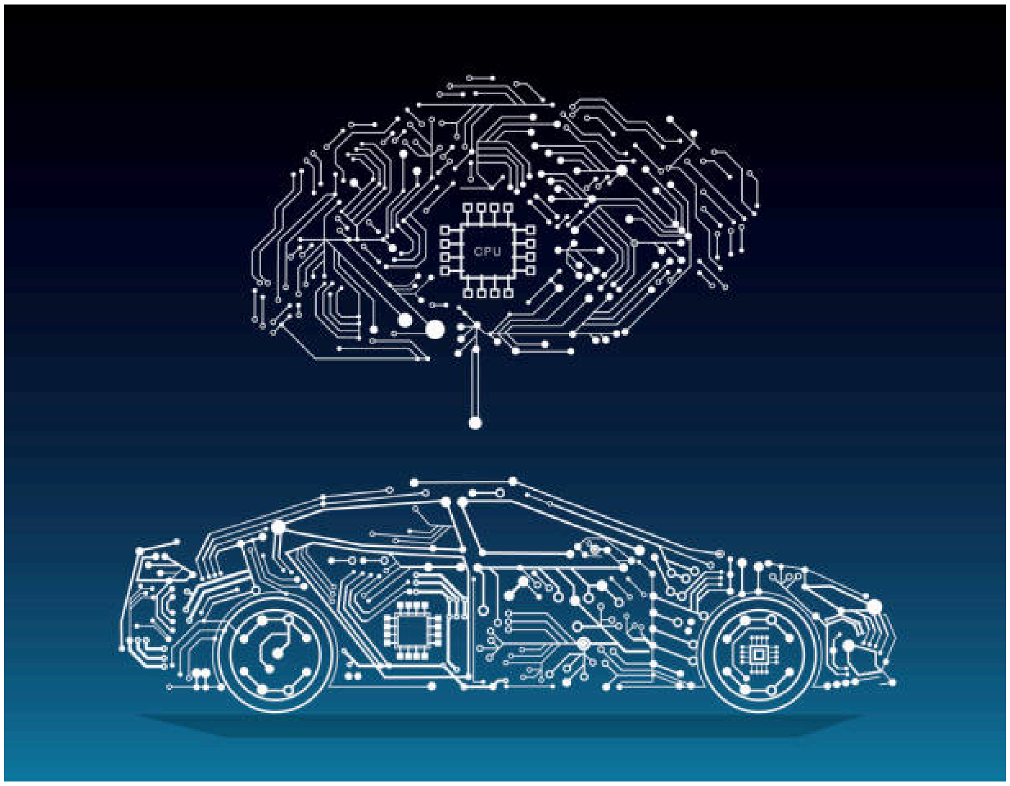
Hugs the Lanes

Team Name: Vehicular Vision ©

Members: Annabelle Crescenzo, Paige DiGirolamo, Skye Jorgensen, Kailey Totland



Phase 1: April 23, 2023

A Project by Alset IoT

TABLE OF CONTENTS

1. Introduction

No longer a thing of sci-fi films, self-driving cars are becoming a popular technology of the present. By the mid 2010s, multiple companies began researching and investing in autonomous vehicles, however full autonomy has proved challenging. Self-driving cars are ranked from zero to five with zero meaning no autonomous features and five meaning fully automated. Currently, there are no cars available for the general public's purchase beyond level two. A major ride sharing app, Uber, decided to withdraw from the self-driving technology due to lawsuits and safety concerns. Tesla has developed a popular autonomous package for its cars that allows for hands-free control during highway and freeway driving, but it has also experienced lawsuits over multiple accidents related to its self-driving package. Our research team Vehicular Vision is not afraid of the challenge and is actively designing a new software for self-driving cars of the future that will surpass the efficiency and safety of current models.

1.1 Purpose

This project comes at a critical point in the development of self-driving cars. As these cars and other autonomous vehicles begin to hit the market and become increasingly more accepted by society, it is vital that projects like this one are carried through to ensure that these vehicles are safe, efficient, reliable, and have a place in the infrastructure of the world. The cars produced by Vehicular Vision offer many features that most competitors lack. This hole in safety and reliability in the autonomous vehicle market needs to be addressed. This project places emphasis on these issues and uses them as a jumping ground for development. Unlike other companies that focus on profit and aesthetics, we use most of our energy and resources on safety, reliability, and availability while still having enough focus on appearance. This project is a mission critical real-time embedded system. Our software is real time because it makes decisions in real time, like checking the distance between the car and an obstacle and automatically braking in response. If the decision is not reached in the appropriate time an accident could occur; this software failure might cause catastrophic consequences, such as death and property damage, making this project mission critical. This emphasizes why our team is motivated to create an available and reliable system that focuses on safety. The cars developed from this project bring a new, improved focus to the market that gives customers more options and safer autonomous cars for people.

1.2 Features

Many of the features surrounding the design of our IoT vehicle are based on the actions and responses humans need in order to drive a non-autonomous vehicle. However, humans are not perfect. They often make mistakes when driving cars, endangering the lives of themselves and others. For this reason, the features planned out for our vehicle are more closely based on the ideal response of human drivers, rather than the average response. Listed below are several features that we plan to implement. However, they will not all be available in phase one.

1.2.1 Functional

One of the most important features the car must have is the ability to sense its surroundings and environment. Our vehicle is able to track the direction of the road, and adjust its course accordingly to remain in the center of the lane, as the road changes. When making turns it accurately understands which side of the road it has to remain on. This aligns with following all driving laws, such as complying with lights, signs, lane changes, other drivers, ambulances, and other emergency vehicles. Another important feature of sensing surroundings is sensing unexpected obstacles. This ranges from people, animals, and debris. In response to these factors, the car should also be able to adjust its course appropriately. Based on where the passenger is in their journey, the car automatically is able to shift gears, just as any other car has the ability to do so. The rest of the functional features include features that are necessary to aid the driver. These features include: automatic headlights that sense the brightness of its environment, tire pressure monitors and notifications, and fuel gauge notifications. The dashboard of the car has a display system to notify the driver of various things. If the car is due for an inspection, then a signal lights up on the dashboard. There is a blindspot signal that goes off when the vehicle detects there is something in that area. The display also shows the speed limit of the road to remind the driver. This updates as the speed limit changes. A cruise control light also appears when the car is in cruise control mode. The dashboard of the car acts as a form of communication between the driver and the vehicle.

1.2.2 Technical

Many of the other features in our vehicle include the additional technology that improves the reliability of the driving experience for all those that purchase the vehicle. The vehicle is connected to a sort of database that collects data on the car and its surroundings. Since the car is driving based on the destination given by the passenger, the information obtained throughout the drive can be used to help others traveling on similar routes. Some of this information includes frequent obstacles, construction, and hazardous conditions of roads. The pairing of the functional sensors of the car with collecting data allows other passengers of the vehicle to enjoy a readjusted course. These additional safety features can only be provided from the advanced architecture of IoT products. Information about the car itself also plays an important component in future software improvements. By collecting data while the vehicle is operating, there is almost a constant flow of information that demonstrates when the car is operating at its peak performance and where it must be changed. This allows us to improve in different areas, for future models. The connection to a larger network also allows the car to easily obtain these specific software improvements, as soon as they are developed.

1.3 Software Development Process

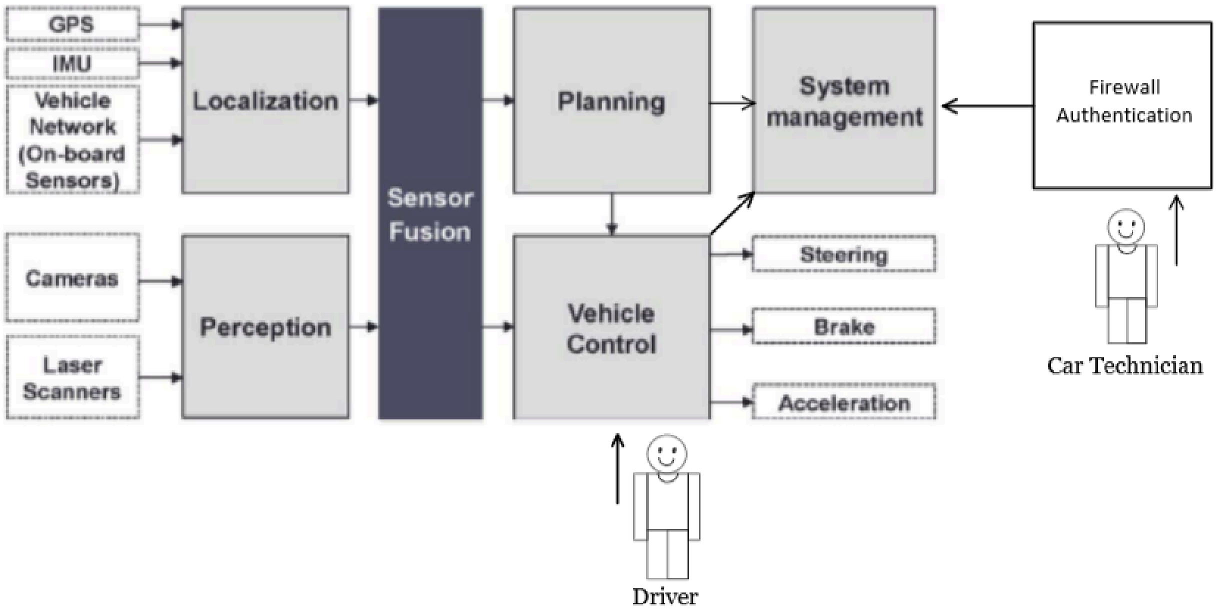
To ensure our team's success, we are using an evolutionary software development process called the spiral model, which was initially proposed by Boehm. Our emphasis is on the first release of our product, but this would be followed by several other releases in the coming years. These releases will occur incrementally to produce a finalized software that is available for general use. The spiral model is an adaptable and a continuous process, so it can be applied to our product as our self-driving car design is modified over time. Each spiral starts with the communication phase. We will discuss our approach to designing some of the features of our self-driving car and its function. Our focus will be on safety and the best way we can implement the features in an effective and secure manner. We will then move onto the planning stage of the spiral, where we will schedule and set deadlines for an organized process. Continuing to the modeling stage, we will begin designing what mechanical features are being used and how the software interacts with them. After writing a complete pseudocode, we will transition into the construction stage, where we will translate our pseudocode into a programming language that can be tested. During the deployment stage, we will deliver that code for feedback from our team members. If changes are suggested, we will go around the spiral again to examine and implement those changes. Phase two starts with the communication stage again to discuss the next features we want to implement and continue through the spiral model with each phase. This software development process allows us to develop high-quality software products in an iterative method so we can secure the safety of our self-driving cars.

1.4 Team Qualification

The team behind this project consists of four experienced and knowledgeable members who all bring different perspectives and ideas to the table. All four members have studied computer science at Stevens Institute of Technology and have used a variety of programming languages throughout their college and personal experience, such as Python, C/C++, Java, JavaScript, Scheme, R, and OCaml. Our team consists of capable, problem-solving members who strive for success given any challenge. We have a mix of people that are detail oriented and some that focus on big picture plans. We also have more technologically experienced people and members that have worked on group projects multiple times before. All four members of our team are hardworking, responsible, and dedicated to success. Most importantly, we trust each other and value the skills and unique ideas each of us bring.

2. Functional Architecture

Functional architecture is a map of the system's modules and their interactions with one another, and it illustrates what each module performs to complete a task. For a self-driving car, its main tasks are to drive with and without driver assistance, to be trustworthy and capable of self-driving, and to be easy to interact as a driver observes or takes control. Our self-driving car's functional design is illustrated below.



Our team's goal is to create a level three autonomous vehicle following this functional architecture design. This model breaks down the system into six separate components and shows how data flows between each module. The six modules are localization, perception, sensor fusion, planning, vehicle control, and system management. The white boxes represent the inputs and outputs of the system.

2.1 Perception and Localization

The architecture for our autonomous vehicle can be broadly described as the combination of several sensors and processors. Going deeper, we can divide these sensors into two groups. One type of sensor deals with the vehicle itself, and the other focuses on the surrounding environment. Sensors like GPS, IMU, Vehicle Network, etc. that give details about the vehicle itself go to the Localization module. This module deals with the direction and position of the vehicle. Cameras and lasers are types of sensors that relate information about the environment by telling the Perception module what is around the car. Cameras show visual information about the vehicle's surroundings, while the LiDAR laser provides 3D information to complement the visual information.

2.2 Sensor Fusion

These sensors relay information to either the Localization module or the Perception module. The Localization and Perception modules send this information to the Sensor Fusion. Sometimes, information feeded into the Localization and Perception modules is unclear or inconsistent, and it is the sensor fusion's job to clean this data up. It calculates averages and approximations from the given data. This minimizes uncertainty and offers a clearer picture of reality for the vehicle. The Sensor Fusion also puts all this information on a timeline so these events become something happening at a specific time. Without the Sensor Fusion, the data from the sensors would have no meaning and just be floating numbers. The Sensor Fusion grabs this data, cleans it, and then synchronizes the data to a timeline to allow it to be processed by other modules.

2.3 Planning Module

The calculations and data discussed above move from the Sensor Fusion to the Planning module. The Planning module is the brain of the software. It analyzes the information and makes a conclusion based on the situation and the environment. While the Sensor Fusion makes sense of the data, the Planning module deciphers what this data means about the vehicle and makes a decision to counteract any problems. These decisions are used to form instructions that allow the car to react in an appropriate manner to any issues it may experience. As an example, if the laser sensed an obstacle in the middle of the road, this data would feed through the Perception module, the Sensor Fusion, and eventually the Planning module. The Planning module would calculate that there is a risk of a collision and form instructions to avoid this. This could be telling the car to brake or steer away. These instructions are sent to Vehicle Control, which enacts these counteracting measures to avoid accidents and maintain passenger safety.

2.4 Vehicle Control

Vehicle Control executes the planned motion by implementing the vehicles' actuators. It has access to mechanisms that change the movement of the car, such as steering, acceleration, and braking. This module could be broken down further based on the direction of planned action. Longitudinal control, which includes propulsion and braking, impacts the forward and backward motion of the car on the road. Lateral control, which includes steering, impacts the side to side direction of the car. For example, if there is a stop sign or red traffic light, the Planning module tells vehicle control to slow to a stop. If the car needs to cross lanes, the Planning module tells the vehicle control to steer left or right. There is also reactive control, which deals with sudden decisions to avoid collisions. This would occur when a person steps in front of the car, and the Planning module tells Vehicle Control to apply emergency brakes.

Even though our goal is to create a high-level autonomous vehicle, we still plan for a driver to be present. This person can override the decision made by the Planning module and command Vehicle Control directly. Vehicle Control is run by software, so when the driver alters the movement of the car, these actions are sent as signals to Vehicle Control. When the driver wants to slow down, they press on the brake pedal which tells vehicle control to apply the brakes. This allows for human intervention to occur at any time.

2.5 System Management

The Planning and Vehicle Control module sends logs of the car's recent status and discernable events to System Management, which maintains the log, performs backups to the Cloud, and interacts with the technician. The technician has a user identification and password to get access to the log file, which contains a history of the car's activity and performance. The log file is automatically backed up and sent to the Cloud when the car is connected to a network. There are potential cybersecurity drawbacks with the log being connected to a network. The log is sensitive to hacking and malware, and if the car's software is accessible via the network, then the vehicle could also be hacked. If a hacker with malicious intent breaks into the operating system, the car owner's personal information could be leaked online. However, an authorized user, like a car technician, should have access to the car's software and logs to determine issues in the sensors or vehicle control. Having the Firewall Authentication module emphasizes our focus on the safety and security of our passengers.

2.6 Level of Autonomous Features

The scale that measures autonomous features in vehicles is not new. With zero having no autonomous features and five being completely autonomous, there is a lot of range between the two extremes that can be explored within the development of our vehicle. Since society has not developed regulations and laws for fully autonomous vehicles, it would not be practical to plan for such a design. Instead our team is planning for a level three vehicle on the autonomous scale. This means that while we are developing autonomous features, we wish to combine these features with the capabilities and skills of human drivers. Based on the current trend of vehicles today, however, our vehicle depends mostly on the driver. The autonomous features serve more of a role in elevating both safety and comfort. This is demonstrated in the image presented on page 6 of the document. Both the sensing of the vehicle's location and surroundings are extremely useful tools to a driver. This information is essential in allowing drivers to make the safest and the most reliable decisions behind the wheel. In both emergency scenarios, such as unexpected obstacles, and non-emergency scenarios, such as typical navigation, the constant availability of information for the driver is essential for our team's goal in development. The information also flows to the System Management. By having data accessible to an outside system and vice versa, our systems always are able to improve the software of the vehicles through updates.

3. Requirements

These requirements assume that all of the mechanical features are working as they should and are not broken.

3.1 Functional Requirements

3.1.1 Detect an object in front of the car and apply brake to avoid collision

Preconditions

- The car is on and in drive
- The car's speed > 0 mph
- The driver is present

Post Conditions

- If an object is present and within 1000 feet of the car, the car stops within 5 feet of the object, avoiding collision

Requirements

3.1.1.1 Object Sensor detects an object at L feet away
3.1.1.2 Object Sensor sends object data (proximity, size) to Sensor Fusion
3.1.1.3 Sensor Fusion normalizes data and sends to Planning
3.1.1.4 Planning sends brake request to the Vehicle Control System (VCS)
    3.1.1.4.1 When L = 200 feet, brake intensity one is requested
    3.1.1.4.2 When L = 150 feet, brake with intensity two is requested
    3.1.1.4.3 When L = 100 feet, brake with intensity three is requested
    3.1.1.4.4 When L = 50 feet, brake intensity four is requested
    3.1.1.4.5 When L = 10 feet, maximum brake intensity five is requested
3.1.1.5 If the object is no longer present, then the car does not brake
3.1.1.6 IOT HTL logs the event in System Management

3.1.2 Automated Parking Assistance

Preconditions

- The car is on and in drive
- The car's speed < 10 miles per hour
- The driver is present

Post Conditions

- The car is fully stopped within the confines of the parking spot.

Requirements

3.1.2.1 Parallel Parking
    3.1.2.1.1 Front and rear cameras collect information about positioning of the car

3.1.2.1.2 Location and distance data from the Perception module is sent to Sensor Fusion
3.1.2.1.3 Sensor Fusion normalizes data and sends to Planning
3.1.2.1.4 Planning determines the steering and acceleration requests to be sent to VCS based on the positioning of the car
3.1.2.1.5 Planning sends brake requests to VCS once car is correctly positioned in the parking spot,
3.1.2.1.6 VCS applies brakes so that car is completely stopped
3.1.2.1.7 IOT HTL logs the event in System Management

3.1.2.2 Backing Into Parking Spot
3.1.2.2.1 Rear cameras collect information about positioning of the car
Remaining 3.1.2.2.2 to 3.1.2.2.7 are same as Parallel Parking

3.1.2.3 Pulling Into Parking Spot
3.1.2.3.1 Front cameras collect information about positioning of the car
Remaining 3.1.2.3.2 to 3.1.2.3.7 are same as Parallel Parking

3.1.3 Automatic Headlights

Preconditions
- The car is on
- The driver is present
- Headlights are at incorrect light level

Post Conditions
- The car's low beams or high beams are turned on, or the headlights are turned off depending on the level of brightness outside.

Requirements
3.1.3.1 Sensors and camera collect data measure ambient light of level L lux
3.1.3.2 Ambient light levels flow from the Perception Module to the Sensor Fusion
3.1.3.3 Sensor Fusion normalizes data and sends to Planning
3.1.3.4 Planning sends request to VCS to turn headlights on or off depending on value of L
3.1.3.4.1 When $L > 1000$ lux, turn headlights off
3.1.3.4.2 When 200 lux $\leq L \leq$ 1000 lux, turn low beams on
3.1.3.4.3 When $L < 200$ lux, turn high beams on
3.1.3.5 Planning continues to update requests depending on the ambient lighting
3.1.3.5 IOT HTL logs the event in System Management

3.1.4 Blind Spot Warning

Preconditions

- The driver is present
- The car is on and moving ≥ 20 mph

Post Conditions

- The car alerts the driver through an icon on the dashboard if there is a vehicle in or approaching the driver's blindspot, so that the driver can respond accurately and avoid collision

Requirements

3.1.4.1 Sensors and cameras detect a vehicle within 5 feet of either side of the vehicle

3.1.4.2 Sensors send information (location, proximity, size) to the Sensor Fusion

3.1.4.3 Sensor Fusion normalizes the information and sends to Planning

3.1.4.4 Planning sends a request to VCS that lights up the Blind Spot Icon on the dashboard

3.1.4.5 VCS lights the Blind Spot Icon on the dashboard to alert the driver there is a vehicle present in their blind spot

3.1.4.6 IOT HTL logs the event in System Management

3.1.5 Speed Limit Sign Detection

Preconditions

- Driver is present
- The car is on and moving ≥ 5 miles per hour
- Cruise Control is on
- The car is being driven on roads with standard speed limit signs
- There is a speed limit sign in miles per hour on the right hand side of the road

Post Conditions

- The car adjusted its speed to either go faster or slower or remain the same based on the new speed limit sign that it detected

Requirements

3.1.5.1 Cameras in the front of the car that are pointing 45 degrees to the right, take pictures of the side of the road diagonal to the car

3.1.5.2 The cameras send the pictures taken to Perception, which then sends them to the Sensor Fusion

3.1.5.3 The Sensor Fusion normalizes the pictures and data and sends it to Planning

3.1.5.4 Planning uses AI to tell if there is a speed limit sign in the pictures taken by the cameras

3.1.5.5 If a speed sign is detected, the number on it is used as the speed limit baseline and Planning sends commands to VCS that either applies the brakes or the accelerator

3.1.5.5.1 If the new speed limit is less than the current speed of the car by 10 miles per hour, then the brakes are applied with intensity 1

3.1.5.5.2 If the new speed limit is less than the current speed of the car by 20 miles per hour, then the brakes are applied with intensity 2

3.1.5.5.3 If the new speed limit is less than the current speed of the car by greater than 20 miles per hour, then the brakes are applied with intensity 3

3.1.5.5.4 If the new speed limit is greater than the current speed of the car by less than 10 miles per hour, then the accelerator is applied with intensity 1

3.1.5.5.5 If the new speed limit is greater than the current speed of the car by greater than 10 miles per hour, then the accelerator is applied with intensity 2

3.1.5.6 Driver has the control over the car to continue driving same speed if they do not wish to slow down or speed up according to the speed limit change

3.1.5.7 The new speed limit lights up on the dashboard of the car so the driver is aware of the change

3.1.5.8 IOT HTL logs the event in System Management

3.1.6 Car Approaches a Traffic Light and Acts Accordingly

Preconditions

- Driver is present
- The car is on and moving ≥ 5 miles per hour
- The car is in cruise control mode

Post Conditions

- The car stops if the traffic light is red
- If the light is yellow, the car either slows down or keeps going
- The car does not accelerate or brake if the light is green

Requirements

3.1.6.1 Cameras in the front of the car that are pointed 45 degrees upwards take pictures of the road in front of the car

3.1.6.2 The cameras send the pictures taken to Perception

3.1.6.3 The pictures then move from Perception to the Sensor Fusion

3.1.6.4 The Sensor Fusion normalizes the pictures and sends them to Planning

3.1.6.5 Planning analyzes the pictures taken and uses AI to detect traffic signals in the pictures that are facing the road the car is driving in

3.1.6.6 If Planning detects a traffic signal, it analyzes the light that is on

3.1.6.6.1 If the light is green, the cars speed does not change and no commands are sent to VCS

3.1.6.6.2 If the light is yellow, the car detects the distance from the traffic light to the car. Planning does calculations to see if the car can continue at that speed to go past the traffic light and intersection within the three seconds yellow lights normally are.

3.1.6.6.2.1 If the current speed is enough to make it past the traffic light and no cars in front are slowing down, then the car does not slow down and no commands are send to VCS

3.1.6.6.2.2 If the current speed is enough to make it past the traffic light, but cars in front are slowing down, then the car brakes

3.1.6.6.2.3 If the current speed is not enough to make it past the intersection, then Planning sends commands to VCS that the brakes are applied

3.1.6.6.3 If the light is red, then the brakes are applied and the car stops either behind the other cars or at the lines that are in front of traffic lights, designating where to stop on red

3.1.6.7 IOT HTL logs the event in System Management

3.1.7 Lane Departure Adjustments

Preconditions

- Driver is present
- The car is on and moving ≥ 40 miles per hour
- The car detects lane markers on either side of the vehicle

Post Conditions

- The car remains moving inside the lane

Requirements

3.1.7.1 Cameras behind the rearview mirror take pictures of the ground in front of the car

3.1.7.2 The cameras send the pictures to Perception Module

3.1.7.3 The pictures are then sent to Sensor Fusion

3.1.7.4 Sensor Fusion normalizes the pictures and sends them to Planning

3.1.7.5 The Planning Module analyzes the photos and detects the white or yellow lines on either side of the car

3.1.7.5.1 If the distance between the center of the car and the left white/yellow line is with one foot of the distance between the center of the car and the right white/yellow line, then no commands are sent to VCS and the cars motion stays the same

3.1.7.5.2 If the distance between the center of the car and the left white/yellow line is greater than one foot of the distance between the center of the car and the right white/yellow line, then Planning Module sends instructions to the VCS to adjust the steering of the car to move to the left to realign the car to be in the center. So it moves to the left the same distance that it is skewed to the right side

3.1.7.5.2 If the distance between the center of the car and the right white/yellow line is greater than one foot of the distance between the center of the car and the left white/yellow line, then Planning Module sends instructions to the VCS to adjust the steering of the car to move to the right more and recenters itself

3.1.7.6 IOT HTL logs the event in System Management

3.1.8 Automatic Windshield Wipers

Preconditions

- The car is on
- The driver is present
- It is raining/snowing/hailing

Post Conditions

- The car's windshield wipers are turned on or off depending on the amount of precipitation outside.

Requirements

3.1.8.1 Sensors and camera collect data on the amount of raindrops per minute R on the windshield

3.1.8.2 Data flows from the Perception Module to the Sensor Fusion

3.1.8.3 Sensor Fusion normalizes data and sends to Planning

3.1.8.4 Planning sends request to VCS to turn windshield wipers on/off

3.1.8.4.1 more than 400 rain drops per minute = windshield wipers are turned on intensity 4

3.1.8.4.2 300 rain drops per minute = windshield wipers are turned on intensity 3

3.1.8.4.3 200 rain drops per minute = windshield wipers are turned on intensity 2

3.1.8.4.4 100 rain drops per minute = windshield wipers are turned on intensity 1

3.1.8.4.5 No rain drops detected in a five minute interval, turn off windshield wipers

3.1.8.5 IOT HTL logs the event in System Management

3.1.9 Traction Control Onset

Preconditions

- The car is on

- The driver is present
- There are bad weather conditions recognized from the Sensor Fusion OR Yaw sensors have determined a problem in the velocity of one or more wheels

Post Conditions

- The velocities of the wheels are equalized

Requirements

3.1.9.1 Yaw sensors determine that the velocity of the wheels are indicating that one or more wheels is spinning faster than the others

3.1.9.2 Data flows from the Localization Module to the Sensor Fusion

3.1.9.3 Sensor Fusion normalizes data and sends to Planning

3.1.9.4 Planning sends request to VCS to automatically
slow the speed of the wheel

3.1.9.4.1 If the wheel's velocity is faster than the average of the other 3 wheels, slow the wheel's velocity to the average

3.1.9.4.2 If the wheel's velocity is less than the average of the other 3 wheels, no change the wheel's velocity

3.1. 9.5 IOT HTL logs the event in System Management

3.1.10 Cruise Control

Preconditions

- Car is on and in drive
- Cruise Control switch is deactivated
- GPS recognizes nearby desired turns as no sharper than 100 degree angles

Postconditions

- Cruise Control is activated
- Desired speed is maintained by cruise control
- Cruise Control turns off when driver steps on accelerator pedal

Requirements

3.1.10.1 Driver turns on the cruise control switch

3.1.10.2 Sensor fusion senses this change and sends a message to Planning, telling it that cruise control has been switched on

3.1.10.3 Cruise Control light shows on dashboard to indicate to driver cruise has been successfully turned on

3.1.10.4 Planning waits for Driver to input speed

3.1.10.5 Driver then uses steering wheel commands (user interface) to set designated speed

3.1.10.6 Sensor Fusion senses this action and sends desired speed to Planning

3.1.10.7 Planning accumulates data received and calculates necessary action in both acceleration and braking to get to desired speed

3.1.10.8 VCS receives Plannings requests and controls acceleration and braking so that desired speed is achieved and maintained

3.1.10.9 IOT HTL logs the event and actions of accelerator in System Management

3.2 Nonfunctional Requirements

3.2.1 Software Update

Preconditions

- The car is on and in park
- The car is connected to the network

Postconditions

- New software is installed

Requirements

3.2.1 User (technician or driver) logs in via the Firewall Authentication to install software update.

3.2.2 User accepts or denies the new software update

3.2.2.1 If accepted, restart and upload new version

3.2.2.2 If denied, maintain current version

3.2.3 Indicate completion of update

3.2.2 User Interface

3.2.1 Dashboard dials

3.2.1.1 Visual display for fuel level

3.2.1.2 Visual display for speed

3.2.2 Touch screen display

3.2.2.1 Interactive screen for GPS/map

3.2.2.2 Interactive screen that lists common and saved destinations

3.2.2.3 Interactive screen that lists saved contacts/phone book

3.2.2.4 Interactive screen for radio stations/bluetooth

3.2.2.5 Interactive screen that displays temperature and weather alerts

3.2.3 Indicators

3.2.3.1 Light to indicate headlights on

3.2.3.2 Light to indicate obstacle in blind spot detection

3.2.3.3 Light to indicate cruise control on

3.2.3.4 Light to indicate issues within the software and that it is time for a check in with a technician

3.2.3 Reliability

3.2.3.1 The functional requirements listed in the section above will not fail more than once per 100,000 miles of operation
3.2.3.2 If there are any minor issues with part of the software, the log in System Management indicates these issues and the technician is able to see them during routine maintenance
3.2.3.3 If there are any critical issues within the software, an indicator lights up on the dashboard, alerting the driver before any damage
3.2.3.4 A user with a valid login (username and password) can retrieve the log file and perform diagnostics

3.2.4 Performance

3.2.4.1 Sensors sends data to sensor fusion every 100 ns
3.2.4.2 The system is capable of supporting 1000 sensors.

3.2.5 Security

3.2.5.1 Locks on all doors activated within 1 minute of moving vehicle.
3.2.5.2 Firewall on the System Management so only a user with a valid login (username and password) can access the system.
3.2.5.3 Locks engaged in shutdown vehicles within 30 minutes.
3.2.5.4 If the car is already turned on, the car turns off automatically if keys are not sensed within 50 feet.

4. Requirement Modeling

4.1 Use Case 1: User Login

4.1.1 Use Case Scenario

4.1.1.1 Actors

- Technician or Driver

4.1.1.2 Trigger

- Driver goes to login on an account associated with the vehicle or using an employee access account

4.1.1.3 Preconditions

- Actor is not already logged into the system

4.1.1.4 Postconditions

- Actors are logged into account, either for a particular vehicle or for the overall system associated with the IOT vehicles

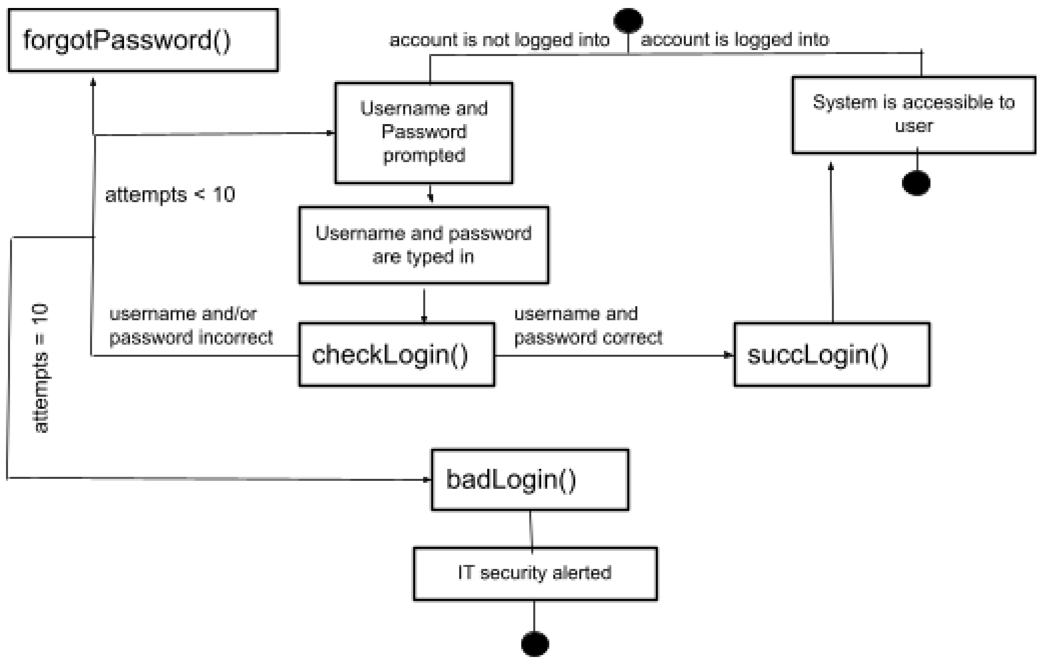
4.1.1.5 Steps of the Activity

1. The User utilizes the internet to get to the system login site
2. The User types in username associated with the system
3. The User types in the password associated with the account
4. Username is compared to the system's list of users
5. Password is verified as the correct password associated with the account
6. If either username or password are incorrect, the system displays a message to indicate error and prompts the User/Technician to try again
    a. Forgotten password feature can be activated
7. If both username and password are correct then User is granted access to the system and is successfully logged in

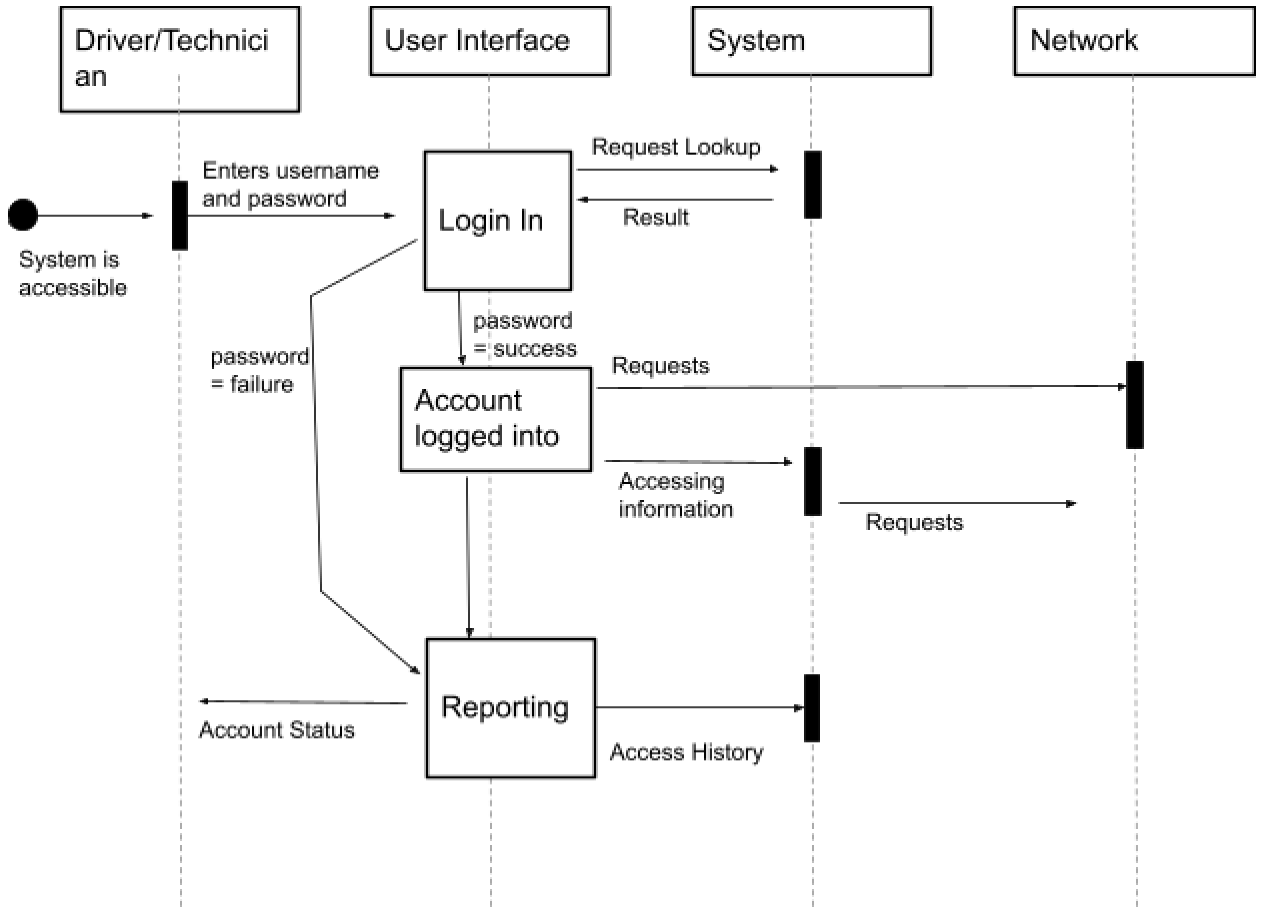
4.1.1.6 Exceptions

- If the username is correct and password is incorrect during 10 attempts, IT security is alerted for suspicious activity
- If a logged in account does not detect activity for 30 minutes, the account is automatically logged out of

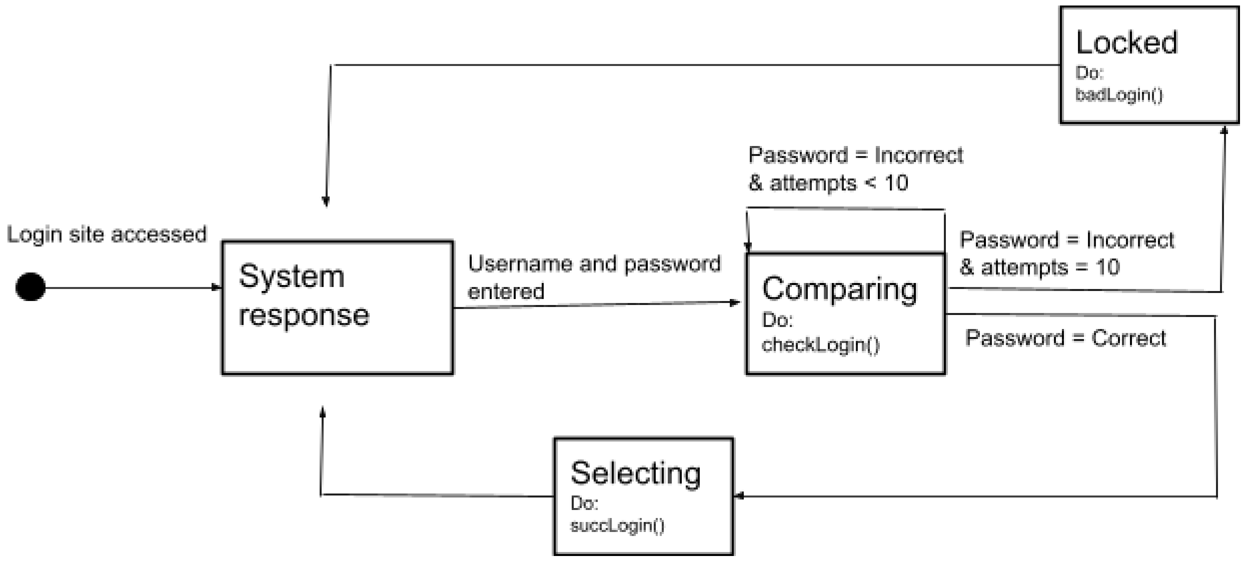
4.1.2 Activity Diagram


forgotPassword()
account is not logged into
account is logged into
Username and Password prompted
System is accessible to user
attempts < 10
Username and password are typed in
attempts = 10
username and/or password incorrect
checkLogin()
username and password correct
succLogin()
badLogin()
IT security alerted

4.1.3 Sequence Diagram



4.1.4 State Diagram



4.2 Use Case 2: Technician Completes a Software Update

4.2.1 Use Case Scenario

4.2.1.1 Actors

- Technician

4.2.1.2 Trigger

- Technician turns on car and selects software update under settings on the user interface

4.2.1.3 Preconditions

- The car is on and in park
- The car is connected to the network

4.2.1.4 Postconditions

- New software is installed successfully

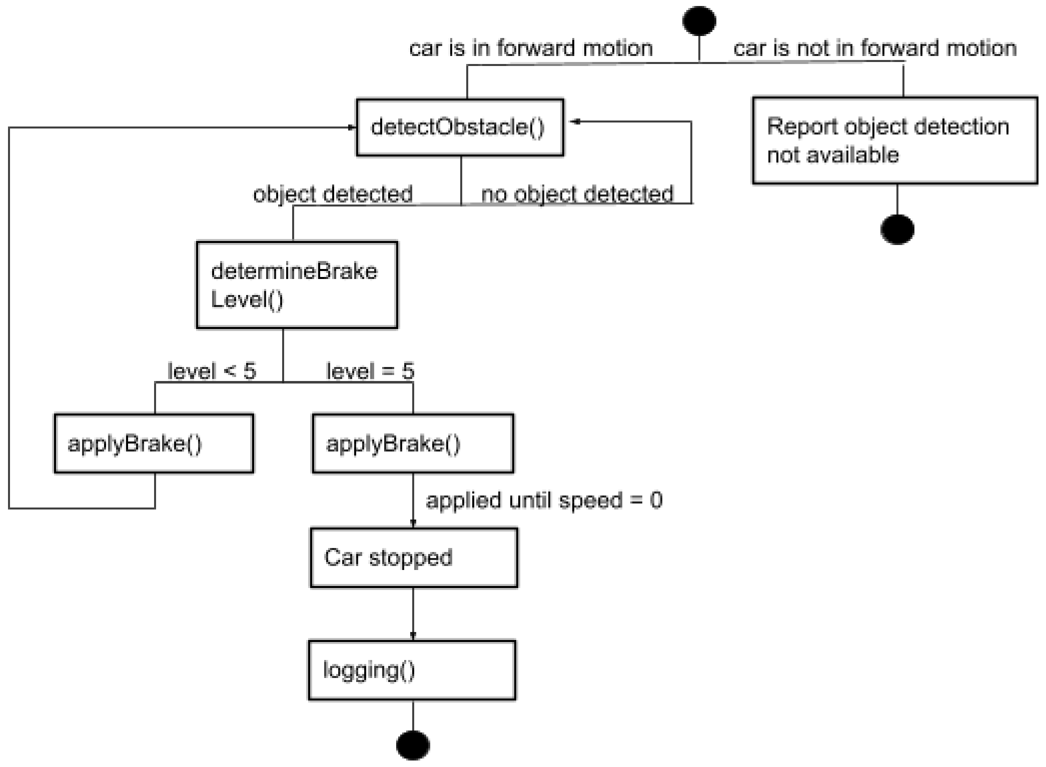
4.2.1.5 Steps of the Activity

1. The technician selects software update within the settings of the user interface
2. The technician enters the correct admin username/password
3. The Firewall Authentication module confirms the validity of the login
4. The Software Management module is successfully updated with the new software update

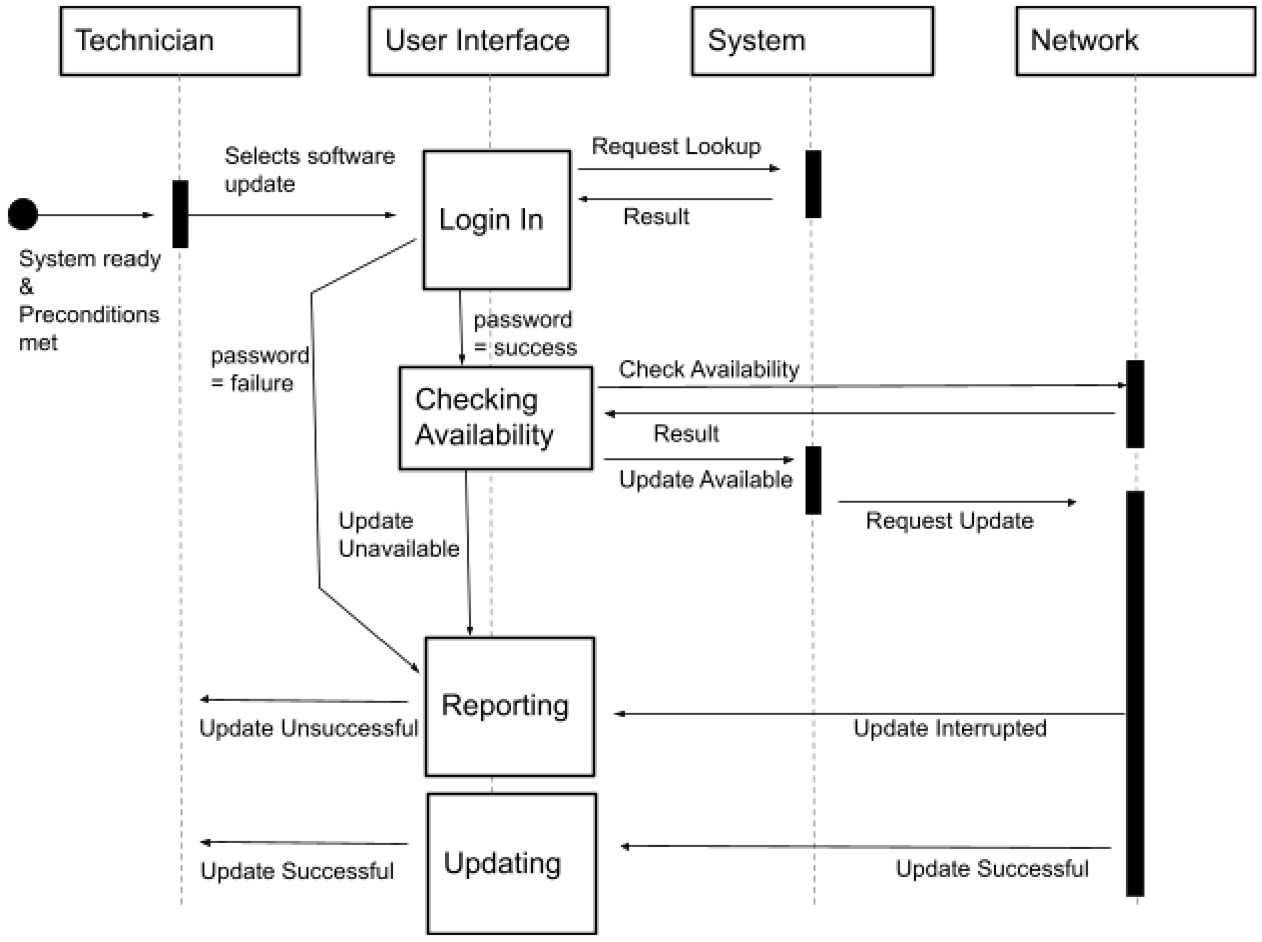
4.2.1.6 Exceptions

- No software update available
- Software update interrupted
- Software update attempted while car is in any other setting beside park (drive, neutral, etc)
- Incorrect admin username/password used by technician

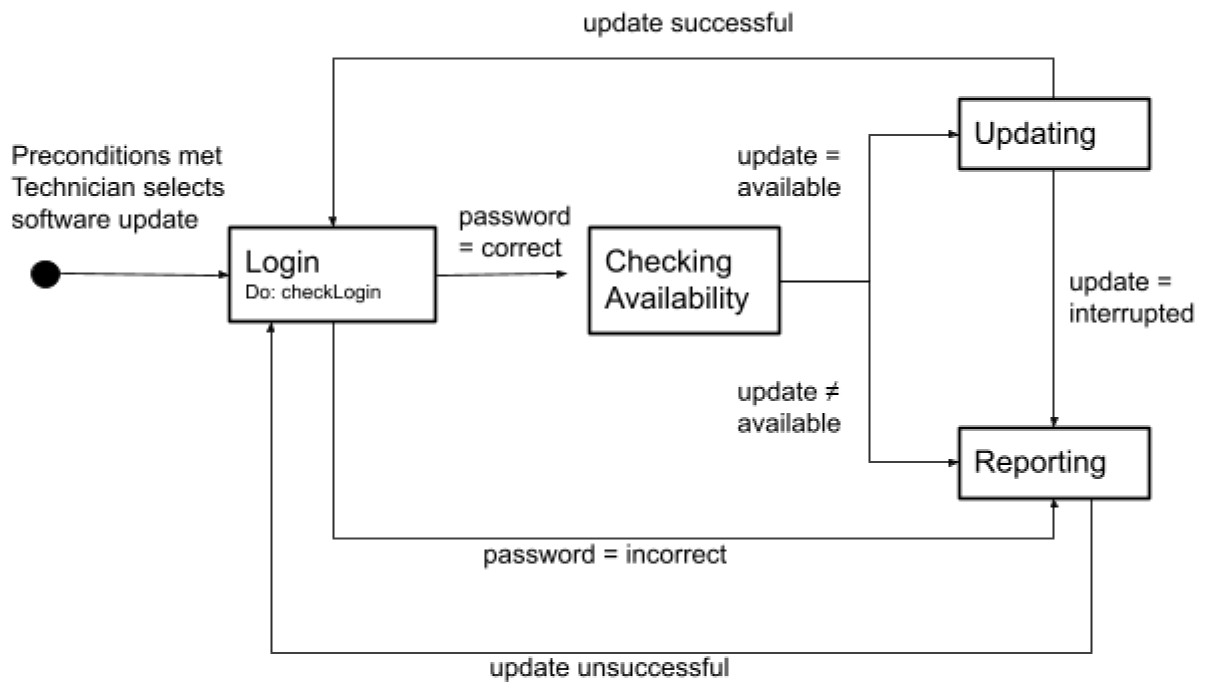
4.2.2 Activity Diagram



4.2.3 Sequence Diagram



4.2.4 State Diagram



4.3 Use Case 3: Object Detection is Enforced by Vehicle

4.3.1 Use Case Scenario

4.3.1.1 Actors

- Sensors

4.3.1.2 Trigger

- Sensors detect an object within 1000 feet in front of the car while it is in motion

4.3.1.3 Preconditions

- The car is on and in drive
- The car's speed > 0 mph
- The driver is present

4.3.1.4 Postconditions

- If an object is present and within 1000 feet of the car, the car avoids collision by applying brakes and stopping within 5 feet of the object

4.3.1.5 Steps of the Activity

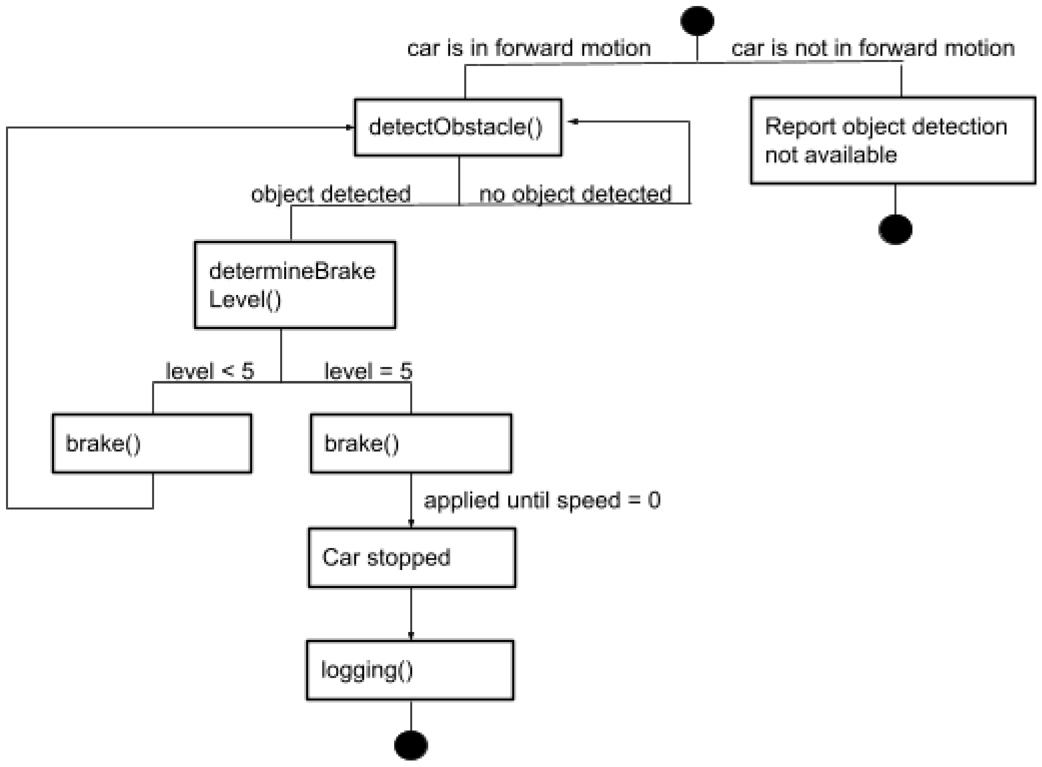
1. The sensors detect an obstacle in front of the car
2. The camera and laser sensors send current speed and distance from object to Planning
3. Planning determines the level of brake to apply and sends request to VCS

4. VCS applies the level of brake
5. Sensors continue to update information sent to Planning
6. Planning continues to update brake level request sent to VCS
7. Vehicle is fully stopped before the object
8. Planning sends all related data to System Management for logging
9. VCS sends all related data to System Management for logging

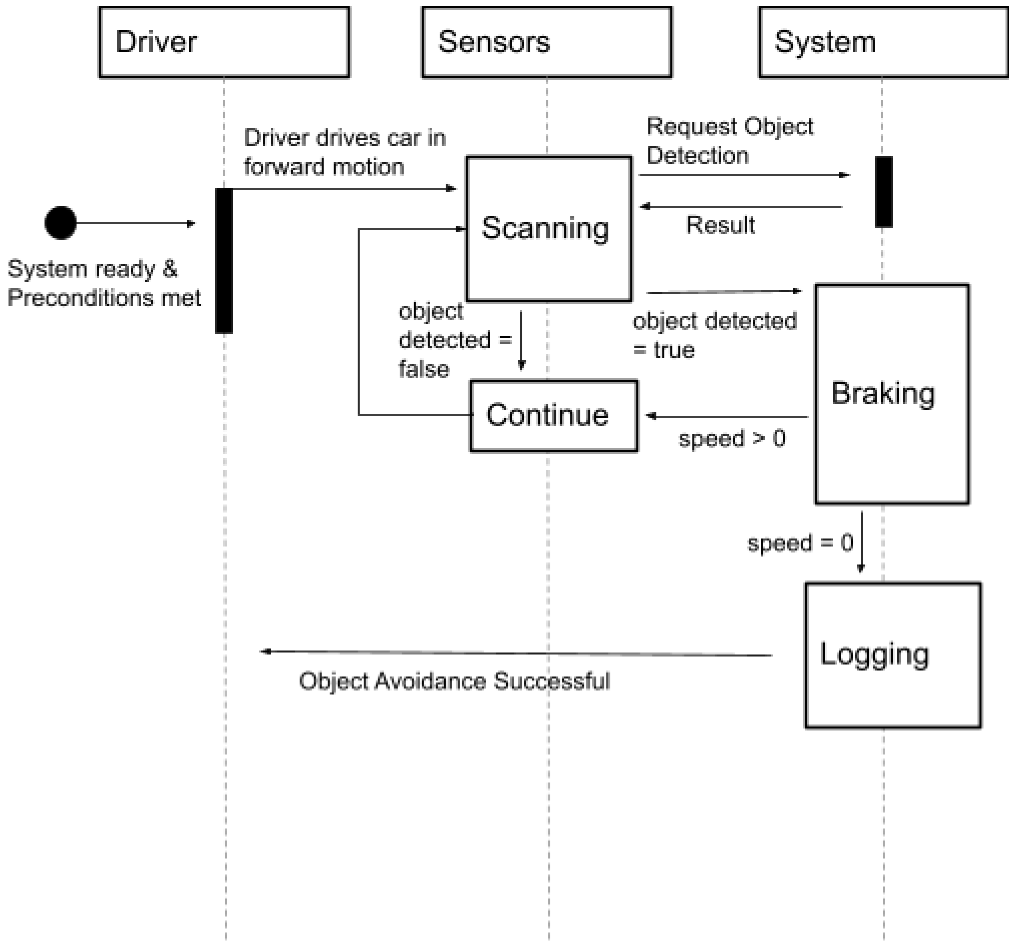
4.3.1.6 Exceptions

- The obstacle is removed from the path of motion of the car
- Car is not in forward motion

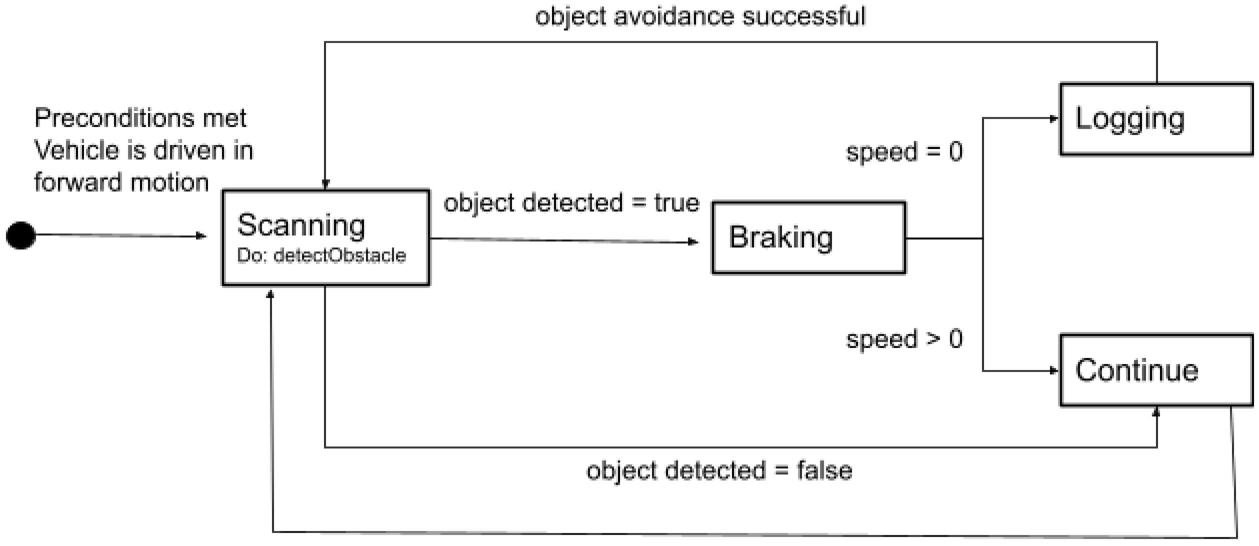
4.3.2 Activity Diagram



4.3.3 Sequence Diagram



4.3.4 State Diagram



4.4 Use Case 4: Headlights turned On

4.4.1 Use Case Scenario

4.4.1.1 Actors

- Sensors
- Driver

4.4.1.2 Trigger

- Sensors detection lighting outside the car to be less than or equal to 1000 lux

4.4.1.3 Preconditions

- The car is on
- The driver is present

4.4.1.4 Postconditions

- The headlights are turned on

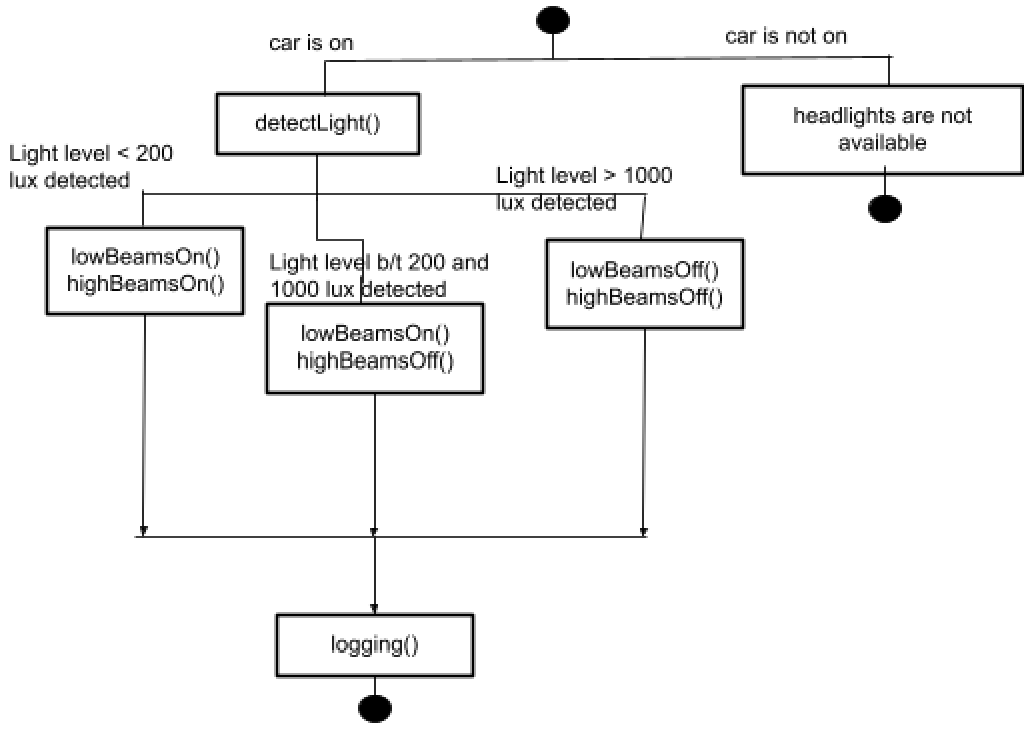
4.4.1.5 Steps of the Activity

1. Sensors determine the light level outside the car is determined to be less than or equal to 1000 lux
2. Sensors send the light level to Sensor Fusion, which sends the cleaned data to Planning
3. If the light is determined to be < 200 lux, a request is sent to VCS to turn high beams on
4. Otherwise, a request is sent to VCS to turn low beams on and to turn high beams off
5. Sensors continue to check the light level outside the car
6. Planning updates headlight request to VCS
7. Planning sends all related data to System Management for logging
8. VCS sends all related data to System Management for logging

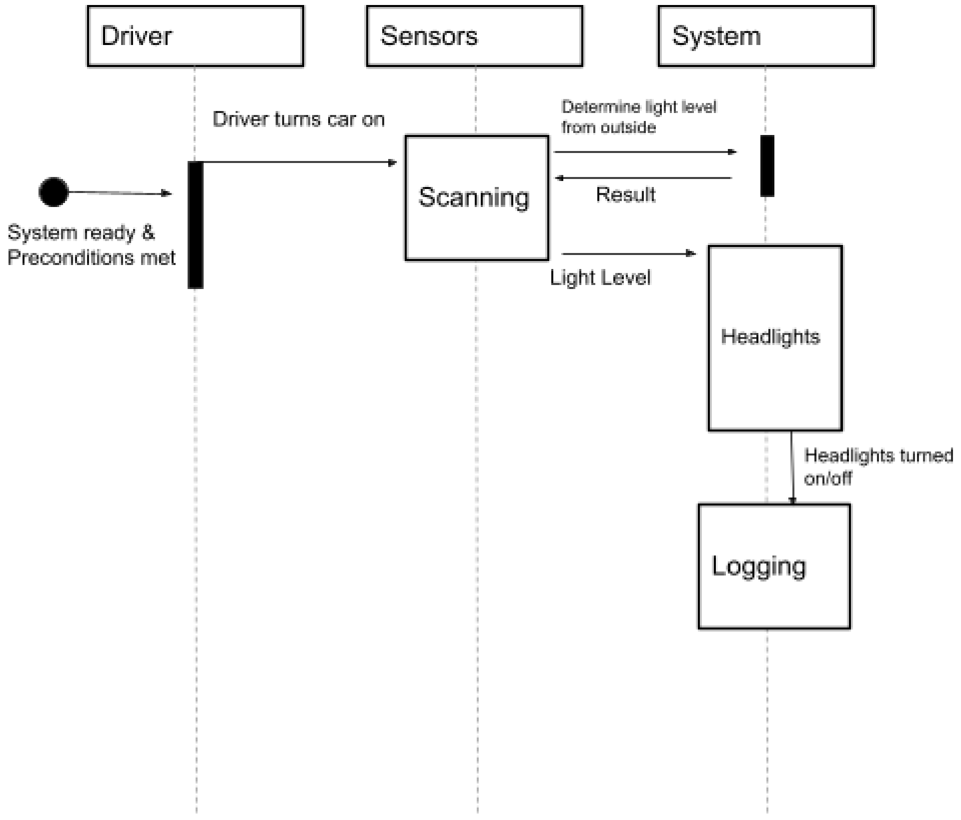
4.4.1.6 Exceptions

- The driver manually adjusts the headlights

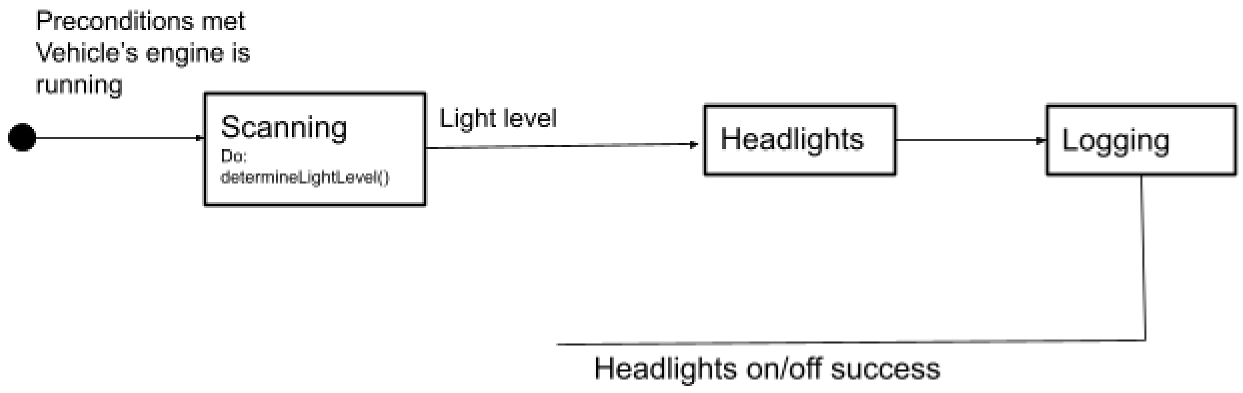
4.4.2 Activity Diagram


car is on
car is not on
detectLight()
headlights are not available
Light level < 200 lux detected
Light level > 1000 lux detected
lowBeamsOn() highBeamsOn()
Light level b/t 200 and 1000 lux detected
lowBeamsOff() highBeamsOff()
lowBeamsOn() highBeamsOff()
logging()

4.4.3 Sequence Diagram



4.4.4 State Diagram


Preconditions met
Vehicle's engine is running
Scanning
Do: determineLightLevel()
Light level
Headlights
Logging
Headlights on/off success

4.5 Use Case 5: User Starts the Vehicle

4.5.1 Use Case Scenario

4.5.1.1 Actors

- Driver

4.5.1.2 Trigger

- The Driver presses the start engine button

4.5.1.3 Preconditions

- The car is off
- The driver is present

4.5.1.4 Postconditions

- The dashboard Display on the car is on and shows the driver the features of the car such as current speed, amount of gasoline in the car, the headlights, the windshield wipers, etc.
- The engine and the software of the car turn on and wait for input.

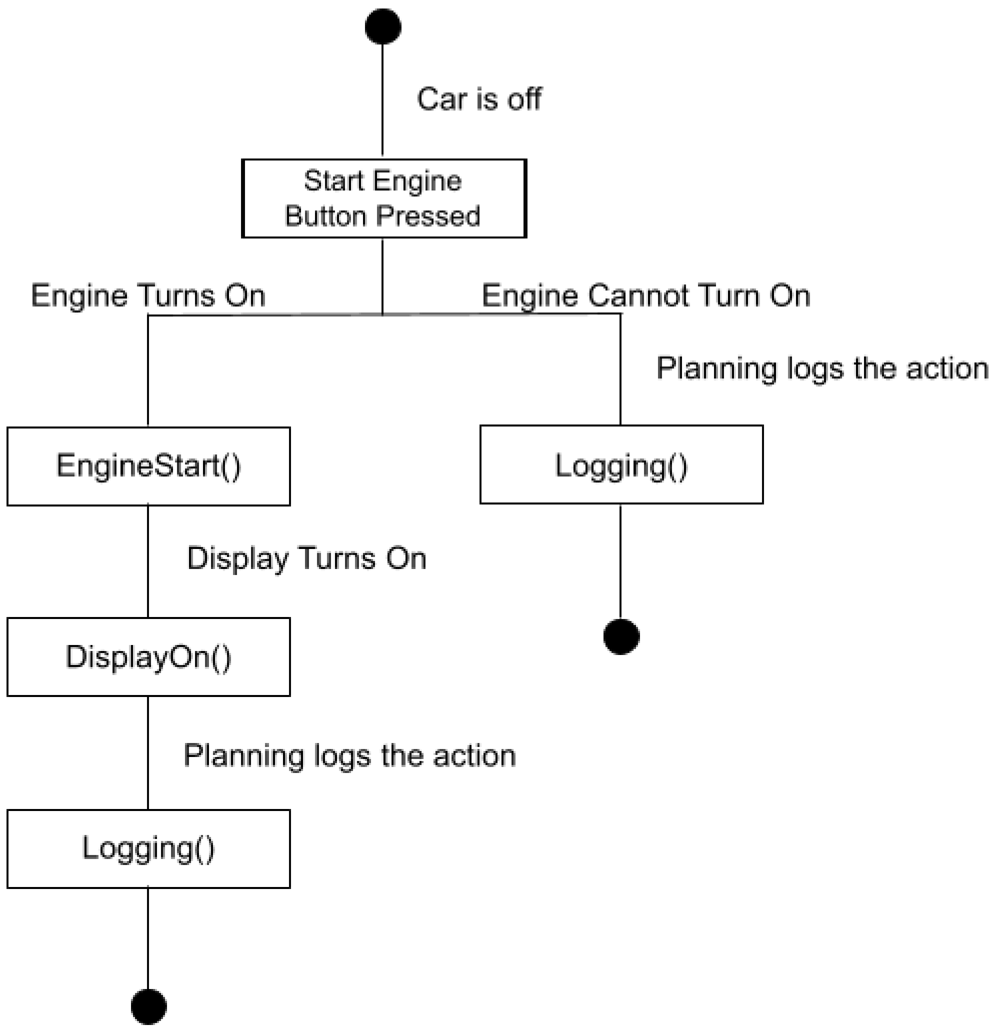
4.5.1.5 Steps of the Activity

1. The driver activates the start engine button
2. Sensors determine if brake is applied by driver
3. Planning sends relevant data to the display such as the amount of gasoline in the car, the current speed of the car, the activity of the headlights, etc.
4. The Display shows these features.
5. Planning sends that the engined successfully or couldn't turn on log

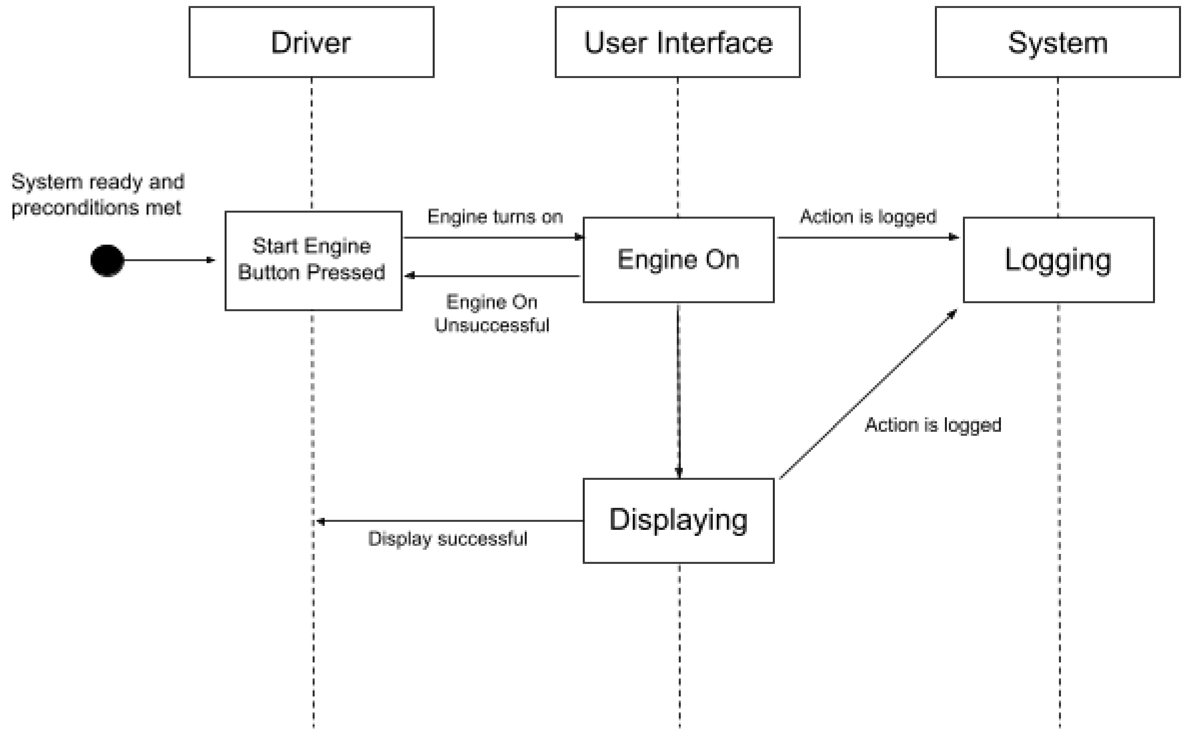
4.5.1.6 Exceptions

- The engine cannot start

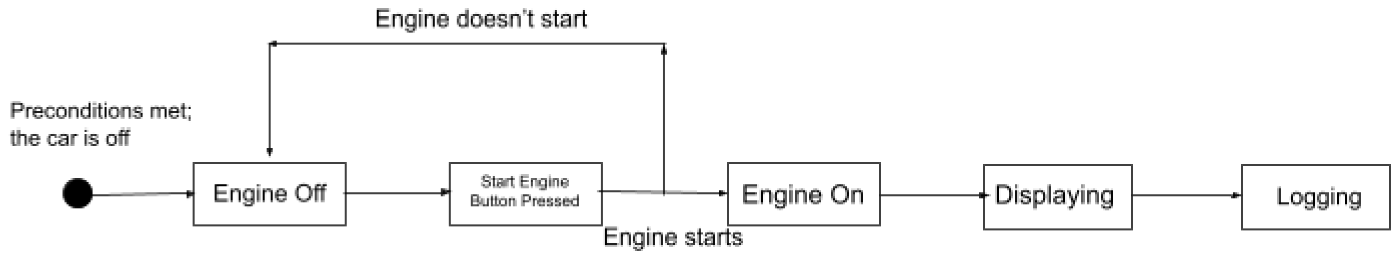
4.5.2 Activity Diagram



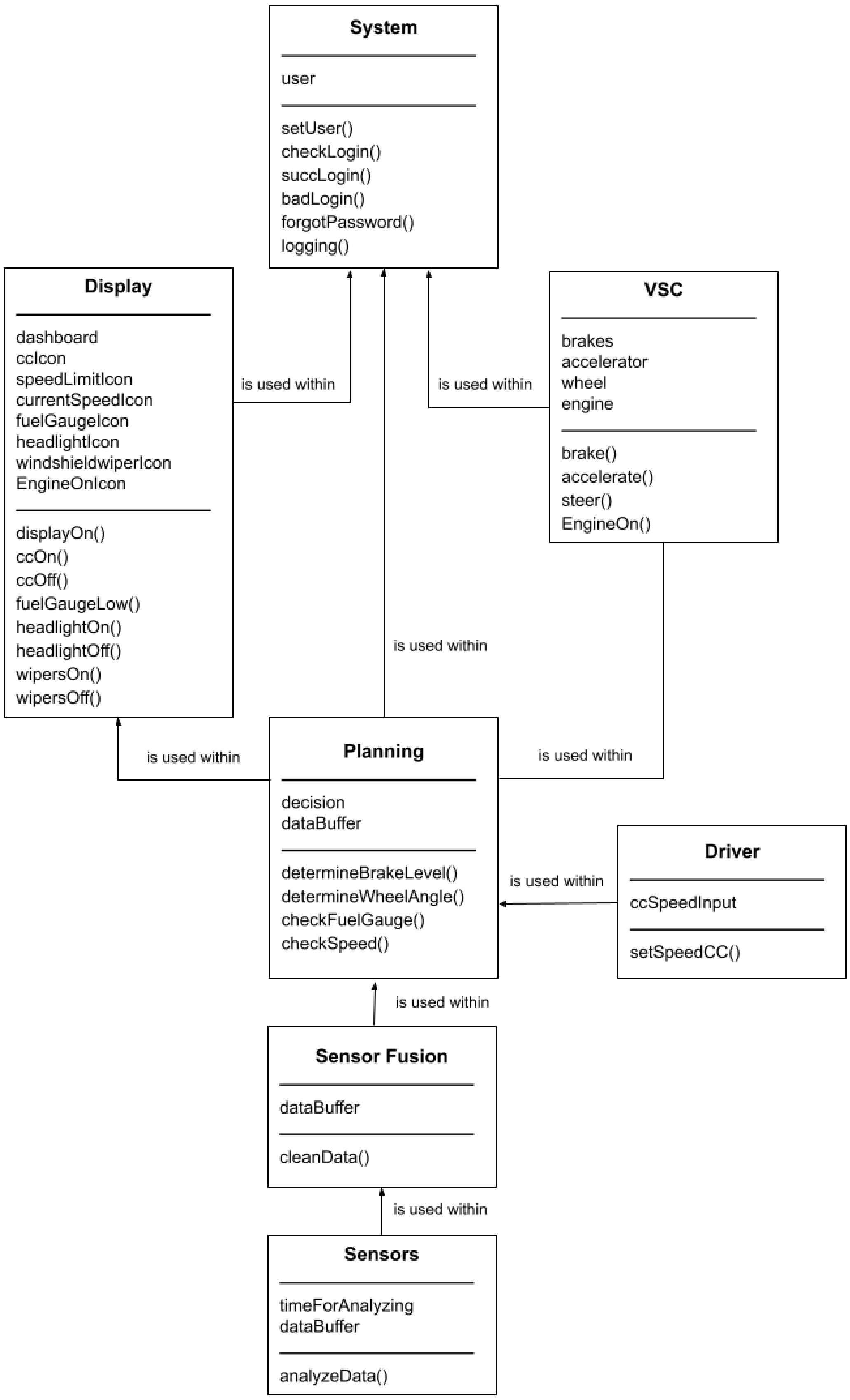
4.5.3 Sequence Diagram



4.5.4 State Diagram



4.6 Classes


System
user
setUser()
checkLogin()
succLogin()
badLogin()
forgotPassword()
logging()
Display
dashboard
ccIcon
speedLimitIcon
currentSpeedIcon
fuelGaugeIcon
headlightIcon
windshieldwiperIcon
EngineOnIcon
displayOn()
ccOn()
ccOff()
fuelGaugeLow()
headlightOn()
headlightOff()
wipersOn()
wipersOff()
VSC
brakes
accelerator
wheel
engine
brake()
accelerate()
steer()
EngineOn()
Planning
decision
dataBuffer
determineBrakeLevel()
determineWheelAngle()
checkFuelGauge()
checkSpeed()
Driver
ccSpeedInput
setSpeedCC()
Sensor Fusion
dataBuffer
cleanData()
Sensors
timeForAnalyzing
dataBuffer
analyzeData()
is used within

5. Design

5.1 Software Architecture

There are seven major types of software architectural styles that all deal with data flow. They all depict the motion of data through a system and different actions performed on the data and the system. They differ in the layout of the components and the movement of data.

5.1.1 Examining Every Software Architecture Model

5.1.1.1 Data-Centered

The first architecture discussed is Data-Centered Architectures are linked by the component that they all have a data store in the center. This could be a database or a file that is then connected to every other aspect of the architecture. These other components are known as client software, and they are able to modify the data in the store.

Pros:

- It is extremely easy to integrate changes without affecting other clients since the client components are independent of each other.
- The use of centralized data makes it easy to store large amounts of data, and our project will need to store a lot of data since every action needs to be logged.

Cons:

- Any changes to the data would be difficult, expensive, and affect all of the clients. This would be difficult for our project because updates are likely in the future.
- This style of architecture is more vulnerable to failure. Since we are programming a car, failure could be detrimental.
- There is a centered data store, but our data moves through several clients before going to a store.

In this project, implementing a data-centered architecture is not a good fit since our data has to move through different modules. We do not have a database or file with all the information on it, instead we have sensors that move data through various modules.

5.1.1.2 Data-Flow

A second type of software architecture is Data-Flow architectures. Similar to data-centered, this type of architecture is all about moving data. It takes data in as an input and has several filters that each independently change the data to a specified form.

Pros:

- Each filter doesn't need to rely on others.
- Able to divide up the system into smaller steps which are easier to implement.

Cons:

- There is not a lot of data moving through the system at one time, so it takes a while for the data to go through the entire system.
- There is a high latency because each filter must be finished before moving to the next filter.

This architecture is not an ideal fit for our project because the data flows through the pipes and filters in one direction, however our system does not have this hierarchical path. Our data moves in many directions, and on multiple occasions, loops through specific components until a parameter is met.

5.1.1.3 Call-and-Return

Call-and-Return Architectures have a main program that calls other components that then call others. It takes on a tree type of diagram structure since there is root program that calls all the others.

Pros:

- Easy to modify and scale.
- Enable code modularity.
- The performance of the software is efficient since it divides up the tasks to different components.

Cons:

- Hard to change one program without having to alter the ones that rely on it.
- Data coupling is prone to occur if the components are not well defined.

In terms of our project, this architecture would be a relatively poor fit. In our system, we do not have a main program or module that controls the other. The Planning module is the brain of the system as it does give information and instructions to VCS, but Planning gets information from The Sensor Fusion. There is not a hierarchy to our system, so this architecture type is not an ideal fit.

5.1.1.4 Object-Oriented

Another type of major architecture is Object-Oriented Architectures. This type of architecture clearly displays the actor and the components that are involved in the system.

Pros:

- The components are organized into reusable and self-sufficient objects, which makes it easy to maintain.
- Able to manage the errors during execution.
- Able to provide reusability of components through polymorphism and abstraction

Cons:

- In a complex system, it can be difficult to determine all the necessary classes and objects.
- There is the possibility that some components will be forgotten.

For our system, this architecture is a good fit because it is easy to depict what actions occur and in what order. Many of our use cases rely on overlapping and looping components that are constantly in use to make the system work successfully.

5.1.1.5 Layered

Layered architectures have several layers that all have their own components. The outer layer is the user interface and the innermost layer is the core layer that is closer to machine instructions. Each layer has specific components that do have a role in the process.

Pros:

- Since every layer is independent of others it is easy to change one layer without affecting the whole system.
- Incremental development is easy with this style. This type of development is easy to implement and makes the design process easier.

Cons:

- It is easy to add too many layers that slow the process down.
- The layered design is hard to implement in projects that have several components that call each other. Our project does not have a hierarchical design.

This type of architecture is not a good fit for our project because we do not have this type of hierarchy with our data. We will not have a module that correlates with the core layer since our data flows in different directions.

5.1.1.6 Model View Controller

Model View Controller architecture involves a client, a server, and external data. The client sends requests or data to the server. Inside the server is where data is moved and the request is fulfilled through the controller, model, and view modules.

Pros:

- It is easy to modify since the modules do not depend on each other.
- The user is implemented in this model, which makes it easier to visualize the process while coding.
- There are three predefined components with this style, which makes the coding process easier since this part is already defined for us.

Cons:

- This architecture is more complex than the others, which makes it hard to understand. We have limited experience in software coding, so choosing one of the harder ones to implement would not be wise.
- Changes and updates are costly.

It would not be a good fit for our project because we have more modules that the data flows through. Our data moves around more than the data is capable of in this architecture. This architecture offers three components that the data moves through.

5.1.1.7 Finite State Machine

The last major type of architecture is Finite State Machine architecture. This architecture shows what states the system performs during an operation and the transitions in between each state.

Pros:

- The state of the system can be easily determined.
- It is a flexible architecture.
- It is detailed, so that there can be an easy transition from an abstract diagram to implementation with code.

Cons:

- It focuses mainly on the states of the system and does not fully consider other aspects in the system, like the actor.
- Can be complicated quickly when there are multiple parts on a large scale system.

This architecture is a good fit for displaying use cases where we want to be able to quickly determine the state of the system. For example, the technician login because this architecture clearly shows whether the state is locked or unlocked.

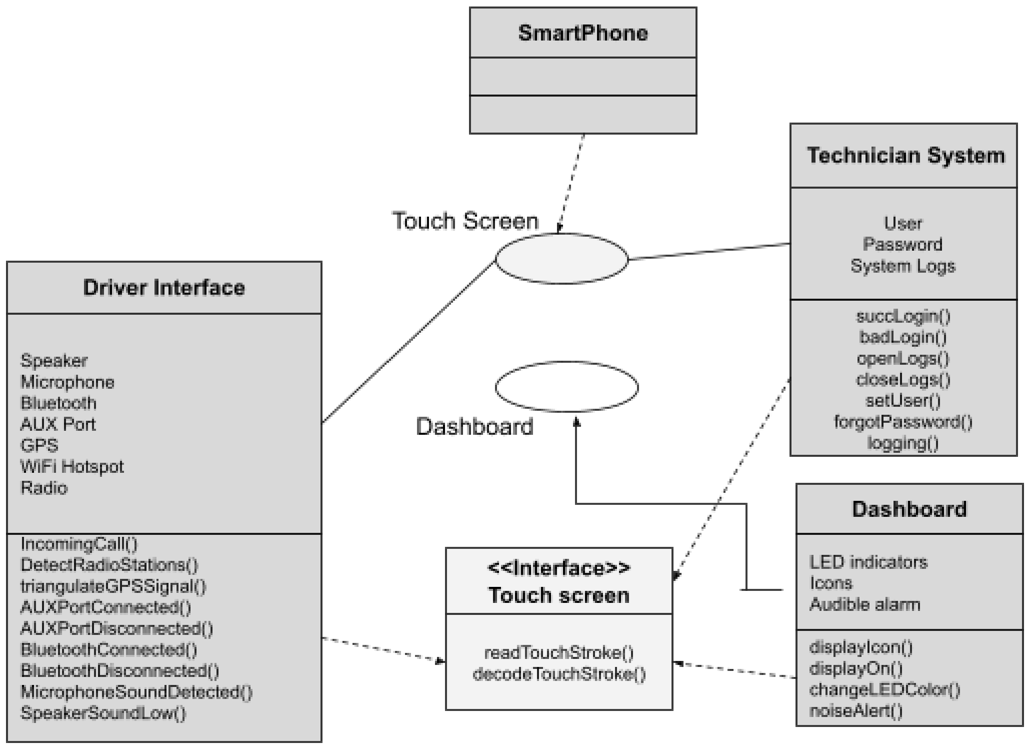
5.1.2 Our Project's Architecture

Object-Oriented Architecture is overall the best fit for our project because with each function of our car, we will be able to follow a process that reflects this type of architecture. As an example, to write the headlights-on function and correlate it to this architecture, the actor would be the sensors. The sensors send data through Sensor Fusion and Planning, and Planning decides if headlights need to be put on and how high they should be. If LowBeamsOn() or HighBeamsOn() is true, then VCS will turn the low or high beams on. This information is then sent to the system to log the action.

For some specific cases, Finite State Machine Architecture is the optimal fit for our project. For example, technician login with the user interface involves being in a locked and unlocked state. Within the Finite State Machine Architecture these states are clearly depicted and allows the viewer to see where the data is flowing and what the current state of the system is. Initially it is depicted as locked, and then after the correct password is verified, the state changes to unlocked.

5.2 Interface Design

This diagram represents all the available interfaces that an IOT vehicle uses to interact with both the Driver and the Technician. The software mainly interacts with the Driver through the Dashboard, and the Driver mainly interacts with the software through the Touch Screen. The Technician also interacts with the software through the Touch Screen in order to access log files.



5.3 Component Level Design

5.3.1 Planning Component

- determineBrakeLevel()
  - Based on the vehicle speed and distance from the object, determine level of brake from 1-5 to apply
- determineWheelAngle()
  - Based on the angle at which steering wheel is moved, determine the angle that the vehicle tires should be rotated
- checkFuelGuage()
  - Determines the amount of gas within the vehicle's tank
- checkSpeed()
  - Determines the vehicle's speed

5.3.2 Sensor Fusion Component

- cleanData()
  - Used to merge, clean, and organize the data to be used as inputs for Planning decision making

5.3.3 Sensor Component

- analyzeData()
  - Collects and analyzes the input data from the cameras, laser, etc. of the vehicle

5.3.4 System Management Component

- setUser()
  - Allows the user to create username and password on the digital interface
- checkLogin()
  - Compares the inputted password with the ones in the system
- succLogin()
  - Set to true if login is successful, false otherwise
- badLogin()
  - Set to true if login is unsuccessful, false otherwise
- forgotPassword()
  - Allows user to recreate password if selected on the digital interface that password is forgotten
- logging()
  - Updates the system with the data from planning and VSC

5.3.5 Vehicle Control Component

- brake()
  - Applies the brake of the vehicle
- accelerate()
  - Applies the accelerator of the vehicle
- steer()
  - Applies the steering wheel to rotate the vehicle wheels
- EngineOn()
  - Turns the vehicle on when start button is pressed and vehicle is off
- EngineOff()
  - Turns the vehicle off when start button is pressed and vehicle is on

5.3.6 Display Component

- displayOn()
  - Turns the digital interface of the vehicle on when the car starts
- ccOn()
  - Lights up the cruise control icon on dashboard
- ccOff()
  - Turns off the cruise control icon on the dashboard
- fuelGuageLow()

    - Lights up the low fuel icon on the dashboard of the vehicle to warn drivers
- headlightOn()
    - Lights up the headlights icon to indicate headlights are on
- headlightsOff()
    - Turns off the headlights icon on the dashboard
- wipersOn()
    - Lights up wiper icon on dashboard if wipers are being used
- wipersOff()
    - Turns off the wiper icon on the dashboard

6. Project Code

Below is the code for the requirements listed in section three in the language Python.

```
#must have libraries installed on computer
#pip install tabulate

from tabulate import tabulate
import functools
import datetime

#variables for car
caron = False
update = 0
speed = 0
turnLeft = 0
turnRight = 0

leftBlinker = False
rightBlinker = False

blindSpotProtection = False

#Position/Magnitude of steering wheel (neg =left, pos= right)
wheelPosition=0

#variables of wipers
wipersintensity = 0
```

```
#variables for headlights
lowbeams = False
highbeams = False

#variables for cruise control
ccOn = False

#variables for logging in
ownerusername = "hugthelanesowner"
techusername = "hugthelanestechnician"
ownerpassword = 1234
techpassword = 5678
loginsuccess=1

#variables for logging information
log = []

#outside variables
isSign = 0
sign = 0
isTrafficLight = 0
colorTL = "red"
obstacle = 0
isLines = 0
lineDistLeft = 0
lineDistRight = 0
parkingAssist = False
blindSpotProtection = False

#velocity of wheels
v1 = 0
v2 = 0
v3 = 0
v4 = 0

##########################################################
#code to reset variables when testing
def reset():
    global caron, update, speed, turnLeft, turnRight, leftBlinker, leftBlinker, wheelPosition,
wipersintensity, lowbeams, highbeams, ccOn, ownerusername, techusername, ownerpassword,
techpassword, loginsuccess, log, isSign, sign, isTrafficLight, colorTL, obstacle, isLines,
lineDistLeft, lineDistRight, parkingAssist, blindSpotProtection, v1, v2, v3, v4

    #variables for car
    caron = False
    update = 0
    speed = 0
    turnLeft = 0
    turnRight = 0

    leftBlinker = False
    leftBlinker = False

    #Position/Magnitude of steering wheel (neg =left, pos= right)
    wheelPosition=0

    #variables of wipers
    wipersintensity = 0

    #variables for headlights
    lowbeams = False
    highbeams = False

    #variables for cruise control
    ccOn = False

    #variables for logging in
    ownerusername = "hugthelanesowner"
    techusername = "hugthelanetechnician"
    ownerpassword = 1234
    techpassword = 5678
    loginsuccess=1

    #variables for logging information
    log = []

    #outside variables
    isSign = 0
    sign = 0
    isTrafficLight = 0
```

```
    colorTL = "red"
    obstacle = 0
    isLines = 0
    lineDistLeft = 0
    lineDistRight = 0
    parkingAssist = False
    blindSpotProtection = False

    #velocity of wheels
    v1 = 0
    v2 = 0
    v3 = 0
    v4 = 0

#########################################################
#system logging code
def logging(command):
    @functools.wraps(command)
    def inside(*args, **kwargs):
        now = datetime.datetime.now().strftime('%Y-%m-%d %H:%M:%S')
        with open('logfile.txt', 'a') as log:
            log.write(f'{now}: {command.__name__}\n')
        return command(*args, **kwargs)
    return inside

def clearlog():
    with open('logfile.txt', 'w') as f:
        f.write('')

#########################################################
#starting car code
@logging
def enginestart():
    global caron
    caron = True
    displayon()
    return 0

#########################################################
#display interface code
```

```
def maketable():
   global caron, speed, ccOn, highbeams, lowbeams, wipersintensity, leftBlinker,
rightBlinker,parkingAssist, wheelPosition, blindSpotProtection
   return [
      ["Engine On", caron],
      ["Current Speed", speed],
      ["Cruise Control On", ccOn],
      ["Highbeams On", highbeams],
      ["Lowbeams On", lowbeams],
      ["Windshield Wiper Level", wipersintensity],
      ["Left Blinker On", leftBlinker],
      ["Right Blinker On", rightBlinker],
      ["Parking Assistance On", parkingAssist],
      ["Wheel Position", wheelPosition],
      ["Blind Spot Detection", blindSpotProtection]
   ]

@logging
def displayon():
   table = maketable()
   print(tabulate(table, headers=['Parameter','Value'], tablefmt='fancy_grid'))
   return 0

#############################################################
#headlights activation code
@logging
def lowbeamson():
   global lowbeams
   lowbeams = True
   return 0

@logging
def highbeamson():
   global highbeams
   highbeams = True
   return 0

@logging
def lowbeamsoff():
   global lowbeams
```

```
    lowbeams = False
    return 0

@logging
def highbeamsoff():
    global highbeams
    highbeams = False
    return 0

@logging
def detectlight(light):
    if light < 1000:        #if inadequete light
        if light < 200:     #if very little light -> only highbeams
            highbeamson()
            lowbeamsoff()
        else:
            lowbeamson()    #if some light -> only lowbeams
            highbeamsoff()
    else:                   #adequete light so no headlights
        highbeamsoff()
        lowbeamsoff()
    return 0

########################################################
#cruise control code
@logging
def applybrake(intensity):
    global speed
    if intensity == 5:
        speed = 0           #full stop
    elif intensity == 4:
        speed = speed * 0.8 #decrease speed by 80%
    elif intensity == 3:
        speed = speed * 0.6 #decrease speed by 60%
    elif intensity == 2:
        speed = speed * 0.4 #decrease speed by 40%
    elif intensity == 1:
        speed = speed * 0.2 #decrease speed by 20%
    return 0
```

```
@logging
def applyaccel(intensity):
    global speed
    if intensity == 5:
        speed = speed * 0.5 #increase speed by 50%
    elif intensity == 4:
        speed = speed * 0.4 #increase speed by 40%
    elif intensity == 3:
        speed = speed * 0.3 #increase speed by 30%
    elif intensity == 2:
        speed = speed * 0.2 #increase speed by 20%
    elif intensity == 1:
        speed = speed * 0.1 #increase speed by 10%
    return 0

@logging
def cruisecontrol():
    global ccOn, speed
    if(ccOn == True):
        desired_speed = input("Enter Desired Speed: ")
        speed = int(desired_speed)
    return 0

@logging
def changeCruiseControl():
    global ccOn
    if(ccOn == False):
        ccOn = True
    else:
        ccOn = False

##########################################################
#speed limit sign detection code
@logging
def speedlimitsign():
    global speed
    global sign
    global isSign
    if (isSign == 1):
        if (speed > sign + 10):
            speed = sign
        elif (speed < sign + 10):
            speed = sign
    return 0

#########################################################
#windshield wipers activation code
@logging
def windshieldwiperson(intensity):
    global wipersintensity
    wipersintensity = intensity
    return 0

@logging
def windshieldwipersoff():
    global wipersintensity
    wipersintensity = 0
    return 0

@logging
def detectwater(rainpermin):
    if rainpermin > 399:                      #more than 400
        windshieldwiperson(4)
    elif rainpermin < 400 and rainpermin > 299: #more than 300
        windshieldwiperson(3)
    elif rainpermin < 300 and rainpermin > 199: #more than 200
        windshieldwiperson(2)
    elif rainpermin < 200 and rainpermin > 99:  #more than 100
        windshieldwiperson(1)
    else:
        windshieldwipersoff()
    return 0

#########################################################
#detect object and slow vehicle code
@logging
def detectobject(distance):
    global isSign
    global isTrafficLight
    global obstacle
    if isSign:
        pass
    elif isTrafficLight:
        pass
    else:
        obstacle = 1
        if distance >= 150 and distance < 200: #less than 200
            applybrake(1)
        elif distance >= 100 and distance < 150: #less than 150
            applybrake(2)
        elif distance >= 50 and distance < 100: #less than 100
            applybrake(3)
        elif distance >= 10 and distance < 50: #less than 50
            applybrake(4)
        elif distance < 10: #less than 10
            applybrake(5)
    return 0

########################################################
#parking assist code
@logging
def parking(frontDist, backDist, leftDist, rightDist):
    global speed
    if(frontDist <= 2 ):
        speed = 0
    if(backDist <= 2):
        speed = 0
    if(leftDist <= 2):
        speed = 0
    if(rightDist <= 2):
        speed = 0
    return 0

@logging
def parkingSwitch(fDist, bDist, lDist, rDist):
    global parkingAssist
    if(parkingAssist == True):
        parking(fDist, bDist, lDist, rDist)

@logging
```

```
def changeParkingSetting():
    global parkingAssist
    if(parkingAssist == False):
        parkingAssist = True
    else:
        parkingAssist = False

##########################################################
#blind spot detection code
@logging
def stopTurning():
    global wheelPosition
    global leftBlinker
    global rightBlinker
    if(wheelPosition < 0):
        wheelPosition = 0
        leftBlinker = False
    elif(wheelPosition > 0):
        wheelPosition = 0
        rightBlinker = False

@logging
def blindspotwarning(sideSensor):
    if(sideSensor <= 10):
        stopTurning()

@logging
def turning(leftSensor, rightSensor):
    global wheelPosition
    global leftBlinker
    global rightBlinker
    global blindSpotProtection
    if(blindSpotProtection == True):
        if(wheelPosition != 0):
            blinkerSignal = False
            if(wheelPosition > 0 and rightBlinker == True):
                blinkerSignal = True
            elif(wheelPosition > 0 and leftBlinker == True):
                leftBlinker = False
            elif(wheelPosition < 0 and leftBlinker == True):
                blinkerSignal = True
            elif(wheelPosition < 0 and rightBlinker == True):
                rightBlinker = False
            if(blinkerSignal == True):
                if(wheelPosition < 0):
                    blindspotwarning(leftSensor)
                else:
                    blindspotwarning(rightSensor)

@logging
def changeBlindSpotSetting():
    global blindSpotProtection
    if(blindSpotProtection == False):
        blindSpotProtection = True
    else:
        blindSpotProtection = False

############################################################
#traffic light code
@logging
def trafficlight():
    if (trafficLight == 1):
        if (colorTL == "red"):       #if red, brake to a stop
            applybrake(5)
        elif (colorTL == "yellow"):
            if (speed < 40):
                applybrake(5)
            else:
                applybrake(0)
        elif (colorTL == "green"):  #if green light, keep going
                pass
        else:
            pass
    return 0


############################################################
#lane departure code
@logging
def steerLeft(length):
    global turnLeft
    global lineDistLeft
    global lineDistRight
    turnleft = length
    var = (lineDistLeft + lineDistRight) / 2
    lineDistLeft = var
    lineDistRight = var
    return 0

@logging
def steerRight(length):
    global turnRight
    global lineDistLeft
    global lineDistRight
    turnRight = length
    var = (lineDistLeft + lineDistRight) / 2
    lineDistLeft = var
    lineDistRight = var
    return 0

@logging
def lanedeparture():
    if (isLines == 1):
        if (abs(lineDistLeft - lineDistRight) < 1):     #if centered within 1 foot stay the same
            pass
        else:
            if ((lineDistLeft - lineDistRight > 1)):    #if closer to the right move left
                steerLeft(lineDistLeft - lineDistRight)

            elif ((lineDistRight - lineDistLeft >1)):   #if closer to the left move right
                steerRight(lineDistRight - lineDistLeft)
    return 0

############################################################
#traction code
@logging
def traction():
    global v1
    global v2
    global v3
    global v4
    #if velocity is greater than 10 of the others avg
    if v1 - 10 > (v2 + v3 + v4)/3:
        v1 = (v2 + v3 + v4)/3 #set to avg
    elif v2 - 10 > (v1 + v3 + v4)/3:
        v2 = (v1 + v3 + v4)/3 #set to avg
    elif v3 - 10 > (v1 + v2 + v4)/3:
        v3 = (v1 + v2 + v4)/3 #set to avg
    elif v4 - 10 > (v1 + v2 + v3)/3:
        v4 = (v1 + v2 + v3)/3 #set to avg
    return 0

#########################################################
#user login code
@logging
def forgotpassword(user, passw):
    if user == ownerusername:
        ownerpassword = passw
        succlogin()
    elif user == techusername:
        techpassword = passw
        succlogin()

    return 0

@logging
def badlogin():
    global loginsuccess
    loginsuccess = 0
    return 0

@logging
def succlogin():
    global loginsucces
    loginsuccess = 1
    return 0

@logging
def checklogin(user,passw, tries):
    if user == ownerusername:
        if passw == ownerpassword:
            succlogin()
            return 0
        else:
            badlogin()

    elif user == techusername:
        if passw == techpassword:
            succlogin()
            return 0
        else:
            badlogin()
    else:
        badlogin()
        logging("bad login attempt")

############################################################
#software update code
@logging
def softwareupdate(user, passw, attemptnum):
    global update
    enginestart()                   #turns the car on so display is visible
    if checklogin(user, passw, 0) == 0:    #check for valid user/password
        update = 1                  #how do we update the software lol
        logging("software updated")
    else:
        logging("software update attempt failed")
    return 0

############################################################
#end of code segment
```

7. Testing

In this section we tested the code made in the previous section. Testing was split into two categories: Scenario Based Testing and Validation Testing. Scenario Based Testing tests code created to represent the use-cases in Section 4. Validation Testing looks at the code created for the functional requirements in Section 3.

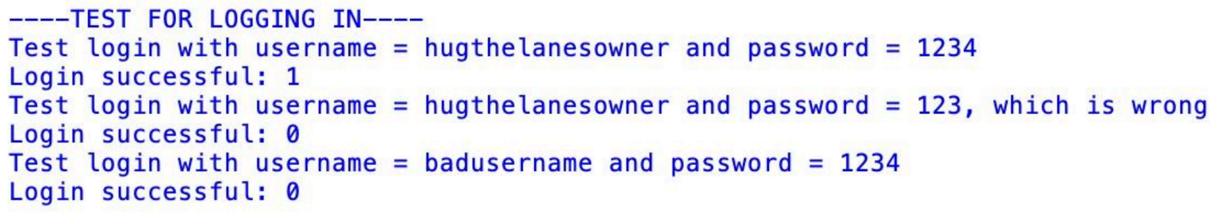
7.1 Scenario-Based Testing

7.1.1 Use-Case 1: User/Technician Log-In

7.1.1.1 Code

```
print("\n")
print("----TEST FOR LOGGING IN----")
print("Test login with username = hugthelanesowner and password = 1234")
checklogin("hugthelanesowner", 1234, 1)
print("Login successful: " + str(loginsuccess))
print("Test login with username = hugthelanesowner and password = 123, which is
wrong")
checklogin("hugthelanesowner",123, 1)
print("Login successful: " + str(loginsuccess))
print("Test login with username = badusername and password = 1234")
checklogin("badusername",1234, 1)
print("Login successful: " + str(loginsuccess))
```

7.1.1.2 Output



```
----TEST FOR LOGGING IN----
Test login with username = hugthelanesowner and password = 1234
Login successful: 1
Test login with username = hugthelanesowner and password = 123, which is wrong
Login successful: 0
Test login with username = badusername and password = 1234
Login successful: 0
```

In the test, a correct username and password are set to: hugthelanesowner and 1234, respectively. Our test successfully determines whether or not the login attempt is correct or not. The "Login successful:" value of 1 represents a correct login, while a value of 0 represents an unsuccessful attempt.

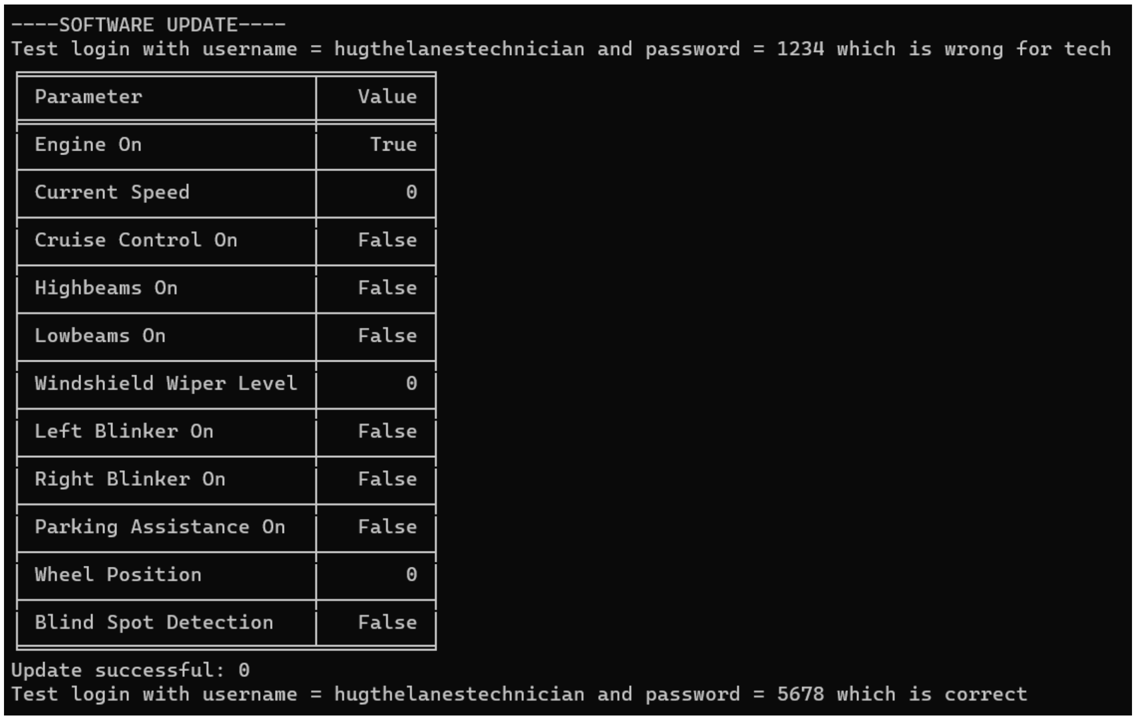
7.1.2 Use-Case 2: Software Update

7.1.2.1 Code

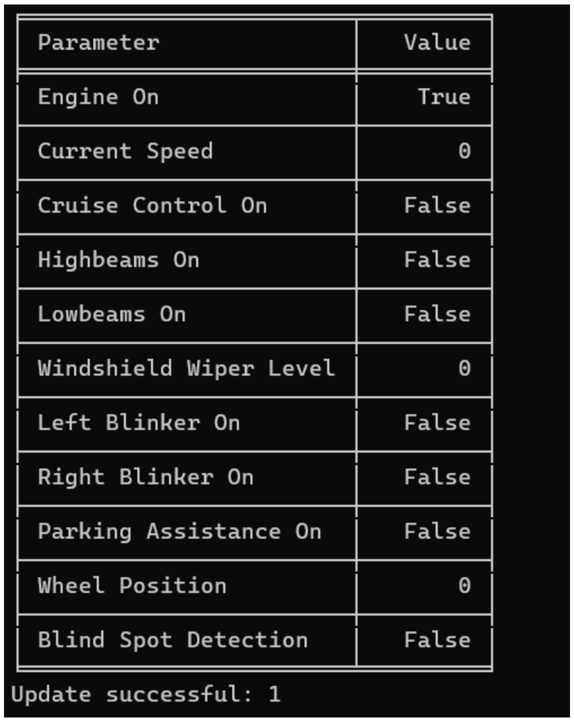
```
print("\n")
print("----TEST FOR SOFTWARE UPDATE----")
print("Test login with username = hugthelanestechnician and password = 1234 which is
wrong for tech")
softwareupdate("hugthelanestechnician",1234, 1)
print("Update successful: " + str(update))
print("Test login with username = hugthelanestechnician and password = 5678 which is
correct")
```

softwareupdate("hugthelanestechnician",5678, 1)
print("Update successful: " + str(update))

7.1.2.2 Output



```
----SOFTWARE UPDATE----
Test login with username = hugthelanestechnician and password = 1234 which is wrong for tech
╒════════════════════════╤═════════╕
│ Parameter              │   Value │
╞════════════════════════╪═════════╡
│ Engine On              │    True │
├────────────────────────┼─────────┤
│ Current Speed          │       0 │
├────────────────────────┼─────────┤
│ Cruise Control On      │   False │
├────────────────────────┼─────────┤
│ Highbeams On           │   False │
├────────────────────────┼─────────┤
│ Lowbeams On            │   False │
├────────────────────────┼─────────┤
│ Windshield Wiper Level │       0 │
├────────────────────────┼─────────┤
│ Left Blinker On        │   False │
├────────────────────────┼─────────┤
│ Right Blinker On       │   False │
├────────────────────────┼─────────┤
│ Parking Assistance On  │   False │
├────────────────────────┼─────────┤
│ Wheel Position         │       0 │
├────────────────────────┼─────────┤
│ Blind Spot Detection   │   False │
╘════════════════════════╧═════════╛
Update successful: 0
Test login with username = hugthelanestechnician and password = 5678 which is correct
```



```
╒════════════════════════╤═════════╕
│ Parameter              │   Value │
╞════════════════════════╪═════════╡
│ Engine On              │    True │
├────────────────────────┼─────────┤
│ Current Speed          │       0 │
├────────────────────────┼─────────┤
│ Cruise Control On      │   False │
├────────────────────────┼─────────┤
│ Highbeams On           │   False │
├────────────────────────┼─────────┤
│ Lowbeams On            │   False │
├────────────────────────┼─────────┤
│ Windshield Wiper Level │       0 │
├────────────────────────┼─────────┤
│ Left Blinker On        │   False │
├────────────────────────┼─────────┤
│ Right Blinker On       │   False │
├────────────────────────┼─────────┤
│ Parking Assistance On  │   False │
├────────────────────────┼─────────┤
│ Wheel Position         │       0 │
├────────────────────────┼─────────┤
│ Blind Spot Detection   │   False │
╘════════════════════════╧═════════╛
Update successful: 1
```

In this case we can see that when software update() is called and the user tries to login with the technician's username but with the wrong password. Therefore, the update failed and the return-value was 0. When the user uses the correct username and password we can see that the return value for the softwareupdate() was 1, to represent true.

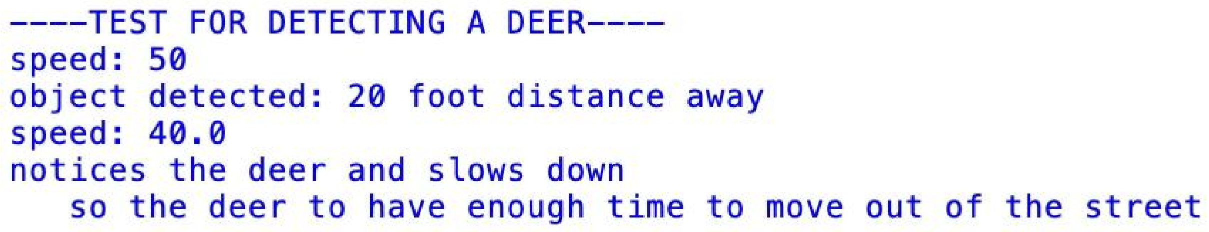
7.1.3 Use-Case 3: Object Detection

7.1.3.1 Code

```
##Test the detect object
print("\n")
print("----TEST FOR DETECTING A DEER----")
##when deer in front of car
speed = 50 #start speed
deer = 20 #distance
yieldsign = 50 #distance

print("speed: " + str(speed))
print("object detected: " + str(deer) +" foot distance away")
detectobject(deer)
print("speed: " + str(speed))
print("notices the deer and slows down \n   so the deer to have enough time to move out
of the street")
reset()
```

7.1.3.2 Output



```
----TEST FOR DETECTING A DEER----
speed: 50
object detected: 20 foot distance away
speed: 40.0
notices the deer and slows down
   so the deer to have enough time to move out of the street
```

This test creates the scenario that a deer has stopped in front of the car. The car is initially set to a speed of 50 mph, while the deer is 20 feet away. The car successfully slows down to 40 mph when the detectobject() command is run. This decrease in speed offers enough time for the deer to move out of the path of the car.

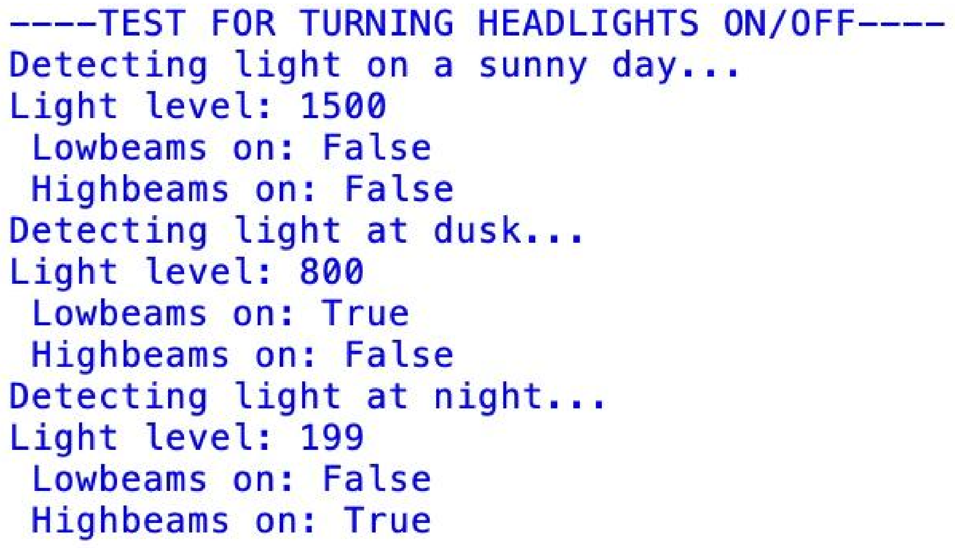
7.1.4 Use-Case 4: Headlights Turn On

7.1.4.1 Code

```
print("\n")
print("----TEST FOR TURNING HEADLIGHTS ON/OFF----")
#light levels for time of day
```

```
night = 199
dusk = 800
day = 1500
detectlight(day)
print("Detecting light on a sunny day...\nLight level: " + str(day))
print(" Lowbeams on: " + str(lowbeams))
print(" Highbeams on: " + str(highbeams))
detectlight(dusk)
print("Detecting light at dusk...\nLight level: " + str(dusk))
print(" Lowbeams on: " + str(lowbeams))
print(" Highbeams on: " + str(highbeams))
detectlight(night)
print("Detecting light at night...\nLight level: " + str(night))
print(" Lowbeams on: " + str(lowbeams))
print(" Highbeams on: " + str(highbeams))
reset()
```

7.1.4.2 Output



```
----TEST FOR TURNING HEADLIGHTS ON/OFF----
Detecting light on a sunny day...
Light level: 1500
 Lowbeams on: False
 Highbeams on: False
Detecting light at dusk...
Light level: 800
 Lowbeams on: True
 Highbeams on: False
Detecting light at night...
Light level: 199
 Lowbeams on: False
 Highbeams on: True
```

This test creates several scenarios with different amounts of light outside. The first has lots of light outside and the car successfully doesn't turn on any headlights. The next is when there is a little amount of light outside and the car turns the low beams on. The last test has no light and successfully turns the high beams on.

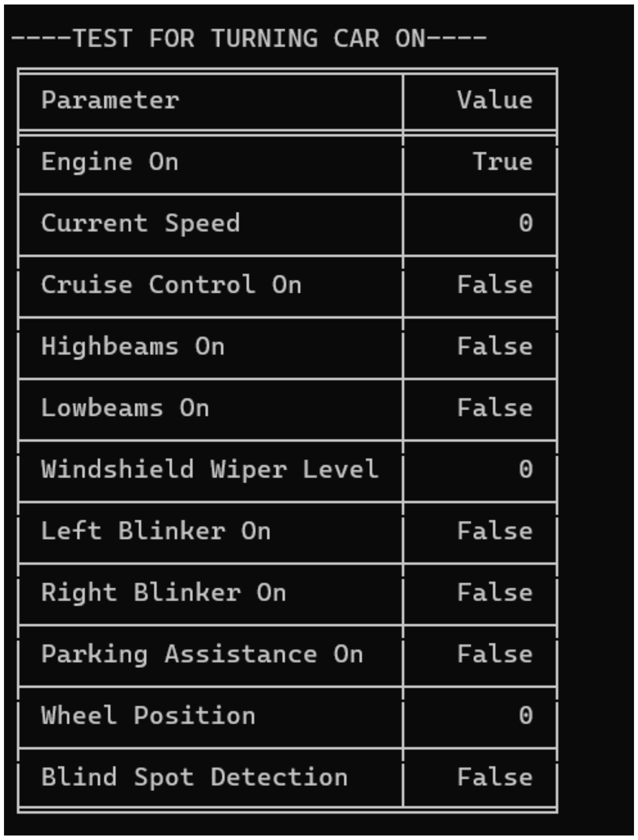
7.1.5 Use-Case 5: Start Vehicle

7.1.5.1 Code

```
print("\n")
print("----TEST FOR TURNING CAR ON----")
```

```
enginestart()
reset()
```

7.1.5.2 Output



```
----TEST FOR TURNING CAR ON----
╔══════════════════════════╦════════╗
║ Parameter                ║  Value ║
╠══════════════════════════╬════════╣
║ Engine On                ║   True ║
║ Current Speed            ║      0 ║
║ Cruise Control On        ║  False ║
║ Highbeams On             ║  False ║
║ Lowbeams On              ║  False ║
║ Windshield Wiper Level   ║      0 ║
║ Left Blinker On          ║  False ║
║ Right Blinker On         ║  False ║
║ Parking Assistance On    ║  False ║
║ Wheel Position           ║      0 ║
║ Blind Spot Detection     ║  False ║
╚══════════════════════════╩════════╝
```

After calling our enginestart() function, we can see that the value that represents the state of the engine is set to True in the display.

7.2 Validation Testing

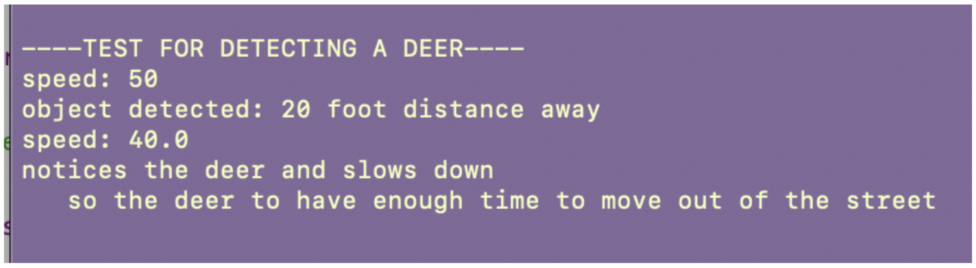
7.2.1 Requirement 1: Detect an object in front of the car and apply brake to avoid collision

7.2.1.1 Code

```
##when deer in front of car
speed = 50 #start speed
deer = 20 #distance
yieldsign = 50 #distance
print("speed: " + str(speed))
print("object detected: " + str(deer) +" foot distance away")
detectobject(deer)
print("speed: " + str(speed))
```

print("notices the deer and slows down \n   so the deer to have enough time to move out of the street")
reset()

7.2.1.2 Output



```
----TEST FOR DETECTING A DEER----
speed: 50
object detected: 20 foot distance away
speed: 40.0
notices the deer and slows down
   so the deer to have enough time to move out of the street
```

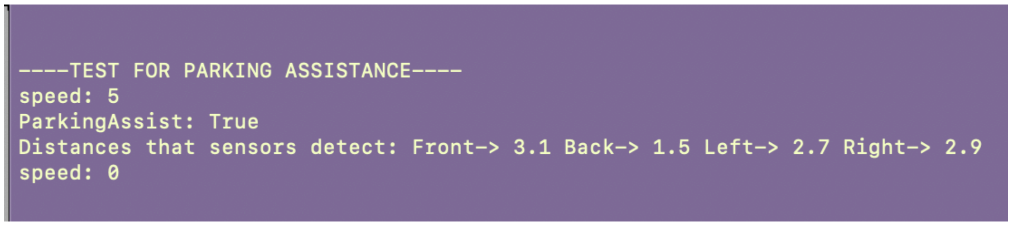
Here the speed before the object was sensed is 50. When the distance of the object detected by the sensor is 20, the speed is reduced accordingly.

7.2.2 Requirement 2: Automated Parking Assistance

7.2.2.1 Code

speed = 5
parkingAssist = True
fDis = 3.1
bDis = 1.5
lDis = 2.7
rDis = 2.9
print("speed: " + str(speed))
print("ParkingAssist: " + str(parkingAssist))
print("Distances that sensors detect: Front-> "+str(fDis)+ " Back-> "+str(bDis)+" Left-> "+str(lDis)+" Right-> " +str(rDis))
parkingSwitch(fDis, bDis, lDis, rDis)
print("speed: " + str(speed))

7.2.2.2 Output



```
----TEST FOR PARKING ASSISTANCE----
speed: 5
ParkingAssist: True
Distances that sensors detect: Front-> 3.1 Back-> 1.5 Left-> 2.7 Right-> 2.9
speed: 0
```

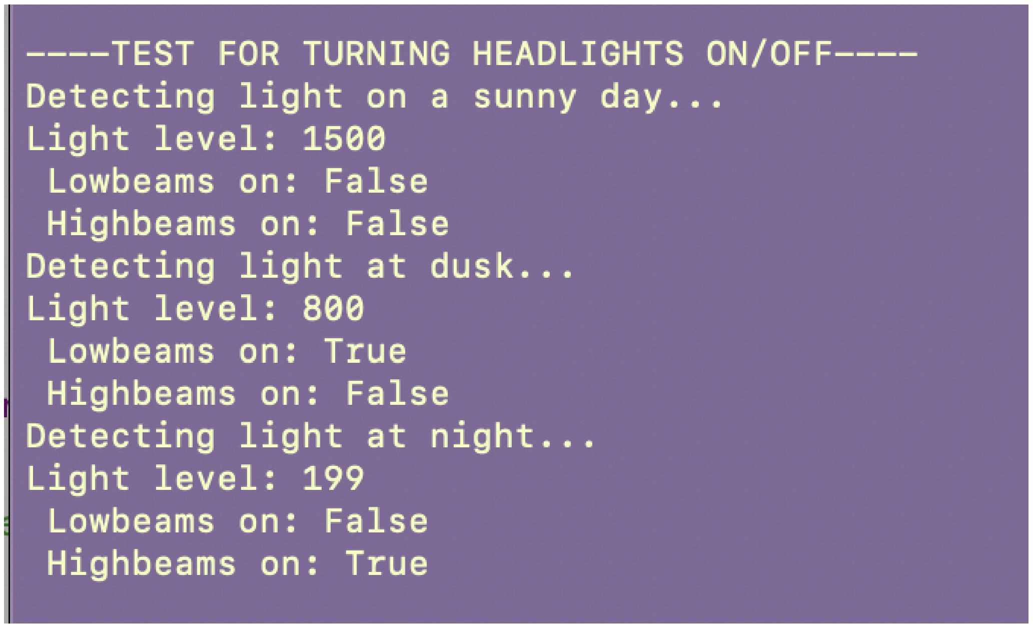
This scenario tests parking when the current speed while parking is 5. Since the Parking Assist setting is set to true, when distance is inputted, since the back distance is less than 2 the speed was reduced to 0.

7.2.3 Requirement 3: Automatic Headlights

7.2.3.1 Code

```
#light levels for time of day
night = 199
dusk = 800
day = 1500
detectlight(day)
print("Detecting light on a sunny day...\nLight level: " + str(day))
print(" Lowbeams on: " + str(lowbeams))
print(" Highbeams on: " + str(highbeams))
detectlight(dusk)
print("Detecting light at dusk...\nLight level: " + str(dusk))
print(" Lowbeams on: " + str(lowbeams))
print(" Highbeams on: " + str(highbeams))
detectlight(night)
print("Detecting light at night...\nLight level: " + str(night))
print(" Lowbeams on: " + str(lowbeams))
print(" Highbeams on: " + str(highbeams))
```

7.2.3.2 Output



```
----TEST FOR TURNING HEADLIGHTS ON/OFF----
Detecting light on a sunny day...
Light level: 1500
 Lowbeams on: False
 Highbeams on: False
Detecting light at dusk...
Light level: 800
 Lowbeams on: True
 Highbeams on: False
Detecting light at night...
Light level: 199
 Lowbeams on: False
 Highbeams on: True
```

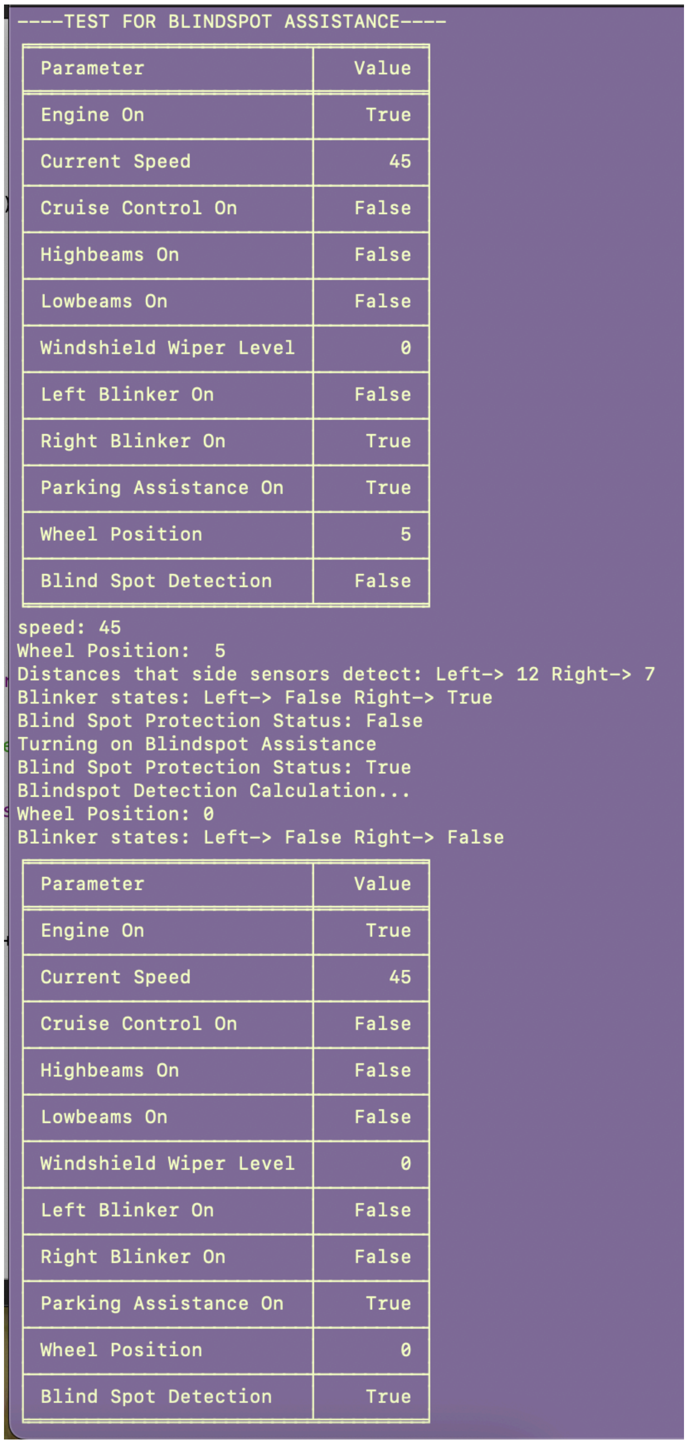
This test creates several scenarios with different amounts of light outside. The first has lots of light outside and the car successfully doesn't turn on any headlights. The next is when there is a little amount of light outside and the car turns the low beams on. The last test has no light and successfully turns the high beams on.

7.2.4 Requirement 4: Blind Spot Warning

7.2.4.1 Code

```
speed = 45
wheelPosition = 5
lDis = 12
rDis = 7
rightBlinker = True
leftBlinker = False
enginestart()
print("speed: " + str(speed))
print("Wheel Position: " , wheelPosition)
print("Distances that side sensors detect: Left-> "+str(lDis)+" Right-> " +str(rDis))
print("Blinker states: Left-> " + str(leftBlinker)+ " Right-> "+str(rightBlinker))
print("Blind Spot Protection Status: " + str(blindSpotProtection))
print("Turning on Blindspot Assistance")
changeBlindSpotSetting()
print("Blind Spot Protection Status: " + str(blindSpotProtection))
print("Blindspot Detection Calculation...")
turning(lDis, rDis)
print("Wheel Position: " + str(wheelPosition))
print("Blinker states: Left-> " + str(leftBlinker)+ " Right-> "+str(rightBlinker))
enginestart()
```

7.2.4.2 Output



```
----TEST FOR BLINDSPOT ASSISTANCE----
+--------------------------+---------+
| Parameter                |   Value |
+--------------------------+---------+
| Engine On                |    True |
| Current Speed            |      45 |
| Cruise Control On        |   False |
| Highbeams On             |   False |
| Lowbeams On              |   False |
| Windshield Wiper Level   |       0 |
| Left Blinker On          |   False |
| Right Blinker On         |    True |
| Parking Assistance On    |    True |
| Wheel Position           |       5 |
| Blind Spot Detection     |   False |
+--------------------------+---------+
speed: 45
Wheel Position:  5
Distances that side sensors detect: Left-> 12 Right-> 7
Blinker states: Left-> False Right-> True
Blind Spot Protection Status: False
Turning on Blindspot Assistance
Blind Spot Protection Status: True
Blindspot Detection Calculation...
Wheel Position: 0
Blinker states: Left-> False Right-> False
+--------------------------+---------+
| Parameter                |   Value |
+--------------------------+---------+
| Engine On                |    True |
| Current Speed            |      45 |
| Cruise Control On        |   False |
| Highbeams On             |   False |
| Lowbeams On              |   False |
| Windshield Wiper Level   |       0 |
| Left Blinker On          |   False |
| Right Blinker On         |   False |
| Parking Assistance On    |    True |
| Wheel Position           |       0 |
| Blind Spot Detection     |    True |
+--------------------------+---------+
```

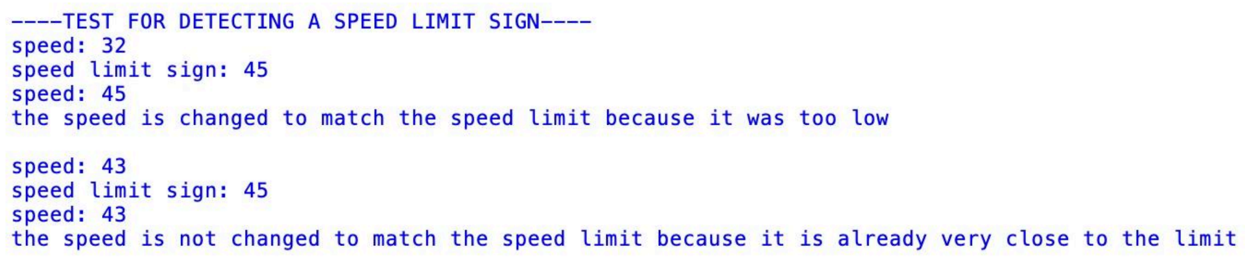
In this scenario, we turn the right blinker on to indicate we want to move the car to the right. After this blind spot assistance turns on and we see that Blind Spot Detection is now True on the display. This shows that the car successfully gave a warning to the driver about the blind spot.

7.2.5 Requirement 5: Speed Limit Sign Detection

7.2.5.1 Code

```
speed = 32
sign = 45
isSign = 1
print("speed: " + str(speed) + "\nspeed limit sign: " + str(sign))
speedlimitsign()
print("speed: " + str(speed))
print("the speed is changed to match the speed limit because it was too low\n")
reset()
speed = 43
sign = 45
print("speed: " + str(speed) + "\nspeed limit sign: " + str(sign))
speedlimitsign()
print("speed: " + str(speed))
print("the speed is not changed to match the speed limit because it is already very close to
the limit\n")
reset()
```

7.2.5.2 Output



```
----TEST FOR DETECTING A SPEED LIMIT SIGN----
speed: 32
speed limit sign: 45
speed: 45
the speed is changed to match the speed limit because it was too low

speed: 43
speed limit sign: 45
speed: 43
the speed is not changed to match the speed limit because it is already very close to the limit
```

This test goes through two different scenarios with the speed limit sign. The first one tests when the car’s speed is significantly below the speed limit, so the speed is set to the speed limit. The second one tests when the car's speed is relatively close to the speed limit. In this case, the speed limit of the car does not change.

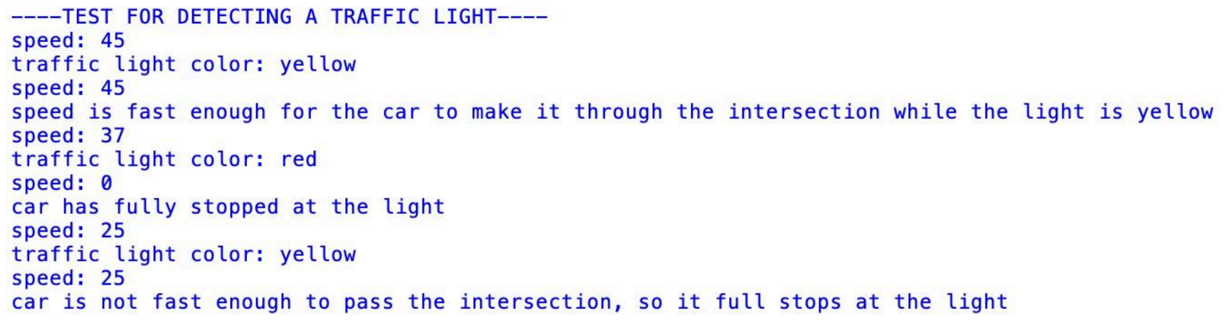
7.2.6 Requirement 6: Car Approaches a Traffic Light and Acts Accordingly

7.2.6.1 Code

```
print("\n")
print("----TEST FOR DETECTING A TRAFFIC LIGHT----")
speed = 45
trafficLight = 1
colorTL = "yellow"
print("speed: " + str(speed) + "\ntraffic light color: " + str(colorTL))
```

```
trafficlight()
print("speed: " + str(speed))
print("speed is fast enough for the car to make it through the intersection while the light is yellow\n")
reset()
speed = 37
trafficLight = 1
colorTL = "red"
print("speed: " + str(speed) + "\ntraffic light color: " + str(colorTL))
trafficlight()
print("speed: " + str(speed))
print("car has fully stopped at the light\n")
reset()
speed = 25
trafficLight = 1
colorTL = "yellow"
print("speed: " + str(speed) + "\ntraffic light color: " + str(colorTL))
trafficlight()
print("speed: " + str(speed))
print("car is not fast enough to pass the intersection, so it full stops at the light")
reset()
```

7.2.6.2 Output



```
----TEST FOR DETECTING A TRAFFIC LIGHT----
speed: 45
traffic light color: yellow
speed: 45
speed is fast enough for the car to make it through the intersection while the light is yellow
speed: 37
traffic light color: red
speed: 0
car has fully stopped at the light
speed: 25
traffic light color: yellow
speed: 25
car is not fast enough to pass the intersection, so it full stops at the light
```

This test runs through 3 different scenarios with the traffic light. The first situation shows the car successfully passes through the intersection when the speed is greater than 40 mph if the light is yellow. The second situation shows that the car stops at a red light. The third situation shows that the car will slow and fully stop at a yellow light if the speed is under 40 mph.

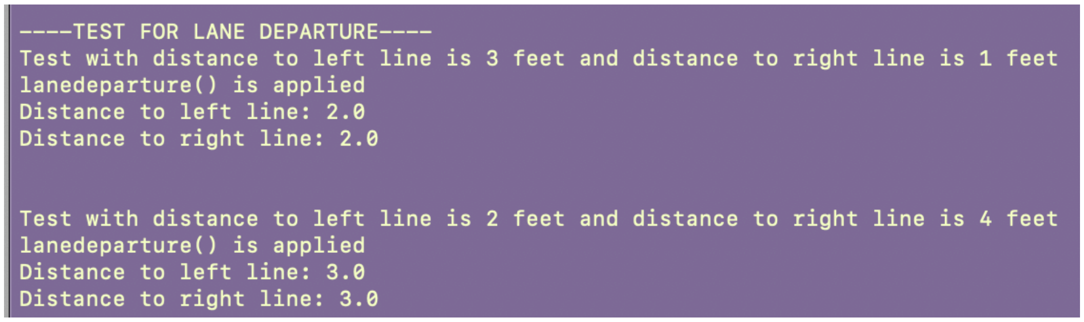
7.2.7 Requirement 7: Lane Departure Adjustments

7.2.7.1 Code

```
isLines = 1
```

```
print("Test with distance to left line is 3 feet and distance to right line is 1 feet")
lineDistLeft = 3
lineDistRight = 1
lanedeparture()
print("lanedeparture() is applied")
print("Distance to left line: " + str(lineDistLeft))
print("Distance to right line: " + str(lineDistRight))
print("\n")
print("Test with distance to left line is 2 feet and distance to right line is 4 feet")
isLines = 1
lineDistLeft = 2
lineDistRight = 4
lanedeparture()
print("lanedeparture() is applied")
print("Distance to left line: " + str(lineDistLeft))
print("Distance to right line: " + str(lineDistRight))
```

7.2.7.2 Output



```
----TEST FOR LANE DEPARTURE----
Test with distance to left line is 3 feet and distance to right line is 1 feet
lanedeparture() is applied
Distance to left line: 2.0
Distance to right line: 2.0


Test with distance to left line is 2 feet and distance to right line is 4 feet
lanedeparture() is applied
Distance to left line: 3.0
Distance to right line: 3.0
```

This test case has two scenarios. One where the car is further to the left than the right. After lanedeparture() is called, we see that the car recentered itself to have equal distance to the left and right line. The new distance is the average of the previous two distances. The other test case also successfully worked with the car closer to the left than the right and then recentered itself.

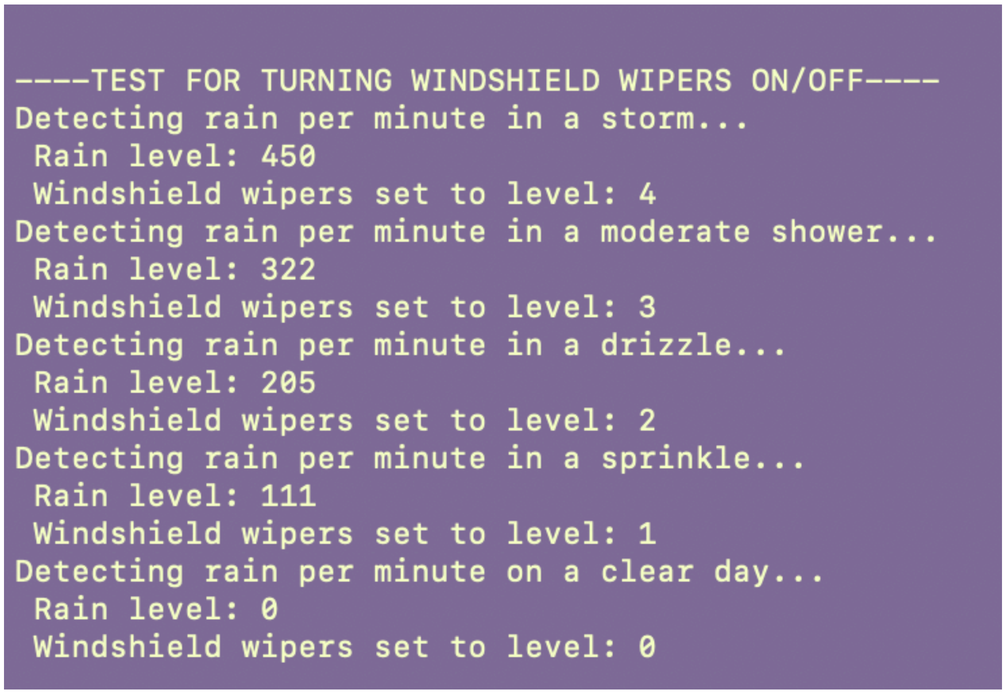
7.2.8 Requirement 8: Automatic Windshield Wipers

7.2.8.1 Code

```
storm = 450
shower = 322
drizzle = 205
sprinkle = 111
clear = 0
detectwater(storm)
```

```
print("Detecting rain per minute in a storm...\n Rain level: " + str(storm))
print(" Windshield wipers set to level: " + str(wipersintensity))
detectwater(shower)
print("Detecting rain per minute in a moderate shower...\n Rain level: " + str(shower))
print(" Windshield wipers set to level: " + str(wipersintensity))
detectwater(drizzle)
print("Detecting rain per minute in a drizzle...\n Rain level: " + str(drizzle))
print(" Windshield wipers set to level: " + str(wipersintensity))
detectwater(sprinkle)
print("Detecting rain per minute in a sprinkle...\n Rain level: " + str(sprinkle))
print(" Windshield wipers set to level: " + str(wipersintensity))
detectwater(clear)
print("Detecting rain per minute on a clear day...\n Rain level: " + str(clear))
print(" Windshield wipers set to level: " + str(wipersintensity))
reset()
```

7.2.8.2 Output



```
----TEST FOR TURNING WINDSHIELD WIPERS ON/OFF----
Detecting rain per minute in a storm...
 Rain level: 450
 Windshield wipers set to level: 4
Detecting rain per minute in a moderate shower...
 Rain level: 322
 Windshield wipers set to level: 3
Detecting rain per minute in a drizzle...
 Rain level: 205
 Windshield wipers set to level: 2
Detecting rain per minute in a sprinkle...
 Rain level: 111
 Windshield wipers set to level: 1
Detecting rain per minute on a clear day...
 Rain level: 0
 Windshield wipers set to level: 0
```

This test case has several scenarios with varying amounts of rain on the windshield. We see that as the amount of raindrops per minute decreases, the intensity of the windshield wipers decreases with it. The first test case is during a storm with a lot of rain, so the intensity is at 4. The last test case is on a clear day, so the intensity is 0, meaning the windshield wipers are off. We see that each test case works successfully.

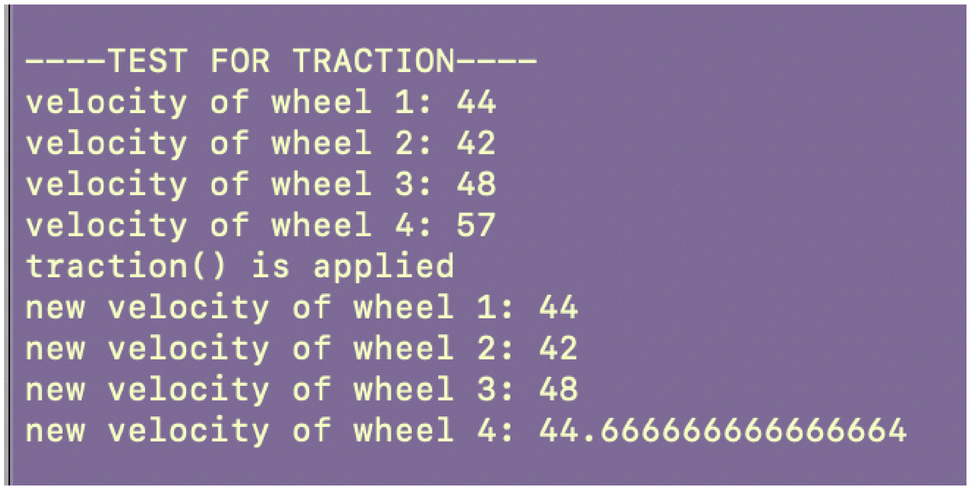
7.2.9 Requirement 9: Traction Control Onset

7.2.9.1 Code

```
v1 = 44
v2 = 42
v3 = 48
v4 = 57

print("velocity of wheel 1: " + str(v1))
print("velocity of wheel 2: " + str(v2))
print("velocity of wheel 3: " + str(v3))
print("velocity of wheel 4: " + str(v4))
traction()
print("traction() is applied")
print("new velocity of wheel 1: " + str(v1))
print("new velocity of wheel 2: " + str(v2))
print("new velocity of wheel 3: " + str(v3))
print("new velocity of wheel 4: " + str(v4))
reset()
```

7.2.9.2 Output



```
----TEST FOR TRACTION----
velocity of wheel 1: 44
velocity of wheel 2: 42
velocity of wheel 3: 48
velocity of wheel 4: 57
traction() is applied
new velocity of wheel 1: 44
new velocity of wheel 2: 42
new velocity of wheel 3: 48
new velocity of wheel 4: 44.666666666666664
```

This test checks the speed of the wheels. Prior to traction() being called, wheel 4 was significantly faster than the others. After the function was called the velocity of wheel 4 equaled the average of the others. This shows our traction() function working successfully.

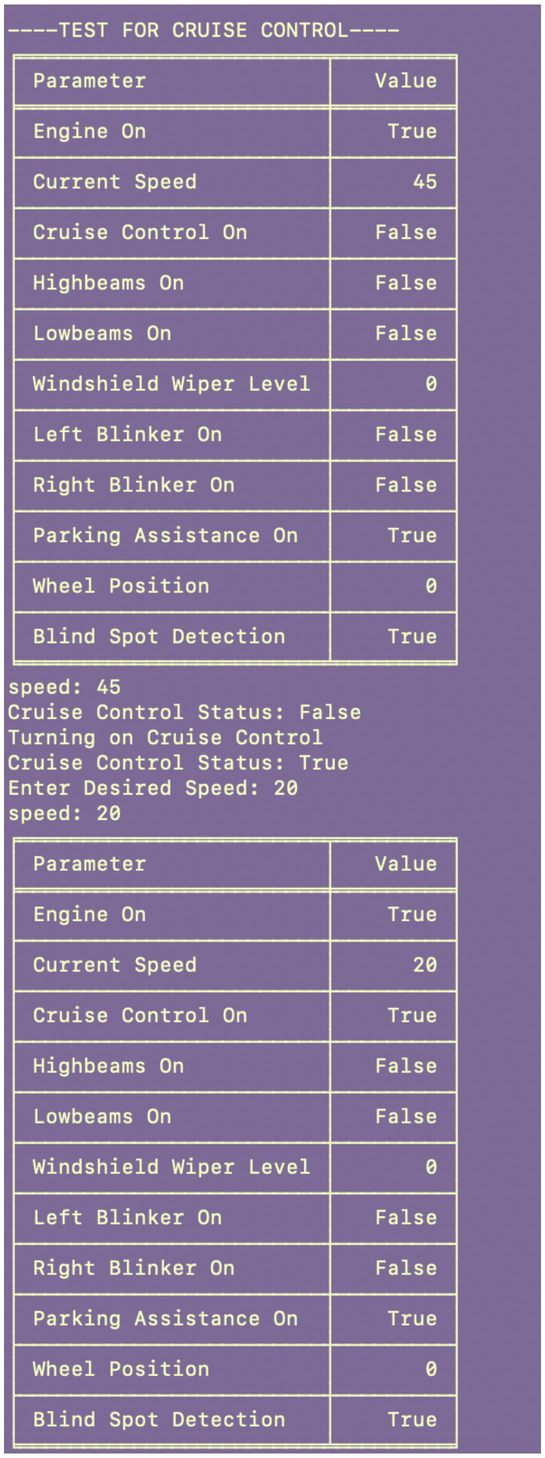
7.2.10 Requirement 10: Cruise Control

7.2.10.1 Code

```
speed = 45
```

```
enginestart()
print("speed: " + str(speed))
print("Cruise Control Status: " + str(ccOn))
print("Turning on Cruise Control")
changeCruiseControl()
print("Cruise Control Status: " + str(ccOn))
cruisecontrol()
print("speed: " + str(speed))
enginestart()
```

7.2.10.2 Output



```
----TEST FOR CRUISE CONTROL----
```

Parameter	Value
Engine On	True
Current Speed	45
Cruise Control On	False
Highbeams On	False
Lowbeams On	False
Windshield Wiper Level	0
Left Blinker On	False
Right Blinker On	False
Parking Assistance On	True
Wheel Position	0
Blind Spot Detection	True

```
speed: 45
Cruise Control Status: False
Turning on Cruise Control
Cruise Control Status: True
Enter Desired Speed: 20
speed: 20
```

Parameter	Value
Engine On	True
Current Speed	20
Cruise Control On	True
Highbeams On	False
Lowbeams On	False
Windshield Wiper Level	0
Left Blinker On	False
Right Blinker On	False
Parking Assistance On	True
Wheel Position	0
Blind Spot Detection	True

In this test case, the car is originally traveling at 45 miles per hour. When cruiseControl() is called, the user is prompted to enter the desired speed. We chose to change the speed to 20 in this scenario. We see that the car successfully slowed down and the current speed changed. And is displayed on the dashboard.