TECHNICAL UNIVERSITY OF MUNICH

DEPARTMENT OF INFORMATICS

Master's Thesis in Informatics

An Online Verification Framework for Autonomous Driving Using Sampling-Based Motion Planners

Jiaying Huang

TECHNICAL UNIVERSITY OF MUNICH

DEPARTMENT OF INFORMATICS

Master's Thesis in Informatics

An Online Verification Framework for Autonomous Driving Using Sampling-Based Motion Planners

Ein Online-Verifikationsframework für Autonomes Fahren mit Stichprobenbasierten Bewegungsplanern

Author:	Jiaying Huang
Supervisor:	Prof. Dr.-Ing. Matthias Althoff
Advisor:	Xiao Wang, M.Sc.
Submission Date:	02.08.2021

Abstract

Online Verification and Intel RSS Verification are both safety layers in the motion planning of autonomous driving. Online Verification generates fail-safe trajectories according to future occupancy of traffic participants, while RSS provides acceleration limits according to safe distances. This paper compares the performance of Online Verification and Intel RSS v1.1.0 based on the open-source software platform Apollo6.0. Methods of integrating the Online Verification framework into Apollo6.0 are provided, and some tools are adjusted to achieve less computation effort in the real-time simulation system. Experiments are done in multiple scenarios including Apollo scenarios, scenarios converted from CommonRoad and hand-crafted scenarios. Results show that RSS helps generate collision-free trajectories in most scenarios and lead to more aggressive behaviors of the automated vehicles, it also failed to ensure legal safety in some critical cases. Online Verification, on the other hand, ensures legal safety in all tested scenarios.

Contents

1 Introduction

Autonomous driving is currently a topic of much interest within and outside academia. It has been widely studied in the last few decades, and has been put into practice in the early 21st century. Automated cars and trucks have been tested on roads and put up for sale. Autonomous driving system consists mainly of three components:

- environmental perception
- behavioral decision
- control

The environmental perception component refers to sensors and algorithms. Sensors, such as cameras, lidars and GPS, are used to obtain information of the ego vehicle (the subject automated vehicle under test) and its surrounding environment. Algorithms extract traffic and map information from images or videos provided by sensors. The algorithms can include traditional or learning-based computer vision algorithms.

The behavioral decision component plans trajectories for the ego vehicle that are collision-free and follow traffic rules according to sensed map, traffic and ego states. A trajectory contains the position, heading, velocity and acceleration of the ego vehicle at each time step.

Finally, the control component gives commands to the accelerator, the brakes and the steering wheel of the ego vehicle to make it drive on the planned trajectories.

1.1 Motion Planning in Autonomous Driving

After receiving information of perceived obstacles, behavior predictions of the obstacles will be made according to their current states. The predictions can be done with traditional or learning-based methods, and the results can be possible trajectories or occupancy sets (polygons that show all possible positions of obstacles during a certain time interval). Then a route is planned from the start position to the goal position of the ego vehicle. The route planning does not take obstacles into account, it only

gives a reference path from the starting point to the end goal. Next, the behavior of the ego vehicle is planned to decide what course of action should be taken, such as performing lane changing, car following or braking. After that, the planner will generate a collision-free trajectory based on the real-time obstacle predictions and the route.

Typical motion planning algorithms are classified into the following types: graph search (e.g., Dijkstra and A-star), sampling (e.g., Rapidly-exploring Random Tree(RRT) and RRT*), interpolating (e.g., polynomial curves and spline curves) and numerical optimization (e.g., function optimization) [1].

1.2 Safety in Motion Planning

The safety of autonomous driving is one of the most important and challenging topic. Safety of existing machine learning-based prediction and planning algorithm is not guaranteed in scenarios with complex traffic. Several traffic accidents have occurred throughout the world since the application of autonomous vehicles on real-world road trips.

Safety problems of autonomous driving can be caused by multiple modules such as perception and decision making. In the perception module, sensors as well as bad weather conditions such as sun glare, rain and fog, or occlusion may increase the difficulty of vision tasks (classification or image segmentation). In the decision module, trajectory planning is done according to predicted obstacle behaviors. The predictions of obstacles can be several most possible trajectories calculated by learning-based methods, or occupancy sets that covers all possible positions of an obstacle in the next several time steps. If the predictions show large deviation from the obstacles' real motion behaviors, the planned trajectory may not be safe.

To deal with uncertainties of the motion of traffic participants, sensor and perception module, a safety layer can be added before the execution of planned trajectories. The safety in motion planning could either be improved by Online Verification additional (performing set-based prediction, collision check and fail-safe trajectory generation after getting a planned intended trajectory), or by adding an Intel Responsibility-Sensitive Safety model (RSS) which is parallel to the planning process and calculates restrictions for accelerations.

1.3 Motivation

In the aspect of motion planning, ensuring the generation of safe trajectories in arbitrary [2] traffic situations is essential. Unlike the safety problems in robot motions, dangerous situations in autonomous driving are often more serious. Although accidents do not often occur, each accident may cause large damage, and it is worth every effort to avoid safety problems in autonomous driving. Thus, critical scenarios that may have chances to cause collisions, also deserve great attention even if they seldom occur. Danger can be more easily show up if the predicted trajectories deviate largely from the real trajectories of obstacles, and the ego vehicle continues to follow the planned trajectory, which is considered safe according to the false prediction. When the ego vehicle detects the danger, it may be already too late to take any action to avoid the collision. The tool CommonRoad Set-based Prediction of Traffic Participants (SPOT) calculates the set-based prediction of all perceived obstacles based on their current states. It takes every legal motions of obstacles into account and does not have the disadvantage of false predicting the intention of obstacles. Theoretically, it should have the ability to verify the legal safety of the intended ego trajectory and help provide a fail-safe trajectory. In this paper, various traffic flows will be generated to test the safety of fail-safe trajectories generated according to this set-based prediction method. Its performance will be compared with the Intel RSS module applied in Apollo open-source platform.

1.4 Overview

Chapter 2 introduces the related works about the safety in motion planning, including intel RSS and CommonRoad Online Verification. Chapter 3 lists the foundation of the following experiments, including the introduction of tools used for testing and evaluating the set-based prediction method. In chapter 4, methods for establishing the testing environment, providing necessary inputs and generating traffic flows are discussed. Also, methods of reducing time consumption in each module are applied to improve the run-time performance of the online verification process. Chapter 5 explains the requirements for the experiments, and specifies the settings of necessary parameters in tools. Then some typical scenarios are analyzed, and the performance of RSS and CommonRoad Online Verification are compared. Finally, a summary of the testing results and a discussion of some future work are given in chapter 6.

2 Related work

2.1 Intel Responsibility-Sensitive Safety model (RSS)

RSS is a formal model for safety assurance of automated vehicles. More specifically, it is an open and transparent model that uses mathematical formula for safe decision making by defining dangerous situations and proper response. It imitates human driving behavior and ensures safety by keeping safe distances longitudinally and laterally. The definitions of longitudinal safe distance and lateral safe distance from RSS is given below [3][4]:

- A lane-based coordinate system:
  Road in real word are seldom straight. To make the longitudinal axes and lateral axes meaningful, RSS uses a lane-based coordinate system to express the positions and velocities of vehicles.

- Longitudinal safe distance:
  Let c_r be a vehicle which is behind the vehicle c_f on its longitudinal axis, then the longitudinal distance is the distance between the front-most point of c_r and the rear-most point of c_f. The longitudinal distance between the two vehicles is considered safe if the rear car c_r is able to avoid a collision even if the front car c_f abruptly applies full braking force.

- Lateral safe distance:
  Let c_1 is a vehicle to the left of vehicle c_2, then the lateral distance is the distance between the right side of the left vehicle c_1 and the left side of the right vehicle c_2. The lateral distance is considered safe if they accelerate toward each other in a reaction time period, and then apply lateral braking until they reach zero lateral velocity, they still have a certain final lateral distance.

The above introduced longitudinal and lateral safe distance is used to ensure the safety of autonomous driving with the following three steps:

1. Formalize: During this phase, automated vehicles drive with a normal speed, and senses the information of surrounding vehicles. The automated vehicle keeps a longitudinal safe distance and a lateral distance according to the sensed data.

2. Identify: If a potential dangerous situation occurs, either longitudinal or lateral, the automated vehicle should be able to identify the potential danger.

3. Execute: Use braking to keep longitudinal and lateral safe distances.

Automated vehicles are designed to imitate human driving behaviors and improve the safety of driving process. Human drivers usually drive according to traffic rules. However, unlike machines, human and vehicles may not follow all the traffic rules at any time. Sometimes, better driving efficiency is achieved if certain traffic rules are slightly obeyed, and obeying these traffic rules does not cause potential dangers. This leads to the important balance between safety and efficiency, and is related to the implicit driving rules of human drivers, which are more flexible than the traffic rules. To imitate the human driving behaviors, RSS is modeled based on the following common senses [4]:

1. Do not hit someone from behind.

2. Do not cut-in recklessly.

3. Right-of-way is given, not taken.

4. Be careful of areas with limited visibility

5. If you can avoid an accident without causing another one, you must do it.

As mentioned above, the basic idea of RSS is to help the automated vehicle keep a longitudinal and a lateral safe distance and give proper response under different dangerous situations. Here the calculation of safe distances is introduced.

2.1.1 Longitudinal Safe Distance (Same Direction)

If two vehicles (a rear vehicle and a front vehicle) drive in the same direction, the longitudinal safe distance can be determined by the following expression:

$$d_{min} = \left[v_r \rho + \frac{1}{2} a_{max,accel} \rho^2 + \frac{\left(v_r + \rho a_{max,accel}\right)^2}{2 a_{min,brake}} - \frac{v_f^2}{2 a_{max,brake}} \right]_+ \tag{2.1}$$

In the above expression, d_{min} is the longitudinal safe distance of the pair of rear and front vehicle in the same direction. v_r and v_f are the velocities of the rear and front vehicles, ρ is the reaction time of the rear vehicle before braking. $a_{max,accel}$ is the maximal acceleration of the rear vehicle, and $a_{min,brake}$ is the maximal braking acceleration of

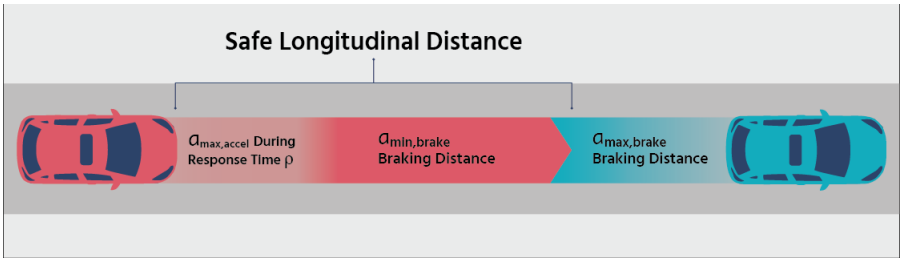

Figure 2.1: Longitudinal Safe Distance of Same Direction [3]

the front vehicle. $a_{min,brake}$ shows the minimal allowed braking acceleration of the rear vehicle, which means to keep the current longitudinal distance safe, the rear vehicle should brake with an acceleration larger equal $a_{min,brake}$. So the above expression implies that, if the front vehicle brakes with its maximal braking acceleration, while the rear vehicle accelerates with an acceleration smaller equal its maximal acceleration, after a reaction time, the rear vehicle starts braking with a minimal braking acceleration, and the two vehicles do not collide with each other. So to keep the current longitudinal distance safe, the rear vehicle should, according to RSS, accelerate with no more than $a_{max,accel}$ before reaction, and brake with no less than $a_{min,brake}$ after reaction.

2.1.2 Longitudinal Safe Distance (Opposite Direction)

If two vehicles drive in the opposite direction, with velocity v_1, v_2 (suppose $v_1 > 0$; $v_2 < 0$), and the reaction time is ρ, minimal braking accelerations are $a_{min,brake,correct}$ and $a_{min,brake}$ ($a_{min,brake,correct}$ is set to be the reasonable braking force expected from the car driving in its correct lane and direction, and $a_{min,brake,correct} < a_{min,brake}$, because the car driving in its correct lane is expected to brake more moderately than the other car in its wrong direction [3]), acceleration of both vehicles before reaction is $a_{max,accel}$, then the expression for the longitudinal distance is:

$$d_{min} = \frac{v_1 + v_{1,\rho}}{2}\rho + \frac{v_{1,\rho}^2}{2a_{min,brake,correct}} + \frac{|v_2| + v_{2,\rho}}{2}\rho + \frac{v_{2,\rho}^2}{2a_{min,brake}}, \tag{2.2}$$

where $v_{1,\rho} = v_1 + \rho a_{max,accel}$, $v_2 = |v_2| + \rho a_{max,accel}$. The above formula shows that to ensure the safety of current longitudinal distance in opposite directions, both vehicles should not accelerate with acceleration more than $a_{max,accel}$ before reaction, and should not brake with acceleration less than $a_{min,brake,correct}$ and $a_{min,brake}$.

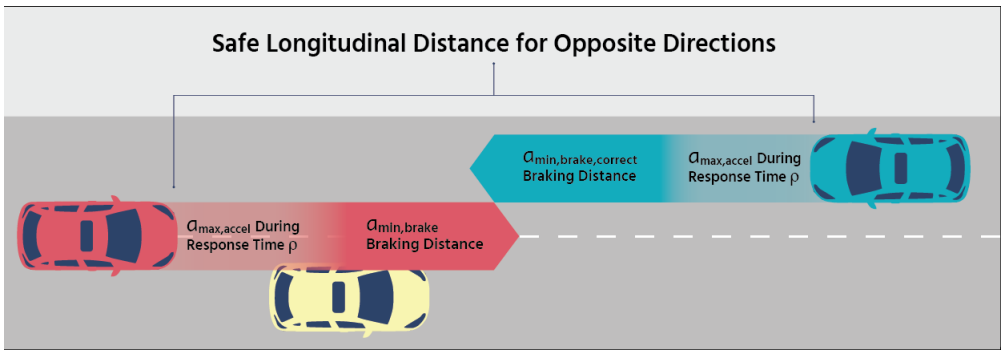

Figure 2.2: Longitudinal Safe Distance of Opposite Direction [3]

2.1.3 Lateral Safe Distance

If two vehicles with velocities v_1, v_2 drive next to each other, and the reaction time is ρ, minimal lateral braking acceleration is $a^{lat}_{min,brake}$, maximal lateral braking acceleration is $a^{lat}_{max,accel}$, and the final lateral distance between the two vehicles should be μ, then the lateral safe distance can be expressed with the following formula:

$$d_{min} = \mu + \left[\frac{v_1 + v_{1,\rho}}{2}\rho + \frac{v_{1,\rho}^2}{2a^{lat}_{min,brake}} - \left(\frac{v_2 + v_{2,\rho}}{2}\rho - \frac{v_{2,\rho}^2}{2a^{lat}_{min,brake}} \right) \right]_+, \tag{2.3}$$

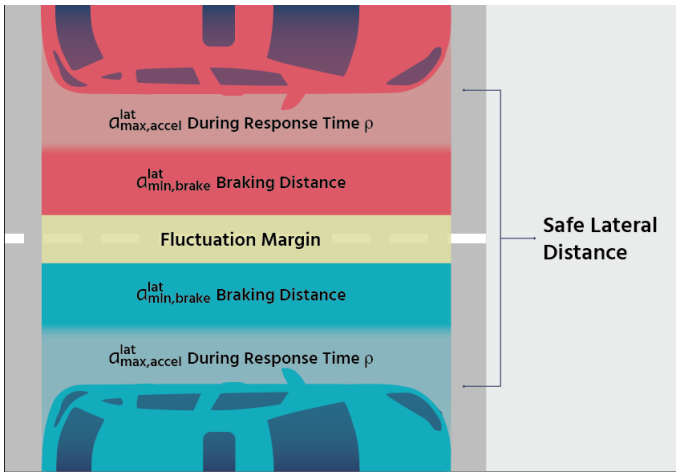

Figure 2.3: Lateral Safe Distance [3]

where $v_{1,\rho} = v_1 + \rho a^{lat}_{max,accel}$, $v_{2,\rho} = v_2 - \rho a^{lat}_{max,accel}$. This means, to keep the current lateral distance safe, and to ensure a distance μ between the two vehicles at the end of the reaction, the ego vehicle should not accelerate faster than $a^{lat}_{max,accel}$ before reaction, and should not brake with acceleration less than $a^{lat}_{min,brake}$.

2.2 Online Verification (CommonRoad)

CommonRoad is a collection of composable benchmarks for motion planning on roads [1]. It provides useful tools for set-based prediction, road boundary generation, collision check, route planning, fail-safe trajectory planning, motion primitives generation etc. Online verification of motion planning can be done with CommonRoad tools on CommonRoad Scenarios. Long-term trajectories can be verified whether they are legally safe, and if not, fallback solutions are provided [2]. Here the basic methods of set-based prediction and fail-safe trajectory planner are briefly introduced, which are the main parts of online verification.

2.2.1 Set-based Prediction of Traffic Participants (SPOT)

Set-Based Prediction of Traffic Participants [2], a tool which predicts the future occupancy of other traffic participants based on reachability analysis, including all possible maneuvers, by considering physical constraints and assuming that the traffic participants abide by the traffic rules [5]. In this project, SPOT generates the occupancy set of dynamic obstacles in a certain time horizon. Then the occupancy sets are used in collision checks, and the calculation of time to react. Occupancy sets of dynamic obstacles consists of three parts: acceleration-based occupancy O_1, lane-following occupancy O_2, and safe distance space O_3 [5]. The meanings of the three occupancy sets are briefly explained here:

- O_1: The acceleration-based occupancy of an obstacle takes only its current position, velocity and the limit of its absolute acceleration into account. The reachable area during a time interval can be calculated according to these information, and a convex polygon of the resulting area is taken as an over-approximation. The O_1 occupancy of a dynamic obstacle is usually the yellow polygon shown in Figure 2.4.

- O_2: The lane-following occupancy of an obstacle assumes that the obstacle follows traffic rules, maximal velocity and maximal engine power. For example, it drives

[1] https://commonroad.in.tum.de/
[2] https://commonroad.in.tum.de/spot

only in the positive driving direction, and drives within reachable lanes of the road network. The resulting O_2 occupancy is usually the blue area shown in Figure 2.4.

- O_3: The safe distance occupancy is a solution to reduce the final occupancy of a dynamic obstacle, and to provide more space for the planning of ego vehicle's trajectories. According to the Vienna Convention on Road Traffic, a vehicle can only change lanes if this action does not endanger the following vehicles. This safety can be ensured by leaving out a safe distance based on the pair of relevant vehicles. Thus, for a dynamic obstacle that is on any of the adjacent lanes of the ego lane, an occupancy is calculated according to the safe distance between the obstacle and the ego vehicle. This occupancy means that the obstacle should not appear in this area in a certain time interval, otherwise its lane changing action is against the Vienna Convention on Road Traffic. O_3 of a dynamic obstacle is usually the red area shown in Figure 2.4.

The final set-based prediction of a dynamic obstacle is a combination of the above three occupancies. An obstacle should not only follow its maximum acceleration but also the traffic rules, which means the intersection of O_1 and O_2. At the same time it should leave out the safe distance occupancy to avoid illegal lane changing. So the combined occupancy can be expressed as:

$$O(t) = O_1(t) \cap O_2(t) \cap O_3^C(t). \tag{2.4}$$

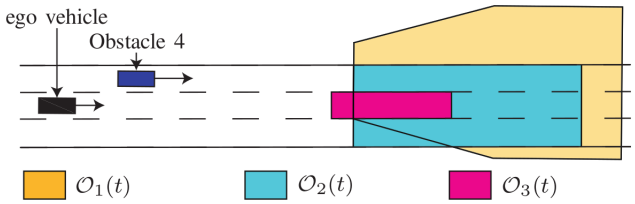

Figure 2.4: Occupancy sets in SPOT [5]

2.2.2 Reactive Planner

Reactive Planner [3] plans a trajectory for the ego vehicle based on a CommonRoad scenario. It is sampling based planner and works in a frenet frame [6]. For trajectory

[3] https://gitlab.lrz.de/cps/reactive-planner

planning, it first samples a bundle of fifth degree polynomials as trajectories. Then kinematics check and collision check are applied on the generated trajectories to remove invalid candidates. An optimal trajectory is then selected according to its cost. In this project, reactive planner is used to generate a fail-safe trajectory in each CommonRoad scenario converted from Apollo.

3 Foundation

3.1 Apollo6.0

Baidu Apollo is an open-source platform for autonomous driving [1]. It provides algorithms for autonomous driving modules such as perception, prediction, localization, routing, planning, vehicle control, vehicle operating systems, as well as a complete set of testing tools and other functions. Algorithms can be modified with or replaced by self-developed ones. The system can be used for simulation or on real vehicles. If Apollo is used for simulation, its visualization tool "Apollo Dreamview" helps to visualize Apollo maps, traffic flow and prediction for obstacles, the ego vehicle, as well as planned routing and trajectories. Through the result of visualization, the functionality of important modules can be checked and debugged. Here the input and output of important modules are listed:

- Perception: The perception module receives point cloud information of obstacles from sensors, and uses learning-based method to output the states (positions, velocities, accelerations etc.) and shapes of perceived obstacles.
- Prediction: The prediction module takes the states of obstacles as input, and predicts a most probable trajectory for each obstacle according to its current position, velocity and acceleration for a time horizon of 8s using learning-based method.
- Localization: The localization module provides the current state (position, velocity, acceleration etc.) of the ego vehicle at a very high frequency.
- Routing: The routing module receives a routing request, which is a set of waypoints, as input. These waypoints can be given from the users by selecting points on "Apollo Dreamview". A waypoint can be described by the coordinate of its position, the id of the lane that this point is on, and the distance of this point to the start of the lane. Then the routing module finds the a route that follows the road network and passes through all the waypoints. The coordinates of the lane center vertices on this route are the output of Routing.

[1] https://github.com/ApolloAuto/apollo

- Planning: The planning module generates trajectories for the ego vehicle. A trajectory should contain states of the ego vehicle in certain time steps, so that the ego vehicle follows roughly a reference path and avoids collisions as much as possible. A state shows the position, velocity and acceleration etc. of the ego vehicle at a time step. To generate trajectories, the planning module needs input information such as current state of the ego vehicle given by Localization module, routing response, which is also used as reference path, given by Routing module, possible future trajectories of surrounding obstacles given by Prediction module.
- Control: The control module receives the planned trajectories and gives control commands to steering, throttle and brake etc. to make the ego vehicle follow the planned trajectories.

As can be seen from above, different modules of Apollo requires information from each other. So the modules need to communicate with each other. In some other autonomous driving systems, a common middle-ware ROS is used for managing the communications among modules. Apollo has developed a middle-ware Cyber RT by themselves. Apollo Cyber RT is an open source runtime framework designed specifically for autonomous driving scenarios [2]. This new middle-ware is developed to adapt to the system of autonomous driving rather than robots. An effective centralized computing model is needed here to ensure high concurrency and low latency. With Cyber RT, Apollo modules can exchange information by subscribing or publishing to cyber channels. Modules publish their outputs to a channel by creating messages according to predefined google protobuf message structures. Then the modules that need the corresponding information as input, should subscribe to this specific cyber channel. A google protobuf message class is usually a class that has listed attributes with types such as "optional", "repeated" etc. After compilation, it provides C++ and Python interface, where the class of protobuf corresponds to a C++ class or a Python class. A "repeated" attribute of this class is usually a vector in C++ or a list in Python. Also, each type of attribute automatically has some common functions, for example, "repeated" attributes have add and clear functions, where add function adds an element at the end of the vector/list, and clear function clear all the existing elements. In this project, Apollo 6.0 is chosen, because its Cyber RT supports Python3 API.
Apollo modules such as Perception, Prediction, Localization, Routing and Planning etc. all have their own google protobuf message classes. Each message carries the output information of the module. And each module sends out its messages to its cyber channel with a certain frequency. The following Table 3.1 shows the cyber channel, the google protobuf message class and the sending frequency of important Apollo modules.

[2] https://github.com/ApolloAuto/apollo/blob/master/docs/cyber/CyberRT_FAQs.md

Table 3.1: Cyber Messages of Apollo Modules

Apollo Module	Cyber Channel	Message	Frequency
Perception	/apollo/perception/obstacles	PerceptionObstacles	10Hz
Prediction	/apollo/prediction	PredictionObstacles	10Hz
Localization	/apollo/localization/pose	LocalizationEstimate	100Hz
Routing	/apollo/routing_response	RoutingResponse	Once
Planning	/apollo/planning	ADCTrajectory	2Hz

3.2 LGSVL

LG Simulator is an open-source unity-based autonomous vehicle simulator for testing autonomous vehicle algorithms. It can generate HD Maps, and be used for system testing and validation [3]. Apollo6.0 already supports LG Simulator in its official source code. In this project, LG Simulator is used to provide HD Maps, and traffic participants that can interact with the ego vehicle.

3.3 RSS Library in Apollo

Intel provides a C++ library that implemented RSS for autonomous driving named "ad-rss-lib" [4]. The input of this rss library is processed sensor information, and it outputs restrictions on vehicle actuators, such as the maximal allowed longitudinal and lateral accelerations etc. The processed sensor information refers to a list of objects (surrounding traffic participants in RSS) near the ego vehicle. These RSS objects are created with sensor information given by platforms such as Apollo. To create RSS objects, users should implement a converter by themselves that form RSS objects with the sensor information in their own platforms. Apollo 6.0 is already integrated with the ad-rss-lib and provided a converter from Apollo world to RSS world. After receiving the RSS objects, the RSS module creates "situations" for each object-ego vehicle pair. These situations are then used for safety check and proper response calculation. For the output, the RSS module combines the responses of all object-ego vehicle pair and

[3]https://www.svlsimulator.com/
[4]https://github.com/intel/ad-rss-lib

returns longitudinal and lateral limits on accelerations. The outputs still needs to be converted into actuation commands of automated vehicles, which should also be implemented by the users according to their specific software and hardware of the vehicles. Apollo 6.0 also provides this part of implementation.
Apollo 6.0 uses RSS modules to calculate longitudinal and lateral limitations of accelerations and to check if current state is rss-safe. This process is defined as a decision task during trajectory planning and can be enabled or disabled easily with a flag.

3.4 RSS vs. Online Verification

RSS and the CommonRoad Online Verification method are both used as a deterministic safety layer on top of the learning-based operation in autonomous driving systems. In each cycle of trajectory planning, RSS takes the current states of the ego vehicle and its front obstacles to calculate limits of accelerations, so that appropriate actuation commands can be given from the control module of automated vehicles to make sure it keeps safe distances with its current front vehicles. In this process, a potential dangerous situation may occur, because the current states are not taken continuously, there are about 0.5 seconds between two cycles of trajectory planning. Although RSS takes the worst case assumption and can ensure the safety of the ego vehicle in a time interval, it only checks the safe distances of the front obstacles of the current position of the ego vehicle. If during the 0.5s time interval between two neighboring cycles, an obstacle becomes a new "front obstacle", and with the join of this front obstacle, the limits of accelerations should have been updated. In this case, the ego vehicle is already driving in an unsafe situation. Online Verification is however not limited by the time interval between planning cycles. It takes all surrounding obstacles into account, and in each cycle, several over-approximated occupancy sets of each obstacle during the dt of the scenario are calculated. These occupancy sets already contains all legal positions of the obstacles during the whole time covered by the fail-safe trajectory. Thus, even if the next planning cycle never starts due to some reasons, the ego vehicle can still safely drive according to the last fail-safe trajectory until it reaches the final state with velocity 0.

The computation effort of RSS is much smaller than Online Verification. In some critical cases, occupancy sets of obstacles may cover a large amount of driving area, which leads to again a large time consumption for sampling fail-safe trajectories to fulfill the restricted constraints of set-based prediction.

4 Approaches

The basic method for the performance test is to first use LG bridge to connect Apollo6.0 for the maps and traffic flow as shown in Figure 4.1. Then Apollo Perception module receives point clouds from LG, and output perceived obstacles. Apollo Prediction module calculates the most probable predicted trajectory for each perceived obstacles. Then Apollo Planning module generates a long-term trajectory according to the predicted obstacle trajectories, and sends the trajectory and the map around the ego vehicle to a redirected Cyber channel. Then a Python interface outside of docker listen to the messages from Apollo inside docker, and call CommonRoad SPOT to calculate set-based prediction for obstacles and determine the time-to-react for each intended trajectory. And then use CommonRoad reactive planner to generate a fail-safe trajectory, and send the whole trajectory to Apollo Cyber channel for execution.

4.1 Cross Container Communication

This project uses a decoupled method to run CommonRoad SPOT and planner on the Apollo driving platform. So that the CommonRoad working environment is installed outside of Apollo docker container instead of inside the container. This decoupled method saves a lot of installation effort, since Apollo docker container has different tool versions as the versions CommonRoad environment requires. In this section, a method for establishing cross Apollo docker container communication is provided, as well as methods for calling Apollo CyberRT functions in Python scripts outside of Apollo docker container.

4.1.1 IP Configuration

One way to communicate across docker container is to use bridge. Apollo provides a cyber bridge for cross container communication. All we need to do is to set an appropriate CYBER_IP both inside and outside the Apollo docker container. The CYBER_IP inside container in this case does not need to be changed, the default setting should work, which is the IP of localhost: 127.0.0.1. In the similar way, the CYBER_IP outside of container, in my case in the host, should be set to the ip of the docker. This is usually 172.17.0.1. To configure this ip in the host, the following steps maybe helpful.

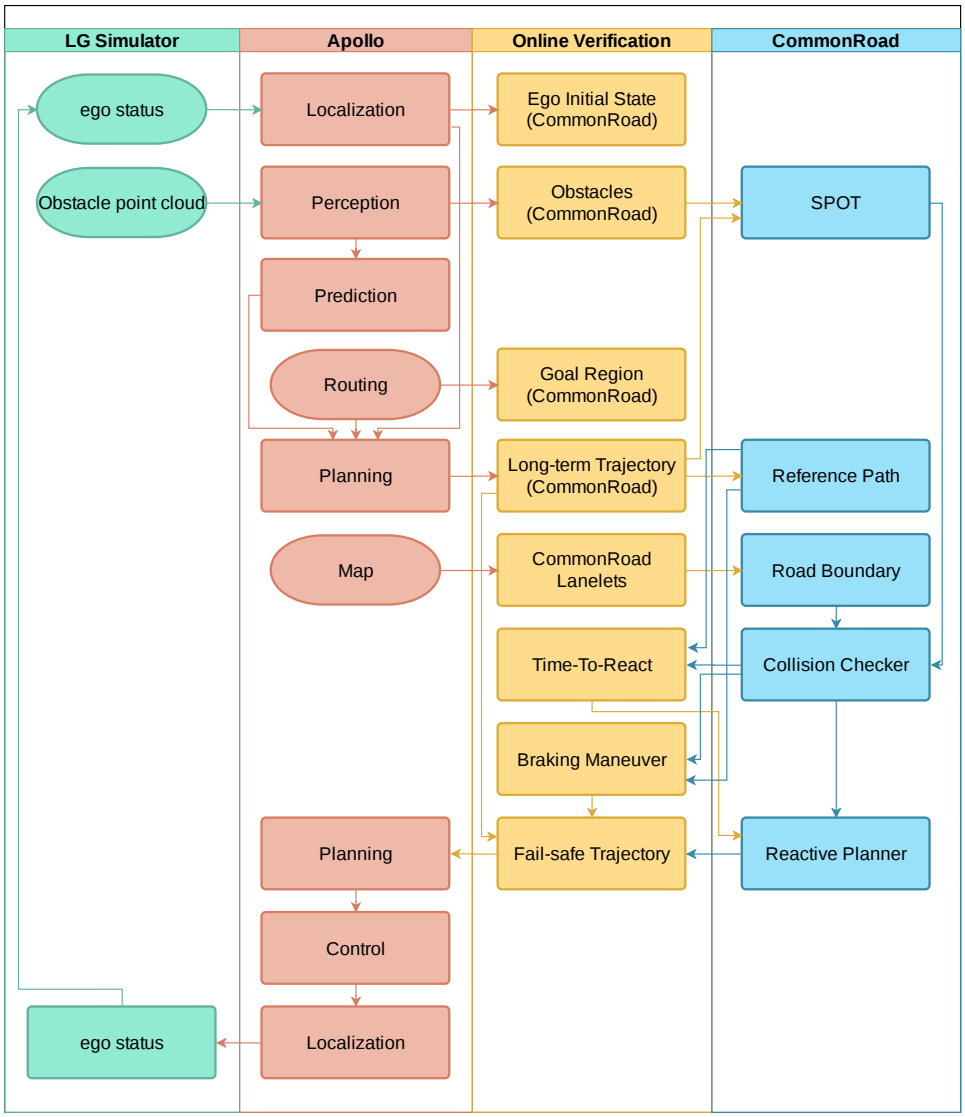

Figure 4.1: Online Verification Call Graph

1. Add line $export\ CYBER\_IP = 172.17.0.1$ to *.bashrc* in host machine.
2. Run command *source .bashrc*.

4.1.2 Cyber Environment Creation

Since the communication across docker container is established, and the goal is to send messages from the host machine with cyber functions. Next step is to create a Cyber environment in the host machine by copying related libraries from inside docker container.

1. Set variables in *.bashrc*:
   i. $export\ APOLLO\_HOME = /apollo$
   ii. $export\ CYBER\_PATH = /apollo/cyber$
2. Add Apollo root directory path and paths of all Python libraries and binaries ($"/apollo/.cache/bazel/540135163923dd7d5820f3ee4b306b32/execroot/apollo/bazel-out/k8-fastbuild/bin"$) to variable $PYTHONPATH$ in *.bashrc* in the host machine.
3. Copy all Cyber-related libraries from inside docker under $/usr/local/lib/$ in host machine.
   i. Names of libraries are listed here
      libfastcdr.so.1.0.7
      libfastrtps.so.1.5.0
      libglog.so.0.4.0
      libgflags.so.2.2.2
      libprotobuf.so.3.12.3.0
      libPocoFoundation.so.71
      libboost_system.so.1.73.0
      libboost_thread.so.1.73.0
   ii. Add line $/usr/local/lib/$ in file $/etc/ld.so.conf$ in the host machine.
   iii. Run command $sudoldconfig$.

4.1.3 Cyber Talker outside of Container

Now that the configuration for cyber environment is finished, an example Python script for calling Apollo CyberRT functions provided in Apollo source code can be used to test the communication: */apollo/cyber/python/cyber_py3/examples/talker.py*. This Python script sends a simple message to a self-created Cyber channel with a frequency of 1Hz.

```
(commonroad2020) apollo_student@skynet:~/apollo$ python cyber/python/cyber_py3/examples/talker.py
stamp: 9999
seq: 0
content: "py:talker:send Alex!"

================================================================================
write msg -> stamp: 9999
seq: 2
content: "I am python talker."

================================================================================
write msg -> stamp: 9999
seq: 3
content: "I am python talker."

================================================================================
write msg -> stamp: 9999
seq: 4
content: "I am python talker."
```

Figure 4.2: Cyber talker sends messages outside of Apollo container.

4.1.4 Message Reception inside Container

In Apollo docker container, run *cyber_monitor* to see received messages in cyber channels. The following figure shows that, inside the docker, CyberRT is receiving cyber messages sent by the Python scripts in host machine to cyber channel *chatter*. Thus, the communication across docker container succeeds.

```
ChannelName: channel/chatter
MessageType: apollo.cyber.proto.ChatterBenchmark
FrameRatio: 1.00
RawMessage Size: 26 Bytes
stamp: 9999
seq: 16
content: I am python talker.
```

Figure 4.3: Cyber messages received inside Apollo Container

4.2 Preparation for the Input of Online Verification

The basic idea of online verification module is to convert Apollo scenarios to CommonRoad scenarios, and use CommonRoad tools such as SPOT, drivability-checker, reactive planner to evaluate long term trajectories of Apollo Planning module and generate fail-safe trajectories. An Apollo scenario, as input information of the online verification module, includes Apollo Perception obstacles, Apollo Routing requests, Apollo Localization, Apollo Planning trajectories and Apollo Map. The Perception obstacles will later be used in creating CommonRoad obstacles in OV for SPOT and collision checker. The Routing requests contains the goal point of ego vehicle and thus will be used in creating the goal status of CommonRoad PlanningProblem for fail-safe trajectory generation. The Localization is the current state of the ego vehicle, and will be used as the initial state of CommonRoad PlanningProblem. The Planning trajectories from Apollo act as long term trajectories in CommonRoad. Finally, the Apollo Map will be converted to CommonRoad lanelets. The above information can be transferred from Apollo to the OV module outside of Apollo container in the form of cyber messages by subscribing to corresponding cyber channels. Some of the messages can be created by Apollo modules without extra inputs, such as Localization and Planning module. However, modules like Perception obstacles and Routing request need extra input information in order to create messages to their cyber channels. In a simulation, Perception obstacles can be either generated based on point clouds given by simulators such as LG, or directly calculated from json description files, which contain desired waypoints, speeds and headings of simulated obstacles. Routing request can also be given by two ways. One way is to specify desired waypoints for the ego vehicle in Apollo Dreamview, and a *RoutingRequest* message with these waypoints will be sent to cyber channel */apollo/routing_request*. The simplest routing request only contain two waypoints, a start point and a goal point. If we want to send the same waypoints as routing_request for every simulation test, an easier way is to use a Python script that contains the coordinates of the waypoints to send a fixed *RoutingRequest* message to the cyber channel. The remaining input to be given is the Apollo Map. Apollo does not send its map information to a cyber channel, because its Planning module have direct access to the current map pointer. However, the Online Verification module also needs map, and it can only transfer information with Apollo through cyber channels, so the additional map messages will be generated in Apollo Planning module to a self-created cyber channel */apollo/map_cr*. The following sections explain in detail how the Perception obstacles, Routing requests and Apollo Map messages can be sent to cyber channels, so that the input for the online verification module is well-prepared.

4.2.1 Fake Perception Obstacles

If no simulator provides obstacle information, we can use json files to describe some fake perception obstacles. The following table shows an example of the attributes and their values of an obstacle in a json file. As can be seen in the table, an obstacle id, the

Table 4.1: Description of obstacles in json files

Attribute Name	**Attribute Value**
id	995
position	[585787.62279974029, 4139963.4104115185, 0.0]
theta	0.258770929
length	4.0
width	2.0
speed	6.0
type	"VEHICLE"
trace	[[585787.62279974029, 4139963.4104115185, 0.0], [585802.21814418526, 4139967.3825623556, 0.0], [585816.72111500509, 4139972.0013431921, 0.0], [585827.62143681687, 4139982.9016693495, 0.0], [585832.8868465398, 4139998.6055374243, 0.0], [585837.32084362186, 4140015.6027084333, 0.0]]

initial state, the initial heading and the shape of the obstacle should be specified. If the type of the obstacle is expected to be a dynamic obstacle, then its trace should also be specified. The trace is a list of waypoints that contains the start and end point of this obstacle, and some additional points between them. These points only describes the route of the obstacle, and are not given according to time steps or distances between points.

To use the json description file to create cyber messages for Apollo Perception obstacles, a Python script is written to import *cyber* from outside of Apollo docker container and to create a cyber node that talks to cyber channel */apollo/perception/obstacles*. The perception messages are sent every 0.1s, which keeps the default message frequency of Apollo Perception module. Each message contains status information of all perceived obstacles at the current time. So the status of all perceived obstacles are updated every 0.1s by sending the newest message to the perception cyber channel. Thus, the Python script needs to create instances of Perception obstacle for each perceived obstacle according to google protobuf message *PerceptionObstacle*. The following figure

shows the necessary attributes of a *PerceptionObstacle* instance. The position of the

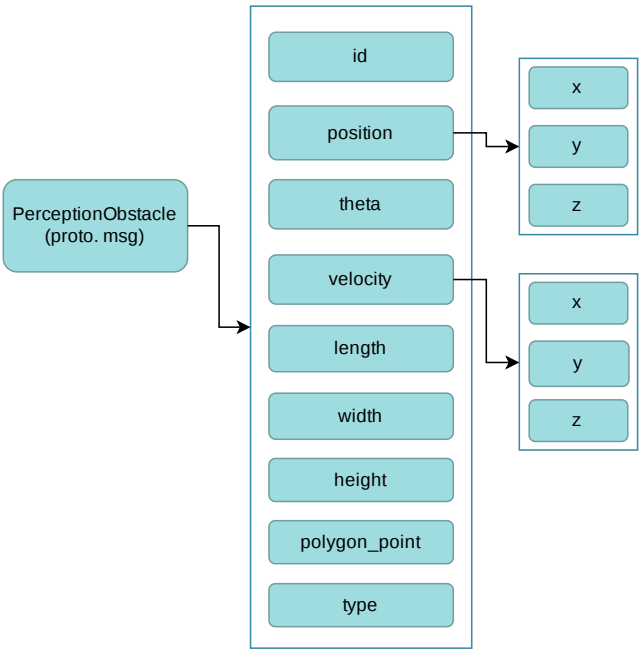

Figure 4.4: Protobuf message of PerceptionObstacles

PerceptionObstacle at every 0.1s can be calculated from the *trace* in json description file. Although the waypoints in *trace* are not given according to time steps or distance, there is an assumption in the json file. All dynamic obstacles are assumed to drive at a constant speed. So starting from the initial position in the json file, if the position of time 0.1s is needed, the obstacle would have the driving distance as 0.1s*speed. Next, find out on which line segment of *trace* this driving distance is, and the coordinate of position can be calculated on this line segment. The same is done when positions of time 0.2s, 0.3s, 0.4s etc. are needed. In this way, every 0.1s, the Python script locates the new status all percepted obstacles according to json file, and sends messages to cyber channel */apollo/perception/obstacles*.

4.2.2 Mock Routing Request

RoutingRequest message is needed for calculating *RoutingResponse* message, which is a necessary input of Apollo planning module for the generation of reference path. Usually, *RoutingRequest* is given by pointing out desired waypoints of the ego vehicle in

Dreamview. But when testing and debugging the online verification module, *RoutingRequest* message is frequently needed, and the same message content is often enough. Pointing out desired waypoints in Dreamview under this circumstance is tedious. So a Python script is written to send *RoutingRequest* messages with coordinates of specific waypoints from outside of Apollo docker container. The message *RoutingRequest* is sent only once at the beginning of each test. For each message, the Python script should create a new *RoutingRequest* instance according to its google protobuf message. The following figure shows the necessary attributes of a *RoutingRequest* instance. From

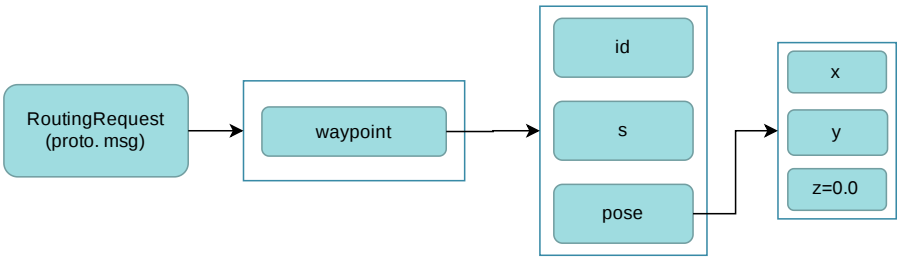

Figure 4.5: Protobuf message of RoutingRequest

the protobuf message of *RoutingRequest* it can be seen that, a list of waypoints are needed to describe a routing request. The first waypoint is the initial position of this planning problem, and the last waypoint is the goal position. So a routing request requires at least two waypoints. For each waypoint, attributes like *id*, *s* and *pose* are needed. *id* refers to the id of the lane where this waypoint is. *s* means the distance from the waypoint to the start point of the lane. Finally, the *pose* is the coordinate of this waypoint and is determined by its x and y. The Python script specifies two waypoints for the *RoutingRequest* instance, and send it as protobuf message to the cyber channel */apollo/routing_request* before the planning process, so that the Apollo planning module can generate long term trajectories according to the Routing information.

4.2.3 C++ Map Component

Using CommonRoad tools such as SPOT and reactive planner requires converted CommonRoad scenarios from Apollo Map and Obstacles. The Obstacles information can be received from listening to corresponding Cyber channel. But Map messages is not originally published by Apollo, additional Cyber writer for local map messages needs to be created. The Map message is obtained in the planning component, and sent out to a self-created Cyber channel */apollo/map_cr*. So CommonRoad tools can have access to local map information through the Python interface.

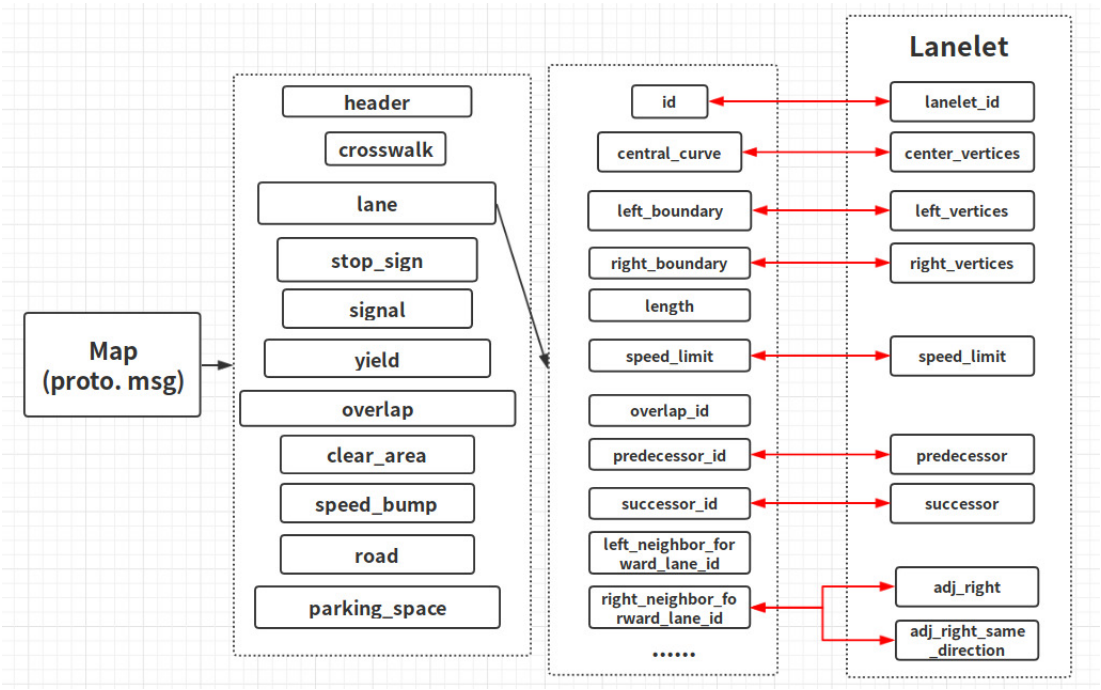

Figure 4.6: Protobuf message of MapLane [7]

4.3 Python Interface

The Python interface listens to Apollo messages from modules: Routing, Localization, Perception, Planning and the self-created Map Component. Whenever there is a message published to cyber channels, a corresponding call back function is executed to convert the information from Apollo objects to CommonRoad objects. Figure 4.7 shows the mapping between CommonRoad scenarios and Apollo scenarios. The left part includes the components that are needed in CommonRoad scenarios, and right part includes Apollo Cyber messages that describe Apollo scenarios. The numbers on the arrows show the sending frequencies of Apollo Cyber messages.

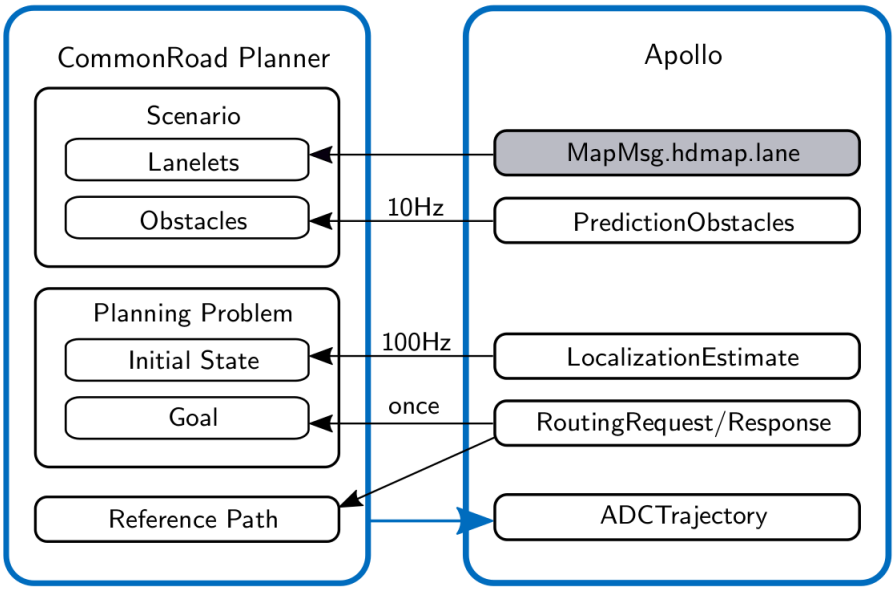

Figure 4.7: Mapping between CommonRoad scenarios and Apollo scenarios [7]

4.3.1 Input Message Conversion

In this subsection, the conversion process of input messages from Apollo modules Routing, Localization, Perception, and Map are explained by Figure 4.8, 4.10, 4.12, 4.14. And the life cycles of the conversion threads are shown in Figure 4.9, 4.11, 4.13, 4.15.

Apollo Routing is converted to goal center of CommonRoad planning problem. Apollo Localization is converted to the initial state of CommonRoad ego vehicle, as shown in Table 4.2. Apollo Perception Obstacle is converted to CommonRoad obstacles, as

shown in Table 4.3. Apollo map is converted to CommonRoad lanelets.

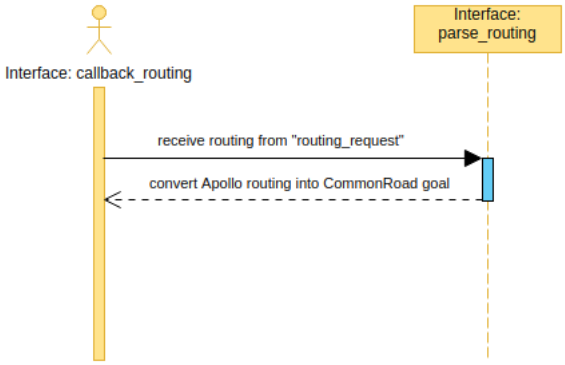

Figure 4.8: Conversion process of RoutingRequest cyber message

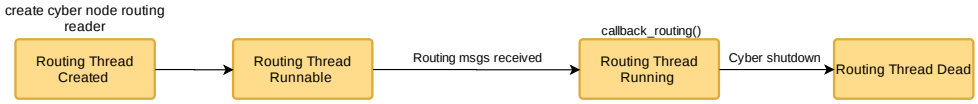

Figure 4.9: Life cycle of Routing thread

Table 4.2: Required information for generating CommonRoad ego vehicle

Apollo Localization	**CommonRoad Ego Vehicle**
position.x: double	initial_state.position: np.array
position.y: double	
linear_velocity.x: double	initial_state.velocity: double
linear_velocity.y: double	
heading: double	initial_state.orientation: double
linear_acceleration.x: double	initial_state.acceleration: double
linear_acceleration.y: double	

4.3.2 Planned Trajectory Conversion and Fail-Safe Trajectory Generation

And finally Apollo planning is redirected to a self-created Cyber channel, so that Apollo Control module does not directly execute the original Apollo trajectory. The

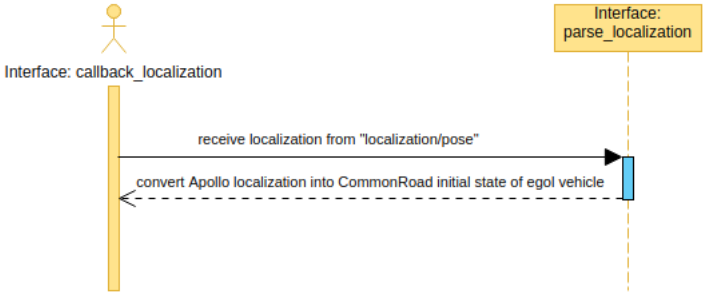

Figure 4.10: Conversion process of Localization cyber message

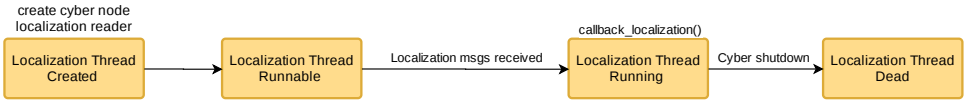

Figure 4.11: Life cycle of Localization thread

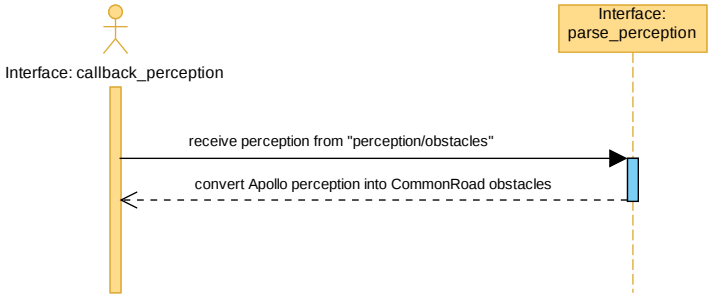

Figure 4.12: Conversion process of PerceptionObstacles cyber message

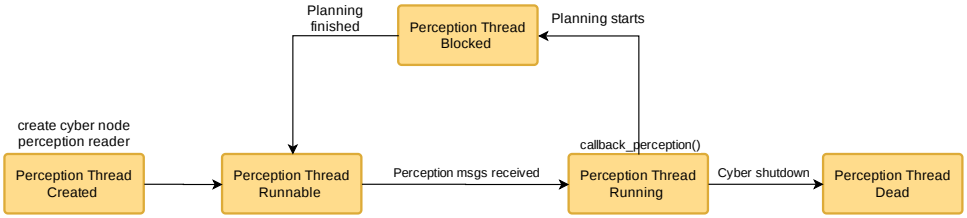

Figure 4.13: Life cycle of PerceptionObstacles thread

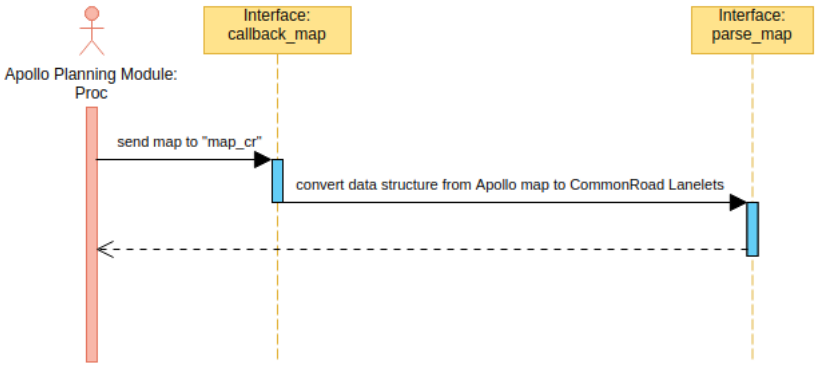

Figure 4.14: Conversion process of Map cyber message

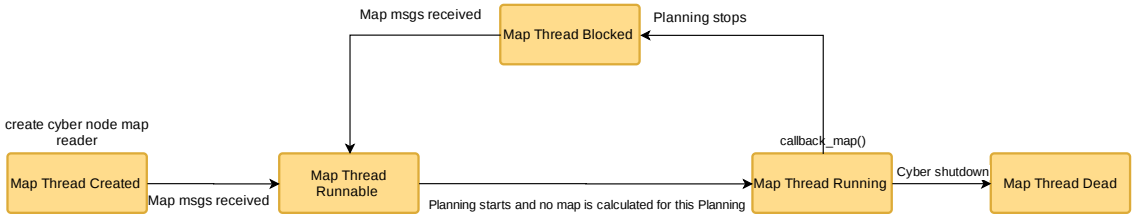

Figure 4.15: Life cycle of Map thread

Table 4.3: Required information for generating CommonRoad obstacles

Apollo Perception	**CommonRoad Obstacles**
position: double	initial_state.position: np.array
velocity: double	initial_state.velocity: double
theta: double	initial_state.orientation: double
acceleration: double	initial_state.acceleration: double
width: double	obstacle_shape: Rectangle
length: double	
type: int	obstacle_type: int

Table 4.4: Required information for generating CommonRoad lanelets

Apollo Map	**CommonRoad Lanelets**
id: string	lanelet_id: int
central_curve: Curve	center_vertices: np.array
left_boundary: LaneBoundary	left_vertices: np.array
right_boundary: LaneBoundary	right_vertices: np.array
speed_limit: double	speed_limit: double
predecessor_id: string	predecessor: int
successor_id: string	successor: int
left_neighbor_forward_lane_id: string	adj_left: int
	adj_left_same_direction: bool
right_neighbor_forward_lane_id: string	adj_right: int
	adj_right_same_direction: bool

Python interface converts the first 4s (all parameters are listed in Section 5.2) of Apollo long-term trajectory into CommonRoad intended trajectory with time step 0.2s.

A reference path is resampled and extended according to the intended trajectory.

Then the interface calls CommonRoad SPOT to generate set-based occupancy set for each dynamic obstacles. The Occupancy of obstacles is computed every time interval 0.2s, for 4+2=6s (length of intended trajectory + length of fail-safe trajectory to be generated). The road boundary is next created according to the converted CommonRoad lanelets. An example of created road boundary is shown in Figure 4.16.

Then the occupancy set of obstacles and the road boundary form a collision checker.

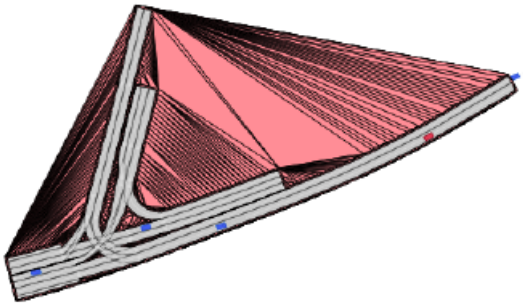

Figure 4.16: Roadboundary of map Sunnyvale Loop

This collision checker is used to determine time-to-react, by calculating a braking maneuver with maximum breaking acceleration at each time step and find the first braking trajectory that has collision.

Then call reactive planner to generate a collision free fail-safe trajectory from time-to-react. If no optimal solution is found by the planner, then the last valid braking maneuver is taken as fail-safe trajectory.

Finally, the fail-safe trajectory and the first part of intended trajectory is concatenated and converted back to an Apollo planning trajectory. A relative time offset is calculated to compensate the time consumption during the whole planning process.

The converted planning message is then published to Apollo planning Cyber channel, and executed by Apollo Control.

Table 4.5: Required information for CommonRoad trajectories

Apollo Planning	**CommonRoad Trajectory**
path_point.x: double	position: np.array
path_point.y: double	
path_point.theta: double	orientation: double
a: double	acceleration: double
v: double	velocity: double
relative_time: double	time_step: int

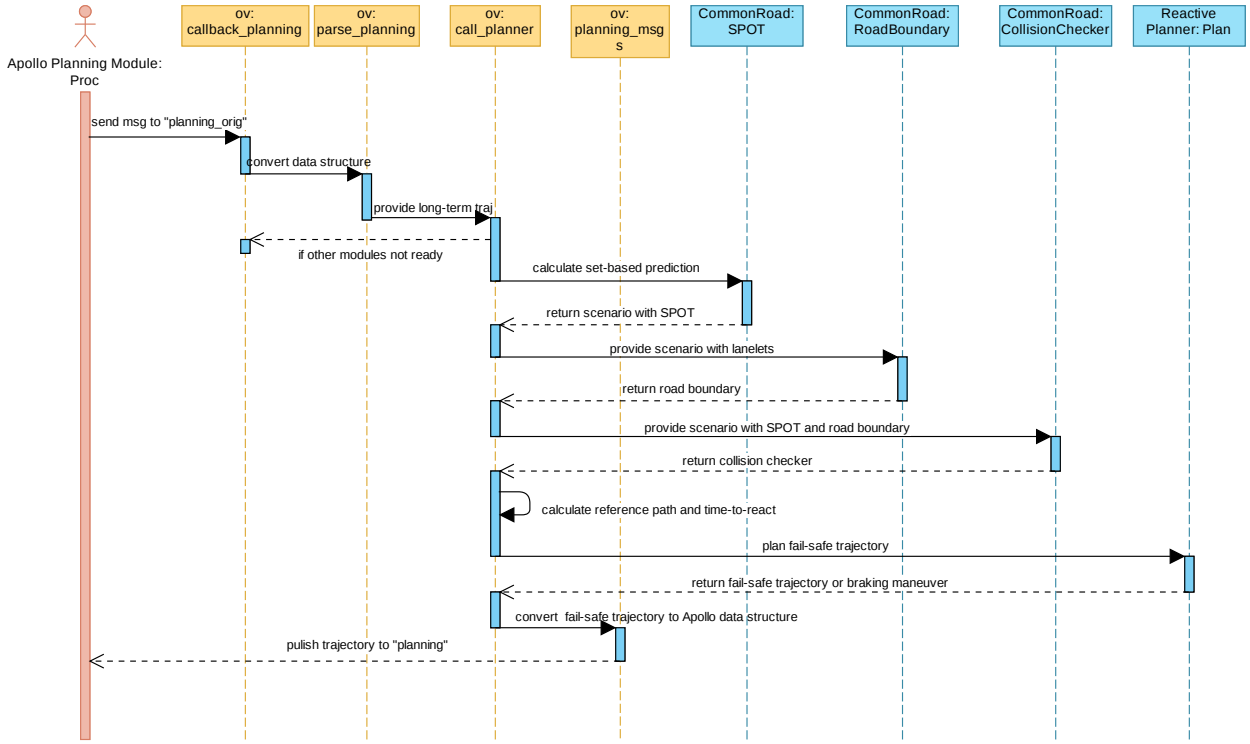

Figure 4.17: Process of planning conversion and fail-safe trajectories generation

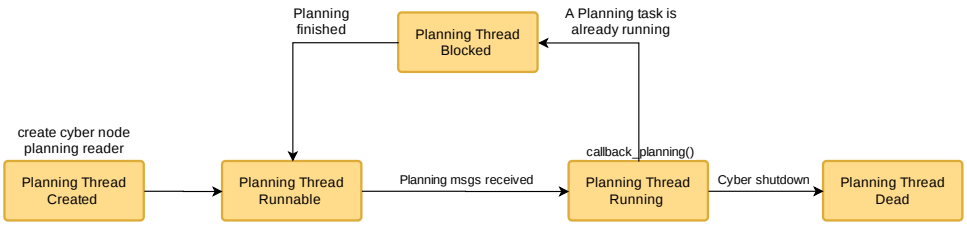

Figure 4.18: Life cycle of Planning thread

4.4 Module Time Consumption

Since Apollo is a real-time system, the computation time of the whole online verification framework is an essential problem. A new fail-safe trajectory must be successfully published before the last fail-safe trajectory is executed to the end. Table 4.6 shows the original average computation time of each module in the Python interface. The total time consumption of a planning cycle is already 3.14s, which is already very large since the valid time covered by a fail-safe trajectory is 2s-6s (2s if the long-term trajectory is unsafe from the first time step, 6s if all states of the long-term trajectory are safe). It is problematic if a new fail-safe trajectory cannot be provided before the last one is executed to the final state. To reduce the time consumption, some modifications are

Table 4.6: Time Consumption of each Thread

Average time consumption for generating a single fail-safe trajectory of 154 scenarios				
3.14s				
Localization	Map	Perception	Planning	
8.86×10^{-5}s	5.22×10^{-2}s	6.15×10^{-4}s	1.873s	
			spot per obstacle	0.08s
			road boundary	0.0884s
			collision checker	0.2s
			reference path	0.046s
			ttr	0.2583s
			plan	0.806s (sampling level: 5)

done in this project.

- CommonRoad SPOT + Collision checker:
  The Python interface uses CommonRoad tools SPOT and collision checker to calculate set-based prediction and form a collision checker instantiation that is later used for determining time-to-react and generating fail-safe trajectories. As can be seen from Table4.6, the time consumption for calculating spot per obstacle and creating collision checker are both high. SPOT and collision checker are both tools written in C++, the connection between them is done by CommonRoad scenarios written in Python. The slowest part is converting C++ occupancy vertices calculated by SPOT into Python objects, and updating the obstacle predictions in CommonRoad scenarios, then the Python objects are again converted into C++ vertices in order to use the C++ collision checker tool. In this project, the above

conversions are no longer done with the Python interfaces provided by the two tools. An additional function is written to directly use the output C++ vertices of SPOT to generate a collision checker instantiation.

- Time-To-React (ttr) determination:
  Originally, the time-to-react is determined by generating braking maneuver from every time step of the long-term trajectory, and see if the braking maneuver collides with the corresponding set-based prediction of obstacles. The first time step that has a collision is set to be ttr. In this project, a binary search is used to search this time step.

After these modifications are applied to corresponding tools, the new computation time can be seen in Table 4.7.

Table 4.7: New Time Consumption after Modifications

Average time consumption for generating a single fail-safe trajectory of 154 scenarios				
1.03s				
Localization	Map	Perception	Planning	
8.85×10^{-5}s	4.33×10^{-2}s	6.15×10^{-4}s	0.734s	
			spot per obstacle	0.035s
			road boundary	0.096s
			collision checker	0.007s
			reference path	0.046s
			ttr	0.054s
			plan	0.374s (sampling level: 3)

4.5 Performance Comparing Method

4.5.1 CommonRoad Scenarios Conversion

To test the performance of CommonRoad Online Verification framework and Intel RSS on Apollo driving platform, various scenarios are needed. One way to obtain large amounts of maps and obstacles is to use CommonRoad scenarios.

Map Conversion

CommonRoad scenarios use lanelets to describe maps. Each lanelet has left and right vertices, and its center vertices. All these information for one scenario can be stored in an xml file. Apollo Map is consist of several bin files. These bin files has format Apollo OpenDRIVE Map, which is a variation of standard OpenDRIVE Map. Several options have been tried for the conversion of CommonRoad Map to Apollo Map.

a) Directly convert CommonRoad xml file to Apollo in file: There is no such tool that can do this conversion so far.

b) Convert xml file to osm lanelets (lanelet2), then convert osm lanelets to OpenDRIVE, and finally use Apollo map tools to convert OpenDRIVE to bin files (Apollo OpenDRIVE).

- CommonRoad xml $\rightarrow$ osm lanelets (lanelet2):
  The conversion from CommonRoad xml to osm lanelets can be done by tool opendrive2lanelet. The tool opendrive2lanelet can convert map information from OpenDRIVE to CommonRoad lanelet, but the inverse conversion, from CommonRoad lanelet to OpenDRIVE is not available. However, it is still worth trying to first convert CommonRoad lanelet to osm lanelets, and then use other tools to convert osm lanelets to OpenDRIVE. The tool opendrive2lanelet enables the conversion from CommonRoad lanelet to osm lanelets by running the following command.

  "osm-convert -r demo.xml -o demo.osm -a"

  The *demo.xml* in the above command is the path to the input CommonRoad xml file, and the *demo.osm* is the path to output, the converted osm lanelet file.

- Osm lanelets (lanelet2) $\rightarrow$ OpenDRIVE:
  Sumo provides a tool: netconvert that can convert osm lanelets to OpenDRIVE.

  "./netconvert –osm emo.osm –opendrive-output=demo.xodr"

- OpenDRIVE $\rightarrow$ Apollo bin file (Apollo OpenDRIVE): Apollo source code provides a tool *proto_map_generator*. This tools takes OpenDRIVE file as input, and generates Apollo map files including bins and txts. The bin files contain maps that can be read by other modules of Apollo. And the txt files shows human-readable contents of bin files. These files mainly contain lane information such as center vertices, left/right boundaries, and predecessors/successors of lanes. The following command shows the usage of the Apollo tool *proto_map_generator*:

"bazel-bin/modules/map/tools/proto_map_generator −−map_dir=[]
−−output_dir=[]"

However, this tool has a high demand for the completeness of the input OpenDRIVE file. Many of the required variables are usually not given by the OpenDRIVE files that are converted from sumo tool *netconvert*.

In conclusion, converting a CommonRoad xml file with process: $CommonRoad xml \rightarrow osmlanelets \rightarrow OpenDRIVE \rightarrow Apollo bin$ using the above mentioned tools is currently infeasible.

c) Use LG simulator unity project to convert osm/OpenDRIVE to Apollo bin.
LG simulator can be downloaded in two forms: one is directly the executable *lgsvlsimulator-linux64-2020.06.zip*, and another one is the source code of LG simulator unity project *sourcecode.zip*. In this method, the LG simulator unity project is required, because some unity plug-ins implemented by LG are needed for map conversion. Steps for installing unity and building the LG simulator unity project can be found here. $(https://www.lgsvlsimulator.com/docs/build-instructions)$ The step that needs to be especially noticed is including Mono support for both Windows and Linux when installing Unity. This step enables LG simulator plug-ins. And the functions for converting maps is exactly one of these LG simulator plug-ins. The plug-in for converting maps from osm/OpenDRIVE to Apollo bin can be found under the tab *simulator*, select the option $ImportHDMap$, and choose an osm/OpenDRIVE map as import file (Figure 4.19). After the importing is successfully done, select the option $ExportHDMap$ and give the path for the targeted output Apollo bin file (Figure 4.20). Here are some options for using the LG simulator unity project to generate Apollo bin maps.

- Choose the OpenDRIVE file that is converted from lanelets by opendrive2lanelet and sumo, as mentioned in b), as input file. The input process can be done successfully. However, after exporting, the resulting file is an empty file. After the importing is successfully done, select the option $ExportHDMap$ and give the path for the targeted output Apollo bin file.

- Choose the osm file that is converted by tool opendrive2lanelet as input file. To convert osm files, select the import format as *lanelet*2 rather than $OpenDRIVEMap$. Figure 4.21 shows the map in LG simulator after successfully importing the osm file. The import map is originally converted from a CommonRoad xml file $BEL\_Wervik-1\_4\_T-1.xml$. From the visualization in the LG simulator unity project, it can be seen that the lanelets themselves are correctly converted and look the same as the original CommonRoad lanelets. Also, after exporting the

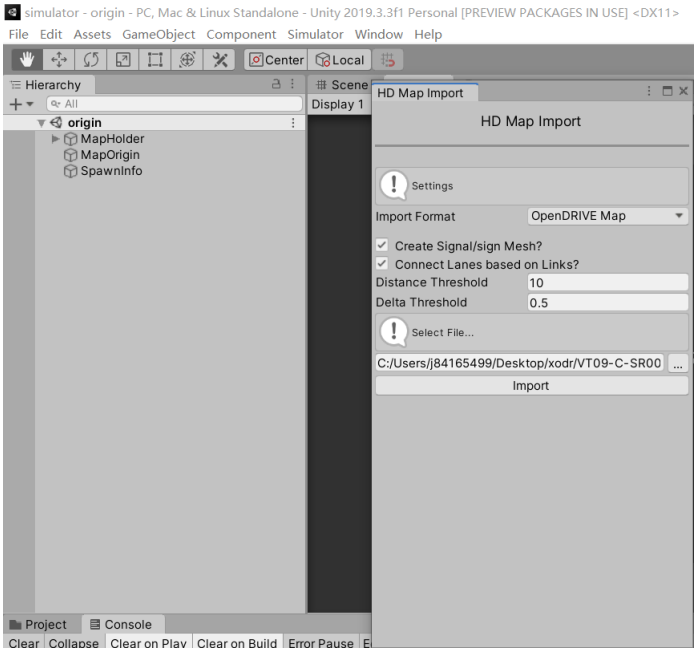

Figure 4.19: Import osm files in LG Simulator

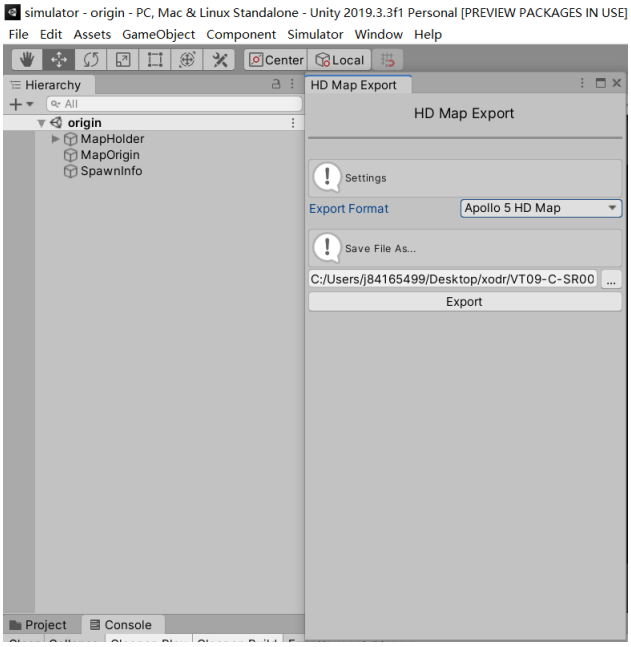

Figure 4.20: Export Apollo bin maps from LG Simulator

file to Apollo bin maps, no errors occurred. However, the resulting Apollo map did not correctly captured all the lanelet relationships, such as adjacent lanelets, including merging and forking, and predecessor / successor lanelets. Also, the map origin is changed. These two deviations leads to the following results:

1. After converting CommonRoad obstacles to Apollo perception obstacles, the obstacles will not appear on the correct lanelets, because the the map origin is changed by LG simulator during conversion.

2. Since adjacent lanelets are not detected during the map conversion, there will be solid road boundaries between adjacent lanelets. As a result, no lane changing behavior would be performed during trajectory generation of the ego vehicle.

3. Similarly, predecessor and successor lanelets information are also missing in the converted Apollo bin files. This means, the ego vehicle would not continue driving when it reaches the end of the current lanelet.

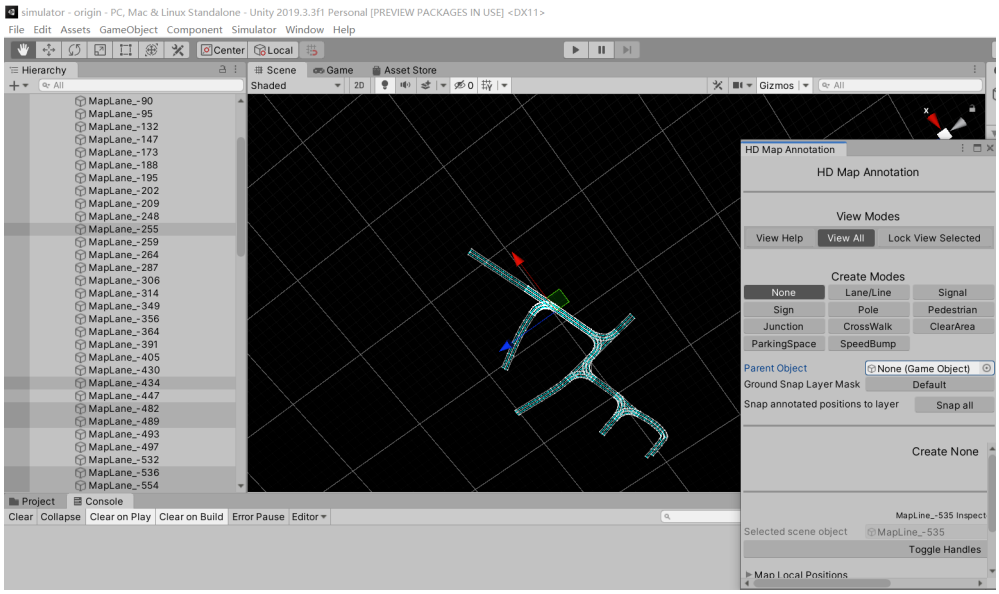

Figure 4.21: Imported map displayed in LG Simulator

d) To use LG simulator unity project to correctly convert osm lanelets to Apollo bin map, modifications need to be made in its source code. The source code for LG simulator plug-ins can be found under folder *simulator/Assets/Scripts*.

- Dealing with map origin:

LG simulator has its own way for calculating map origins. However, the position of CommonRoad obstacles cannot be converted from LG simulator unity project. To make the coordinates of maps and obstacles consistent, the code for calculating additional map origins in LG simulator needs to be disabled. The calculation of map origin is done in the file simulator/Assets/Scripts/Editor/ApolloMapTool.cs. In function "Export", an offset for the map origin is set:

"OriginEasting=mapOrigin.OriginEasting;
OriginNorthing=mapOrigin.OriginNorthing;"

So, one way to eliminate the offset for map origin is to set the both variables to 0.

"OriginEasting = 0.0; OriginNorthing = 0.0;"

However, even if the map origin offset is set to 0, the map origin itself is not taken from CommonRoad map origin. The converted map origin is chosen as the coordinates of the first node on the first input lanelet. Since modifying the calculation of the map origin in LG simulator unity project requires more effort than doing the same conversion when converting CommonRoad obstacles, this part of correction is done in the conversion of obstacles.

- Dealing with lanelet relationships:
  The LG simulator plug-ins for importing maps can be found in the folder *simulator/Assets/Scripts/Editor*. Each map format correspond to a different importer file. Since in this method, osm files are imported, the source code is written in file *Lanelet2MapImporter.cs*. From the code, it can be seen that, when importing , all lanelets are converted to instantiations of class MapLane. The class MapLane is defined in file *simulator/Assets/Scripts/Map/MapLane.cs*. In this file, attributes such as $leftLaneForward, rightLaneForward, leftLaneRevese, rightLaneReverse$ are hidden. So that these attributes would not appear in LG simulator. To monitor and debug the following modifications for detecting lanelet relationships, these attributes should be enabled in this file by commenting out the lines [*SystemNonSerialized*] before the above attributes.
  Next step is to add the process for finding out potential relationships between lanelets when importing osm maps. The function *ImportLanelet2MapCalculate* handles the generation of MapLanes and the calculation of its attributes. However, the neighboring MapLane attributes are not correctly dealt with. So in this function, additional code for detecting neighboring MapLanes are added, and dictionaries for adjacent MapLanes, pre- and post- MapLanes are created for each MapLane. These relations are then added to Apollo bin maps in file

simulator/Assets/Scripts/Editor/ApolloMapTool.cs when executing the LG simulator plug-in for exporting Apollo maps.
Figure 4.22 shows the process of detecting and classifing neighboring MapLanes. Here are some variable names used in the figure:

left_way_ids: A dictionary to store left boundary ids for every MapLane
right_way_ids: A dictionary to store right boundary ids for every MapLane

Figure 4.22 shows the process of finding directly adjacent MapLanes. By finding which MapLane uses the right boundary of the current MapLane as its left boundary, the directly right adjacent lane with the same direction can be found. Similarly, the right adjacent lane with the opposite direction, the left adjacent lane with the same / opposite direction can also be found. Except for the directly adjacent lanes, lanes with very small distances to the current lane can also be potential adjacent lanes. A node distance tolerance is set to 0.01, and an adjacent way distance tolerance is set to 0.02 for searching potential adjacent lanes. After this process, adjacent lanes are added as attributes to each MapLane. Now lane changing driving behavior can be performed on the converted Apollo maps.

e) Use Apollo tools to generate complete map information:
LG simulator unity project can export a basic Apollo map base_map.bin. Apollo uses this base map file as scenario information to perform trajectories generation and obstacles' trajectory prediction.
For each map, however, Apollo requires 3 map files for multiple uses. The first one of them is the *base_map.bin*, for planning and prediction. The second one is a *routing_map.bin*, it is a simplified version of the whole map, that is used only when generating routing responses according to routing requests. The last one is a *sim_map.bin*, it is used in Apollo map visualization in Dreamview.
To generate the latter two maps, two Apollo tools are used by running the following commands inside docker, under the root directory of apollo.
For routing_map.bin:

"dir_name=modules/map/data/demo # example map directory"
"./scripts/generate_routing_topo_graph.sh −−map_dir $dir_name"

For sim_map.bin:

"dir_name=modules/map/data/demo # example map directory"
"bazel-bin/modules/map/tools/sim_map_generator −−map_dir=$dir_name
−−output_dir=$dir_name"

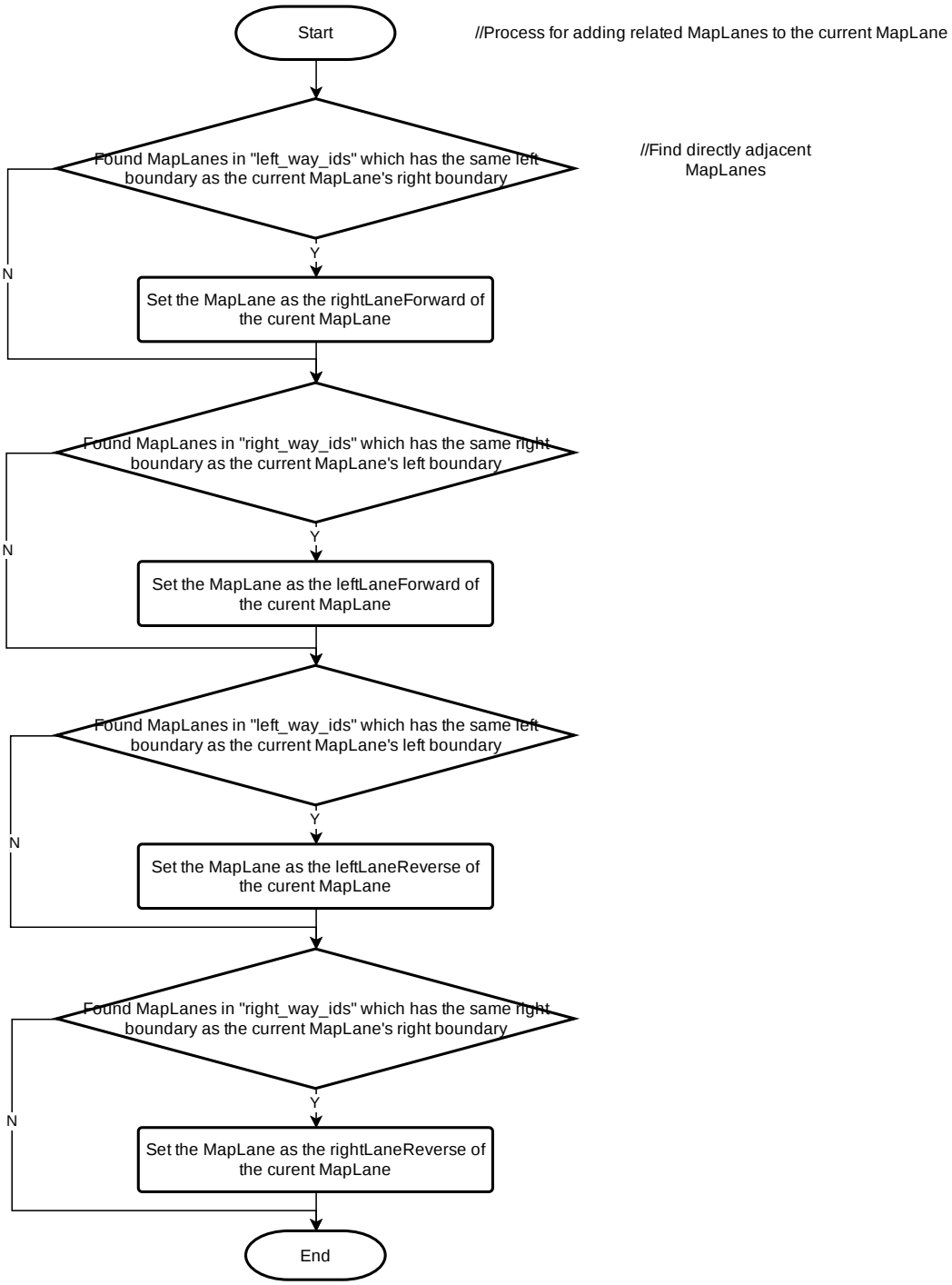

Figure 4.22: Process of creating lanelet relationships in LG

Figure 4.23 shows the result of Apollo map converted from CommonRoad xml file: $BEL\_Wervik-1\_4\_T-1.xml$.

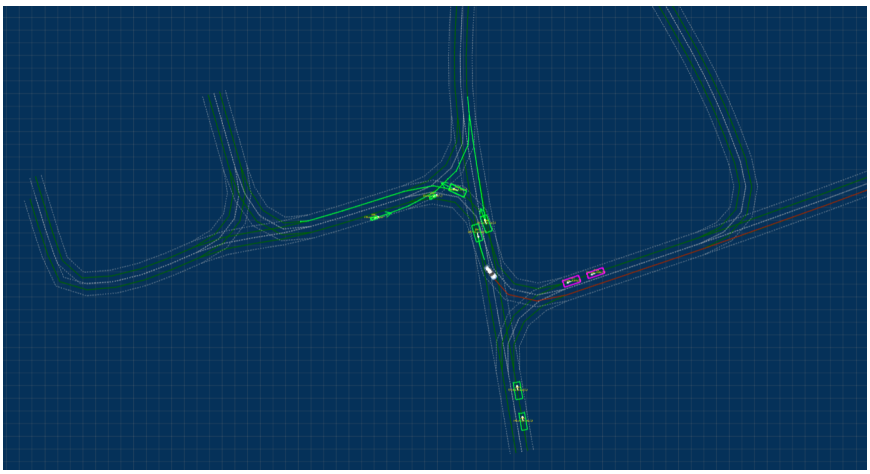

Figure 4.23: Converted Apollo map visualized in Apollo Dreamview

f) Automatically convert multiple CommonRoad xml files to Apollo map files.
The process of converting an osm file to a base map of Apollo in LG simulator unity project is manual. And the importing and exporting actions only deal with a single map conversion at a time. If multiple CommonRoad xml files need to be converted, manually importing osm files and exporting Apollo maps for every single conversion, much effort is required. So in this project, a plug-in for LG simulator is written to batch process the conversion. The plug-in file is placed here: *simulator/Assets/Tests/Map/ConvertMap.cs*. In this file, all osm files in a certain directory are traversed. And for each osm file, it is imported after the current map is cleared, a new directory with its name is created, and the imported map is exported to the newly created directory. With this LG simulator plug-in, one can conveniently execute this plug-in from the GUI of LG simulator unity project by going to *Simulator* tab, and selecting *Test*.... In the popped up *TestRunner* window, find the plug-in named *Convert* under *Simulator.Tests.dll* $\rightarrow$ *Simulator* $\rightarrow$ *Tests* $\rightarrow$ *Map* $\rightarrow$ *ConvertMap* and double click the option. Then all osm files in the certain directory are converted to Apollo maps in newly created directories with their names. Figure 4.24 shows a successful batch conversion of maps in LG simulator unity project using the ConvertMap.cs plug-in. Other automatic conversion such as converting CommonRoad xml files to osm files, and converting Apollo *base_maps* to *routing_maps* and *sim_maps* are implemented with self-written Python scripts *convert.py*, *generate.py*.

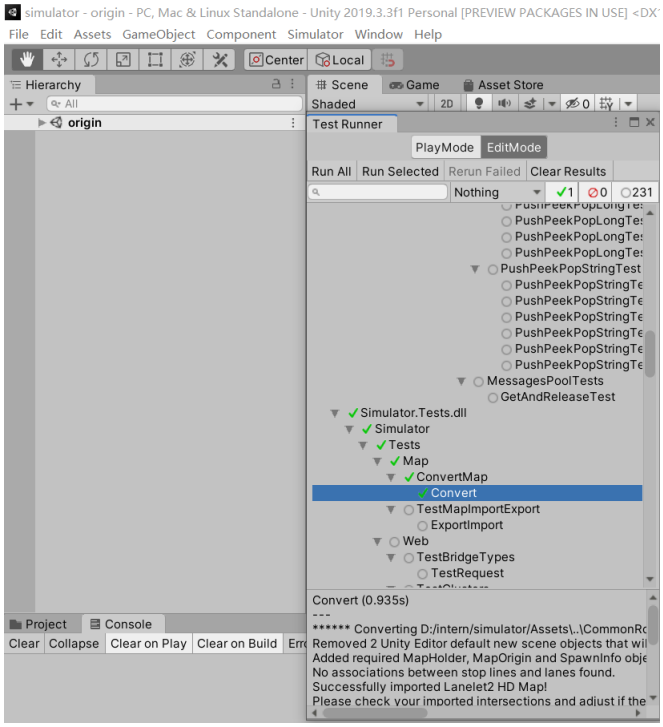

Figure 4.24: LG plug-in for automatically converting multiple CommonRoad xmls

Routing Request Conversion

Routing response is a required input for Apollo Planning module, only after receiving a routing response message in the cyber channel */apollo/routing_response*, will the Apollo Planning module be able to start calculating a long term trajectory. To obtain the Routing response message, a Routing request need to be given to the cyber channel */apollo/routing_request*, so that the Apollo Routing module, will calculate a routing response and send the information as message to the corresponding cyber channel. A *RoutingRequest* message can either be given to the cyber channel by selecting way points in Dreamview, or using Python script to set way points and generate messages. To send *RoutingRequest* messages automatically, Python script is used in this project. In the Python script *mock_routing_2p.py*, Python API for cyber is imported from outside of Apollo docker container. A cyber writer is created to write messages to cyber channel */apollo/routing_request*. The corresponding protobuf message format is *routing_pb2.RoutingRequest*. This message can be found in file *apollo/modules/routing/proto/routing.proto*. The key information in *RoutingRequest* message is a start point and a goal point of ego vehicle. In a CommonRoad scenario, the start point and the goal point of ego vehicle is the position of the initial state and goal state in the planning problem. To convert CommonRoad planning problems automatically to *RoutingRequest* messages in batches, a Python script *convert_routing.py* is written. It generates multiple *mock_routing_2p.py* files for every CommonRoad scenarios in a certain directory. By running each *mock_routing_2p.py* Python script, a pair of start and goal point, corresponding to a map, will be sent to the cyber channel */apollo/routing_request*.

Perception Obstacles Conversion

Apollo Planning module also needs predicted trajectories of percepted obstacles as input. This module listens to messages in cyber channel */apollo/prediction*, only after receiving the messages sent to this cyber channel by Apollo Prediction module, will the Apollo Planning module be able to start its planning procedure. Apollo Prediction module predicts trajectories of percepted obstacles, it listens to perception information from cyber channel */apollo/perception/obstacles*. There are several ways to generate perception obstacle messages.

a) Use LG simulator to generate point clouds of obstacles and send the information to Apollo Perception module. The Perception module then creates *PerceptionObstacles* according to the give point clouds of obstacles, and send their information as messages to cyber channel */apollo/perception/obstacles*.

b) Directly create a cyber node and send self-created *PerceptionObstacles* messages to cyber channel */apollo/perception/obstacles* by Python scripts. Perception messages are usually sent with a frequency of 10Hz. So ideally the Python scripts also send updated status of obstacles every 0.1s. The updated status of obstacles can be calculated according to json description files. In these json files, one can specify waypoints as trace for obstacles, their velocities, obstacle types etc. In this project, two different ways for calculating new status of obstacles in each time step are used.

- Manually set obstacle waypoints:
  For generation of simple test cases, it can be assumed that all dynamic obstacles move at constant speeds. Based on this assumption, one only need to provide some via waypoints from start to end position in the json file, so that the Python script can calculate an exact position for obstacles every 0.1s. These waypoints are not necessarily the positions with time interval 0.1s. Since one obstacle moves at a constant speed, exact positions at time 0.1s, 0.2, 0.3s etc. will be located by calculating on which line segment this obstacle will be, and how far will this obstacle be driving on its trace with its constant speed. Apart from the positions, the heading of each time step can also be calculated. One need only give the initial heading of each obstacle in the json file, the rest will be set as the line segment direction of each time step. To conclude, to generate simple test cases for obstacles, only one speed and heading information, and a series of waypoints need to be specified, other calculations are done in the Python scripts.

- Use obstacle information in CommonRoad scenarios:
  CommonRoad scenarios provide status information each time step for all obstacles. Typically the time interval is set to 0.1s, which is the same as the time interval of Apollo Perception update. So, if CommonRoad scenarios are used as test cases, converting CommonRoad obstacles to Apollo Perception obstacles by Python scripts is easier. The exact status (including position, speed, heading) at each time steps are already given in the CommonRoad scenarios, and they can be simply written into json description files. Next the Python script need directly use these exact information for each time step to create Apollo Perception, and send Perception messages to cyber channel /apollo/perception/obstacles with frequency 10Hz.

Running CommonRoad Scenarios in Apollo Dreamview

Steps for running CommonRoad scenarios in Apollo Dreamview:

a) Run *convert.py* to use command $osm - convert$ to convert all CommonRoad xml files in a certain directory into osm files in batch.

b) Launch LG simulator unity project and use the plug-in *ConvertMap* to convert all osm files into Apollo base_maps in batch.

c) Use *generate.py* to generate Apollo routing_maps and sim_maps based on base_maps, so that all Apollo maps are complete.

d) Put the correponding base_map, routing_map and sim_map of each Apollo map into a directory, and put the new directory under */apollo/modules/map/data/*.

e) Run *convert_obstacles.py* to convert CommonRoad Obstacles into json description files.

f) Run *convert_routing.py* to convert CommonRoad PlanningProblems into Apollo routing coordinates and generate Python scripts *mock_routing_2p.py* for sending messages to cyber channel */apollo/routing_request*.

g) Launch Apollo Dreamview and go to localhost:8888

h) Select a converted map to run it as a test case.

i) Turn on *simcontrol* and modules *Routing* and *Prediction*.

j) Run the *mock_routing_2p.py* file outside of docker that is generated from the same CommonRoad scenario as the chosen map. Then the ego vehicle will appear on the start point of the routing message.

k) Run Python script *replay_perception.py*, and give a json file as input. This json file should also be generated from the same CommonRoad scenario as the chosen map. Since the Prediction module is turned on, you can see the Apollo predicted trajectories of each obstacles on the map.

Figure 4.25 shows the visualization of the Apollo scenario convert from CommonRoad xml file $C - USA\_Lanker - 2\_1\_T - 1.xml$.

4.5.2 Using Motion Primitives for Critical Scenarios Generation

The above method creates traffic flows with multiple traffic participants. Most scenarios contain normal cases that have lower risks of danger during autonomous driving. Some scenarios contain critical cases. But the critical scenarios, converted from CommonRoad scenarios, still do not have enough variety for the testing of RSS and Online

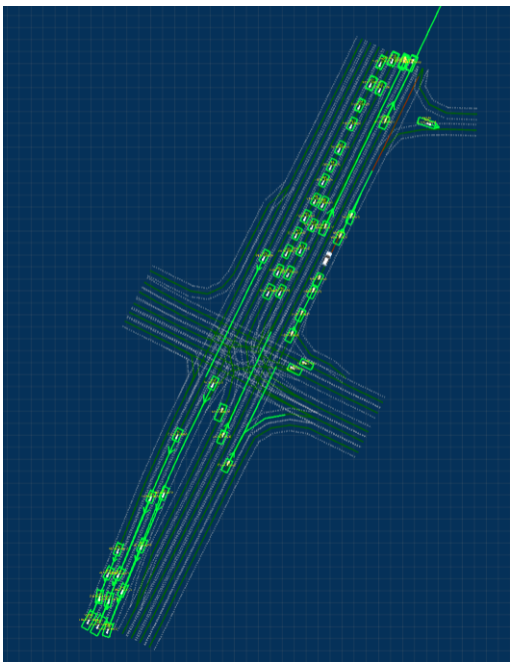

Figure 4.25: Apollo map converted automatically from LG plug-in

Verification performance in dangerous situations. To get a large enough amount of potentially critical test cases, one way is to create feasible obstacle trajectories by combining multiple motion primitives. These motion primitives can be generated with the tool CommonRoad-Search. A motion primitive is a short list of states that follow the basic rules of real vehicles' driving behaviors. Each motion primitive can have different initial velocities, initial steering angles, final velocities, final steering angles and total number of time steps. These values are sampled according to some sampling constraint parameters in the configuration file *generator_config.yaml*. Several motion primitives can be concatenated and form a trajectory. To ensure feasibility while concatenating a motion primitive to the previous one, its initial velocity and initial steering angle should have deviation less than 0.02 from the final velocity and final steering angle of the previous motion primitive in this trajectory. If the above constraints are satisfied, then this motion primitive can be translated and rotated to a certain pose such that it can be added to the end of the current trajectory. Thus, a large amount of trajectories can be generated by iterating through all pairs of motion primitives.

To use the generated trajectories as test cases, they will be first translated and rotated according to the orientation and position of a certain MapLane in an Apollo

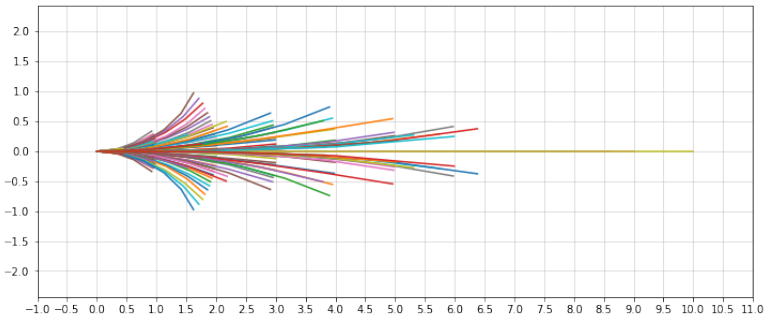

Figure 4.26: Motion primitives generated according to configuration

map. Then the resulting trajectories will be written in a json file, and be converted and sent to cyber channel as Apollo *PerceptionObstacles* with file *replay_perception.py* while testing.

4.5.3 Performance Comparison Criterion

To evaluate the safety of applying RSS and Online Verification on Apollo Planning module, the Apollo driving system is tested twice (each time with RSS or Online Verification applied) on the normal and critical cases generated by the methods mentioned in section 4.5.1 and section 4.5.2. An intuitive way to compare RSS and Online Verification performance in these tests is to see if collisions occur during the driving process in normal and critical scenarios. However, collision may seldom occur even if the safety insurance method is not good enough or the situation is already a potentially dangerous one. To make it still possible in these situations to compare the performance between two safety methods, another danger criterion needs to be defined. One simple way is to calculate the safe distance between the preceding obstacle and the following ego vehicle. The safe distance can be calculated according to the following formula [5]:

$$\xi_{safe} = \frac{1}{2a_{max}} \left(v_{f,0}^2 - v_{p,0}^2 \right) + v_{f,0}\delta \tag{4.1}$$

where $v_{f,0}$ is the current velocity of the following ego vehicle, $v_{p,0}$ is the current velocity of the preceding obstacle, δ is the reaction time. In this paper, the reaction time is taken as 0.3s. This equation for safe distance shows the shortest distance that needs to be kept between the center of the two vehicles on the same lane, so that if the preceding vehicle suddenly brakes at its full acceleration, the following ego vehicle is able to react after a short time and brakes also with the maximal acceleration to avoid a collision.

To calculate safe distances during the Apollo driving process and monitor the violation of this criterion, a Python script *safe_distance.py* is written to provide safe distances every 0.1s with the current obstacle and ego states. Obstacle velocities and positions can be received by listening to the cyber channel */apollo/perception/obstacles* and reading *PerceptionObstacle* protobuf messages. And parameters such as the current accelerations, velocities, positions and headings of the ego vehicle can be taken from the protobuf messages *LocalizationEstimate* given by the cyber channel */apollo/localization/pose*. Since *PerceptionObstacle* is updated with a frequency of 10Hz, and *LocalizationEstimate* with the frequency 100Hz, the valid safe distance is give whenever obstacle states are updated, that is every 0.1s. After preparing all information of obstacles that may be needed, those that are not related to the safe distance will be ignored. Ignored obstacles are those obstacles that are not in front of the ego vehicle, and obstacles that have large lateral distances to the ego vehicle even if they are in front of it. Next, each of the remaining obstacles will be used to calculate the current distance and the safe distance with the ego vehicle. Every safe distance is checked if it is violated, and if yes, this violation is logged.

This safe distance can be used to decide whether an obstacle on the adjacent lane of the ego vehicle is allowed to merge into the current lane (corresponding to the calculation of M3 in SPOT), and it can be also used to judge the safety of the ego vehicle's trajectory when two vehicles are already on the same lane. If in the latter case, the real distance between the following ego vehicle and the preceding obstacle violates the safe distance, then it can be considered as a dangerous situation. At the same time, the obstacle trajectories generated from the motion primitives should not have a merge action if the current distance is smaller than the safe distance.

So, now consider the following scenario, where an obstacle is driving with a relatively high speed on the adjacent lane from longitudinally behind the ego vehicle, and after it drives longitudinally in front of the ego vehicle, it merges into the ego lane without violating the safe distance. Then it suddenly brakes at its largest braking acceleration on the ego lane. Now, if the ego vehicle's trajectory fails to avoid a collision, or if it violates the safe distance, then a dangerous situation is found, where the ego vehicle is responsible.

The following experiments is done by trying to generate such scenarios and evaluations of RSS and Online Verification performance are done with the help of safe distances.

5 Experiments

5.1 Requirements

5.1.1 Versions of related Tools

- commonroad-io == 2020.3
- commonroad-drivability-checker == 2020.1
- commonroad-curvilinear-coordinate-system == 2019.1
- commonroad-vehicle-models == 1.0.0
- commonroad-search (branch: master)
- spot-cpp (tag: initial_submission_TIV2020)
- reactive-planner (branch: feature_jiaying)
- opendrive2lanelet == 1.2.0
- apollo6.0
- lgsvl 2020.06 source code

5.1.2 Tree of Added File Structure

```
root
├── apollo
│   ├── cyber
│   │   └── python
│   │       └── cyber_py3
│   │           └── examples
│   │               ├── fake_all.json
│   │               ├── fake_perception_all.bash
│   │               ├── interface.py
│   │               ├── mock_routing_2p.py
│   │               └── replay_perception_all.py
│   └── modules
│       ├── planning
│       │   ├── planning_component.cc
│       │   ├── planning_component.h
│       │   └── tasks
│       │       └── deciders
│       │           └── rss_decider
│       └── map
│           └── data
│               └── belwervik
│                   ├── base_map.bin
│                   ├── sim_map.bin
│                   └── routing_map.bin
├── commonroad
│   └── tools
│       ├── commonroad-curvilinear-coordinate-system
│       ├── commonroad-drivability-checker
│       ├── commonroad-search
│       │   └── tutorials
│       │       └── 3_motion_primitives_generator
│       │           └── safe_distance.py
│       ├── reactive-planner
│       ├── spot-cpp
│       ├── select_scenarios.py
│       └── convert_obstacles.py
└── simulator
    └── Assets
        └── Tests
            └── Map
                └── ConvertMap.cs
```

5.2 Parameter Setups

1. Time duration between two time steps of a converted CommonRoad Scenarios $scenario.dt = 0.2$:
   The computation time of converting Apollo *ADCTrajectories* into CommonRoad long-term trajectories is determined by $scenario.dt$. With smaller $scenario.dt$, more CommonRoad states need to be calculated and stored, which also leads to more time consumption for SPOT calculation, collision check and fail-safe trajectory generation. For this reason, a relatively large $dt = 0.2s$ is chosen.

2. Length of long-term trajectories converted from Apollo *ADCTrajectories* $long\_timestep = 20$:
   Another factor that determines the computation time of trajectory conversion, and especially SPOT calculation and collision check, is the total time steps taken from the Apollo *ADCTrajectories*. An entire Apollo *ADCTrajectory* usually contains state information in the next 8 seconds, with changing time durations between states: 0.1s if velocity is relatively constant, otherwise 0.02s. If all 8 seconds of the *ADCTrajectory* states are converted into long-term trajectory states, Then the time consumption would be large. In the following tests, the length of long-term trajectories are set to be 20, so it covers the first $20 \times 0.2s = 4s$ of the Apollo *ADCTrajectory*. The time duration of long-term trajectory should be longer than the time interval between two Planning messages, so that the next calculated fail-safe trajectory can be sent out as Planning message before the previous trajectory is executed to the last state.

3. Time duration of fail-safe trajectories $t\_fs = 2.0$:
   The total time duration of a fail-safe trajectory is also important for reducing calculation time of SPOT, since time variant obstacles will be generated for all dynamic obstacles in the next $long\_timestep \times scenario.dt + t\_fs$ seconds from SPOT to ensure also the safety of the fail-safe trajectories. So the length of fail-safe trajectory is set to be relatively small number 2.0. It should also not be too small, otherwise, no braking maneuver could be sampled to make ego vehicle brake in the time duration. With $t\_fs = 2.0$, the resulting safe trajectories will last for:

   $$ttr \times scenario.dt + t\_fs = [0 \times 0.2s + 2.0s, 20 \times 0.2s + 2.0s] = [2.0s, 6.0s]$$

   where ttr means time-to-react, and refers to the first time step that has no collision-free braking maneuver, if the ego vehicle follows the route of the long-term trajectory. And the sampled fail-safe trajectory will be concatenated to the safe part of the long-term trajectory.

4. Sampling level for generating fail-safe trajectories
   $sampling\_level = 3$:
   After the set-based prediction of all dynamic obstacles are calculated, the time-to-react is checked by generating a braking maneuver along the long-term trajectory. So the generated braking maneuver is already a solution for fail-safe trajectory. To find the best fail-safe trajectory, CommonRoad reactive planner is used to sample solutions and select the one with the lowest cost. The sampling level of CommonRoad reactive planner determines for how many levels the planner will resample if in the last level fails to find a solution. If the sampling level is reached and solution is still not found, then the reactive planner stops sampling and return with no solution. In this case, the previous generated braking maneuver will be used as fail-safe trajectory. In the following experiments, CommonRoad reactive planner will sample for 3 levels, before the braking maneuver is used, because each additional sampling level costs much longer computation time.

5. Configuration for motion primitives generation:
   To generate obstacle trajectories with merging and braking behavior, the following configuration is used:

   vehicle_type_id: 2

   duration: 2

   dt_simulation: 0.05

   velocity_sample_min: 0.0

   velocity_sample_max: 30.0

   num_sample_velocity: 10

   steering_angle_sample_min: 0

   steering_angle_sample_max: 0.0

   num_sample_steering_angle: 3

   num_segments_trajectory: 6

   num_simulations: 50

   In addition to the above configuration, extra rules for generating sample trajectories are applied. The first $\frac{2}{3}$ part of the trajectory, that are the first 4 motion primitive segments out of 6, must have initial steering angles as 0. In this way, it is more likely that the resulting trajectory can make the obstacle on the adjacent

lane accelerate and drive far enough to safely merge into ego lane beyond the safe distance.

6. Parameters for safe distance calculation:

   amax = 8.0

   delay = 0.3

   front obstacle angle: 90deg

   lateral distance between vehicles: 1m

   The maximum acceleration of ego vehicle is set to $0.8 m/s^2$, which is the same as the acceleration used in SPOT and reactive planner. The reaction time delay is $0.3s$, which also corresponds to the one used in SPOT.
   For detecting front obstacles in safe distance calculation, a front obstacle angle and a lateral distance between vehicles are defined to ignore unrelated obstacles. The front obstacle angle is set to be 90 degree, it means if the center of the obstacle is longitudinally behind the ego vehicle, then it will be considered irrelevant during the following calculation of safe distances. Among the rest of the obstacles, only those that are laterally close to the ego vehicle are taken into account. The lateral distance between vehicles is set to 1m, it means if the boundaries of the two sides of obstacle and ego vehicle have lateral distance less than 1 meter, then they are considered relevant. Then the longitudinal distance between each relevant obstacles and the ego vehicle is evaluated by the calculated safe distance, and if the safe distances are violated, this situation will be recorded.

5.3 Test Scenarios

5.3.1 Normal Apollo scenario: Apollo Sunnyvale Loop Map + Fake perception obstacles

This is a normal scenario using the Apollo default map *SunnyvaleLoop*. This map has adjacent lanes with same direction, and forking lanes, which helps offer various scenarios, if different obstacle trajectories are given. In this scenario, a set of fake perception obstacles are created, including four dynamic obstacles and one static obstacle. All dynamic obstacles are descibed in a json file, and published as *PerceptionObstacle* messages by *replay_perception_all.py*. The route of every dynamic obstacle is given directly by some via points, and it is assumed that each dynamic obstacle drives at a constant speed. The traffic contains vehicles from both the same and the opposite

direction, and also including turning behavior. In this scenario, no emergency braking or merging is performed, so it can be considered as a normal case.

5.3.2 Critical CommonRoad Scenario: Critical Zero-start CommonRoad Scenario + CommonRoad Obstacles

CommonRoad Scenarios needs to be converted into Apollo scenarios, and then Apollo planner generates *ADCTrajectories* according to the surrounding obstacles. Apollo planner can only plan trajectories starting from zero initial velocities. However, the initial velocity in the Planning Problem of most CommonRoad Scenarios are not zero. To make use of CommonRoad Scenarios for normal case testing, scenarios that have zero initial velocities in Planning Problems. The rest of the CommonRoad obstacles will be converted into json description files, and sent as *PerceptionObstacles* messages when performing tests. Critical CommonRoad Scenarios can be obtained by selecting CommonRoad Scenarios with tag *critical*. In the experiments, the CommonRoad scenario $BEL\_Wervik-1\_4\_T-1.xml$ is used.

5.3.3 Critical Apollo scenario: Apollo Sunnyvale Loop Map + Obstacles with motion primitives

To create a large amount critical cases that have fewer obstacles with non-constant velocities, motion primitives are used to generate obstacle trajectories with merging and braking behavior. With the same Apollo default map: Sunnyvale Loop, a large amount of fake obstacles can be applied on it and used for performance testing, without have to switch between maps frequently. This method largely increases the testing efficiency.

5.4 Test Results

5.4.1 Test results of normal Apollo scenario: Apollo Sunnyvale Loop Map + Fake perception obstacles

This scenario is a normal situation, where the ego vehicle successfully avoid collision with the obstacles when either RSS or Online Verification is applied. In this scenario, four fake dynamic obstacles and one static obstacle are created with a *json* files. Three of the dynamic obstacles have the same initial driving direction as the ego vehicle, one of them is driving with an opposite direction. The scenario with RSS applied is shown in Figure 5.1. The transparent black rectangles are the occupancy sets of the ego vehicle according to its 4s long-term trajectory. The solid red rectangle with black margin is the static obstacle. The transparent blue rectangles are the set-based prediction of the

obstacles, which covers the same time interval as the long-term trajectory of the ego vehicle.

When Online Verification is applied, the total trajectory contains also the 2s fail-safe trajectory. The scenario is shown in Figure 5.2. The transparent black rectangles are the occupancy sets of the ego vehicle according to its legally safe part of long-term trajectory. The transparent red rectangles show the occupancy sets during the 2s fail-safe trajectory. The solid red rectangle with black margin is the static obstacle. The transparent blue rectangles are the set-based prediction of the obstacle.

Figure 5.3 shows the velocities and accelerations of the ego vehicle during the test with RSS and Online Verification.

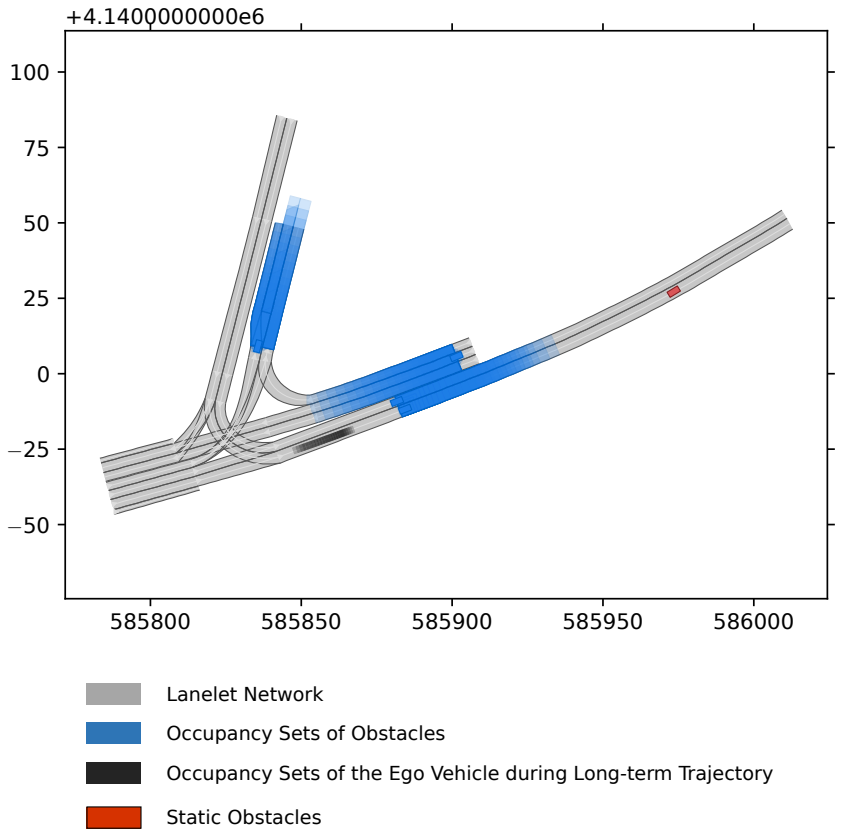

Figure 5.1: Normal Apollo scenario when using RSS

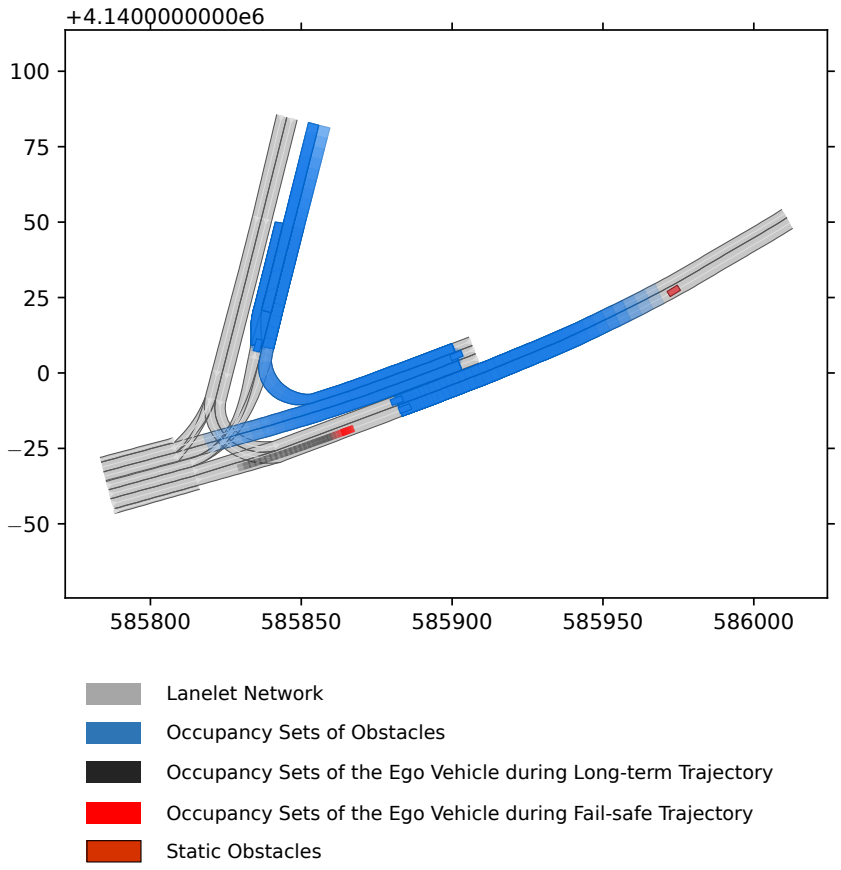

Figure 5.2: Normal Apollo scenario when using Online Verification

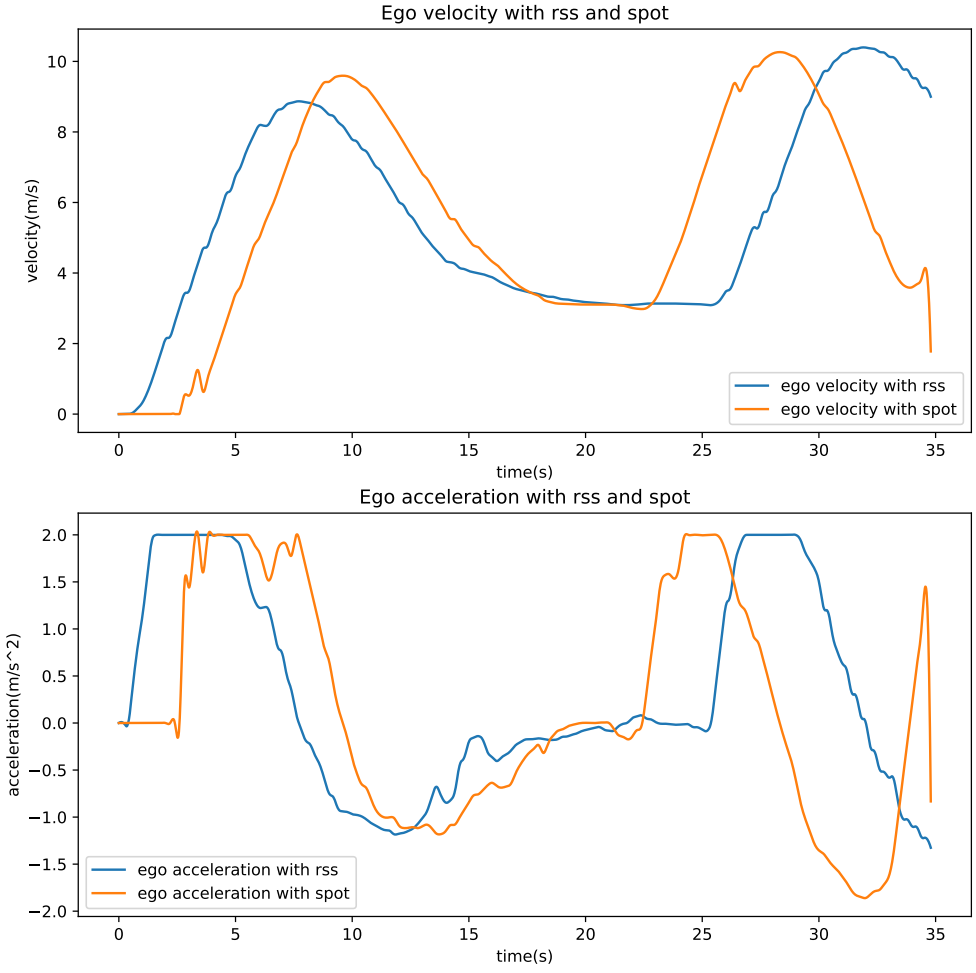

Figure 5.3: Velocity-time graph and acceleration-time graph of ego vehicle in normal Apollo scenario

5.4.2 Test results of critical CommonRoad scenario: Critical Zero-start CommonRoad Scenario + CommonRoad Obstacles

This scenario $BEL\_Wervik-1\_4\_T-1.xml$ is a critical zero-start CommonRoad Scenario. Both RSS and Online Verification failed to avoid the "collision" by generating valid ego vehicle trajectories. Because in the Apollo real-time simulation system, the best time for ego vehicle to avoid the collision is missed due to long computation time. Finally, in both cases, the ego vehicle failed to make a move, and the bicycle on the opposite lane crashed on the ego vehicle.

In the case of RSS, the planner first give a small ego trajectory but in the next planning cycle, it detects the danger, then brakes and remains stop until the end. In the case of Online Verification, the occupancy sets of obstacles takes the driving area of ego vehicle, and the ego vehicle remains unmoved until the end. In both cases, the "collision" is not the responsibility of the ego vehicle, the bicycle should have stopped or started avoiding before the "collision" occurs, but the scenario is not interactive.

The scenario with RSS is shown in Figure 5.4, and the scenario with Online Verification is shown in Figure 5.5. The changing of velocity and acceleration of the ego vehicle during the whole process is shown in Figure 5.6.

5.4.3 Test results of critical Apollo scenario: Apollo Sunnyvale Loop Map + Obstacles with motion primitives

The following scenario is a critical situation, where the ego vehicle failed to avoid collision with the obstacle when RSS is used, and it successfully stopped without violating the safe distance when Online Verification is applied. The obstacle started with a high speed from behind the ego vehicle on the adjacent lane. Then it merged into the ego lane without violating the safe distance. After reaching the ego lane, it braked with a large deceleration. During the first period of the obstacle's braking, Apollo prediction module had always been predicting, that the obstacle will drive back to the adjacent lane. Thus, the ego vehicle did not take any actions such as braking or avoiding, and still accelerated towards the braking obstacle. Only after the ego vehicle violated the safe distance, did it try to brake. However, it was too late for the ego vehicle to avoid a collision. The collision scenario is shown in Figure 5.7. The transparent black rectangles are the occupancy sets of the ego vehicle according to its 4s long-term trajectory. The transparent blue rectangles are the set-based prediction of the obstacle, which covers the same time interval as the long-term trajectory of the ego vehicle. As can be seen from Figure 5.7, the ego-vehicle collides with the occupancy

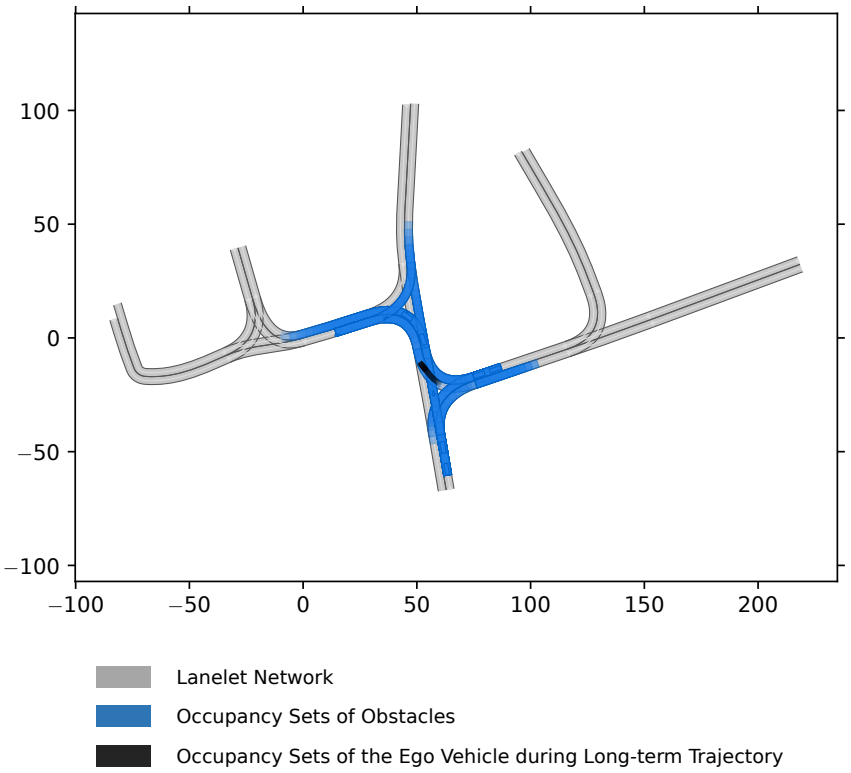

Figure 5.4: Critical CommonRoad scenario when using RSS

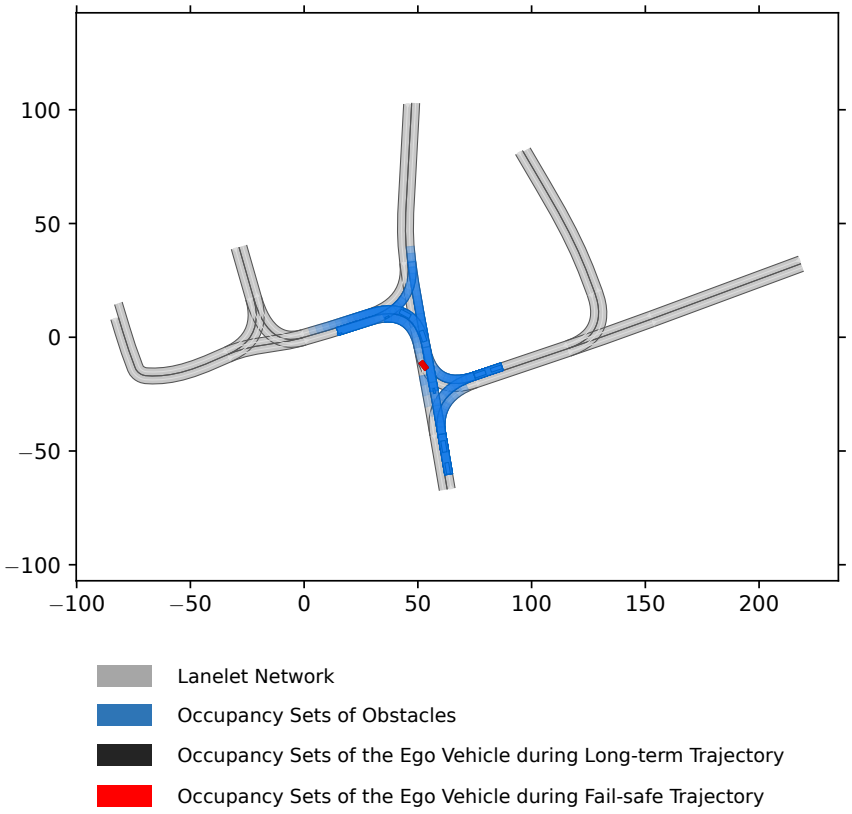

Figure 5.5: Critical CommonRoad when using Online Verification

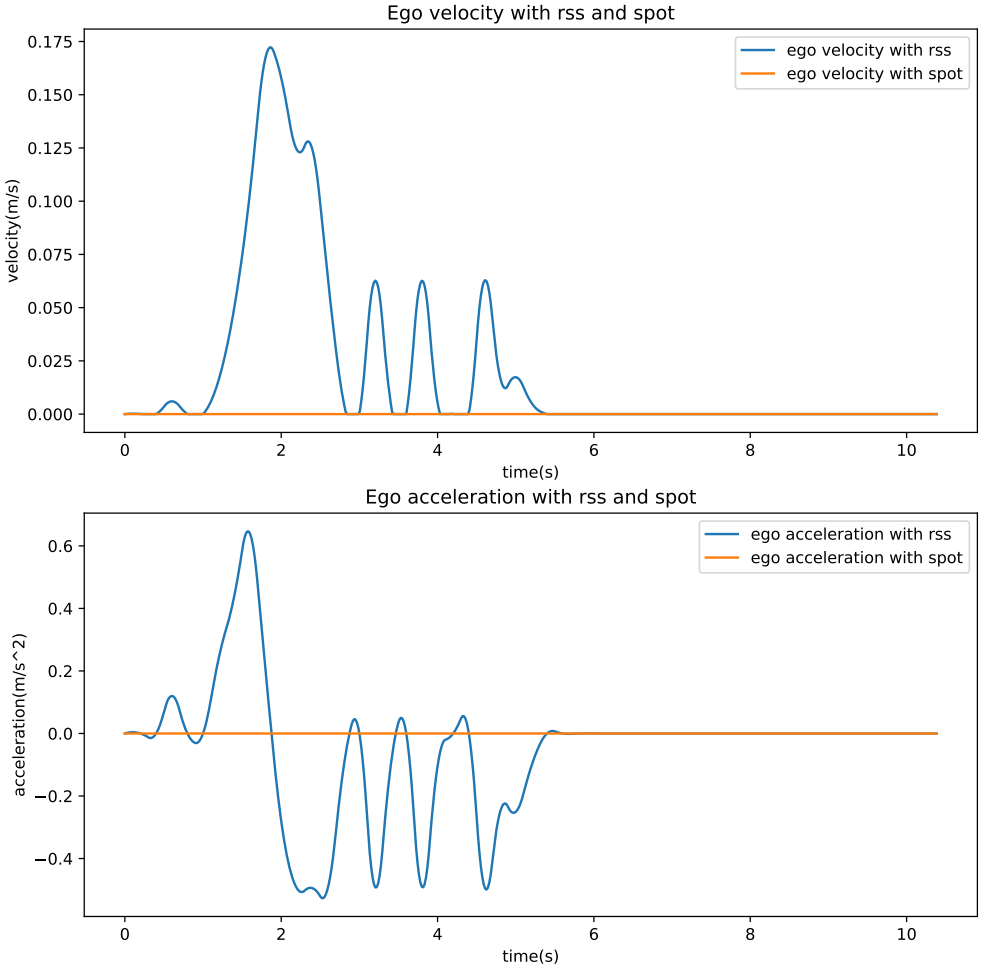

Figure 5.6: Velocity-time graph and acceleration-time graph of ego vehicle in critical CommonRoad scenario

sets of the obstacle if it continues to drive according to its long-term trajectory.

With Online Verification applied, the legal safety of the long-term trajectory is verified, the time-to-react is calculated and the fail-safe trajectory that lasts for 2s is concatenated to the end of the safe part of the long-term trajectory. The ego vehicle was able to start braking earlier because the set-based prediction covers all legal position of the obstacle, and the ego vehicle can foresee the braking behavior of the obstacle, thus it successfully stopped before hitting the front obstacle. The successful scenario is shown in Figure 5.8. During the whole process, the safe distance was never violated. The transparent black rectangles are the occupancy sets of the ego vehicle according to its legally safe part of long-term trajectory. The transparent red rectangles show the occupancy sets during the 2s fail-safe trajectory. The transparent blue rectangles are the set-based prediction of the obstacle. In this scenario, the safe part of long-term trajectory lasts for 2.8s, and the time-to-react is 15.

The following Figure 5.9 shows the velocities and accelerations of the ego vehicle during the test with RSS and Online Verification. The maximal acceleration reached in RSS case is $-5.6m/s^2$, in Online Verification case is $-7.8m/s^2$.

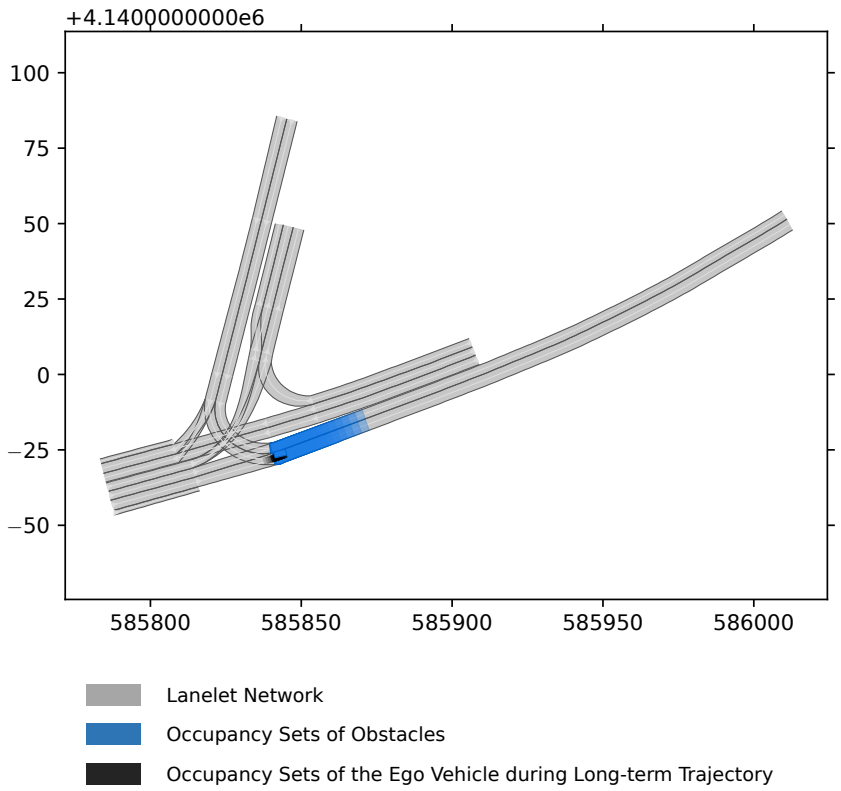

Figure 5.7: The collision scenario when using RSS

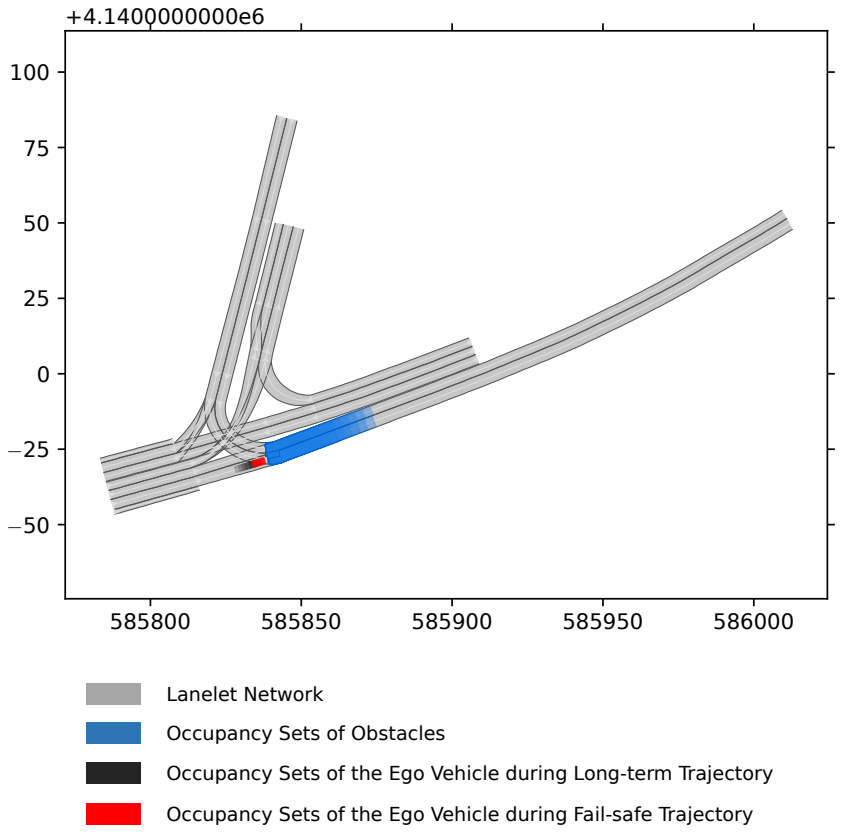

Figure 5.8: The successful scenario when using Online Verification

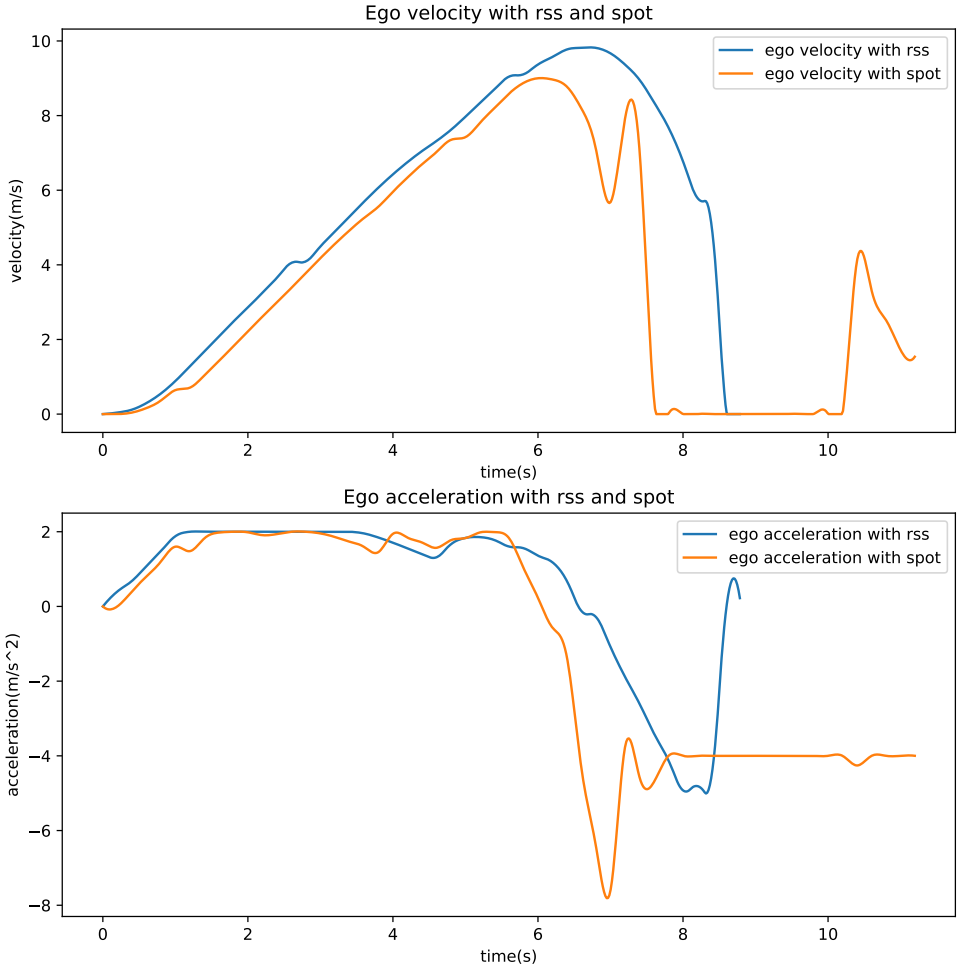

Figure 5.9: Velocity-time graph and acceleration-time graph of ego vehicle in critical Apollo scenario

6 Conclusion

The whole execution process of each test scenarios in Section 5.3 corresponds to multiple scenarios with lanelet network, obstacles and a planning problem. Except for the test scenarios that are listed, more than 100 test scenarios that are generated based on motion primitives as mentioned in Subsection 4.5.2 are tested with either RSS and Online Verification applied. In most cases, both safety methods successfully avoid collision during the whole driving process, as shown in Figure 5.1 and Figure 5.2. When Online Verification is applied in these normal cases, the time-to-react is usually the last timesteps of the long-term trajectories, and the fail-safe trajectories are never executed. In some critical cases, due to the non-interactive actions of the obstacles, some collisions are unavoidable even if the ego vehicle always receives standstill trajectories. But in these critical cases, it is still obvious that RSS leads to a more aggressive trajectory than Online Verification as shown in the velocity and acceleration graphs in Figure 5.6. The occupancy sets of obstacles cover every possible positions of obstacles, and if obstacles have large velocities and the ego vehicle starts from 0 initial velocity, the set-based prediction of obstacles cover almost all the drivable area of ego vehicle. With Online Verification applied, the ego vehicle tends to wait until the critical case is gone before it starts to drive. In some rare cases, where RSS couldn't recognize the suddenly cut-in front obstacle between planning cycles, a collision may occur, see Figure 5.7. In these cases, Online Verification can still sense the danger and decrease the velocity of the ego vehicle in advance, see Figure 5.8. In all test scenarios during the experiments, Online Verification always lead to safe driving behavior of the ego vehicle, except for the cases where obstacles drive illegally. RSS performs some more aggressive driving bahavior, but collide with obstacles in this specific scenario. Although the collision cases seldom occur, it still means a huge amount of accidents if the autonomous driving is widely used. Each of the accidents causes also great loss. To conclude, the application of Online Verification does improve driving safety in comparison with RSS, and this improvement has great worth.

In the future, the computation effort of each module in CommonRoad can be decreased. Although the current planning cycle lasts for 0.5s, which is already shorter than the output trajectories from 2s-6s, safety of perception module still have impact on the safety of autonomous driving. Perception errors may be corrected before starting

the next planning cycle, with faster planning cycle, the errors may have smaller negative impact on the safety of output trajectories.

List of Figures

List of Tables

Bibliography

[1] D. González, J. Pérez, V. Milanés, and F. Nashashibi. "A Review of Motion Planning Techniques for Automated Vehicles." In: *IEEE Transactions on Intelligent Transportation Systems* 17.4 (2016), pp. 1135–1145.

[2] C. Pek, S. Manzinger, M. Koschi, and M. Althoff. "Using online verification to prevent autonomous vehicles from causing accidents." In: *Nature Machine Intelligence* 2.9 (2020), 518–528.

[3] Mobileye. *Implementing the RSS Model on NHTSA Pre-Crash Scenarios*.

[4] S. Shalev-Shwartz, S. Shammah, and A. Shashua. *On a Formal Model of Safe and Scalable Self-driving Cars*. 2017.

[5] M. Koschi and M. Althoff. "SPOT: A tool for set-based prediction of traffic participants." In: *Intelligent Vehicles Symposium*. 2017.

[6] M. Werling, J. Ziegler, S. Kammel, and S. Thrun. "Optimal Trajectory Generation for Dynamic Street Scenarios in a Frenet Frame." In: *Robotics and Automation (ICRA), 2010 IEEE International Conference on*. 2010.

[7] L. Z. and H. J. *CommonRoad-Apollo Integration*. 2020.