the x, y, and time grid
#                                 for the empty road.
#
# @return     @bool indicating whether or not the car will crash into
#             another driver during the previous -> current time step if
#             they take a certain control action
#
def crashed(self, prevLane, prevVel, carX, carY, carT,
            minSpeedInPrevLane, minSpeedInCarXLane, POS):

    # time steps are unit length for simplification
    timeStepLength = 1
    prevY = carY - prevVel * (timeStepLength)
    prevT = carT - timeStepLength

    # POS[x, y, t] = True (1) if another car is at the specific (x,y)
    # location of the road at time t
    distDownTheHwy = prevVel - minSpeedInPrevLane
    sumPrevLane = 0
```

```python
    for ii in range(0, distDownTheHwy):
        sumPrevLane += POS[prevLane, prevY + ii, prevT]

    distDownTheHwy = prevVel - minSpeedInCarXLane
    sumCarXLane = 0
    for ii in range(0, distDownTheHwy):
        sumCarXLane += POS[carX, prevY + ii, prevT]

    numCollisions = sumPrevLane + sumCarXLane

    if numCollisions > 0:
        return True
    else:
        return False

#
# @brief      Returns a tuple of physically possible lanes to change to
#
# @param      self      The TransitionSystem object instance
# @param      currLane  The car's current lane number
# @param      allLanes  A tuple of all of the different lm
#
# @return     The physically meaningful, allowed adj lanes from currLane
#
def getAdjLanes(self, currLane, allLanes):

    maxLane = max(allLanes)
    minLane = min(allLanes)

    if currLane == maxLane:
        return (currLane - 1, currLane)
    elif currLane == minLane:
        return (currLane, currLane + 1)
    else:
        return (currLane - 1, currLane, currLane + 1)
```

G. LDBA.py

```python
import Node
import DFA


class LDBA:
    # @brief implementation of the LTL Deterministic Buchi Automata

    #
    # @brief      Constructs the LDBA object.
    #
    # @param      self        The LDBA object instance
    # @param      LTLFormula  The ltl formula to contruct this automata
    #                         automatically
    #
    def __init__(self, LTLFormula):

        state0 = Node.NodeState(q=0)
        state1 = Node.NodeState(q=1)
        state2 = Node.NodeState(q=2)

        node0 = Node.Node(state=state0, index=0, isAccepting=False)
        node1 = Node.Node(state=state1, index=1, isAccepting=False)
        node2 = Node.Node(state=state2, index=2, isAccepting=True)

        accepts = [2]

        startNode = node0

        Nodes = []
        Nodes.append(node0)
        Nodes.append(node1)
        Nodes.append(node2)

        def transFcn(q, obs):
            if q == 0:
                if obs.atGoal and (not obs.crashed) and (not obs.speeding):
                    return 2
                elif obs.crashed or obs.speeding:
                    return 1
                else:
                    return 0
            elif q == 1:
                return 1
            elif q == 2:
                if obs.crashed or obs.speeding:
                    return 1
                else:
                    return 2

        self.DFA = DFA.DFA(nodes=Nodes, startNode=startNode,
                           transFcn=transFcn, accepts=accepts)
```