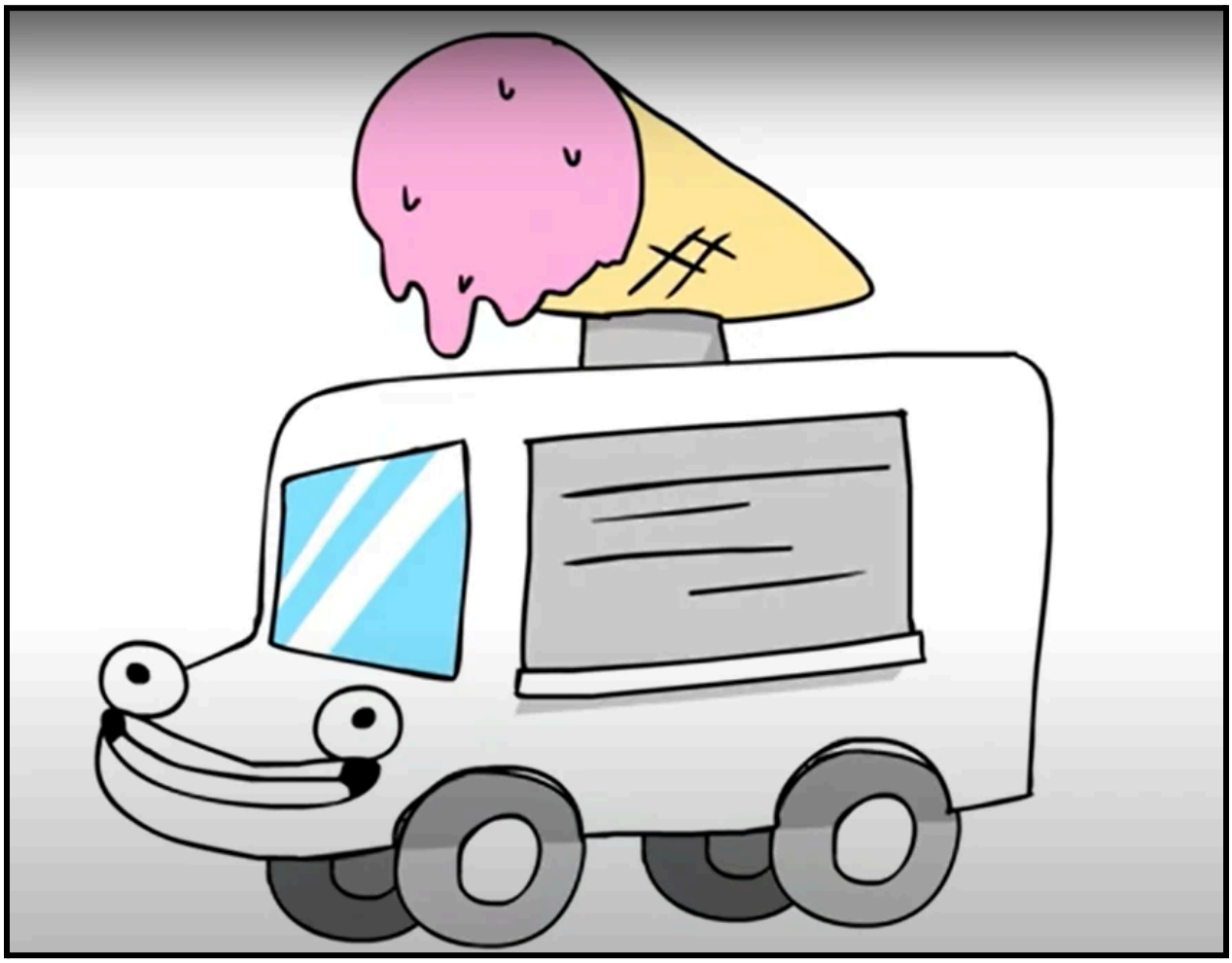
Version 1.5

ALSET - Ice Cream Factory



Brandon Boutin, Tanner Marshall, Aidan Ouckama, Jimmy Zhang

Pledge: I pledge my honor that I have abided by the Stevens Honor System.

Table of Contents

Section 1: Introduction

In this rapidly developing landscape of the automotive industry, our team, Ice Cream Factory, is embarking on a mission to develop the future of transportation – autonomous vehicles (AV). Our primary responsibility lies in creating the software that will power this vehicle of the future. This involves developing advanced algorithms for navigation, obstacle detection, and decision-making to ensure a safe and efficient autonomous driving experience. This document outlines the overview of our project, its scope, features, and the critical aspects we are responsible for in its development.

Overview

Our project's scope is reflected in this initial release, to be followed by subsequent releases that incrementally enhance and add more features to the software. This phased approach allows us to refine and expand our software iteratively. This approach also allows us to add features that the consumers want by using their feedback in our future development. This first release focuses on the foundational elements such as obstacle detection and movement. In the future, we'd like to add two notable features: advanced lane-keeping assistance and intelligent adaptive cruise control. These features will showcase our commitment to the user's safety and convenience.

Advanced Architecture

Our project is a mission-critical real-time embedded system, emphasizing the importance of reliability. We are dedicated to exceeding industry standards for safety and reliability. Therefore, in crafting the software, we will rely on the IoT as our advanced architecture of choice. The IoT framework ensures seamless connectivity and data exchange in real-time. This will place our vehicle at the forefront of this race to develop the best AV.

Software Development Process

To ensure that we succeed, we follow an iterative waterfall software development process encompassing requirements analysis, design, implementation, testing, deployment, and the cycle will repeat. This comprehensive approach begins with a thorough requirement analysis phase,

where we will engage in a detailed exploration of the necessary functionals that our vehicle will need. This involves collaborating closely with stakeholders and end-users, to gather and document precise requirements.

Team Members

Our team consists of highly motivated individuals with experience in coding and problem solving. We all have experience from relevant coursework that we have done. Brandon has done past work with different programming languages as well as used an agile framework on a past project. He is also a great problem-solver, analyzer, and is very focused on any task on hand. Jimmy has experience in rapid prototyping and coding in Java, Python, and C++. He is also skilled in problem solving, effectively communicating, and has an outgoing personality to add to the team's overall cohesiveness. Tanner has worked on a project where an Agile framework was used. He also has experience with many different programming languages including Java and Python, and has worked on many different machine learning projects before. Tanner is good at problem-solving, remaining on task, and effectively communicating with the team. Aidan has experience in project creation, along with agile methodologies. He also has experience in Python and JavaScript. He has expertise in coding, problem solving, and is skilled in communication. With our past experiences, we believe that we have what it takes to tackle this project. Furthermore, we will place a strong emphasis on maintaining organized documentation and following industry-standard processes. This will ensure that our collaboration will be as efficient and well-coordinated as possible, contributing to the success of our project. In the following sections, we provide detailed insights into our project, covering aspects such as brainstorming, planning, architecture, features, and future development processes.

Section 2: Functional Architecture

As the development of an AV presents itself as a complex endeavor with numerous challenges to address, it is imperative to have a comprehensive and organized functional architecture. This architecture will serve as the blueprint for the integration of various modules. These smaller modules will culminate to create a fully autonomous car capable of navigating diverse environments and adhere to all legal requirements.

Architecture

The functional architecture of the AV will encompass a multitude of interconnected modules, each tasked with a specific function essential to the vehicle's operation. These modules can be categorized into four main domains:

1. Sensor Fusion
2. Planning Module
3. Vehicle Control Unit
4. System Management

Within each of these domains, various sub-domains handle individual tasks ranging from object detection and decision-making to system monitoring and maintenance.

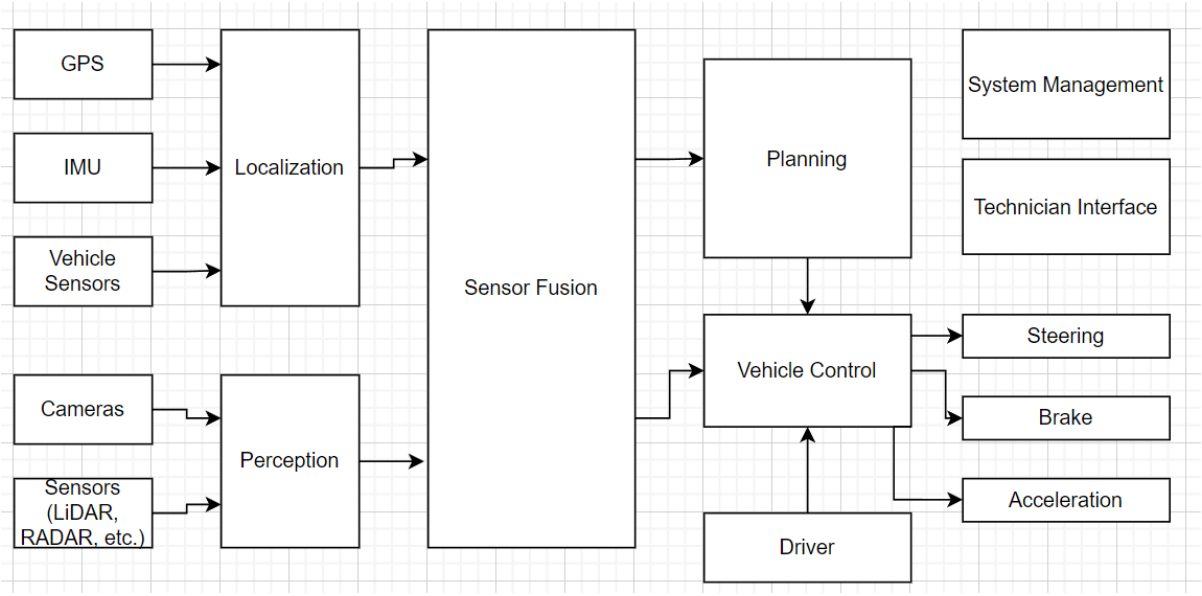


Fig 1. Functional Architecture

Sensor Fusion

The Sensor Fusion module focuses on collecting data from the car's surroundings using its sensors and technology such as GPS, LiDAR, RADAR, and cameras. This data is then integrated using sensor fusion techniques to create a comprehensive representation of the vehicle's surroundings.

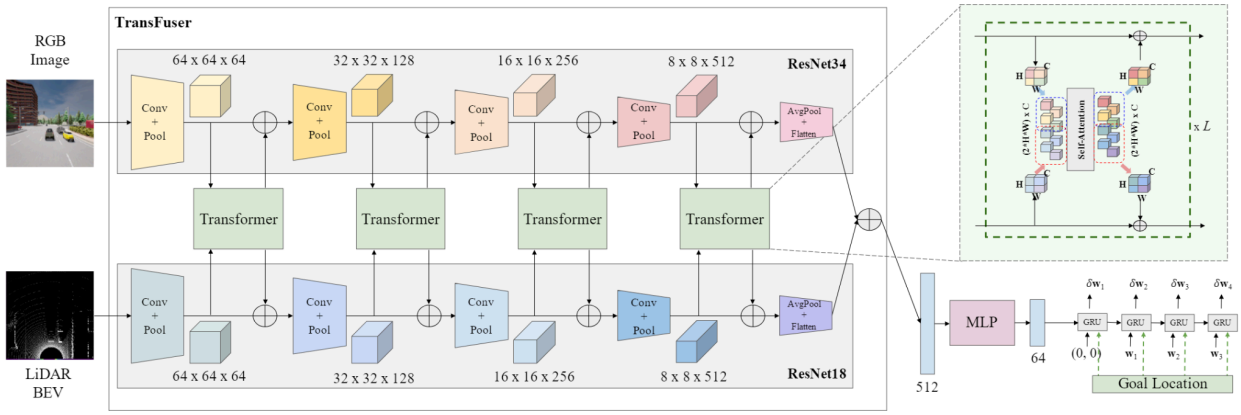


Fig 2. "Object Detection and Classification Unit"

Sub-domains within Sensor Fusion include:

- Localization: Determining the vehicle's precise location relative to its surroundings.
- Perception: Taking inputs from sensor data to identify and recognize objects and features in the environment.

Planning Module

The Planning Module is responsible for decision-making and route planning based on the information gathered by the Sensor Fusion.

Sub-domains within the Planning Module include:

- Route Planning: Determining the optimal path(s) from the origin to the destination.
- Decision Making: Analyzing data from Sensor Fusion and generating possible actions based on predetermined algorithms and objectives.

Vehicle Control Unit

The Vehicle Control Unit focuses on controlling the physical mechanisms of the vehicle, such as steering, accelerating, and braking. This is the module that will execute the decisions made by the Planning Module.

Sub-domains within the Vehicle Control Unit include:

- Steering Control: Adjusting the vehicle's direction based on navigation instructions.
- Acceleration and Braking Control: Maintaining a specified speed or slowing down to adhere to speed limits, traffic conditions, and safety requirements.
- Emergency Response: Implementing emergency responses to imminent threats or unexpected events that are decided on by the Planning Module.

System Management

The System Management module focuses on the management and maintenance of the AV's various components and subsystems. Examples of functions include:

- Diagnostic Monitoring: Continuously monitor the status and performance of critical vehicle systems similar to modern cars. It will check braking, steering, and acceleration, and alerts operators in cases of unexpected problems.
- Log Files: Generate log files that contain timestamps and actions/alerts of the vehicle.
- Software Update: This module enables remote deployment of new algorithms, bug fixes, and improvements to enhance the vehicle's capabilities. This would be done by having a connection to Wi-Fi.

Interface and Communication

The functional architecture needs a robust interface and communication channel between modules to allow for fast and reliable data exchange, coordination, and integration. Inter-module communication relies on our IoT framework to provide this channel, ensuring speed and reliability. By implementing a robust interface and communication framework, our autonomous

vehicle architecture can effectively integrate diverse modules, enable seamless data exchange, and support reliable operation in complex and dynamic environments.

Technician Interface

The Technician Interface sub-domain provides an interface for technicians to interact with the AV system. This interface will allow technicians to monitor system health, diagnose issues, perform troubleshooting, and read log files. Key features of the Technician Interface include:

- Real-time Monitoring: Displaying real-time data and diagnostics information about the AV's components and subsystems.
- Diagnostic Tools: Providing tools and utilities for conducting diagnostic tests and analyzing log files.
- Maintenance Alerts: Alerting technicians to scheduled maintenance tasks, component failures, or critical system errors that require immediate attention.
- Remote Assistance: Allows for remote support and collaboration between on-site technicians and off-site experts for complex troubleshooting.

Cloud Communication

The Cloud Communication sub-domain allows for the establishment of a remote connection between the AV and cloud-based services. This communication enables various functionalities, including data storage and upload, software updates, and remote monitoring. Key components of Cloud Communication include:

- Data Storage and Upload: Transmitting sensor data, logs, and information from the AV to cloud servers for storage.
- Software Updates: Allows for the deployment of OTA software updates, bug fixes, and security updates to the AV's onboard systems from cloud-based repositories.
- Remote Monitoring: Enables operators and administrators to monitor the health, performance, and operational status of the AV from a remote location.

Example

To justify and exhibit the validity of the architecture, let's take a look at the Driver Assistance domain within the AV architecture. Driver Assistance modules play a crucial role in enhancing the driving experience and safety. The module contains sub-modules such as Collision Prevention.

- Collision Prevention: This module employs the vehicle's sensors and cameras. Then transmits that data to the Sensor Fusion to generate a comprehensive image of the surroundings. This new image is then transmitted to the Planning Module which will determine if there is an imminent collision that will happen. If so, the Planning Module will formulate the best action to take (to brake) and send a signal to the Vehicle Control Unit. The Vehicle Control Unit will then execute the braking action. This will then all be compiled into a log file and sent to System Management.

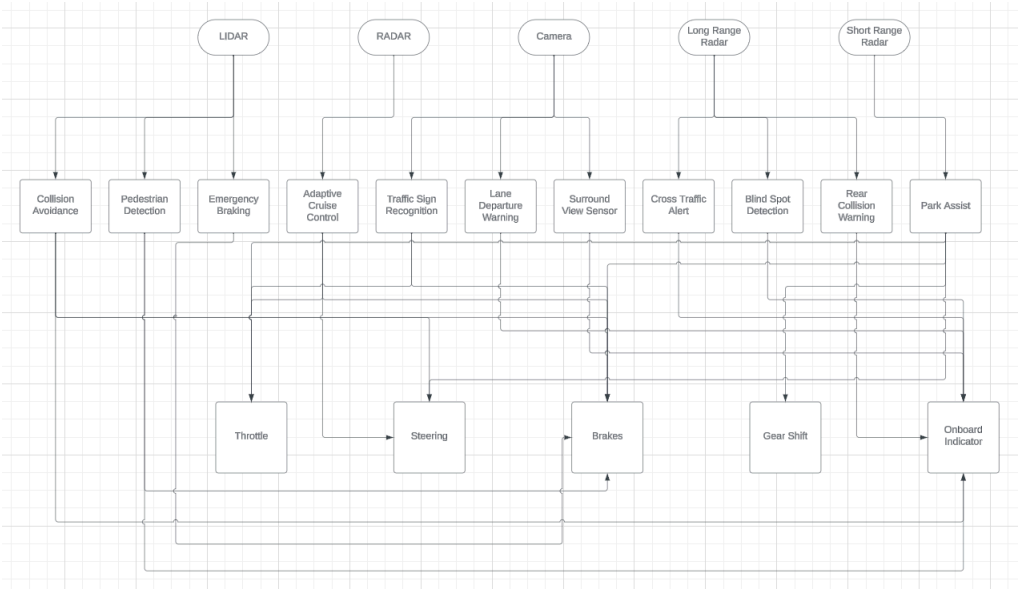


Fig 3. "Wireless Communication and Sensors in Self-Driving Cars"

Section 3: Requirements

In this section, we outline the functional and non-functional requirements that define the capabilities of our autonomous vehicle. These requirements serve as the foundation for the design, development, and testing phases of the project, ensuring that our vehicle meets the highest standards of safety, reliability, and performance. Each requirement is set to address specific functionalities and features of the autonomous vehicle.

Functional Requirements

In this subsection, we delve into the specific functional capabilities that our autonomous vehicle must possess to operate safely and effectively. These functional requirements outline the essential behaviors, features, and performance criteria that our vehicle must meet to fulfill its intended objectives.

3.1 Object Avoidance

Pre-conditions:

1. Driver's seat is occupied and the key is within the vehicle.
2. The vehicle is in motion (Sensor speed > 5 mph).
3. The car's sensors detect obstacle(s) within the system's detection range.
4. The system determines that there is sufficient space and time to maneuver around the obstacle(s) safely.

Post-condition:

1. The vehicle will initiate evasive maneuvers to avoid collision with the detected obstacle(s).
2. The vehicle will resume normal driving operations after successfully avoiding the obstacle(s).

Requirements:

1. Driver's seat sensor detects that the seat is taken, Key Sensor detects that the key is in the car, and Sensor Fusion detects that the vehicle is in motion (> 5 mph).
2. Sensors detect an object or obstacle in the vehicle's path in 200 meters, Sensor Fusion will send the data to the Planning Module.

    a. If the object is detected within 20 feet, sensor fusion will send data to apply brake level 1.
    b. If the object is detected within 15 feet, sensor fusion will send data to apply brake level 2.
    c. If the object is detected within 10 feet, sensor fusion will send data to apply the highest brake level.
3. The Planning module will send a signal to the Vehicle Control Unit to execute the brake action.
4. The system will provide visual and auditory alerts to notify the driver of the impending obstacle and the activation of the Object Avoidance system.
5. Once the obstacle has been successfully avoided, the system will return control to the driver and resume normal driving operations.
6. The Planning Module sends all data to System Management for logging.

3.2 Driver Activate Cruise Control

Pre-conditions:

1. Driver's seat is occupied and the key is within the vehicle.
2. Driver activates cruise control.
3. Vehicle is in motion (sensor speed > 5 mph).
4. The car's sensors (at least 10 feet away from the car in front) and algorithms have determined that it is safe to accelerate/maintain speed.

Post-conditions:

1. The vehicle will maintain the desired speed until the driver manually disables cruise control or if pre-conditions are no longer met.

Requirements:

1. Driver's seat sensor detects that the seat is taken, Key Sensor detects that the key is in the car, and Sensor Fusion detects that the vehicle is in motion (> 5 mph).
2. Driver will locate the cruise control button on the steering wheel and press it.
3. The Cruise Control Sensor will receive the signal and send data to the Sensor Fusion.
4. The Speed Sensor will transmit data of the current speed to the Sensor Fusion.

5. Sensor Fusion will check pre-conditions and if pre-conditions are met, Sensor Fusion will send the signal to the Planning module.
6. Driver will set the speed that they would like the vehicle to cruise at by using the speed adjustment buttons on the steering wheel.
7. Driver will activate the cruise control and the vehicle's sensors will check for all pre-conditions.
8. Vehicle will accelerate to set speed and remain at that speed unless:
    a. obstacles are detected in front of the car.
    b. driver manually applies the brakes.
    c. Sensor Fusion and cameras detect an decrease in speed limit.
    d. driver manually changes the cruise speed.
9. Cruise Control data and adjustments are logged in System Management.

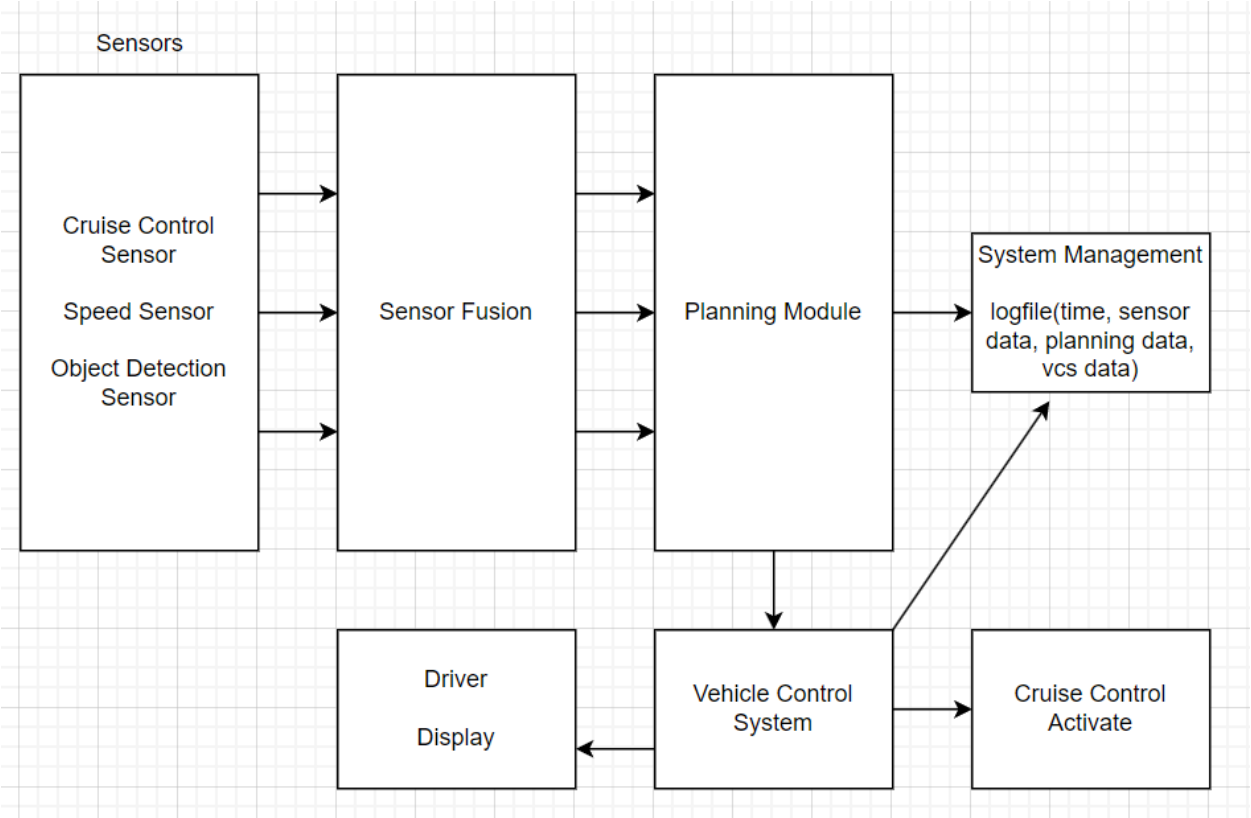


Fig 4. Cruise Control Architecture

3.3 Automated Emergency Braking

Pre-condition:

1. Driver's seat is occupied and the key is within the vehicle.
2. Vehicle is in motion (sensor speed > 5 mph).
3. The car's sensors detect an imminent collision with an obstacle in the vehicle's path.
4. The distance from the obstacle is within the braking distance threshold (15 feet) and the vehicle has not decelerated yet.

Post-condition:

1. The vehicle will initiate emergency braking to mitigate or avoid the collision.
2. The vehicle's speed will be brought to a stop.

Requirements:

1. Driver's seat sensor detects that the seat is taken, Key Sensor detects that the key is in the car, and Sensor Fusion detects that the vehicle is in motion (> 5 mph).
2. Vehicle's sensors are constantly monitoring the vehicle's surroundings and feeding the information to Sensor Fusion.
3. Sensor Fusion will check for imminent collisions.
4. When a potential collision is detected, the system will calculate the time to impact and compare it with the braking distance available (15 feet).
5. If the calculated time to impact is less than the available braking distance, the Automated Emergency Brake system will activate.
6. The system will send a signal to the Brake Control Module to initiate emergency braking.
7. Emergency braking will be applied with enough force to achieve the maximum deceleration possible without posing a danger to passengers.
8. The system will provide visual and auditory alerts to notify the driver of the activation of emergency braking.
9. Once the collision risk is resolved, the system will release the brakes and return control of the vehicle to the driver.
10. The Automated Emergency Braking system will be disabled after the driver manually applies the brakes.
11. The Planning Module sends all data to System Management for logging.

3.4 Lane Keeping Assist (LKA)

Pre-condition:

1. Driver's seat is occupied and the key is within the vehicle.
2. The vehicle must be in motion on a road with visible lane markings.
3. Lane Keeping Assist is enabled by the driver through the vehicle's control interface.
4. The vehicle's sensors are operational and able to detect lane markings.

Post-condition:

1. If the vehicle begins to drift out of its lane without a turn signal activated, the Vehicle Control Unit will apply corrective steering inputs to return the vehicle to the center of the lane.
2. Once the vehicle is back to the center of the lane or the driver takes manual control, LKA will stop making steering corrections.
3. The driver remains responsible for maintaining control of the vehicle and monitoring the road ahead.
4. LKA will provide visual and auditory alerts to prompt the driver that LKA has been activated.

Requirements:

1. Driver's seat sensor detects that the seat is taken, Key Sensor detects that the key is in the car, and Sensor Fusion detects that the vehicle is in motion (> 5 mph).
2. Sensor Fusion will detect the vehicle's position within lane markings on the road using onboard sensors and Sensor Fusion.
3. The onboard sensors must be capable of detecting lane markings under various environmental conditions, including daylight, nighttime, inclement weather, and varying road surface conditions.
4. When the vehicle deviates from its lane without the use of a turn signal, the Planning Module will send corrective steering inputs to the Vehicle Control Unit.
    a. If left tires are within 1 feet of the left lane marking, Sensor Fusion will send data to the Planning module.
    b. If right tires are within 1 feet of the right lane marking, Sensor Fusion will send data to the Planning module.

5. The Planning module will send data to the Vehicle Control Unit to:
    a. Move forward at a 10 degree angle right if left tires are within 1 feet of left lane markings.
    b. Move forward at a 10 degree angle left if the right tires are within 1 feet of right lane markings.
6. The Vehicle Control Unit will send corrective steering inputs to guide the vehicle back to the center of the lane.
7. In the event of lane markings being obscured or unavailable, LKA shall deactivate and provide visual and auditory alerts to prompt the driver that LKA is unavailable.
8. The Planning Module sends all data to System Management for logging.

3.5 Automated Pathfinding

Pre-condition:

1. Driver's seat is occupied and the key is within the vehicle.
2. Vehicle is at a stop (Sensor speed = 0 mph)
3. User does not take manual control of the vehicle.
4. All vehicle sensors are currently working.
5. A valid destination with a valid route has been chosen by the user.

Post-condition:

1. The vehicle will follow the intended route designated by the user.
2. If there is another vehicle ahead, the vehicle will maintain a safe distance while either maintaining the same speed as the distant vehicle or slowing down to keep distance.
3. If the vehicle approaches an intersection it will determine if it needs to stop then safely make a turn when possible.

Requirements:

1. Driver's seat sensor detects that the seat is taken, Key Sensor detects that the key is in the car, and Sensor Fusion detects that the vehicle is not in motion (0 mph).
2. Vehicle's sensors are constantly feeding data to the Sensor Fusion regarding nearby vehicles, markings on the road, stop signs, and traffic signals.

3. Readings from sensors will be made into an image using sensor fusion for 360 degree awareness.
4. The vehicle will not deviate from the chosen route without user intervention.
5. In case of emergency the system will notify the user and prompt them for manual intervention.
6. The Planning Module sends all data to System Management for logging.

3.6 GPS

Pre-condition:

1. Vehicle is started.
2. Driver's seat is taken and the key is in the car.
3. Vehicle's GPS function is working.
4. Vehicle's wireless communications functions are working.
5. Vehicle is at a stop (Sensor speed = 0 mph).

Post-condition:

1. Vehicle will display the current position on a map to the user.
2. Vehicle will display the user's chosen route if one has been designated.
3. Vehicle will indicate traffic or accidents on the map.

Requirements:

1. Driver's seat sensor detects that the seat is taken, Key Sensor detects that the key is in the car, and Sensor Fusion detects that the vehicle is not in motion (0 mph).
2. Driver will input an address to the GPS through the dashboard screen.
3. Inputted data will be sent to the Planning Module.
4. The Planning Module will calculate the quickest route to the desired destination and send it to the display.
5. Vehicle's GPS sensor will constantly update the vehicle's current position on the map.
6. Vehicle will use wireless communication to identify any road hazards along or near the route to be indicated to the user.
7. Vehicle will allow the user to interact with the map in order to show more or less information on the display.

8. The Planning Module sends all data to System Management for logging.

3.7 Automated Parking

Pre-condition:

1. Driver's seat is occupied and the key is within the vehicle.
2. Parking assistance is enabled by the user.
3. Vehicle Sensors are operational.
4. Vehicle is at a stop (Sensor speed = 0 mph).

Post-condition:

1. Vehicle will pull into an open spot at a safe speed.
2. Vehicle will come to a complete stop.
3. Vehicle will place itself into park and inform the user that the parking process is complete.

Requirements:

1. Driver's seat sensor detects that the seat is taken, Key Sensor detects that the key is in the car, and Sensor Fusion detects that the vehicle is in motion (> 5 mph).
2. Onboard sensors will detect the nearby parking space and will detect the relative position and size in respect to the vehicle using sensor fusion.
3. Using the data collected, the vehicle will make the necessary adjustments to steering and will slowly move into the parking space.
4. If necessary the vehicle will come to a complete stop then slowly reverse in order to adjust the vehicle position for a successful park.
5. Once the vehicle has come to a complete stop within the parking area it will put the vehicle into park and inform the user that the parking process has been completed.
6. The Planning Module sends all data to System Management for logging.

3.8 Blind Spot Detection

Pre-condition:

1. Driver's seat is occupied and the key is within the vehicle.
2. Driver takes manual control of the vehicle.

3. Vehicle sensors are operational.
4. Vehicle is in motion (Sensor speed > 5 mph).
5. Vehicle sensors detect a car in the driver's blind spot.

Post-condition:

1. Vehicle will indicate through lights on the rear-view mirrors that there is a car in the driver's blind spot.
2. If the driver attempts to merge into the lane with the blind spot and an obstacle is detected, the system will provide an auditory warning.

Requirements:

1. Driver’s seat sensor detects that the seat is taken, Key Sensor detects that the key is in the car, and Sensor Fusion detects that the vehicle is in motion (> 5 mph).
2. Vehicle sensors will constantly be checking the driver's blind spot for another car.
3. Sensors will constantly be sending data to the Sensor Fusion.
4. Sensor Fusion will analyze the data and determine whether or not there is a car in the blind spot.
5. If there is a car in the blind spot the Sensor Fusion sends a signal to the Planning Module.
6. The Planning Module will send a signal to the Vehicle Control Unit to activate the blind spot detection light and auditory warning.
7. The Planning Module sends all data to System Management for logging.

3.9 Backup Camera and Detection

Pre-condition:

1. The driver's seat is occupied and the key is within the vehicle.
2. Driver takes manual control of the vehicle.
3. Vehicle sensors are operational.
4. Vehicle has been placed in reverse.

Post-condition:

1. Vehicle will display feed from the backup camera on the screen in the car.
2. Vehicle will detect and alert the driver if they are approaching an object.
3. Vehicle will beep faster as the driver gets closer to any object.

Requirements:

1. Driver’s seat sensor detects that the seat is taken and the Key Sensor detects that the key is in the car and the VCU detects that the car is in reverse.
2. Backup camera will be turned on and will display its feed on the screen in the front of the car.
3. Vehicle sensors in the rear and sides of the car will constantly send data to the Sensor Fusion.
4. Sensor Fusion will detect the distance to any obstacles that may be in the path of the car.
5. Sensor Fusion will send out signals for the vehicle to start beeping and will control the speed of that beep based on the distance to any objects.
6. The Planning Module sends all data to System Management for logging.

3.10 Adaptive Headlights

Pre-condition:

1. The driver’s seat is occupied and the key is within the vehicle.
2. Vehicles headlights have been turned on.
3. Vehicle sensors are operational.
4. Adaptive headlights' swivel mechanism is operational.

Post-condition:

1. Vehicle will swivel the headlights based on the direction that the car is being steered to allow for better visibility around curves.

Requirements:

1. Driver’s seat sensor detects that the seat is taken and the Key Sensor detects that the key is in the car.
2. The vehicle’s steering wheel input will be sent to the Planning module.
3. The Planning module will continuously calculate the angle for the headlights using the input.
4. The calculations will be sent to the Vehicle Control Unit to angle the headlights in the optimal angle.

5. Vehicle will continuously change the angle of the headlights based on the calculated angle.
6. The Planning Module sends all data to System Management for logging.

3.11 Turn Car On

Pre-condition:

1. Driver has a key within 10 feet of the car.

Post-condition:

1. Car is turned on.

Requirements:

1. Key Sensor records key within 10 feet of the vehicle.
2. Sensor Fusion receives key present (key present value is 1).
3. The Seat Sensor detects that the driver is on the drive seat.
4. Driver pushes the “Engine On” button.
5. Sensor Fusion passes “Key Present” and “Engine On” to the Planning module.
6. The Planning Module will send a request to VCU to turn the engine on.
7. VCU sends turn on to starter.
8. Display indicated “Engine On”.
9. The Planning Module sends all data to System Management for logging.

Non-Functional Requirements

In this section, we explore the critical non-functional requirements that define the quality attributes, constraints, and characteristics of our autonomous vehicle. Non-functional requirements encompass a wide range of considerations, including reliability, security, usability, and scalability, among others.

3.12 Reliability

Pre-condition:

1. System software is up to date on the most recent software update.
2. System components, including hardware and software, are properly installed and configured.

3. Vehicle's power supply is stable and within acceptable voltage ranges to prevent sudden system failures or interruptions.

Post-condition:

1. System software shall have a five-nine reliability (99.999% reliability).
2. Automated monitoring and alerting mechanisms are in place to detect and promptly address any deviations from expected reliability metrics.
3. Regular performance and reliability audits are conducted to verify compliance

Requirements:

1. System software will achieve a five-nine reliability, with software-related failures occurring less than 0.001% of the time over the system's operational lifespan.
2. Rigorous testing and validation procedures will be conducted throughout the development lifecycle to verify the reliability.

3.13 Performance

Pre-condition:

1. System software is operational; sensor system is actively scanning the vehicle's condition.
2. The communication channels between sensors and the VCU (Vehicle Control Unit) activator are clear and unobstructed.
3. VCU activator is ready to receive and process messages from sensors.

Post-condition:

1. Duration of a message from sensor to VCU activator shall be less than 1 ms.
2. The response time of the autonomous functions, such as object avoidance and lane keeping, shall not degrade by more than 5% over the vehicles lifetime
3. System shall maintain consistent performance under varying environmental conditions, including extreme temperatures (-20°C to 50°C) and inclement weather.

Requirements:

1. Vehicle will maintain >95% accuracy of autonomous vehicle functions throughout vehicle lifetime.

2. Vehicle will maintain consistent performance throughout varied weather conditions, including rain, snow, and extreme temperatures.

3.14 System Management

Pre-condition:

1. Technician is logged into the system.
2. The key is within the vehicle.
3. Vehicle is at a stop (Sensor Speed = 0 mph).
4. System has cloud access.
5. System shall consistently track performance metrics, including response time and system efficiency, which shall be regularly monitored and logged.

Post-condition:

1. Technician shall be able to retrieve and read log files.
2. Technician shall be able to perform diagnostics on vehicle.
3. Technician will have access to update software.

Requirements:

1. The technician is logged into the system with required credentials.
2. The system shall provide the technician with the ability to retrieve and read log files generated by the Vehicle Control Unit.
3. Tehnician shall have access to diagnostic tools and functionalities to perform comprehensive diagnostics on the vehicle's system.
4. Technician shall have access to update software installed on the vehicle.
5. The system should continuously track performance metrics, including response time with time stamps, in real-time.

3.15 Security

Pre-condition:

1. System software is operational; sensors such as key sensors and locking mechanisms are functioning.
2. Vehicle is at a stop (Sensor Speed = 0 mph).
3. System logs are generated and maintained.

Post-condition:

1. When the key sensor detects that the key is more than 10 feet away, the car doors will automatically lock.
2. Technician-only access to security system through username and password login.
3. System will track all sensor activities and access attempts for audit and monitoring purposes.

Requirements:

1. The vehicle's key sensor will monitor distance between the key and the vehicle.
2. When the key is beyond 10 feet away from the vehicle, the key sensor will send a signal to the Vehicle Control Unit to lock the vehicle.

3.16 Software Update

Pre-condition:

1. Vehicle is at a stop (Sensor Speed = 0 mph).
2. The driver's seat is occupied and the key is within the vehicle.
3. System Management has cloud access.
4. System is operational.

Post-condition:

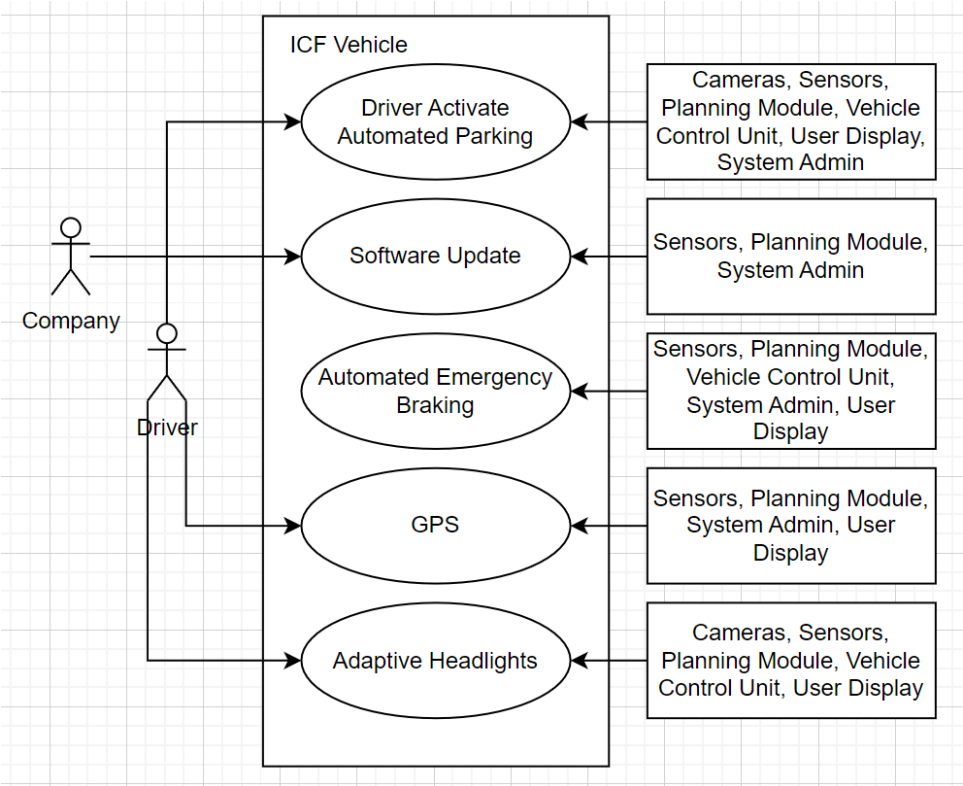
1. Vehicle checks for available software updates on activation.
2. System will make daily fetches for software updates.
3. System will have access to system updates through cloud access.
4. Technician will have access to manual updates and system version changes.

Requirements:

1. The vehicle's software system checks for available software updates via connection to the cloud.
2. If there is an update, the system will give a visual prompt to the user asking if the user would like to update the system.
3. If the user accepts, the system will update, else the system will remind the user when they start the car again.

Section 4: Requirement Modeling

4.1 Use Case Scenarios



4.1.1 Driver Activate Automated Parking

Precondition: Car is on, sensors are functional, car is not in motion.
Postcondition: Car is in park, display indicates successful park.
Trigger: Driver activates the automated parking system.

1. Driver presses the automated parking button.
2. Display shows the “Request” state and that sensors are looking for valid parking spots.
3. Automated parking sensors detect an available parking spot and send the data to sensor fusion.
4. Sensor fusion sends the current location and orientation of the parking spot, vehicle and nearby obstructions to the Planning module.
5. Planning module calculates if the designated parking spot is valid.
6. If the parking spot is valid, the Planning module will send the request to VCU to maneuver into the parking spot.
7. Exception: If there are no valid parking spots then planning will send the message “Denied” to the driver display.
8. Planning module sends all related data to System Admin for logging.

9. VCU sends all related data to System Admin for logging.
10. Other possible exceptions to automated parking:
    a. Driver activates the brakes (canceling automated parking).

4.1.2 Software Update

Precondition: Vehicle's built-in modem is functional.
Postcondition: Vehicle's software is updated to the latest version.
Triggers: Company launches new update for the vehicle's software via the cloud.

1. Company launches new update(s) via the cloud.
2. Planning module will check if the vehicle's built-in modem is functional. If yes, proceed to the next step, else notify the driver that the modem is faulty using visual and audio cues.
3. Planning module will check if the vehicle's battery is above 10 percent. If yes, proceed to the next step, else notify the driver that the battery is below threshold for update using visual and audio cues.
4. Vehicle downloads the new update(s).
5. System Admin will log all actions taken during the process of updating.
6. Exceptions to Software Update:
    a. Built-in cellular modem is not functional.
    b. Built-in cellular modem experiences low connectivity.
    c. Vehicle's battery is too low (below 10%).
    d. The vehicle's hardware is outdated.
    e. User opted out of software updates.

4.1.3 Automated Emergency Braking

Precondition: Car is on, sensors are functional, and vehicle is in motion (speed greater than 10 mph).
Postcondition: Vehicle will be brought to a stop, visual and audio cues will be shown.
Trigger: The Planning module determines that there is an imminent collision from data from Sensor Fusion.

1. Sensors are constantly sending data of the vehicle's surroundings to Sensor Fusion.
2. Sensor Fusion sends data to the Planning module.
3. The Planning module calculates if there is an imminent collision in front of the vehicle.
4. Once detected, the Planning module will calculate how much braking force should be applied depending on
    a. Distance from front of vehicle to the object of collision.
    b. Speed the vehicle is traveling at.
        i. Distance <= 10 feet, speed <= 25 mph: apply brake level 1
        ii. Distance <= 10 feet, 25 mph < speed <= 35 mph: apply brake level 2
        iii. Distance <= 10 feet, 35 mph < speed: apply max brake level

5. The Planning module will send a signal to VCU to execute braking and play audio and visual cue.
6. Display shows visual cue of Emergency Braking and audio cue will play via speakers.
7. The Planning module sends all relevant information to System Admin for logging.
8. VCU sends all relevant information to System Admin for logging.
9. Exceptions to Emergency Braking:
    a. Driver brakes and stops the collision manually.

4.1.4 GPS

Precondition: Vehicle is on and not in motion. GPS is functional. The vehicle's localization module is functional.
Postcondition: GPS will show on the cars display the drivers path.
Triggers: The driver activates GPS.

1. Vehicle is not in motion and the driver activates GPS through the vehicle's display.
2. Driver inputs address which gets sent to the Planning module.
3. The Planning module calculates the quickest route to the destination.
4. Exception: If the address can not be found the system will alert the driver on the display.
5. Planning sends the quickest route back to the display and shows the map view with the path along with the drivers next direction.
6. GPS sensors constantly send the vehicle's location back to planning.
7. Planning constantly updates the display and the quickest route based on the vehicle's current position.
8. Wireless communication is constantly sending data to planning on any road hazards.
9. Planning alerts the driver based on any hazards.
10. Other possible exceptions to GPS:
    a. Route can not be calculated.
    b. The inputted address does not exist.

4.1.5 Adaptive Headlights

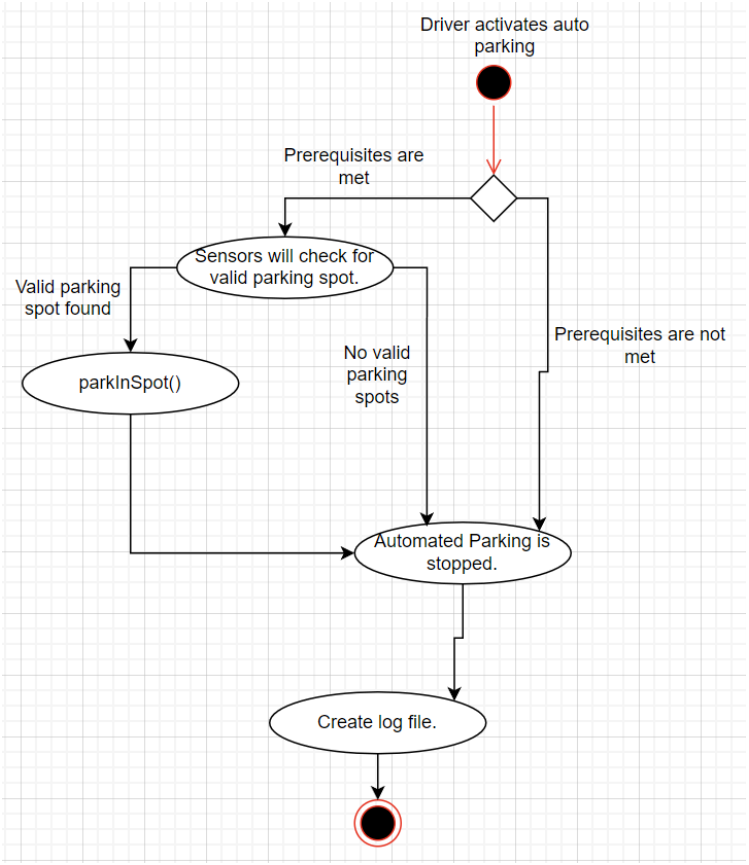
Precondition: Vehicle is on, sensors and cameras are functional, adaptive headlight's swivel mechanism is functional.
Postcondition: Headlights will be swiveled to provide better visibility around curves and turns.
Triggers: The driver activates the headlights.

1. The driver activates adaptive headlights using the driver interface.
2. The vehicle's sensors continuously monitor the surroundings to detect lighting conditions and the vehicle's movement.
3. This data is sent to Sensor Fusion which creates a processed image that is sent to the Planning Module..

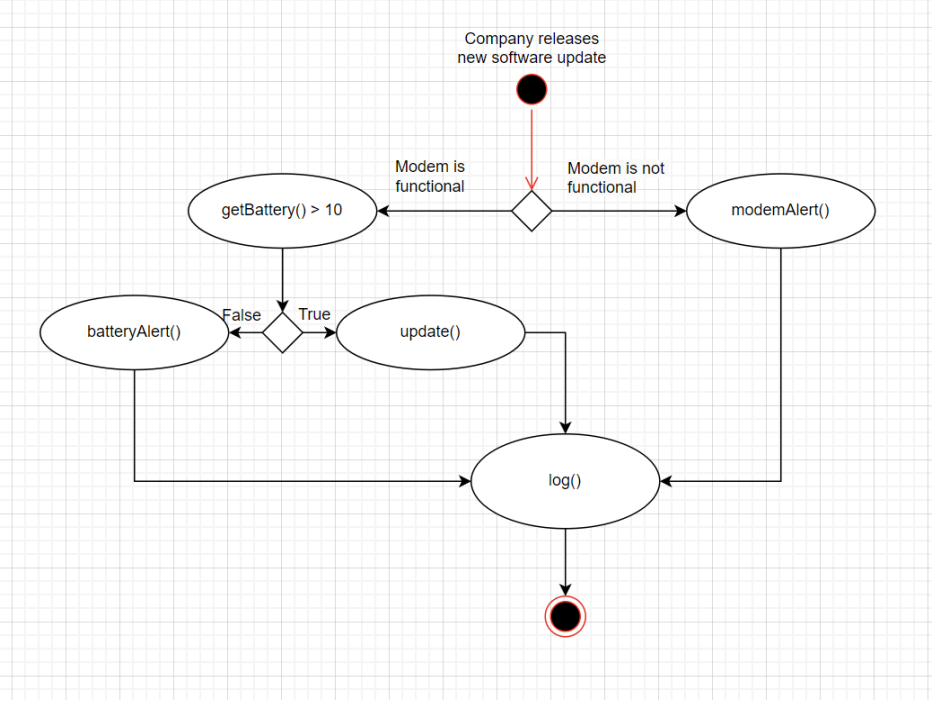
4. The Planning module determines the appropriate swivel angle for the headlights based on the vehicle's speed, steering angle, and detected curvature of the road and sends a signal to the VCU to execute the command.
    a. ⅛ the angle of the turn in the same direction if the road is curved.
    b. Swivels back to straight when the road is straight.
5. The Vehicle Control Unit will swivel the headlights following instructions from the Planning module.
6. Visual cues on the dashboard indicate the activation of Adaptive Headlights, ensuring the driver is aware of the system's operation.
7. Relevant information regarding the activation of Adaptive Headlights is logged by the System Admin.
8. Exceptions:
    a. Manual deactivation by the driver.
    b. Vehicle swivel is detected to have malfunctioned.
        i. Alerts driver via visual cue on the dashboard.
        ii. Headlights are adjusted to standard fixed position.

4.2 Activity Diagrams

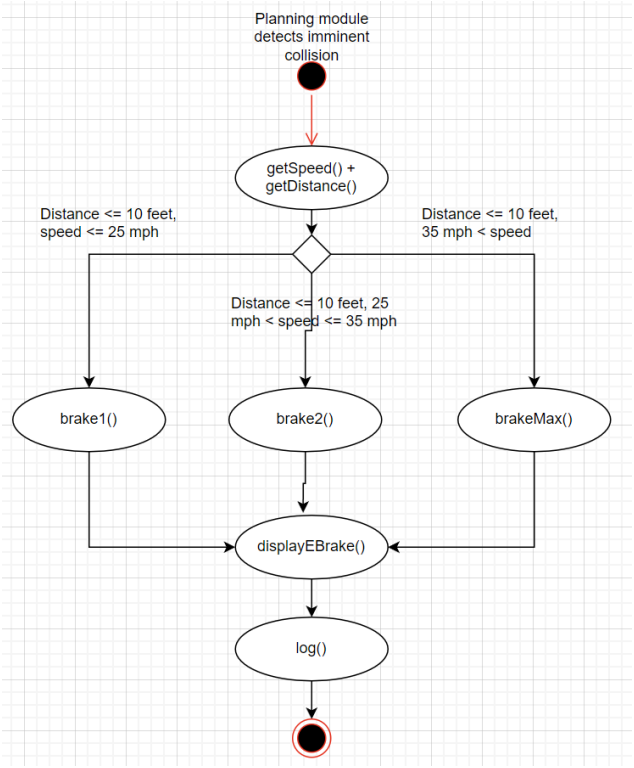
4.2.1 Driver Activate Automated Parking



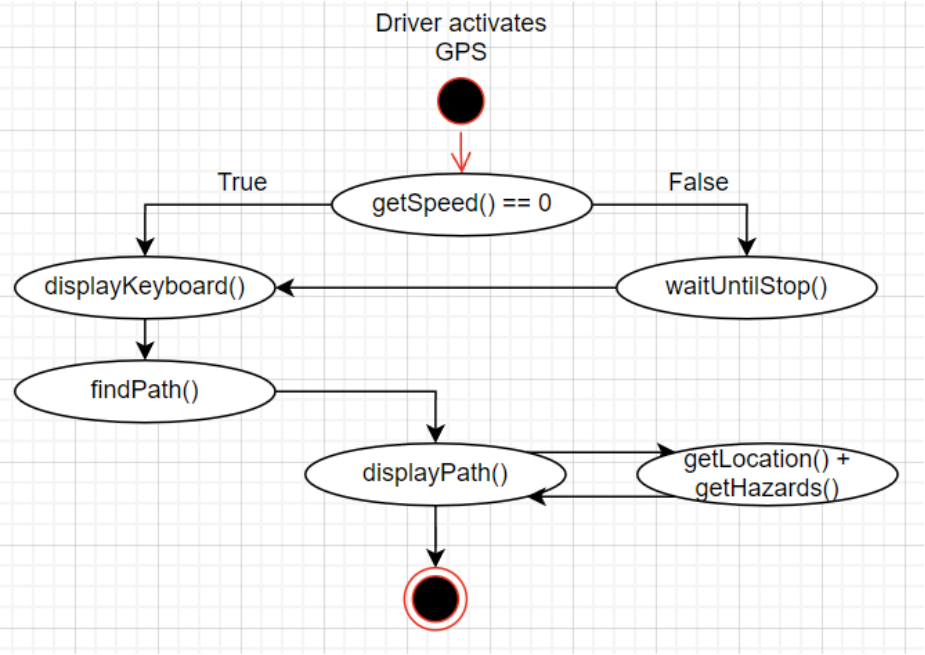
4.2.2 Software Update



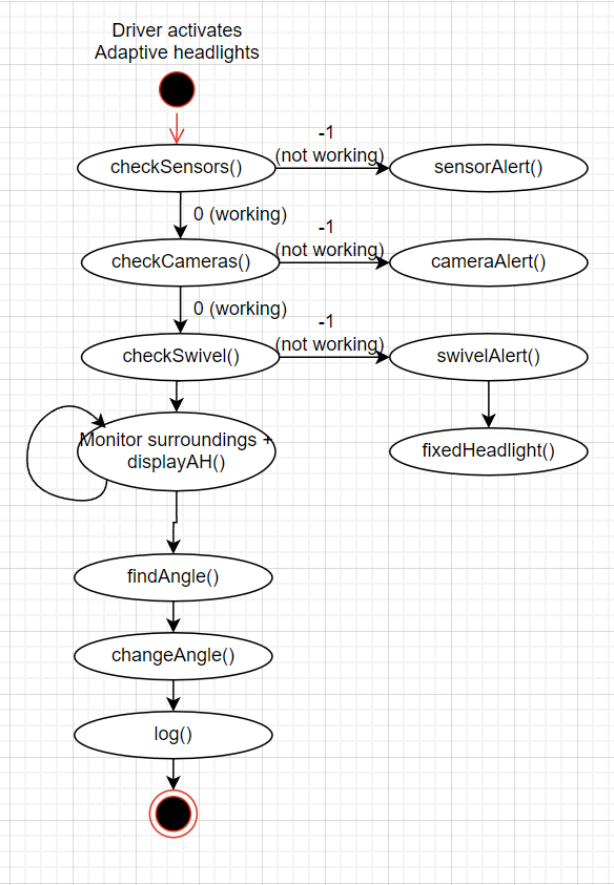
4.2.3 Automated Emergency Braking



4.2.4 GPS

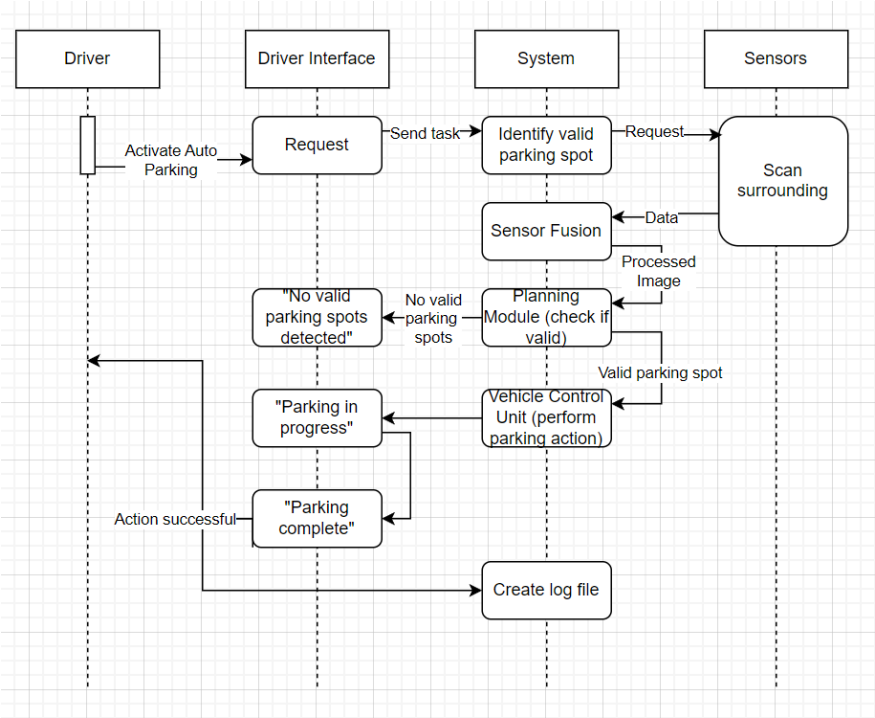


4.2.5 Adaptive Headlights

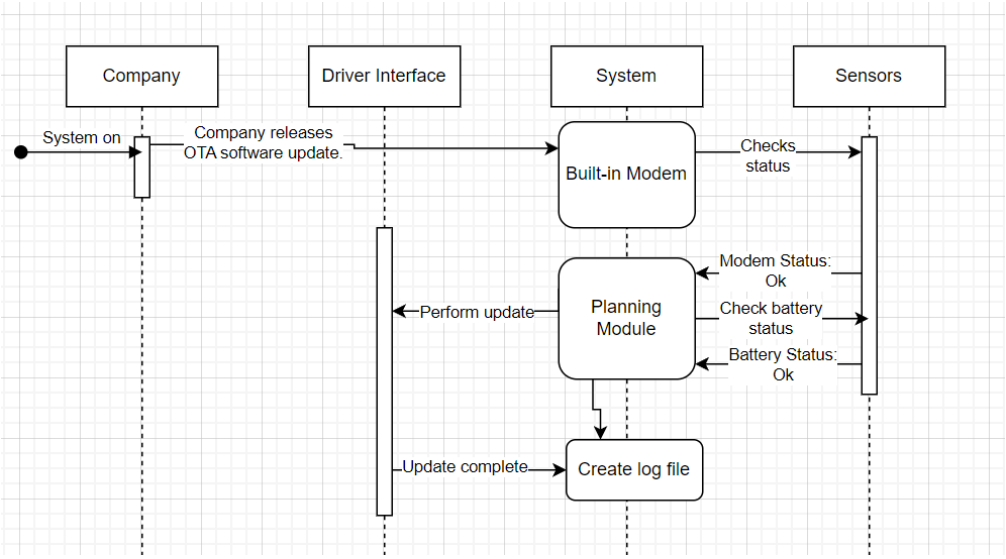


4.3 Sequence Diagrams

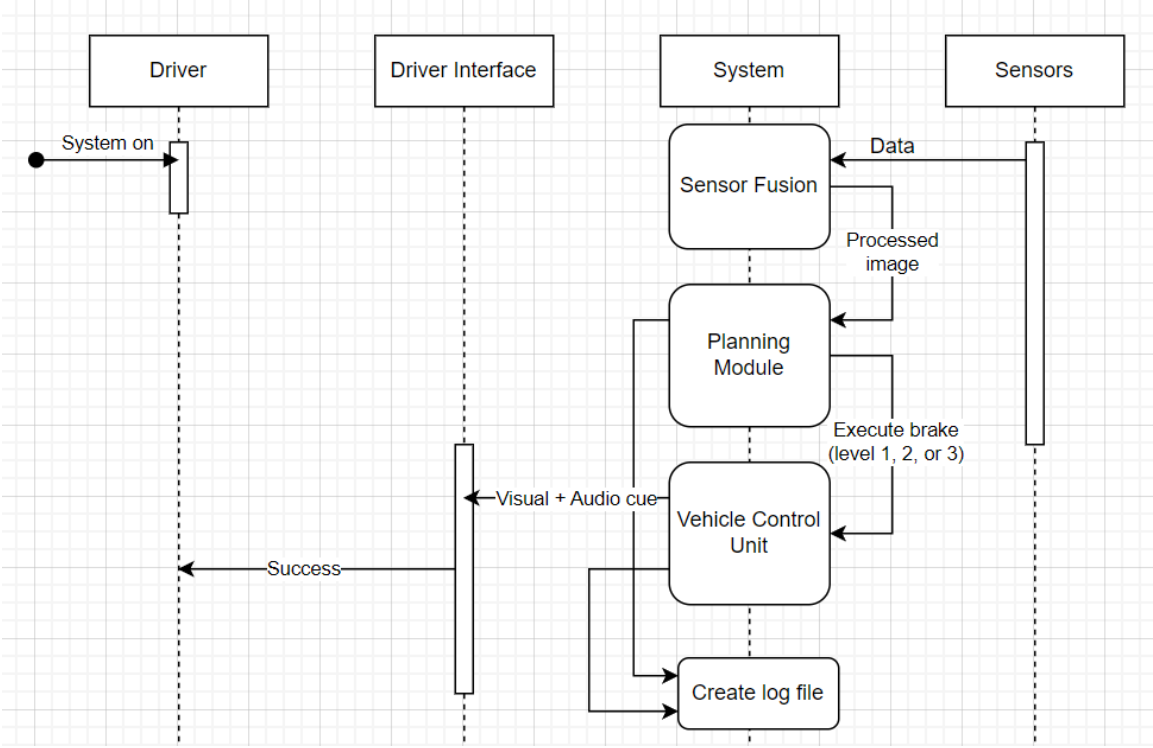
4.3.1 Driver Activate Automated Parking



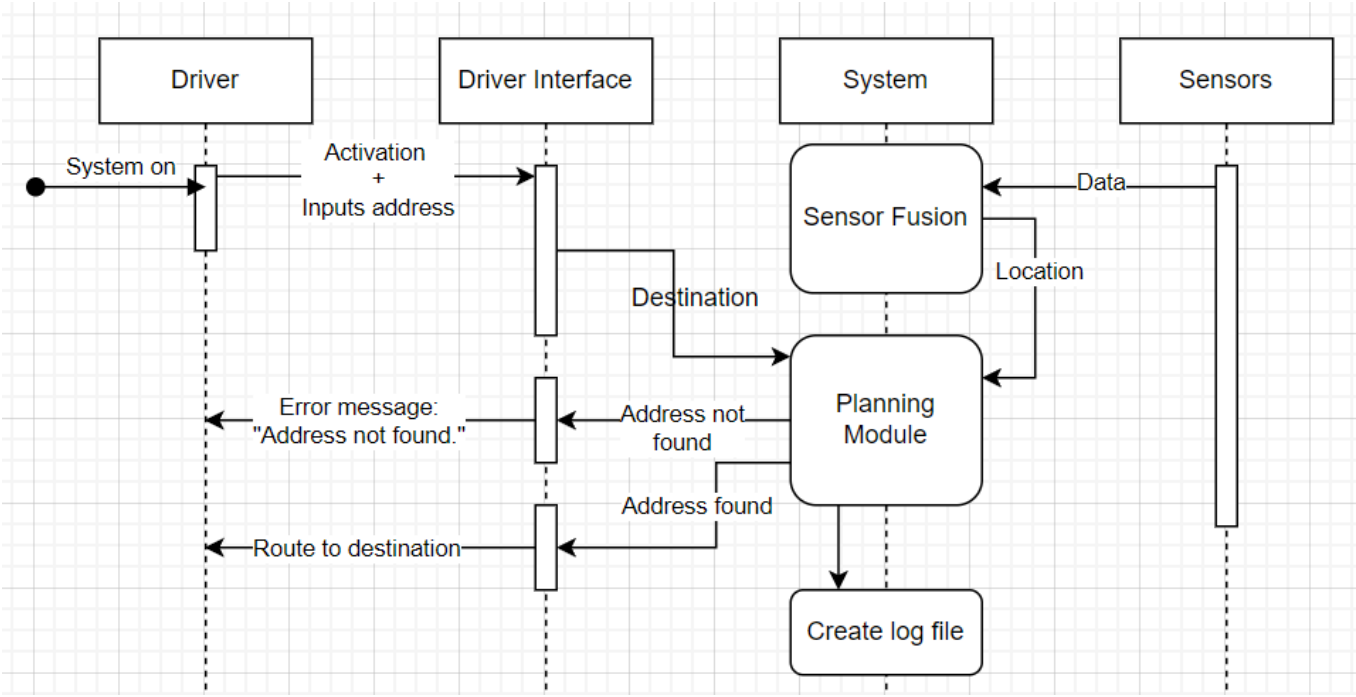
4.3.2 Software Update



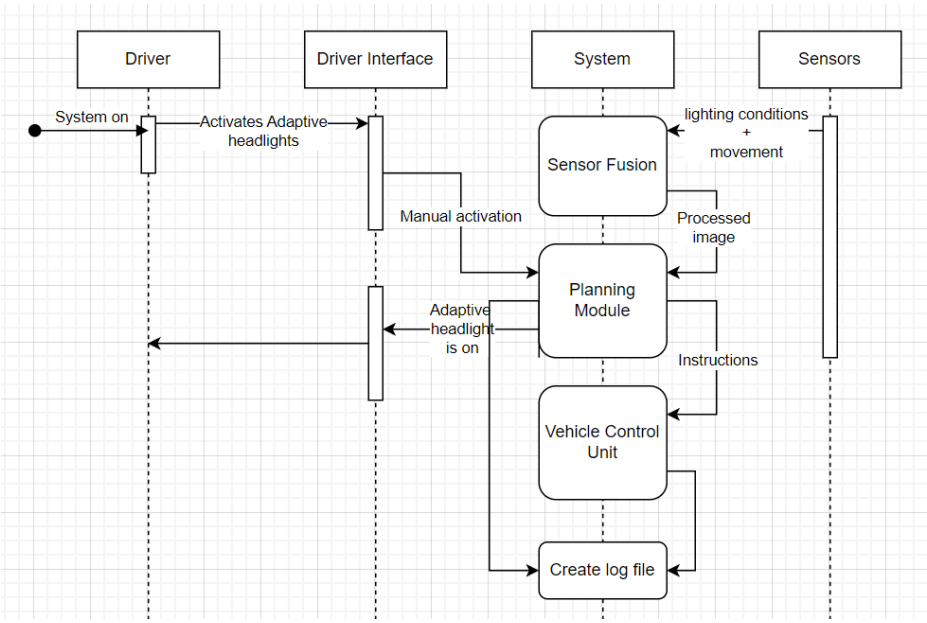
4.3.3 Automated Emergency Braking



4.3.4 GPS

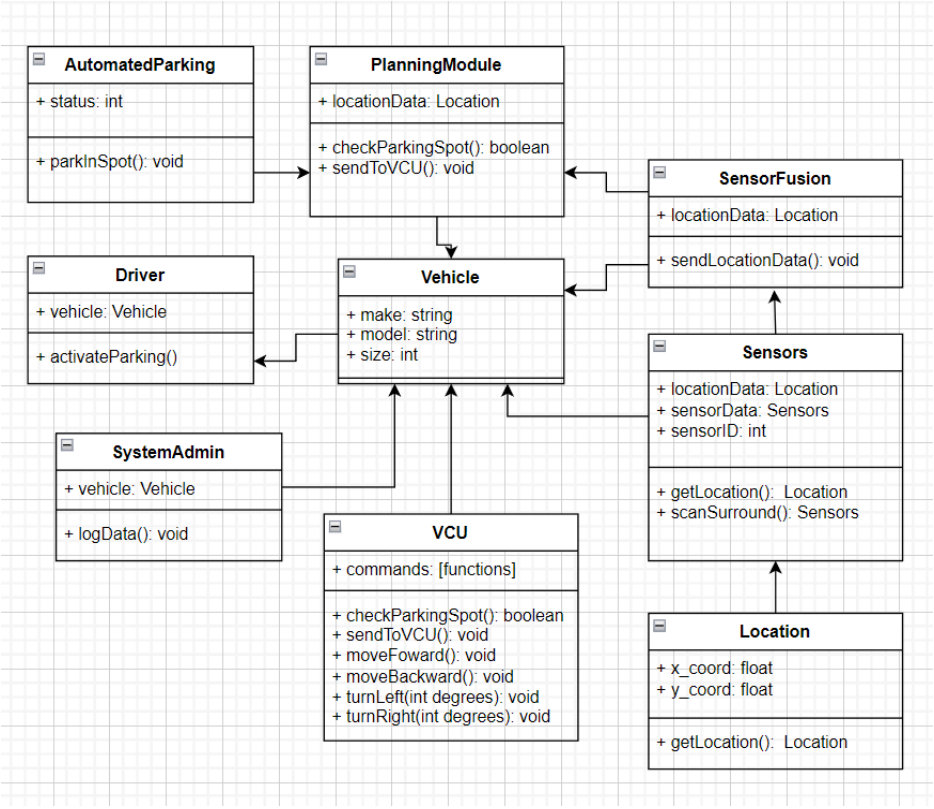


4.3.5 Adaptive Headlights

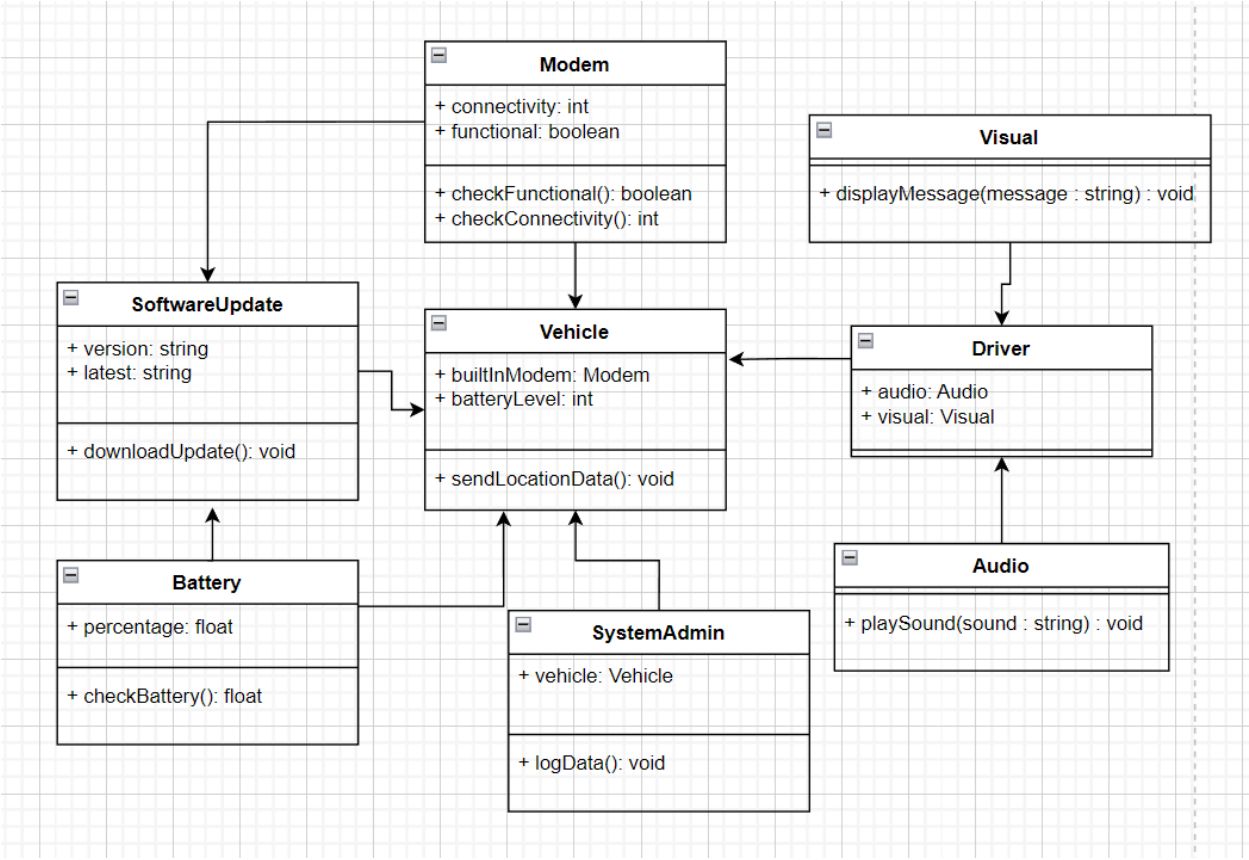


4.4 Classes

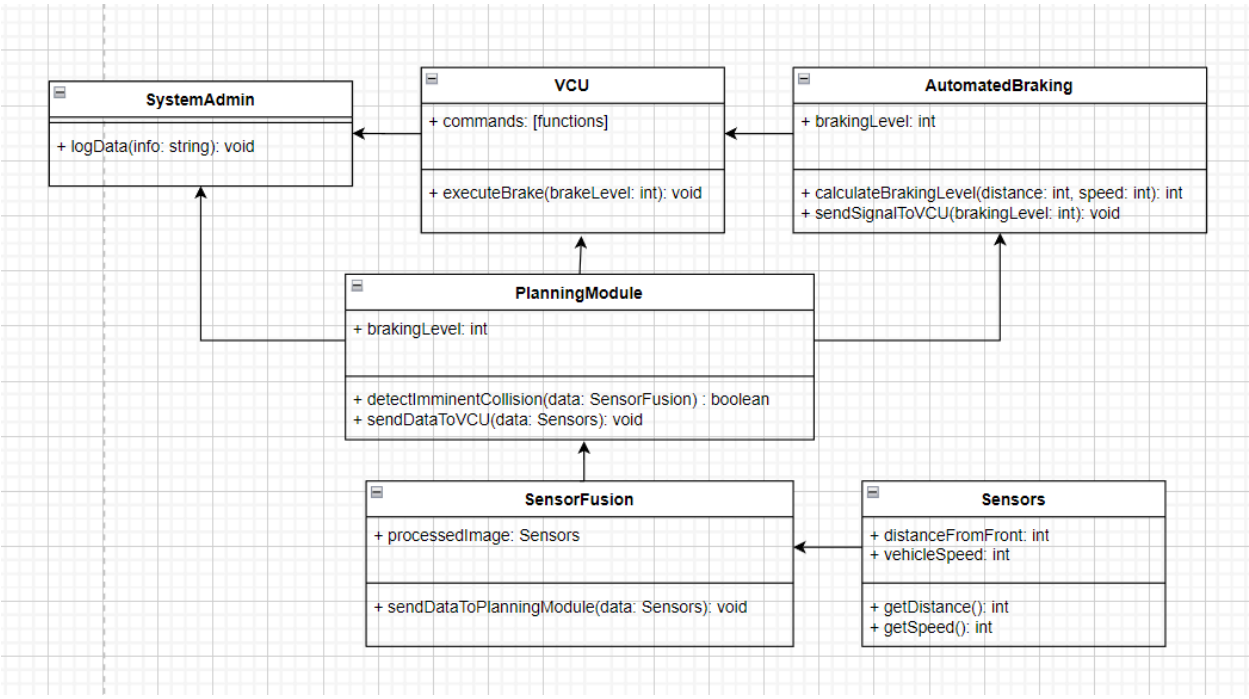
4.4.1 Driver Activate Automated Parking



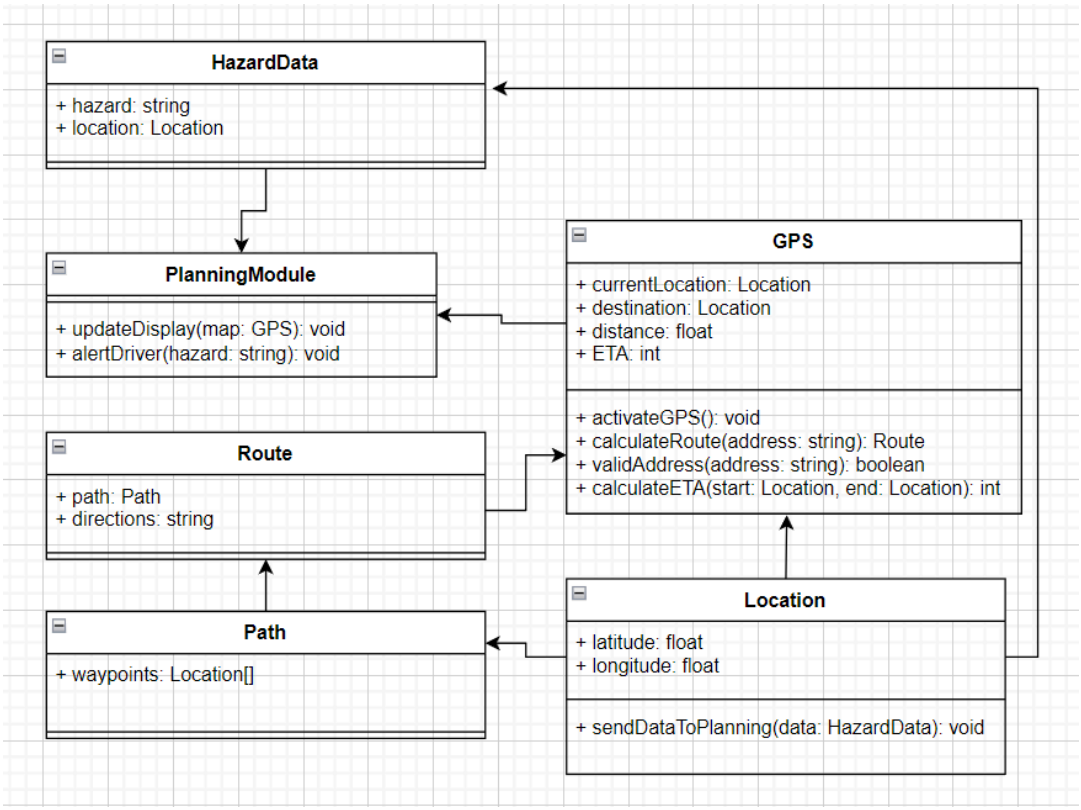
4.4.2 Software Update



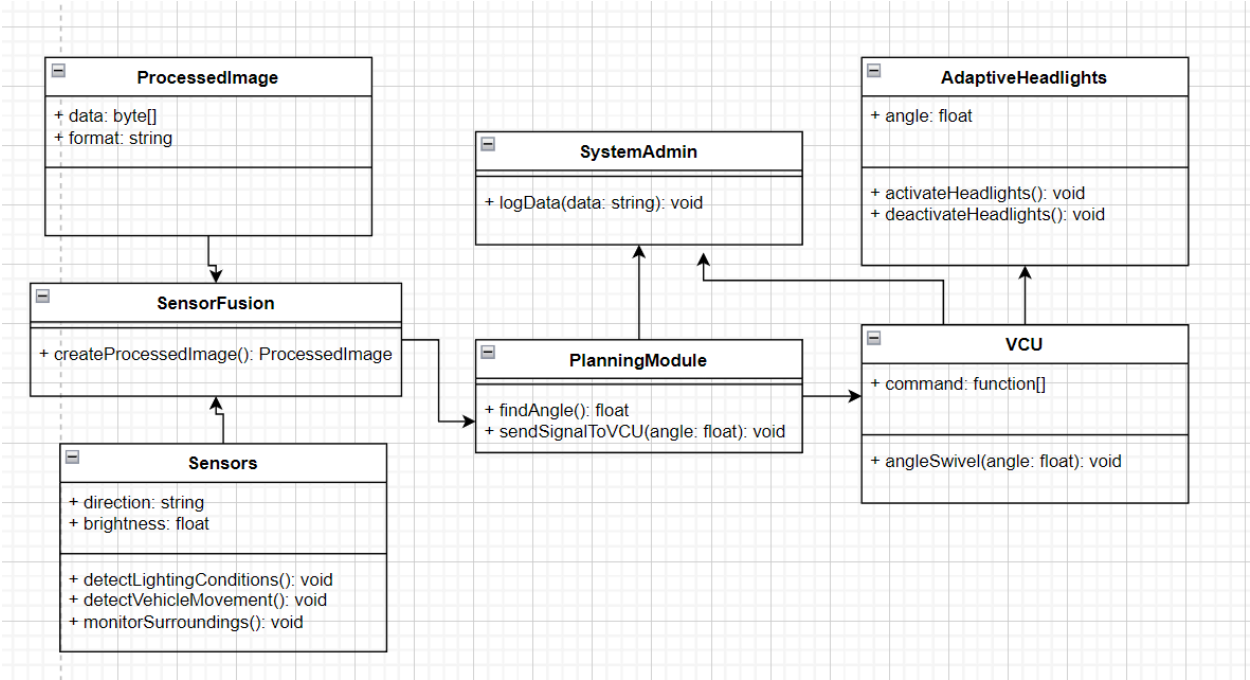
4.4.3 Automated Emergency Braking



4.4.4 GPS

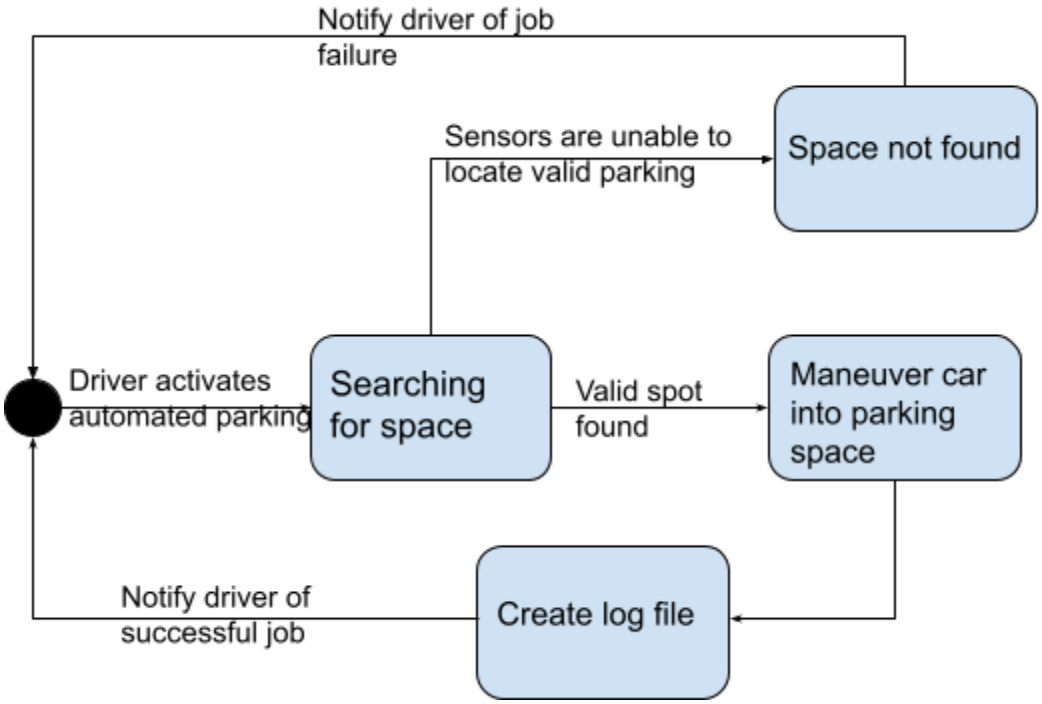


4.4.5 Adaptive Headlights

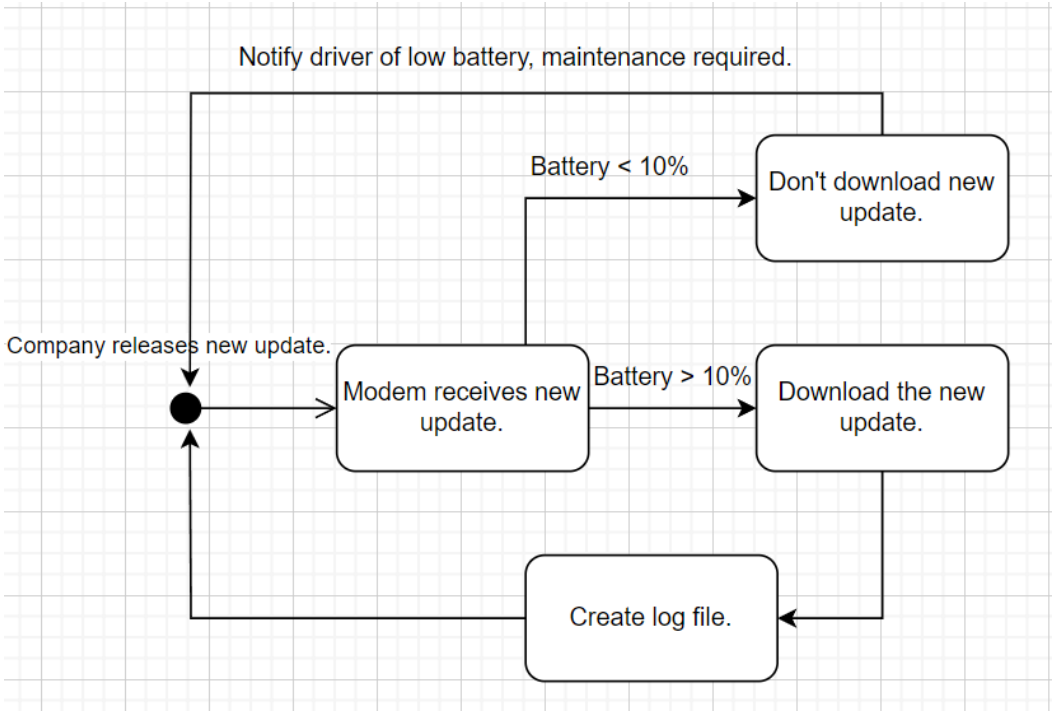


4.5 State Diagrams

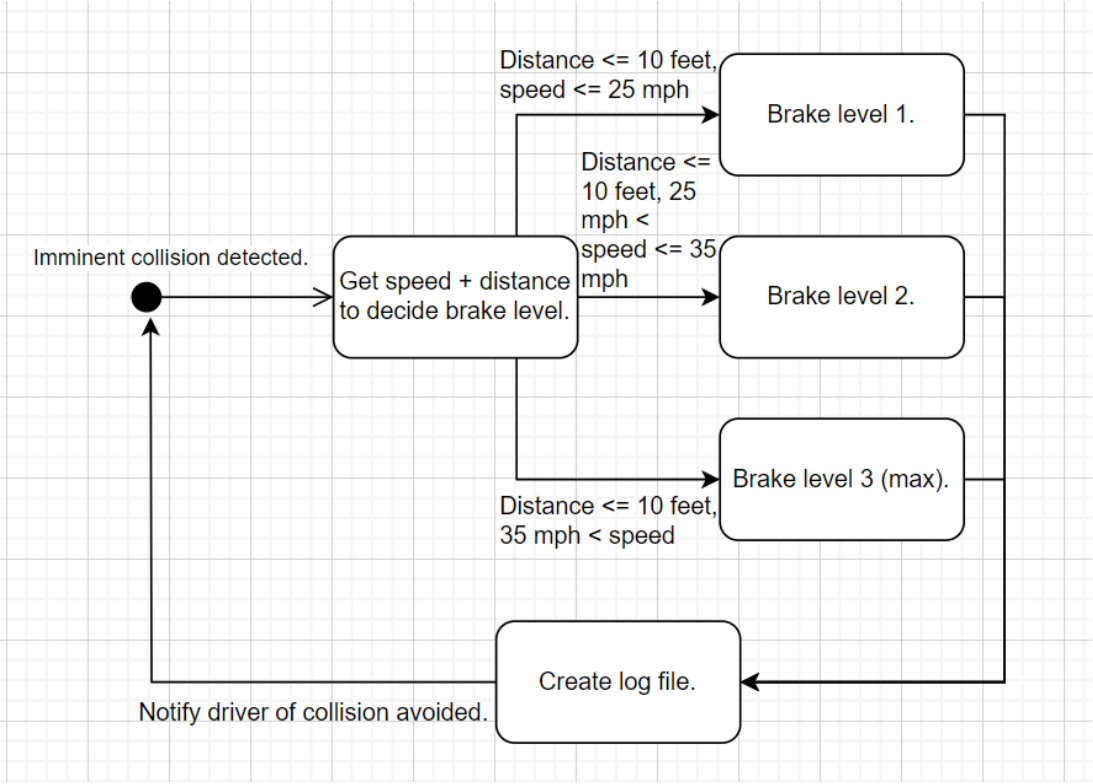
4.5.1 Driver Activate Automated Parking



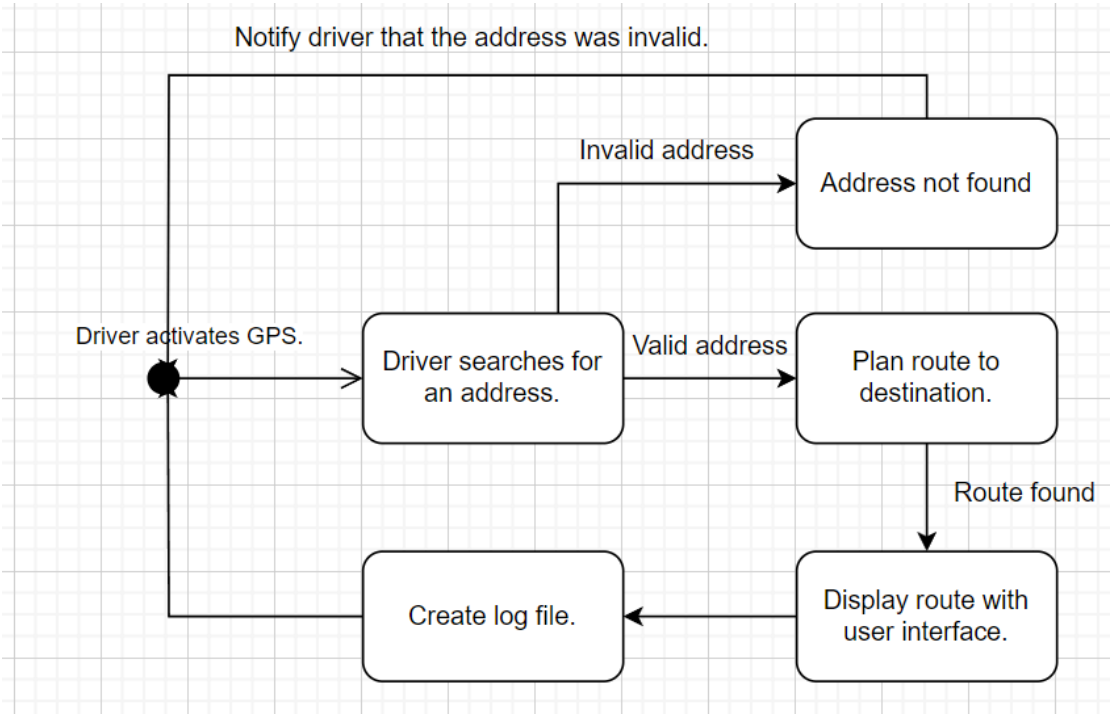
4.5.2 Software Update



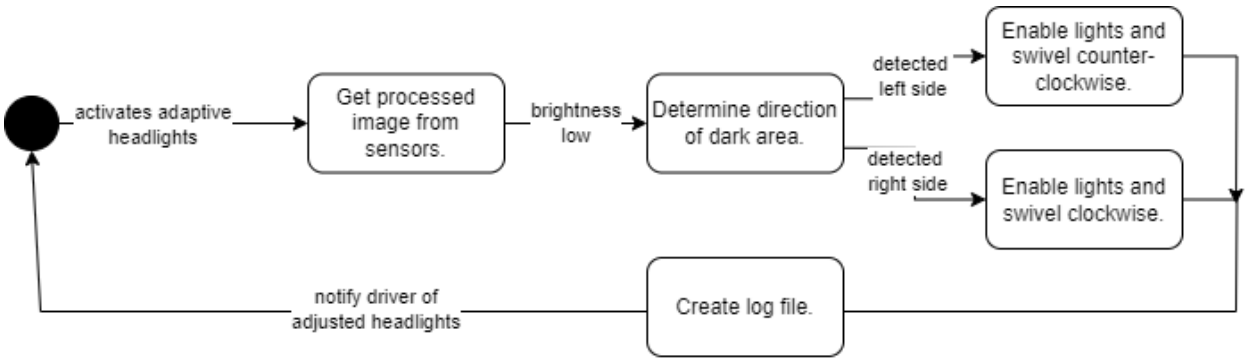
4.5.3 Automated Emergency Braking



4.5.4 GPS



4.5.5 Adaptive Headlights

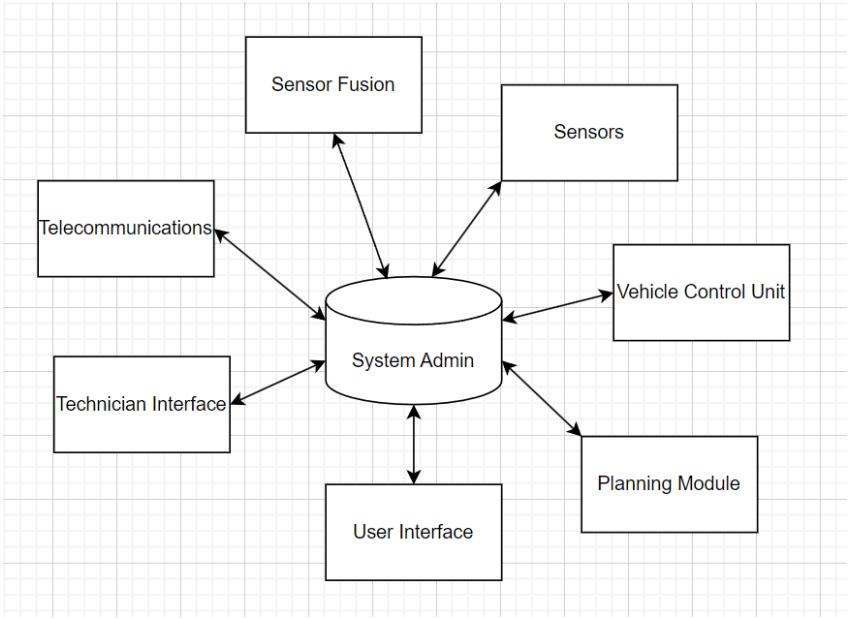


Section 5: Design

5.1 Software Architecture

When building large scale software, choosing an architecture is crucial for the organization and effectiveness of the software.

5.1.1 Data Centered Architecture



Description: This architectural style is built around a central data repository that facilitates communication between all components of the system.

Pros:

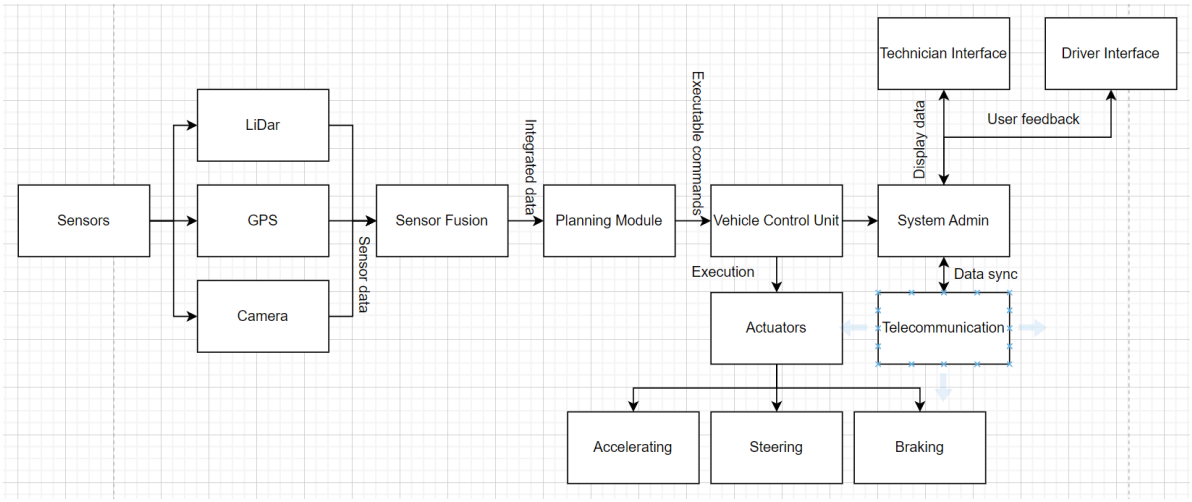
- Flexibility: Offers flexibility in integrating new sensors or modifying existing algorithms.
- Real-time decision making: By continuously analyzing incoming data streams, the AV can make real-time decisions.

Cons:

- Latency: Constantly retrieving data causes large amounts of latency.
- Dependency on central data repository: The entire system would stop working if the central data repository malfunctions.
- Complexity: Implementing a real-time system adds complexity to the design and development process.
- Resource Intensive: Requires significant computational resources and memory, which might be challenging for resource-constrained IoT devices.

Feasibility: For our IoT HTL, adopting a data-centric architecture could be viable due to its emphasis on managing, processing, and analyzing data from various sensors. However, given how complex and resource intensive it is, this would not be the best option for us.

5.1.2 Data Flow Architecture



Description: This architecture enables parallel processing of sensor data and controls logic by passing data through a series of processing stages. In this architecture, data flows through the system in a structured manner, undergoing various transformations and analyses at each stage.

Pros:

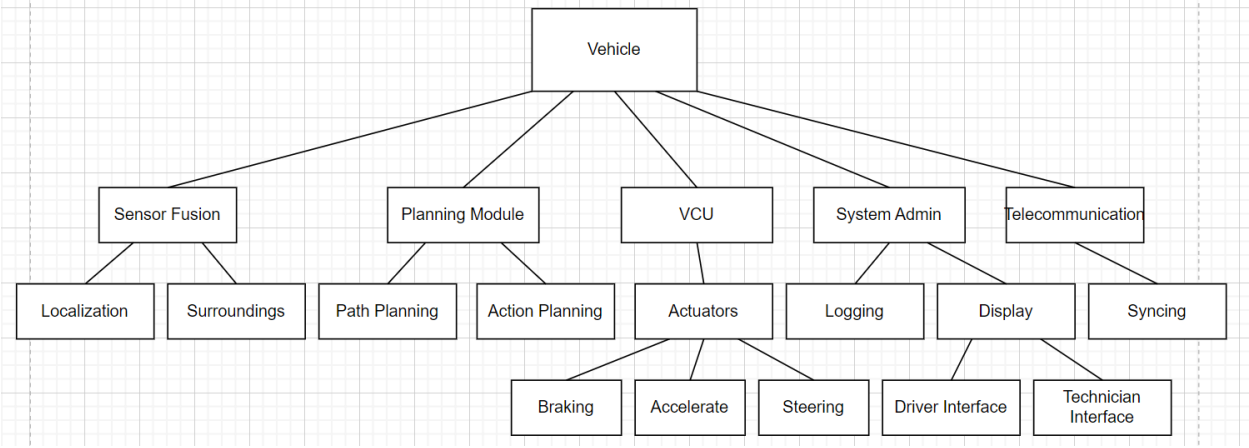
- Parallel Processing: Data Flow Architecture supports parallel processing of sensor data, allowing for efficient utilization of computing resources and improved system responsiveness.
- Flexibility: This architecture allows for integration of new sors or algorithms easily, making it adaptable to changing requirements in IoT HTL systems.
- Modularity: Modularizes the system into independent units for input processing and output generation.

Cons:

- Complexity: Designing and debugging data flow networks can be complex, especially in a system like ours that have numerous interconnected processing stages.
- Resource Intensive: This system's biggest advantage is parallel processing, but that requires significant computational resources, leading to higher power consumption and hardware costs.
- Latency: Processing delays between stages and communication overhead will introduce latency.
- Synchronization: Maintaining synchronization between different processing stages and ensuring data consistency can be challenging.

Feasibility: Overall, Data Flow Architecture offers significant advantages for IoT HTL systems with its parallel processing capabilities and flexibility. However, its complexity and resource requirements make it not ideal for our system. It also lacks the ability for real-time analysis due to factors such as processing speed and latency. Due to this, we will not be using Data Flow Architecture.

5.1.3 Call Return Architecture



Description: Call Return Architecture follows a synchronous execution model, where control is passed between modules through function calls and return statements. In this architecture, modules are organized hierarchically, with higher-level modules calling lower-level modules to perform specific tasks.

Pros:

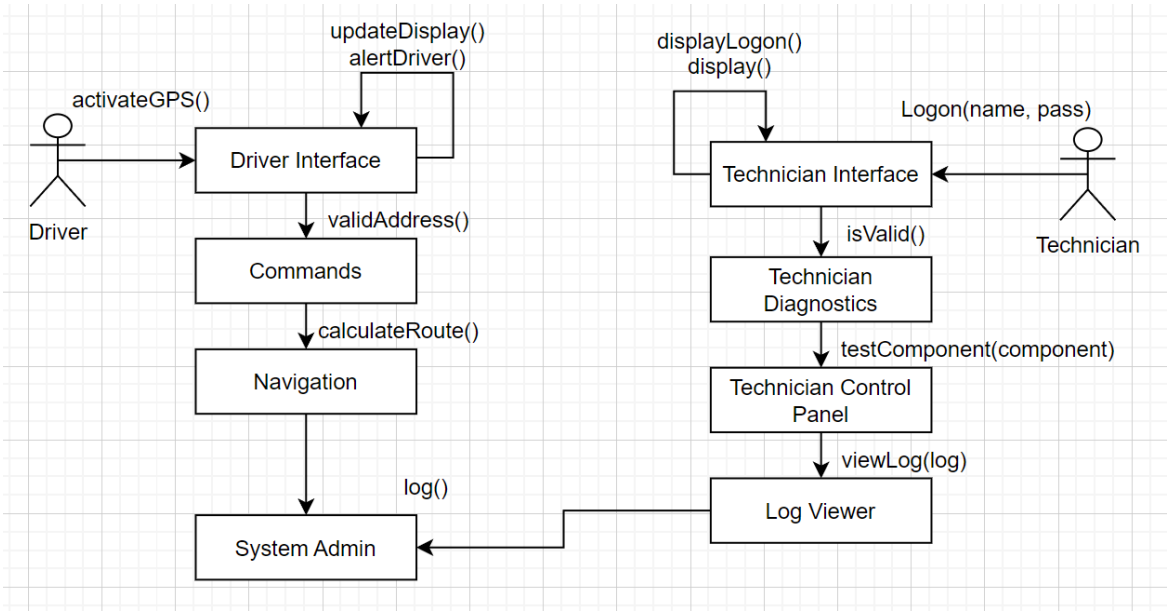
- Modularity: Modularizes the system, allowing for easier integration or removal of features.
- Simplified Flow Control: The synchronous execution model simplifies control flow, making it easier to understand and debug the system.

Cons:

- Limited Scalability: The synchronous nature of call-return interactions may limit scalability, especially in distributed IoT systems with a large number of devices and concurrent tasks.
- Latency: Need to wait for a function call to return to proceed, leading to significant latency.
- Real-time Responsiveness: The synchronous execution model will introduce delays in processing, hindering real-time responsiveness of the IoT HTL system.

Feasibility: This architecture is not a good fit for our purpose. Its lack of scalability and ability for real-time responsiveness are critical flaws that make this an unsuitable architecture.

5.1.4 Object-Oriented Architecture



Description: Object-Oriented Architecture organizes software components into objects, each having its own data and behaviors. This modular approach promotes code reusability, maintainability, and scalability by modeling real-world items as objects with properties and methods.

Pros:

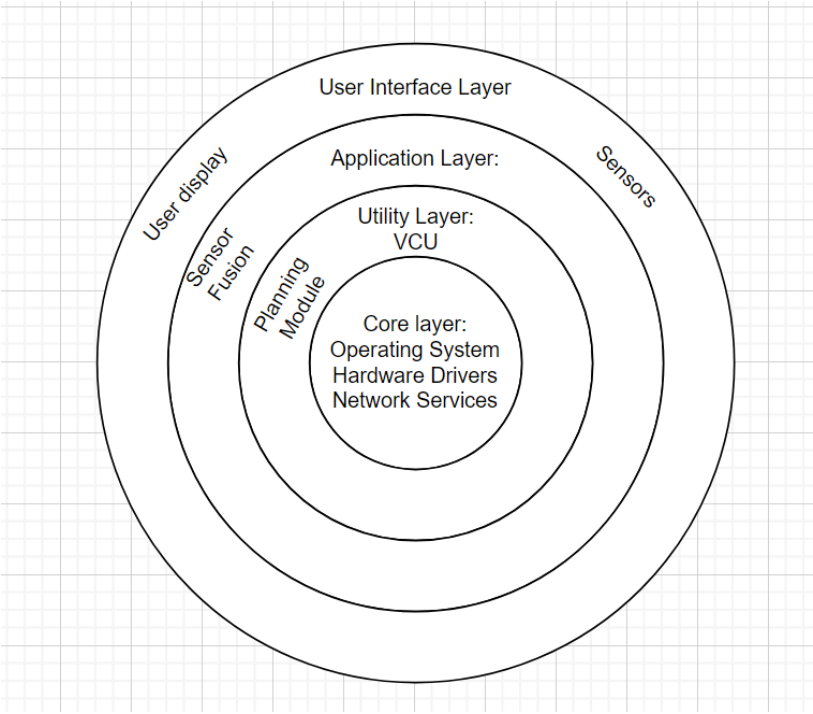
- Modularity: Objects have their own data and behaviors, promoting modularity and code reusability in IoT HTL systems.
- Flexibility: Allows for easy integration of new sensors or functionalities, making it adaptable to changing requirements.

Cons:

- Resource Intensive: Object-Oriented Architecture requires significant memory resources to store object instances and metadata.
- Complexity: IoT device objects and control logic are interconnected, resulting in complex relationships between objects and functions.
- Real-time Decision Making: Depends on the efficiency of algorithms and resource constraints.

Feasibility: Not the best choice to use for all of the IoT HTL system due to its real-time decision making being too dependent on the efficiency of algorithms in different classes.

5.1.5 Layered Architecture



Description: This architecture organizes the system into software layers, stacked from inside to outside - from user interface layer to internal core layer.

Pros:

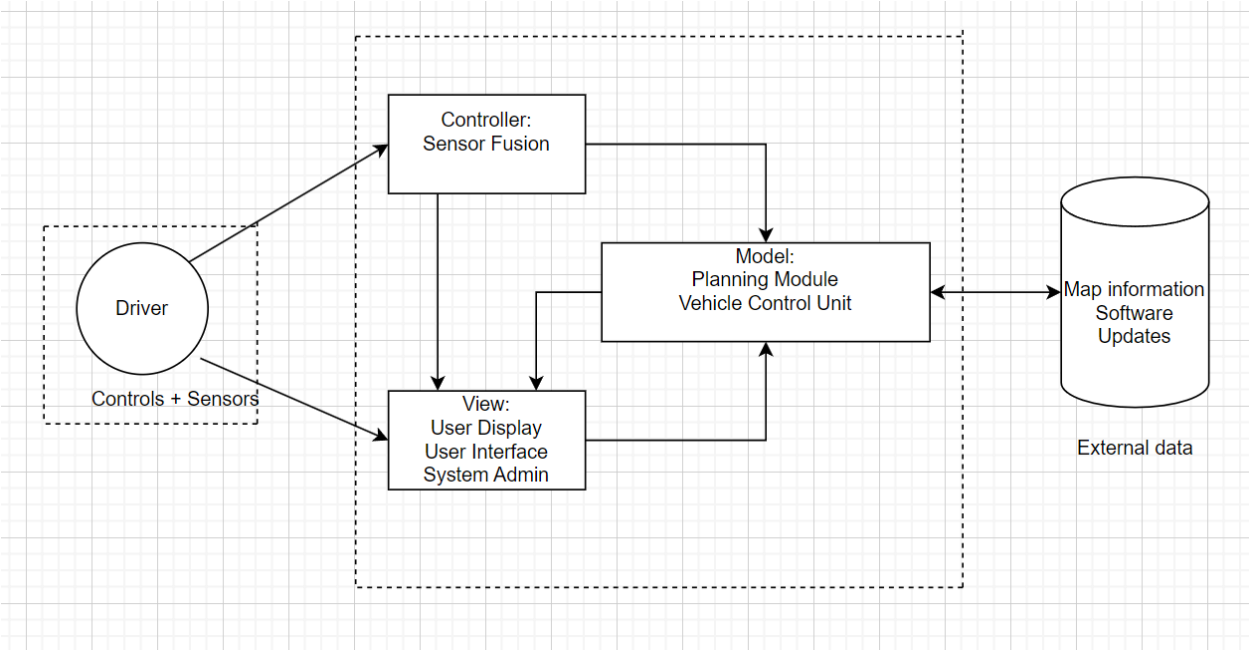
- Modularity: The clear separation of jobs into layers facilitates modular design and development, promoting code reusability in IoT HTL systems.
- Scalability: Layered Architecture supports scalability by allowing individual layers to be scaled independently, enabling the system to handle increasing data loads and processing requirements.
- Ease of Maintenance: The modular structure of Layered Architecture simplifies maintenance tasks, as changes can happen to specific layers without affecting the entire system.

Cons:

- Latency: In Layered Architecture, communication typically occurs between adjacent layers. This would cause latency, affecting the vehicle's response time to changing environments.
- Rigid Structure: Layered Architecture imposes a strict separation of duty, with each layer responsible for specific functionalities. This hinders flexibility in accommodating variations in system requirements encountered during driving.

Feasibility: Not the optimal architecture for an autonomous vehicle. This architecture's lack of real-time responsiveness is a critical flaw that we cannot accept. Therefore, we will not implement Layered Architecture for the IoT HTL system.

5.1.6 Model View Controller(MVC) Architecture



Description: This architecture divides the software into three distinct components: the controller, the model, and the view.

Pros:

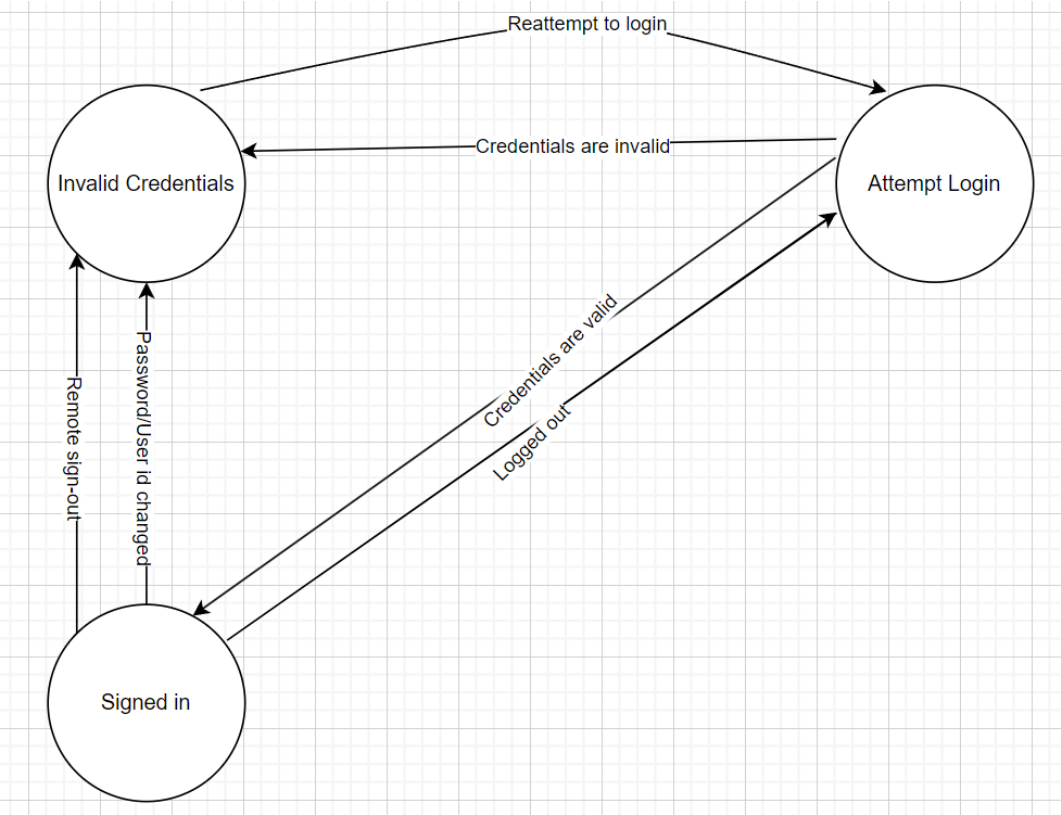
- Modularity: MVC Architecture separates the application into distinct components, making it easier to develop, test, and maintain.
- Separation of Duty: The clear separation of duties between the model, view, and controller components allows for focused development and optimization of each component independently.
- Scalability: MVC Architecture supports scalability by allowing the addition or replacement of components without affecting the entire system.
- External Data: Incorporates external data with model components.
- Real-time Decision Making: Has a structured framework for handling real-time decision making. Can effectively manage decision-making logic while ensuring adaptability to dynamic driving conditions because of the separation of duty.

Cons:

- Dependence between components: MVC Architecture results in high dependence between components, especially in systems with frequent updates to the user interface or complex data processing requirements.

Feasibility: Overall, MVC Architecture is our choice for our IoT HTL system because it provides a structured framework for handling the diverse requirements of autonomous vehicle applications. It also allows for constant input of sensor and driver data to the Model.

5.1.7 Finite State Machine(FSM) Architecture



Description: This architecture models system behavior as a finite set of states, transitions, and actions. Each state represents a specific mode or condition of the system, and transitions between states are triggered by events or conditions.

Pros:

- Real-time Decision Making: FSM Architecture inherently supports real-time decision making by enabling rapid state transitions in response to changing environments or sensor inputs.
- Predetermined Behavior: Precise control and decision making based on predefined states and transitions.
- Modularity: Decomposes system behavior into manageable states and transitions.

Cons:

- Complexity: Managing a large number of states and transitions is extremely complex. Careful design and documentation are necessary to ensure clarity and maintainability.
- Latency: FSM Architecture can only operate in one state at any given time, this results in latency if more than one state needs to be executed.

Feasibility: FSM Architecture can be viable but it is more suitable as architecture for our login system for users and technicians by using different states as different stages of authenticating login credentials.

5.2 Interface Design

5.2.1 Technician Interface

1. Dashboard Interface:
    - Class: Dashboard
    - Data: Sensor readings, alerts, system status
    - Description: The dashboard provides a real-time overview of the IoT HTL system's health. It displays key sensor readings, such as temperature, humidity, and pressure. Alerts for any anomalies or malfunctions are highlighted. The system status section indicates whether all components are functioning properly.
2. Control Panel Interface:
    - Class: ControlPanel
    - Data: Control commands, settings
    - Description: The control panel allows technicians to interact with the IoT HTL system. It provides controls for adjusting system settings, such as temperature thresholds and alert configurations. Technicians can also issue commands for manual interventions or system maintenance tasks.
3. Diagnostic Tools:
    - Class: DiagnosticTools
    - Data: test results, component status
    - Description: The diagnostic tools module allows technicians to perform diagnostic tests on various components of the vehicle.
4. Log Viewer:
    - Class: LogViewer
    - Data: System logs, warnings, notifications
    - Description: The interface will be able to display logs that were generated by the System Admin, warnings, and notifications for technicians to review. Provides a historical record of past events, aiding in identifying recurring issues and tracking system health over time.

5.2.2 Driver Interface

1. Navigation System Interface:
    - Class: Navigation
    - Data: Maps, route guidance
    - Description: The navigation system assists drivers in navigating their routes efficiently. It provides interactive maps that suggest optimal routes based on distance and/or tolls. Drivers can input their destination and receive turn-by-turn guidance throughout their journeys.
2. Vehicle Status Display Interface::
    - Class: StatusDisplay
    - Data: Vehicle health, fuel status, battery status
    - Description: The vehicle status display interface presents essential information about the vehicle's health and performance. It shows metrics such as fuel level,

battery status, and engine diagnostics. Any critical alerts or warnings related to vehicle maintenance are displayed to ensure driver safety.

3. Commands Interface:
    - Class: CommandsInterface
    - Data: driver instructions
    - Description: Enables drivers to communicate with the vehicle's onboard system using physical buttons/controls.

5.3 Component-level Design

In this section, the components of the autonomous vehicle system are transformed into procedural descriptions, detailing their functionalities, inputs, outputs, and interactions.

5.3.1 Sensors

- Functionality: Capture environmental data (objects, obstacles, road conditions) and vehicle's own condition (speed, location, temperature, battery level) to provide real-time perception of the vehicle.
- Inputs: Vehicle condition (speed, location, temperature, battery level), environmental stimuli (objects, obstacles, road conditions)
- Outputs: Raw sensor data streams containing information derived from inputs.
- Interactions: Interfaces with onboard sensor systems and vehicle subsystems for data acquisition and monitoring of the vehicle's condition.

5.3.2 Camera

- Functionality: Captures visual and spatial information about the vehicle's surroundings.
- Inputs: Visual stimuli from the environment.
- Output: Image data with depth maps/distance estimates for objects in the image.
- Interactions: Interfaces with sensor fusion module to integrate visual and spatial data into perception, providing insights into the vehicle's surroundings for navigation and obstacle detection.

5.3.3 LiDAR

- Functionality: Measures distances to objects by illuminating them with thin laser light and analyzing the reflected pulses.
- Inputs: Spatial information from laser reflections.
- Outputs: Data representing the environment's geometry.
- Interactions: Interfaces with sensor fusion module for integrating LiDAR data into perception.

5.3.4 GPS

- Functionality: Determines the vehicle's global position by receiving signals from satellites.
- Inputs: Signals from satellites.

- Output: Geospatial coordinates (latitude and longitude).
- Interactions: Interfaces with localization module to incorporate GPS data into position estimation.

5.3.5 Sensor Fusion

- Functionality: Integrates data from multiple sources of input (Sensors, Camera, LiDAR, GPS) to generate a comprehensive image of the vehicle's surroundings.
- Inputs: Raw data streams.
- Output: Fused sensor data, including object detection, localization, and environmental mapping.
- Interactions: Fuses data streams to generate an image that is given to the Planning Module to decide on further actions.

5.3.6 Planning Module

- Functionality: Generates signals that are sent to the Vehicle Control Unit to execute tasks.
- Inputs: Fused sensor data, map data, user input
- Outputs: Planned actions, log file input, waypoint sequences
- Interactions: Interfaces with sensor fusion, mapping, and control modules for data exchange.

5.3.7 Vehicle Control Unit

- Functionality: Executes actions decided on by the Planning Module.
- Inputs: Planned actions
- Outputs: Actuator commands for steering, braking, and accelerating, log file input
- Interactions: Interfaces with the planning module for directions.

5.3.8 System Admin

- Functionality: Manages system configuration, monitoring, log files, and diagnostics.
- Inputs: User commands, system status updates
- Outputs: Visual displays, audio cues, diagnostic reports
- Interactions: Interfaces with user interfaces, sensor fusion, and planning module for user interaction and system monitoring. Interfaces with the planning module and vehicle control unit to create log files.

5.3.9 On-board Modem

- Functionality: Provides communication capabilities for the vehicle, enabling OTA software updates.
- Inputs: Data packets for software updates.
- Output: Data packets for transmission to external system, log file
- Interactions: Interfaces with System Admin to log when the software update was completed. Sends data packets to the external system to notify the distributor that the update is complete.

Section 6: Coding

6.1 Function Codes

6.1.1 Object Avoidance

```
class ObjectAvoidance:
    def __init__(self):
        pass

    def CalculateBrakingLevel(distance: int, speed: int) -> int:
         return FuncToCalculateBreakingLevel(distance, speed)

    def SendSignalToVCU(brakingLevel: int):
        VCU.executeBrake(brakingLevel)

class VCU:
    def executeBrake(brakingLevel):
        brakes = True
        brakeLevel = AutomatedBraking.brakingLevel

class PlanningModule:
    brakingLevel = 0

    def detectImminentCollision(data: SensorFusion):
        if(data == imminentCollision):
            level = AutomatedBraking.CalculateBrakingLevel(data.distance,
data.speed)
            AutomatedBraking.VCUBrake(level)

class SensorFusion:
    def sendDataToPlanningModule(data: Sensors):
        return processSensorData(data)

class Sensors:
    distanceFromFron = 0
    vehicleSpeed = 0

    def getDistance():
        return distance

    def getSpeed():
```

```
        return speed

class Visual:
    message = ""

    def __init__(self) -> None:
        self.message = ""

    def displayMessage(message: str) -> None:
        self.message = message

class Audio:

    def __init__(self) -> None:
        pass

    def playSound(sound: str) -> None:
        audioDevice = getAudioDevice()
        audioDevice.play(sound)
```

6.1.2 Driver Activate Cruise Control

```
class CruiseControl:
    def __init__(self):
        pass

    def SendSignalToVCU(speed: int):
        VCU.SetSpeed(speed)

class VCU:
    def SetSpeed(speed: int) -> bool:
        return FuncToSetSpeedOfVehicle(speed)

class SensorFusion:
    def sendDataToPlanningModule(data: Sensors):
        return processSensorData(data)

class Sensors:
    vehicleSpeed = Vehicle.speed

    def getSpeed():
```

```
        return vehicleSpeed

    def ActivateCruiseControl():
        return True
```

6.1.3 Automated Emergency Braking

```
class AutomatedBraking:
    def __init__(self):
        pass

    def CalculateBrakingLevel(distance: int, speed: int) -> int:
        return FuncToCalculateBreakingLevel(distance, speed)

    def SendSignalToVCU(brakingLevel: int):
        VCU.executeBrake(brakingLevel)

class VCU:
    def executeBrake(brakingLevel):
        brakes = True
        brakeLevel = AutomatedBraking.brakingLevel

class PlanningModule:
    brakingLevel = 0

    def detectImminentCollision(data: SensorFusion):
        if(data == imminentCollision):
            level = AutomatedBraking.CalculateBrakingLevel(data.distance, data.speed)
            AutomatedBraking.VCUBrake(level)

class SensorFusion:
    def sendDataToPlanningModule(data: Sensors):
        return processSensorData(data)

class Sensors:
    distanceFromFron = 0
    vehicleSpeed = 0

    def getDistance():
        return distance

    def getSpeed():
```

```
    return speed
```

6.1.4 Lane Keeping Assist (LKA)

```
class LaneKeepingAssist:
    def __init__(self):
        pass

    def CalculateAngleChange(distance: int, speed: int) -> float:
         return FuncToCalculateAngleNeeded(distance, speed)

    def SendSignalToVCU(angle: float):
        VCU.changeAngle(angle)

class VCU:
    def changeAngle(angle):
        Vehicle.angle = angle

class PlanningModule:
    angle = Vehicle.angle

    def detectLaneMarkings(data: SensorFusion):
        if(data == nearMarkings):
            angle = AutomatedBraking.CalculateAngleChange(data.distance,
data.speed)
            LaneKeepingAssist.SendSignalToVCU(angle)

class SensorFusion:
    def sendDataToPlanningModule(data: Sensors):
        return processSensorData(data)

class Sensors:
    distanceFromLeftMarking = 0
    distanceFromRightMarking = 0
    vehicleSpeed = 0

    def getDistance(side: int):
        return distance

    def getSpeed():
        return speed
```

```
class Visual:
    message = ""

    def __init__(self) -> None:
        self.message = ""

    def displayMessage(message: str) -> None:
        self.message = message

class Audio:

    def __init__(self) -> None:
        pass

    def playSound(sound: str) -> None:
        audioDevice = getAudioDevice()
        audioDevice.play(sound)
```

6.1.5 Automated Pathfinding

```
from typing import List

class Location:
    latitude, longitude = 0.0, 0.0

    # constructor
    def __init__(self, latitude: float = 0.0, longitude: float = 0.0) ->
None:
        self.latitude = latitude
        self.longitude = longitude

class Path:
    waypoints = []

    # constructor
    def __init__(self, waypoints: List[Location] = []) -> None:
        self.waypoints = waypoints

class Route:
    path = Path()
    directions = ""
```

```
    # constructor
    def __init__(self, path: Path = Path(), directions: str = "") -> None:
        self.path = path
        self.directions = directions

class PlanningModule:
    # constructor
    def __init__(self) -> None:
        pass

    # methods
    updateDisplay(map: AutomatedPathfinding) -> None:
        displayStart(AutomatedPathfinding.currentLocation)
        displayEnd(AutomatedPathfinding.destination)

    alertDriver(hazard: str) -> None:
        alert(hazard)

class AutomatedPathfinding:
    currentLocation, destination = Location(), Location()
    distance = 0.0

    # constructor
    def __init__(self, currentLocation: Location = Location()) -> None:
        self.currentLocation = currentLocation

    # methods
    def calculateFastestRoute(address: str) -> Route:
        self.destination = address
        directions = getDirections(self.currentLocation, address)
        path = Path(getWaypoints(self.currentLocation, address))
        return Route(path, directions)

    def validAddress(address: str) -> bool:
        return API.isAddress(address)
```

6.1.6 GPS

```
from typing import List

class Location:
```

```
    latitude, longitude = 0.0, 0.0

    # constructor
    def __init__(self, latitude: float = 0.0, longitude: float = 0.0) -> None:
        self.latitude = latitude
        self.longitude = longitude

    # methods
    sendDataToPlanning(self, data: HazardData) -> None:
        data.location = self

class Path:
    waypoints = []

    # constructor
    def __init__(self, waypoints: List[Location] = []) -> None:
        self.waypoints = waypoints

class Route:
    path = Path()
    directions = ""

    # constructor
    def __init__(self, path: Path = Path(), directions: str = "") -> None:
        self.path = path
        self.directions = directions

class HazardData:
    hazard = ""
    location = Location()

    # constructor
    def __init__(self, hazard: str = "", location: Location = Location()) -> None:
        self.hazard = hazard
        self.location = location

class PlanningModule:
    # constructor
    def __init__(self) -> None:
        pass

    # methods
    updateDisplay(map: GPS) -> None:
```

```
        displayStart(GPS.currentLocation)
        displayEnd(GPS.destination)

    alertDriver(hazard: str) -> None:
        alert(hazard)

class GPS:
    currentLocation, destination = Location(), Location()
    distance = 0.0
    ETA = 0

    # constructor
    def __init__(self, currentLocation: Location = Location()) -> None:
        self.currentLocation = currentLocation

    # methods
    def activateGPS() -> None:
        turnOn()

    def calculateRoute(address: str) -> Route:
        self.destination = address
        directions = getDirections(self.currentLocation, address)
        path = Path(getWaypoints(self.currentLocation, address))
        return Route(path, directions)

    def validAddress(address: str) -> bool:
        return API.isAddress(address)

    def calculateETA(start: Location, end: Location) -> int:
        distance = Math.distance(start, end)
        self.distance = distance

        ETA = distance // getAvgSpeed()
        self.ETA = ETA
        return ETA
```

6.1.7 Automated Parking

```
class AutomatedParking:
    def __init__(self):
        pass
```

```
    def SendSignalToVCU(steps: array):
        VCU.executePark(steps)

class VCU:
    def executePark(steps):
        steps = AutomatedParking.steps

class PlanningModule:
    steps = []

    def CalculateParkingManeuver(data: SensorFusion):
        if (data == validParking):
            steps = AutomatedParking.CalculateParking(data.distance,
data.speed, data.spot)
            AutomatedParking.park(steps)

class SensorFusion:
    def sendDataToPlanningModule(data: Sensors):
        return processSensorData(data)

class Sensors:
    distanceFromSpot = 0
    vehicleSpeed = 0

    def getDistance():
        return distance

    def getSpeed():
        return speed

    def findSpot():
        return spot
```

6.1.8 Blind Spot Detection

```
class BlindSpotDetection:
    beepSpeed = 0

    def __init__(self, beepSpeed):
        self.beepSpeed = beepSpeed
        Display.showBackupCam()
```

```
    def calcBeepSpeed():
        distanceToObject = Sensors.getDistanceToObjectRear()
        beepSpeed = funcToCalcBeepSpeed(distanceToObject)

    def sendSignalToAudioController():
        return Audio.backupBeep(beepSpeed)

class Audio:
    def __init__(self) -> None:
        pass

    def backupBeep(beepSpeed):
        beep = beepSpeed

class Sensors:
    leftBlindSpot = False
    rightBlindSpot = False

    def detectBlindSpot():
        return detected

class Display:
    def blindSpotDetected(side: int):
        LightUp(side)
```

6.1.9 Backup Camera and Detection

```
class BackupCamera:
  beepSpeed = 0

  def __init__(self, beepSpeed):
    self.beepSpeed = beepSpeed
    Display.showBackupCam()

  def calcBeepSpeed():
    distanceToObject = Sensors.getDistanceToObjectRear()
    beepSpeed = funcToCalcBeepSpeed(distanceToObject)

  def sendSignalToAudioController():
    return Audio.backupBeep(beepSpeed)

class Audio:
```

```
    def __init__(self) -> None:
        pass

    def backupBeep(beepSpeed):
        beep = beepSpeed

class Sensors:
    closestDistanceToObject = 0

    def getDistanceToObjectRear():
        return closestDistanceToObject

class Display:
    def showBackupCam():
        ClearDisplay()
        ShowBackupCamera()

class VCU:
    if(gear == reverse):
        BackupCamera.__init__
```

6.1.10 Adaptive Headlights

```
class AdaptiveHeadlights:
    headlightAngle = 0

    def __init__(self, headlightAngle):
        self.headlightAngle = headlightAngle

    def calcHeadlightAngle(steeringWheelAngle):
        headlightAngle = funcToCalc(steeringWheelAngle)

    def sendVCUSignal():
        VCU.moveHeadlights = headlightAngle

class PlanningModule:
    def headlights():
        AdaptiveHeadlights.calcHeadLightAngle(Sensors.steeringWheelAngle)
        AdaptiveHeadlights.sendVCUSignle()

class Sensors:
    steeringWheelAngle = steeringSensor.steeringAngle
```

```
class VCU:
   def moveHeadlights(angle):
      headlightAngle = angle
```

6.1.11 Turn Car On

```
class TurnCarOn:
    def __init__(self, keyPresent):
        self.keyPresent = keyPresent
        Display.showOn()

    def sendSignalToDisplay():
        return Display.displayMessage("Vehicle turning on")

class Display:
    def __init__(self) -> None:
        pass

    def displayMessage(message: string):
        display(message)

class Sensors:
    keyInCar = false

    def detectKey():
        if (keyPresent):
            return True
        else:
            return False

class VCU:
    engineOn

    def startEngine():
        engineOn = True
```

6.2 Non-Function Codes

6.2.1 Software Update

```
class Visual:
    message = ""

    # constructor
    def __init__(self) -> None:
        self.message = ""

    # methods
    def displayMessage(message: str) -> None:
        self.message = message

class Audio:
    # constructor
    def __init__(self) -> None:
        pass

    # methods
    def playSound(sound: str) -> None:
        audioDevice = getAudioDevice()
        audioDevice.play(sound)

class Driver:
    audio = Audio()
    visual = Visual()

    # constructor
    def __init__(self, audio, visual) -> None:
        self.audio = audio
        self.visual = visual

class Battery:
    percentage = 0.0

    # constructor
    def __init__(self, percentage: float = 0.0) -> None:
        self.percentage = percentage

    # methods
    def checkBattery() -> float:
        return percentage
```

```
class Modem:
    connectivity = 0
    functional = False

    # constructor
    def __init__(self, connectivity: int = 0, functional: bool = False) -> None:
        self.connectivity = connectivity
        self.functional = functional

    # methods
    def checkConnectivity() -> int:
        return connectivity

    def checkFunctional() -> bool:
        return functional

class SoftwareUpdate:
    version = "0.0.0"
    latest = "0.0.0"

    # constructor
    def __init__(self, version: str = "0.0.0", latest: str = "0.0.0") -> None:
        self.version = version
        self.latest = latest

    # methods
    def downloadUpdate() -> None:
        update = Cloud.fetchLatestUpdate()
        version = update.version

        self.version = version
        self.latest = version

        download(update)
```