CISC 332
A2: Concrete Architecture of Apollo
3/21/2022

Marc Brasoveanu	18mtb5@queensu.ca
Andrew Wilker	18ajw15@queensu.ca
Nathan Perriman	17nmp2@queensu.ca
Theo Raptis	17tjr5@queensu.ca
John Mahone	21jgm18@queensu.ca

Table Of Contents

Abstract

This report focuses on the concrete architecture implementation of the Apollo autonomous driving library. Through the use of provided graph visualisations of pub-sub message traffic and use of the Understand tool we were able to recover the static code dependencies from the Apollo source code. From this recovered information we were able to generate a dependency graph which would then go on to be used in the creation of the concrete architecture.

The major findings in our report begin with the description of the top level concrete subsystems where we have established that the main architectural style is the pub-sub style.

The analysis of the CanBus subsystem has shown it to be a bridge between the Apollo autonomous driving software and vehicle. The main purpose of CanBus is to accept commands which are sent to the control module and then execute them through an interface on the physical vehicle. The three main components of CanBus were found to be the main CanBus module, CanBus driver module, and the vehicle controller subsystem. The main CanBus module takes in ControlCommand type objects as input and returns Chassis and ChassisDetail objects which describe the status of the vehicle. The main CanBus module supplies information to other modules such as the vehicle controller module. The vehicle control module handles information transfer between the main CanBus module and the CanBus driver module and deals with translation between different vehicle models. The driver module is tasked with communicating directly with the vehicle through function calls at higher level modules such as the vehicle module. Through the use of the Understand tool references to the driver module were unexpectedly discovered from the main CanBus module. These references however only summed to 5 as compared to the 656 references between the vehicle controller module and the driver module.

We found that upon analysing the differences between our initial conceptual architecture and the concrete architecture that we recovered a number of noteworthy differences were uncovered. Some of the differences include the number of dependencies to the Map module which ended up totaling 7 along with the fact that the Control module does not send information directly to the CanBus but rather to the Data module. The Localization module was only uncovered to send data to the Data module and not to other modules. The Storyteller module was found to send information directly to the Map module rather than to various other modules. The final main difference was that the Task Manager module does not have direct dependencies because of the Publish/Subscribe nature of the architecture and instead receives events.

Two sequence diagrams were presented for two use cases, the first being that the automobile must travel to a destination and the second being that the automobile must park itself.

Introduction and Overview

The purpose of this report is to model out the concrete architecture of the Apollo autonomous driving open source platform, with regards to the conceptual architecture that we modelled in our previous report.

To begin this report, we describe the derivation process that we used to generate our concrete architecture, including the use of the software called Understand. With this software, we used our proposed conceptual architecture and then generated the graph of dependencies.

The next section, we showcase the concrete architecture of the entire system as shown within Understand. We briefly explore the differences between the conceptual and concrete architecture, which will be discussed even further in a later section.

Having looked at the top level of the architecture, we then focus into one of the specific subsystems and analyse it. For our report, we have selected the CanBus subsystem. In this section, we discuss the subsystem in detail and list the functions that it performs within the software system. Following on, we then propose a conceptual architecture for the CanBus subsystem which is then used alongside the Understand tool once again to see if the concrete architecture of the system matches our expectations.

In the next section of the report, we further discuss the differences between our conceptual architecture and concrete architecture. We focus on the major differences that we found, which includes: Map Dependencies, Canbus-Control Interaction, Localization Data, Storyteller Publishing, and Task Manager. For each of these, we explain the differences that we found between our conceptual and concrete architecture in regards to these specific areas.

We then identify two use cases and provide sequence diagrams with concrete function calls. The first use case being that the automobile travels from one destination to another and the second being that the automobile must park itself.

Lessons learned were then summarised which included reinforcing the differences between a conceptual architecture and concrete architecture, learning how to use the Understand tool to derive a concrete architecture, and how to understand and interpret developer documentation to derive a concrete architecture.

The report is then summarised with a conclusion restating the major findings.

The final sections include a data dictionary in which key terms are explained along with a naming conventions section followed by references.

Derivation Process

For this report, we used the Understand program to derive our concrete architecture for Apollo. Understand is a tool that is used to analyse the code base of a given project and map the dependencies contained within, with a visual map showing these links within the Apollo modules.

After loading the code base into Understand, we create an architecture based upon the subsystems that we identified within the conceptual architecture. A recap of this conceptual architecture has been given below, in ***Figure 1***. The creation of the top level concrete architecture started off pretty simply by mapping modules that matched the name of a subsystem first. Once these were done, we read through the code in the remaining modules and assigned them to the appropriate subsystem in our concrete architecture. For example, the common folder contains information considered miscellaneous and thus the contents of that folder was split between subsystems that best matched the described functionality. Outside of this, the Misc and Bridge subsystems were removed as we focused on certain parts of the Apollo system.

Following the completion of this process, we then generated a dependency graph based on the mapped modules from this top level architecture. From this generated graph, we recreated our concrete architecture, which is shown in the Top Level Concrete Subsystems section.

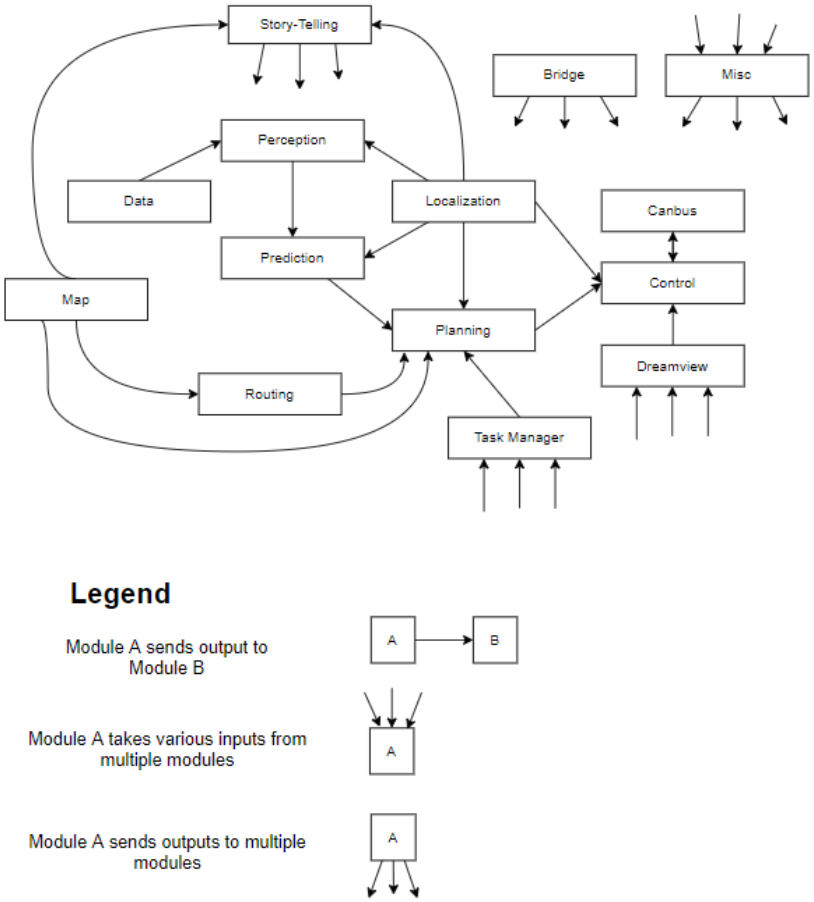


Figure 1 - Recap of Conceptual Architecture

A similar process was followed in order to derive the architecture of one of the subsystems of Apollo. First, we engaged in a review of the available subsystems of Apollo in order to select the most suitable subsystem to analyse. After the selection of the CanBus subsystem, documentation from both the official Apollo GitHub and website as well a developer blog were used as sources in order to gain further understanding about the subsystem's operation and structure. Following this, a conceptual architecture was proposed and then compared with a concrete architecture derived using the Understand tool. Git command line tools were utilised in order to identify the source of discrepancies between the conceptual and concrete architecture, using a sticky-notes type analysis technique by looking at commit messages to determine developer motivation behind the identified differences.

Architecture

Top-Level Concrete Subsystems

Having created a dependency graph for the Apollo self driving software, we created a new top level concrete architecture based upon these graphs. The generated

graph shows the new dependencies alongside the ones that still exist from the previous conceptual architecture. There are differences between our conceptual and concrete architectures shown, primarily with the data subsystem which is actually used by a number of subsystems as opposed to just the perception subsystem. There are also many usages of the Map subsystem being called throughout the system.

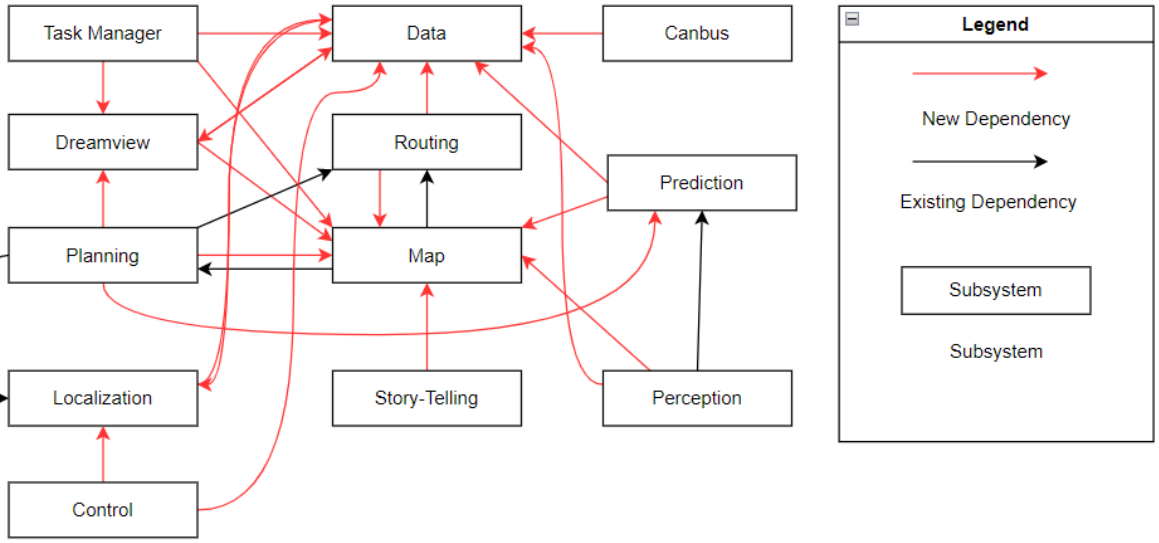


Figure 2 - Concrete Architecture based upon Understand Dependency Graph

As we noted in our report on the conceptual architecture of the system, Apollo follows a mixture of the layered, publisher-subscriber, and the process-control architectural styles. As we delved deeper into the software, it became noticeable that it makes use of the pub-sub style the most. As shown in our updated concrete architectural layout, information is being transmitted to the Data module from a number of other sources. This comes from a number of different information publishers, including the Canbus, Perception, and Prediction modules.

Subsystem Report: CanBus

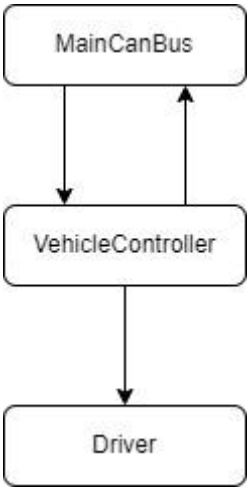
The CanBus subsystem is the bridge between the Apollo autonomous driving software and the vehicle. The principle responsibility of CanBus is to accept commands sent to the control module, execute these commands via an interface with the vehicle itself, and collect information from the vehicle to provide feedback to the rest of the control module. As CanBus is a relatively simple subsystem, there are only three main components to cover. The three main components of CanBus we focus on for this report are the main CanBus module, which is also referred to in documentation as “upper CanBus” (found within apollo/modules/canbus), the CanBus driver module (found within apollo/modules/drivers/canbus), and the vehicle controller subsystem (found within apollo/modules/canbus/vehicle).

The first submodule of CanBus we will focus on is the main CanBus module. The main CanBus module accepts ControlCommand objects as input, and outputs Chassis and ChassisDetail objects, both of which contain information about the status of the car itself. As such, the main CanBus module can be thought of as the highest-level submodule of the CanBus module, as it is responsible for receiving input from modules external to CanBus itself, as well as having the responsibility of supplying input to the other components of CanBus and returning output to external modules. The main CanBus component will need to interface with the vehicle controller, which serves as an intermediate between main CanBus and the vehicle itself.

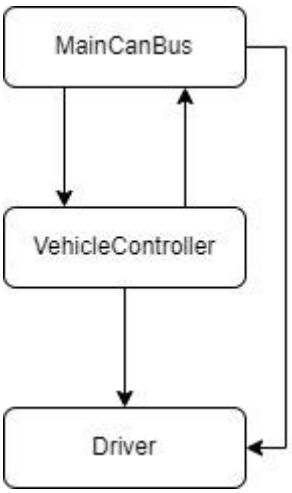
The driver component of CanBus is responsible for sending and receiving messages from the vehicle itself. It can be thought of as the lowest level member of CanBus since it is the closest that Apollo gets to the control systems on the vehicle itself. Functions within the driver module will be invoked by a higher level component of CanBus (the vehicle controller) in order to send commands to the vehicle's own control systems. The nature of these messages will change depending on the make of vehicle used for autonomous driving. The vehicle controller, as the module responsible for this translation, must be at a higher level than the driver.

The next submodule of focus is the vehicle controller. The vehicle controller module handles the interaction between the main CanBus module and the CanBus driver module and provides a method for communicating with different makes of vehicles. It will have a dependency with both main CanBus as well as the driver itself in order to fulfil its role.

With the above analysis, we propose the following conceptual architecture for the CanBus subsystem.



Having proposed the above conceptual architecture for CanBus, the next step is to determine the concrete architecture of this subsystem and compare it with our conceptual model. We achieved this through the use of the Understand tool by mapping each submodule of CanBus based on the folder structure explained in the beginning of this section. The resulting analysis by the understand tool revealed the following architecture.



While the concrete architecture closely matches the majority of our proposed conceptual architecture, Understand revealed an unexpected dependency between the upper CanBus submodule and the Driver submodule. References to the driver module were made in 2 files, the canbus_component.h header file and the canbus_component.cc file. Git logs show that this dependency has existed in Apollo for several versions now, first appearing in code via a commit made on October 13th, 2017. The commit in which this dependency was added succeeds two prior commits in which the driver module was moved out of the CanBus folder and to the common folder, and then shortly thereafter moved to the drivers folder. The dependency between these two modules seems to be a legacy of a much earlier version of Apollo (Apollo 1.0) where they were both contained within the same module.

The motivations for moving the driver module are not made clear through these commits, but it may have been due to the driver module becoming a larger component of CanBus that was increasingly disjoint with the rest of the main CanBus module. This can be inferred through the observation that Understand was able to identify 656 references between the vehicle controller module and the driver, as compared to only 5 between the main CanBus module and the driver.

Conceptual and Concrete Architecture

Looking back at the conceptual architecture initially proposed by our group, we can see that the concrete architecture contains a number of unexpected dependencies. Although covering every new dependency would be outside the scope of this paper, a few notable examples are listed:

Map Dependencies

We found within the concrete architecture that the Map module requires 7 different modules' inputs in order to function. This is because the map requires information from every part of the vehicle in order to display them on-board for the user to see. Examples include requiring information about the vehicle's position from the planning

module, its surroundings from the perception module, the predicted route from the prediction module, the simulated world as a result of the dreamview module, and more.

Canbus-Control Interactions

Within our conceptual architecture, we listed the Canbus and Control modules as having shared dependencies. However, as discussed above, the Canbus module has a majority of its data being sent to the Drivers module instead. This is because the Drivers module has its own subsystem dedicated to interpreting the results of Canbus. The Control module also doesn't send any data directly to CanBus either, and instead both send their respective information to the Data module to record the status of the car and calculate its required moves.

Localization Data

In the concrete architecture, the Localization module only directly sends anything to the Data module, unlike the conceptual architecture in which it's listed as a dependency for several modules. The reason that this change occurs in the concrete architecture is because the Data module's components will automatically localize the data they receive, moving the task of converting to localized data from each individual module and onto the Data exclusively.

Storyteller Publishing

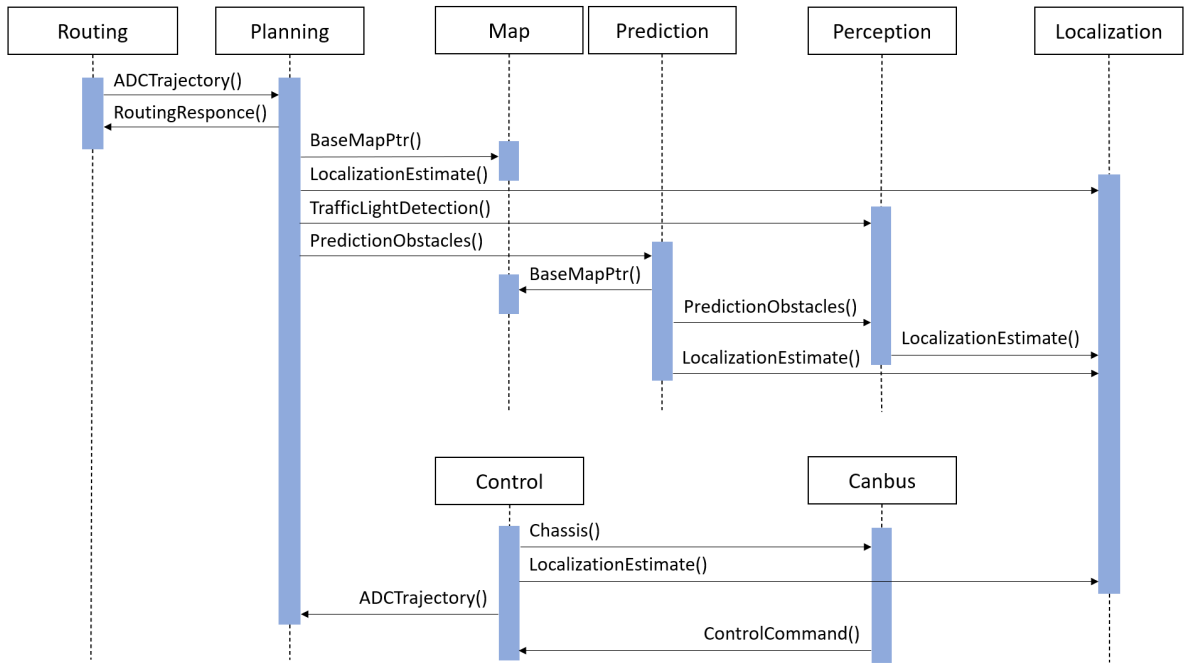
In the conceptual architecture, we had listed that the Storyteller module sent an output to various modules, and although that technically is true, the way it was proposed in the architecture was misleading. The only data actually sent to another module directly is actually to the map, where the storyteller will inform the map of any predicted outcomes when the vehicle approaches an intersection. The majority of the Storyteller's data is sent in the form of the Publish-Subscribe style, in which the Storyteller can simply publish the stories as it calculates them, and if they come true then the subscribed modules can read the story in then. The concrete architecture is updated to reflect this.

Task Manager

In the conceptual architecture, the Task Manager is shown as receiving various inputs from other modules to function. However, it instead doesn't require any dependencies. This is because of the system's Publish/Subscribe system, as any event that the task manager is required for will be announced as a story.

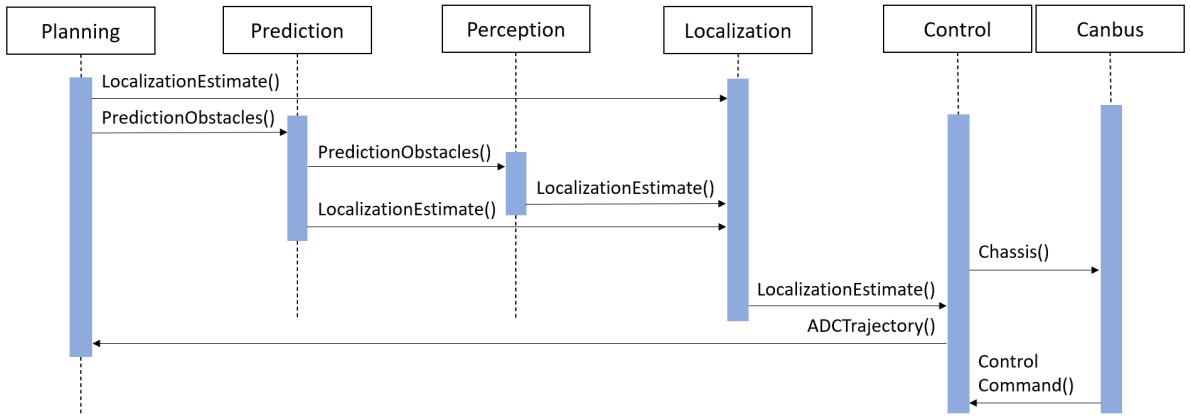
Sequence Diagrams for Use-Cases

Use Case 1: The automobile must travel to a destination



Due to the nature of Apollo's architecture, most communications between modules is done via pub-sub. An exception to this is the BaseMapPtr() method call. The planning module gets information from the routing, map, prediction, perception, and localization module. In conjunction with the canbus and localization module, the planning module tells the control module what to do, which in turn sends the control commands to the canbus and thus the vehicle.

Use Case 2: The automobile must park itself.



The automobile makes a plan on how it will park, then sends it over to the control module to execute on. The automobile still needs the perception and prediction modules in case people or animals are aiming to get hit by the car.

External Interfaces

The Apollo software system receives information from a variety of external sources. Hardware wise, it receives information from camera, IMU, radio, and LiDAR signals and other sensor inputs from the car. Apollo's map module is populated by the HD Map, which itself resides on the cloud. Apollo has an HMI for development, which interacts with the dreamview module; it contains information pertaining to the vehicle.

Naming Conventions

GPS - Global Positioning System. A system that uses a receiver's connection to a series of satellites in order to determine its location on the globe.
HMI - Human-Machine Interface. A dashboard that is used to connect a user to the machine, by displaying information and controls for the user.
IMU - Inertial Measurement Unit. A device that uses a series of gyroscopes and accelerometers to determine its orientation, angular rate and acceleration.
LiDAR - Light Detection And Ranging. A type of radar used to detect the distance of objects using a pulsed laser.
HD - High-definition.

Data Dictionary

Pub-Sub/Publish-Subscribe – Architecture style where modules subscribe to a publisher module
Process Control – Realtime architecture for controlling processes
Concurrency – Multiple items are run at the same time
Signal – A message sent to a module
Localization – Module that lets the automobile know where it is
Canbus – Medium between the automobile's hardware-software components
Control – Module that sends control commands
Dreamview – Developer HMI
Guardian – Safety module
Map – HD map module
Monitor – System monitor
Perception – Module that detects objects in images
Planning – Module that plans the auto's motion
Prediction – Module that predicts what is happening in its environment
Routing – Module that makes routes
Storytelling – Manages other modules
Sequence Diagram – Diagram that shows the sequence of events over its modules

Conclusions

In conclusion some of the key findings were that the architecture is pub-sub style as mentioned in our conceptual architecture. The CanBus system was shown to be a bridge between the vehicle and the Apollo software. Its purpose was to accept commands sent to the control module and execute them in the physical vehicle. Three main components made up the CanBus system which include the main CanBus module, the vehicle controller module and the driver module. After completing reflection between the conceptual architecture of CanBus and the concrete architecture we concluded that they were very similar however the concrete architecture included additional unforeseen references between the main CanBus module and the Driver module. We also found that after analysing the differences between our conceptual architecture and our concrete architecture some of the main differences we uncovered included the type of dependencies associated with the following modules: Map Dependencies, Canbus-Control Interaction, Localization Data, Storyteller Publishing, and Task Manager.

Lessons Learned

As a group we learned significantly more about the reflexion process throughout the creation of this report and were able to put to use many of the ideas and practices discussed in class lectures during our analysis of Apollo's concrete architecture. As a group, we feel that we learned three key lessons while creating this report.

The first lesson we learned regards the difference between conceptual vs. concrete architecture. While we had previously learned in class that conceptual and concrete architecture are not usually the same, it was still enlightening to go through the reflexion process and see the difference between our proposed conceptual architecture and the actual concrete architecture of Apollo. It was certainly a learning experience to have to confront the differences between our conceptual architecture and the concrete architecture and try to understand why these differences exist. Having to compare our conceptual architecture to the concrete architecture of Apollo exposed to us the difference between concept and reality and has reinforced the fact that conceptual architecture will rarely be the same as concrete.

Another important lesson was how much of an impact the quality of developer comments and documentation can have on the analysis process. A limitation of this report is that some developer comments regarding unexpected dependencies we explored, such as that in the subsystem report, were vague and either gave little detail regarding the purpose of a change or did not provide any detail at all. In addition, one of the developer documents which turned out to be especially important to the subsystem report was written in Mandarin. A version of this report translated to English with Google translate provided critical details about the CanBus subsystem

which were omitted from English Apollo documentation. However, computer translation is still imperfect and there were certainly details that were lost, or perhaps translated incorrectly.

Finally, many group members got to use the Understand tool for the first time. Being able to quickly extract the concrete architecture from such a large software project illustrated the usefulness of this tool in analysing a software project.

References

1. https://github.com/ApolloAuto/apollo/tree/master/modules/canbus
2. https://zhuanlan-zhihu-com.translate.goog/p/85083829?_x_tr_sl=zh-CN&_x_tr_tl=en&_x_tr_hl=en&_x_tr_pto=sc
3. https://apollo.auto/developer.html