COUPLING APOLLO WITH THE COMMONROAD MOTION PLANNING FRAMEWORK

Xiao Wang[*1], **Anna-Katharina Rettinger**[*1], **Md Tawhid Bin Waez**[2], **and Matthias Althoff**[1]

[*] *Equal contribution*
[1] *Technical University of Munich, Department of Informatics*
85748 Garching, Germany (E-mail: xiao.wang@tum.de, anna-katharina.rettinger@tum.de)
[2] *Ford Motor Company, MI, 48124*

ABSTRACT: Open-source software platforms are becoming more important for researchers in the field of autonomous driving. One of the leading platforms for autonomous driving is Apollo, whose purpose is accelerating the testing and deployment of autonomous vehicles. However, Apollo's complex software structure hampers an easy integration of software modules, especially for motion planning. Moreover, the standalone version of Apollo requires users to upload their algorithms to a cloud platform to allow testing in diverse scenarios, which is unacceptable for many companies. In contrast, the open-source CommonRoad benchmark suite contains diverse testing scenarios in different settings, such as highways, urban environments, dense traffic, and settings where interaction with bicyclists and pedestrians is particularly important. In addition, CommonRoad provides a motion planning framework in Python, which enables rapid prototyping of motion planners, along with additional tools, such as an efficient drivability checker, a map format converter, and interfaces to the traffic simulators SUMO and CARLA. In this work, we introduce a Python API between the planning module of the Apollo platform and the CommonRoad software framework, which bridges the gap between rapid prototyping for planning algorithms and their validation through real test drives. We demonstrate our interface with two scenarios.

KEYWORDS: Autonomous driving, motion planning, open-source software framework, CommonRoad, and Apollo

1. Introduction

Due to the complexity and the wide variety of possible traffic situations, autonomous driving requires extensive virtual testing as well as real test drives. While virtual testing of motion planners only requires simulators (e.g., the open-source traffic simulator SUMO [1]) or recordings of vehicle movements (e.g., from the highD dataset [2]), real test drives obviously require the entire software stack from perception to action.

To bridge the gap between rapid prototyping of planning algorithms and real test drives, we coupled the Apollo open-source software stack[1] with our open-source software framework CommonRoad [3] for benchmarking and motion planning of automated vehicles. Our interface allows one to test arbitrary motion planners on scenarios from the CommonRoad benchmark suite as well as on a real vehicle using Apollo without the need for any modifications. Additionally, our interface can be used for safety verification frameworks as introduced in [4, 5, 6]. Furthermore, our interface can record test results in the CommonRoad format from Apollo for offline analyses.

1.1. Related Work

Early software stacks for autonomous driving, such as the autonomous Bertha Benz Memorial Drive [7], are not publicly available. However, several open-source software stacks other than Apollo have been released in the past years (e.g., Autoware[2], Nvidia Drive-Works[3], and openpilot[4]). In particular, Kessler

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

[2] https://github.com/Autoware-AI/autoware.ai
[3] https://developer.nvidia.com/drive/drive-software
[4] https://github.com/commaai/openPilot

et al. [8] presented the required steps for setting up the Apollo software stack in a research vehicle. We selected the Baidu Apollo software stack to test our safety verification framework in order to align with Ford's collaboration with Baidu[5].

Testing a motion planner in a simulation is crucial before running it on a real vehicle. Therefore, some open-source software stacks can be connected to a simulator, like Apollo, which can be used with the LGSVL simulator[6] or the CARLA simulator[7] [9]. However, LGSVL and CARLA focus on all aspects of autonomous driving, including perception, which unnecessarily increases the complexity of testing solely the motion planning module. Instead, lightweight simulators such as SUMO [1], which can be coupled with CommonRoad [10], can be used for traffic simulation. By combining the interface of this work with [10], one can import generated traffic from SUMO into Apollo. In addition, one can utilize dataset converters in CommonRoad to load recordings from the highD dataset [2], the Next Generation Simulation (NGSIM) program [11], and the INTERACTION dataset [12] to perform extensive virtual testing. Moreover, the CommonRoad framework provides an efficient drivability checker [13], which consists of a collision checker, a road-compliance checker, and a feasibility checker.

1.2. Challenges

The Apollo platform has been continuously updated since its initial launch. In this work, we provide an interface with the most up-to-date version Apollo 5.0. In addition, Apollo 3.0 is the latest version based on the Robot Operating System (ROS)[8], which is the most commonly used robotics middleware for robot software development. For this reason, we present the coupling with Apollo 3.0 as well. Note that we test our safety verification modules on Apollo 3.0, whereas the interface to Apollo 5.0 allows one to test an arbitrary motion planner.

To integrate CommonRoad modules in Apollo 3.0 and 5.0, we have overcome the following major challenges:

- Communication between an independent component from outside docker and Apollo modules is quite cumbersome when using Apollo 3.0[9].
- Apollo 3.0 does not support Python 3, which is required by CommonRoad.
- Apollo 3.0 runs on Ubuntu 14, which provides outdated library dependencies and compilers.
- There is a lack of clear documentation in Apollo for integrating one's own modules.
- Although Apollo 3.0 is ROS-based, it significantly deviates from ROS so that ROS software modules are not directly interchangeable.

To tackle these challenges, we have developed an interface for Apollo 3.0 that is directly integrated into Apollo modules, supports Python 3, and runs on Ubuntu 14. For the Apollo 5.0 integration, we only have to convert the attributes of the messages. We show our approach in detail in Sec. 3.

1.3. Contributions

This paper presents an easy-to-use interface bridging the CommonRoad motion planning framework with the Apollo open-source driving stack. Specifically, the Apollo-CommonRoad interface[10]

- can be used with arbitrary motion planners and supports fail-safe planning (see Sec. 2.3);
- provides a well-documented Python API;
- makes a unified interface for several motion planning tools available, such as an efficient drivability checker and a map converter;
- realizes offline generation of traffic from the open-source traffic simulator SUMO to Apollo;
- enables recording test scenarios in the CommonRoad format for offline analyses.

The remainder of this paper is organized as follows. In Sec. 2, we present the motion planning framework in CommonRoad and the associated safety verification framework. Sec. 3 introduces our interface between the Apollo driving stack and CommonRoad. In Sec. 4, we demonstrate our approach in two scenarios. The paper ends with conclusions in Sec. 5.

2. Preliminaries

We start by introducing Apollo, followed by the motion planning framework and the safety verification framework in CommonRoad.

2.1. Apollo Platform

The structure of Apollo 3.0 is shown in Fig. 1 and consists of the following main modules:

- **Map engine**, which provides structured information regarding the road network;
- **Localization**, which leverages various information sources, such as GPS, LiDAR, and IMU to estimate the current location of the autonomous vehicle;

[5] `https://media.ford.com`
[6] `https://www.lgsvlsimulator.com`
[7] `https://github.com/AuroAi/carla_apollo_bridge`
[8] `https://www.ros.org`
[9] `https://github.com/ApolloAuto/apollo/issues/4892`
[10] available at `commonroad.in.tum.de`

Figure 1. Apollo 3.0 Open Software Platform (Source: `https://github.com/ApolloAuto/apollo`)

- **Perception**, which identifies the world surrounding the autonomous vehicle;
- **Planning**, which plans the spatio-temporal trajectory for the autonomous vehicle to execute;
- **Control**, which executes the planned spatio-temporal trajectory by generating control commands such as throttle, brake, and steering;
- **End-to-end**, which combines perception, planning, and control using machine learning as an alternative solution;
- **HMI**, which uses Dreamview as a human machine interface for displaying the status of the vehicle, testing other modules, and controlling the vehicle functions in real-time.

Each module runs as a separate CarOS-based ROS node as well as publishes and subscribes certain topics.

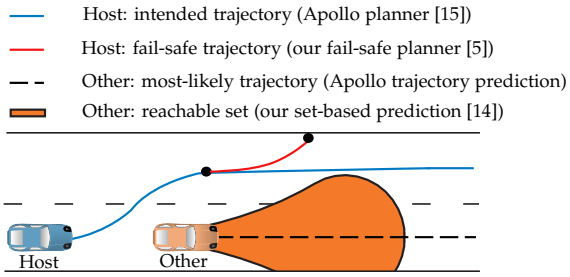
2.2. CommonRoad

The CommonRoad API provides methods for handling CommonRoad [3] scenarios, which contain the basic information for solving a motion planning problem. A CommonRoad scenario consists of a road network, the movement of other traffic participants, the initial state, and a goal region. The CommonRoad API provides various motion planning tools, including

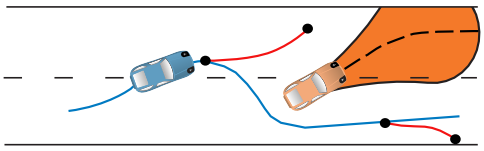
- **a drivability checker** [13], which consists of a collision checker, a road-compliance checker, and a feasibility checker;
- **a map converter** [3], which converts the OpenDRIVE and the OpenStreetMap format to the lanelet format;
- **a set-based prediction tool** [14], which predicts all possible future states of other traffic participants;
- **CommonRoad-SUMO interface** [10], which couples CommonRoad with the macroscopic traffic simulator SUMO.

2.3. Safety Verification Framework

We demonstrate our Apollo-CommonRoad interface for safety verification as presented in [4]. The basic idea is shown in Fig. 2: we computed a collision-free fail-safe trajectory with respect to all possible legal behaviors of surrounding vehicles at all times (Fig. 2a). These behaviors are contained in reachable sets, obtained by set-based prediction [14]. While the ego vehicle follows its intended trajectory, a new fail-safe trajectory is computed to ensure safety at any time (Fig. 2b). In the case where other traffic participants deviate from their most-likely trajectory and no new valid fail-safe trajectory is found, the previously-computed fail-safe trajectory is executed. The set-based prediction module and the fail-safe trajectory planning module are briefly described below.



(a) Initial Scenario



(b) Future Scenario

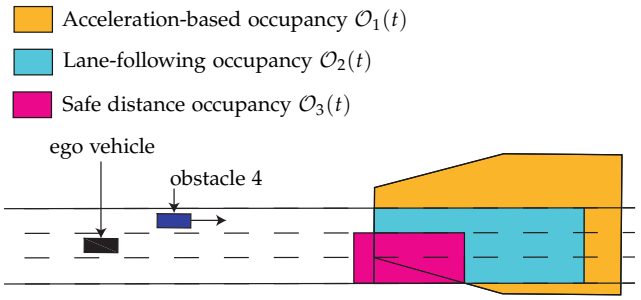
Figure 2. Fail-safe planning for online verification [6].

2.3.1. Set-Based Prediction

The set-based prediction tool SPOT [14] predicts the set of future occupancies of other traffic participants based on reachable sets, considering physical constraints and assuming that traffic participants abide by the traffic rules. Assumptions for each traffic participant are removed individually as soon as a violation of a traffic rule is detected. The set of the overall occupancy of a traffic participant is obtained as

$$\mathcal{O}(t) = \mathcal{O}_1(t) \cap \mathcal{O}_2(t) \cap \mathcal{O}_3^{\complement}(t), \tag{1}$$

where $\mathcal{O}_1(t)$ denotes the acceleration-based occupancy, $\mathcal{O}_2(t)$ denotes the lane-following occupancy, and $\mathcal{O}_3^{\complement}(t)$ denotes the complement of the safe distance occupancy $\mathcal{O}_3(t)$, as depicted in Fig. 3. For computing the reachable sets, SPOT requires the current



(a) Separate occupancies $\mathcal{O}_1(t)$, $\mathcal{O}_2(t)$, and $\mathcal{O}_3(t)$.

(b) Combined occupancy $\mathcal{O}(t) = \mathcal{O}_1(t) \cap \mathcal{O}_2(t) \cap \mathcal{O}_3^{\complement}(t)$.

Figure 3. Occupancy of Obstacle 4 in a Multi-Lane Highway Scenario. The plot shows the initial configuration at t_0 and the predicted occupancies $\mathcal{O}(t)$ for $t \in [1.0\,\mathrm{s}, 1.5\,\mathrm{s}]$ [14].

set of possible states of the ego vehicle capturing measurement uncertainties.

2.3.2. Fail-Safe Trajectory Planning

The module for fail-safe trajectory planning requires an intended trajectory and the set-based prediction of other traffic participants. Before we can compute the fail-safe trajectory, the time t_{TTR}, at which the fail-safe trajectory branches off the intended trajectory, has to be determined. For t_{TTR}, we choose the Time-To-React (TTR), which is the maximum time we can continue the intended trajectory before we have to execute an evasive maneuver to avoid a possible collision [16]. Starting from the state at t_{TTR}, we compute an evasive trajectory based on [17], which is checked for collision with the reachable sets of other traffic participants. The fail-safe module returns a verified trajectory consisting of the intended trajectory for $t < t_{\mathrm{TTR}}$ and an evasive trajectory.

3. CommonRoad-Apollo Interface

The interfaces of Apollo 3.0 and Apollo 5.0 have different structures because of the different versions of operation systems provided by Apollo docker images. Both interfaces convert Apollo messages to CommonRoad Python classes and vice versa. Since our motion planner uses a different time step size than the Apollo planner, trajectories must be resampled via linear interpolation.

3.1. Structure

CommonRoad-Apollo3.0 Interface: Since Apollo 3.0 does not support Python 3, we had to build Python 3 as a Bazel library, which enables us to integrate our Python scripts directly into the Apollo Planning module in C++, as depicted in Fig. 4. Our interface accesses the C++ objects directly and converts them to Python CommonRoad objects, which is directly used in the Python scripts calling our safety modules.

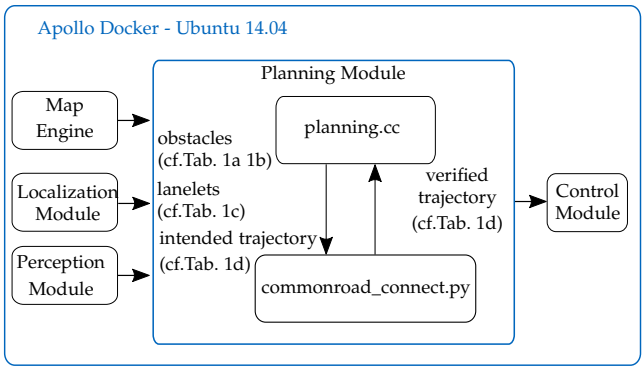


Figure 4. Structure of the CommonRoad-Apollo 3.0 Interface.

CommonRoad-Apollo5.0 Interface: Since Apollo 5.0 allows us to send messages from outside the Docker system without building a communication mechanism, we can deploy our planner module as an independent module running on the host machine, as shown in Fig. 5, which significantly simplifies our interface. The interface subscribes the corresponding messages from Apollo, transfers them to the CommonRoad planner, and publishes the planning messages generated by the CommonRoad planner to Apollo.

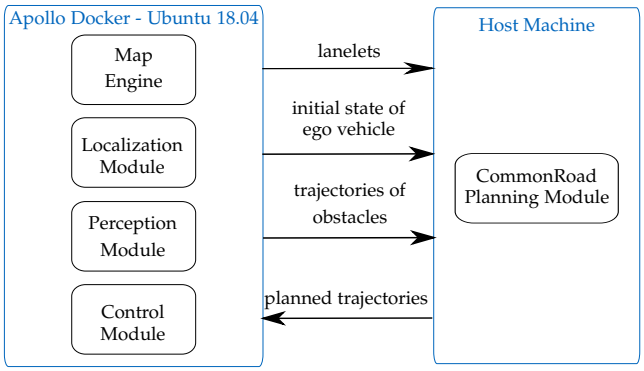


Figure 5. Structure of the CommonRoad-Apollo 5.0 Interface.

3.2. Message Conversion

The actual conversion of the attributes of the Apollo messages and CommonRoad Python classes for both interfaces is very similar. The only difference is that

C++-Python conversion is used for Apollo 3.0 and messages subscription and publication is used for Apollo 5.0. Therefore, we only introduce the message conversion for Apollo 3.0. For more details about message conversion for Apollo 5.0, please refer to the source code at commonroad.in.tum.de.

As previously mentioned, our software module requires the road network, the initial states of the other traffic participants, and the intended trajectory of the planner that needs to be verified, as shown in Fig. 4. The conversion between the corresponding Apollo messages and CommonRoad Python objects is shown in Fig. 6. Since Apollo uses a UTM coordinate system and CommonRoad uses a local coordinate system, we convert the coordinates of the obstacles, the waypoints of the lanes, and the intended trajectory according to the current position of the ego vehicle, which is provided by the `LocalizationEstimate` message.

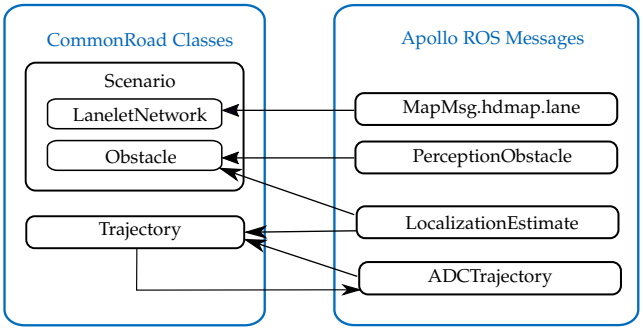


Figure 6. Message Conversion of the CommonRoad-Apollo Interface.

Table 1 shows how we convert each attribute of Apollo messages to CommonRoad Python classes. Note that the intended trajectory of the Apollo planner consists of multiple `ADCTrajectoryPoints`, while the fail-safe trajectory of our planner consists of multiple `States`.

3.3. Sampling of Trajectories

Since our fail-safe planner has a time step size of $\Delta t = 0.2$s and the time step size of Apollo `ADCTrajectory` changes from 0.02 s to 0.1 s, we have to downsample the trajectory points of Apollo `ADCTrajectory` to 0.2 s and upsample the CommonRoad `Trajectory` to 0.02 s and 0.1 s. We sampled the points of the trajectories using linear interpolation at certain time steps. We denote the first time step of Apollo `ADCTrajectory` by t_{cp}, which has a time interval of 0.1 s from its previous point (e.g., $t_{\text{cp}} = 13$ in Fig. 7a). To downsample an Apollo `ADCTrajectory` to a CommonRoad `Trajectory` for $t < t_{\text{cp}}$, we take one point from the Apollo trajectory every ten points; for $t \geq t_{\text{cp}}$, we linearly interpolate points of the Apollo trajectory, as depicted in Fig. 7a. To upsample a CommonRoad

Table 1. Conversion between Apollo ROS Messages and CommonRoad Classes

(a) Conversion of Obstacles

Apollo PerceptionObstacle	CommonRoad Obstacle
PerceptionObstacle.position.x - LocalizationEstimate.pose.position.x	initial_state.position[0]
PerceptionObstacle.position.y - LocalizationEstimate.pose.position.y	initial_state.position[1]
$(\text{PerceptionObstacle.velocity.x}^2 + \text{PerceptionObstacle.velocity.y}^2)^{\frac{1}{2}}$	initial_state.velocity
PerceptionObstacle.theta	initial_state.orientation
PerceptionObstacle.width	obstacle_shape.width
PerceptionObstacle.length	obstacle_shape.length
PerceptionObstacle.type	obstacle_type

(b) Conversion of Obstacle Types

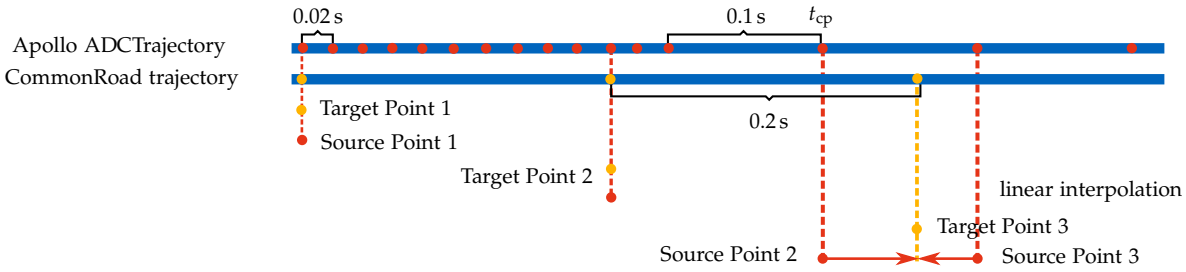
Apollo PerceptionObstacle.type	CommonRoad Obstacle
UNKNOWN_MOVABLE	UNKNOWN
UNKNOWN_UNMOVABLE	PARKED_VEHICLE
PEDESTRIAN	PEDESTRIAN
BICYCLE	BICYCLE
VEHICLE	CAR

(c) Conversion of Lanelets

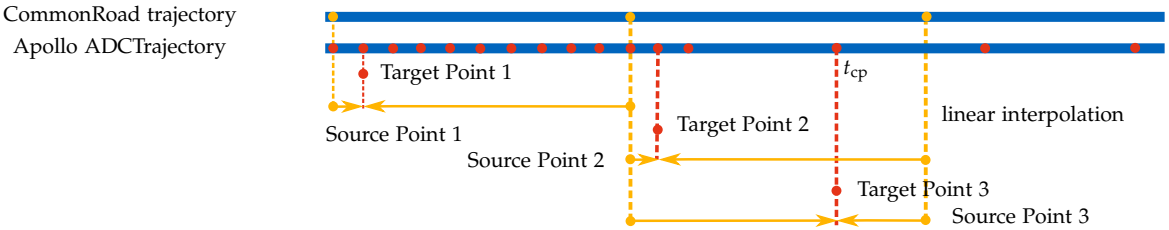
Apollo MapMsg.hdmap.lane	CommonRoad Lanelet
id.id.data	lanelet_id
left_boundary.curve.segment. line_segment.point -LocalizationEstimate.pose.position	left_vertices
right_boundary.curve.segment. line_segment.point - LocalizationEstimate.pose.position	right_vertices
successor_id	successor
predecessor_id	predecessor
left_neighbor_forward_lane_id	adjacent_left_same_direction
left_neighbor_reverse_lane_id	adjacent_left
right_neighbor_forward_lane_id	adjacent_right_same_direction
right_neighbor_reverse_lane_id	adjacent_right
speed_limit	speed_limit

(d) Conversion of Trajectory State

Apollo ADCTrajectoryPoint	CommonRoad State
ADCTrajectoryPoint.path_point.x - LocalizationEstimate.pose.position.x	state.position[0]
ADCTrajectoryPoint.path_point.y - LocalizationEstimate.pose.position.y	state.position[1]
ADCTrajectoryPoint.path_point.theta	state.orientation
ADCTrajectoryPoint.path_point.kappa	state.yaw_rate
ADCTrajectoryPoint.v	state.velocity
ADCTrajectoryPoint.a	state.acceleration
ADCTrajectoryPoint.relative_time	state.time_step



(a) Downsampling of an Apollo ADCTrajectory to a CommonRoad trajectory.



(b) Upsampling of a CommonRoad trajectory to an Apollo ADCTrajectory.

Figure 7. Sampling of trajectory points between an Apollo ADCTrajectory and a CommonRoad trajectory. Source points are the points of the trajectory that should be converted, whereas target points are the points of the trajectory that we obtain after the conversion.

`Trajectory` to an Apollo `ADCTrajectory`, we linearly interpolate points of the CommonRoad trajectory to 0.02 s for $t < t_{cp}$ and 0.1 s for $t \geq t_{cp}$, as depicted in Fig. 7b.

4. Evaluation

We demonstrate our interface and our software modules in two scenarios, where the second one is more complex. The maps of the scenarios are offered by the open-source LGSVL simulator and the obstacles are generated offline from SUMO [1] through our CommonRoad-SUMO interface [10].

Fig. 8 shows an example scenario visualized in Apollo Dreamview. We recorded the test scenarios in CommonRoad format and visualized the results in CommonRoad for offline analyses, as shown in Fig. 9. In Fig. 9a, the predicted occupancies of the other vehicle do not intersect with the planned trajectory so that $t_{\text{TTR}} = t_{\text{end}}$, where t_{end} is the final time step of the intended trajectory. In such a situation, the fail-safe planner concatenates a braking trajectory to the intended trajectory to verify the intended plan as safe for all times. In Fig. 9b, the intended trajectory turns left at the intersection, which intersects with the predicted occupancies of the other vehicles. Therefore, the fail-safe planner computes a braking trajectory from t_{TTR}, as presented in Sec. 2.

Table. 2 shows the individual computation times for the two demonstration scenarios of each module. The computation times have been obtained using a single

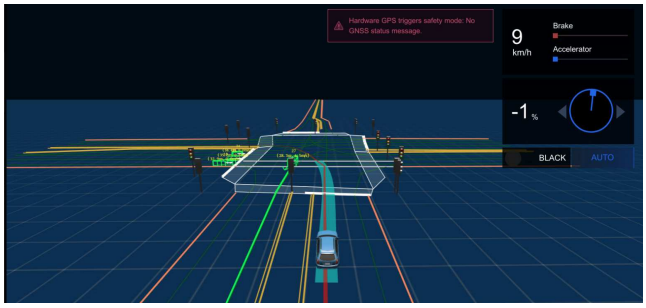
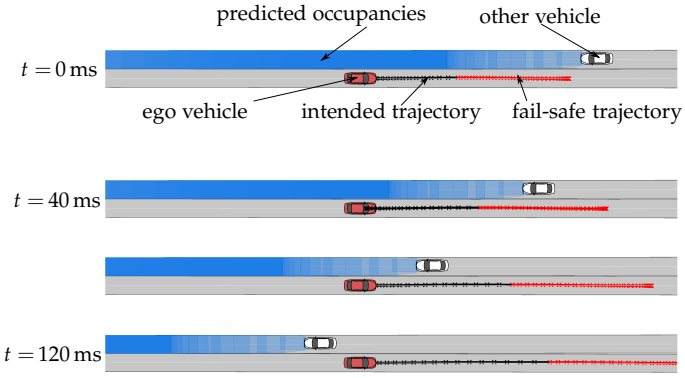


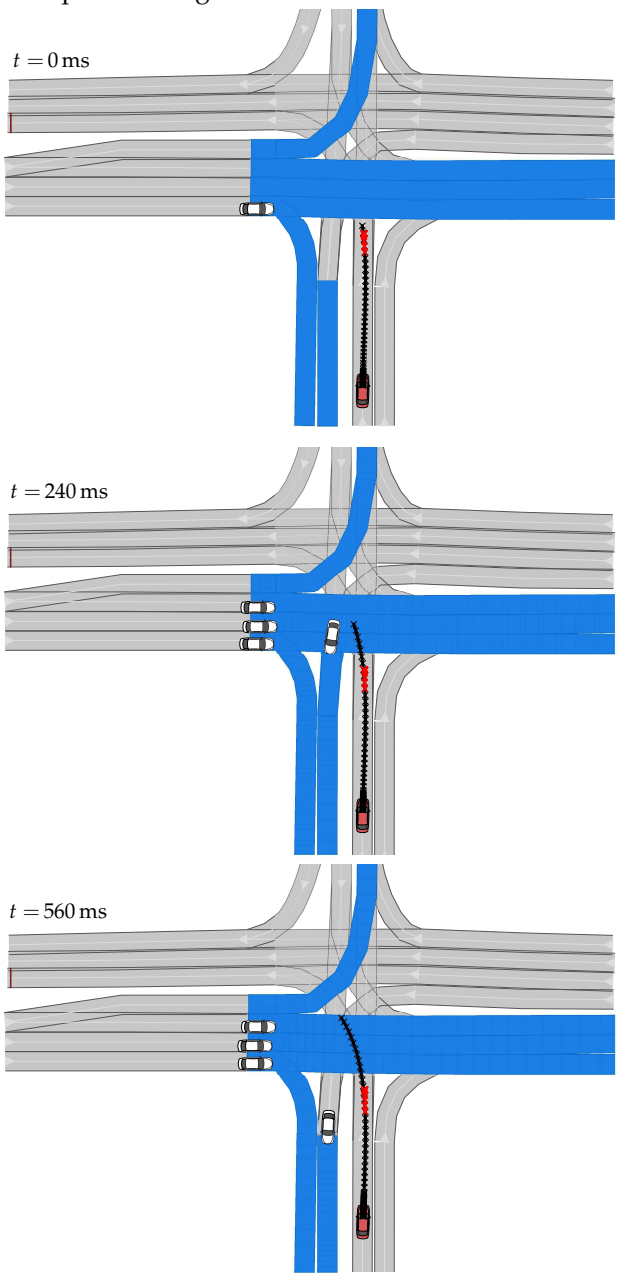
Figure 8. Example Scenario of Borregas Ave Map Visualized in Dreamview. The green boxes depict the obstacles. The red curve shows the rough path that the ego vehicle needs to follow according to the routing module. The blue curve shows the planned trajectory by the ego vehicle.

Table 2. Computation Times

Module	Single Lane Road	Borregas Ave
Apollo-CommonRoad Interface	3 ms	24 ms
Set-Based Prediction	0.75 ms	22.67 ms
Fail-Safe Planner	46.76 ms	130.47 ms



(a) An example scenario within a single lane road map containing one obstacle.



(b) An example scenario within Borregas Ave map containing four obstacles.

Figure 9. Example Scenarios Generated in Apollo 3.0 and Visualized in CommonRoad.

core on a machine with an Intel Xeon W-2155 3.30 GHz processor and 32 GB of DDR4 2666 MHz memory.

5. Conclusion

In this paper, we present an interface between the planning module of Apollo and the open-source CommonRoad motion planning framework. Our interface converts the road network and obstacle information between Apollo and CommonRoad. Combining the presented interface and the CommonRoad-SUMO interface, we can generate traffic for offline testing in Apollo. Developers can first test their planners in diverse scenarios from the CommonRoad benchmark suite and directly on a real vehicle afterwards using the Apollo platform with our interface. We have demonstrated our interface and our safety modules in a simple two-lane scenario as well as a complex urban scenario. Numerical experiments confirm the real-time capability of our software.

References

[1] D. Krajzewicz, G. Hertkorn, C. Rössel, and P. Wagner, "SUMO (Simulation of Urban MObility) - an open-source traffic simulation," in *Proc. of the Middle East Symposium on Simulation and Modelling*, 2002, pp. 183–187.

[2] R. Krajewski, J. Bock, L. Kloeker, and L. Eckstein, "The highD Dataset: A drone dataset of naturalistic vehicle trajectories on German highways for validation of highly automated driving systems," in *Proc. of the IEEE Int. Conf. on Intelligent Transportation Systems*, 2018, pp. 2118–2125.

[3] M. Althoff, M. Koschi, and S. Manzinger, "CommonRoad: Composable benchmarks for motion planning on roads," in *Proc. of the IEEE Intelligent Vehicles Symposium*, 2017, pp. 719–726.

[4] M. Althoff and J. M. Dolan, "Online verification of automated road vehicles using reachability analysis," *IEEE Transactions on Robotics*, vol. 30, no. 4, pp. 903–918, 2014.

[5] S. Magdici and M. Althoff, "Fail-safe motion planning of autonomous vehicles," in *Proc. of the IEEE Int. Conf. on Intelligent Transportation Systems*, 2016, pp. 452–458.

[6] C. Pek, S. Manzinger, M. Koschi, and M. Althoff, "Using online verification to prevent autonomous vehicles from causing accidents," *Nature Machine Intelligence*, vol. 2, no. 9, pp. 518–528, 2020.

[7] J. Ziegler, P. Bender, M. Schreiber, H. Lategahn *et al.*, "Making bertha drive - an autonomous journey on a historic route," *IEEE Intelligent Transportation Systems Magazine*, vol. 6, no. 2, pp. 8–20, 2014.

[8] T. Kessler, J. Bernhard, M. Buechel, K. Esterle *et al.*, "Bridging the gap between open source software and vehicle hardware for autonomous driving," in *Proc. of the IEEE Intelligent Vehicles Symposium*, 2019, pp. 1612–1619.

[9] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez *et al.*, "CARLA: An open urban driving simulator," in *Proceedings of the 1st Annual Conference on Robot Learning*, 2017, pp. 1–16.

[10] M. Klischat, O. Dragoi, M. Eissa, and M. Althoff, "Coupling SUMO with a motion planning framework for automated vehicles," in *SUMO User Conference*, 2019.

[11] V. Punzo, M. T. Borzacchiello, and B. Ciuffo, "On the assessment of vehicle trajectory data accuracy and application to the Next Generation SIMulation (NGSIM) program data," *Transportation Research Part C: Emerging Technologies*, vol. 19, no. 6, pp. 1243 – 1262, 2011.

[12] W. Zhan, L. Sun, D. Wang, H. Shi *et al.*, "INTERACTION dataset: An INTERnational, Adversarial and Cooperative moTION dataset in interactive driving scenarios with semantic maps," *ArXiv*, vol. abs/1910.03088, 2019.

[13] C. Pek, V. Rusinov, S. Manzinger, M. Can Üste *et al.*, "CommonRoad drivability checker: Simplifying the development and validation of motion planning algorithms," in *Proc. of the IEEE Intelligent Vehicles Symposium*, 2020, pp. 1–8.

[14] M. Koschi and M. Althoff, "SPOT: A tool for set-based prediction of traffic participants," in *Proc. of the IEEE Intelligent Vehicles Symposium*, 2017, pp. 1686–1693.

[15] H. Fan, F. Zhu, C. Liu, L. Zhang *et al.*, "Baidu Apollo EM motion planner," *arXiv preprint arXiv:1807.08048*, 2018.

[16] J. Hillenbrand, A. M. Spieker, and K. Kroschel, "A multilevel collision mitigation approach - its situation assessment, decision making, and performance tradeoffs," *IEEE Transactions on Intelligent Transportation Systems*, vol. 7, pp. 528–540, 2006.

[17] M. Werling, J. Ziegler, S. Kammel, and S. Thrun, "Optimal trajectory generation for dynamic street scenarios in a frenét frame," in *Proc. of the IEEE International Conference on Robotics and Automation*, 2010, pp. 987–993.

Acknowledgement

The authors would like to thank Zhenyu Li for his work on the implementation of the Apollo-CommonRoad-Interface and Jiaying Huang for generating traffic from SUMO to Apollo for the numerical experiments. We gratefully acknowledge the partial financial support provided to us by the Ford Motor Company and the German Research Foundation (DFG) under grant number AL 1185/3-2.