UNIVERSITY OF CALIFORNIA,
IRVINE

Transferring Test Scenarios Between Autonomous Driving Systems

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Software Engineering

by

Changnam Hong

Thesis Committee:
Assistant Professor Joshua Garcia, Chair
Professor Sam Malek
Assistant Professor Qi Alfred Chen

2024

TABLE OF CONTENTS

Page

LIST OF FIGURES **iv**

LIST OF TABLES **vi**

LIST OF ALGORITHMS **vii**

ACKNOWLEDGMENTS **viii**

ABSTRACT OF THE THESIS **ix**

1 Introduction **1**

2 Background and Related Work **4**
2.1 Background 4
2.1.1 Autonomous Driving Software 4
2.1.2 Autonomous Driving Software and Simulator 5
2.2 Related work 7
2.2.1 Scenario Generation Approaches 7

3 Approach **10**
3.1 Architecture Overview 10
3.2 Coordinate Conversion 12
3.2.1 Lane Finding Algorithm 13
3.3 Transformation 16
3.3.1 Transformation Source in Apollo Scenario 16
3.3.2 Movement Patterns of Obstacles 17
3.3.3 Status Changes of Traffic Signals 21
3.4 Embedding Routing Trajectory of the Ego Car 23

4 Implementation **25**

5 Evaluation **26**
5.1 Experiment Setup 27
5.2 RQ1: Fidelity of Transformed Scenarios 27
5.3 RQ2: Ego Car Behaviors in Original and Transformed Scenarios 30

6 Future Work **34**

7 Conclusion **36**

Bibliography **37**

LIST OF FIGURES

Page

3.1 ADS Scenario Transformer Architecture 11
3.2 ADS Scenario Transformer Coordinate Conversion Process 12
3.3 Lane Determination Based on Routing Trajectory Points and Routing Graph. The object begins its movement on lane 12 and ends on lane 144. By utilizing a routing graph, we can construct a path from lane 12 to 144 that necessarily includes lane 210 along its route, as the object should pass through it. Conversely, the route from lane 12 to 144 will not include lane 298, as it is not part of this path. Therefore, we can conclude that the target point belongs to lane 210. 15
3.4 Example of calculating velocity conversion for each obstacle 19
3.5 Input space and Output Models of Traffic Signal Transformation 21

5.1 Comparison of results shown by the transformed scenario and the original scenario on the simulator. The upper image shows the original Apollo scenario; the bottom image shows the transformed Autoware scenario. 28
5.2 A case where incomplete vehicle speed control alters the behavior of the ego car. In the original scenario, the obstacle stops at the routing path of the ego car, but it stops after passing the routing path in the transformed scenario. . 29
5.3 Comparison of obstacle movement predictions by ego cars: In the Apollo scenario (left), the ego car predicts obstacles that are expected to enter its driving path and yields to those obstacles to move safely, as indicated by the purple fence. In contrast, in the Autoware scenario (right), the ego car does not yield to or stop for an obstacle unless that obstacle is exactly in its driving path. 31
5.4 A case where the result of the scenario changes due to differences in ADS speed control. In the Apollo scenario, the ego car successfully overtakes a stationary vehicle. Conversely, in the Autoware scenario, the ego car changes lanes slowly, allowing the vehicle on the road to go first, thereby preventing the ego car from completing the overtaking maneuver. 32

5.5 Comparison between Apollo and Autoware ego car's movement in exiting junction if a pedestrian is on the crosswalk. In the Apollo scenario, the ego car does not stop if pedestrians are not in danger due to the car's movement in green light. However, in the Autoware scenario, the ego car always stops in front of the crosswalk if a pedestrian is present, regardless of their movement and traffic light states. 33

LIST OF TABLES

Page

2.1 Scenario and Map Format Specification for ADS-Simulator Environments . . 5
2.2 ADS-Simulator environments used by each approach to demonstrate their work 8

LIST OF ALGORITHMS

		Page
1	Target Lane Finding Algorithm	14
2	Obstacle Routing Trajectory Transformation	18
3	Obstacle Speed Transformation	20
4	Traffic Signal Transformation	22
5	Embedding Routing Trajectory	24

ACKNOWLEDGMENTS

I would like to express my gratitude to my advisor, Professor Joshua Garcia, for his invaluable guidance and support, and to Professor Qi Alfred Chen, whose assistance was essential throughout this journey. Their expertise and insights have been instrumental in the completion of this work.

I am also profoundly thankful to all my research lab members, Yuqi Huai, Yuntianyi Chen, and Shilong Li, for their collaboration and support. Their feedback and assistance have been invaluable, and their friendship has made this experience truly rewarding.

Thank you all for being integral parts of this journey.

ABSTRACT OF THE THESIS

Transferring Test Scenarios Between Autonomous Driving Systems

By

Changnam Hong

Master of Science in Software Engineering

University of California, Irvine, 2024

Assistant Professor Joshua Garcia, Chair

Autonomous vehicles (AVs) are rapidly becoming a part of our everyday lives, and the number of AVs running on the road is growing faster. With the increased presence of AVs, there has been a rise in accidents, which underscores the critical need for enhanced safety and reliability in Autonomous Driving Software (ADS). Recent studies have proposed scenario-generation approaches to facilitate testing autonomous vehicles in broader scenarios. However, many of these approaches create scenarios that only run within a single ADS-Simulator environment, affecting the generalizability of the resulting approaches. One primary reason scenario generation approaches are limited to specific ADS-simulator environments is that each system uses distinct scenario and map formats. This format incompatibility limits the utility of scenarios that expose critical bugs, which is essential for improving ADS safety. To that end, we propose an ADS Scenario Transformer that enables transfer scenarios running on one ADS-Simulator to others. The transformation process targets two critical elements of a driving scenario: (1) the movement patterns of obstacles within the simulation, and (2) status changes of traffic signals. Our study investigates whether scenarios are transferred between different ADS-simulator environments while preserving original scenario information and explores differences in ADS behavior between original and transformed scenarios.

Chapter 1

Introduction

Autonomous vehicles (AVs) are rapidly becoming a part of our everyday lives, and the number of AVs running on the road is growing faster. The California DMV indicates that AVs have logged over 9 million miles of testing in the state within 12 months (December 1, 2022 - November 30, 2023). With the increased presence of AVs on the roads, there has been a rise in accidents, totaling 705 collisions involving AVs as reported up to April 25, 2024 [6, 7]. The growing number of accidents underscores the critical need for enhanced safety and reliability in Autonomous Driving Software (ADS). To address these concerns, researchers and developers in the ADS industry test their ADS in Simulation Testing, Closed-Course Testing, and Real-World Driving stages [22]. Among these, simulation testing offers notable benefits in terms of time efficiency and cost-effectiveness. Specifically, it enables testing ADS functions without deploying them in software on physical vehicles. Moreover, simulation testing allows for the repeated reconstruction of real-world events that are difficult to reproduce, such as vehicle collisions and traffic rule violations [13].

Given these advantages, recent studies have proposed scenario-generation approaches to facilitate testing AVs in broader scenarios [10, 11, 12, 23]. However, many of these approaches

create scenarios that only run within specific ADS-Simulator environments, affecting the generalizability of the resulting approaches. If researchers want to reproduce scenarios in diverse ADS-Simulator environments, they still need to manually modify scenarios, often leading to numerous errors during conversion. One of the main reasons for this irreproducibility is that ADS simulators use distinct scenarios and map formats. Different ADS have evolved to meet unique requirements, leading to the selection of particular formats for scenarios and maps. For instance, the format of the Apollo scenario is Cyber Record, which allows Apollo/SimControl to replay it directly without any modifications [4]. Conversely, Autoware adopts the modified OpenScenario format to enhance compatibility with existing open-source tools when executing scenarios through Scenario Simulator v2 [17, 19].

Additionally, this format incompatibility limits the utility of scenarios that expose critical bugs, essential for improving ADS safety. Not all testing scenarios carry the same weight; some merely verify the functionality of the AVs, while others reveal critical bugs that provide valuable insights for enhancing safety [16]. For example, some scenarios might show the AV causing a collision or speeding violation under certain circumstances. However, although studies have identified critical violation-revealing scenarios, the scenarios' non-reproducibility limits test them across different ADSs to confirm if they consistently reveal violations.

To that end, we propose an ADS Scenario Transformer that enables transfer scenarios running on one ADS-Simulator to the others. In particular, ADS Scenario Transformer transfers scenarios running on Apollo and SimControl environments to scenarios running on Autoware and TIER IV Scenario Simulator v2 environments. The transformation process targets two critical elements of a driving scenario: (1) the movement patterns of obstacles within the simulation, and (2) the status changes of traffic signals.

Our research reproduces the same scenario in different ADS-Simulator environments. This approach allows a single scenario to be executed across different ADS-simulator environments, enabling researchers to address the following research question accordingly.

- RQ1: How well does the transformed scenario preserve information from the original scenario?
- RQ2: How do the ego car behaviors compare in the original and transformed scenarios?

We evaluated the ADS Scenario Transformer by manually transforming and analyzing 13 scenarios to address these research questions. Additionally, during this investigation, we identified cases where Apollo and Autoware's ego cars behave differently in similar environments, and we examined these differences. Contributions of this paper are as follows:

- A well-structured approach to automatically transfer Apollo scenarios to Autoware Scenarios while conserving their original context and core information.
- Protocol Buffer models based on the OpenScenario v1.2 specification are fully reusable in future projects.

Chapter 2

Background and Related Work

2.1 Background

2.1.1 Autonomous Driving Software

Autonomous Driving Software(ADS) consists of highly sophisticated and complex modules that interact to achieve autonomous driving. Among the various ADSes, Baidu's Apollo [4] and the Autoware Foundation's Autoware [17] are leading open-source ADSes. Due to their complexity, these software systems operate through multiple modules, each with a specific role. Although Apollo and Autoware have slightly different naming conventions, they share modules, each performing the following functions:

- **Perception** detects and tracks the movement of objects and traffic status. The Perception module uses information collected from LiDAR, cameras, and various sensors to understand the surrounding environment.

- **Localization** determines the vehicle's location, orientation, and speed. It combines map data and sensor information to provide details on where the AV is, which direction it is facing, and how fast it is moving.

- **Planning** constructs the vehicle's driving path. Based on information from the Perception and Localization modules, the Planning module creates the vehicle's driving route. In Apollo, a separate Prediction module anticipates the positions of surrounding vehicles. In contrast, Autoware integrates such functions within its Mission Planning and other Planning calculations.

- **Control-CAN Bus** translates planning into vehicle action. Known as the Control-Vehicle Interface in Autoware, this module executes the plans generated by the Planning module by sending commands to the vehicle's actuators to control speed, steering, and braking.

2.1.2 Autonomous Driving Software and Simulator

State-of-the-art ADS use various testing approaches to ensure safety and reliability, one of which is simulation testing. Simulation testing involves using simulators to evaluate an ADS in different virtual scenarios. Table 2.1 presents two leading open-source ADS platforms, Apollo and Autoware, along with their respective simulators.

Table 2.1: Scenario and Map Format Specification for ADS-Simulator Environments

ADS	Simulator	Scenario Description Format	Map Description Format	Ownership
Apollo	SimControl	Cyber Record	Apollo HD Map	First-party
	LGSVL	VSE Scenario		Third-party
	Carla	OpenScenario	OpenDrive	Third-party
Autoware	Scenario simulator v2	TIER IV Scenario Format or OpenScenario	Autoware Vector Map	First-party
	AWSIM			
	LGSVL	VSE Scenario		Third-party
	Carla	OpenScenario	OpenDrive	Third-party

Apollo uses its native simulator, SimControl, while Autoware employs Scenario Simulator v2 [19] and AWSIM [18]. Each of these first-party simulators requires unique scenario and map formats. TIER IV, a member of the Autoware Foundation, is actively developing the simulators for Autoware. One of their simulators, Scenario Simulator v2, is designed to test Autoware's Planning Module. It operates based on the TIER IV Scenario Format and Autoware Vector Map. The TIER IV Scenario Format is an extended version of OpenSCENARIO [3], allowing Scenario Simulator v2 to play scenarios in the OpenSCENARIO format. AWSIM, also developed by TIER IV, is a Unity-based simulator visualizing 3D simulations. AWSIM uses the scenario execution engine of Scenario Simulator v2, making scenarios and maps compatible with the two simulators.

LGSVL [15] was widely used in the field but was deprecated in 2022, so it only supports older versions of Apollo and Autoware. Alternatively, the Carla simulator [9] can run simulations using the same format for scenarios (OpenSCENARIO) and maps (OpenDrive [2]). This compatibility is a significant benefit as it allows an ADS to operate using a single format for scenarios and maps. However, Carla requires a bridge package to communicate with each ADS, and this bridge still needs to be mature enough to run scenarios reliably.

Scenario and Scenario-based Testing

Ulbrich et al. [21] provide a unified definition of a scenario for the field of ADS, describing it as the temporal progression through a series of scenes over time, including the actions and events of various objects as they work towards achieving defined goals. Scenario-based testing treats each scenario as a test case, evaluating whether it successfully achieves its goals throughout the series of scenes. This approach is widely employed in the ADS domain due to the complicated nature of the environment and the system, highlighting AVs should be tested with comprehensive scenarios [5]. Scenario-based testing is cost-effective because it communicates solely through the control module that manages the hardware rather than

interacting directly with the ADS hardware. Additionally, it is time-efficient, as functionality can be tested without the need for hardware deployment.

2.2 Related work

2.2.1 Scenario Generation Approaches

Recent studies have proposed integrating ADS domain knowledge with traditional software testing techniques, employing test input generation techniques to facilitate the testing of autonomous vehicles in a broader range of scenarios. Some significant recent studies are as follows. AutoFuzz [23] mutates existing driving scenarios by altering initial scene configurations, such as the ego car's route and the movements of obstacles, generating semantically valid scenarios that can reveal traffic violations. To achieve this, AutoFuzz uses a constrained neural network in its mutation strategy, which allows it to explore a more extensive search space and produce a wider variety of output scenarios. scenoRITA [10] generates scenarios incorporating domain-specific constraints and diverse obstacles by leveraging genetic algorithms. It allows the obstacles to be entirely mutable, creating a wide range of testing conditions. In addition to generating new scenarios, scenoRITA includes an oracle that replays scenarios to detect and eliminate duplicate violations. This duplicate scenario removal ensures that the final output consists of unique scenarios where the ADS has detected violations.

DriveFuzz [12] generates diverse and dynamic scenarios by mutating components such as the mission, actors, puddles, and weather. It randomly changes the destination of the ego car to alter the scenario's goal. DriveFuzz also places vehicles or pedestrians at random locations and configures the ego car with randomly chosen settings. Additionally, it creates puddles or changes weather conditions to make the scenarios more realistic. DoppelTest [11] creates

scenarios that reveal bugs by ensuring that the ADS under test is responsible for keeping all vehicles in the simulation following traffic rules and reacting appropriately to each other. Additionally, DoppelTest uses genetic algorithms to create diverse scenarios by adding more traffic participants, thereby increasing the complexity and variability of the scenarios.

Scenarios can also be generated by transferring them from the other source of scenarios. SCTrans [8] transforms scenarios based on real-world records into simulation scenarios compatible with Apollo-LGSVL and Autoware-Carla environments leveraging a model transformation approach. Since the source of scenarios is based on real-world records, it outputs diverse and realistic scenarios containing the natural behaviors of road traffic participants. Also, SCTrans formalizes its transformation process using a model transformation approach to ensure consistent and accurate application of its transformation rules.

Table 2.2: ADS-Simulator environments used by each approach to demonstrate their work

Approach	ADS-Simulator
AutoFuzz	None-Carla Apollo-LGSVL
DriveFuzz	Autoware-Carla
scenoRITA	Apollo-SimControl
DoppelTest	
SCTrans	Apollo-LGSVL Autoware-Carla

The approaches discussed in each study create executable scenarios within specific ADS-simulator environments. Table 2.2 highlights the ADS-simulator environments used by each study to create these playable scenarios. AutoFuzz showcases its work using Carla and performs initial testing in the Apollo-LGSVL environment. DriveFuzz evaluates its method within the Autoware-Carla environment. Both scenoRITA and DoppelTest create scenarios for Apollo-SimControl. SCTrans produces scenarios compatible with both Apollo-LGSVL and Autoware-Carla. scenoRITA and DoppelTest work directly with their native simulators,

whereas the other methods rely on third-party simulators and require additional bridge packages to run their scenarios.

Chapter 3

Approach

The ADS Scenario Transformer is a tool to transfer scenarios running on one ADS-Simulator to others. Specifically, it converts scenarios created in the Apollo-SimControl environments, which are in Cyber Record format, into scenarios compatible with the Autoware-Scenario Simulator environments, formatted in TIER IV Scenario format. The transformation process targets two critical elements of a driving scenario: (1) the movement patterns of obstacles within the simulation and (2) the status changes of traffic signals.

3.1 Architecture Overview

Figure 3.1 illustrates the architecture of the ADS Scenario Transformer. ADS Scenario Transformer converts Apollo scenarios in Cyber Record format into Autoware scenarios in TIER IV Scenario format. The process begins with parsing the necessary channel data from the Apollo scenarios. Apollo scenarios store data in multiple channels such as *PerceptionObstacles*, and *TrafficSignals*.

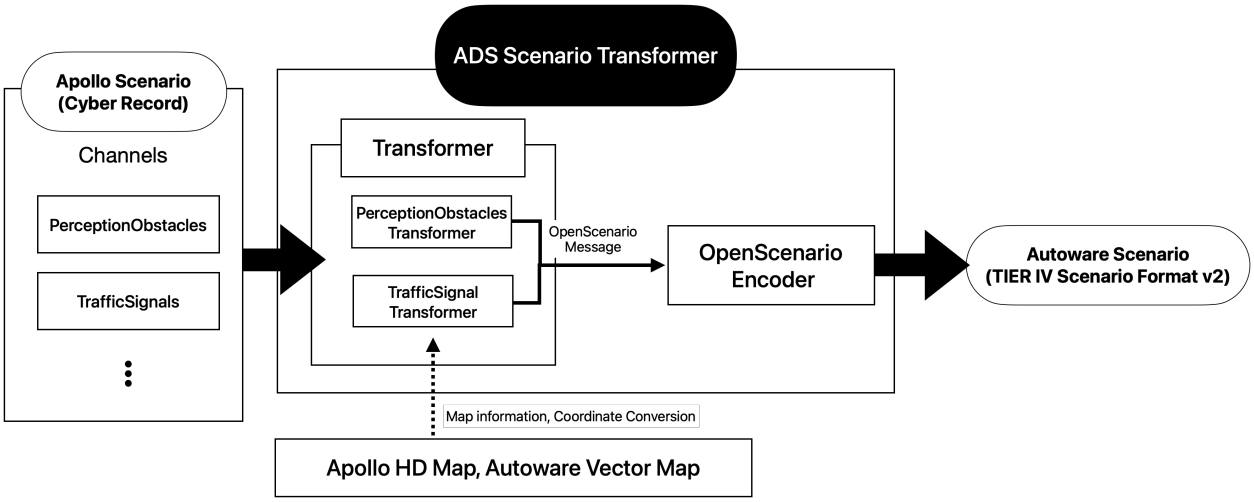


Figure 3.1: ADS Scenario Transformer Architecture

ADS Scenario Transformer defines channel transformers specifically for the channels needed to create Autoware scenarios, with each transformer responsible for generating parts of the OpenSCENARIO message. We chose the OpenSCENARIO format because the TIER IV scenario format is an extended version of OpenSCENARIO, and the scenario simulator is run by internally interpreting the scenarios in OpenSCENARIO format. During this transformation process, some channel transformers require coordinate values projected onto the Autoware Vector map to position objects accurately. These coordinates are initially obtained from the Apollo scenario and projected onto the Autoware Vector map.

Once all channel transformers generate outputs, all outputs are compiled into a single OpenSCENARIO message. This message is not immediately runnable because it does not adhere to the key-value naming rules defined by the TIER IV Scenario format. Therefore, the OpenSCENARIO encoder processes the message to ensure it is ready to run on Autoware-Scenario Simulator environments without any modifications.

3.2 Coordinate Conversion

Coordinate conversion is critical in transforming scenarios, as all routing trajectory points of objects in simulation are represented in coordinate values. Figure 3.2 illustrates the conversion of coordinate values, making them suitable for use in Autoware scenarios.

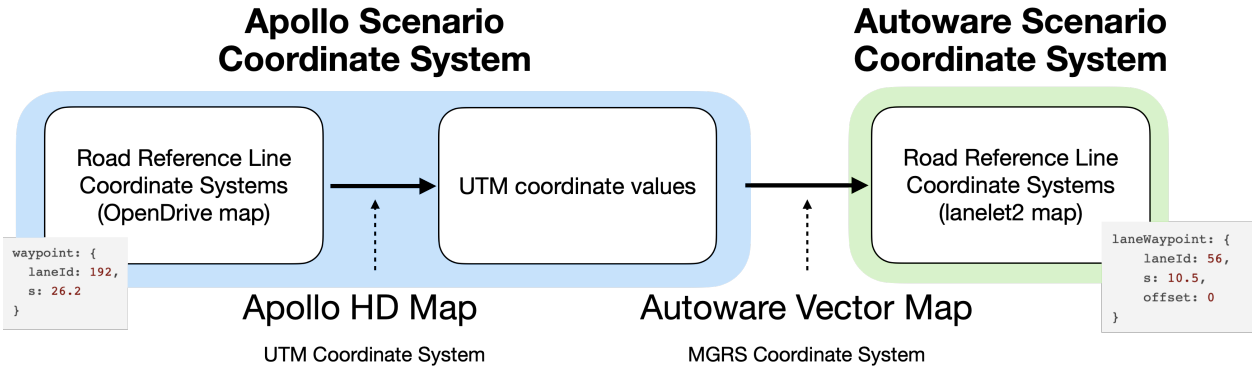


Figure 3.2: ADS Scenario Transformer Coordinate Conversion Process

ADS Scenario Transformer initially converts coordinates from the Apollo HD map into global coordinates and subsequently adapts them into coordinate values based on the Autoware Vector map. This conversion assumes that both the original scenario and the transformed one are based on maps containing the same geographical area aligned with the global coordinate system.

In Apollo scenarios, coordinates are initially in UTM (Universal Transverse Mercator) coordinates or the Road Reference Line Coordinate Systems (RRLCS) [1]. Coordinates in RRLCS format, which include lane identifier and s value (representing the distance between the target point and the beginning of the lane), are initially converted into UTM coordinates. This step is necessary because the specific lane information only pertains to the Apollo HD map. The UTM coordinates are then projected onto the Autoware Vector Map. This projection forwards the global coordinates into a format usable within Autoware scenarios. The projected coordinates are finally transformed into an RRLCS format suitable for use within the Autoware and Scenario simulator environment.

3.2.1 Lane Finding Algorithm

One of the significant challenges in this conversion process occurs when the target coordinate falls within areas where multiple lanes overlap, commonly at junctions. In such cases, it is not straightforward to determine which lane the coordinate belongs to based on the coordinate value itself. Figure 3.3 illustrates such a case where it is challenging to identify the correct lane for a target point marked in red. We cannot decide where the red point belongs by just knowing its coordinate value if it is on the map where multiple lanes overlap. The issue arises because we assume that the Apollo HD map and the Autoware Vector map represent the same geographical region based on global coordinate values for transformation. However, we do not assume that the components defined within the maps have different identifiers. Additionally, in the Apollo scenario, each object is positioned using only coordinate values. However, in the Autoware scenario, each object requires a specific lane identifier to be accurately placed on the map. It is also necessary when objects are positioned where multiple lanes overlap, necessitating the identification of the correct lane for each point.

To mitigate this issue, we utilize the routing trajectory of objects to map coordinates to specific lanes. The Autoware Vector Map includes detailed direction information for each lane and supports the construction of a routing graph that delineates reachable paths for each lane [14]. Figure 3.3 illustrates this process. The figure shows that the object's routing trajectory starts at lane 12, passes through lane 210 in the middle, and ends at lane 144. Using a routing graph, we can construct a route from lane 12 to lane 144 that necessarily includes lane 210, as the object should pass through it. Conversely, the route from lane 12 to lane 144 does not include lane 298, as it is not part of this path. Therefore, we can conclude that the target point belongs to lane 210. The routing trajectory is commonly accessible in our coordinate conversion, as all coordinates are used to transform objects' trajectories.

Algorithm 1 Target Lane Finding Algorithm

Input: $p_{target}, p_{start}, p_{end} \in Point$, $v \in AutowareVectorMap$, $o \in Object$
Output: *lane* where p_{target} belongs to

$RoutingGraph \leftarrow v.graph(o)$
$lanes_{start} \leftarrow$ NEARBY_LANES(v, p_{start}, e)
$lanes_{end} \leftarrow$ NEARBY_LANES(v, p_{end}, e)
$lanes_{target} \leftarrow$ NEARBY_LANES(v, p_{target}, e)
$lanes_{result} \leftarrow Set()$
for $lane_{start}$ **in** $lanes_{start}$ **do**
    for $lane_{end}$ **in** $lanes_{end}$ **do**
        $Route_{[s,e]} \leftarrow RoutingGraph$.ROUTE($lane_{start}$, $lane_{end}$)
        $lanes_{result}$.add(lane in SHORTEST_PATH($Route_{[s,e]}$))
        for $lane_{target}$ **in** $lanes_{target}$ **do**
            $Route_{[s,t,e]} \leftarrow RoutingGraph$.ROUTE($lane_{start}$, $lane_{target}$, $lane_{end}$)
            $lanes_{result}$.add(lane in SHORTEST_PATH($Route_{[s,t,e]}$))
        end for
    end for
end for
for *lane* **in** $lanes_{target}$ sorted by increasing distance **do**
    if *lane* **in** $lanes_{result}$ **then**
        return *lane*
    end if
end for

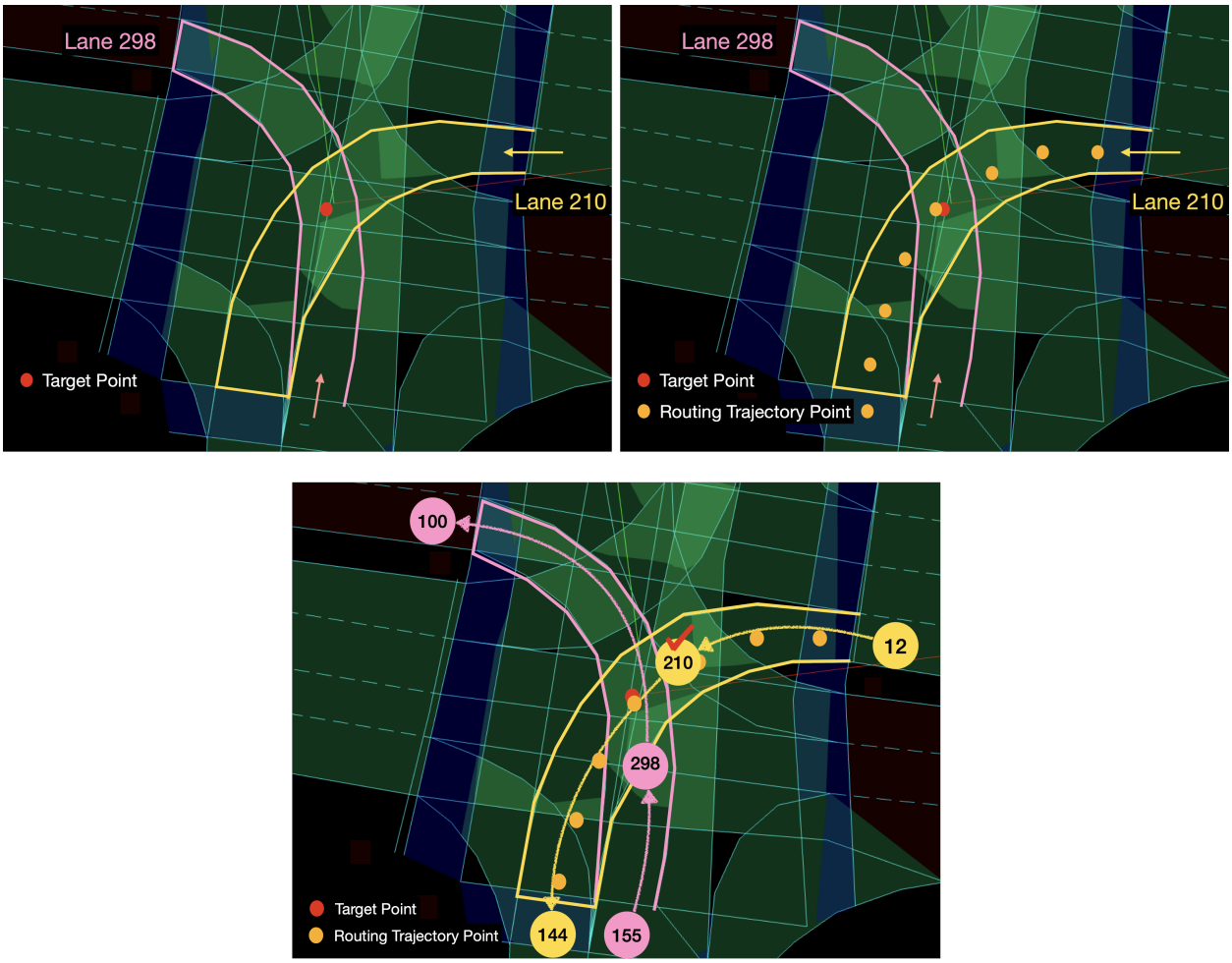


Figure 3.3: Lane Determination Based on Routing Trajectory Points and Routing Graph. The object begins its movement on lane 12 and ends on lane 144. By utilizing a routing graph, we can construct a path from lane 12 to 144 that necessarily includes lane 210 along its route, as the object should pass through it. Conversely, the route from lane 12 to 144 will not include lane 298, as it is not part of this path. Therefore, we can conclude that the target point belongs to lane 210.

Overall, the appropriate lane finding algorithm 1 is as follows: We first construct the map's routing graph, *RoutingGraph*. Then, we identify nearby lanes of the points p_{start}, p_{end}, and p_{target}. Subsequently, we retrieve all paths from p_{start} to p_{end}, from p_{start} to p_{end} with intermediate p_{target}. It's important to note that some routing trajectories may have the same lane for p_{start} and p_{end}. Therefore, we also consider paths from p_{start} to p_{end} with intermediate p_{target} to obtain more precise results. We then obtain the shortest paths for each route and use them as available lanes to which points can belong. Finally, we iterate over the list of

target lanes, $lanes_{target}$, and select the nearest lane, which exists in all available paths and target lanes.

3.3 Transformation

The goal of the transformation process is to conserve the context of the scenario and the behavior of objects. To achieve this, we aim to transform two critical elements of a driving scenario: (1) the movement patterns of obstacles within the simulation, and (2) the status changes of traffic signals.

3.3.1 Transformation Source in Apollo Scenario

The ADS Scenario Transformer reads and parses data from Apollo scenarios to extract critical elements. Apollo scenarios include different channels with data in binary protocol buffer format. Among these channels, the ADS Scenario Transformer reads the *PerceptionObstacles* and *TrafficLight* channels to construct output scenarios.

- **PerceptionObstacles** contains information about all perceived obstacles, including their location, heading, and speed. The transformer uses this data to estimate obstacles' movement patterns.
- **TrafficLight** includes the color of each traffic signal at each timestamp existing on the map. The transformer defines the status change of the traffic signal using this data.

3.3.2 Movement Patterns of Obstacles

In a simulation environment, obstacles are elements that either move around or remain stationary, existing independently of the ego car. ADS perceives and categorizes these obstacles accurately while driving on the roads to anticipate their future actions and ensure the vehicle's safety. To align this purpose, the ADS Scenario Transformer translates the original behavior of obstacles from the Apollo scenario to the Autoware one. In the Apollo scenario, the behavior of obstacles is stored in the *PerceptionObstacles* channel. The ADS Scenario Transformer reads this channel and generates corresponding obstacle behaviors compatible with the Autoware scenario.

ADS Scenario Transformer adds entities to the scenario, considering their type and size. The trajectory of each object is constructed using the location information of each timestamp. In addition, it reflects the obstacle's speed information and accelerates or decelerates appropriately to increase or decrease the speed.

Routing Trajectory Transformation

Converting the routing trajectory of obstacles involves two main tasks: (1) Finding the routing trajectory of each obstacle involves determining the path each obstacle will take. It is similar to what we did for the ego car, but obstacles may begin moving slowly when the simulation starts. Thus, it is necessary to identify each obstacle's movement's start and end time and initiate and stop its motion accordingly. (2) We need to downsample data in the *PerceptionObstacles* channel since the frequency of data is very high, often around 40 to 50 milliseconds. This generates many routing points, which can be unwieldy, especially for more extended scenarios. To address this, we downsample the data to reduce the number of routing waypoints for each obstacle, typically aiming for selecting a waypoint every 5

seconds. This filtering makes the scenario file more manageable and runnable and improves its readability. Algorithm 2 shows the overall process.

Algorithm 2 Obstacle Routing Trajectory Transformation

Input: $obstacles_{all}$, where $obstacles$ is of type $PerceptionObstacles$
$v \in AutowareVectorMap$, $o \in Object$, $ref = RoutingTrajectory_{[start,end]}$
Output: $RoutingEvents$

for $obstacles_{target}$ **in** $obstacles_{all}$ **do**
    p_{start} ←findStartLanePosition($obstacles_{target}$, v, o, ref)
    t_{start} ← First timestamp of $obstacles_{target}$
    $events$ ← [locateObstacle(p_{start}, t_{start}, e)]
    if isMovingObstacle($obstacles$) **then**
        t_s ← START_MOVING_TIME(obstacles)
        $p_{routing}$ ← []
        for i **in** selectIndices($obstacle$, 5) **do**
            p_i ←findLanePosition(obstacle[i].position, v, o, ref)
            $p_{routing}$.append(p_i)
        end for
        $events$.append(routingObstacle($p_{routing}$, t_s))
    end if
end for
return $events$

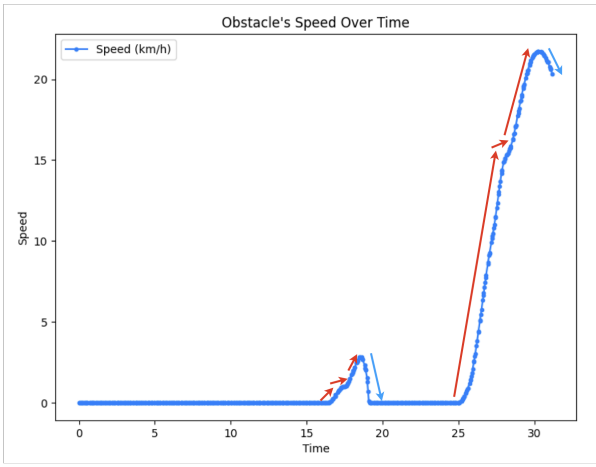
We determine the initial location of each obstacle and position it on the map using Algorithm 1. Upon the obstacle's initial appearance, we place it on the map based on its timestamp. Subsequently, if the obstacle is identified as a moving obstacle, we determine the timestamp at which it begins moving and initiate its routing accordingly. To establish the routing path, we select the obstacle's position every 5 seconds and designate these points as routing waypoints.

Speed Transformation

Obstacles on the road can dynamically adjust their speeds. The obstacle's movement transformation process needs to ensure that the converted scenario accurately preserves the information from the original scenario. Implementing a naive approach involves updating the speed of each obstacle for every timestamp, which can lead to system overload and unnec-

essary, redundant actions. To minimize the number of speed updates while retaining the original speed states of obstacles, we propose a method to calculate the acceleration and deceleration sections of obstacles by analyzing their speeds at each timestamp.

Figure 3.4 shows an example of the change in obstacle speed over time during a simulation. The obstacle repeatedly stops, speeds up, and slows down. By analyzing the slope of the obstacle's speed changes, we can identify acceleration, deceleration, and constant speed sections. We then apply this speed change information for each time interval to adjust the obstacle's speed accordingly. Algorithm 3 illustrates this approach, demonstrating how we can effectively manage obstacle speeds by identifying and handling acceleration and deceleration phases.



Time Interval(s)	Movement State	Speed (km/s)
0-16	Constant	0
16-17	↗ Increasing	2
17-17.5	↗ Increasing	2.3
17.5-18	↗ Increasing	3.5
18.5-19	↘ Decreasing	0
19-25	Constant	0
25-27	↗ Increasing	16
27-28	↗ Increasing	17
28-30	↗ Increasing	23
30-31	↘ Decreasing	20

Figure 3.4: Example of calculating velocity conversion for each obstacle

The *PerceptionObstacles* channel contains the linear velocity of each obstacle across timestamps. We traverse these velocities with two pointers, $speed_{prev}$ and $speed_{current}$, detecting changes surpassing a predefined threshold. For each such change, it discerns whether the obstacle accelerates, decelerates, or stays at the same speed, marking these transitions as state change points. These points are stored with range from the last changed index, $bound_{prev}$, to the current index, i. With this state change information, the algorithm aims to create

Algorithm 3 Obstacle Speed Transformation

Input: List of linear *velocities* where *velocities* is sorted in timestamp
Output: Speed States for each range, *states*

$threshold \leftarrow 0.01$; $speed_{\text{prev}} \leftarrow 0$; $speed_{\text{current}} \leftarrow 0$
$states \leftarrow []$
$bound_{prev} \leftarrow 0$
$State_{current} \leftarrow$ Constant
for i, v in enumerate(velocities) **do**
    if $i = 0$ **then**
        $speed_{\text{prev}} = v$
        continue
    end if
    $speed_{\text{current}} = v$
    $speed_{\text{diff}} = speed_{\text{current}} - speed_{\text{prev}}$
    if $|speed_{\text{diff}}| >$ threshold **then**
        if $speed_{\text{diff}} >$ threshold & $State_{current} \neq$ Increasing **then**
            $states$.append($(range(bound_{prev}, i)$, $State_{current})$)
            $bound_{prev} \leftarrow i$
        else
            if $State_{current} \neq Decreasing$ **then**
                $states$.append($(range(bound_{prev}, i)$, $State_{current})$)
                $bound_{prev} \leftarrow i$
            end if
        end if
    end if
    $speed_{\text{prev}} = speed_{\text{current}}$
end for
$states$.append($(range(bound_{prev}, len(velocities))$, $State_{current})$)
return *states*

a minimum number of speed change actions in the output scenario while conserving their dynamic speed change.

Limitation of Pedestrian Movement

In version 1.0 of Scenario Simulator v2, the simulator does not support pedestrians moving along a routing trajectory. The simulator only supports pedestrians moving in a straight line or adjusting their speed based on their initial direction. Consequently, the ADS Scenario Transformer does not support converting scenarios where pedestrians change direction and move in various ways, unlike vehicles such as cars and bicycles. The ADS Scenario Transformer identifies the initial direction of a pedestrian and checks if this direction changes by more than 10 degrees during the scenario. If the pedestrian's direction changes within 10 degrees, it is considered a straight-line movement. The pedestrian will then move in a straight line at a predefined speed or stop accordingly.

3.3.3 Status Changes of Traffic Signals

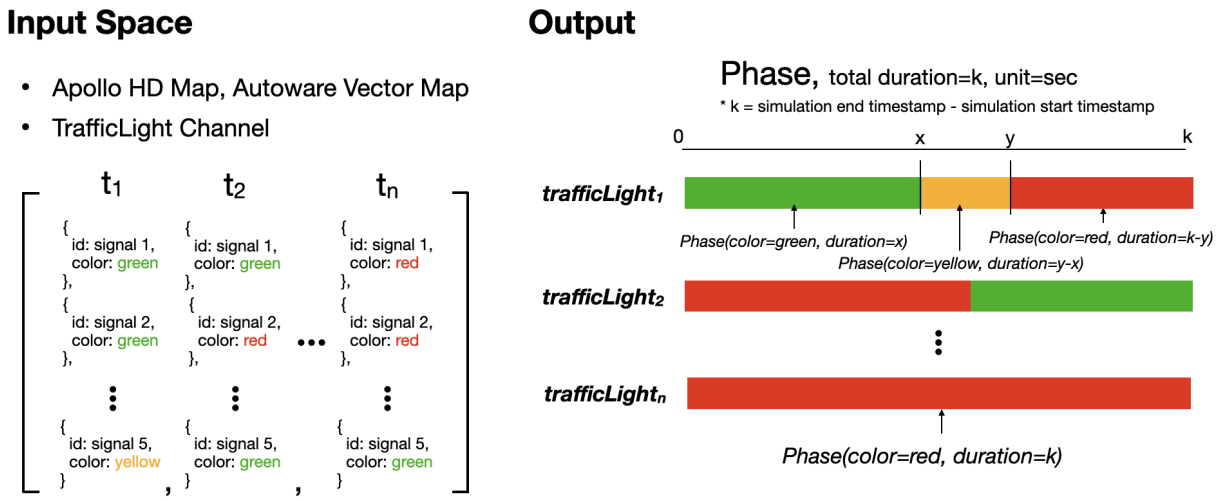


Figure 3.5: Input space and Output Models of Traffic Signal Transformation

In certain sections of lanes, traffic signals regulate the behavior of objects on the road. Failure to incorporate changes in traffic signal status from the original scenario into the transformed scenario can lead to significantly different outcomes. In Apollo scenarios, information about traffic signal status is stored in the $TrafficLight$ channel. ADS Scenario Transformer reads this channel and transfers changes in traffic signal status from the original scenario during simulation. Figure 3.5 illustrates the input and output of traffic signal data models. In Apollo scenarios, the $TrafficLight$ channel contains the status of each traffic signal at each timestamp. Conversely, in Autoware scenarios, each traffic light is defined within multiple Phase models, each specifying the color and duration of the traffic lights.

Algorithm 4 Traffic Signal Transformation

Input: Traffic signal identifier $\text{map}_{[T_{apollo}, T_{autoware}]}$, List of signal state, $States_{\text{signal}}$
Output: List of $Phase$, representing transformed traffic signals

$Phases \leftarrow []$
$\text{color}_{\text{prev}} \leftarrow \text{None}; t_{\text{prev}} \leftarrow \text{None}$
for $i, (s_{\text{current}}, t_{\text{current}})$ in $States_{\text{signal}}$ **do**
    if $\neg\text{color}_{\text{prev}}$ **then**
        $\text{color}_{\text{prev}} \leftarrow s_{\text{current}}.\text{color}; t_{\text{prev}} \leftarrow t_{\text{current}}$
        continue
    end if
    if $\text{color}_{\text{prev}} \neq s.\text{color}$ or i is last index **then**
        Phases.append(
            Phase(id = $\text{map}_{[T_{apollo}, T_{autoware}]}[s.\text{id}]$,
                color = $\text{color}_{\text{prev}}$,
                duration = $t_{\text{current}} - t_{\text{prev}}$)
)
        $\text{color}_{\text{prev}} \leftarrow s_{\text{current}}.\text{color}; t_{\text{prev}} \leftarrow t_{\text{current}}$
    end if
end for
return $Phases$

To transform traffic signals, we need to establish correspondence between traffic signals in the Apollo and Autoware scenarios. This association is achieved by comparing the coordinate values of each signal in both maps: the Apollo HD map and the Autoware Vector map. ADS Scenario Transformer converts the coordinates of traffic signals in the Apollo map and identifies the nearest traffic signals in the Autoware Vector map. Once the mapping of

identifiers for each traffic signal is completed, we set the changes in traffic lights throughout the original scenario by reading the $TrafficLight$ channel. Subsequently, the state of each traffic light is set in each timestamp of the simulation using the Phases model. Algorithm 4 illustrates this process.

3.4 Embedding Routing Trajectory of the Ego Car

Nevertheless, since both the Apollo and Autoware scenarios aim to test the behavior of the ego car and the ego car's driving plays a crucial role in scenario playback, we need to define the ego car's routing trajectory within the scenario to run it and verify whether it meets its goals. Therefore, the ADS Scenario Transformer uses the routing trajectory of the ego car in the Apollo scenario to define the start and end of the scenario. In other words, the scenario begins by placing the ego car at the starting point of the routing trajectory, and the scenario ends once the ego car reaches the endpoint of this trajectory.

To achieve this, the ADS Scenario Transformer reads the routing trajectory of the ego car in the Apollo scenario and constructs a routing trajectory in the Autoware scenario. In the Apollo scenario, *RoutingRequest* channel stores the routing trajectory of the ego car, containing waypoints that the ego car should visit. Thus, we need to read the data from the *RoutingRequest* channel and reflect it into a format that works in the Autoware scenario. Algorithm 5 demonstrates how this process is performed. Most of the work to identify the position of each waypoint is handled by Algorithm 1.

Moreover, we leverage the *Localization* channel in the Apollo scenario to supplement the routing trajectory of the ego car in cases where the *RoutingRequest* fails to accurately represent the trajectory in the scenario or when the *RoutingRequest* specifies an unreachable route within the Autoware Vector map. For instance, specific scenarios finish prematurely

Algorithm 5 Embedding Routing Trajectory

Input: $r \in RoutingRequest$ $v \in AutowareVectorMap$ $o \in Object$
$ref = RoutingTrajectory_{[start,end]}$
Output: List of p_l, where $p_l \in LanePosition$

```
positions ← []
for waypoint in r.waypoints do
    positions.append(findLanePosition(waypoint.point, v, o, ref))
end for
return positions
```

before reaching the final waypoint of the routing trajectory. In such cases, setting the success condition of the scenario to reach the last point of the routing trajectory would invariably result in scenario failure. To address this issue, our transformer supports generating a routing trajectory based on the positional data available in the *Localization* channel. The *Localization* channel contains precise information about the ego car's location at each time stamp, enabling us to determine the location of the final waypoint in the scenarios and ensuring that the final point in the *Localization* channel is reachable.

Chapter 4

Implementation

ADS Scenario Transformer is built on Python 3 and is designed to work independently of Apollo and Autoware. Upon receiving input, the top-level $ScenarioTransformer$ manages all transformation processes, executes channel transformers based on user arguments, and retrieves all results to output Autoware scenarios. The output scenario format is TIER IV Scenario Format v2, an extended version of OpenSCENARIO v1.2. We define OpenSCENARIO messages using protocol buffers and generate them as outputs for each transformer.

Also, the OpenSCENARIO specification outlines the conditions for creating OpenSCENARIO messages, so we define builders to ensure compliance during their creation. However, the output of the top-level scenario object is not directly runnable, even if they are converted to a .xosc format file. This is because the naming convention for defining OpenSCENARIO protocol buffers differs from the OpenSCENARIO XML schema. To address this, we encode the scenario object after creation, changing the key and value of compatible fields, resulting in a scenario file that is runnable in the Autoware-Scenario Simulator environment.

Chapter 5

Evaluation

We explore the following research questions to empirically evaluate the ADS Scenario Transformer:

- RQ1: How well does the transformed scenario preserve information from the original scenario?
- RQ2: How do the ego car behaviors compare in the original and transformed scenarios?

We evaluated the ADS Scenario Transformer by manually transforming and analyzing 13 scenarios to address this research question. Additionally, during this investigation, we identified cases where Apollo and Autoware's ego cars behave differently in similar environments, and we examined these differences.

5.1 Experiment Setup

Our evaluation includes the process of converting Apollo scenarios to Autoware scenarios. We selected Apollo scenarios that can be executed in the Apollo 9.0.0 and SimControl simulator environments. The transformed Autoware scenarios are created to be executable in Autoware 1.0 and Scenario Simulator v2 1.0. We randomly selected 13 Apollo scenarios for transformation from the scenarios generated by DoppelTest [11] and scenoRITA [10]. They are state-of-the-art tools designed to create Apollo scenarios that enhance the safety of an ADS, offering diversity in scenarios, detecting various violations, and providing dynamic behavior of obstacles within the scenarios.

Additionally, the scenarios are based on Borregas Avenue, a road located in Sunnyvale City, using the Apollo HD Map and the Autoware Vector Map. The Apollo HD Map of Borregas Avenue is based on a publicly available map. However, for the Autoware Vector Map, only a lanelet2 format map was available. Therefore, we converted this lanelet2 format map to the Autoware Vector Map format using the Vector Map Builder tool [20]. We manually added information not existing in the original lanelet2 map, such as the lane's speed limits, to match the values from the Apollo HD Map of Borregas Avenue.

5.2 RQ1: Fidelity of Transformed Scenarios

We investigated how well the ADS Scenario Transformer converts an input Apollo Scenario to an Autoware Scenario by focusing on two main aspects: (1) whether the movements of obstacles are similar in the transferred scenario to the original obstacles and (2) whether the traffic light information is similar in the transferred scenario compared to the original. We did not verify the ego car's behavior similarity when comparing the original and transformed scenarios as the ADSes running the ego car, i.e., Apollo or Autoware, are the systems under

test after test scenarios have been transferred. To evaluate this, we transformed 13 scenarios and manually compared each in a simulator to ensure that the ADS Scenario Transformer effectively preserves the information during the transformation process.

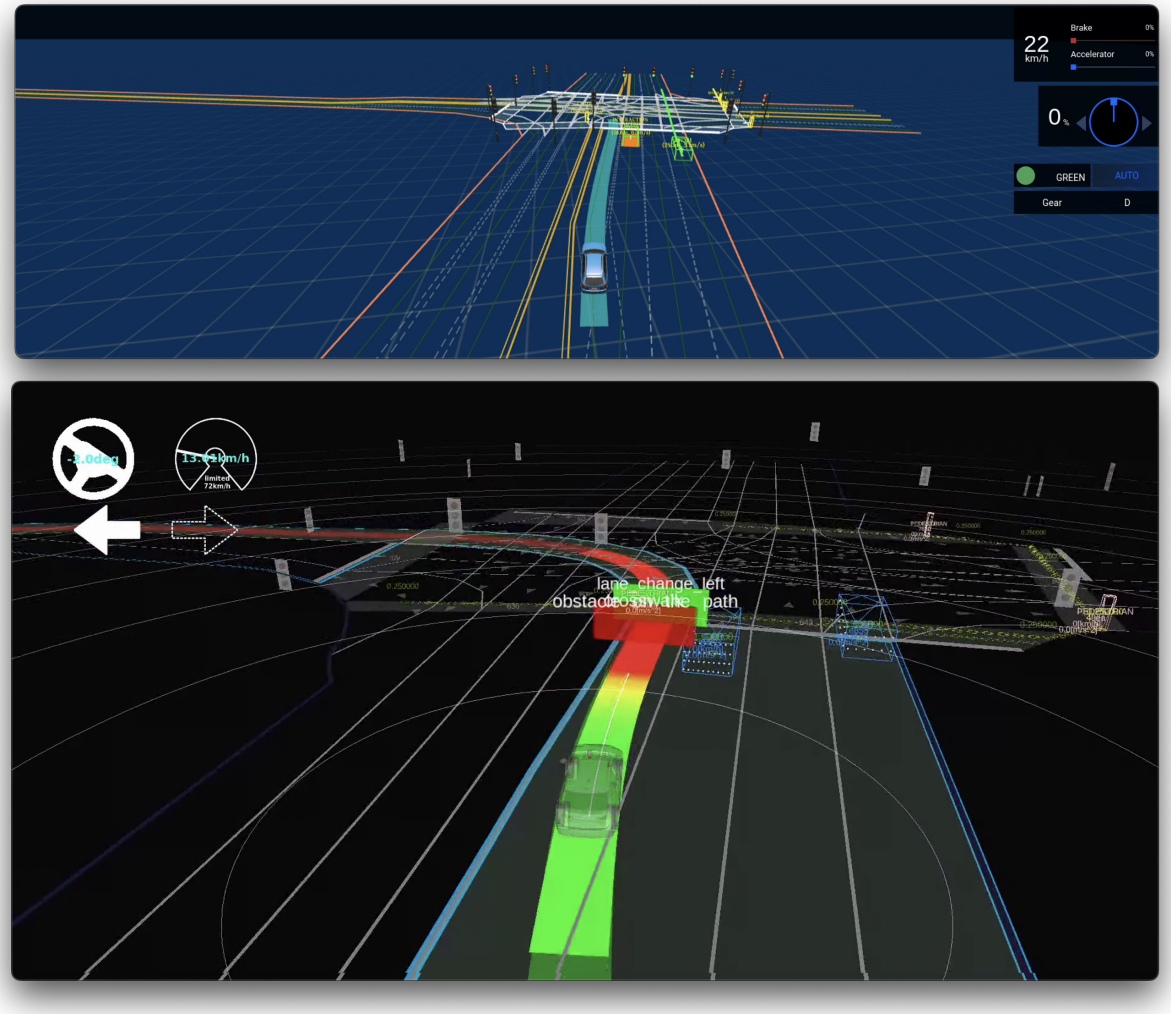
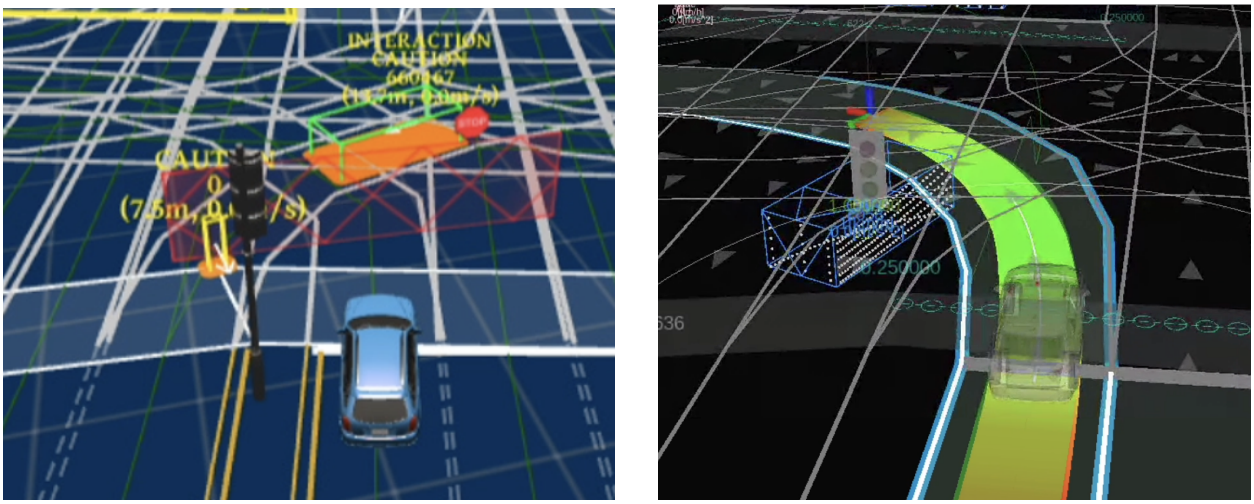


Figure 5.1: Comparison of results shown by the transformed scenario and the original scenario on the simulator. The upper image shows the original Apollo scenario; the bottom image shows the transformed Autoware scenario.

In Figure 5.1, you can observe the overall output of the transformed scenario. We found that most transformed objects are positioned similarly to where the objects were placed in the original scenario. The results of investigating the transformation in obstacle movement and traffic signal changes are detailed in the folowing paragraphs.

Vehicle Behavior: The driving paths of vehicles and the speeds of obstacles in the transferred scenarios closely resemble those in the original scenarios. However, we observed minor variations in vehicle speeds that influenced the outcomes of the scenarios. For example, in the original scenario shown in Figure 5.2, a vehicle turning left at a junction stops and blocks the ego car's path. In the corresponding transformed scenario, the vehicle moves in the same direction but does not entirely block the ego car's path, allowing it to reach its destination. This difference arises because the vehicle does not stop at the same place in the junction, and this issue is caused by the difference in the vehicle's speeds between the original and transformed scenarios.



(a) Original Scenario (b) Transformed Scenario

Figure 5.2: A case where incomplete vehicle speed control alters the behavior of the ego car. In the original scenario, the obstacle stops at the routing path of the ego car, but it stops after passing the routing path in the transformed scenario.

Pedestrian Behavior: In the context of pedestrian behavior, we identified cases where pedestrians moved in the original scenario but remained stationary in the transformed scenario. This inconsistency arises for two reasons. First, we filter out scenarios where pedestrians change their direction while moving, but some still have them. To determine if a pedestrian changes direction, we check if the heading changes by more than 10 degrees from the initial heading. However, this method still allows for cases where pedestrian movement

is not straight, and the Autoware-Scenario Simulator environment does not support these non-linear movements. Second, pedestrians cannot move beyond the lane they start in, causing them to stop at the edge of the lane even if they have a non-zero speed and a valid heading. One potential solution to both issues is to respawn pedestrians when they change direction and attempt to move beyond their original lane. This approach requires more accurate estimates of pedestrian movements, and it is necessary to determine whether the scenario simulator is powerful enough to apply this approach.

Traffic Signals: All 13 transferred scenarios are investigated to retain the traffic signal change information from the original scenarios. Not all original scenarios contain traffic signal information. Therefore, the ADS Scenario Transformer has been observed to appropriately apply traffic signal information when it exists in the scenario while refraining from applying traffic signal information when it is absent.

5.3 RQ2: Ego Car Behaviors in Original and Transformed Scenarios

We investigated how Apollo and Autoware behave differently in similar environments, comparing the original Apollo scenario and transformed Autoware scenarios. We identified three main differences in their behaviors.

Difference in Obstacle Prediction and Stop Decision: In the Apollo scenario, the ego car makes decisions earlier than Autoware's ego car by predicting the movement of obstacles. Autoware's ego car checks for obstacles when entering a junction or passing a crosswalk, determining if it can drive safely. Once it decides it is safe, the ego car continues moving until it detects an obstacle directly in its path. Figure 5.3 illustrates this situation in detail. In the Apollo scenario, the ego car predicts obstacle movements and yields when

it is dangerous to move forward. In contrast, in the Autoware scenario, the ego car does not consider the future movement of an obstacle until it is directly in its driving path. Consequently, it does not slow down even if the obstacle is about to move into its path.

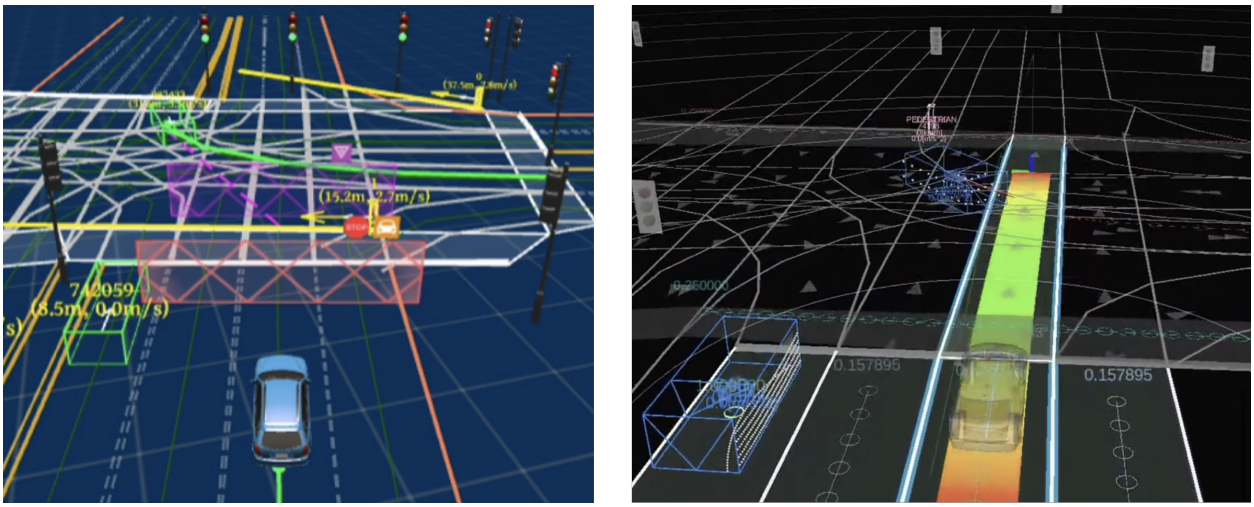


Figure 5.3: Comparison of obstacle movement predictions by ego cars: In the Apollo scenario (left), the ego car predicts obstacles that are expected to enter its driving path and yields to those obstacles to move safely, as indicated by the purple fence. In contrast, in the Autoware scenario (right), the ego car does not yield to or stop for an obstacle unless that obstacle is exactly in its driving path.

Speed and Acceleration: In the Apollo scenario, the ego car accelerates relatively faster than in the Autoware scenario. The speed of the ego car varies based on its position and the lane's speed limit in both scenarios. However, when the ego car encounters identical situations and speed limits in both scenarios, the Apollo ego car exhibits relatively more rapid acceleration compared to the Autoware ego car. This acceleration difference can also influence the outcome of the scenario. Figure 5.4 illustrates the ego car attempting to change lanes with identical speed limits in each scenario. In the Apollo scenario, the ego car accelerates rapidly, completing the lane change in approximately 10 seconds and overtaking the stationary obstacle. Conversely, in the Autoware scenario, the ego car accelerates more slowly, taking about 15 seconds to change lanes and failing to overtake the obstacle.

Stopping Behavior in Junctions: We observed that Autoware's ego car independently processes traffic signal recognition and pedestrian recognition at junctions, whereas Apollo's

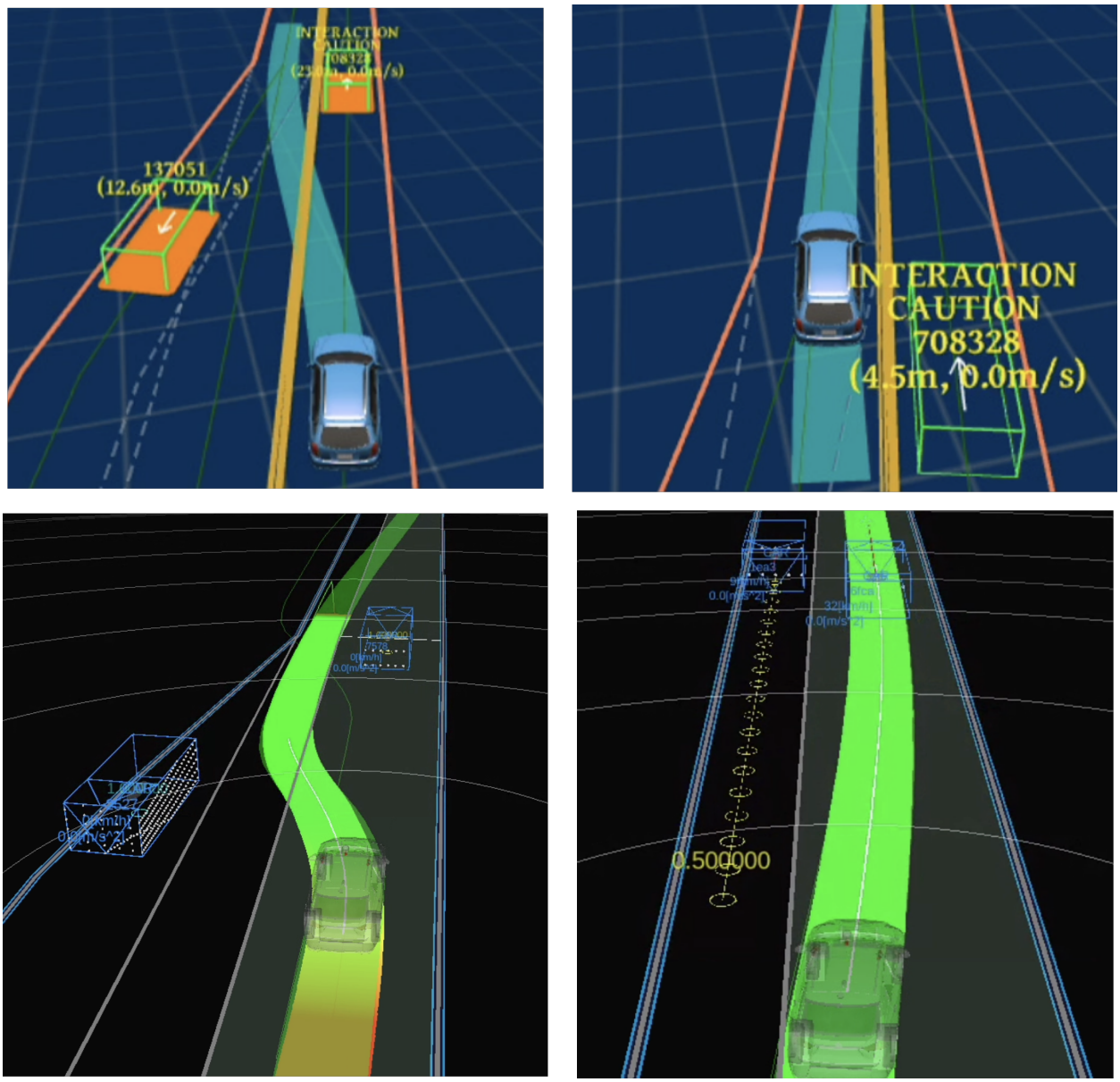


Figure 5.4: A case where the result of the scenario changes due to differences in ADS speed control. In the Apollo scenario, the ego car successfully overtakes a stationary vehicle. Conversely, in the Autoware scenario, the ego car changes lanes slowly, allowing the vehicle on the road to go first, thereby preventing the ego car from completing the overtaking maneuver.

ego car integrates these tasks. Autoware's ego car consistently stops at crosswalks when pedestrians are present. However, if it can safely pass without endangering pedestrians, Apollo's ego car proceeds through crosswalks without stopping. Figure 5.5 illustrates the different behaviors of two ADSes. In both cases, the ego car is exiting the junction on a green

light while pedestrians are on the crosswalk. Apollo predicts the pedestrians' movements and decides whether it can proceed safely, not stopping if there are no issues. On the other hand, Autoware stops in front of the crosswalk whenever there are pedestrians, even though it is not dangerous for pedestrians for the ego car to pass through the crosswalk.

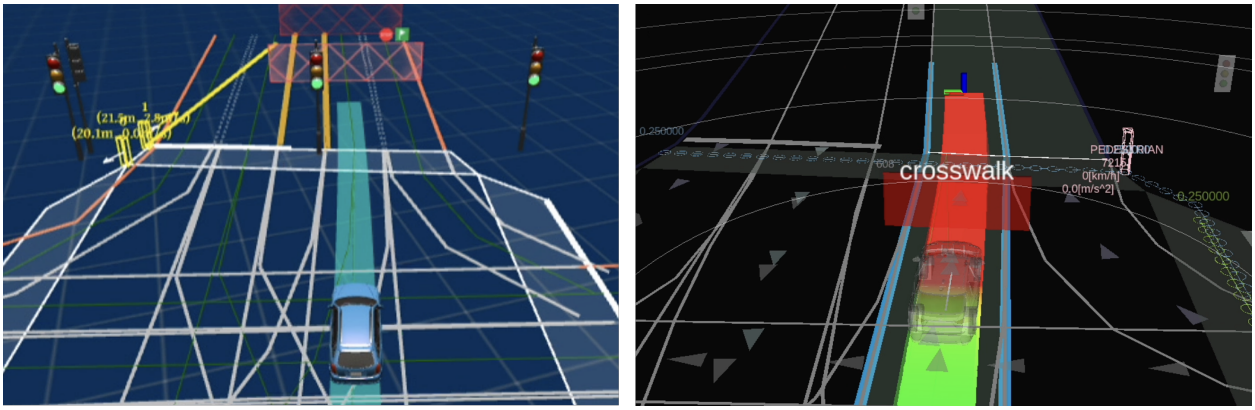


Figure 5.5: Comparison between Apollo and Autoware ego car's movement in exiting junction if a pedestrian is on the crosswalk. In the Apollo scenario, the ego car does not stop if pedestrians are not in danger due to the car's movement in green light. However, in the Autoware scenario, the ego car always stops in front of the crosswalk if a pedestrian is present, regardless of their movement and traffic light states.

Chapter 6

Future Work

In order to enhance the robustness of scenario transferring and evaluate the output of the ADS Scenario Transformer, we can consider the following future works:

Employ Record-Based Analysis: Transformations are evaluated by manually comparing the original and transformed scenarios. This process can be improved by adopting a record-based analysis. By recording both scenarios, we can track the movement of each obstacle through these records. This approach lets us directly compare each obstacle's routing trajectory and speed at each timestamp, enabling a quantitative evaluation of the transformer.

Validate Traffic Signal Phases: The original scenario input includes both scenario files and traffic signal phases, indicating which color each traffic signal should display at specific timestamps. We can use this information as an oracle to validate the traffic signal phases in the transformed scenario. This validation ensures that the traffic signal transformations accurately preserve the original timing and color changes.

Improve Lane Path Finding Approach: Finding the correct lane path for each object is one of the significant challenges of the ADS Scenario Transformer. In addition to defining

a more precise algorithm that uses additional information beyond the routing trajectory, we can consider another approach. This approach involves constructing all possible trajectories based on candidate lanes and then selecting the correct trajectory by determining if it is navigable by the object and comparing its similarity to the object's trajectory in the original scenario. Specifically, when we need to find a lane at a particular point, we explore all lanes and construct paths from these lanes. We first determine if these lane paths are navigable on the routing graph. Then, we can identify the correct lane paths by comparing the similarity between the object's actual movement and the trajectory formed by these lane paths.

Chapter 7

Conclusion

We present an ADS Scenario Transformer that enables scenarios running on one ADS simulator to be transferred to others. Our research reproduces the same scenario in different ADS simulator environments, allowing a single scenario to be executed across various ADS simulators. We evaluated the ADS Scenario Transformer by manually transforming and analyzing 13 scenarios. We investigated whether the movement of obstacles and changes in traffic signals appeared similarly in the generated scenarios. Additionally, we identified cases where Apollo and Autoware's ego cars behaved differently in similar environments and examined these differences. For future work, we suggest employing record-based analysis, validating traffic signal phases, and improving lane path-finding approaches to enhance the robustness of the transformations and evaluations.

Bibliography

[1] Association for Standardization of Automation and Measuring Systems (ASAM). 8.3 Road reference line coordinate systems :: OpenDRIVE®.

[2] Association for Standardization of Automation and Measuring Systems (ASAM). OpenDRIVE: Road Networks for Vehicle Simulation.

[3] Association for Standardization of Automation and Measuring Systems (ASAM). OpenSCENARIO V1.2 Standard.

[4] Baidu. Baidu Apollo-Autonomous Driving, solutions for intelligent vehicles.

[5] F. Batsch, S. Kanarachos, M. Cheah, R. Ponticelli, and M. Blundell. A taxonomy of validation strategies to ensure the safe operation of highly automated vehicles. *Journal of Intelligent Transportation Systems*, 26(1):14–33, Jan. 2022.

[6] California DMV. Autonomous Vehicle Collision Reports.

[7] California DMV. Autonomous Vehicle Permit Holders Report a Record 9 Million Test Miles in California in 12 Months.

[8] J. Dai, B. Gao, M. Luo, Z. Huang, Z. Li, Y. Zhang, and M. Yang. SCTrans: Constructing a Large Public Scenario Dataset for Simulation Testing of Autonomous Driving Systems. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, pages 1–13, New York, NY, USA, Feb. 2024. Association for Computing Machinery.

[9] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun. CARLA: An open urban driving simulator. In S. Levine, V. Vanhoucke, and K. Goldberg, editors, *Proceedings of the 1st Annual Conference on Robot Learning*, volume 78 of *Proceedings of Machine Learning Research*, pages 1–16. PMLR, 13–15 Nov 2017.

[10] Y. Huai, S. Almanee, Y. Chen, X. Wu, Q. A. Chen, and J. Garcia. scenoRITA: Generating Diverse, Fully Mutable, Test Scenarios for Autonomous Vehicle Planning. *IEEE Transactions on Software Engineering*, 49(10):4656–4676, Oct. 2023.

[11] Y. Huai, Y. Chen, S. Almanee, T. Ngo, X. Liao, Z. Wan, Q. A. Chen, and J. Garcia. Doppelgänger Test Generation for Revealing Bugs in Autonomous Driving Software. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2591–2603, May 2023. ISSN: 1558-1225.

[12] S. Kim, M. Liu, J. J. Rhee, Y. Jeon, Y. Kwon, and C. H. Kim. DriveFuzz: Discovering Autonomous Driving Bugs through Driving Quality-Guided Fuzzing. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, pages 1753–1767, New York, NY, USA, Nov. 2022. Association for Computing Machinery.

[13] G. Lou, Y. Deng, X. Zheng, M. Zhang, and T. Zhang. Testing of autonomous driving systems: where are we and where should we go? In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 31–43, 2022.

[14] F. Poggenhans, J.-H. Pauls, J. Janosovits, S. Orf, M. Naumann, F. Kuhnt, and M. Mayr. Lanelet2: A high-definition map framework for the future of automated driving. In *2018 21st International Conference on Intelligent Transportation Systems (ITSC)*, pages 1672–1679, Nov. 2018. ISSN: 2153-0017.

[15] G. Rong, B. H. Shin, H. Tabatabaee, Q. Lu, S. Lemke, M. Možeiko, E. Boise, G. Uhm, M. Gerow, S. Mehta, et al. Lgsvl simulator: A high fidelity simulator for autonomous driving. In *2020 IEEE 23rd International conference on intelligent transportation systems (ITSC)*, pages 1–6. IEEE, 2020.

[16] Q. Song, E. Engström, and P. Runeson. Industry practices for challenging autonomous driving systems with critical scenarios. *ACM Transactions on Software Engineering and Methodology*, 2023.

[17] The Autoware Foundation. Autoware.

[18] TIER IV, Inc. AWSIM.

[19] TIER IV, Inc. Scenario testing framework for Autoware.

[20] TIER IV, Inc. Vector Map Builder - Autoware Tools.

[21] S. Ulbrich, T. Menzel, A. Reschka, F. Schuldt, and M. Maurer. Defining and Substantiating the Terms Scene, Situation, and Scenario for Automated Driving. In *2015 IEEE 18th International Conference on Intelligent Transportation Systems*, pages 982–988, Sept. 2015. ISSN: 2153-0017.

[22] Waymo. Waymo safety report - 2021. `https://waymo.community/resources/waymo-safety-report-2021.html`, 2021.

[23] Z. Zhong, G. Kaiser, and B. Ray. Neural Network Guided Evolutionary Fuzzing for Finding Traffic Violations of Autonomous Vehicles, July 2022. arXiv:2109.06126 [cs].