Version 1.7

Alset - Project Ohm

Team 12

Haig Emirzian, Christos Zervas, Michael Buzzetta, Nicolas Buendia

“Overcoming resistance to drive autonomously.”

Table of Contents

Section 4: Requirement Modeling

7.2 Drive Test

7.3 Automatic Parking Test

7.4 Acceleration Launch Test

7.5 Headlights Test

7.6 Obstacle Avoidance Test

7.7 Traffic Sign Detection Test

7.8 Navigation Test

7.9 Crash Detection Test

7.10 Cruise Control Test

Section 1: Introduction

1.1 Introduction

The means of this project are to research and figure out a path to human assistance driving and Level-5 autonomous driving; albeit, not the most efficient path. In this groundbreaking venture, our collaborative efforts in developing the self-driving car of the future are nothing short of revolutionary. As of today, there simply doesn't exist a Level-5, consumer-grade vehicle - a vehicle that can operate fully autonomously, to the point where a steering wheel and pedals are not necessary.

1.2 Scope of the Project

Developing a fully autonomous vehicle is a complex process that involves several technical, regulatory, ethical, and societal considerations. Although the driver can take over at any time, it is important to create a product that does not have the driver hovering over the brake pedal when in use. Naturally, transportation of any kind is highly regulated, and it would be wise to follow laws and policies set by each jurisdiction. With compliance in mind, restrictions yield the way for innovation.

Furthermore, this team is driven to protect users by considering ethical dilemmas and security and privacy concerns when developing the software. However, it is essential to note that the initial release is just the beginning. The team envisions an incremental approach where subsequent releases will build upon the foundation laid by the first version. Each release will introduce new features, improvements, and refinements based on user feedback and evolving requirements. By embracing this iterative and incremental process, the team aims to create a robust and adaptable autonomous vehicle system.

1.3 Features

Project Ohm utilizes a myriad of features that set it apart in the autonomous driving world. From advanced sensor arrays for real-time environment perception to machine learning algorithms for quick and effective decision-making, our vehicle is equipped to handle diverse driving scenarios that we face in this current world. Adaptive cruise control, automatic lane-changing, and comprehensive collision avoidance mechanisms are just a glimpse into the capabilities of Project Ohm.

1.4 Insight Into Some Features

Delving deeper, our vehicle offers a glimpse into the new future of transportation. Augmented reality interfaces, predictive maintenance protocols, and advanced communication systems exemplify the innovative features that enhance user experience and safety. These features not only showcase our commitment to cutting-edge technology but also emphasize our dedication to building a holistic and reliable autonomous driving solution.

1.5 Nature of the project

Project Ohm is not merely a technological feat; it stands as a mission-critical real-time embedded system. The nature of our endeavor requires the highest levels of availability and reliability. As we dive into the intricacies of developing an autonomous driving system, we acknowledge the profound impact our work will have on the safety and well-being of the targeted users. Our commitment to creating a software system that operates seamlessly in real time is fundamental to ensuring the success and widespread adoption of autonomous vehicles.

1.6 Relying on IoT

In our pursuit of excellence, we recognize the significance of leveraging IoT architecture. The integration of IoT technologies enhances connectivity, allowing our autonomous vehicles to interact with the surrounding environment, other vehicles, and infrastructure seamlessly. The continuous exchange of data in real-time not only optimizes the vehicle's overall performance but also contributes to the collective intelligence of the transportation ecosystem. Project Ohm's reliance on IoT is a testament to our commitment to staying at the forefront of technological advancements.

1.7 Following a Software Development Process

A key pillar of our approach is the adherence to a stabilized software development process. The complexity of developing autonomous driving software demands a systematic approach: the waterfall model. We follow a structured methodology that encompasses requirements analysis, design, implementation, testing, and continuous iteration. Regular reviews and audits are embedded into this methodology to ensure the integrity and coherence of our codebase, as documentation serves as a vital communication not only between team members but anyone else who wishes to implement our process at any given time. Meetings with team members are regularly conducted to discuss progress, address challenges, and strategize a plan to move forward. By embracing discipline and the best practices in software development, we ensure the reliability, scalability, and maintainability of Project Ohm's software architecture.

1.8 Iterative Waterfall Process

1.8.1. Requirements Gathering: The iterative waterfall process for a self-driving car project commences with meticulous requirements gathering, where overarching functionalities and safety standards are defined and prioritized.

1.8.2. System Design: This transitions into the intricate system design phase, where the architecture is meticulously crafted, delineating hardware components like sensors and processors, and software modules for perception and decision-making.

1.8.3. Implementation: Implementation takes center stage, focusing on the initial iteration of the system, honing in on core functionalities like sensor data processing and basic driving behaviors.

1.8.4. Testing: Rigorous testing ensues, encompassing unit and integration testing to ensure module integrity and system coherence.

1.8.5. Deployment: Following successful testing, deployment unfolds, launching the nascent system into a controlled environment for real world experimentation and feedback gathering.

1.8.6. Evaluation: Evaluation becomes paramount, scrutinizing performance against predefined metrics, while iterative development refines and expands the system based on insights gleaned.

1.8.7. Validation and Verification: Continuous validation and verification underpin the process, ensuring compliance and reliability throughout.

1.8.8. Release: Finally, releases punctuate the journey, marking milestones of progress and evolution.

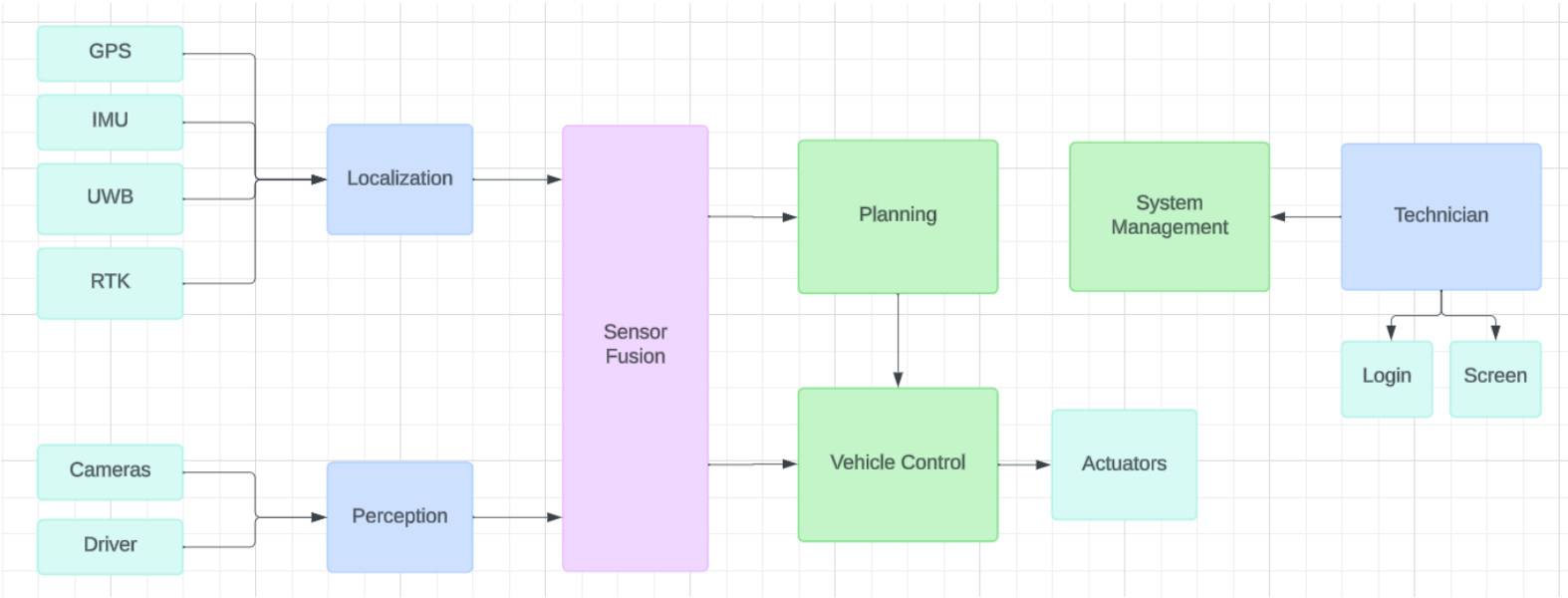
1.8.9. Maintenance and Support: Ongoing maintenance and support sustain the system's integrity and functionality over time, completing the iterative waterfall process for the self-driving car project.

1.9 Team and Qualifications

Our team - Haig Emirzian, Christos Zervas, Michael Buzzetta, and Nicolas Buendi - is composed of seasoned professionals with diverse backgrounds in technology. The synergy of our skills forms the backbone of Project Ohm. Half of our team is made up of those with a Computer Science background, while the other half is made up of those with a Cybersecurity background. With this composition, we are projected to succeed, since we all have a strong foundation to build this product and can be security-focused from day 1. Our three major qualifications not only match but differ as well. This sets us apart from teams that might be homogenous. We have a team composed of like minded individuals who possess coding, project management, and problem solving skills. On the other hand, we have individuals like Christos who stand out by having communication as one of his major qualifications, a skill no one else decided to put as their top choices. Michael stands out because of his innate curiosity to simplify the problem. More technically speaking, we all are well versed in Python which will act as our primary programming language. If for any reason we need more management of memory, we can always adapt to C or C++ as an alternative. Different minds can come up with different solutions, which was the goal when recruiting this team.

Section 2: Software Architecture

2.1 Functional Architecture



2.2 Functionality

2.2.1 Localization: Integrating GPS and IMU

In the context of localization, Project Ohm integrates GPS (Global Positioning System) and IMU (Inertial Measurement Unit) modules to establish precise positioning. The GPS module receives signals from satellites, determining the vehicle's position, velocity, and time. Simultaneously, the IMU module measures acceleration and angular rate, providing data on orientation and motion. This data integration is facilitated through the Localization Module, which acts as a central hub for processing and fusing information from both GPS and IMU. The resultant accurate position and orientation data are essential for the vehicle's navigation tasks.

Expanding on hardware requirements, this integration necessitates advanced GPS receivers capable of high-precision positioning. Additionally, the IMU hardware must be equipped to capture precise acceleration and angular rate data. The synchronization and

coordination between these hardware components are crucial for successful sensor fusion and localization accuracy

2.2.2 UWB and RTK Technologies for Localization Enhancement

Project Ohm employs UWB (Ultra-Wideband) and RTK (Real-Time Kinematics) technologies to enhance localization accuracy. UWB facilitates high-precision ranging by utilizing a wide spectrum of frequencies, providing accurate distance measurements between the vehicle and its surroundings. RTK enhances precision by providing centimeter-level accuracy to the position data obtained from GPS receivers. This combination ensures the vehicle operates with an unprecedented level of precision, crucial for navigating complex environments and executing precise maneuvers.

2.2.3 Perception: Vision-Based Systems with Cameras

Project Ohm relies on a vision-centric approach for perception, utilizing cameras as versatile sensors. The vehicle incorporates 8 strategically positioned cameras, capturing rich visual data including color, texture, and depth perception. These cameras facilitate a vision-only approach, aligning with human-like perception for driving. Despite the vehicle's autonomous capabilities, user input remains imperative for safety, creating a checks-and-balances system.

2.2.4 Sensor Fusion

Sensor Fusion is a critical component bridging Perception and Localization functionalities. In the pursuit of Level 5 autonomy, Project Ohm employs a multi-sensor approach, including vision-based systems, cameras, LiDAR, radar, UWB, and RTK. Sensor Fusion integrates and reconciles data from these diverse sensors, creating a cohesive representation of the environment. This approach optimizes sensor redundancy, compensating for

potential limitations in individual sensors and enhancing the system's robustness and reliability. The fused information is crucial for Planning and Vehicle Control.

2.2.4.1 Sensor Fusion Optimization

Sensor Fusion plays a pivotal role in optimizing sensor redundancy within Project Ohm's architecture. By fusing data from multiple sensors, the system compensates for potential limitations or blind spots in individual sensors. For instance, adverse weather conditions affecting one sensor's performance can be compensated by others. This redundancy enhances the robustness and reliability of the perception system, crucial for safe autonomous driving.

2.2.4.2 System Management in Sensor Fusion

The System Management component in Sensor Fusion oversees continuous monitoring and feedback loops, ensuring the integrity of sensor data. Real-time assessment of sensor performance allows for the identification of anomalies, triggering corrective actions and maintaining the reliability and consistency of the Sensor Fusion process. The System Management's role is crucial for ensuring that the perception system operates optimally under various conditions.

2.2.5 Planning and Vehicle Control

The Planning and Vehicle Control functionality rely on essential information gathered by the Sensor Fusion component. This integrated data, derived from both Localization and Perception sensors, provides comprehensive insights into the vehicle's surroundings. These insights influence decisions on the car's next move, timing, and speed, directly impacting the vehicle control component. The vehicle control component dictates actions such as steering adjustments, braking intensity, and acceleration duration to execute maneuvers effectively.

2.2.6 System Management for Regular Maintenance

While Project Ohm's architecture may seem simpler than other designs, it emphasizes the importance of technicians who can fine-tune configurations. Regular maintenance for both localization and perception modules ensures optimal vehicle performance. In system administration, administrators assess data, identify issues, and make informed decisions about architecture adjustments. For example, if reconfiguring cameras through the UI does not work, bringing the vehicle to a service center allows a qualified technician to enter the software through "Service Mode" and gather more information for diagnostics.

2.3 Requirements and Contract Section

In this section, we outline the specific requirements and contractual aspects of Project Ohm's autonomous driving system.

2.3.1 Adaptive Cruise Control Implementation

The implementation of Adaptive Cruise Control (ACC) within Project Ohm's architecture aligns with the system's contract. ACC utilizes Sensor Fusion, combining data from Localization and Perception sensors to maintain a safe distance from the vehicle ahead during cruising. The continuous data exchange, monitored by the System Management component, ensures real-time optimization of ACC performance. Cloud connectivity is crucial for remote monitoring and updates, keeping ACC aligned with the latest algorithms and regulations.

2.3.2 Autonomous Lane Changing Feature

The implementation of the Autonomous Lane Changing (ALC) feature aligns with Project Ohm's contractual commitments. ALC utilizes Perception sensors, particularly cameras, to detect surrounding traffic and assess the feasibility of changing lanes safely. The data processing through the Sensor Fusion module generates a comprehensive understanding of the

vehicle's surroundings. The Planning and Vehicle Control component then autonomously plans and executes the lane change maneuver, considering factors such as the speed and distance of nearby vehicles. The System Management module ensures the reliability of the entire process by monitoring the performance of the sensors and actuators involved.

2.3.3 Predictive Maintenance Strategy

Project Ohm's architecture aligns with contractual requirements for predictive maintenance. The network and cloud connection capabilities facilitate continuous data collection from various sensors embedded within the vehicle. This data is analyzed using advanced analytics algorithms in the cloud to detect patterns indicative of potential failures or maintenance needs. The System Management module triggers alerts and notifications to system administrators, enabling proactive servicing to prevent unexpected breakdowns and optimize vehicle uptime.

This section ensures that Project Ohm's architecture not only meets functional expectations but also aligns with contractual commitments, emphasizing the importance of reliability, safety, and adherence to regulations.

Section 3: Requirements

3.1 Functional Requirements

3.1.1 IoT

3.1.1.1 The IoT module ensures that the car is ready for operation once it's turned on. Before any driving activity can commence, this module checks if the car is in a suitable state to drive, setting the precondition that the car is turned on. Once the necessary checks are completed, the post-condition confirms that the car is prepared for driving, ensuring a smooth transition from startup to driving mode.

3.1.1.2 **Pre-Condition:** Driver has a phone key or physical card key to open the Car when the Driver is 5 feet away. Car is turned on by entering the car and pressing the brake pedal. Speedometer reads 0 mph.

3.1.1.3 **Post-Condition:** All necessary systems are initialized and operational. The Car is ready to drive.

3.1.2 Drive

3.1.2.1 The Drive module oversees the crucial aspect of driving, ensuring that the vehicle is safe to operate and that the driver reaches their destination securely. It sets the pre-condition, verifying that the battery has sufficient charge and all system checks pass, ensuring the vehicle's readiness for the journey. The post-condition guarantees that the driver arrives safely at their destination with adequate charge, emphasizing both safety and functionality in driving scenarios.

3.1.2.2 **Pre-Condition:** Battery charge level is above the minimum threshold required for driving. Driver sets the gear to “Drive.” Vehicle systems are operational and free of critical faults.

3.1.2.3 **Post-Condition:** Vehicle reaches the destination without any critical performance issues. Battery charge level is above the minimum required for further operations.

3.1.3 Automatic Parking

3.1.3.1 When the car is stationary and not in use, the Parked module manages its state to maintain security and readiness. It sets the pre-condition by verifying that the car is in the parked gear with either open or closed doors. Once parked, the module ensures that the car doors are securely locked, and it maintains enough charge to resume driving after idling, providing peace of mind to the user regarding the vehicle's status.

3.1.3.2 **Pre-Condition:** Driver activates the automatic parking feature.

Cameras detect available parking spaces and obstacles around the vehicle.

Planning calculates the optimal path for parking based on the detected spaces and obstacles.

3.1.3.3 VCS controls the steering, throttle, and brake to maneuver the vehicle into the parking space.

3.1.3.4 **Post-Condition:** The vehicle parks itself accurately in the selected parking space.

3.1.4 Acceleration Launch

3.1.4.1 Designed to optimize the vehicle's launch performance during acceleration. It starts with the driver engaging the feature, which triggers Sensor Fusion to assess road conditions and engine capabilities. Planning then calculates the best torque distribution and throttle response, while VCS adjusts engine output and traction control settings. The end result is a smooth and controlled launch that maximizes acceleration while maintaining traction and stability, enhancing the overall driving experience.

3.1.4.2 **Pre-Condition:** Driver engages the acceleration launch control feature through the vehicle's infotainment system or dedicated button. Sensor Fusion assesses road surface conditions, tire traction, and engine performance to determine the optimal launch parameters. Planning calculates the ideal torque distribution, gear ratios, and throttle response for maximum acceleration without wheel slip. VCS adjusts the motor output, transmission settings, and traction control systems based on the calculated launch parameters.

3.1.4.3 **Post-Condition:** Vehicle executes a smooth and controlled launch, maximizing acceleration performance while ensuring traction and stability.

3.1.5 Headlights

3.1.5.1 Essential for visibility in low-light conditions, the Headlights module activates when ambient light levels drop below level 3 light or when manually triggered by the driver. It ensures optimal illumination of the road ahead, enhancing safety during nighttime driving. The post-condition confirms that the headlights are operational, providing clear visibility for the driver and other road users.

3.1.5.2 **Pre-Condition:** Ambient light level drops below 3 or the driver manually activates the headlights to low or high settings.

3.1.5.3 **Post-Condition:** Headlights are turned on and functioning properly, providing adequate illumination of the road ahead for the driver and other road users.

3.1.6 Obstacle Avoidance

3.1.6.1 Dedicated to collision prevention, the Obstacle Avoidance module utilizes 8 cameras to detect obstacles in the car's path. It initiates evasive maneuvers to navigate around potential hazards safely, prioritizing the safety of both occupants and surrounding objects. The post-condition ensures that the vehicle successfully avoids the obstacle, minimizing the risk of accidents and enhancing overall driving safety.

3.1.6.2 **Pre-Condition:** Object detection sensors detect an obstacle in the car’s path. The obstacle avoidance system is engaged and ready to respond to detected obstacles.

3.1.6.3 **Post-Condition:** The car successfully maneuvers to avoid the obstacle, ensuring the safety of both occupants and surrounding objects.

3.1.7 Traffic Sign Detection

3.1.7.1 Integral to obeying traffic regulations, the Traffic Sign Detection module identifies and interprets traffic signs using visual recognition technology. It adjusts the car's behavior accordingly, ensuring compliance with speed limits, stop signs, and other road directives, the vehicle will slow by 5 mph each second when above the speed limit and it will apply the brakes in an exponential degree according to how close the vehicle is to the detected stop sign. The post-condition confirms that the car responds appropriately to detected traffic signs, contributing to safer and more law-abiding driving practices.

3.1.7.2 **Pre-Condition:** The visual recognition system detects a traffic sign within the vehicle's field of view.

3.1.7.3 **Post-Condition:** The vehicle interprets the detected traffic sign accurately and adjusts its behavior, such as adjusting speed or obeying stop signs, in accordance with traffic regulations.

3.1.8 Navigation

3.1.8.1 Facilitating route guidance, the Navigation module assists the driver in reaching their destination efficiently. It processes user-inputted destinations or calculates optimal routes based on various factors, such as traffic and road conditions. The post-condition ensures that the car follows the planned route accurately, guiding the driver to their destination with ease and precision.

3.1.8.2 **Pre-Condition:** Driver inputs a destination or the navigation system calculates a route. The GPS module is engaged and ready to provide route guidance.

3.1.8.3 **Post-Condition:** Car follows the planned route accurately, guiding the driver to the destination with ease and precision. Driver can follow the planned route as well.

3.1.9 Crash Detection

3.1.9.1 The Crash Detection module continuously monitors vehicle sensors to identify potential collisions. Using accelerometers and impact sensors, it can detect rapid unexpected decelerations or sudden jolts indicating an accident. Working alongside airbags and seatbelt pretensioners, swift and reliable crash detection is crucial for optimizing occupant safety. By differentiating between minor incidents and bonafide collisions, the system activates only when needed to maximize

safety without false positives. The extreme acceleration forces and velocities involved in crashes also require precise timing of life-saving measures during those critical first moments. Post-condition ensures that Automatic Emergency Braking engages within the necessary timeframe to reduce some impact velocity. It also ensures airbag deployment occurs to protect occupants when a collision registers above acceleration thresholds indicating a major accident.

3.1.9.2 **Pre-Condition:** The vehicle experiences an impact registering at least 5 g-force or a deceleration exceeding 0.5 g over 0.2 seconds, indicating a potential collision.

3.1.9.3 **Post-Condition:** Within 0.04 seconds, automatic emergency braking engages to reduce impact velocity by at least 15 mph. Airbags deploy for front and side impacts exceeding 10 g-forces. If airbag sensors register a deployment, emergency services are notified via LTE network within 15 seconds, with option of cancellation during that threshold.

3.1.10 Cruise Control

3.1.10.1 Designed Cruise control functionality aims to reduce driver fatigue on long, open-road trips. By maintaining a consistent speed set by the user, it allows temporary hands-free driving so the user can relax while not needing to manually control the throttle. As an assisted driving feature, cruise control still requires the driver to remain attentive and override it when necessary. For safety and reliability, cruise control has constraints around vehicle speed and environment. It functions best on flat, uncongested roads in smooth driving conditions. Multiple

internal systems monitor for any irregularities needing intervention to deactivate cruise control when unsafe. Post-condition ensures that the vehicle maintains fixed speed control within defined tolerance levels by continuously managing propulsion and braking. It also ensures cruise control disengages appropriately if speeds drop excessively indicating loss of control or if the driver utilizes acceleration/braking to override.

3.1.10.2 **Pre-Condition:** Cruise control activated by button hold press (minimum 2 seconds) when vehicle traveling between 25-80 mph on a grade of less than 8%. No emergency braking events or traction control activation in the past 3 seconds. Sensor Fusion receives the speed request and forwards it to Planning.Planning calculates the optimal throttle and brake settings based on the inputted speed, current vehicle speed, and road conditions.

3.1.10.3 **Post-Condition:** Vehicle maintains set speed within +/- 2.5 mph on flat roads by automatically controlling the throttle and brakes every 0.2 seconds. Disengages if vehicle slowed by more than 10 mph within 2 seconds or any override of controls. Override of controls can differ from steering wheel adjustment, gear change, weight shift in seat, and noticeable fluctuations in environment.

3.2 Non-Functional Requirements

3.2.1 Security

3.2.1.2 With vehicles containing a broad attack surface of complex components and external connectivity, comprehensive cybersecurity protections are essential to guard against malicious threats. From hacking into core control systems to

stealing private customer data, the repercussions of a successful security breach can be severe. Employing a stringent defense-in-depth strategy is therefore critical - integrating firewalls, intrusion detection, anomaly monitoring, event logging, and encryption technologies to provide robust, layered cybersecurity defenses. Additional protections against vulnerabilities must be continually tested and updated as risks evolve. Post-condition ensures all external network traffic is monitored and unauthorized access attempts are instantly blocked by the firewall. It ensures comprehensive logging of all system access attempts as well as expected blocking of unauthorized packets. Encrypted data access is ensured to be swiftly detected and terminated if decryption protections are breached.

3.2.1.3 **Pre-Condition:** System powered on with firewall, encryption, access controls, intrusion prevention, and anomaly detection systems standing by. Log storage has a minimum 2 TB free space available.

3.2.1.4 **Post-Condition:** All inbound network traffic unauthorized by security policy is blocked instantly via firewall. System logs all access attempts along with packets lost from blocking. Cryptographic modules detect and terminate any unauthorized access attempts to decrypted data within 0.01 seconds.

3.2.2 Safety

3.2.2.1 As safety is the utmost priority in vehicle design, comprehensive precautions are necessary to avoid human injury during all conceivable scenarios on public roads. Whether routine driving, system failures, or extreme weather, maintaining control and preventing accidents must be guaranteed. Employing both passive measures

like airbags alongside active collision avoidance and advanced driver aids provides multi-layered protection. With extensive testing across thousands of simulated and real-world test miles, safety systems aim to bring risk as close to zero as technologically feasible. Post-condition ensures that across emergency scenarios from component failures to hazardous weather, the vehicle achieves stability through steering and braking control intervention. It ensures that the systems minimize kinetic energy and avoid damage or injury considering occupants along with external persons/property based on regulatory standards.

3.2.2.2 **Pre-Condition:** All vehicle active safety systems, sensors, redundancies, and diagnostics online and reporting normal status. Environmental sensors analyzing weather, road conditions, and nearby vehicles/objects.

3.2.2.3 **Post-Condition:** In emergency situations, the car achieves the best possible outcome by controlling braking, steering, etc. Risk is minimized by avoiding damage, injury, and fatalities, considering both vehicle occupants and external human/property with equal priority per regulations.

3.2.3 Software Updates

3.2.3.1 Regular software updates are essential to fix bugs, patch vulnerabilities, and add new capabilities over a vehicle's lifetime. With increasing firmware complexity across multiple electronic control units, applying updates is also complicated, necessitating a robust update mechanism. A modular, fail-safe architecture maintains operational safety during the update process. Cryptographic verification of update code authenticity and integrity prevents unauthorized or corrupted

firmware changes. Together with backward compatibility testing and minimum version enforcement, updates can enable continuous improvement. Post-condition ensures the software update files are rapidly verified for authenticity and integrity through cryptographic mechanisms before storage and installation. It ensures updates of failing checks are blocked while successful changes are atomically saved and reported to management infrastructure. The system ensures rollbacks after failed installs and restarts after successful updates to properly load the new software.

3.2.3.2 **Pre-Condition:** Vehicle in Park with at least 2 bars LTE signal strength. Dependent systems safely deactivated per update specifications.

3.2.3.3 **Post-Condition:** Update package cryptographically verified and saved atomically to secondary storage location within 5 seconds. Successful updates reported to the management server. Any update failures or integrity issues halt the process to prevent corruption. Vehicle reboot conducted to load updates if necessary while the user is notified of status.

3.2.4 Maintenance

3.2.4.1 Ensure prolonged system reliability and performance by implementing the Maintenance module, designed to detect and address issues requiring maintenance or diagnostic checks efficiently. Notify relevant stakeholders and schedule maintenance activities without compromising vehicle functionality.

3.2.4.2 **Pre-Condition:** Implement a mechanism to detect maintenance needs based on predefined criteria.

3.2.4.3 **Post-Condition:** Establish a notification system to inform relevant parties and schedule maintenance activities, ensuring system functionality remains intact.

3.2.5 Reliability

3.2.5.1 Code the Reliability module to guarantee consistent and predictable system operation under diverse conditions, minimizing the risk of unexpected failures or malfunctions. Maintain performance standards and system integrity to enhance user trust and satisfaction.

3.2.5.2 **Pre-Condition:** System is in use.

3.2.5.3 **Post-Condition:** Confirm that the system operates consistently and predictably under varying conditions, avoiding unexpected failures or malfunctions.

3.2.6 Performance

3.2.6.1 Implement the Performance module to optimize system responsiveness to user inputs and environmental changes. Optimize resource utilization and task execution for a seamless and efficient user experience.

3.2.6.2 **Pre-Condition:** Ensure the system is operational and ready to respond to user inputs and environmental changes.

3.2.6.3 **Post-Condition:** Verify that the system responds promptly to user inputs and environmental changes, providing a smooth and efficient operation.

3.2.7 Technician Login

3.2.7.1 Facilitating system maintenance and diagnostics, the Technician Login module grants authorized technicians full access to the system management interface. It prompts technicians to authenticate their credentials securely before providing access to sensitive functionalities such as file log retrieval and software updates. The post-condition confirms successful login, enabling technicians to perform their tasks effectively and efficiently.

3.2.7.2 **Pre-Condition:** Prompt the system management interface to request technician credentials.

3.2.7.3 **Post-Condition:** Enable technicians to access the system securely, granting them full capabilities for file log retrieval, software updates, and diagnostics.

3.2.8 Display

3.2.8.1 Develop the Display module as the interface for technicians to navigate and verify system updates. Provide a user-friendly interface for easy verification of installed updates and system status during maintenance procedures.

3.2.8.2 **Pre-Condition:** Ensure the display offers a UI for technicians to navigate through.

3.2.8.3 **Post-Condition:** Ensure technicians can effortlessly verify the correct installation of all system updates through the display interface.

Section 4: Requirement Modeling

4.1 Use Cases

4.1.1 Use Case 1: Automatic Parking

- **Pre-Condition:** Driver activates the automatic parking feature.
- **Post-Condition:** The Car parks itself accurately in the selected parking space.
- **Trigger:** Driver activates automatic parking through a button on the screen.

1.) Driver activates the automatic parking feature from the control panel.

2.) The Parked module verifies that the car is in the parked gear and doors are closed.

3.) Cameras detect available parking spaces and obstacles around the vehicle.

4.) Planning calculates the optimal path for parking based on the detected spaces and obstacles.

5.) VCS controls the steering, throttle, and brake to maneuver the vehicle into the parking space.

6.) The Car parks itself accurately in the selected parking space.

7.) The Parked module ensures that the doors are securely locked.

8.) The Parked module maintains enough charge for the vehicle to resume driving after idling.

9.) Confirmation message is displayed on the control panel indicating successful parking.

10.) The system logs all related data, including parking location, time, and any exceptions encountered.

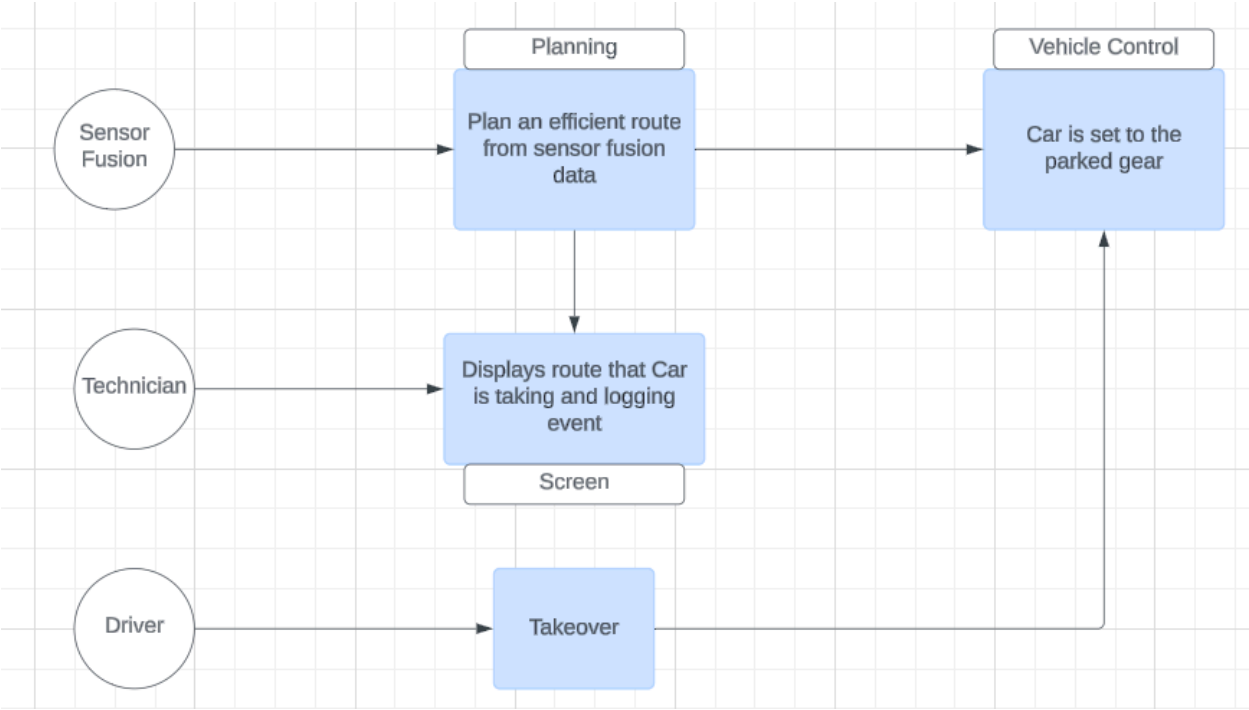
11.) Exceptions:

a.) If the Car is not in the parked gear or the doors are not closed, the Parked module displays an error message and prevents activation of automatic parking.

b.) If obstacles are detected in the selected parking space that cannot be navigated around, the system notifies the driver and prompts for manual intervention.

c.) If the Driver cancels the automatic parking process midway, the system returns to the initial pre-parking state and awaits further instructions.

d.) Any technical malfunction detected during the parking process triggers an error message and logs the issue for system maintenance.

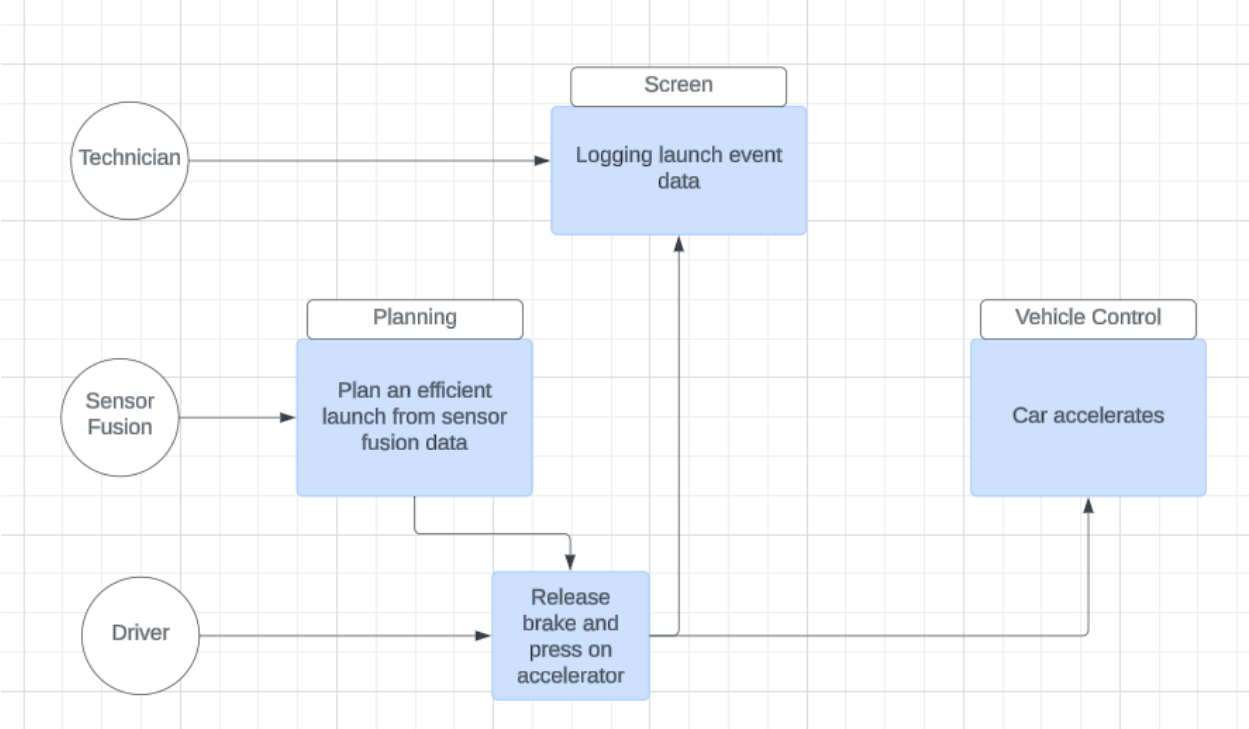


4.1.2 Use Case 2: Acceleration Launch

- **Pre-Condition:** The Driver engages the acceleration launch control feature through the Car's infotainment system or a dedicated button.
- **Post-Condition:** The Car executes a smooth and controlled launch, maximizing acceleration performance while ensuring traction and stability.
- **Trigger:** The Driver activates the acceleration launch control feature through the Car's infotainment system or a dedicated button.

1.) The Driver engages the acceleration launch control feature through the Car's infotainment system or a dedicated button.
2.) Sensor Fusion assesses road surface conditions, tire traction, and engine performance to determine the optimal launch parameters.
3.) Planning calculates the ideal torque distribution, gear ratios, and throttle response for maximum acceleration without wheel slip.
4.) VCS adjusts the motor output, transmission settings, and traction control systems based on the calculated launch parameters.
5.) The Car's motor output and traction control settings are optimized to achieve a smooth and controlled launch.
6.) The Driver releases the brake pedal while maintaining firm pressure on the accelerator pedal.
7.) The Car executes the launch with maximum acceleration while ensuring traction and stability.
8.) The acceleration launch control system monitors wheel slip and adjusts torque distribution as needed during the launch process.
9.) Once the Car reaches the desired speed or acceleration, the acceleration launch control system returns to normal driving mode.
10.) The system logs all acceleration launch control-related data, including activation, parameters, and performance metrics.
11.) Exceptions:
    a.) If road surface conditions are deemed unsafe for acceleration launch (e.g., wet or icy roads), the system alerts the driver and recommends deactivating the feature.

b.) If motor performance or traction control issues are detected during the assessment phase, the system advises the driver to check the Car's motor and traction systems before attempting an acceleration launch.



4.1.3 Use Case 3: Obstacle Avoidance

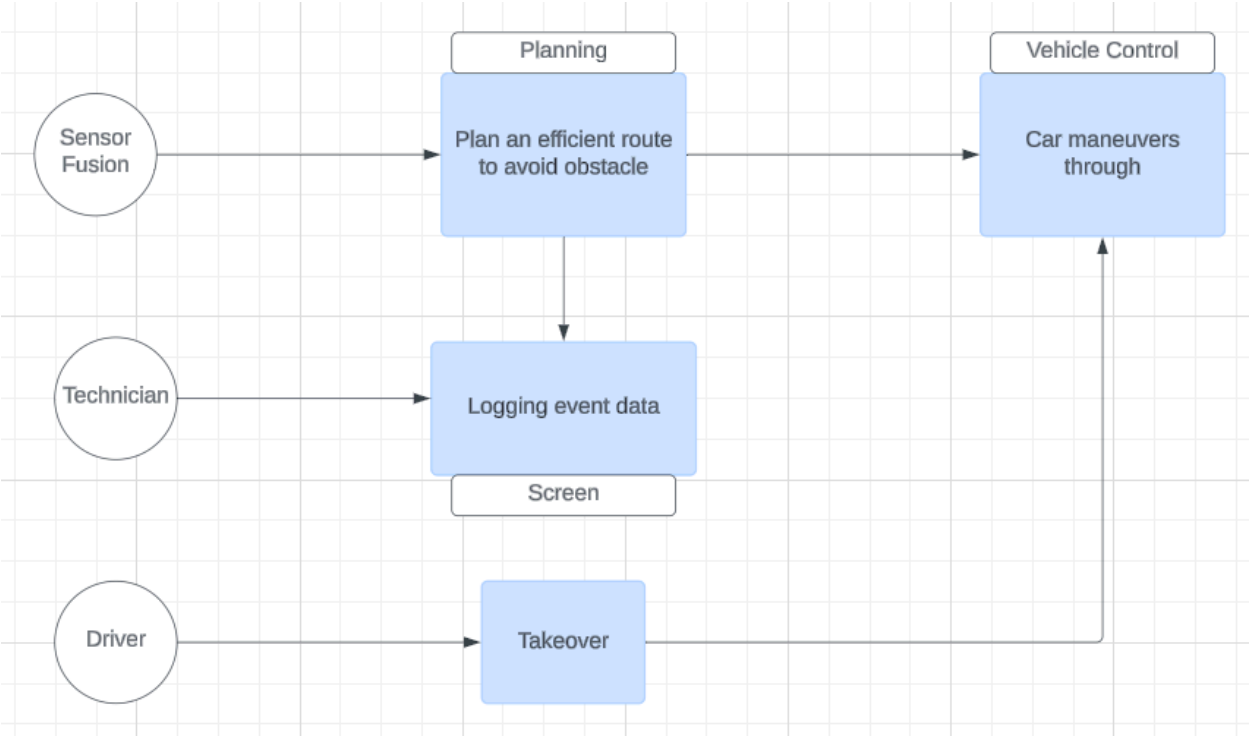
- **Pre-Condition:** Object detection sensors detect an obstacle in the car's path.
- **Post-Condition:** The Car successfully maneuvers to avoid the obstacle.
- **Trigger:** Cameras detect an obstacle is in the Car's way.

1.) Cameras detect an obstacle in the car's path and send data to the obstacle avoidance system.

2.) The obstacle avoidance system analyzes the data received from the sensors to determine the size, distance, and trajectory of the obstacle.

3.) Based on the analysis, the obstacle avoidance system calculates an evasive maneuver to safely navigate around the obstacle.

4.) The system prioritizes the safety of both occupants and surrounding objects in planning the evasive maneuver.
5.) The system communicates with the VCS to execute the calculated evasive maneuver.
6.) VCS controls the steering, throttle, and brake to maneuver the vehicle and avoid the obstacle safely.
7.) The vehicle successfully executes the evasive maneuver, ensuring the safety of both occupants and surrounding objects.
8.) The system logs all related data, including the obstacle detected, evasive maneuver executed, and any exceptions encountered during the process.
9.) Exceptions:
    a.) If the obstacle is too close or too large to navigate around safely, the system alerts the driver and suggests alternative actions, such as stopping the vehicle or changing direction.
    b.) In case of multiple obstacles detected simultaneously, the system prioritizes the most immediate threat and plans the evasive maneuver accordingly.
    c.) If the obstacle avoidance system experiences a technical malfunction or failure, the system notifies the driver and advises manual intervention or system maintenance.
    d.) Any unexpected environmental conditions, such as extreme weather or road surface changes, may affect the effectiveness of the obstacle avoidance system. The system alerts the driver to exercise caution in such situations.



4.1.4 Use Case 4: Traffic Sign Detection

- **Pre-Condition:** Cameras detect a traffic sign within the Car's field of view.
- **Post-Condition:** The Car interprets the detected traffic sign accurately.
- **Trigger:** Cameras detect a traffic sign.

1.) Cameras analyze the data to identify and interpret the detected traffic sign.

2.) Data is sent to the sensor fusion module.

3.) If the traffic sign indicates a speed limit, the VCS calculates the difference between the current vehicle speed and the speed limit.

4.) If the Car is exceeding the speed limit, the VCS initiates a gradual speed reduction by 5 mph each second until the speed limit is reached.

5.) If the traffic sign indicates a stop sign, the VCS calculates the distance to the stop sign and applies the brakes in an exponential manner based on the proximity to the stop sign.

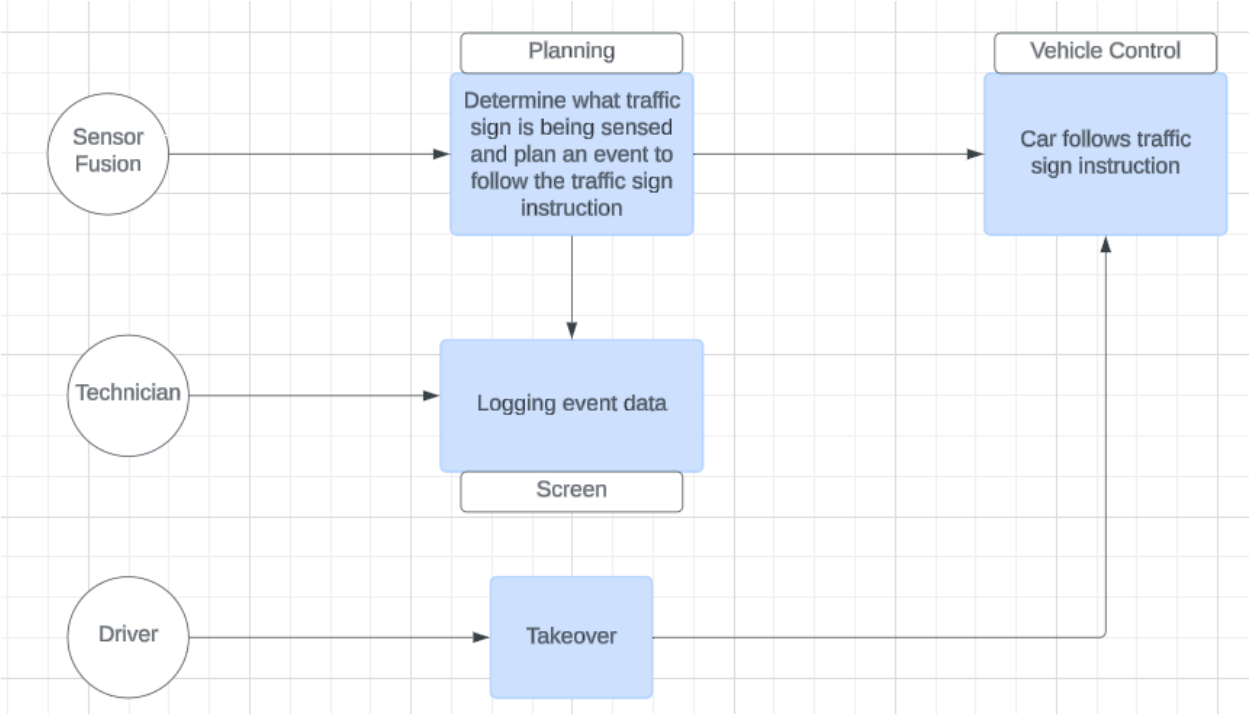
6.) The Car adjusts its behavior according to the interpreted traffic sign, such as slowing down to comply with speed limits or coming to a complete stop at stop signs.

7.) The system logs all related data, including the type of traffic sign detected, actions taken by the vehicle, and any exceptions encountered during the process.

8.) Exceptions:

a.) If the visual recognition system fails to detect a traffic sign or misinterprets the sign, the system alerts the driver and advises manual intervention.

b.) In case of conflicting or ambiguous traffic signs, the system prioritizes the most relevant sign or prompts the driver for clarification.



4.1.5 Use Case 5: Cruise Control

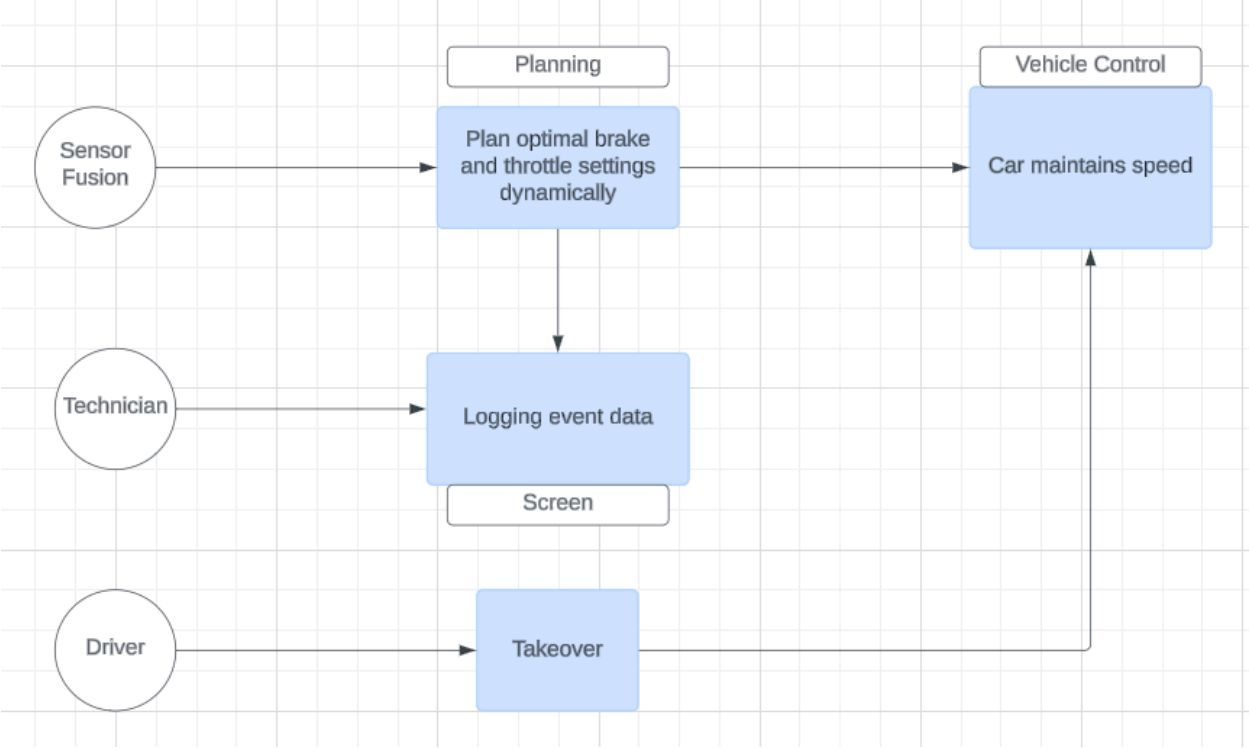
Pre-Condition: Cruise control is activated by the Driver.

Post-Condition: Car maintains the set speed within +/- 2.5 mph on flat roads.

Trigger: The Driver activates cruise control by holding the cruise control button for at least 2 seconds.

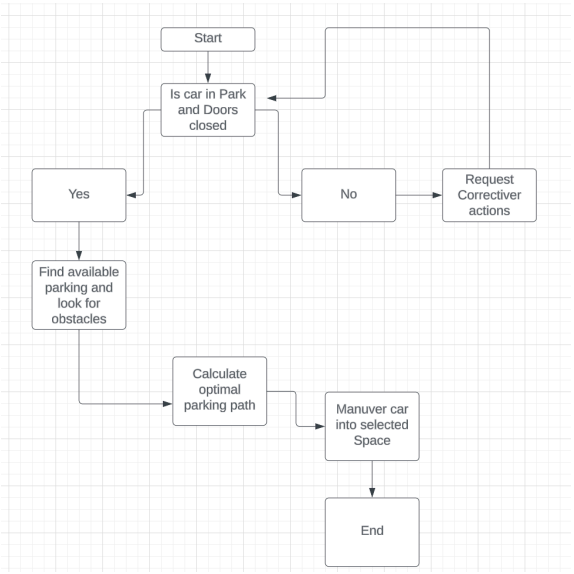
1.) The Driver activates cruise control by holding the cruise control button for a minimum of 2 seconds while the Car is traveling between 25-80 mph on a grade of less than 8%.
2.) Sensor Fusion receives the speed request and forwards it to the Planning module.
3.) Planning calculates the optimal throttle and brake settings based on the inputted speed, current Car speed, and road conditions.
4.) The Car's propulsion and braking systems are automatically controlled by the Planning module every 0.2 seconds to maintain the set speed within +/- 2.5 mph on flat roads.
5.) If the Car experiences an emergency braking event or traction control activation within the past 3 seconds, cruise control remains deactivated.
6.) Cruise control disengages if the Car slows by more than 10 mph within 2 seconds or if any controls are overridden, such as steering wheel adjustments, gear changes, weight shifts in the seat, or noticeable fluctuations in the environment.
7.) The Driver can override cruise control at any time by applying the brakes or accelerating manually.
8.) Cruise control continuously monitors for irregularities and intervenes to deactivate itself if unsafe conditions are detected.
9.) The system logs all cruise control-related data, including activation, operation, overrides, and disengagement events.
10.) Exceptions:
    a.) If the road grade exceeds 8% or the vehicle speed is outside the allowed range (25-80 mph), cruise control remains inactive.

b.) If emergency braking events or traction control activations occur within the past 3 seconds, cruise control remains deactivated until the system verifies safe conditions.

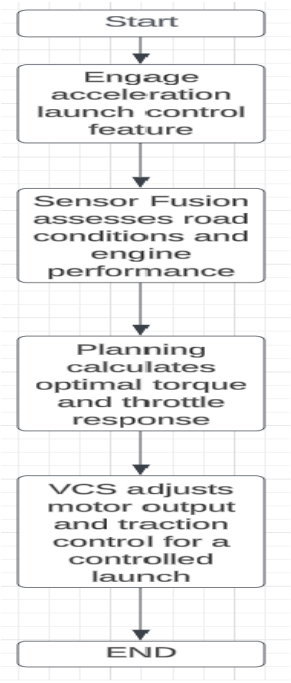


4.2 Activity Diagrams

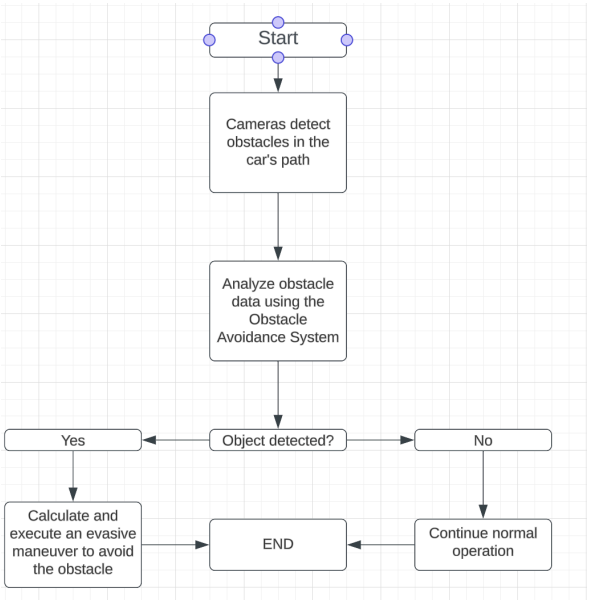
4.2.1 Automatic Parking Activity Diagram



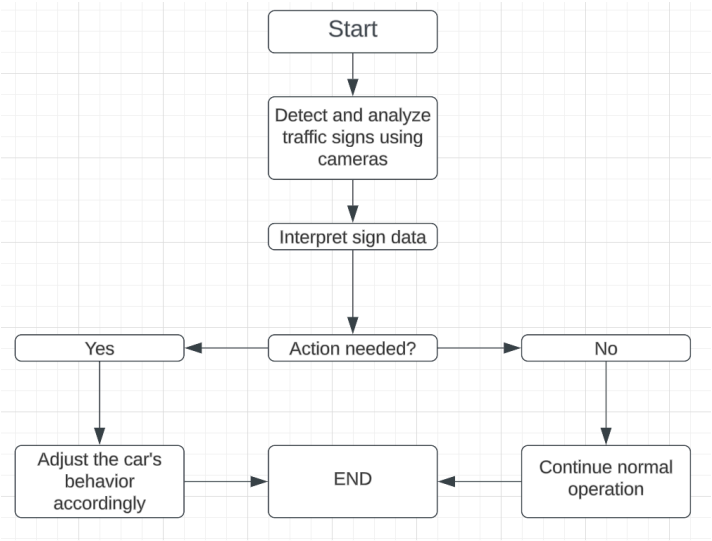
4.2.2 Acceleration Launch Activity Diagram



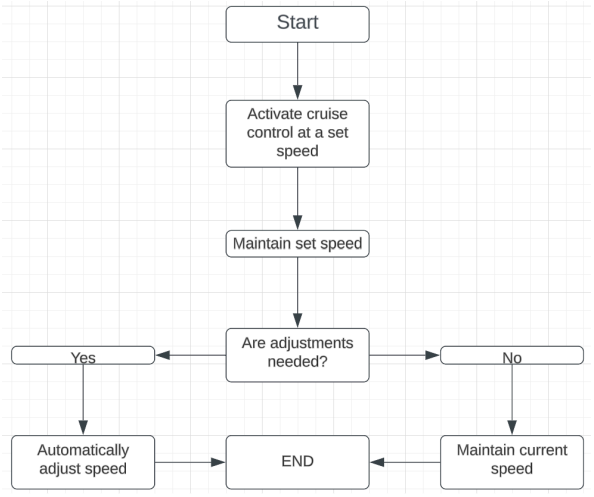
4.2.3 Obstacle Avoidance Activity Diagram



4.2.4 Traffic Sign Detection Activity Diagram

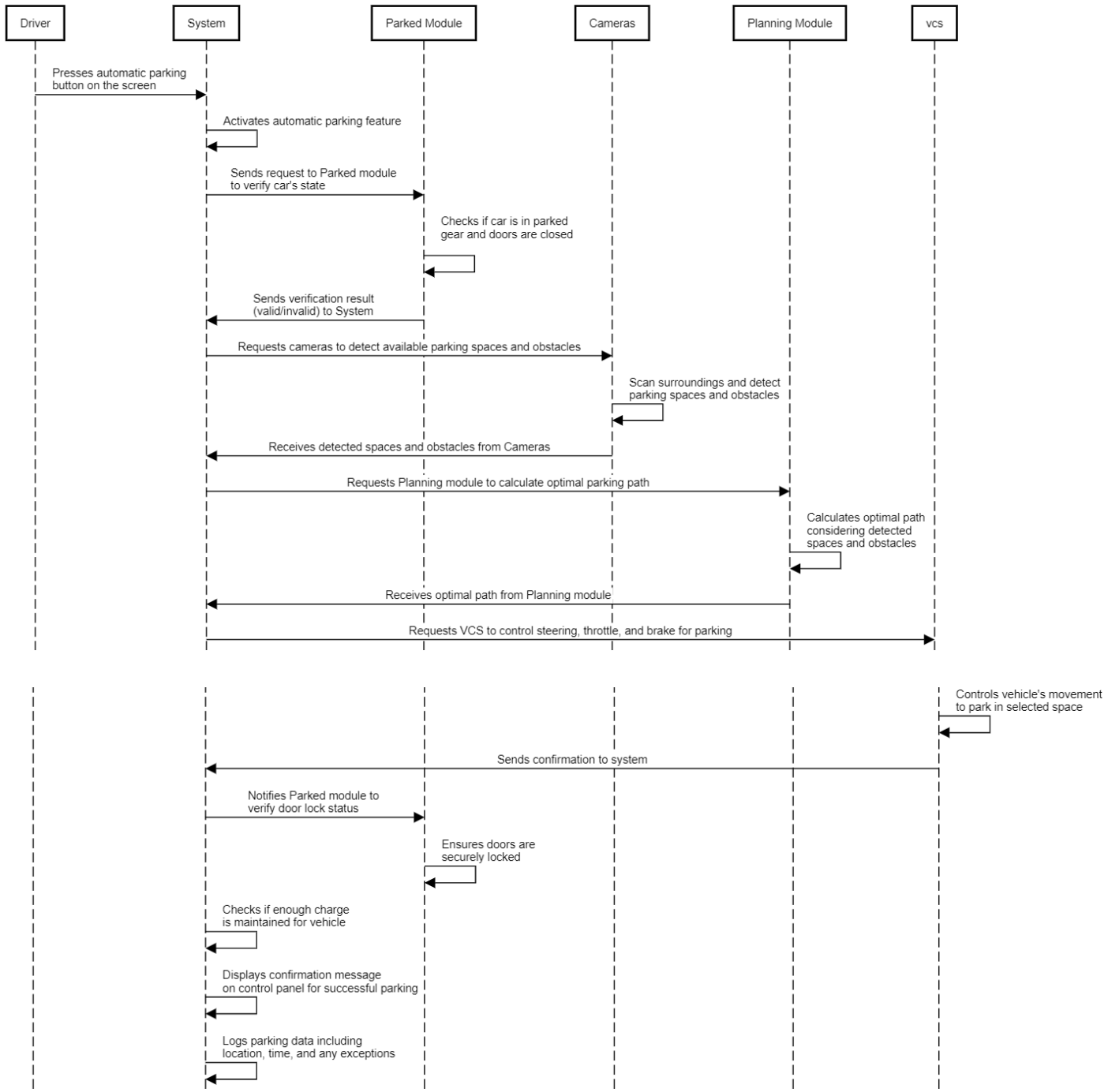


4.2.5 Cruise Control Activity Diagram

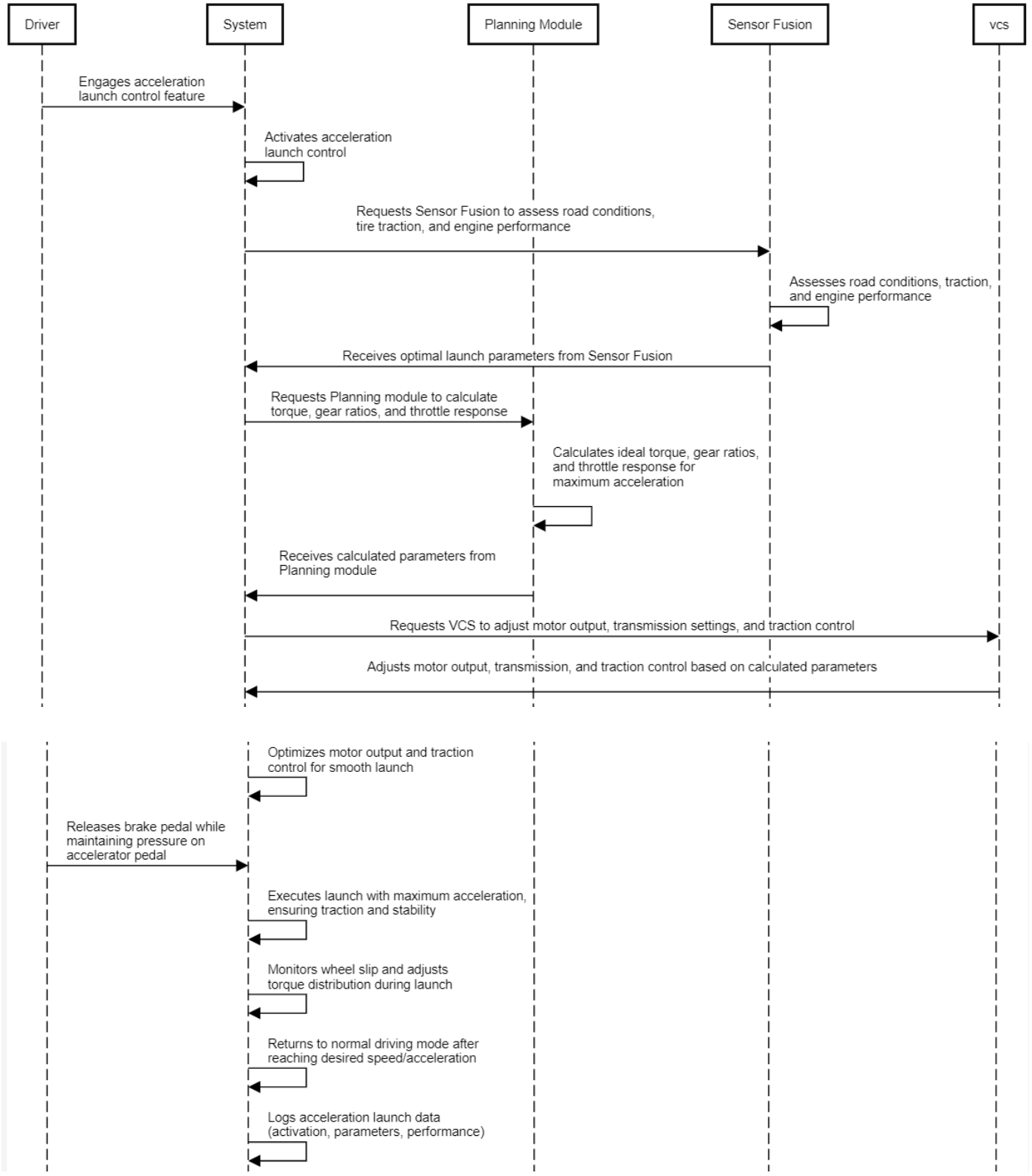


4.3 Sequence Diagrams

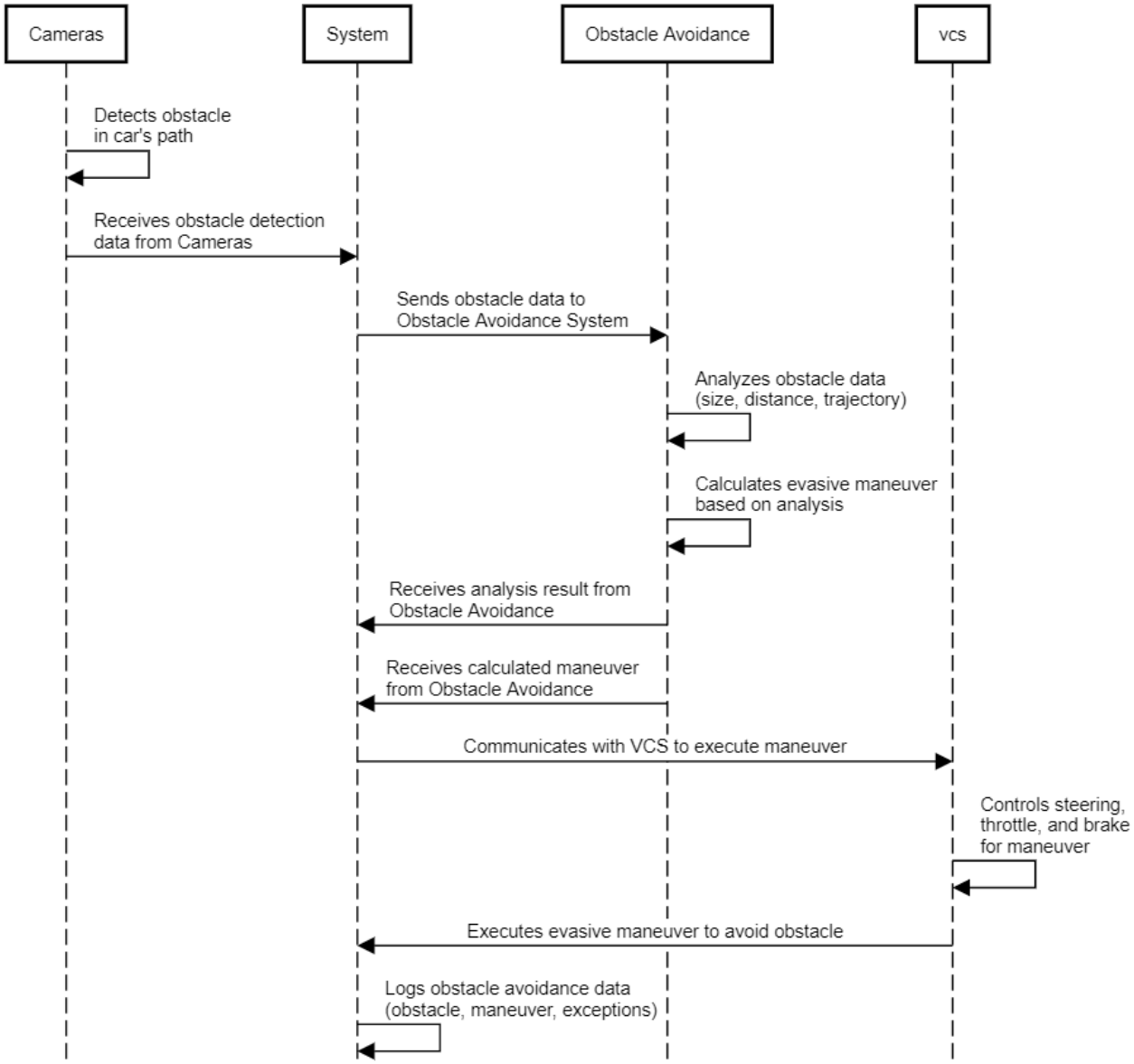
4.3.1 Automatic Parking Sequence Diagram



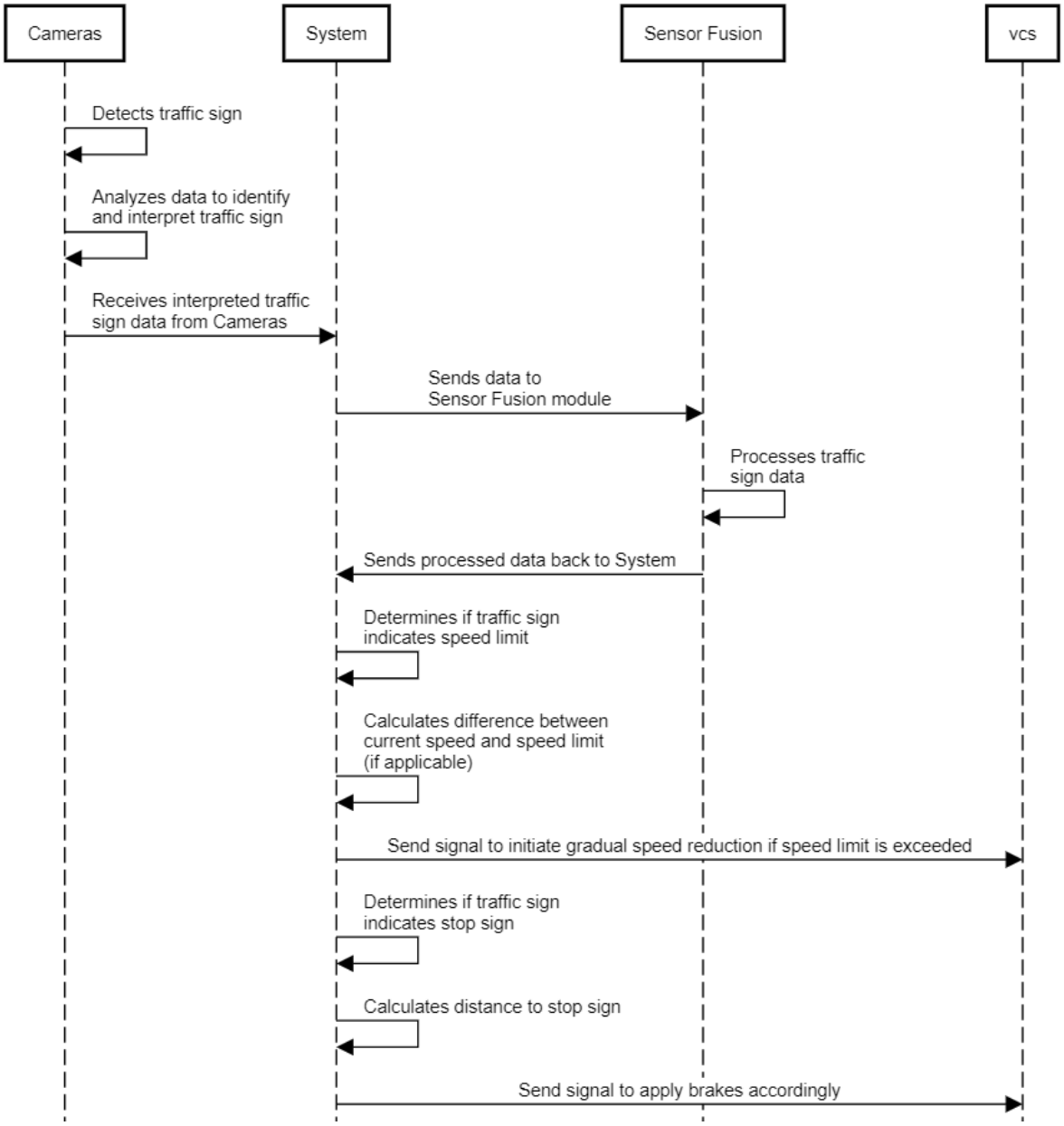
4.3.2 Acceleration Launch Sequence Diagram

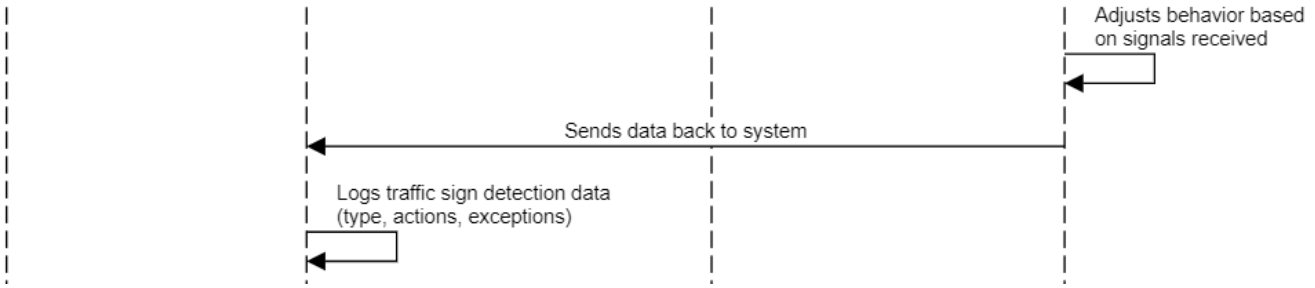


4.3.3 Obstacle Avoidance Sequence Diagram



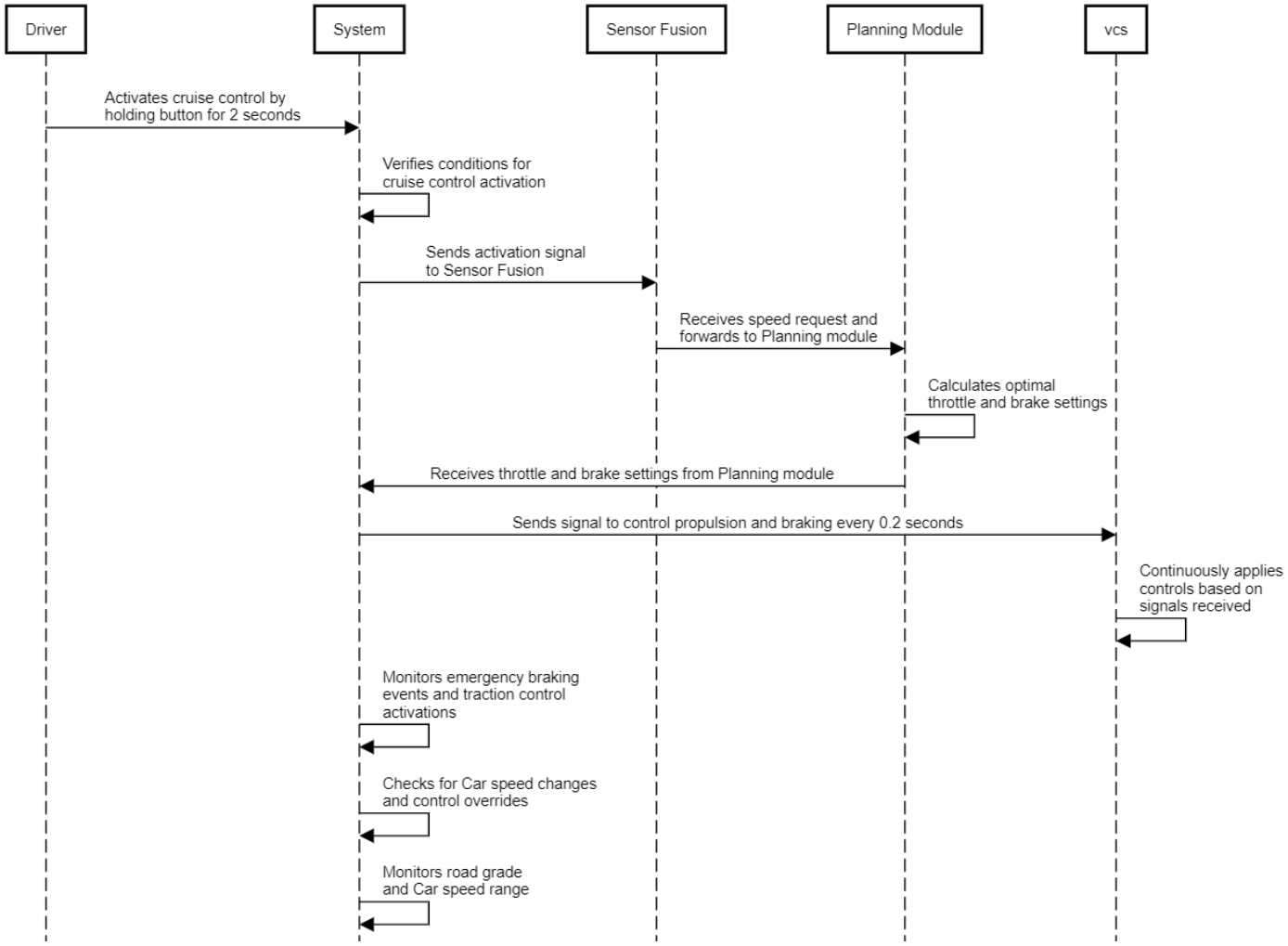
4.3.4 Traffic Sign Detection Sequence Diagram

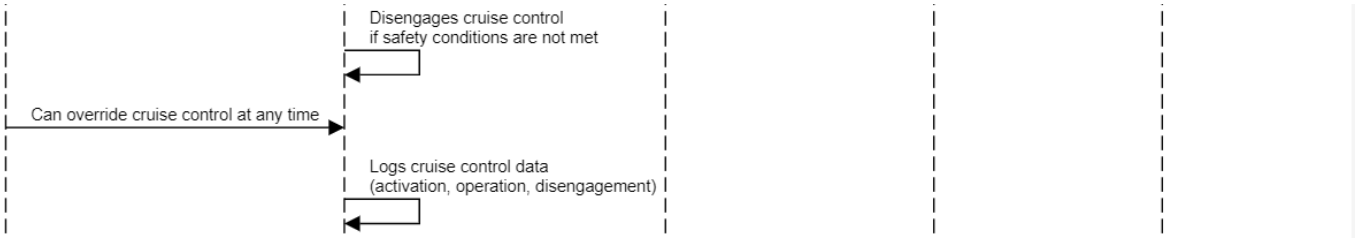




4.3.5 Cruise Control Sequence Diagram

Participant | Interaction





4.4 Classes

4.4.1 Automatic Parking Class

- AutomaticParkingSystem: Activate automatic parking feature, coordinate communication between modules, handle preconditions and postconditions, log data (long struct).
- ParkedModule: Verify car state, ensure secure parking (bool), maintain charge for resuming driving, display confirmation message.
- Cameras: Detect parking spaces and obstacles, send long integer array data to the Planning module.
- Planning: Calculate optimal parking path, communicate with VCS for vehicle control (list of long int).
- VCS (Vehicle Control System): Control vehicle movement for parking, adjust motor output and traction control.

4.4.2 Acceleration Launch Class

- AccelerationLaunchSystem: Activate acceleration launch control, coordinate communication, handle preconditions and postconditions, log data (long int list).
- SensorFusion: Assess road conditions and engine performance, send data to the Planning module (struct of long ints and bools).

- Planning: Calculate torque distribution and throttle response, communicate with VCS for control (two long ints).

- VCS (Vehicle Control System): Control motor output, transmission settings, and traction control for launch.

4.4.3 Obstacle Avoidance Class

- ObstacleAvoidanceSystem: Detect obstacles (long int array), analyze data for evasive maneuver, coordinate communication (long int list), log data (struct of long int and list of int).

- SensorFusion: Receive obstacle data, send it to Planning module (long int array).

- Planning: Calculate evasive maneuver, communicate with VCS for execution (long int list).

- VCS (Vehicle Control System): Control steering, throttle, and brake for obstacle avoidance.

4.4.4 Traffic Sign Detection Class

- TrafficSignDetectionSystem: Detect traffic signs, interpret signs, coordinate communication(list of int/bool structs), log data.

- SensorFusion: Receive traffic sign data, send it to Planning module (list of long integers).

- Planning: Interpret signs, calculate necessary actions, communicate with VCS for execution (long int list).

- VCS (Vehicle Control System): Control propulsion and braking based on traffic signs.

4.4.5 Cruise Control Class

- CruiseControlSystem: Activate and manage cruise control, monitor vehicle conditions, coordinate communication (list of bools and a long integer), log data.

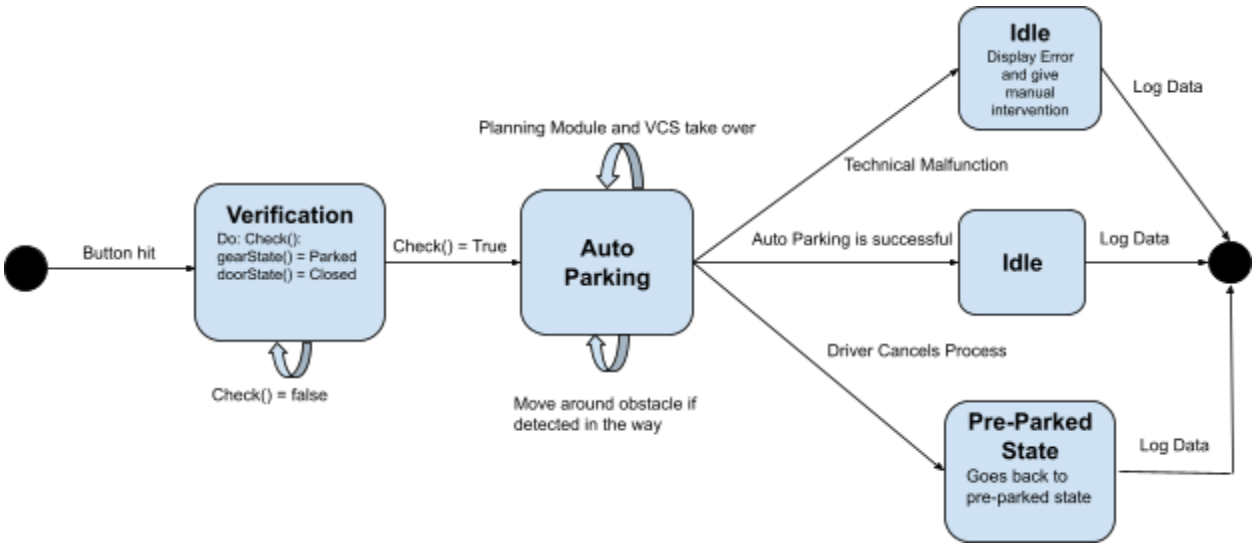
- SensorFusion: Receive speed request, send data to Planning module (list of long int/bool structs).

- Planning: Calculate throttle and brake settings, monitor events, communicate with VCS for control (long int list).

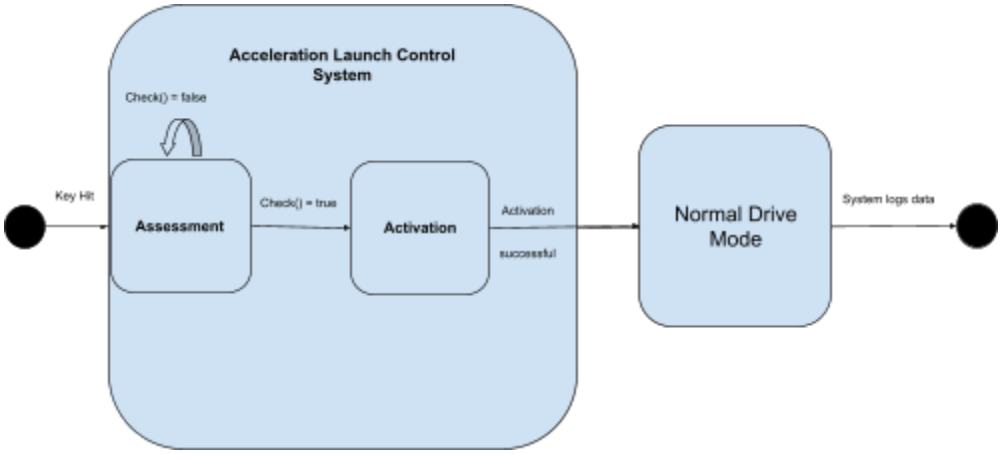
- VCS (Vehicle Control System): Control propulsion and braking, deactivate cruise control if necessary.

4.5 State Diagrams

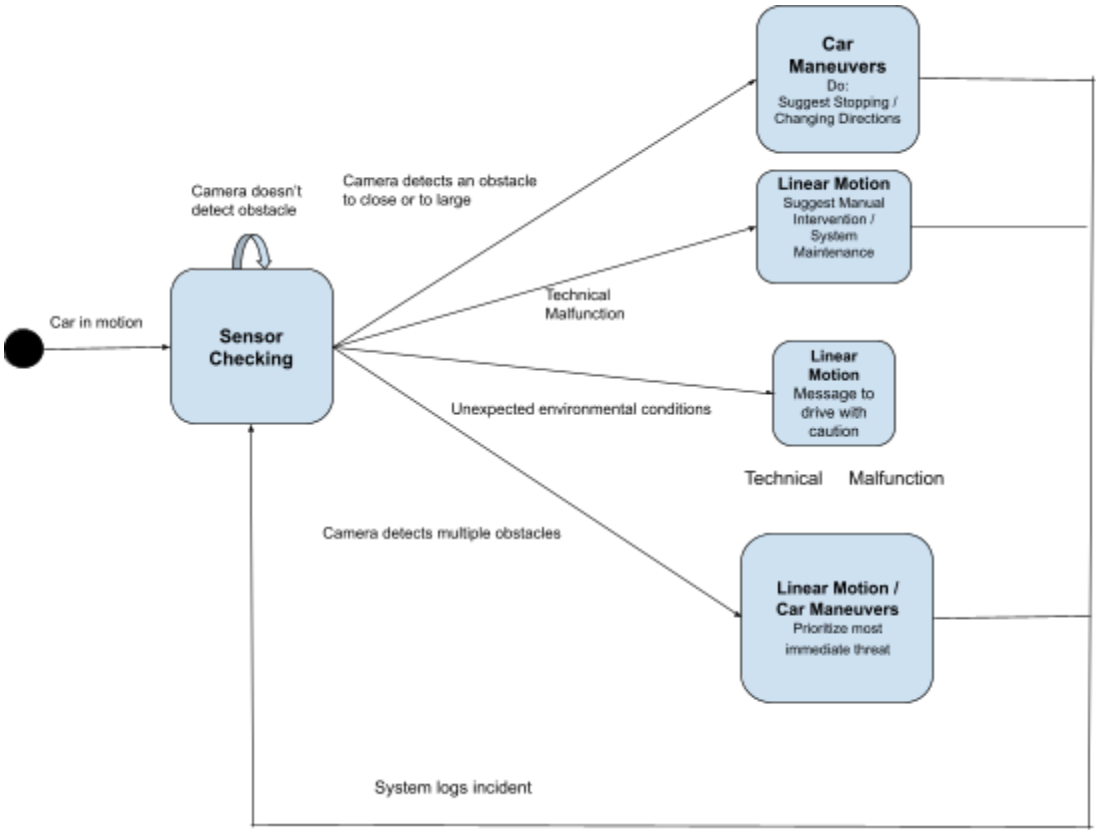
4.5.1 Automatic Parking State Diagram



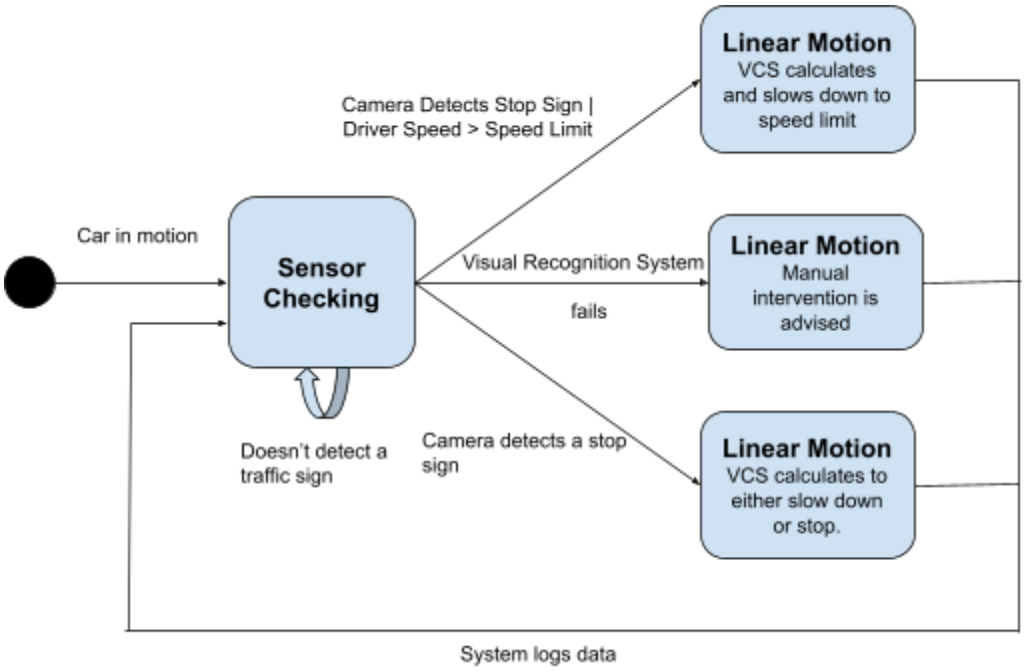
4.5.2 Acceleration Launch State Diagram



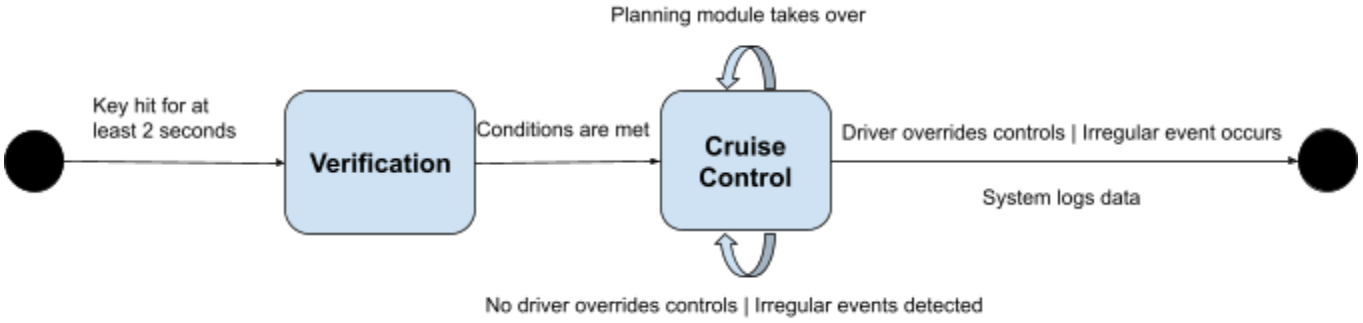
4.5.3 Obstacle Avoidance State Diagram



4.5.4 Traffic Sign Detection State Diagram



4.5.5 Cruise Control State Diagram



Section 5: Design

5.1 Software Architecture

In this section, we will look over several software design architectures. After analyzing the pros and cons of each, we will determine which software architecture is best suited for Project Ohm.

5.1.1 Data Centered Architecture

Pros:

- Direct access to cloud data
    - Every class within the architecture has the ability to access cloud-based data directly. This means that there's no need for convoluted pathways or intermediary steps to retrieve or manipulate data stored in the cloud.
- Simple and linear structures
    - The architecture follows a linear structure, which means that each software class communicates through a single line. This design ensures that every process within the system takes a similar amount of time, leading to a more predictable and efficient system overall.
- Reduced complexity
    - By having a clear and straightforward structure, the data-centered architecture minimizes the risk of errors or data entanglement. Since there's only one bidirectional connection line between each client software, there are fewer opportunities for data mix-ups or confusion.
- Simplified debugging

- Debugging becomes easier in a data-centered architecture due to its simplicity. With fewer potential points of failure and a clear communication pathway, developers can quickly identify and resolve issues that may arise within the system.

Cons:

- Lack of interconnected access
  - One limitation of this architecture is that classes do not have direct interconnected access. This means that if two client software components need to communicate with each other, they must always go through the data store, adding an extra step to the process.

The data-centered architecture is a good fit for IoT due to its direct access to cloud data, simplified debugging, and reduced complexity, which align well with the data management needs of IoT devices.

5.1.2 Data Flow Architecture

Pros:

- Enhanced connectivity
  - The data flow architecture facilitates more extensive connectivity, allowing multiple software components and interfaces to share a greater amount of information simultaneously.
- Non-linear data flow
  - Unlike a strictly linear flow of data, this architecture incorporates cyclic data flows in certain areas. This cyclic flow can be beneficial for ongoing processes and interactions within the system.

- Sequential data transmission
    - Data in this architecture flows through a series of interconnected pipes. This sequential flow ensures that data reaching a specific point has passed through preceding stages, aiding in tracking and monitoring data transmission.

Cons:

- Increased complexity in structure
    - The interconnected nature of the architecture makes it more challenging to identify and resolve errors. A bug originating in one software class that serves as input for another may propagate incorrectly and become difficult to trace back to its source.
- Multiple incoming data pipes
    - The presence of multiple incoming pipes can make it harder to pinpoint the source of errors or potential issues. Identifying and addressing errors stemming from multiple pipes can increase the complexity of troubleshooting within the system.

The data flow architecture's enhanced connectivity and non-linear data flow make it a strong choice for IoT applications. However, its increased complexity in structure and multiple incoming data pipes may pose challenges in error identification and troubleshooting, requiring robust debugging mechanisms for efficient system maintenance.

5.1.3 Call Return Architecture

Pros:

- Hierarchical Structure

    - The call return architecture features a hierarchical structure that can simplify data management tasks. This organized approach helps in structuring and handling data more effectively within the system.
- Subprogram Organization
    - Organizing software into programs and subprograms is a key feature of this architecture. This organizational hierarchy not only makes coding more manageable but also provides greater control over smaller details within specific software sections.

Cons:

- Data Transmission Directionality
    - Unlike some architectures with predefined data flow directions, the connections in call return architecture allow for bidirectional data transmission between different programs and subprograms. This lack of strict directionality can sometimes lead to data being transferred back and forth without following a specific order or pattern.
- Complexity with Subprogram
    - One challenge of call return architecture is the potential proliferation of subprograms. As the number of subprograms (and subprograms of subprograms) increases, managing and tracking the code can become complex. This complexity is particularly evident when minor changes need to be applied across multiple subprograms.

The call return architecture is an excellent fit for IoT HTL due to its hierarchical structure and subprogram organization. This structured approach streamlines data management tasks, making

it easier to handle the extensive data flow typical of IoT devices in a hotel environment. The hierarchical arrangement not only simplifies coding but also enhances control over software sections, ensuring greater efficiency in managing and processing data. Overall, the call return architecture offers a robust framework that aligns well with the data management needs of IoT applications in the hotel industry.

5.1.4 Object-Oriented Architecture

Pros:

- Instance Creation
    - Object-oriented architecture allows for the creation of multiple instances of a single class. This means that class attributes only need to be declared once, and any subsequent instances will be linked back to the class constructor. This approach simplifies data handling tasks.
- Reusability and Abstraction
    - Through features like polymorphism and abstraction, object-oriented architecture promotes code reusability. This means that less code needs to be written, but the existing code can be reused or modified efficiently. As a result, development time is significantly reduced.

Cons:

- Restriction to Classes and Objects
    - A drawback of object-oriented architecture is its limitation to classes and objects. Determining which classes are necessary for specific functionalities can be challenging. This may lead to frequent revisions of class numbers and attributes to accommodate changing requirements.

Additionally, simple tasks may require unnecessary implementation within a class, adding to development workload.

- Limited Reusability in General Cases
    - While object-oriented designs offer reusability, they can be highly specific to their architecture. This specificity may limit their reusability to very particular and similar scenarios, making them less versatile in general cases.

Object-oriented architecture is well-suited for IoT applications due to its instance creation capabilities and code reusability features. The ability to create multiple instances of a single class simplifies data handling, making it ideal for managing diverse data streams typical of IoT devices. Features like polymorphism and abstraction promote efficient code reuse, reducing development time and effort. While there may be challenges in determining necessary classes and limitations in reusability for specific scenarios, the overall benefits align well with the complexities and data management needs of IoT applications.

5.1.5 Layered Architecture

Pros:

- Organized Structure with Layers
    - Layered architecture is organized into distinct sections or layers, each representing a specific interface. This structure facilitates grouping related functionalities together, making it easier to work on and manage them collectively.
- Simple and Cooperative Framework

    - Implementing each layer in a layered architecture is typically straightforward and cooperative. This simplicity ensures that developers can focus on developing each layer without facing overwhelming challenges.
- Improved Testing Capabilities
    - Testing is better facilitated in layered architecture due to the separation of components within each layer. This isolation allows for more efficient testing, as engineers can test components within a layer without interference from components in other layers.

Cons:

- Inter-Layer Dependencies
    - One drawback of layered architecture is the potential for harmful effects of layers on each other. If one layer is dependent on another and experiences issues, it may cause a chain reaction of bugs or errors across the dependent layers.
- Revisions Impacting Multiple Layers
    - Revisions or changes made to one layer may necessitate revisions to other layers as well. In some cases, a modification in one component within a layer can trigger the need to revise code in multiple layers, increasing development effort and complexity.

Layered architecture is a strong choice for IoT applications due to its organized structure, cooperative framework, and improved testing capabilities. The distinct layers allow for grouping related functionalities, making development and management more efficient. Additionally, the

separation of components within each layer facilitates better testing practices, ensuring robustness and reliability in IoT systems. While inter-layer dependencies and the potential impact of revisions across multiple layers are drawbacks, the overall benefits make layered architecture well-suited for addressing the complexities and scalability requirements of IoT applications.

5.1.6 Model View Controller Architecture

Pros:

- Parallel Development Capability
    - MVC architecture allows for parallel development of its three subsets: model, view, and controller. Developers can work on these components simultaneously, and once completed, they can link them together. This parallel approach can significantly speed up the development process.
- Localized Software Revisions
    - Modifications or revisions made to one part of the MVC architecture do not necessarily impact the entire structure. Changes can be confined to the specific part where they are needed. As long as the links between the model, view, and controller remain intact, the overall architecture does not require complete remodeling.

Cons:

- Hindrance in Parallel Development
    - Although parallel development is a benefit, it can also pose challenges. Developing all three parts concurrently may hinder communication between them until they are linked together. This can lead to instances

where one part implements something that another part does not recognize or may reject.

- Differing Structural Models for Components
    - Each part of the MVC architecture (model, view, controller) does not have to follow the same structural model. The controller's construction, for instance, does not have to mirror that of the model or view parts. This flexibility can result in code that appears disorganized or disconnected between the three components.

The Model View Controller (MVC) architecture offers significant advantages for IoT applications. Its parallel development capability allows developers to work on the model, view, and controller components simultaneously, speeding up the development process. Additionally, localized software revisions ensure that modifications to one part of the architecture do not disrupt the entire structure, enhancing flexibility and maintainability. While challenges such as hindrances in parallel development and differing structural models for components exist, the overall benefits of MVC architecture make it a suitable choice for addressing the complexities and scalability needs of IoT applications.

5.1.7 Finite State Machine Architecture

Pros:

- Defined State Transitions
    - FSM architecture ensures that transitions between states are clearly defined based on specific conditions. This structured approach makes data transmission management easier, as data follows a predetermined path without getting lost or misplaced.

- Increased Predictability
    - The mapping of data flow paths through states and their transitions enhances predictability. Knowing how data will be transmitted based on current states and transitions makes it easier to anticipate outcomes and troubleshoot issues by pinpointing where and why they occurred.
- Single Active State
    - Only one state is active at any given time in an FSM architecture. This simplicity reduces complexity in tracking data and ensures that data progresses along the correct path without confusion.

Cons:

- Complex Transitions
    - While state transitions are manageable, the potential for multiple transitions to and from states can make it challenging to keep track of all possible data flow paths. This complexity increases as the system grows.
- Scalability
    - FSM architectures may face difficulties when implemented on larger scales. Managing numerous transitions and accounting for various possibilities becomes more complex in larger projects, leading to potential mistakes and difficulty in identifying and rectifying them.

Finite State Machine (FSM) architecture is a solid choice for IoT applications due to its structured approach to state transitions and increased predictability. The defined state transitions ensure that data follows a clear path, making data transmission management more manageable and reducing the risk of data loss or confusion. The single active state feature simplifies tracking

data, contributing to a more streamlined and efficient system. However, FSM architectures may face challenges with complex transitions and scalability on larger scales, requiring careful planning and management to address potential complexities and ensure optimal performance in IoT applications.

Conclusion:

After looking at the advantages and disadvantages of each architecture, we have decided to go with the Call Return Architecture. We believe the pros outweigh the cons and there are tradeoffs that we are happy to comply with. The subprograms and hierarchical structure will greatly aid in designing the flow of our software.

5.2 Interface Design

5.2.1 Driver Interface

1. Automatic Parking Interface

- Name: AutoParkUI
- Class: AutomaticParkingSystem
- Communication Elements:
  - Software Elements: Activate automatic parking feature, coordinate communication between modules, handle preconditions and postconditions.
  - Hardware Elements: Cameras (detect parking spaces and obstacles), Planning module (calculate optimal parking path).
  - End-User Interaction: Display confirmation message upon successful parking.
- Data Displayed: Activation status, confirmation message.

2. Acceleration Launch Interface

- Name: AccLaunchUI

- Class: AccelerationLaunchSystem

- Communication Elements:

  - Software Elements: Activate acceleration launch control, coordinate communication, handle preconditions and postconditions.

  - Hardware Elements: SensorFusion (assess road conditions), Planning module (calculate torque distribution).

  - End-User Interaction: Display confirmation message upon successful launch.

- Data Displayed: Activation status, confirmation message.

3. Obstacle Avoidance Interface

- Name: ObstacleAvoidUI

- Class: ObstacleAvoidanceSystem

- Communication Elements:

  - Software Elements: Detect obstacles, analyze data for evasive maneuver, coordinate communication.

  - Hardware Elements: SensorFusion (receive obstacle data), Planning module (calculate evasive maneuver).

  - End-User Interaction: Alert for detected obstacles, status of evasive maneuver execution.

- Data Displayed: Obstacle alerts, maneuver status.

4. Traffic Sign Detection Interface

- Name: TrafficSignUI
- Class: TrafficSignDetectionSystem
- Communication Elements:
  - Software Elements: Detect traffic signs, interpret signs, coordinate communication.
  - Hardware Elements: SensorFusion (receive traffic sign data), Planning module (interpret signs).
  - End-User Interaction: Display detected signs and actions taken based on sign interpretation.
- Data Displayed: Detected traffic signs, actions based on sign interpretation.

5. Cruise Control Interface

- Name: CruiseControlUI
- Class: CruiseControlSystem
- Communication Elements:
  - Software Elements: Activate and manage cruise control, monitor vehicle conditions, coordinate communication.
  - Hardware Elements: SensorFusion (receive speed request), Planning module (calculate throttle and brake settings).
  - End-User Interaction: Display cruise control activation status, monitor vehicle conditions.
- Data Displayed: Cruise control status, vehicle condition monitoring.

5.2.2 Technician Interface

1. Automatic Parking Technician Interface

- Name: AutoParkTechUI
- Class: AutomaticParkingSystem
- Communication Elements:
  - Software Elements: Coordinate communication between modules, handle preconditions and postconditions, log data.
  - Hardware Elements: Cameras (send data to Planning module).
  - End-User Interaction: N/A (primarily for system monitoring and management).
- Data Displayed: Communication logs, system status.

2. Acceleration Launch Technician Interface

- Name: AccLaunchTechUI
- Class: AccelerationLaunchSystem
- Communication Elements:
  - Software Elements: Coordinate communication, handle preconditions and postconditions, log data.
  - Hardware Elements: SensorFusion (send data to Planning module).
  - End-User Interaction: N/A (primarily for system monitoring and management).
- Data Displayed: Communication logs, system status.

3. Obstacle Avoidance Technician Interface

- Name: ObstacleAvoidTechUI

- Class: ObstacleAvoidanceSystem
- Communication Elements:
  - Software Elements: Analyze data for evasive maneuver, coordinate communication, log data.
  - Hardware Elements: SensorFusion (send data to Planning module).
  - End-User Interaction: N/A (primarily for system monitoring and management).
- Data Displayed: Communication logs, system status.

4. Traffic Sign Detection Technician Interface
- Name: TrafficSignTechUI
- Class: TrafficSignDetectionSystem
- Communication Elements:
  - Software Elements: Interpret signs, calculate necessary actions, coordinate communication, log data.
  - Hardware Elements: SensorFusion (send data to Planning module).
  - End-User Interaction: N/A (primarily for system monitoring and management).
- Data Displayed: Communication logs, system status.

5. Cruise Control Technician Interface
- Name: CruiseControlTechUI
- Class: CruiseControlSystem
- Communication Elements:

- Software Elements: Manage cruise control, monitor events, coordinate communication, log data.

- Hardware Elements: SensorFusion (send data to Planning module).

- End-User Interaction: N/A (primarily for system monitoring and management).

- Data Displayed: Communication logs, system status.

5.2.3 UI Interface

1. Automatic Parking UI Interface

- Name: AutoParkUI

- Class: AutomaticParkingSystem

- Communication Elements:

- Software Elements: Activate automatic parking feature, display relevant information to the driver.

- Hardware Elements: Interaction with touchscreen displays, audio alerts for confirmation.

- End-User Interaction: Visual cues for parking path, confirmation message upon successful parking.

- Data Displayed: Parking path, confirmation message.

2. Acceleration Launch UI Interface

- Name: AccLaunchUI

- Class: AccelerationLaunchSystem

- Communication Elements:

- Software Elements: Activate acceleration launch control, display relevant information to the driver.

- Hardware Elements: Interaction with touchscreen displays, audio alerts for confirmation.

- End-User Interaction: Visual cues for launch readiness, confirmation message upon successful launch.

- Data Displayed: Launch readiness, confirmation message.

3. Obstacle Avoidance UI Interface

- Name: ObstacleAvoidUI

- Class: ObstacleAvoidanceSystem

- Communication Elements:

- Software Elements: Alert driver of detected obstacles, provide options for evasive maneuvers.

- Hardware Elements: Visual and auditory alerts, touchscreen interaction for selecting maneuvers.

- End-User Interaction: Alerts for obstacles, options for evasive actions.

- Data Displayed: Obstacle alerts, maneuver options.

4. Traffic Sign Detection UI Interface

- Name: TrafficSignUI

- Class: TrafficSignDetectionSystem

- Communication Elements:

- Software Elements: Display detected traffic signs, provide relevant information to the driver.

- Hardware Elements: Visual displays of traffic signs, audible alerts for critical signs.

- End-User Interaction: Display detected signs and recommended actions.

- Data Displayed: Detected traffic signs, recommended actions.

5. Cruise Control UI Interface

- Name: CruiseControlUI

- Class: CruiseControlSystem

- Communication Elements:

- Software Elements: Activate and manage cruise control, display cruise control status to the driver.

- Hardware Elements: Dashboard display of cruise control status, controls for activation/deactivation.

- End-User Interaction: Visual indicators for cruise control status, controls for adjustments.

- Data Displayed: Cruise control status, adjustments.

5.2.4 UX Interface

1. Automatic Parking UX Interface

- Name: AutoParkUX

- Class: AutomaticParkingSystem

- Communication Elements:

- Software Elements: Design intuitive user flows for automatic parking activation and confirmation.

- Hardware Elements: Ensure seamless interaction with touchscreen displays and audio feedback.

- End-User Interaction: User-friendly interface for smooth parking experience, clear confirmation messages.

- Data

Displayed: Intuitive parking process, clear feedback.

2. Acceleration Launch UX Interface

- Name: AccLaunchUX

- Class: AccelerationLaunchSystem

- Communication Elements:

- Software Elements: Design user-friendly flows for launch control activation and confirmation.

- Hardware Elements: Ensure intuitive interaction with touchscreen displays and audible cues.

- End-User Interaction: Smooth launch control activation process, clear confirmation feedback.

- Data Displayed: User-friendly launch control experience, confirmation feedback.

3. Obstacle Avoidance UX Interface

- Name: ObstacleAvoidUX
- Class: ObstacleAvoidanceSystem
- Communication Elements:
  - Software Elements: Design intuitive interfaces for obstacle alerts and evasive maneuver options.
  - Hardware Elements: Clear visual and auditory alerts, easy-to-use touchscreen for maneuver selection.
  - End-User Interaction: Easy understanding of obstacles and available evasive actions.
- Data Displayed: Obstacle alerts, easy maneuver selection.

4. Traffic Sign Detection UX Interface
- Name: TrafficSignUX
- Class: TrafficSignDetectionSystem
- Communication Elements:
  - Software Elements: Design clear interfaces for displaying detected traffic signs and recommended actions.
  - Hardware Elements: Visual representation of signs, audible alerts for critical signs.
  - End-User Interaction: Easy comprehension of detected signs and recommended actions.
- Data Displayed: Detected signs, clear recommendations.

5. Cruise Control UX Interface
- Name: CruiseControlUX

- Class: CruiseControlSystem
- Communication Elements:
  - Software Elements: Design intuitive controls and feedback for cruise control activation and adjustments.
  - Hardware Elements: Dashboard display of cruise control status, user-friendly controls.
  - End-User Interaction: Easy activation and adjustment of cruise control settings.
- Data Displayed: User-friendly cruise control interface, status updates.

5.3 Component-Level Design

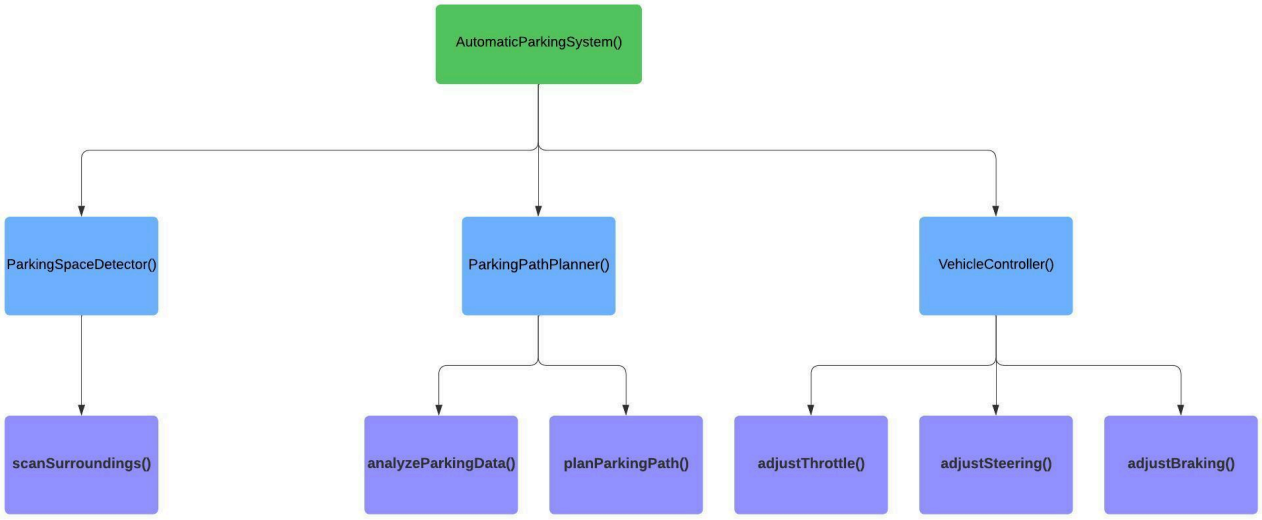
As mentioned above, we decided to move forward with utilizing the Call Return Architecture, mainly because of its code reusability and modularity. This allows us to maintain our agile development process while reducing risk management. It also offers a clean and friendly look to anyone wanting to understand the architectural designs as shown below.

5.3.1 Automatic Parking Design

Main Program: AutomaticParkingSystem()

1. Call ParkingSpaceDetector to scanSurroundings() and identify available parking spaces and obstacles.
2. Call ParkingPathPlanner to analyzeParkingData() and planParkingPath().
3. Call VehicleController to executeParking(), which:
    a. Adjusts steering with adjustSteering().
    b. Adjusts throttle with adjustThrottle().
    c. Adjusts braking with adjustBraking().

4. Monitor the parking process and make adjustments as needed.
5. Return control to the main program once parking is completed.

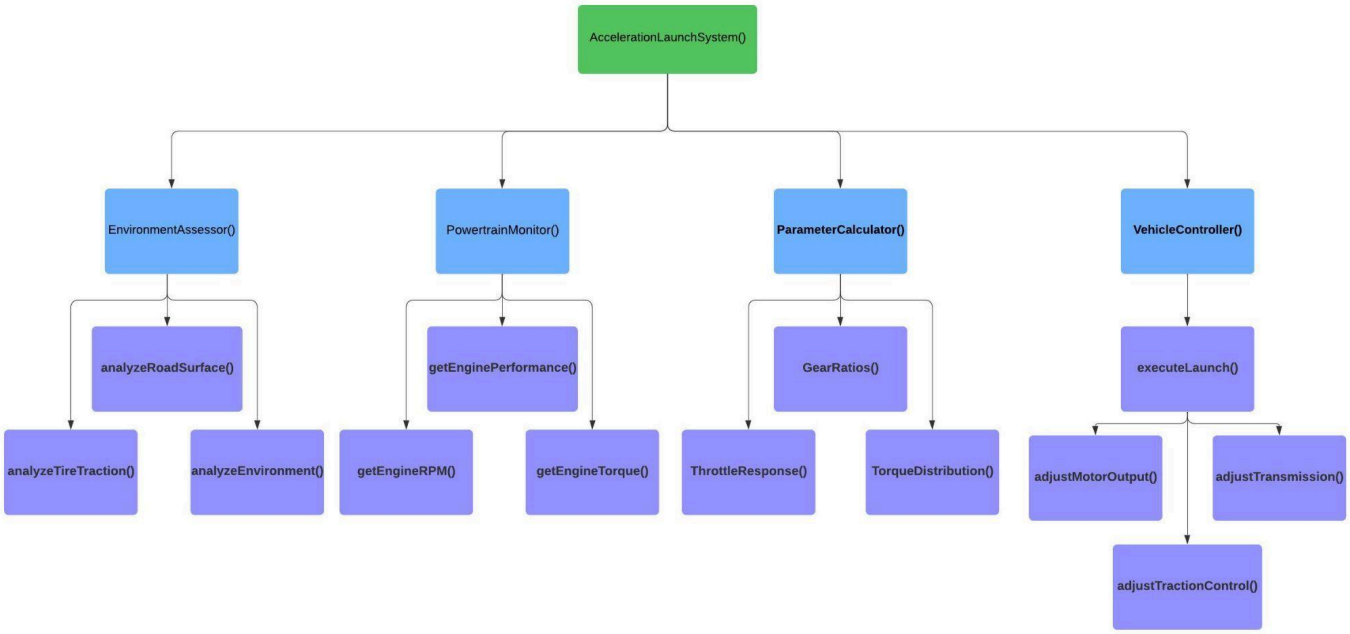


5.3.2 Acceleration Launch Design

Main Program: AccelerationLaunchSystem()

1. Call EnvironmentAssessor to analyzeRoadSurface(), analyzeTireTraction(), and analyzeEnvironment().
2. Call PowertrainMonitor to getEnginePerformance(), getEngineTorque(), and getEngineRPM().
3. Call LaunchParameterCalculator to calculateTorqueDistribution(), calculateGearRatios(), and calculateThrottleResponse().
4. Call VehicleController to executeLaunch():
    a. Adjust motor output with adjustMotorOutput().
    b. Adjust transmission with adjustTransmission().
    c. Adjust traction control with adjustTractionControl().

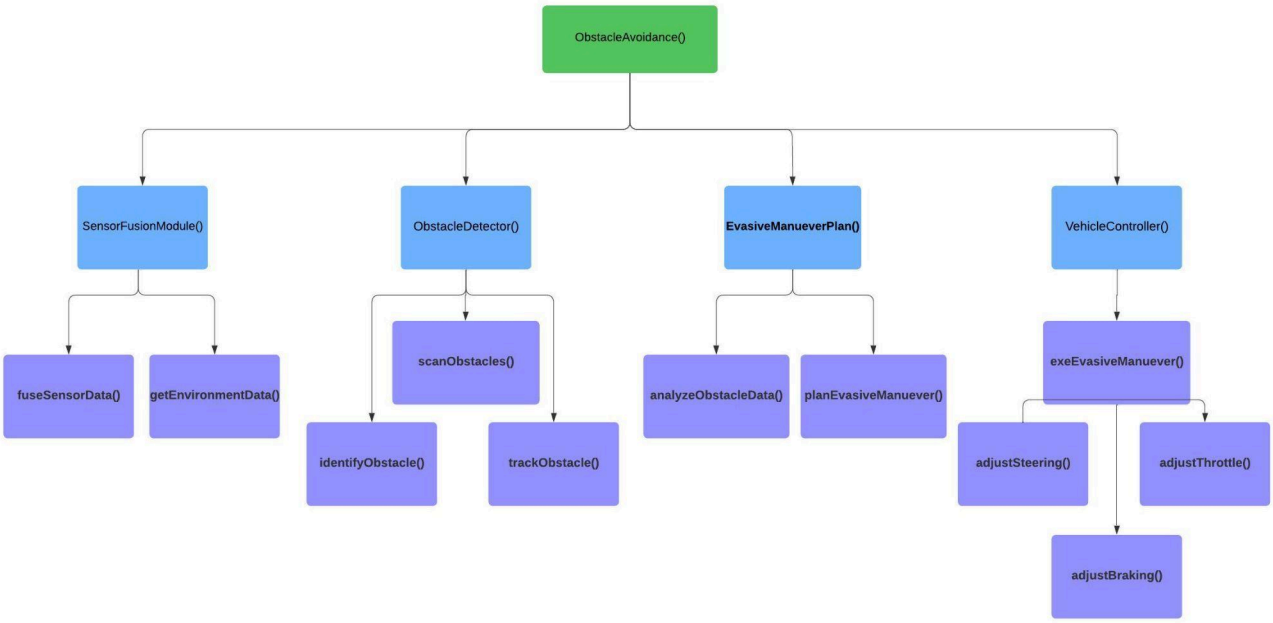
5. Monitor wheel slip and adjust torque distribution during launch as needed.
6. Return control to the main program once the desired speed/acceleration is reached.



5.3.3 Obstacle Avoidance Design

Main Program: ObstacleAvoidanceSystem()

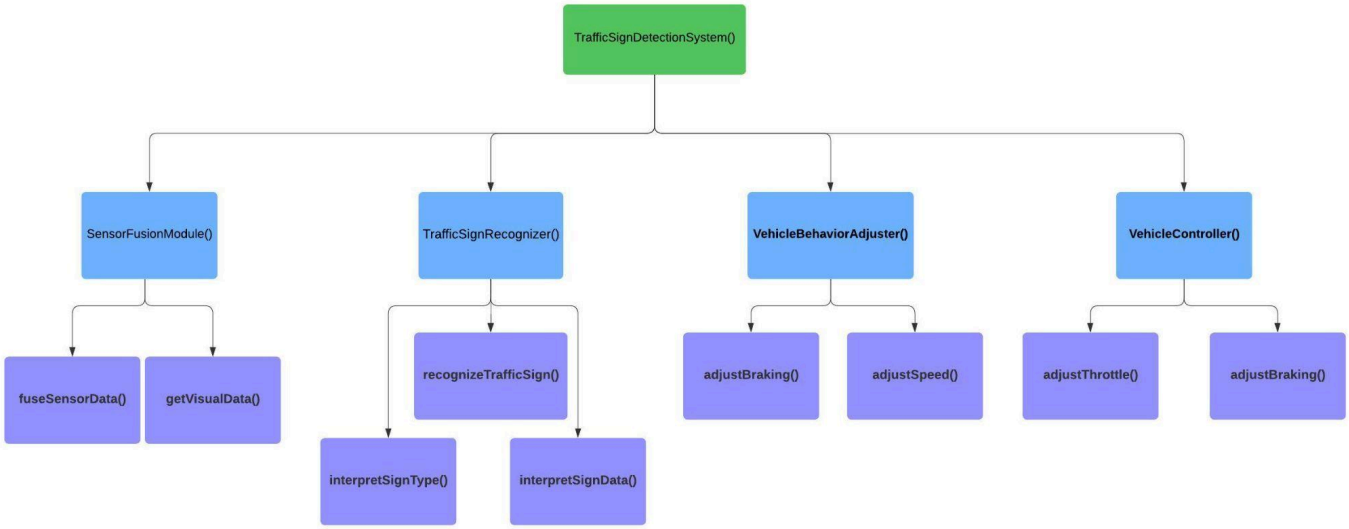
1. Call SensorFusionModule to fuseSensorData() and getEnvironmentData().
2. Call ObstacleDetector to scanForObstacles(), identifyObstacles(), and trackObstacles().
3. Call EvasiveManeuverPlanner to analyzeObstacleData() and planEvasiveManeuver().
4. Call VehicleController to executeEvasiveManeuver():
    a. Adjust steering with adjustSteering().
    b. Adjust throttle with adjustThrottle().
    c. Adjust braking with adjustBraking().
5. Monitor the evasive maneuver process and make adjustments as needed.
6. Return control to the main program once the obstacle is safely avoided.



5.3.4 Traffic Sign Design

Main Program: TrafficSignDetectionSystem

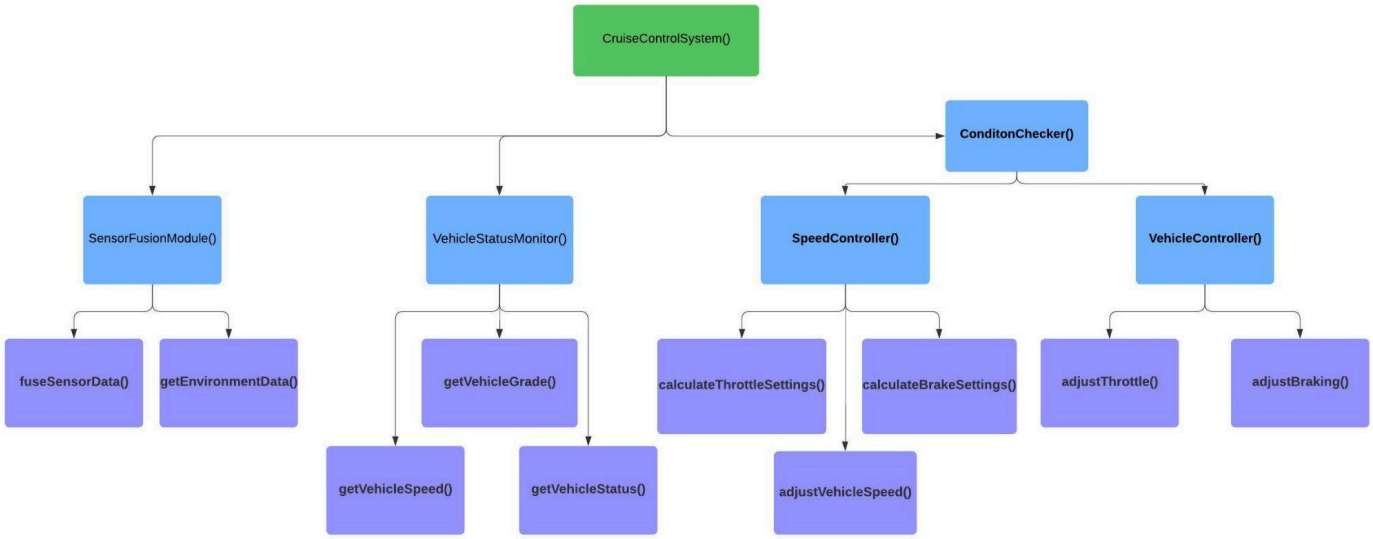
1. Call SensorFusionModule to fuseSensorData() and getVisualData().
2. Call TrafficSignRecognizer to recognizeTrafficSign(), interpretSignType(), and interpretSignData().
3. Call VehicleBehaviorAdjuster to adjustSpeed() and adjustBraking() based on the detected traffic sign.
4. Call VehicleController to adjust the vehicle's behavior:
    a. Adjust throttle with adjustThrottle().
    b. Adjust braking with adjustBraking().
5. Monitor the traffic sign detection process and make adjustments as needed.
6. Return control to the main program once the vehicle has responded to the detected traffic sign.



5.3.5 Cruise Control Design

Main Program: CruiseControlSystem

1. Call SensorFusionModule to fuseSensorData() and getEnvironmentData().
2. Call VehicleStatusMonitor to getVehicleSpeed(), getVehicleGrade(), and getVehicleStatus().
3. If conditions are met, call SpeedController to calculateThrottleSettings(), calculateBrakeSettings(), and adjustVehicleSpeed().
4. Call VehicleController to maintain the set speed:
    a. Adjust throttle with adjustThrottle().
    b. Adjust braking with adjustBraking().
5. Continuously monitor conditions and disengage cruise control if necessary.
6. Return control to the main program when cruise control is disengaged or the destination is reached.



Section 6: Project Code

6.1 IOT

```
class IoT:
    # Constructor for the IoT class
    def __init__(self):
        # Initialize the state of the driver's key, car's status, and speedometer
        self.driver_has_key = False
        self.car_is_on = False
        self.speedometer = 0

    # Method to check the preconditions for driving
    def check_preconditions(self, driver_has_key, car_is_on, speedometer):
        # Update the state of the driver's key, car's status, and speedometer
        self.driver_has_key = driver_has_key
        self.car_is_on = car_is_on
        self.speedometer = speedometer

        # Check if the driver has the key, the car is on, and the speedometer reads 0
mph
        if self.driver_has_key and self.car_is_on and self.speedometer == 0:
            return True
        else:
            return False
```

```
    # Method to initialize the systems for driving
    def initialize_systems(self):
        # Check the preconditions for driving
        if self.check_preconditions(self.driver_has_key, self.car_is_on,
self.speedometer):
            # If the preconditions are met, print a message and return True
            print("All necessary systems are initialized and operational.")
            return True
        else:
            # If the preconditions are not met, print a message and return False
            print("Preconditions not met. Please ensure you have the key, the car is
on, and the speedometer reads 0 mph.")
            return False

    # Method to check if the car is ready to drive
    def ready_to_drive(self):
        # Check if the systems are initialized for driving
        if self.initialize_systems():
            # If the systems are initialized, print a message and return True
            print("The Car is ready to drive.")
            return True
        else:
            # If the systems are not initialized, print a message and return False
            print("The Car is not ready to drive.")
            return False
```

6.2 Drive

```
class Drive:
    # Constructor for the Drive class
    def __init__(self):
        # Initialize the state of the battery charge level, gear, and system's
operational status
        self.battery_charge_level = 0
        self.gear = ""
        self.systems_operational = False

    # Method to check the preconditions for driving
```

```
    def check_preconditions(self, battery_charge_level, gear, systems_operational):
        # Update the state of the battery charge level, gear, and system's
operational status
        self.battery_charge_level = battery_charge_level
        self.gear = gear
        self.systems_operational = systems_operational

        # Check if the battery charge level is above 0, the gear is set to 'Drive',
and the systems are operational
        if self.battery_charge_level > 0 and self.gear == "Drive" and
self.systems_operational:
            return True
        else:
            return False

    # Method to start driving
    def start_drive(self):
        # Check the preconditions for driving
        if self.check_preconditions(self.battery_charge_level, self.gear,
self.systems_operational):
            # If the preconditions are met, print a message and return True
            print("Vehicle is ready to drive.")
            return True
        else:
            # If the preconditions are not met, print a message and return False
            print("Preconditions not met. Please ensure the battery charge level is
above the minimum threshold, the gear is set to 'Drive', and all vehicle systems are
operational.")
            return False

    # Method to end driving
    def end_drive(self, destination_reached, battery_charge_level):
        # Check if the destination has been reached and the battery charge level is
above 0
        if destination_reached and battery_charge_level > 0:
            # If the postconditions are met, print a message and return True
            print("Vehicle has reached the destination without any critical
performance issues. Battery charge level is above the minimum required for further
operations.")
            return True
        else:
```

```
            # If the postconditions are not met, print a message and return False
            print("Postconditions not met. Please ensure the vehicle has reached the
destination and the battery charge level is above the minimum required for further
operations.")
            return False
```

6.3 Automatic Parking

```
class AutomaticParking:
    def __init__(self, parking_button):
        self.parking_button = False

    # Activate automatic parking feature
    def AutomaticParkingSystem(self):
        self.parking_button = False
        self.motor_output = 10
        self.traction_control = 100

    # Verify car state
    def ParkedModule(self, parking_button):
        self.AutomaticParkingSystem(parking_button)
        if self.parking_button:
            print("Car is parked successfully")
        else:
            print("Car could not park successfully")

    # Detect parking spaces and obstacles
    def Cameras(self):
        return [0, 1, 0, 0, 1]

    # Find parking spaces to use
    def Planning(self):
        if 0 in self.Cameras():
            print("Parking space detected")
        else:
            print("No parking space detected")

    # Control vehicle movement for parking
    def VCS(self, motor_output, traction_control):
```

```
        self.AutomaticParkingSystem(motor_output, traction_control)
        if self.motor_output and self.traction_control:
            print("Vehicle is moving")
        else:
            print("Vehicle is not moving")
```

6.4 Acceleration Launch

```
class AccelerationLaunch:
    def __init__(self):
        self.launch_control = False
        self.torque_distribution = 0
        self.throttle_response = 0

    # Activate acceleration launch control
    def AccelerationLaunchSystem(self, launch_control, torque_distribution,
throttle_response):
        self.launch_control = launch_control
        self.torque_distribution = torque_distribution
        self.throttle_response = throttle_response

    # Assess road conditions and engine performance
    def SensorFusion(self):
        return [0, 1, 0, 0, 1]

    # Calculate torque distribution and throttle response
    def Planning(self):
        if 0 in self.SensorFusion():
            print("Road conditions and engine performance are optimal for launch")
        else:
            print("Road conditions or engine performance are not optimal for launch")

    # Control motor output, transmission settings, and traction control for launch
    def VCS(self, motor_output, transmission_settings, traction_control):
        self.AccelerationLaunchSystem(motor_output, transmission_settings,
traction_control)
        if self.motor_output and self.transmission_settings and
self.traction_control:
            print("Vehicle is launching")
```

```
        else:
            print("Vehicle is not launching")
```

6.5 Headlights

```
class Headlights:
    def __init__(self):
        self.ambient_light_level = 0
        self.headlights_status = False
        self.headlights_setting = ""

    # Method to check the preconditions for turning on the headlights
    def check_preconditions(self, ambient_light_level, headlights_setting):
        self.ambient_light_level = ambient_light_level
        self.headlights_setting = headlights_setting

        # Check if the ambient light level is below 3 or the headlights setting is
set to 'low' or 'high'
        if self.ambient_light_level < 3 or self.headlights_setting in ['low',
'high']:
            return True
        else:
            return False

    # Method to turn on the headlights
    def turn_on_headlights(self):
        if self.check_preconditions(self.ambient_light_level,
self.headlights_setting):
            self.headlights_status = True
            print("Headlights are turned on and functioning properly, providing
adequate illumination of the road ahead for the driver and other road users.")
            return True
        else:
            print("Preconditions not met. Please ensure the ambient light level is
below 3 or the headlights setting is set to 'low' or 'high'.")
            return False
```

6.6 Obstacle Avoidance

```
class ObstacleAvoidance:
    def __init__(self):
        self.obstacle_data = []
        self.evasive_maneuver = 0

    # Detect obstacles
    # Analyze data for evasive maneuver
    def ObstacleAvoidanceSystem(self, obstacle_data, evasive_maneuver):
        self.obstacle_data = obstacle_data
        self.evasive_maneuver = evasive_maneuver

    # Receive obstacle data
    def SensorFusion(self, obstacle_data):
        self.obstacle_data = [0, 1, 0, 0, 0, 1, 1]
        return self.obstacle_data

    # Calculate evasive maneuver
    # 5 represents a level 5 evasive maneuver
    def Planning(self, evasive_maneuver):
        self.evasive_maneuver = 5
        return self.evasive_maneuver

    # Control steering, throttle, and brake
    def VCS(self, steering, throttle, brake):
        if steering and throttle and brake:
            print("Vehicle is avoiding obstacle")
        else:
            print("Vehicle is not avoiding obstacle")
```

6.7 Traffic Sign Detection

```
class TrafficSignDetection:
    def __init__(self):
        self.traffic_sign_data = []
        self.necessary_actions = ["stop", "yield", "speed limit", "no entry", "no
turn", "no parking", "no stopping"]

    # Detect and interpret traffic signs
    def TrafficSignDetectionSystem(self, traffic_sign_data, necessary_actions):
```

```
        self.traffic_sign_data = traffic_sign_data
        self.necessary_actions = necessary_actions

    # Receive traffic sign data
    # Series of 1s and 0s to represent traffic signs
    def SensorFusion(self, traffic_sign_data):
        self.traffic_sign_data = [0, 1, 1, 0, 0, 1, 0]
        return self.traffic_sign_data

    # Calculate necessary actions
    # Pick a necessary action based on traffic sign interpretation
    def Planning(self, necessary_actions):
        self.necessary_actions = necessary_actions[0]
        return necessary_actions

    def VCS(self, necessary_actions):
        # Execute necessary actions based on traffic sign interpretation
        if necessary_actions:
            print("Vehicle is executing necessary actions based on traffic sign
interpretation")
        else:
            print("Vehicle is not executing necessary actions based on traffic sign
interpretation")
```

6.8 Navigation

```
class Navigation:
    def __init__(self):
        self.destination_inputted = False
        self.route_calculated = False
        self.gps_engaged = False

    # Method to check the preconditions for navigation
    def check_preconditions(self, destination_inputted, route_calculated,
gps_engaged):
        self.destination_inputted = destination_inputted
        self.route_calculated = route_calculated
        self.gps_engaged = gps_engaged
```

```
        # Check if a destination has been inputted or a route has been calculated,
and the GPS is engaged
        if (self.destination_inputted or self.route_calculated) and self.gps_engaged:
            return True
        else:
            return False

    # Method to initiate navigation
    def initiate_navigation(self):
        if self.check_preconditions(self.destination_inputted, self.route_calculated,
self.gps_engaged):
            print("Navigation initiated. Following the planned route.")
            # Code for following the route would go here
            print("Route followed accurately. Driver guided to the destination with
ease and precision.")
            return True
        else:
            print("Preconditions not met. Please ensure a destination is inputted or
a route is calculated, and the GPS is engaged.")
            return False
```

6.9 Crash Detection

```
class CrashDetection:
    def __init__(self):
        self.impact_force = 0
        self.deceleration = 0
        self.time_interval = 0

    # Method to check the preconditions for crash detection
    def check_preconditions(self, impact_force, deceleration, time_interval):
        self.impact_force = impact_force
        self.deceleration = deceleration
        self.time_interval = time_interval

        # Check if the vehicle experiences an impact registering at least 5 g-force
or a deceleration exceeding 0.5 g over 0.2 seconds
        if (self.impact_force >= 5 or (self.deceleration >= 0.5 and
self.time_interval <= 0.2)):
            return True
```

```
        else:
            return False

    # Method to initiate crash detection
    def initiate_crash_detection(self):
        if self.check_preconditions(self.impact_force, self.deceleration,
self.time_interval):
            print("Potential collision detected. Initiating automatic emergency
braking and airbag deployment.")
            # Code for automatic emergency braking and airbag deployment would go
here
            print("Automatic emergency braking engaged. Impact velocity reduced.
Airbags deployed. Emergency services notified.")
            return True
        else:
            print("Preconditions not met. Please ensure the vehicle experiences an
impact registering at least 5 g-force or a deceleration exceeding 0.5 g over 0.2
seconds.")
            return False
```

6.10 Cruise Control

```
class CruiseControl:
    def __init__(self):
        self.cruise_control_active = False
        self.speed_request = 0
        self.throttle_brake_settings = 0
        self.current_speed = 60
        self.speed_limit = 60

    # Activate and manage cruise control
    # Monitor vehicle conditions
    def CruiseControlSystem(self, cruise_control_active, speed_request):
        self.cruise_control_active = cruise_control_active
        self.speed_request = speed_request

    # Set the current speed as the speed request
    def SensorFusion(self, speed_request, current_speed):
        # If speed is less than 80, set speed request
        if self.current_speed < 80:
```

```
            self.speed_request = current_speed
            return speed_request
        else:
            print("Cannot enable cruise control at current speed")

    # Calculate throttle and brake settings
    def Planning(self, throttle_brake_settings, speed_limit):
        # If current speed is less than speed limit, set throttle
        if self.current_speed < self.speed_limit:
            self.throttle_brake_settings = 1
            return throttle_brake_settings
        # If current speed is greater than speed limit, set brake
        else:
            self.throttle_brake_settings = -1
            return throttle_brake_settings

    # Control propulsion and braking
    def VCS(self, propulsion, braking):
        if propulsion and braking:
            print("Vehicle is in cruise control mode")
        else:
            print("Vehicle is not in cruise control mode")
            self.cruise_control_active = False
```

Section 7: Testing

7.1 IOT Test

```
import unittest
from io import StringIO
import sys

class TestIoTSystem(unittest.TestCase):
    def setUp(self):
        # Redirect stdout for testing print outputs
        self.saved_stdout = sys.stdout
        self.out = StringIO()
        sys.stdout = self.out

    def tearDown(self):
```

```
        # Restore stdout
        sys.stdout = self.saved_stdout

    def test_car_ready_to_drive(self):
        # Create an instance of the IoT class
        iot_system = IoT()

        # Set the driver_has_key, car_is_on, and speedometer values to simulate
readiness
        iot_system.driver_has_key = True
        iot_system.car_is_on = True
        iot_system.speedometer = 0

        # Call the ready_to_drive method
        result = iot_system.ready_to_drive()

        # Assert that the method prints the correct message and returns True
        self.assertEqual(self.out.getvalue().strip(), "The Car is ready to
drive.")
        self.assertTrue(result)

if __name__ == '__main__':
    unittest.main()
```

7.2 Drive Test

```
class TestDriveSystem(unittest.TestCase):
    def setUp(self):
        # Redirect stdout for testing print outputs
        self.saved_stdout = sys.stdout
        self.out = StringIO()
        sys.stdout = self.out

    def tearDown(self):
        # Restore stdout
        sys.stdout = self.saved_stdout

    def test_start_drive(self):
```

```
        # Create an instance of the Drive class
        drive_system = Drive()

        # Set the battery_charge_level, gear, and systems_operational values to
simulate readiness
        drive_system.battery_charge_level = 50
        drive_system.gear = "Drive"
        drive_system.systems_operational = True

        # Call the start_drive method
        result = drive_system.start_drive()

        # Assert that the method prints the correct message and returns True
        self.assertEqual(self.out.getvalue().strip(), "Vehicle is ready to
drive.")
        self.assertTrue(result)
```

7.3 Automatic Parking Test

```
class TestAutomaticParkingSystem(unittest.TestCase):
    def setUp(self):
        # Redirect stdout for testing print outputs
        self.saved_stdout = sys.stdout
        self.out = StringIO()
        sys.stdout = self.out

    def tearDown(self):
        # Restore stdout
        sys.stdout = self.saved_stdout

    def test_parked_module_success(self):
        # Create an instance of the AutomaticParking class
        parking_system = AutomaticParking(parking_button=True)

        # Call the ParkedModule method
        parking_system.ParkedModule(parking_button=True)

        # Assert that the method prints the correct message
```

```
        self.assertEqual(self.out.getvalue().strip(), "Car is parked
successfully")

    def test_parked_module_failure(self):
        # Create an instance of the AutomaticParking class
        parking_system = AutomaticParking(parking_button=False)

        # Call the ParkedModule method
        parking_system.ParkedModule(parking_button=False)

        # Assert that the method prints the correct message
        self.assertEqual(self.out.getvalue().strip(), "Car could not park
successfully")
```

7.4 Acceleration Launch Test

```
class TestAccelerationLaunchSystem(unittest.TestCase):
    def setUp(self):
        # Redirect stdout for testing print outputs
        self.saved_stdout = sys.stdout
        self.out = StringIO()
        sys.stdout = self.out

    def tearDown(self):
        # Restore stdout
        sys.stdout = self.saved_stdout

    def test_planning_optimal_conditions(self):
        # Create an instance of the AccelerationLaunch class
        acceleration_system = AccelerationLaunch()

        # Mock SensorFusion method to return optimal conditions for launch
        acceleration_system.SensorFusion = lambda: [0, 1, 0, 0, 1]

        # Call the Planning method
        acceleration_system.Planning()

        # Assert that the method prints the correct message
```

```
        self.assertEqual(self.out.getvalue().strip(), "Road conditions and engine
performance are optimal for launch.")

    def test_planning_non_optimal_conditions(self):
        # Create an instance of the AccelerationLaunch class
        acceleration_system = AccelerationLaunch()

        # Mock SensorFusion method to return non-optimal conditions for launch
        acceleration_system.SensorFusion = lambda: [1, 1, 0, 0, 1]

        # Call the Planning method
        acceleration_system.Planning()

        # Assert that the method prints the correct message
        self.assertEqual(self.out.getvalue().strip(), "Road conditions or engine
performance are not optimal for launch.")
```

7.5 Headlights Test

```
class TestHeadlightsSystem(unittest.TestCase):
    def setUp(self):
        # Redirect stdout for testing print outputs
        self.saved_stdout = sys.stdout
        self.out = StringIO()
        sys.stdout = self.out

    def tearDown(self):
        # Restore stdout
        sys.stdout = self.saved_stdout

    def test_turn_on_headlights_success(self):
        # Create an instance of the Headlights class
        headlights_system = Headlights()

        # Set the ambient_light_level and headlights_setting to meet
preconditions
        headlights_system.ambient_light_level = 2
        headlights_system.headlights_setting = "low"
```

```
        # Call the turn_on_headlights method
        result = headlights_system.turn_on_headlights()

        # Assert that the method prints the correct message and returns True
        self.assertEqual(self.out.getvalue().strip(), "Headlights are turned on
and functioning properly, providing adequate illumination of the road ahead for
the driver and other road users.")
        self.assertTrue(result)

    def test_turn_on_headlights_failure(self):
        # Create an instance of the Headlights class
        headlights_system = Headlights()

        # Set the ambient_light_level and headlights_setting to fail
preconditions
        headlights_system.ambient_light_level = 5
        headlights_system.headlights_setting = "off"

        # Call the turn_on_headlights method
        result = headlights_system.turn_on_headlights()

        # Assert that the method prints the correct message and returns False
        self.assertEqual(self.out.getvalue().strip(), "Preconditions not met.
Please ensure the ambient light level is below 3 or the headlights setting is set
to 'low' or 'high'.")
        self.assertFalse(result)
```

7.6 Obstacle Avoidance Test

```
class TestObstacleAvoidanceSystem(unittest.TestCase):
    def setUp(self):
        # Redirect stdout for testing print outputs
        self.saved_stdout = sys.stdout
        self.out = StringIO()
        sys.stdout = self.out

    def tearDown(self):
```

```
        # Restore stdout
        sys.stdout = self.saved_stdout

    def test_vcs_obstacle_avoidance(self):
        # Create an instance of the ObstacleAvoidance class
        avoidance_system = ObstacleAvoidance()

        # Call the VCS method with all parameters True to simulate avoiding an
obstacle
        avoidance_system.VCS(steering=True, throttle=True, brake=True)

        # Assert that the method prints the correct message
        self.assertEqual(self.out.getvalue().strip(), "Vehicle is avoiding
obstacle")

    def test_vcs_no_obstacle_avoidance(self):
        # Create an instance of the ObstacleAvoidance class
        avoidance_system = ObstacleAvoidance()

        # Call the VCS method with all parameters False to simulate not avoiding
an obstacle
        avoidance_system.VCS(steering=False, throttle=False, brake=False)

        # Assert that the method prints the correct message
        self.assertEqual(self.out.getvalue().strip(), "Vehicle is not avoiding
obstacle")
```

7.7 Traffic Sign Detection Test

```
class TestTrafficSignDetectionSystem(unittest.TestCase):
    def setUp(self):
        # Redirect stdout for testing print outputs
        self.saved_stdout = sys.stdout
        self.out = StringIO()
        sys.stdout = self.out

    def tearDown(self):
        # Restore stdout
```

```
        sys.stdout = self.saved_stdout

    def test_vcs_execute_actions(self):
        # Create an instance of the TrafficSignDetection class
        detection_system = TrafficSignDetection()

        # Call the VCS method with a list of necessary actions to simulate
execution
        detection_system.VCS(necessary_actions=["stop"])

        # Assert that the method prints the correct message
        self.assertEqual(self.out.getvalue().strip(), "Vehicle is executing
necessary actions based on traffic sign interpretation")

    def test_vcs_no_actions(self):
        # Create an instance of the TrafficSignDetection class
        detection_system = TrafficSignDetection()

        # Call the VCS method with an empty list of necessary actions to simulate
no execution
        detection_system.VCS(necessary_actions=[])

        # Assert that the method prints the correct message
        self.assertEqual(self.out.getvalue().strip(), "Vehicle is not executing
necessary actions based on traffic sign interpretation")
```

7.8 Navigation Test

```
class TestNavigationSystem(unittest.TestCase):
    def setUp(self):
        # Redirect stdout for testing print outputs
        self.saved_stdout = sys.stdout
        self.out = StringIO()
        sys.stdout = self.out

    def tearDown(self):
        # Restore stdout
        sys.stdout = self.saved_stdout
```

```
    def test_initiate_navigation_success(self):
        # Create an instance of the Navigation class
        navigation_system = Navigation()

        # Set the preconditions to True to simulate successful navigation
initiation
        navigation_system.destination_inputted = True
        navigation_system.route_calculated = True
        navigation_system.gps_engaged = True

        # Call the initiate_navigation method
        result = navigation_system.initiate_navigation()

        # Assert that the method prints the correct messages and returns True
        expected_output = "Navigation initiated. Following the planned
route.\nRoute followed accurately. Driver guided to the destination with ease and
precision."
        self.assertEqual(self.out.getvalue().strip(), expected_output)
        self.assertTrue(result)

    def test_initiate_navigation_failure(self):
        # Create an instance of the Navigation class
        navigation_system = Navigation()

        # Call the initiate_navigation method without meeting preconditions to
simulate failure
        result = navigation_system.initiate_navigation()

        # Assert that the method prints the correct message and returns False
        self.assertEqual(self.out.getvalue().strip(), "Preconditions not met.
Please ensure a destination is inputted or a route is calculated, and the GPS is
engaged.")
        self.assertFalse(result)
```

7.9 Crash Detection Test

```
class TestCrashDetectionSystem(unittest.TestCase):
    def setUp(self):
        # Redirect stdout for testing print outputs
        self.saved_stdout = sys.stdout
        self.out = StringIO()
        sys.stdout = self.out

    def tearDown(self):
        # Restore stdout
        sys.stdout = self.saved_stdout

    def test_initiate_crash_detection_success(self):
        # Create an instance of the CrashDetection class
        crash_detection_system = CrashDetection()

        # Set the preconditions to True to simulate successful crash detection
initiation
        crash_detection_system.impact_force = 6  # Greater than 5 g-force
        crash_detection_system.deceleration = 0.6  # Greater than 0.5 g
        crash_detection_system.time_interval = 0.1  # Less than 0.2 seconds

        # Call the initiate_crash_detection method
        result = crash_detection_system.initiate_crash_detection()

        # Assert that the method prints the correct messages and returns True
        expected_output = "Potential collision detected. Initiating automatic
emergency braking and airbag deployment.\nAutomatic emergency braking engaged.
Impact velocity reduced. Airbags deployed. Emergency services notified."
        self.assertEqual(self.out.getvalue().strip(), expected_output)
        self.assertTrue(result)

    def test_initiate_crash_detection_failure(self):
        # Create an instance of the CrashDetection class
        crash_detection_system = CrashDetection()

        # Call the initiate_crash_detection method without meeting preconditions
to simulate failure
```

```
        result = crash_detection_system.initiate_crash_detection()

        # Assert that the method prints the correct message and returns False
        self.assertEqual(self.out.getvalue().strip(), "Preconditions not met.
Please ensure the vehicle experiences an impact registering at least 5 g-force or
a deceleration exceeding 0.5 g over 0.2 seconds.")
        self.assertFalse(result)
```

7.10 Cruise Control Test

```
class TestCruiseControlSystem(unittest.TestCase):
    def setUp(self):
        # Redirect stdout for testing print outputs
        self.saved_stdout = sys.stdout
        self.out = StringIO()
        sys.stdout = self.out

    def tearDown(self):
        # Restore stdout
        sys.stdout = self.saved_stdout

    def test_vcs_cruise_control_active(self):
        # Create an instance of the CruiseControl class
        cruise_control_system = CruiseControl()

        # Set the cruise_control_active to True to simulate cruise control active
state
        cruise_control_system.cruise_control_active = True

        # Call the VCS method with propulsion and braking True to simulate cruise
control mode
        cruise_control_system.VCS(propulsion=True, braking=True)

        # Assert that the method prints the correct message
        self.assertEqual(self.out.getvalue().strip(), "Vehicle is in cruise
control mode")

    def test_vcs_cruise_control_inactive(self):
```

```
        # Create an instance of the CruiseControl class
        cruise_control_system = CruiseControl()

        # Call the VCS method without cruise control active to simulate not in
cruise control mode
        cruise_control_system.VCS(propulsion=True, braking=True)

        # Assert that the method prints the correct message and sets
cruise_control_active to False
        expected_output = "Vehicle is not in cruise control mode"
        self.assertEqual(self.out.getvalue().strip(), expected_output)
        self.assertFalse(cruise_control_system.cruise_control_active)
```