|  |
| --- |
| C:\Users\odonnellcc\Documents\Full-Leidos-Logo-Suite\Logo Suite\Raster\PNG\INT_lds_rgb_pos.png |
| Development of Connected and Automated Vehicle Capabilities: Integrated Prototype I |
| CARMA Platform v2  Architecture |
| CoverPicture.JPG |
| Project Number: DTFH6116D00030 Task Order 13  Submitted: July 20, 2018  Version: 3 |
|  |

Table of Contents

[1. Introduction 1](#_Toc488318748)

[1.1. System Overview 1](#_Toc488318749)

[1.2. Referenced Documents 2](#_Toc488318750)

[2. Design Decisions 2](#_Toc488318751)

[2.1. Overall Design Decisions 2](#_Toc488318752)

[2.1.1. Use of ROS 2](#_Toc488318756)

[2.1.2. No Real-Time Operating System 2](#_Toc488318757)

[2.1.3. Trajectory Planning 3](#_Toc488318758)

[2.1.4. Inertial Reference Frame 3](#_Toc488318759)

[2.1.5. Electrical Power 3](#_Toc488318760)

[2.1.6. Fake Steering Driver 4](#_Toc488318761)

[2.1.7. Mobility Messages 4](#_Toc488318762)

[2.1.8. Communication Protocols 5](#_Toc488318763)

[2.1.9. Wireless Usage 6](#_Toc488318764)

[2.1.10. Physical Mounting of the Platform 7](#_Toc488318765)

[2.2. Design Trade-offs 7](#_Toc488318766)

[2.2.1. MicroAutoBox versus PC 7](#_Toc488318768)

[2.2.2. 2016 versus Older Message Standard 8](#_Toc488318769)

[3. Functional Architecture 8](#_Toc488318770)

[3.1. CAV Platform Conceptual Model 8](#_Toc488318771)

[3.2. System Behavior 9](#_Toc488318772)

[3.3. Platform Interfaces 12](#_Toc488318773)

[3.4. Logical Structure 13](#_Toc488318774)

[3.4.1. Hardware 13](#_Toc488318780)

[3.4.2. Software 14](#_Toc488318781)

[4. Physical Architecture 24](#_Toc488318782)

[4.1. Hardware Components 24](#_Toc488318783)

[4.1.1. Control System 24](#_Toc488318786)

[4.1.2. Communications 25](#_Toc488318787)

[4.1.3. Data Acquisition 25](#_Toc488318788)

[4.1.4. Power System 25](#_Toc488318789)

[4.1.5. OEM Systems 25](#_Toc488318790)

[4.2. Software Components 26](#_Toc488318791)

[4.2.1. Deployment Configuration 26](#_Toc488318792)

[4.2.2. ROS Messages 27](#_Toc488318793)

Table of Figures

[Figure 1. Top level view of the CAV Platform and guidance algorithm plug-ins 9](#_Toc488318794)

[Figure 2. Hierarchy of switches that make guidance capability usable. 11](#_Toc488318795)

[Figure 3. Platform hardware structure 14](#_Toc488318796)

[Figure 4. Top level software package diagram. 15](#_Toc488318797)

[Figure 5. Software components and internal interfaces. 27](#_Toc488318798)

# Introduction

## System Overview

Between 2014 and 2016 the FHWA’s Saxton Transportation Operations Laboratory (STOL) developed two sets of connected automation research vehicles. The Cooperative Automation Research for Mobility Applications (CARMA) fleet consisted of five 2013 Cadillac SRXs with custom longitudinal control hardware as well as a rack full of associated electronics to provide communications, positioning and guidance computations. In addition to the CARMA fleet a 2010 Ford Escape hybrid was similarly outfitted with control and electronic hardware. These vehicles successfully executed several stand-alone applications that demonstrated benefits of combining vehicle-to-vehicle (V2V) and vehicle-to-infrastructure (V2I) communications with partial vehicle automation. Key pieces of the software built for these vehicles came to be known as CARMA version 1, as they were reused (with some difficulty) among multiple applications.

In March 2017 the FHWA contracted the Leidos team to build a second generation fleet, under the STOL II contract, DTFH6116D00030, Task Order 13, Development of Connected and Automated Vehicle Capabilities: Integrated Prototype I. The second generation fleet, referred to as the CAV Platform, will reuse many of the components of the vehicles described above, but is specifically intended to be a reusable, extensible platform on which new cooperation algorithms can be tested. The CAV Platform itself will not provide sophisticated guidance logic, rather it will provide fundamental vehicle interfaces so that new guidance and cooperation algorithms can be dropped on top of it. This version of the architecture document reflects the as-built CARMA Platform after having been revised and tested over many iterations during 2017 and 2018. The system is now identified as CARMA v2 (specifically v2.7.2).

Managing automated vehicle motion involves three aspects. The first is *navigation*, which is the act of determining where the vehicle currently is with respect to the earth and with respect to the desired path of travel (its planned route). The second is *guidance*, which includes the processes of determining how the vehicle is to move from its current location to its destination. The destination and route will be handed to the guidance algorithms, and they then determine how the vehicle’s motion needs to be adjusted at any time in order to follow the route. The third aspect of automated vehicle motion is *control*, which covers the actuation of the vehicle’s physical devices to induce changes in motion (for land vehicles these are typically causing the wheels to rotate faster or slower and turning the steering wheel). Therefore, the navigation solution becomes an input to the guidance function, and the guidance solution becomes an input to the control function. As the vehicle moves, the navigation function constantly needs to update its solution and the cycle iterates as quickly as necessary to produce a smooth and accurate vehicle motion. The required rate of iteration is largely determined by the expected speed of the vehicle.

The architecture described here is vehicle-centric. That is, it addresses the construction of a CAV Platform, of which there may be several copies built, that is to be installed in a single Saxton Lab research vehicle. There are several components in this platform that need to communicate with each other; much of this inter-component communication will use the Robot Operating System (ROS) framework, but ROS communications are restricted to the network within a single vehicle. The platform will also enable communications between vehicles (V2V) or between the host vehicle and various roadside infrastructure components (V2I) using other communication protocols, such as DSRC messaging.

This document describes the architecture for CAV Platform version 2. There are forward looking notes throughout, but they are not normative.

## Referenced Documents

SAE J1939, Recommended Practice for a Serial Control and Communications Vehicle Network, revised Oct 2007.

SAE J2735, Dedicated Short Range Communications (DSRC) Message Set Dictionary, revised March 2016.

SAE J2945/1, On-Board System Requirements for V2V Safety Communications, issued Mar 2016.

CAV Platform v1.0 Requirements Specification, version 1, released June 22, 2017.

# Design Decisions

## Overall Design Decisions

### Use of ROS

The Robot Operating System (ROS) is a flexible framework for writing robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms. Because of its stability, reliability, and reusability, ROS has become a popular open source robotics framework, and is widely used for automated vehicle development. Many Urban Challenge vehicles, BMW, Toyota, and other research teams and manufacturers are using ROS to build their own automated vehicles. ROS consists of open source libraries and applications to support the development and operation of robotic systems. ROS enables the construction of discrete and reusable nodes that serve a single function (sensing, navigation, control, etc.) within the robotic system. This node/network structure allows for sharing and reuse of code between projects and organizations by providing a well-defined interface for how each node interacts with the rest of the network. With the research goals of the project in mind we have chosen ROS as our platform of choice as a way to maximize shareability, reusability, and adaptability of the code base as well as providing a platform and environment that outside researchers wishing to extend our platform may be familiar with.

### No Real-Time Operating System

Initially the usage of a real-time operating system (RTOS) seemed necessary for the main guidance computer to ensure vehicle safety. Our choice of ROS as a platform conflicted with this notion, however, due to the fact that ROS itself sits atop the Ubuntu Linux operating system, which no longer offers a real-time version. ROS as a system is designed to perform without need for real-time scheduling capabilities. Furthermore, the rigid restrictions required for deterministic timing in an RTOS environment would greatly increase the cost of our software development process while only promising reliability gains for the few modules that required these real-time features. (The system will incorporate real-time devices for actuator control and sensor interfaces.) Due to the presence of trained safety drivers in our experimental vehicles as the ultimate arbitrator of safety (these vehicles will never be driven by the general public), we have decided against utilizing an RTOS as our operating system of choice and have instead opted for a standard Ubuntu Linux distribution. We feel confident that the combination of experienced driver intervention, non-real time safety features, and a rigorous engineering development process provides a practical mitigation to the concerns of safe vehicle operation.

### Trajectory Planning

We use the term *trajectory* to refer to the motion of the vehicle (in both longitudinal and lateral dimensions) over a fairly short future that can be planned tactically, as opposed to an entire route that satisfies our overall strategic goal. (Full route planning is outside the scope of this project – the platform will be designed to read in preplanned route data created by some external mechanism.) The creation of tactical trajectory plans is quite complex, even assuming that all of the sensor data has already been gathered, fused, and interpreted. The desirable software solution that would satisfy a wide range of complex real-world scenarios is not going to be achievable in the near term, as it is well beyond the reach of our staffing resources. Further, the team believes it is not necessary, because the goal of this platform is to support an R&D environment that emphasizes cooperation and not automation. Many commercial manufacturers have already solved these automation problems so it would be an unwise investment of FHWA resources to attempt to create similar solutions from scratch. The R&D scenarios that this platform will face can be much more controlled so that chaotic traffic events seen in real world driving situations will not need to be handled in this version of CARMA. Accepting this limitation has allowed us to produce a much simpler architecture that allows trajectories to be planned piece-wise without consideration of overly involved decision trees and cost trade-offs.

### Electrical Power

Because the platform will draw power from the main vehicle battery, the power budget is limited to what can be provided by the stock vehicle. This is based on the capacity of both the alternator and the battery. A power analysis was previously done for the existing equipment, which will be present in the updated architecture. The total power consumption would increase with the addition of an Advanced Distributed Modular Acquisition System (ADMAS) unit. However, this can be offset by a reduction in the available power for research laptops. With this reduction, the power consumption of the updated platform would remain the same. Table 1 shows the power budget for the platform, with the research laptop allocation already adjusted downward to accommodate the ADMAS. There is a MicroAutoBox shown in the component list, as it needs to be present to support legacy projects; it is not considered part of the new CAV Platform.

Table 1. Electrical Power Consumption

|  |  |
| --- | --- |
| Component | Average Power Consumption (W) |
| Guidance PC | 35 |
| CACC Lights | 30 |
| DSRC Radio | 25 |
| MicroAutoBox | 25 |
| Access Point | 15 |
| PinPoint | 12 |
| Touchscreen HMI | 10 |
| Video Logger/Encoder | 10 |
| Ethernet Switch | 9 |
| Longitudinal Controller ECUs (Qty 2) | 10 |
| Cameras (Qty 4) | 12 |
| CAN Data Logger | 5 |
| ADMAS | 25 |
| Optional Researcher Laptops | 225 |
| TOTAL | 448 |

### Fake Steering Driver

The platform version described here will only provide automated longitudinal control, but some of the vehicle actions (e.g. lane merge) will require lateral motion to be planned as well. These lateral motions will be executed by the human driver. In an effort to make the platform software more easily extensible to automated lateral control in the future, the architecture will provide a facility for the guidance plug-ins to provide plans and commands for lateral motion. These commands will be passed to the controller interface the same as longitudinal commands, so that the guidance software will have no idea whether control is being provided by a human or an automated device. For the current version, there will be a steering driver that accepts commands from the controller interface and simply reroutes these commands to the Operator UI. The driver UI will then present these commands to the vehicle’s human driver (operator) as required maneuvers (e.g. displaying a left arrow to indicate a request to change lanes to the left). It will, of course, be up to the human driver to accomplish the commanded maneuver in an appropriate time.

### Mobility Messages

Since the platform will be used largely for research in cooperative maneuvering, a central function of the platform and its guidance plug-ins will be to negotiate these maneuvers with nearby entities (e.g. other vehicles) to avoid crashes and to maximize efficiency of movement (fuel consumption and roadway occupancy). The current SAE communication standards provide messages, in particular the BSM, that only support communication of current and past vehicle state information.

State information is necessary, but far from sufficient, to enable negotiation of a cooperative future event. Negotiations are going to require messages that describe each entity’s intended strategies and specific maneuver sequencing. They will also require messages that acknowledge receipt of such messages and allow the actual negotiation process to proceed where multiple suggestions are made among various entities. These negotiations may involve voting, expressing agreement, disagreement, counter-proposals, and final acceptance or rejection of a given proposal. They will also require plan timing information, including expiration of the plans themselves and also of tentative agreements and proposal information. Finally, they may require messages or message content that can ensure privacy of each particular conversation, such as passing entity identity information, moving the conversation to another channel or medium (which would have the side benefit of freeing up bandwidth on the safety channel), and agreeing on encryption details. Some of these uses are not implemented yet, but the architecture has been devised to accommodate easy extension to add them if they become appropriate. This is expected to be a big area of future research, so we expect the concepts here to be fluid for some time.

None of the existing standards support these kinds of messaging, which are necessary for cooperative mobility. Therefore, the project team will need to invent the protocols from scratch. A delicate balance will be necessary to produce a protocol that will be reasonably reusable in future platform versions, while recognizing that we have neither the charter nor the staffing to produce a fully generalized and standardized protocol. The resulting message protocol will therefore likely be fairly lean and fragile, and subject to frequent changes as we learn more about the bare minimum that is needed to support our current and obvious near-term objectives.

### Communication Protocols

There are three potential communication media considered for use with the new platform to enable messaging among transportation system entities. Each is discussed here.

The first, and most obvious, option for communication medium is DSRC. This will be the predominant medium for use in the new CAV Platform. The platform will receive and transmit several message types according to the SAE J2735 standard, version 2016, to include the BSM, SPAT, MAP and various mobility messages. DSRC messages are broadcast over the 5.9 GHz band to all entities within range (up to about 300 m). Because it is a broadcast medium, it is ideal for the process of discovery of new entities near the host vehicle. Once new entities are discovered, private messaging can then be initiated if desired. In addition to the standard message types defined in J2735, this platform will need to implement several new message types to support the mobility communications described in the previous section. The project team will create and document formats for these new messages in the style of current J2735 message formats with the hope that our work may influence the standards bodies in their development of future versions.

The second option for communication medium is cellular (3G, 4G or 5G). Cellular communication is inherently point-to-point in that each message must be sent to a single, specific receiver. However, depending on the service coverage area and sophistication of the service and the transceivers being used (3G, 4G or 5G), available bandwidth will be orders of magnitude larger than the needs of anticipated TFHRC experiments. In a V2X environment it is best suited to a subset of mobility messages, and will be possible only after the participating entities have discovered each other (e.g. through initial DSRC mobility exchanges) and agreed to share their cellular addresses, then make the cellular connection. As a result, the existing cellular network is not appropriate for situations where a message to a new recipient is urgently required (e.g. merging onto a freeway from a short ramp or changing lanes to avoid a collision). The broadcast discovery process, agreement to switch to cellular comms, exchanging addresses, then establishing that cellular link (essentially placing a phone call), will be a lengthy process that could take several seconds. Once such a connection is established, however, it would be an efficient medium for a relatively long-standing conversation, such as ongoing negotiations and monitoring of a platooning even that may be in place for several minutes or even hours.

The proposed 5G V2V standard would be appropriate for this work. Its proponents advertise its capability for direct link between vehicles (bypassing the cell tower), one-to-many communication, extended range, and higher speed. However, the standard has not yet matured to a point where its use would be feasible for this project.

The CAV Platform to be built under this Task Order will incorporate legacy Speed Harmonization code that uses cellular communications between each participating vehicle and the infrastructure server, since there was no guarantee of sufficient RSUs on the test route to provide adequate DSRC coverage, and because the current algorithm generates unique messages for each vehicle. Therefore, the first experiments using the new platform will use the cellular medium to some extent. The messaging currently implemented for Speed Harmonization comprises a variety of mobility messages, for which no standards have been defined.

The third, and final, option for communication medium is Bluetooth. This is another point-to-point medium, following the Bluetooth SIG standard, that requires several seconds to establish a connection once the participants have discovered each other by other means and exchanged address information. It uses the 2.4 GHz radio band and is typically limited in range of no more than 10 m, except for some industrial devices, which could have a practical range up to 30 m. Because of the relatively slow connection time and short range, the Bluetooth medium is impractical for extra-vehicular communication, and is not considered further for this project.

### Wireless Usage

It is not possible to accurately quantify the amount of traffic expected on this medium, even within the controlled TFHRC experimental environments, as each message type has variable content, and experimental situations will dictate which messages are needed at what times. However, we can estimate a ballpark bandwidth usage in the following way. Each experimental vehicle will be emitting BSMs at 10 Hz, and a typical BSM can be expected to be 45 bytes (part 1 only). Where BSM part 2 is used the size could increase to several hundred bytes. On top of the message content will be WSMP and transport header information, which will add about 100 more bytes per message on the airwaves. In addition, the host vehicle will receive BSMs from each of the surrounding vehicles at 10 Hz. Anticipated THFRC experiments will involve no more than six vehicles, so a host vehicle will receive from five others. Therefore, total BSM traffic during an experiment will probably be in the neighborhood of 9 kB/sec.

When in the vicinity of a signalized intersection equipped with RSU, the vehicle will receive SPAT messages at a rate of 10 Hz, whose size is probably going to be less than 1000 bytes, depending on the intersection geometry and phase complexity. That intersection will also emit MAP messages at 1 Hz, whose size will probably be around 1000 bytes, depending on the intersection geometry. WSMP and transport headers apply to these messages also, so intersection-specific traffic will account for roughly 12 kB/sec for each intersection in the experiment. Some experiments may involve TIMs or similar messages, which vary in length depending on their contents, but may average on the order of 400 bytes, and could be emitted from roadside infrastructures at varying rates. If we nominally say there are two RSUs transmitting TIMs at 1 Hz, then we can expect that experiment to see 400 B/sec of TIMs. WSA messages are sent by the RSU to advertise services that are available on channels other than the control channel. These also vary in length depending on their contents, but may average on the order of 400 bytes and are sent at 10 Hz. We can expect to see about 4 kB/sec of WSAs.

Finally, there will be sporadic and indeterminate numbers and types of mobility messages exchanged between vehicles. Since these formats haven’t been defined yet, an estimation of their bandwidth needs is very rough and preliminary. We also consider that the mobility messages will only be broadcast during a maneuver. With that caveat, we anticipate five situations requiring inter-vehicle negotiations (e.g. a lane merge maneuver), and each situation will use mobility messages of about 200 bytes (including WSMP and transport headers) at a maximum rate of 10 Hz. Therefore, each negotiated situation may require a peak of about 2 kB/sec. So peak DSRC traffic can be expected to be in the ballpark of 28 kB/sec (9 kB/sec for BSMs, 12 kB/sec for intersection messages, 0.4 kB/sec for TIMs, 4 kB/sec for WSA, and 2 kB/sec for mobility message) during a TFHRC experiment.

### Physical Mounting of the Platform

The existing platform hardware has been designed to fit specifically into the 2013 Cadillac SRX. This package was designed to be minimally intrusive by matching the design of the interior and minimizing the footprint. However, this package is not expected to be appropriate for other vehicles in the CARMA fleet, particularly the heavy truck in which the platform hardware will be installed in the course of this Task Order. In the case of the heavy truck, multiple compartments exist to house the platform hardware so the arrangement is very flexible. However, to accommodate other vehicles that may be added to the fleet in the future, the hardware will be packaged in such a way as to minimize its footprint and promote portability. With this in mind, the platform will be installed on a panel of a suitable size that can easily be transferred between vehicles.

The installation on each vehicle will depend on several characteristics of the vehicle including the available space and the optimal location for splicing into the CAN bus. To the extent possible, cabling will be routed behind trim panels and out of sight of occupants.

## Design Trade-offs



### MicroAutoBox versus PC

Previous work on the CARMA fleet of research vehicles at the Saxton Lab used a MicroAutobox-II (MAB), from dSPACE, hardware platform as the primary computational device, in concert with a standard PC as the secondary computer. The MAB enabled us to execute Simulink models directly inside the vehicle and control the vehicle and it’s communications with that model. While this was a successful integration, it proved challenging from both a configuration management perspective and from a software abstraction perspective. Given the low-level bare metal nature of the MAB hardware, configuration management suffered due to the lack of ability to easily create, load, edit, and swap configuration files to adjust the behavior of the software. In addition, with the MAB being only able to store one program at a time due to its heavily constrained RAM and storage it was difficult to swap applications in and out of the platform. From a software perspective, Simulink lacks the necessary functionality for highly abstract software development. Being unable to easily define interfaces and swap component implementations means Simulink software can be difficult to expand and continue developing as needs and requirements change. Finally, there are steep costs for maintaining the necessary Simulink licenses in order to use the MAB. For these reasons we have chosen to move all core software activity to the standard Linux PC inside the vehicle instead of the MAB.

Note that the MAB will remain in the Cadillacs for at least the remainder of 2017 since other projects depend on that legacy architecture.

### 2016 versus Older Message Standard

Legacy work on Saxton Lab projects have used the SAE J2735 message standard from 2015 and from 2009. In one project a pre-2009 FHWA-specific format was used. There are many significant differences among these versions of the standard, and it will be impractical to attempt to support all of them. In fact, it may be impossible to leave the legacy support as is, given that all of the legacy capabilities need to be integrated into this new platform and be capable of operating simultaneously. Since any two of these capabilities may require messages in different formats, they could not possibly work together. Therefore, the various components must be upgraded to all speak the same language. Since a major revision of J2735 was released in 2016, as the most forward-looking revision yet, it is logical to bring everything up to this version as the commonly supported standard going forward. When a new revision of the standard is available, it will then be much simpler to migrate everything to it at once, since the communications capability will be isolated in only a small area of the CAV Platform.

# Functional Architecture

## CAV Platform Conceptual Model

The CAV Platform essentially provides a “middleware” capability that abstracts the vehicle into a relatively simple application programming interface (API) that research algorithms for vehicle guidance can use. The CAV Platform will provide guidance, navigation and control (GNC) functionality, and connect directly to the vehicle hardware to execute low level control commands and to ingest a variety of vehicle situation data, including a raw navigation solution (position data) from an external global navigation satellite system (GNSS) receiver. It will also connects to add-on hardware, such as environmental sensors and communication devices. It will then provide a well-defined API for abstract guidance algorithms to attach to so that those algorithms don’t need to be concerned with the details of vehicle implementation. In between this guidance API and the vehicle hardware, the CAV Platform will provide computational services such as data logging, sensor fusion, roadway geometry interpretation, and message formatting for communications with other cooperating entities. The platform will also provide an Operator user interface (UI), which will allow the vehicle’s human driver to see various system status information and issue commands to the platform. These relationships are illustrated in Figure 1. The interfaces are described further in Section 3.3 below.

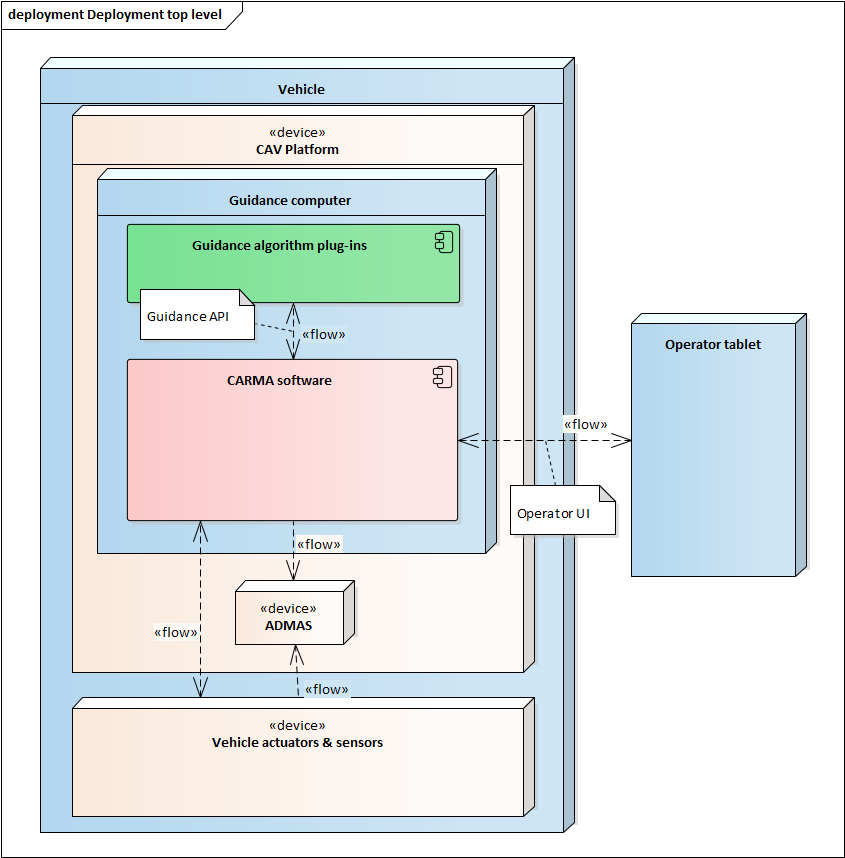


Figure 1. Top level view of the CAV Platform and guidance algorithm plug-ins

## System Behavior

The term *sensor* is used here to refer to a device or subsystem that can determine something about the environment outside the vehicle (while there are certainly many sensors for vehicle internal state, such as brake pedal position, those are not part of this definition). Sensors fall into various categories, such as object sensors, lane line sensors. The current version of the CAV platform will only support object sensors, but the architecture is set to accommodate a straightforward extension to other types of sensors in the future.

The following paragraphs describe several behavioral basics about the CAV platform.

The CAV Platform will come with some basic guidance algorithms plug-ins that are pre-installed. These will be able to take over if the custom plug-in algorithms become unable to perform.

Plug-in algorithms can be installed by an administrator at any time, but the platform software will need to be restarted to recognize them.

On system start-up the platform performs a power-on self-test (POST) to assess connectivity and functionality of each supported component. (The original intent was to have the CARMA software launch at system power-up, hence the term POST, but the team later decided that, due to the research nature of the system, it is better to manually start the software when ready to use. But the name POST was already promulgated.) At this time the platform will also look for installed guidance algorithm plug-ins and will recognize each one that initializes cleanly upon receiving a start command from the plug-in manager. It will then summarize the vehicular resources available for the guidance plug-ins to use. Each of these plug-ins can then decide whether it can be made available to the vehicle operator, based on the vehicular functions it needs (e.g. if forward object detection hardware is not operating properly then a platooning plug-in may decide that it cannot operate).

Once powered up and POSTs have completed, the available platform software functions will remain available to whatever guidance algorithms are installed until system processes are shut down (note, however, that the hardware they connect to may become faulty at any time, but the connected software driver should be able to report that situation).

Despite certain advantages of having the CARMA software always on and ready to accept commands, there are several reasons not to provide this capability. First, on the Cadillac vehicles the electronics need to be powered up prior to starting the vehicle ignition, which would mean that all software device drivers would find inoperable devices and report a failed POST. Second, in the ROS environment, it is normal to collect special log files, called rosbags, which are quite large, and which would be collected the entire time the system is on. During testing we often have vehicles sitting idle for long periods of time between test runs, which would generate enormous amounts of useless log data. Finally, attempting to work around these problems and sequence a proper startup at system boot time would require additional engineering effort that was better directed toward other considerations during v2 development. Therefore, the platform software is manually started by the vehicle operator just before it is ready to be used, and can be manually stopped at any time afterward.

The automated guidance capability may be engaged or disengaged by the platform software (normally as an incoming operator command via the Operator UI). If it is disengaged, the vehicle will be fully under human control. If the guidance capability is engaged, then the platform will check each of the recognized guidance plug-ins. Each one can be flagged as active or inactive independently by the platform software (normally as an incoming operator command via the Operator UI, for example, the operator only wants to test one particular guidance package so de-selects all others). If a particular plug-in package is marked inactive then it is dormant, performing no computations until it is made active again (by the operator selecting it on the UI). If guidance is engaged and a particular plug-in package is marked as active, then that plug-in is responsible for monitoring the vehicle configuration for resources it needs and monitoring the environment to determine if the vehicle is in a situation where it is relevant (e.g. a signal approach guidance package is only relevant on an arterial road in the vicinity of a traffic signal). If all configuration and environment conditions are appropriate, the plug-in will mark itself as available. Each available guidance plug-in package will generate updated candidate trajectory plans optimized to satisfy its own goals whenever requested by the arbitrator. These candidate trajectories are provided to the arbitrator package (described below) for consideration in formulating the vehicle’s final trajectory plan. This hierarchy of guidance capability is shown in Figure 2 below.



Figure 2. Hierarchy of switches that make guidance capability usable.

The platform will require a route to be defined before guidance is engaged, with a precise start point and end point and an approximate line of travel in between. Since, in the current version, the vehicles will have lateral control provided completely by human drivers, there is no need to specify detailed lane instructions, but waypoints defining the centerline of the roadbed will normally be sufficient. The geometry package will overlay the navigation solution onto the selected route, and it will then monitor accumulated distance travelled along the defined route since the starting point. We call this the *downtrack distance*. It will project the vehicle’s position onto the straight line segments between waypoints and compute distance travelled based on the projected distance along those line segments. In this way variations in lateral position from trip to trip will not affect the computed distance travelled. Guidance calculations can then be based on downtrack distance, under the assumption that *crosstrack error* (lateral distance from the straight line segments) stays within a reasonably small bound.

Since the world of surface transportation is naturally a chaotic environment, we cannot expect the guidance algorithms to be able to formulate a usable plan far into the future. Therefore, the platform will use the concept of a *planning horizon*. The current planning horizon will extend for a defined amount of distance downtrack from the vehicle’s current position, which will vary based on recent ability to follow planned trajectories (typically approximating about 10-30 seconds of travel). At the beginning of the route and whenever the current plan is found to be no longer useful, guidance algorithms will be given a new planning horizon and be asked to create a trajectory plan that covers only that span of roadway. They may choose to plan a bit further (e.g. if the vehicle will need to exit the highway at a ramp that is just beyond the planning horizon then the algorithm may want to use that ramp requirement to formulate the current trajectory plan).

One of the goals of Task Order 13 was to implement an Integrated Highway Prototype capability, which involves performing platooning, speed harmonization and ramp merge simultaneously in a freeway situation. The idea is for a platoon to operate under control of a speed harmonization server, which is sending speed commands to the platoon’s lead vehicle to slow it’s approach to heavy traffic, and have an additional vehicle merge into (the end) of the platoon as it enters the freeway, becoming an active member of the platoon. Each of these activities is controlled by a different Guidance plug-in. To make these activities happen smoothly, the Arbitrator needs to coordinate the responsible plug-ins so that they don’t produce conflicting commands. From an arbitration point of view, this is a simple scenario, but it is designed to exercise the arbitration capabilities. In the lead vehicle, for example, the Arbitrator will have to recognize that the speed harm plug-in needs priority over the cruising plug-in and therefore allow speed harm to construct its part of the trajectory first. Similarly, in the merging vehicle its Arbitrator will be juggling three plug-ins: ramp merge, cruising and platooning. Each of these will take precedence at different times during the merging scenario, and it is important to prioritize them so that the trajectory is built with the correct maneuvers to stitch the scenario together as an operator would expect.

Once the arbitrator has formulated a final trajectory plan for the current planning horizon, the core guidance package will translate that plan into sequences of commands for the vehicle controllers, which will actuate the vehicle hardware to achieve the desired motion. The guidance package will continuously monitor the actual sensed trajectory and compare it to the trajectory plan. If it observes a significant deviation from plan it will alert the arbitrator that a new plan is needed, and the arbitrator will request updated plans from each of the currently active guidance plug-in packages, and will start the cycle again. This iterative approach will continue until the guidance capability has been disabled. Disabling could be the result of an operator command or when the native guidance package recognizes that the chosen route has been completed.

## Platform Interfaces

The platform hardware is powered by the vehicle electrical system. To accomplish this, a power cable is run from the battery in the engine compartment to the supply side of a circuit breaker in the rear of the vehicle. For safety, this power cable is fused at the battery. The circuit breaker supply side is also connected to a battery charger and a power meter. The circuit breaker output is connected to an inverter and a pair of power distribution modules with built-in solid-state circuit breakers. A ground distribution terminal provides the return for each device and is grounded locally to the vehicle chassis to complete the circuit.

Vehicle control occurs through the CAN bus. To provide the platform hardware with access to the CAN bus, the appropriate CAN bus wires are cut and connected to the TORC low-level controller. Unpowered, the low-level controller is removed from the CAN bus and the vehicle operates in its stock configuration. When the low-level controller is activated, it inserts itself into the CAN bus, isolating the Active Safety Control Module from the rest of the CAN bus and enabling longitudinal control by the CARMA platform.

There are three external data interfaces to the CAV platform, as shown by the labeled “flow” arrows in Figure 1, above. Each is described here.

The Guidance API is a collection of Java interfaces that belong to the carma.guidance Java package. The intent is for each plugin to be a package built into a single jar file, making references to the API as needed. If properly formed, Guidance’s plugin manager component will find and link to all existing plugins and execute them according to the rules laid out here and further described in the Guidance package detailed design. This API was specifically design so that plugin authors will not have to deal with constructing direct ROS interactions or use ROS code in their plugins, which allows much simpler plugin design and easier management of the API. Note that plug-ins may need access to Java objects that represent individual ROS messages, however, to access the content of those messages as they pass through the Guidance PubSub interface.

The Operator UI is a web site intended to be viewed on the vehicle’s dashboard Android tablet. The site will be built from HTML and JavaScript. It will provide a variety of information that the vehicle’s operator will need in order to initiate and monitor the automated motion. The JavaScript will also capture commands from the vehicle operator, via tablet interactions. The ROS Bridge Server will then translate these commands into ROS messages that can be used by the Guidance package.

Also envisioned, but not yet implemented, are a pair of extensions to the web server. First is the idea of an Observer UI, which would display information nearly identical to what is available on the Operator UI, but would not allow commands to be sent from the tablet back to the web site. Having this restriction would allow any number of people (e.g. riders in the back seat or chase vehicle) to observe what the vehicle’s operator is seeing without the risk of accidentally issuing commands to the vehicle. Second is the addition of websockets that would stream data that is being displayed on the web site so that an external system could subscribe and capture that data for later analysis. These two additions have yet to be implemented as they have not been a high priority so far.

In addition to these two external interfaces, the three internal data flows shown at the bottom of Figure 1 bear some discussion. The flow between the CAV Platform software and the vehicle actuators & sensors is going to be different for each vehicle type, as this is where the software is customized to the OEM interfaces provided. These will typically involve CAN messages customized by the vehicle manufacturer, but may involve Ethernet messages to custom actuators. On the heavy truck the CAN messages will follow the SAE J1939 specification. Messages to and from the DSRC radios will be formed in accordance with the SAE J2735 specification. Content of these messages will conform to the coordinate systems and semantics defined in SAE J2945/1; accuracies will be whatever the provided vehicle and attached equipment are capable of, and may not meet the J2945/1 requirements. The CARMA side of these interfaces will be implemented by a set of device drivers, each customized for the particular vehicle hardware being used.

The ADMAS is a passive collection device that simply copies the contents of any message traffic on the networks to which it is connected. In this case, data from the CAV Platform software will be Ethernet traffic, consisting largely of the ROS messages being passed among the nodes inside the software, including all of the messages to and from the guidance plug-ins. In addition to the in-vehicle Ethernet, the ADMAS will be connected to all of the CAN busses used by the platform (e.g. vehicle high-speed bus for engine data and sensor bus for radar data), and will be able to read all messages generated by those connected devices.

## Logical Structure



### Hardware

The platform hardware is a network of devices, shown in Figure 3, that supply the necessary functions to implement the CARMA platform. The CARMA software resides on the control PC and interacts with the other hardware through Ethernet and CAN. Longitudinal control is enabled by the TORC low-level controller. This acts as an interface between the guidance algorithms on the PC and the vehicle and issues the appropriate commands to control the speed of the vehicle. This controller also interfaces with the light bar, which allows it to be controlled by the CAN bus.

An onboard unit (OBU) receives and transmits DSRC messages. A 4G router/modem receives and transmits data over the cellular networks. A GNSS unit receives position and timing information from GPS satellites and computes a navigation solution. Each of these devices communicates with the control PC via Ethernet switch.

The ADMAS data acquisition system is installed on each communication bus (CAN and Ethernet) to collect and log experimental data for subsequent analysis. Other data acquisition components include a CAN logger, which can be used as needed.



Figure 3. Platform hardware structure

### Software

The logical structure of the CARMA software is a view of the various software modules and their interactions. It does not attempt to show how they are physically assembled into executable components. In fact, some of the modules may be duplicated many times in various executable assemblies. The top level abstract view of the platform software is the UML package diagram shown in Figure 4. A package may contain one or more nested packages and/or it may contain code modules. The content of each package is largely left to the detailed design activity, as long as it conforms to the specifications laid out in this document. The dashed arrows indicate dependency. In strict UML this would mean a build dependency, where the package at the base of the arrow cannot be compiled until the package at the head of the arrow is compiled. However, since this system uses ROS for inter-process communications, many of these dependencies are looser, and simply mean that functionality of the dependent package may be degraded if the indicated package is absent in the runtime environment. Such differences in dependencies are indicated in the package descriptions. With two exceptions, third party packages being used are not included in this diagram. Each of the packages is described below, starting from the bottom of the diagram and working up.

Each package description lays out its responsibilities and major dependencies. Consciously omitted from the dependencies lists are the Utils, Logger, ROS Master Server packages, which will interact with virtually every other package. The ROS Master provides a parameter service that gives each component access to the systems configuration parameters. These are used to describe vehicle characteristics, as well as other configurations peculiar to individual packages.

It is worth noting here that the Drivers layer at the bottom of the diagram represents the software interface to all of the vehicle hardware, including OEM components and aftermarket plug-and-play components. As a result, every driver in this layer is completely dependent on the make, model and version of the hardware to which it connects. For example, if a vehicle has a built-in OBU, then the Comms driver package will include a driver dedicated to talk to that OBU. Then if a secondary, aftermarket OBU is installed, a new driver would need to be written in the Comms driver package, but it would be dedicated to talking to that new model of OBU. All other driver packages are extensible in the same way. Above the Drivers layer, the software will not need to change to accommodate a new piece of hardware unless it provides a uniquely new capability that the software previously did not use.

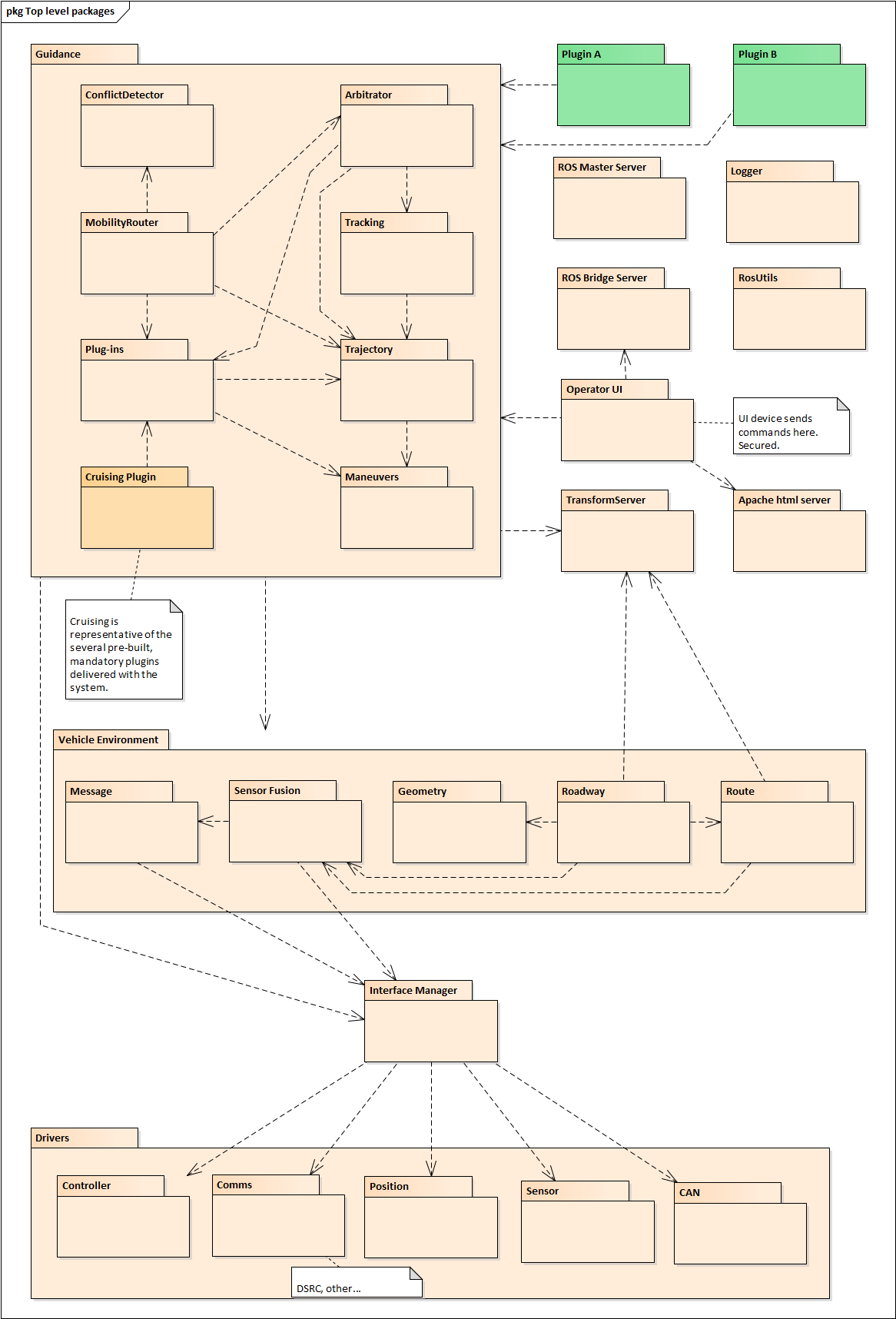


Figure 4. Top level software package diagram.

#### Software Drivers

**Responsibility:** The Drivers package is a container for all of the software drivers in the system. Each driver must comply with a standard interface that will allow the Interface Manager to detect and recognize it and its capabilities, and to allow standardized interaction with the rest of the CARMA software. Because of the possibility of drivers being written in a variety of languages, this interface is not enforced in code, but the set of ROS messages used for interaction define the expectations. In particular, at startup, each driver must provide a POST function that reports the health and availability of the device hardware and communication path to it. A missing or faulty device must not inhibit the higher level software from continuing to execute (although the higher level software may choose to limit its functions based on device fault detection).

**Dependencies:** None.

##### Drivers.Controller

**Responsibility:** This package contains a software driver for each of the supported controller devices that will actuate the vehicle control inputs, such as brake, throttle and steering (there may even be multiple brake controllers for the heavy truck). It also contains drivers for ancillary vehicle manipulation, such as exterior lighting and aftermarket light bar. Each driver is responsible for taking standard command input message (e.g. acceleration command) from CARMA and translating it into the proper signals for the actuator device to understand.

**Dependencies:** Vehicle Characteristics will provide it with PID control constants.

**Note**: Since the current version of the platform only supports longitudinal automation, there will be a dummy driver for lateral control that will simply provide a message to the UI, which can then inform the human operator that it is time to turn the steering wheel to perform the required maneuver.

##### Drivers.Comms

**Responsibility:** This package contains a software driver for each of the supported extra-vehicular communication devices (i.e. radios intended to communicate with other vehicles or infrastructure entities). The initial release provides a generic DSRC OBU driver. Each driver is responsible for passing pre-packaged messages (received as a binary data blob from the caller) to the device in a way that the device will transmit it appropriately. It is also responsible for detecting any messages received by the radio and forwarding them to the higher level consumer software (as a binary data blob).

**Dependencies:** None.

**Note**: The current driver has been used with the Cohda Mk 5 OBU, but it is believed that the driver will work with any model OBU that complies with the same standards. However, it has not been tested with any other OBU.

##### Drivers.Position

**Responsibility:** Contains a driver for each of the supported navigation devices. It will be able to determine and pass on the 3D position of the vehicle on the Earth surface (latitude, longitude, elevation), as well as uncertainties in these quantities. It also provides velocity, heading and odometry information. The current release includes a driver for the TORC Robotics PinPoint device.

**Dependencies:** None

##### Drivers.Sensors

**Responsibility**: Contains a driver for each of the supported external sensors on the vehicle. These sensors may include forward looking radars of various ranges, rearward looking radars, side radars, lane departure sensors, video cameras, and LIDAR or sonar sensors. It needs to translate the raw data provided by the device into a set of objects that the application software knows how to deal with. In the current platform version, since we are only dealing with object detection, the driver must pass on an list of object IDs, distance from the host vehicle, azimuth from the host vehicle’s +X axis, and speed relative to the host vehicle’s speed. It should provide additional data about each object when available, such as classification (vehicle, pedestrian, post, etc), orientation (e.g. velocity vector), size.

**Dependencies**: None.

**Note**: The current version of the platform will only support forward looking radars.

##### Drivers.CAN

**Responsibility**: Contains a driver for each OEM vehicle CAN bus supported (each supported vehicle type may have multiple CAN busses with different message sets). The driver needs to read and interpret each of the bus messages of relevance, as listed in the requirements specification.

**Dependencies**: None.

**Note**: This driver will not interact with custom CAN busses, such as those installed specifically for sensor connectivity.

#### Interface Manager

**Responsibility**: This package forms the hardware abstraction layer. Everything above this layer can operate without knowing details about the hardware or device drivers. This package acts as a communication broker between the drivers below and the more abstract objects above. An object from above this layer will connect to the Interface Manager, letting it know what types of data or capabilities it is looking for. Once the Interface Manager determines which driver(s) are available (passed POST) to fulfill the request, it will facilitate a direct connection to that driver or set of drivers. After that connection is established the Interface Manager will step aside and fall silent, which will eliminate an unnecessary extra message hop between ROS nodes. However, the Interface Manager will continue to monitor the health of each driver as long as the system is running. If it detects a degradation in health, it may choose to alert the rest of the system components, depending on the pre-defined criticality of that device to the system (e.g. a fault detected in the controller driver would cause Interface Manager to initiate a system shutdown since it is a mission critical component).

**Dependencies**: Loose (ROS connection) dependency on Drivers.

#### Vehicle Environment

**Responsibility**: This package forms the third layer of the architecture, and contains packages that represent information about the host vehicle and the environment outside the vehicle. There is no code with direct membership in this package.

**Dependencies**: None.

##### Vehicle Environment.Message

**Responsibility**: Composes and parses all V2V and V2I message traffic. Converts a collection of raw data elements (ROS representations) into properly formatted messages for broadcast, per the J2735 standard or other, as appropriate. These formatted messages are then sent to the appropriate Comms driver for transmission. The current version of the platform will construct BSMs and the four types of mobility messages. It also parses an incoming J2735 message (received over the air and passed in via a Comms driver) into its constituent data elements to be consumed by higher level software components. It will handle BSM, SPAT, MAP and the mobility messages.

**Dependencies**: Loose (ROS messaging) dependence on Interface Manager and Drivers.Comms.

**Note**: Initial release will support mobility messages in a J2735-like format, even though that current standard does not define such message types.

##### Vehicle Environment.Sensor Fusion

**Responsibility**: Fuses all available data about the host vehicle itself and its exterior world to form as complete a picture as possible about the physical situation. Its outputs are a generic set of physical quantities and consistent units of measure, regardless of how they were obtained (e.g. a single quantity called forward speed in m/s, even if the wheel sensor calls it wheel\_speed in ft/s and the positioning device calls it velocity in mi/hr). Sensor fusion is responsible for identifying all available sources of measurement for each provided quantity, determining the accuracy and reliability of each, and weighting or blending them together appropriately to provide the most likely truth value for that quantity. It will also provide an indicator of the confidence level of the resultant value (e.g. an uncertainty variance or standard deviation). Importantly, when providing data regarding the position and motion of other vehicles in the vicinity, it should consider inputs from the host vehicle’s sensors (e.g. radar) as well as BSMs or other digital messages coming from the platform’s comms functions. Outputs related to other vehicles will be provided as a list of objects, with each vehicle object including a static ID that is fixed to that vehicle and independent of sort order.

**Dependencies**: Loose (ROS messaging) dependence on Message, Interface Manager, Drivers.Positioning, Drivers.Sensor, Drivers.CAN.

**Note**: In the current version of CARMA the only data being fused is the forward radar and BSMs to provide a list of neighbor objects near the host vehicle. These objects may be CAVs or NCVs.

##### Vehicle Environment.Geometry

**Responsibility**: An internal library (not a ROS node) that performs geographical location calculations. It provides utilities to convert from geodesic (ECEF) coordinate frame to a local cartesian frame and vice versa. It also performs physical interference calculations (e.g. comparing the locations of two vehicles).

**Dependencies**: None.

##### Vehicle Environment.Roadway

**Responsibility**: Provides a representation of all external objects in route space with objects defined by downtrack and crosstrack distances. Additionally, vehicle sizes and speeds are transformed into route space and objects are given lane ids which can be compared with the host vehicle’s lane ID. This abstraction of objects allows the Guidance package to more easily understand the world around it. As an independent functionality, the Roadway package updates coordinate transforms used to position the vehicle relative to an Earth-Centered Earth-Fixed (ECEF) frame.

**Dependencies**: Loose (ROS messaging) dependenceon Ssensor Fusion

**Note**: In future versions, this may also provide information on road grade and various types of signage or overhead obstacles (e.g. bridges).

##### Vehicle Environment.Route

**Responsibility**: Defines the planned route in terms of a sequential list of waypoints connected by straight line segments. The waypoints will need to be laid out on the roadbed at intervals dependent on the roadway curvature, such that the connecting line segments lay within the bounds of the roadbed (the closer to the centerline of the desired travel lanes the better). A definite starting point and ending point will be represented by the first and last waypoints in the list, respectively. In addition to marking the geometry of the route, each waypoint is capable of providing further information about the route, such as number of lanes available, mile markers, descriptions of other pertinent features or visual cues, required lane positions (e.g. must be in a turn lane to make the coming turn), requirement or prohibition of certain guidance features (e.g. platooning is not allowed on a certain road segment). As the vehicle travels the route, this package will continuously report the current parametric location with respect to the route, in terms of the coordinates (downtrack distance, crosstrack distance). A pre-defined route will be loaded from a data store containing possibly several routes (route definitions will be generated by users to represent test scenarios and are not part of the system, per se).

Route is also responsible for monitoring the Vehicle Environment.Message for any incoming instructions, warnings or recommendations from the infrastructure, such as curve warnings and variable speed limits (such as those imposed by speed harmonization strategies) or lane closures. It will factor these items into the speed limits made available at any requested point along the route.

**Dependencies**: Loose (ROS messaging) dependency on Vehicle Environment.Geometry, Vehicle Environment.Message

**Note**: Future versions of the platform may require lots of functionality in this package in order to achieve Level 2 or higher automation by providing continuous turn and lane change instructions to the guidance package whenever guidance is engaged.

#### Guidance

**Responsibility**: Generates the vehicle’s trajectory and sends commands to the Controller Driver to cause the vehicle to follow that trajectory sufficiently closely.

**Dependencies**: Loose (ROS messaging) dependence on Interface Manager, and each package under Vehicle Environment.

**Note**: This is a large package that includes its own native code plus manages many other embedded packages.

##### Guidance.Arbitrator

**Responsibility**: Manages the activities of all the guidance plug-ins, formulates the final trajectory plan, and orchestrates its execution. Arbitrator logic runs based on a state machine driven by state changes in the vehicle (ACC system state, controller status) and progress along the current route. When the Arbitrator state changes to one where planning of a new trajectory is required (due to system startup, previous trajectory completion, or deviation from the current planned trajectory) the Arbitrator does the following:

* Determine an appropriate planning horizon, based on environmental situation and recent history of planning success.
* Determine which guidance plug-ins are available for planning (availability is dynamic) and request each to make its contribution to the new plan, in priority order. A given plug-in may choose to contribute content for only a portion of the new trajectory, none of it or all of it. Plug-in priorities are defined such that the Route Following Plug-in and the Cruising Plug-in are called last, as they are guaranteed to fill in any empty space in the new trajectory. Thus, the new trajectory covers the entire distance from beginning point to the end of the planning horizon. Plug-in priorities may change dynamically, depending on Arbitrator design.
* The complete trajectory will be subjected to a verification process that looks for inconsistencies, gaps, and other anomalies that would cause it to be unrealistic.
* Once a trajectory plan is verified it is queued up for execution when the vehicle crosses its starting point. Execution will be handled in the Trajectory package.
* Arbitrator will set up a tracking object to monitor actual performance, and use those results to determine if the current plan is to be aborted before it is complete.

A trajectory is made up of discrete maneuvers, which are intended to change the vehicle’s motion. It is possible that any given maneuver could put the host vehicle at risk of colliding with another vehicle or a roadside object. The Trajectory Executor will assume that the maneuvers given to it are safe to execute. Therefore, responsibility for planning a safe trajectory rests with the individual plug-ins.

**Dependencies**: Maneuvers, Trajectory, Tracking, Plug-ins, Vehicle Environment.Roadway, Vehicle Environment.Route.

##### Guidance.Tracking

**Responsibility**: Tracks the current motion of the vehicle and formulates BSMs to broadcast this information at the configured frequency. Continuously compares actual trajectory to the trajectory plan as the vehicle moves downtrack, and determines if there is sufficient discrepancy to signal the need for a new plan, or if the current plan is coming to an end, also signaling the need for a new plan.

**Dependencies**: Trajectory, Vehicle Environment.Message

##### Guidance.Plug-ins

**Responsibility**: Provides the interface point to plug-ins. It provides a choice of interfaces that a plug-in needs to implement and a base class that can be inherited. It also contains several classes for managing a plug-in’s life cycle state machine, for managing the execution of a plug-in while the CAV Platform is running, and for giving the plug-in visibility into all the other plug-ins being used.

**Dependencies**: All other components within the Guidance package.

##### Guidance.Trajectory

**Responsibility**: Defines the trajectory plan objects that are the output of the plug-in algorithms. Also defines the final trajectory plan object that is the output of the arbitrator. A trajectory plan will be represented as two parallel, distance-based sequences of maneuvers, one in the longitudinal dimension and one in the lateral dimension. A trajectory defines its starting point and ending point, in terms of distance downtrack from the start of the host vehicle’s current route. When the vehicle is within these bounds, Guidance will execute the trajectory, meaning it will request from Trajectory the current longitudinal and lateral commands for that point in space so that they can be passed to the vehicle controller.

**Dependencies**: Maneuvers, Vehicle Environment.Route, Vehicle Environment.Message

##### Guidance.Maneuvers

**Responsibility**: A collection of the possible maneuvers that a host vehicle is capable of executing under automated or semi-automated control. There are two major classes of maneuvers: simple and complex. A simple maneuver defines desired motion in one dimension only, has a definite starting and ending condition in that dimension, and can compute definite commands throughout its length at any time. A complex maneuver defines a block of space in both dimensions of a trajectory where no definite maneuvering can be planned. It has starting conditions, but its ending conditions are unknown, as it will not have any pre-computed motion profile. The need for a complex maneuver is to support situations where commands are being generated in real time as the maneuver is executed. Complex maneuvers support strategies like platooning and speed harmonization, where those plug-ins are reacting to sensed roadway conditions as they progress. The complex maneuver essentially reserves a (typically large) portion of a trajectory for this real-time control so that no other maneuvers may be planned there. Once a complex maneuver has been added to a trajectory, no other maneuvers can be added downtrack of it, since its exit condition is unknown. The end of a complex maneuver determines the end of the trajectory.

Supported simple maneuvers are:

* Speed up
* Slow down
* Steady speed
* Stay in lane
* Change lane

These maneuvers may be combined, using one of the longitudinal maneuvers and one of the lateral maneuvers at any given point in the trajectory plan so that they will be executed simultaneously. Each maneuver object specifies its particular constraint parameters, such as acceleration limits, turn radius, etc, and a start distance downtrack of a specified reference point in the trajectory plan (currently beginning of the route). Some of the parameters, including the completion distance, will be dependent on starting vehicle speed and acceleration, which need to be made available to the maneuver. Each simple maneuver object must be capable of commanding its own execution by the host vehicle. It needs to formulate commands to be passed to the Controller Driver for vehicle actuation. All longitudinal maneuvers implicitly involve car following override logic (adaptive cruise control, or ACC) that will reduce the speed as necessary to maintain a minimum gap behind a preceding vehicle that is traveling slower than the host vehicle’s currently commanded speed.

**Dependencies**: None

**Note**: For the current version of the platform, the lateral maneuvers will be represented, but they will be executed by the human driver via a prompt from the platform software. Therefore, they lack specificity in computing exact lateral commands (e.g. steering wheel angle).

##### Guidance.Conflict Detector

**Responsibility**: Maintains knowledge of neighbor vehicles’ future intentions and scans them for possible conflicts with host vehicle’s planned trajectory. It stores neighbor planned paths as they are updated through mobility messages. A search in (downrack-crosstrack-time) is conducted whenever an update is received from another vehicle or when the host plans a new trajectory, and if any of the neighbor paths comes too close to the host’s planned path, this component will notify its parent.

**Dependencies**: Loose dependence (ROS messaging) on Route.

##### Guidance.Mobility Router

**Responsibility**: Manages the routing of mobility messages within Guidance. Guidance components or plug-ins that wish to receive particular types of mobility messages can register themselves with this component as subscribers, and will be sent all messages of that type. Mobility Router also filters incoming mobility messages and determines if they are intended for the host vehicle as a recipient (either as a point-to-point or broadcast message), and discards any messages that are not. Finally, it initiates simple Ack/Nack responses to those incoming messages for which such a response is appropriate. This response depends upon instructions from registered plug-in handlers. The Mobility Router maintains a reference to the Guidance system’s Conflict Detector object, which it keeps updated with newly arrived path information from neighboring vehicles.

**Dependencies**: Strict dependence on Guidance.Arbitrator, Guidance.Conflict Detector, Guidance.Plugins, Guidance.Trajectory and Guidance; loose (ROS messaging) dependence on Message.

##### Guidance.Cruising

**Responsibility**: This is one of the pre-configured plug-ins. It fills a trajectory plan with longitudinal maneuvers to follow the route’s speed limit, using steady speeds plus speed change maneuvers to accommodate changes in speed limit from one route segment to another.

**Dependencies**: Strict dependence on Maneuvers and Guidance.Plugins.

**Note**: This is a pre-defined plug-in that is available as part of the platform installation. It is intended to provide a default longitudinal trajectory where no other plug-ins are recognized or capable of planning.

##### Guidance.LaneChange

**Responsibility**: This is one of the pre-configured plug-ins. This is a tactical plug-in that provides lateral maneuvers to cause the host vehicle to change from its current lane into one of the adjacent lanes. In doing so it first determines if there is a potential conflict with a vehicle in that adjacent lane (via data from sensor fusion). If there appears to be a potential conflict, this plug-in works with Message to broadcast the planned path so that neighboring vehicles will know the host’s intention. It waits an appropriate amount of time, and if it doesn’t receive a Nack message back from a neighbor vehicle, it assumes they will maneuver to accommodate the desired lane change and avoid the conflict, so it will add the lane change maneuver to the trajectory.

**Dependencies**: Strict dependence on Maneuvers and Guidance.Plugins; loose (ROS messaging) dependence on Message and Sensor Fusion.

**Note**: This is a pre-defined plug-in that is available as part of the platform installation.

##### Guidance.RouteFollowing

**Responsibility**: This is one of the pre-configured plug-ins. It provides lateral maneuvers wherever other plugins have not done so. It is currently limited to highway scenarios where the only lateral motion is a change of lanes (CARMA does not yet support turning hard corners). It typically uses same-lane maneuvers, but will insert lane change commands where the route dictates such movement is necessary (e.g. to take a ramp). It uses the LaneChange tactical plug-in to plan the actual lane change activity when it determines such a thing is needed.

**Dependencies**: Strict dependence on Maneuvers, Guidance.Plugins, Guidance.LaneChange; loose (ROS messaging) dependence on Route.

**Note**: This is a pre-defined plug-in that is available as part of the platform installation.

##### Guidance.Yield

**Responsibility**: This is one of the pre-configured plug-ins. It causes the host vehicle to yield to another vehicle when a potential collision is imminent. It is registered with the conflict detector as the handler of such situations. The current version provides a simplistic solution of slowing down whenever a future conflict is detected. The amount of slowdown is dictated by the parameters of the conflict (speed of other vehicle, and distance downtrack of the conflict). If it cannot find a suitable solution, respecting the configured maximum deceleration limit, then it plans a best-effort attempt to reduce the vehicle’s speed to avoid the conflict. The human driver is the ultimate arbiter of safety for the vehicle and is responsible for overriding if the vehicle’s automated Yield is insufficient due to acceleration constraints.

**Dependencies**: Strict dependence on Maneuvers, Plugins, Conflict Detector, Mobility Router.

**Note**: This is a pre-defined plug-in that is available as part of the platform installation.

#### ROS Master Server

**Responsibility**: The ROS Master Server provides naming and registration services to the rest of the Nodes in the ROS system. The role of the Master Server is to enable individual ROS Nodes to locate one another. Once these Nodes have located each other they communicate with each other peer-to-peer.

**Dependencies**: None.

**Note**: This is a pre-built third party component from the ROS framework, but may include some custom configuration data. Upon startup, every ROS node connects to the ROS Master Server and registers itself and its messaging topics. This loose dependence is not mentioned in any of the package descriptions here because it is handled by standard ROS libraries external to the packages described here.

#### Logger

**Responsibility**: Captures logging output from all ROS nodes and records them in one or more log files for future analysis.

**Dependencies**: None.

**Note**: This is a pre-built third party component from the ROS framework, called rosout, but may include some custom configuration data.

#### ROS Bridge Server

**Responsibility**: Acts as a ROS node with the specific purpose of translating ROS messages (topics or services, both in and out) into JSON messages over Websockets to the world outside of the ROS network. In the CAV Platform this capability will be used as the foundation of the Operator UI component, which will use a ROS-specific JavaScript library to connect to the JSON stream and make the data elements available for display in HTML.

**Dependencies**: None.

**Note**: This is a pre-built third party component from the ROS framework, but may include some custom configuration data.

#### ROS Utils

**Responsibility**: A collection of utility classes that will be used by multiple packages to help ensure uniformity of integration with ROS and make it more streamlined.

**Dependencies**: None.

#### Operator UI

**Responsibility**: A set of interfaces that will make GNC data available to external clients, intended to be used for the operator tablet. Also accepts commands from the tablet. Commands to be supported will be:

* Enable/disable automated guidance and control
* Select which guidance plug-ins to activate
* Select which route to drive

**Dependencies**: ROS Bridge Server. Loose (ROS messaging) dependence on Guidance, Vehicle Environment.Route, Vehicle Environment.Geometry.

**Note**: Will be built with html and Javascript, thus requires the apache server to execute the html.

#### Transform Server

**Responsibility**: The Transform Server will provide services for any system component to convert geometry data from one coordinate frame to any other frame defined in the system. Coordinate frames and their transformations will use the ROS tf2 package’s pre-defined coordinate frame operations.

**Dependencies**: None

# Physical Architecture

The physical view of the architecture presents the arrangement of the various system components and their interconnections. The term *component* here is used to mean any individual, functional assembly that can be independently installed or replaced. It can refer to a hardware item, such as a computer or router. It can also refer to a software item, such as an executable or database. Supporting items that do not provide direct system functionality in themselves are not considered, such as fasteners, cables, configuration files and operating system software.

## Hardware Components

### Control System

The platform GNC system will consist of a PC and one or more low-level controllers. The PC will house and execute the guidance algorithms as well as partial solutions for navigation and control functions, and will pass its speed commands to the low-level controller via CAN. Each of the legacy CARMA vehicles already has the described equipment installed, and the new CAV Platform will continue to use it in approximately the same configuration. For vehicles newly added to the fleet, the physical configuration of these components will be determined on a case-by-case basis to fit the available space of that particular vehicle.

The PC will be installed with Ubuntu 16.04 LTS as its operating system since it will be supported until at least 2020. The key third party packages installed will include:

* ROS Kinetic Kame, because it is built to sync with Ubuntu 16.04 and because it supports construction of ROS nodes directly in Java;
* Java 1.8, which will handle the reusable code modules from legacy STOL projects.

### Communications

Communications can be divided into intra-vehicle, which is used between components of the system, and inter-vehicle, which is used between the host vehicle and neighboring vehicles or transportation system entities.

Within the vehicle, the platform components communicate primarily with each other by Ethernet through an Ethernet switch. The low-level controller communicates by CAN over the Research CAN bus. Communication with the vehicle systems is done by CAN over the High-Speed CAN (HS CAN) bus and the Forward Object CAN (FO CAN) bus.

Communications outside the vehicle occur through multiple devices. DSRC messages pass through the DSRC Onboard Unit (OBU) and contain data about other vehicles and roadway users, and infrastructure. 4G data passes through a 4G router and aircard for general IP traffic, such as allowing researchers to remotely log into the PC. GNSS data passes through a GNSS receiver to provide position and timing information to the platform.

### Data Acquisition

The US Army’s ADMAS system will be integrated into the platform to provide data acquisition for system performance evaluation. This will provide a comprehensive set of synchronized data that will log information from CAN bus, Ethernet, wireless, and other sensors for post-experiment analysis.

A CAN logger is also available to log the traffic on the available CAN buses. This device has a remote control to conveniently start and stop logging. Data are stored to a removable SD flash card for easy transfer to a computer for analysis.

### Power System

The hardware components are powered by the main vehicle battery. A switch/circuit breaker provides a means to cut power to the system. Power is distributed to each component through either of two power distribution modules, which include a built-in electronic circuit breaker.

A battery monitor is installed to display various measurements of the battery, including voltage, current, and state-of-charge (SoC). To protect the main battery from over-discharge, the battery monitor can disable the power distribution modules when the battery SoC falls below a configurable threshold.

### OEM Systems

The platform relies on various OEM systems for sensing and operation, including throttle and braking control, speed, odometer, exterior lighting, and radar. The data from each of these systems are available on one of the vehicle CAN buses.

Because of the critical timing required on the vehicle CAN bus, only the TORC low-level controller is allowed to write data to the vehicle CAN bus. The PC has a link to the CAN bus to be used only for monitoring.

## Software Components

### Deployment Configuration

The software components are shown in the UML deployment diagram in Figure 5. In a software deployment diagram each box represents a component, which is a separately buildable and executable piece of software. The lollipop appendages (with solid circles) are provided interfaces, and the appendages with open semi-circles are required interfaces. The dashed lines are data flows. In this platform each of the components is a ROS node, and the provided interfaces are ROS messages. The messages may be either synchronous (services) or asynchronous (topics). Some hardware and COTS software elements are represented here as well to show context. Most of the software components are built from their own top level package, so there is no need for further description of those components here. Exceptions include the Operator UI component, which will include the ROS Bridge and a collection of web pages formulated in HTML and JavaScript. These will provide content for the operator tablet in its local browser. The Operator UI will use the Apache web server to execute its HTML code. There are two other top level packages that don’t form their own components because they will be statically linked into several other components. These are the ROS Utils package and the Geometry packages.

In this diagram the guidance plug-ins are colored differently from the other components. The brown plug-ins are mandatory and pre-configured during the CARMA installation. The green plug-ins are representative of the third-party plug-ins that may be optionally built and installed to customize the system’s behavior. The four shown were developed as part of the Task Order 13, but any of these could be removed and any number of others could be added to a given installation of the system.

The ADMAS unit at the bottom of the diagram will tap into all of the connections between the PC and the hardware devices, so will be able to record data flows in both directions between these. It will also have access to the webservices published by the Operator UI component, which will make available all of the data that is being displayed on the UI tablet.

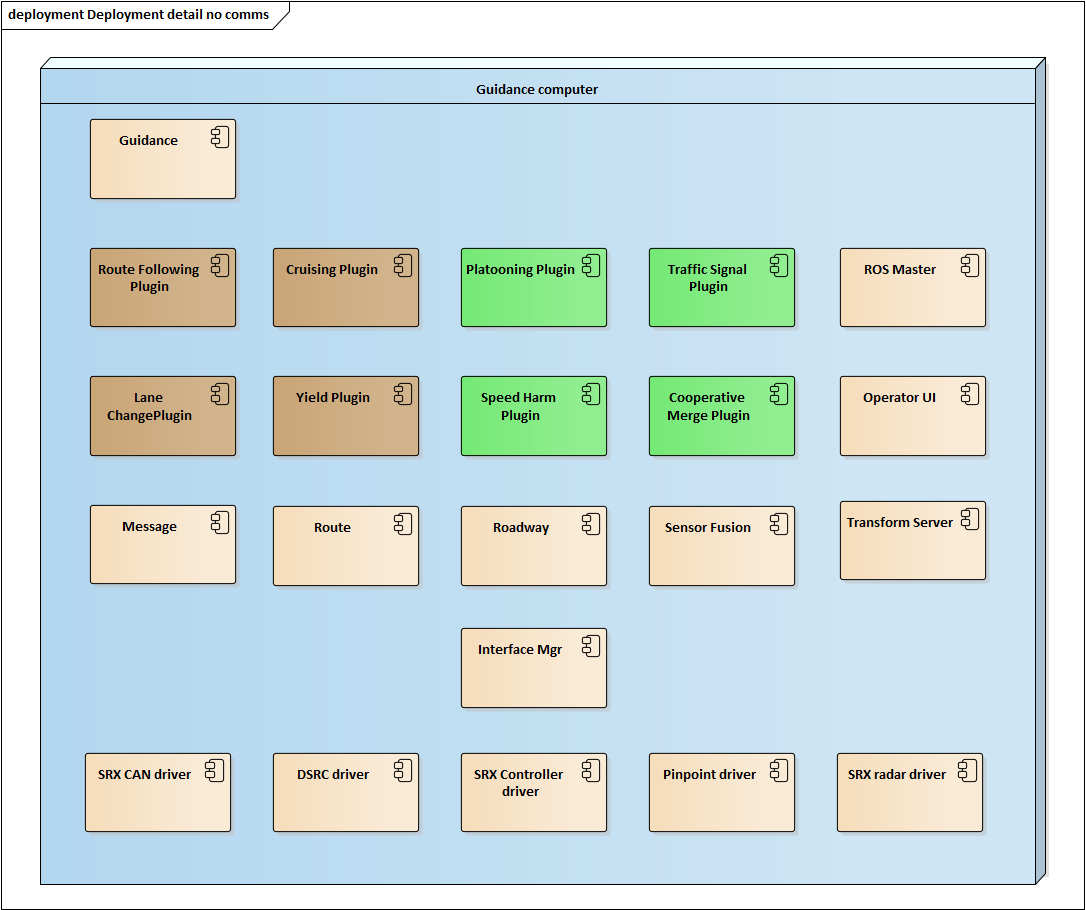


Figure 5. Software components and internal interfaces.

### ROS Messages

Data provided by the various ROS nodes are specified in a separate Excel workbook that is maintained by the development team, as it is not practical to maintain such information in this static document. This spreadsheet is updated with information directly from the source code for these messages whenever it is modified.