3-2023

Specification-based autonomous driving system testing

Yuan ZHOU

Yang SUN
Singapore Management University, yangsun.2020@phdcs.smu.edu.sg

Yun TANG

Yuqi CHEN

Jun SUN
Singapore Management University, junsun@smu.edu.sg

See next page for additional authors

Specification-based Autonomous Driving System Testing

Yuan Zhou, Yang Sun, Yun Tang, Yuqi Chen, Jun Sun, Christopher M. Poskitt, Yang Liu, and Zijiang Yang

Abstract—Autonomous vehicle (AV) systems must be comprehensively tested and evaluated before they can be deployed. High-fidelity simulators such as CARLA or LGSVL allow this to be done safely in very realistic and highly customizable environments. Existing testing approaches, however, fail to test simulated AVs systematically, as they focus on specific scenarios and oracles (e.g., lane following scenario with the "no collision" requirement) and lack any coverage criteria measures. In this paper, we propose `AVUnit`, a framework for systematically testing AV systems against customizable correctness specifications. Designed modularly to support different simulators, `AVUnit` consists of two new languages for specifying dynamic properties of scenes (e.g. changing pedestrian behaviour after waypoints) and fine-grained assertions about the AV's journey. `AVUnit` further supports multiple fuzzing algorithms that automatically search for test cases that violate these assertions, using robustness and coverage measures as fitness metrics. We evaluated the implementation of `AVUnit` for the LGSVL+Apollo simulation environment, finding 19 kinds of issues in Apollo, which indicate that the open-source Apollo does not perform well in complex intersections and lane-changing related scenarios.

Index Terms—Autonomous Driving System, Testing, Specification Languages, Fuzzing, Coverage Criteria.

1 Introduction

Autonomous vehicles (AVs) are set to play an essential role in relieving traffic congestion and reducing accidents in intelligent transportation systems. AVs are controlled by autonomous driving systems (ADSs), which take over various driving tasks from human drivers by interpreting and acting on sensory information (e.g. lidar data). Before AVs can be fully deployed on public roads, however, it is imperative that their ADSs are comprehensively tested to ensure they will *always* behave correctly and safely, even in rare scenarios where pedestrians and human-controlled vehicles behave unpredictably.

The state of the art in AV testing can be broadly divided into two categories: *real-world testing* and *simulator-based testing*. Extensive and rigorous real-world road testing for AVs is necessary and often used as a measure of the technology's progress, e.g. the number of kilometres covered without human intervention. However, it is also costly, and most importantly, insufficient to cover all the situations that AVs must be able to safely react to (e.g. accident scenarios). This motivates the need for simulator-based testing, where developers and safety engineers can systematically assess ADSs against a broader range of scenarios that may occur on the road. These scenarios can be designed at different levels, from the most abstract to the least: *functional*, in which roads and traffic agents, including pedestrians, non-player character (NPC) vehicles (vehicles not controlled by ADSs) and static obstacles are described using a linguistic notation; *logical*, in which parameters and their ranges (e.g. positions) are specified; and *concrete*, in which concrete values for every parameter are given [1]. Test cases are then executable concrete scenarios together with *oracles*, i.e. 'pass/fail' criteria, that the behaviour of the ADSs must satisfy [2].

For simulation-based testing, researchers and engineers rely on high-fidelity simulators such as LGSVL [3] and CARLA [4], which can simulate not only sensors and vehicle dynamics but also controlled traffic flows. While equipped with APIs for configuring scenarios, these simulators differ significantly and are not specifically designed to support systematic testing and analysis. This has motivated the development of tool-independent domain-specific languages (DSLs) that can be used to specify AV scenarios, e.g. [5]–[11]. For example, OpenScenario [6] provides a standard for AV scenarios and an XML format to model scenarios, where the motion of an agent is described by actions and triggers; the triggers are related to other agents and active actions. GeoScenario [7] is designed based on OpenScenario, and the behaviour of a dynamic agent is given by its position and speed profiles, together with reactive triggers. Scenic [8] was first designed for describing different scenes and later extended to dynamic scenarios using simulator-specified actions, which may rely on other traffic participants on the road. Most of the existing scenario description languages primarily focus on depicting complex scenarios involving interactions among different traffic participants, rather than automatically testing ADSs. For example, while some companies that work on ADSs (e.g., Desay, Alibaba, and NIO) could create scenarios using OpenScenario, they

- *Y. Zhou, Y. Tang, and Y. Liu are with the School of Computer Science and Engineering, Nanyang Technology University, Singapore, 639798. E-mail: y.zhou@ntu.eud.sg, yun005@e.ntu.edu.sg, yangliu@ntu.edu.sg*
- *Y. Sun, J. Sun, C.M. Poskitt are with the School of Computing and Information Systems, Singapore Management University, Singapore, 178902. E-mail: yangsun.2020@phdcs.smu.edu.sg, junsun@smu.edu.sg, cposkitt@smu.edu.sg.*
- *Y Chen is with ShanghaiTech University, Shanghai, 201210, China. E-mail: chenyq@shanghaitech.edu.cn*
- *Z. Yang is with Xi'an Jiaotong University, Xi'an, Shaanxi, 710049, China, and also with GuardStrike Inc., Xi'an, Shaanxi, 710074, China. E-mail: zijiang@xjtu.edu.cn.*

often struggle to generate new scenarios automatically due to uncertainty about the appropriate elements that can be altered. Consequently, there is a requirement for a platform that integrates with a scenario description language to facilitate automatic testing of ADSs.

An additional problem is that the aforementioned languages do not support customizing correctness specifications, i.e. the conditions that ADSs must satisfy when navigating a scenario, and automatic test case generation. Various critical scenario generation methods have been proposed based on real-world traffic data/reports [12]–[20] or ADS system models [21]–[25]. In terms of correctness, all of these methods focus on fixed oracles, e.g. "no collisions". However, users, especially ADS developers, may want to test various correctness conditions based on their own requirements, such as the local driving laws (which might vary from country-to-country), and even conditions related to passenger comfort (e.g. by avoiding sudden braking). While there are some specification languages based on signal temporal logic (STL) [26], [27] for describing conditions with respect to certain ADS modules (e.g. perception), to the best of our knowledge, no tool-independent language currently exists for individually expressing various correctness specifications that ADSs should satisfy.

To address these requirements and support automatic ADS testing, we present `AVUnit`, a comprehensive framework for systematical ADS testing. `AVUnit` features rich new DSLs—`SCENEST` and `AVSpec`—for respectively specifying dynamic scenarios and AV correctness conditions, as well as an approach that searches for specification-violating test cases automatically. First, `SCENEST` is an agent-oriented language and specifies a scenario in terms of different elements, such as time, weather, and traffic agents, as well as their concrete behaviours in terms of *waypoints*. It is specifically designed for testing purposes and models each agent independently, which can simplify the initialization of scenarios by following the given template. Second, `AVSpec` uses STL formulas to describe different correctness specifications. The expression of each specification relies only on `SCENEST`, rather than the specific outputs of an ADS, ensuring that `AVSpec` is ADS-independent and can be applied to many different systems. Finally, we propose two feedback-guided fuzzing algorithms: fuzzing for failures and fuzzing for coverage. In fuzzing for failures, a genetic algorithm (GA) is applied to guide the generation of scenarios that violate a given specification. In fuzzing for coverage, we propose specification-based coverage to evaluate testing sufficiency. Compared with the state-of-the-art DSLs, the main unique features of `AVUnit` are that (1) it proposes an agent-oriented language for scenario description and a general way to customize specifications, and (2) it is equipped with fuzzing algorithms to generate critical scenarios automatically.

The implementation of `AVUnit` consists of (1) a parser, which uses antlr4 to extract the elements describing a scenario in the `SCENEST` script and the corresponding specifications; (2) a simulator adapter, which spawn the scenario to a simulator; (3) a fuzzing engine, which implements the feedback-guided fuzzing algorithms; and (4) a middleware, which collects execution data from the ADS. Note that the simulator adapter and the middleware components are customized based on the specific simulator and ADS used, respectively. However, developing the adapter and middleware is a distinct process, and users of `AVUnit` can concentrate only on the testing itself. We evaluate the implementation on the LGSVL simulator [3] and Baidu Apollo ADS [28]. We conduct a comprehensive experiment on Apollo and discover 19 issues, which can be found at https://avunit2021.github.io/.

The main contributions of this paper are as follows.

- We design an agent-oriented DSL for describing ADS testing scenarios. It models the agents in a scenario in a decoupled way following a certain template.
- We design an ADS-independent language to describe fine-grained correctness specifications for AVs.
- We propose two feedback-guided fuzzing algorithms to test ADSs based on the described scenarios and specifications. The first is designed to generate scenarios that violate the defined specifications, and the second one is designed to maximize a test coverage criterion that is defined based on different ways of violating a specification.
- We implemented our framework for LGSVL+Apollo, finding that Apollo does not perform well in complex intersections and lane changing (including overtaking) and is vulnerable to attacks.

2 AVUnit: Background and Overview

Background. AVs are complex systems consisting of various electronic control units (ECUs) and sensors. As the brain of an AV, the ADS reads the data from sensors and generates commands to different ECUs. ADSs are complex software systems containing deterministic components (e.g., conventional motion planning and control algorithms) and non-deterministic components (e.g., deep learning models). An ADS usually consists of the following modules: localization, perception, motion planning, and control. The localization module takes the data from related sensors (e.g., GPS and IMU) as input and locates the position of the AV in the map. The perception module is used to process the data from cameras, LiDAR, and Radar and recognize the surrounding objects (e.g., lane boundaries, traffic signs, traffic lights, other vehicles, and pedestrians) via some deep learning models (e.g., RNNs and CNNs). The motion planning module is to generate collision-free trajectory. It contains global and local motion planning. The global motion planning generates a path offline from the initial position to the target position based on the map, without considering the real-time environment. The local motion planning aims to generate a local collision-free trajectory based on the offline path and the real-time traffic conditions. Finally, the control module generates proper control commands (i.e., steering, throttle, brake) to the chassis.

ADSs play an essential role to guarantee the correctness of AVs' motion. However, ADS testing is still challenging since (1) the structure of an ADS is complex, containing not only traditional software components but also deep learning components, (2) the development of ADSs rely on various aspects, such as software engineering, deep learning, robotics, and control theory, (3) the number of parameters forming a test case is large and non-deterministic due to the open environments, and (4) a test case should specify not only an initial scene but also a detailed scenario, which

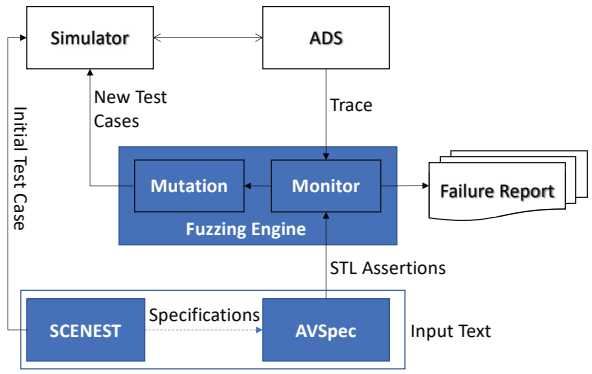

Fig. 1: The architecture of AVUnit.

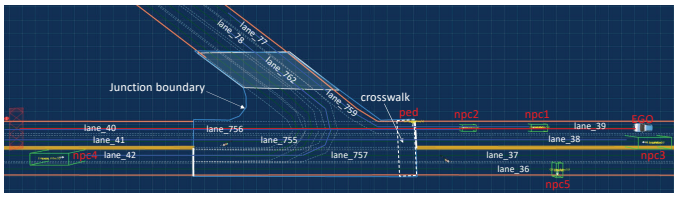

Fig. 2: The initial scene generated by our test script in Apollo Dreamview. The blue lines are the paths for $npc1$–$npc4$.

consists of a sequence of scenes and their transitions; writing such a scenario sometime is overwhelming since different agents have different behaviours.

AVUnit. Hence, to facilitate ADS testing, we propose AVUnit, a script language specifically designed for ADS testing, which can describe not only various scenarios but also different correctness properties. The architecture is shown in Fig. 1. It has three main components, i.e., SCENEST, a script for describing scenes and scenarios (Section 3), AVSpec, an assertion language for specifying correctness (Section 4), and a set of annotations for instructing the fuzzing algorithms (Section 5). In detail, SCENEST is an agent-oriented language to model a scenario for testing. It describes different aspects in a test case, such as the map, the weather conditions, the motion tasks for the ego vehicle, the trajectories for pedestrians and NPC vehicles. AVSpec is an STL-based language that users can customize different specifications in the form of assertions. AVUnit is designed to evaluate ADSs utilised in L4 and L5 autonomous vehicles. However, it can also be applied to test ADASs (Advanced Driver Assistance Systems) utilised in L2 and L3 vehicles, as long as no human control or actions are required during the execution of the ADAS under test. Moreover, AVUnit can be used not only by developers and testers of ADSs but also any interested stakeholders with little knowledge of autonomous driving (e.g., junior researchers).

Motivating Example. Fig. 2 shows an initial scene of a scenario that is generated based on the SCENEST script shown in Listing 1, i.e., the input of a test case. In this example, there are five NPC vehicles and one pedestrian. To describe the concrete scenario, we need to describe the motion task of the 'ego vehicle' (the vehicle under test), the motion behaviour of $npc1$–$npc5$ and the pedestrian ped, which are given as follows.

```
map_name = "san_francisco";

//Ego vehicle
car_model = "Lincoln MKZ 2017";
vehicle_type = (car_model);
ego_init_position = ENU (553078.1, 4182687.8);
ego_init_state = (ego_init_position);
ego_target_position = "lane_40"->50;
ego_target_state = (ego_target_position);
ego_vehicle = AV(ego_init_state, ego_target_state,
    vehicle_type);

//Describe vehicle motion by waypoints
npc1_model = "Sedan";
npc1_heading = 0 deg related to ego_vehicle;
npc1_init_position = IMU (30, 0);
npc1_init_state = (npc1_init_position, npc1_heading,
     6.0);
npc1_wps = (("lane_759"->1, ,2), ("lane_77"->10, ,6)
    );
npc1_destination = ("lane_77"->100);
npc1 = Vehicle(npc1_init_state, Waypoint(npc1_wps),
    npc1_destination, (npc1_model));

// Describe vehicle motion by uniform motion
npc2_model = "Jeep";
npc2_init_position = ENU (ego_init_position - (50,
    0));
npc2_init_state = (npc2_init_position, ,6.0);
npc2 = Vehicle(npc2_init_state, Uniform(
    npc2_init_state), , (npc2_model));

// Describe vehicle motion by default uniform motion
npc3_model = "SchoolBus";
npc3_init_state = ("lane_38"->10.0, ,5.8);
npc3 =Vehicle(npc3_init_state, , , (npc3_model));

// Describe vehicle motion by initial and target
    positions.
npc4_model = "BoxTruck";
npc4_heading = 180 deg related to EGO;
npc4_init_state = ("lane_42"->50.0, ,8.0);
npc4_destination = ("lane_78"->100);
npc4 =Vehicle(npc4_init_state, , npc4_destination, (
    npc4_model));

// Describe a static vehicle
npc5_model = "Hatchback";
npc5_heading = -1/2 pi rad;
npc5_init_state = ("lane_36"->40, npc5_heading);
npc5 = Vehicle(npc5_init_state, , , (npc5_model));

//Describe a pedestrian
ped_model = "Presley";
ped_position = (553011, 4182689.7);
ped_init_state= (ped_position, , 1.0);
ped_state_list = (((553011,4182681.5), , 1.0));
ped_motion = Waypoint(ped_state_list);
ped_destination = ((553011, 4182673.18));
ped = Pedestrian(ped_init_state, ped_motion,
    ped_destination, ped_model);

//Describe Environment
time = 12:00;
weather = {rain:0.5, fog: 0.1, wetness: heavy};
env = Environment(time, weather);

scenario0 = CreateScenario{load(map_name);
              ego_vehicle;
              {npc1, npc2, npc3, npc4, npc5};
              {ped};
              {}; // no static obstacles;
              evnt;};
```

Listing 1: SCENEST script for the motivating example.

```
Trace trace = EXE(scenario0);
ego_vehicle_traj = trace[ego];
npc1_traj_truth = trace[truth][npc1];
npc1_traj = trace[perception][npc1];
diff_dis = diff(npc1_traj_truth, npc1_traj);
dis_dis = dis(ego_vehicle_traj, npc1_traj_truth);
assertion1 = G (dis_dis<=50 -> diff_dis<1);
trace |= assertion1;
assertion2 = G (dis_dis>=1.0);
trace |= assertion2;
vel_dis = spd(ego_vehicle_traj, npc1_traj_truth);
assertion3 = G (dis_dis<3 -> vel_dis<0);
trace |= assertion3;
```

Listing 2: Specification description using AVSpec.

- Ego vehicle: The task of the ego vehicle is to move from the start point to the target location, which is on Lane lane_40 and 50 meters away from the lane's start position.
- $npc1$: $npc1$ is 30 meters ahead to the ego vehicle with the same orientation. It needs to turn right via Lane lane_759 and reach the position which is 100 meters away to the start point of Lane lane_77. The initial speed is $6m/s$, and the speeds on the junction lane and the outgoing lane are $2m/s$ and $6m/s$, respectively.
- $npc2$–$npc4$: We can describe the motion of $npc2$–$npc4$ similarly. It is in general not necessary to specify the full trajectory of an agent. Rather, AVUnit will take the partial trajectory and generate a complete trajectory automatically.
- $npc5$: $npc5$ is placed crossing the Lane lane_36 at the location that is 40 meters away from the stating point.
- ped: ped is required to go through the crosswalk with a constant speed 1m/s.
- time and weather: The time of the day is 12:00, and the weather is rainy, fogging and wetness, whose degrees are 0.5, 0.1, and heavy, respectively.

To evaluate the execution of such a scenario, we can specify different assertions. For example, we may monitor whether the ego vehicle can detect $npc1$ correctly within its sensing range, whether it always keeps a distance of 1 meter to $npc1$, and whether the speed of the ego vehicle is less than that of $npc1$ when their distance is less than a given threshold (e.g., 3 meters). The AVSpec code for these assertions is given in Listing 2. In AVUnit, the input is a text file written in SCENEST and AVSpec. Users must specify the map to be loaded and the motion task of the ego vehicle (i.e., the statements load(map_name) and AV(ego_init_state, ego_target_state)) while others are left as optional.

3 SCENEST: Specifying Dynamic Testing Scenarios

In this section, we summarize the syntax and semantics of SCENEST, our language for specifying dynamic testing scenarios, using Backus-Naur form (BNF). First, the language can be used to specify *scenes*, such as the environment (e.g. weather, time of day) and states of agents (e.g. positions and orientations). Second—distinguishing it from other languages—SCENEST can be used to specify *scenarios*, including the motion of NPC vehicles and pedestrians.

3.1 Describing Scenes

Scenes can be thought of as snapshots of the world, including the environment (e.g. time of day, weather) and the states of all agents within it (e.g. NPC vehicles, pedestrians). In the following, we highlight the most important features of the language for describing AV scenes, namely: (1) the positions of agents, (2) the 'heading' (or orientation) of agents, (3) the states and types of agents, (4) the weather and time of day. SCENEST can also specify several other properties, e.g. size and color. A simplified BNF grammar for these features is given in Figure 3.

Position. In SCENEST, the positions of different agents in scenes are specified using one of two kinds of positional references: (1) coordinates, i.e. a 2D or 3D vector expression; or (2) lane positions, a position relative to a lane's starting point. When specifying a coordinate position ($coordPosition$ in Figure 3), one of three different 'frames' can be chosen: (1) IMU: the ego vehicle coordinate system (forward-left-up), in which the origin is the position of the ego vehicle. (2) ENU: East-north-up, in which coordinates are relative to the map origin. It is the default coordinate frame in SCENEST. (3) WGS84: World geodetic system, whose coordinate origin is the Earth's center of mass and usually is used by GPS (Global Positioning System). Lane positions ($lanePosition$) are relative positions and describe how far an agent is from the starting point of its lane (which is specified using a $laneID$). A lane position is described by the lane ID and the distance/offset ($realExpr$) to the starting point.

For example, as shown in Listing 1, Lines 5 and 13 define the initial positions of the ego vehicle in the map coordinate system and $npc1$ in the ego vehicle coordinate system, and Line 16 specifies the destination position of $npc1$ in lane position, which means the destination is on the lane with ID lane_77 and 100 meters away from the starting point.

Heading. Heading describes the orientation of an agent by specifying its deviation relative to a predefined direction. In SCENEST, predefined directions can be the orientation of a lane point, the ego vehicle, an NPC vehicle, or a pedestrian. In the grammar rule for $heading$, $angleVal$ is the real angle value, $unit$ is either degrees (deg) or radians (rad), and $direction$ is the predefined direction that the angle is relative to. For instance, as stated in Line 12 of Listing 1, $npc1$ has the same orientation as the ego vehicle.

State. Every agent in a scene has an internal state that contains the position, the current direction it is heading, and its current speed. When mutating the state ($state$) of an agent, the position must always be specified, but the heading and speed are optional, defaulting to zero if omitted. A fully-filled state is given in Line 14 of Listing 1, where the last one denotes $npc1$'s speed at the corresponding position.

Agent Types. Agents on the road, whether they are NPC vehicles, pedestrians, or static obstacles, are associated with agent types. SCENEST defines three different agent types as follows: (1) Vehicle types ($velType$), which are utilised to distinguish different vehicles. Vehicle agents can take a specific type (e.g. a detailed vehicle model) or a more generic type (e.g. car, bus, van, bicycle), which can be further customised according to colour and material; (2) Pedestrian types ($pedType$), which describe the different kinds of hu-

$position$	::=	$coordPosition \mid lanePosition$	$lanePosition$	::=	$laneID$ `'->'` $realExpr$
$coordPosition$	::=	$[coordFrame]\ coordExpr$	$laneID$	::=	$stringExpr$
$coordFrame$	::=	`'IMU'` | `'ENU'` | `'WGS84'`			
$heading$	::=	$angleVal\ unit\ [$`'related to'` $direction]$			
$direction$	::=	$lanePosition \mid egoID \mid npcID \mid pedestrianID \mid$ `'EGO'`			
$state$	::=	`'('` $position\ [$`','` $[heading]\ [$`','` $speed]\ ]$ `')'`			
$velType$	::=	`'('` $velSpecificType \mid velGenType[$`','` $color]$ `')'`			
$velGenType$	::=	$car \mid bus \mid van \mid truck \mid bicycle \mid motorcycle \mid tricycle$			
$pedType$	::=	$pedSpecificType \mid pedGenType$	$pedGenType$	::=	`'('` $height$ `','` $color$ `')'`
$staticType$	::=	$sphere \mid box \mid cone \mid cylinder$			
$sphere$	::=	`'('sphere,` $naturalVal$ `')'`	box	::=	`'('box,` $size$ `')'`
$cone$	::=	`'('cone,` $size$ `')'`	$cylinder$	::=	`'('cylinder,` $size$ `')'`
$weather$	::=	$kind$ `':'` $(continuousIndex \mid discreteLevel)$	$time$	::=	$hour$ `':'` $minute$

Fig. 3: Simplified BNF grammar for scenes in SCENEST

man beings in the scene. Similar to vehicles, pedestrians can take the type of a predefined pedestrian model, or a more general form described by height and colour; (3) Static types ($staticType$), which describe the shape and size of static agents (e.g. boxes, cones, cylinders) that serve as obstacles in the scene. For instance, Line 11 in Listing 1 specifies that the model of $npc1$ is *Sedan* with the default colour and material.

Weather & Time. SCENEST can specify combinations of $weather$ in the environment, including different amounts of sun, rain, fog, wetness, and snow. Weather can be quantified using a real value between $[0, 1]$ or a set of labels (e.g. light, middle, heavy) which are mapped to predefined values. Note that different kinds of weather can exist at the same time. In addition to weather, $time$ is another intrinsic component of the environment. It is specified straightforwardly using hours (0–23) and minutes (0–59). For instance, the statements in Lines 47 and 48 of Listing 1 specify that the weather at midday is raining moderately with a little fog and high-level wetness.

3.2 Describing Scenarios

A key distinction of SCENEST in comparison to other scene languages (e.g. [8]) is that it provides a domain-specific way to specify *scenarios*, in which agents are executing independent journey plans in an environment that is dynamically changing. Without specifying scenarios, many real-world behaviours of AVs may not be properly tested. Asking users to fully specify scenarios, however, would be overwhelming. SCENEST strikes a balance by allowing them to specify some key parts (e.g. journey waypoints) while leaving the rest to automatic test generation. In particular, SCENEST supports an agent-oriented style of programming, where the motions of individual agents are specified in a fully decoupled way.

We present the main features of the language that support this design in the following. A simplified BNF grammar of the syntax for scenarios can be found in Figure 4.

Ego Vehicle. The ego vehicle ($egoVehicle$) is the autonomous vehicle equipped with the ADS under test. In SCENEST, it is specified using the keyword `AV`, followed by some optional parameters: the initial state (see $state$ in Figure 3) of the ego vehicle, its target state, and the $type$ of the vehicle. Note that if the parameters are not specified, random values will be generated. Note that the language does not require the user to specify anything about the ego vehicle's motion: this is because it is fully controlled by the ADS under test. For example, Lines 3–9 define a "Lincoln MKZ 2017" ego vehicle with the required motion task, i.e., moving from the position $(553090.05, 4182687.8)$ in the ENU coordinate system to the lane position which is 50 meters away from the start of Lane `lane_40`.

NPCs, Pedestrians, Motion, & Static Obstacles. Scenes can be populated with NPC vehicles ($npcVehicle$), pedestrians ($pedestrian$) and static obstacles ($obstacle$). For NPC vehicles and pedestrians, SCENEST allows the user to (optionally) specify not only their initial and target states but also their $motion$. In contrast to Scenic [8], motion can be specified in a fully decoupled and agent-oriented way. Motion can be either uniform ($uniformMotion$) or based on waypoints ($waypointMotion$). In uniform motion, the NPC vehicle or pedestrian moves with a fixed speed as defined in the $state$. In waypoint-based motion, the agent moves towards the target position via a sequence of points as specified by the user. For instance, Lines 11–17 in Listing 1 specify an NPC vehicle $npc1$, as well as the required motion. For a static obstacle, users only need to describe its position and type ($staticType$).

Environment. SCENEST allows to specify environments, i.e., weather and day of time, in scenarios, which can be changed via fuzzing. We can specify the environment explicitly (env) with the keywords `Environment`, or leave the parameters empty in order to use the default environment settings. For instance, Lines 47–49 in Listing 1 define an Environment by the variable env.

Scenario Definition. Finally, we can create a scenario featuring combinations of all of the above aspects in SCENEST using the `CreateScenario` keyword ($scenario$). The user specifies the map to load, the ego vehicle, lists of NPCs and pedestrians (associated with their independent uniform/waypoint motion configurations), a list of static agents, and (optionally) the environment. Note that our simplified grammar omits the rules for lists of vehicles, pedestrians, and obstacles: they are simply comma-delimited lists defined in the standard way. For instance, Lines 50–55 in Listing 1 give an example of scenario declaration.

```
scenario    ::= 'CreateScenario({' map ';' egoVehicle ';' npcVehicleList ';'
                    pedestrianList ';' obstacleList ';' [env ';'] '}'
egoVehicle  ::= 'AV(' [state ',' state [',' velType] ] ')'
motion      ::= uniformMotion | waypointMotion
uniformMotion ::= 'Uniform(' state ')'
waypointMotion ::= ('Waypoint' | 'WP' | 'W') '(' stateList ')'
npcVehicle  ::= 'Vehicle(' [ state [',' [motion] [',' [state] [',' velType] ] ] ] ')'
pedestrian  ::= 'Pedestrian(' [ state [',' [motion] [',' [state] [',' pedType] ] ] ] ')'
obstacle    ::= 'Obstacle(' [position [',' staticType] ] ')'
env         ::= 'Environment(' [time ','weather] ')'
```

Fig. 4: Simplified BNF grammar for dynamic scenarios in SCENEST

3.3 Semantics of SCENEST

SCENEST specifies the motions of NPC vehicles and pedestrians, and other environment-related information, such as time and weather. Even though SCENEST only describes the motion task of the ego vehicle, the trajectory of the ego vehicle is determined by the ADS under test. Hence, SCENEST describes a complete scenario. We define the semantics of SCENEST as a **trace**, which is a sequence of scenes, describing scene changes in the scenario. Here a scene is a snapshot, at some time instant, of the physical world (including the status of all agents defined in SCENEST) and the ADS world (including the perception results of these agents and traffic conditions by the ADS).

Definition 1. *A scene is a mapping θ such that:*

- $\theta(map)$ *is the map (i.e., a map name);*
- $\theta(time)$ *is the time;*
- $\theta(weather)$ *is the weather;*
- $\theta(ego)$ *is the status of the ego vehicle, including the position, orientation, velocity, and acceleration;*
- $\theta(npc, truth)$ *maps each NPC vehicle npc to its status, including position, orientation, velocity, acceleration, and size, in the real world;*
- $\theta(ped)$ *maps each pedestrian ped to its status, i.e., its position, orientation, velocity, acceleration, and size, in the real world;*
- $\theta(obs)$ *maps each static obstacle obs to its position and size in the real world;*
- $\theta(traffic)$ *returns the status of different traffic signs, such as traffic lights, stop sign, and speed limit, in the real world;*
- $\theta(npc, perception)$ *maps each NPC vehicle npc to its perception status in the ADS world;*
- $\theta(ped, perception)$ *maps each pedestrian ped to its perception status in the ADS world;*
- $\theta(obs, perception)$ *maps each static obstacle obs to its perception status in the ADS world;*
- $\theta(traffic, perception)$ *returns the perception status of different traffic signs in the ADS world.*

In SCENEST, each agent is executed in parallel. The NPC vehicles and pedestrians move according to their defined motions independently, while the motion of the ego vehicle is generated by the ADS, which detects the status of other agents with a given frequency and computes the trajectory. Hence, the execution of SCENEST can be described by a sequence of scenes via time discretization. During the execution, the agents should satisfy some constraints. (1) SCENEST does not allow the change of maps during the execution of a scenario. (2) At each scene, each agent should be on the feasible region of the map, which means that their positions should not exceed the region of the defined map. (3) Each vehicle should move along the lanes in the map and cannot move backward on the same lane. Note that pedestrians may move around the feasible region of the map. (4) A vehicle or pedestrian should follow its speed limit. (5) Different agents in the initial scene θ_0 should be located at different positions. Note that during the execution of the scenario, we do not require, also cannot guarantee, collision avoidance among the agents. (6) If users specify some particular waypoints of an agent in SCENEST, the agent should pass through these waypoints during the scenario execution. Hence, the semantics of SCENEST can be described as follows.

Let $A(\theta_0(map))$ and $L(\theta_0(map))$ be the feasible region and the lane region in the map, respectively, $agent$ be the set of agents defined in SCENEST, $p(\theta(e))$ and $vs(\theta(e))$ be the position and speed of agent e at θ, $vs_{\max}(e)$ be the maximal speed of e, $l(\theta(npc))$ be the lane where the NPC vehicle npc is running along at scene θ. Moreover, $WP(e)$, $e \in agent$, denotes the waypoints of agent e specified in SCENEST. Note that it can be an empty set.

Definition 2. *A trace of SCENEST scenario is a sequence of scenes, denoted as $\pi = \langle \theta_0, \theta_1, \ldots, \theta_n \rangle$, satisfying the following constraints.*

$$\forall \theta_i \in \pi,\ \theta_i(map) = \theta_0(map), \tag{1}$$

$$\forall e \in agent,\ p(\theta_i(e)) \in A(\theta_i(map)), \tag{2}$$

$$p(\theta_i(npc)) \in L(\theta_i(map)), \tag{3}$$

$$l(\theta_i(npc)) = l(\theta_{i+1}(npc)) \triangleq l_i \rightarrow$$
$$d(p(\theta_i(npc)), l_i) \leq d(p(\theta_{i+1}(npc)), l_i), \tag{4}$$

$$vs(\theta_i(e)) \leq vs_{\max}(e), \tag{5}$$

$$\forall e_1, e_2 \in agent,\ p(\theta_0(e_1)) \neq p(\theta_0(e_2)), \tag{6}$$

$$\forall wp \in WP(e),\ \exists \theta_k, \ni wp \in \theta_k(e), \tag{7}$$

where $d(p(\theta_i(npc)), l_i)$ is the distance from the start point of lane l_i to the position $p(\theta_i(npc))$.

Currently, SCENEST models the behaviour of each agent independently. However, it can also depict complex scenarios by separating the interactions among different agents into their individual actions. For instance, we can specify the waypoints of NPC vehicles to describe various interactions among them.

4 AVSpec: Specifying Correctness Properties

In this section, we summarize the syntax and semantics of AVSpec, our language for specifying AV correctness properties. In contrast to existing AV testing approaches, which tend to focus on fixed oracles (e.g. no collisions) and lack the flexibility to customize specifications, AVSpec allows users to tailor richer specifications to their desired requirements and scenarios, independent of the implementation of ADSs, on how the ego vehicle behaves across the multitude of scenarios it encounters during its journey.

4.1 Trajectories, Expressions, and Assertions

Given a trace $\pi = \langle \theta_0, \theta_1, \ldots, \theta_n \rangle$, we can use AVSpec to assert properties along the trace. In this section, we present the syntax of trajectories, five basic types of expressions involving them (i.e., perception difference, position, velocity, speed and acceleration), and assertions. A simplified BNF grammar for these language features is given in Figure 5.

Agent Trajectories. AVSpec can be used to denote the trajectory of the ego vehicle, the perceived trajectories of NPC vehicles and pedestrians by the ADS, and their true trajectories. The trajectory of the ego vehicle with respect to a $trace$ can be obtained by the statement given in Line 2 of Listing 2. In AVSpec, each other agent's truth trajectory and the trajectory perceived by the ADS can be obtained using the form given in Lines 3 and 4 of Listing 2, respectively, which describe the trajectories of $npc1$. Note that the agent can be either an NPC vehicle or a pedestrian specified by the user in a SCENEST scenario.

Perception Difference Expressions. The perception systems of ADSs are currently far from perfect, which makes it important to specify properties regarding the difference between the perceived scene and the actual scene. In AVSpec, such a difference ($perceptionDiffExpr$) is declared with the keyword `diff`. For example, Line 5 in Listing 2 describes the perception difference with respect to $npc1$.

Expressions for Distance, Speed, Velocity, & Acceleration. AVSpec supports multiple kinds of expressions that are important for motion-related specifications. First, one can write an expression over the distance ($distanceExpr$) between two agents or between an agent and a specific position ($position$) along the trace. AVSpec treats the latter as a special single-point polygon. For example, Line 6 in Listing 2 defines the truth distance between the ego vehicle and $npc1$. Second, AVSpec can describe the difference of the speed, velocity and acceleration between two agents or between an agent and a specific value using the keywords `spd`, `vel`, and `acc`, respectively. For instance, Line 11 in Listing 2 specifies the truth speed difference between the ego vehicle and $npc1$. In AVSpec, speed is a real scalar, while velocity and acceleration are vectors.

Note that while not shown in our grammar, basic arithmetic operators can also be applied to the above expressions. For example, the following code:

```
ave_diff = (diff_np1 .+ diff_npc2 .+ diff_ped) ./ 3;
```

expresses the average perception error of the ADS perception module with respect to two NPC vehicles and a pedestrian. Here AVSpec applies ".+", ".-", ".*" and "./" to describe element-wise addition, subtraction, multiplication, division on different expressions, respectively.

Assertions. AVSpec supports different assertions over the previously defined expressions, e.g. properties about the ego vehicle's acceleration behaviour. These assertions support temporal operators (e.g. always, eventually, until) which are interpreted over traces. A simplified BNF grammar for these language features is given in Figure 5.

In AVSpec, the atomic predicates ($atomicPredicate$) are defined over the above expressions and the relational operators (e.g. ==, !=, >, >=). For instance, the following three atomic predicates respectively assert that some distance is larger than one, that some agent is not stationary, and that some average perception difference is less than 0.5:

```
predicate_0 = dis1 > 1;
predicate_1 = speed_dis > 0;
predicate_2 = ave_diff <= 0.5;
```

Based on the atomic predicates, AVSpec specifies general assertions using the Signal Temporal Logic (STL) formalism with temporal operators (i.e., 'always' (G), 'eventually' (F), 'until' (U), and 'next' (X)) and logical operators (i.e., 'and' (&), 'or' (|), 'not' (~), and 'implies' (->)). For example, Lines 7, 9, and 12 in Listing 2 show three general assertions for safety specifications.

4.2 Semantics

In the following, we define the semantics of AVSpec. Let $vs(t)$, $p(t)$, $q(t)$, $v(t)$, $a(t)$, and $\kappa(t)$ be the speed, position, orientation, velocity, acceleration, and shape of some agent at time instant t, where $vs(t) \in R$, $p(t), v(t), a(t) \in R^3$, $q(t) \in R^4$ is a unit quaternion, and $\kappa(t)$ is described as a polygon. We add the subscript to describe the related agent. Hence, at any time instant t, the status of the ego vehicle is denoted as $X_0^e(t) = [p^e(t), q_0^e(t), v_0^e(t), a_0^e(t), \kappa_0^e(t)]$; let $X_i^\alpha(t) = [p_i^\alpha(t),\ q_i^\alpha(t),\ v_i^\alpha(t),\ a_i^\alpha(t),\ \kappa_i^\alpha(t)]$ and $X_i^\beta(t) = [p_i^\beta(t),\ q_i^\beta(t),\ v_i^\beta(t),\ a_i^\beta(t),\ \kappa_i^\beta(t)]$ be the status of agent e_i, either an NPC vehicle or a pedestrian, obtained from the perception module and the ground truth, respectively.

Expression semantics. For the expression semantics, we explain the computation of the sequences of perception difference, position distance, velocity distance, speed distance, and acceleration distance along a trace.

Definition 3. *Given the perception result X_i^α and the ground truth X_i^β of an agent e_i, the perception difference at time instant t is defined as the weighted sum of the errors of position, orientation, velocity, and shape.*

$$\begin{aligned} diff(X_i^\alpha, X_i^\beta)(t) =& w_1 d_1(p_i^\alpha(t), p_i^\beta(t)) + w_2 d_2(q_i^\alpha(t), q_i^\beta(t)) \\ &+ w_3 d_3(v_i^\alpha(t), v_i^\beta(t)) + w_4 d_4(\kappa_i^\alpha(t), \kappa_i^\beta(t)), \end{aligned}$$

where $d_1(p_i^\alpha(t), p_i^\beta(t)) = \|p_i^\alpha(t) - p_i^\beta(t)\|_2$, $d_2(q_i^\alpha(t), q_i^\beta(t)) = \arccos q_i^\alpha(t) \cdot q_i^\beta(t)$, $d_3(v_i^\alpha(t), v_i^\beta(t)) = \|v_i^\alpha(t) - v_i^\beta(t)\|_2$, $d_4(\kappa_i^\alpha(t), \kappa_i^\beta(t)) = 1 - \frac{|\kappa_i^\alpha(t) \bigcap \kappa_i^\beta(t)|}{|\kappa_i^\beta(t)|}$, *$|\kappa|$ denotes the area of the polygon κ, and $w_1 - w_4$ are normalized weights satisfying* $w_1 + w_2 + w_3 + w_4 = 1$.

Given an agent e, let $f_\kappa(e,t)$, $f_s(e,t)$, $f_v(e,t)$, $f_a(e,t)$ be the polygon, speed, velocity, and acceleration of e at t, respectively. Note that here e can also be a single position,

$$
\begin{aligned}
\mathit{objectTrajectory} &::= \mathit{egoTrajectory} \mid \mathit{agentPerceivedTrajectory} \mid \mathit{agentTrueTrajectory}\\
\mathit{egoTrajectory} &::= \mathit{trace}\ \texttt{'[' ego ']'}\\
\mathit{agentPerceivedTrajectory} &::= \mathit{trace}\ \texttt{'[perception][' (npcVehicle | pedestrian | obstacle) ']'}\\
\mathit{agentTrueTrajectory} &::= \mathit{trace}\ \texttt{'[truth][' (npcVehicle | pedestrian | obstacle) ']'}\\
\mathit{perceptionDiffExpr} &::= \texttt{'diff('}\ \mathit{agentPerceivedTrajectory}\ \texttt{','}\ \mathit{agentTrueTrajectory}\ \texttt{')'}\\
\mathit{distanceExpr} &::= \texttt{'dis('}\ (\mathit{position} \mid \mathit{objectTrajectory})\ \texttt{','}\ (\mathit{position} \mid \mathit{objectTrajectory})\ \texttt{')'}\\
\mathit{speedExpr} &::= \texttt{'spd('}\ (\mathit{speed} \mid \mathit{objectTrajectory})\ \texttt{','}\ (\mathit{speed} \mid \mathit{objectTrajectory})\ \texttt{')'}\\
\mathit{velocityExpr} &::= \texttt{'vel('}\ (\mathit{velocity} \mid \mathit{objectTrajectory})\ \texttt{','}\ (\mathit{velocity} \mid \mathit{objectTrajectory})\ \texttt{')'}\\
\mathit{accelerationExpr} &::= \texttt{'acc('}\ (\mathit{velocity} \mid \mathit{objectTrajectory})\ \texttt{','}\ (\mathit{acceleration} \mid \mathit{objectTrajectory})\ \texttt{')'}\\
\mathit{assertion} &::= \mathit{atomicPredicate} \mid \mathit{assertion}\ (\texttt{'\&'} \mid \texttt{'|'} \mid \texttt{'->'})\ \mathit{assertion} \mid \texttt{'~'}\ \mathit{assertion} \mid\\
&\qquad (\texttt{'G'} \mid \texttt{'F'} \mid \texttt{'X'} \mid \texttt{'U'})\ \mathit{assertion} \mid\\
&\qquad (\texttt{'G'} \mid \texttt{'F'} \mid \texttt{'U'})\ \texttt{'['}\ \mathit{realExpr}\ \texttt{':'}\ \mathit{realExpr}\ \texttt{']'}\ \mathit{assertion}
\end{aligned}
$$

Fig. 5: Simplified BNF grammar for trajectories, expressions, and assertions in AVSpec

speed, velocity, or acceleration. We have the following semantics for the position, speed, velocity, and acceleration expressions.

Definition 4. *Given two agents e_1 and e_2, the computations of $dis(e_1, e_2)$, $spd(e_1, e_2)$, $vel(e_1, e_2)$, $acc(e_1, e_2)$ are defined as follows.*

$$
\begin{aligned}
dis(e_1, e_2)(t) =& \min_{\substack{p_1 \in f_\kappa(e_1,t),\\ p_2 \in f_\kappa(e_2,t)}} \|p_1 - p_2\|_2, e_1, e_2 \in X \cup \{p\},\\
spd(e_1, e_2)(t) =& f_s(e_1, t) - f_s(e_2, t), e_1, e_2 \in X \cup \{vs\},\\
vel(e_1, e_2)(t) =& \|f_v(e_1, t) - f_v(e_2, t)\|_2, e_1, e_2 \in X \cup \{v\},\\
acc(e_1, e_2)(t) =& \|f_a(e_1, t) - f_a(e_2, t)\|_2, e_1, e_2 \in X \cup \{a\},
\end{aligned}
$$

where $X = \{X^e(t), X_i^\alpha(t), X_i^\beta(t)\}$, and p, vs, v, and a are constant variables.

Quantitative semantics of AVSpec. In AVSpec, we apply the quantitative semantics of STL to measure the degree of satisfaction of an AVSpec assertion. STL, a kind of temporal logic formalism, can specify a linear-time logical property of continuous real-valued signals [29] and is widely applied in the cyber-physical system whose programs are closed intertwined with the physical world. For the details of the quantitative semantics of STL, please refer to [29]–[31]. The successful application of STL in testing CPSs and AVs [26], [27], [32], [33] motivated us to formalise different specifications. Unlike existing work, AVUnit provides a general and extensible language to formalise some basic specifications (e.g., collision avoidance and task completion) and support customising different specifications.

Let $\mathcal{U}$, $\Box$, $\Diamond$ and $\bigcirc$ be the operators of "until", "always", "eventually" and "next", respectively; $\mathcal{I}$ is a real-time interval $[a, b]$ $(0 \le a < b)$ of $[a, +\infty)$ with the interval operation $t + \mathcal{I} = [t + a, t + b]$. According to the syntax of STL, an AVSpec assertion ϕ can be described as:

$$
\begin{aligned}
\phi \ :=\ & \mu \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \varphi_2 \mid \phi_1 \rightarrow \varphi_2 \mid\\
& \phi_1\, \mathcal{U}_\mathcal{I}\, \varphi_2 \mid \Box_\mathcal{I}\phi \mid \Diamond_\mathcal{I}\phi \mid \bigcirc \phi
\end{aligned}
$$

where the atomic predicate $\mu = f(\pi) \ge 0$ can be written as "f_1 *op* f_2" on a trace π, which is defined as:

$$
f(\pi) = \begin{cases}
f_1(\pi) - f_2(\pi), & op \in \{>, \ge\};\\
f_2(\pi) - f_1(\pi), & op \in \{<, \le\};\\
|f_1(\pi) - f_2(\pi)|, & op \in \{!=\};\\
-|f_1(\pi) - f_2(\pi)|, & op \in \{==\};
\end{cases}
$$

where f_1 and f_2 are two expressions described as the arithmetic combinations of $diff(X_i^\alpha, X_i^\beta)$, $dis(e_1, e_2)$, $spd(e_1, e_2)$, $vel(e_1, e_2)$, and $acc(e_1, e_2)$. Note that at any time instant t, the untimed operators $\mathcal{U}$, $\Box$ and $\Diamond$ are equal to the timed operators with $a = 0$ and $b = |\pi| - t$.

Applying the quantitative semantics of STL, in the following, we define the quantitative semantics for AVSpec, which allows us to evaluate not only whether an AVSpec assertion ϕ is satisfied but also how far it is from being violated or satisfied. Note that the latter is important for our feedback-guided fuzzing algorithm.

Definition 5 (Quantitative Semantics)**.** *Given a trace π and an AVSpec assertion ϕ, the quantitative semantics is defined as the robustness degree $\rho(\phi, \pi, t)$ which returns a real value. The function ρ is defined as follows.*

$$
\begin{aligned}
\rho(\mu, \pi, t) &= f(\pi)(t),\\
\rho(\neg\phi, \pi, t) &= -\rho(\phi, \pi, t),\\
\rho(\phi_1 \wedge \phi_2, \pi, t) &= \min\{\rho(\phi_1, \pi, t), \rho(\phi_2, \pi, t)\},\\
\rho(\phi_1 \vee \phi_2, \pi, t) &= \max\{\rho(\phi_1, \pi, t), \rho(\phi_2, \pi, t)\},\\
\rho(\phi_1\, \mathcal{U}_\mathcal{I}\, \phi_2, \pi, t) &= \sup_{t' \in t+I} \min\{\rho(\phi_2, \pi, t'), \inf_{t'' \in [t, t']} \rho(\phi_1, \pi, t'')\},\\
\rho(\Diamond_\mathcal{I}\phi, \pi, t) &= \sup_{t' \in t+\mathcal{I}} \rho(\phi, \pi, t'),\\
\rho(\Box_\mathcal{I}\phi, \pi, t) &= \inf_{t' \in t+\mathcal{I}} \rho(\phi, \pi, t'),\\
\rho(\bigcirc\phi, \pi, t) &= \rho(\phi, \pi, t + 1).
\end{aligned}
$$

For example, suppose the distance sequence between the ego vehicle and the NPC vehicle $npc1$ of a trace π is $\{(t, d) : (t_0, 10), (t_1, 8.69), (t_2, 7.32), (t_3, 6.3), (t_4, 5.4), (t_5, 4.5), (t_6, 5.0), \ldots, (t_n, d_n), \ldots\}$ where $d_n > 5.0$ when $t_n > t_6$, and the related assertion is $\mu = G(d > 3.0)$, then we have $\rho(\mu, \pi, t_0) = 1.5$, and the minimal one is reached at t_5. It means that the assertion μ is satisfied for π.

5 Fuzzing

AVUnit features two feedback-guided scenario generation algorithms that automatically search for test cases likely to trigger violations of the correctness specification (i.e., the AVSpec assertion), and cover different ways of violating the assertion. They are a kind of search-based fuzzing algorithm [34]. The idea is to keep evolving a test suite using

Algorithm 1: Failure-Inducing Fuzzing Algorithm.

Input: SCENEST script S with assertion ϕ; population size n; crossover and mutation probabilities c_p and m_p; top criteria.
Output: A set of test cases that violate ϕ: $\mathbb{S}$.

$\mathbb{S} = \emptyset$;
Evaluate the initial test case described in S;
if *S violates ϕ* **then**
  $\mathbb{S} = \{S\}$;
end
Generate a set of random test cases based on S, denoted as $T_0 = \{s_1, \ldots, s_n\}$, where $T_0[i] = s_i$;
repeat
  Simulate each test case s_i in T_0 using a simulator and obtain a trace π_i;
  Compute the robustness $\rho(\phi, \pi_i)$;
  if $\rho(\phi, \pi_i) < 0$ **then**
    $\mathbb{S} = \mathbb{S} \cup \{s_i\}$
  end
  $T = \emptyset$;
  Sort T_0 according to $\rho(\phi, \pi_i)$ in ascending order;
  repeat
    Randomly select two test cases, one from the first half of the sorted T_0 and another from T_0;
    Add the test case with lower robustness to T;
  until $|T| < n$;
  $k = 1$;
  while $k < n$ **do**
    Generate a random value $p \in [0, 1]$;
    if $p < c_p$ **then**
      $T[k], T[k+1] = Crossover(T[k], T[k+1])$;
    end
    $k = k + 2$;
  end
  for $k = \{1, 2, \ldots, n\}$ **do**
    Generate a random value $p \in [0, 1]$;
    if $p < m_p$ **then**
      $T[k] = Mutation(T[k])$;
    end
  end
  $T_0 = T$;
until timeout *or the top criteria has been satisfied*;
return $\mathbb{S}$.

the robustness of the test cases against the AVSpec assertion and the failure coverage measure as the objective functions.

5.1 Failure-Inducing Fuzzing

The overall fuzzing algorithm for failures is shown in Algorithm 1. Note that the objective is to identify test cases that fail the assertion in this setting. The overall algorithm is a genetic algorithm (GA), customized for the purpose of testing AVs based on a SCENEST script S and an AVSpec assertion ϕ. The output is a set of test cases that are allowed by S and fail ϕ. We first randomly generate a test suite based on the scenario described in the SCENEST script as the initial population. Following this, we can produce different generations via selection, crossover, and mutation. Although the overall process adheres to the general framework of CPS testing [26], [27], [32], [33], we encounter particular challenges specifically related to the crossover and mutation operations, which will be elaborated on in detail. Before giving the details, we first describe population encoding.

Population encoding. Based on the syntax of SCENEST, a test case contains the ego vehicle, NPC vehicles, pedestrians, static obstacles, and the environment. The operable parameter for the ego vehicle is its initial position. Note that the target position is the task to be tested, so we do not mutate it, while other parameters characterizing the ego vehicle (e.g., vehicle type, vehicle color, and vehicle size) do not affect the execution of the ADS significantly. Without loss of generality, we assume that each NPC vehicle or pedestrian is stationary at the end position (if specified). If a non-zero speed is required at a position, one can set one more waypoint with zero speed after this position. Hence, for each NPC vehicle or pedestrian, we can change every possible parameter except the speed at the last position. For static obstacles and the environment, we can operate on all parameters that are used to characterize them.

We classify these operating parameters into eight categories: *position, speed, vehicle type, pedestrian type, obstacle type, color, day time, and whether*. The category of position is further divided into three sub-categories: vehicle positions (i.e, ego positions and NPC vehicles' positions), pedestrian positions, and obstacle positions. This is because vehicles in a scenario are usually required to move along lanes, while pedestrians may move around the empty regions of the map, and obstacles can be placed at any point of the map (e.g, on the road, in a tree, or on the roof). Similarly, the category of speed is divided into vehicle speed and pedestrian speed since they have different ranges.

Among the above categories, speed and position are continuous variables, so we apply real-value encoding for vehicle speed, pedestrian speed, vehicle position, pedestrian position, and obstacle position. In detail, for an NPC vehicle, even though we can change the lane sequence of a vehicle, users may be interested in this specific route described in SCENEST. So, in our fuzzing algorithm, we do not change the lane sequence. Since vehicles are moving along lanes, their positions can be translated to lane positions, allowing the encoding of each position to be described by its offset in the lane position format. Hence, for each NPC vehicle, its speed and position offset are organized sequentially based on the sequence of waypoints, including the initial and target waypoints. Then, the speed and the position of all NPC vehicles are concatenated in the same sequence to form the chromosomes of vehicle speed and vehicle position, respectively. The encoding of pedestrian speed is the same as that for NPC vehicle speed. For position encoding of pedestrians and obstacles, since their motion may not move along a lane, we use coordinate positions to describe their locations. Each coordinate position is encoded as a gene. Similarly, the positions of all pedestrians (resp., obstacles) are concatenated sequentially to form the chromosome of pedestrian (resp., obstacle) position.

The categories of vehicle type, pedestrian type, obstacle type, color, and time of day are discrete ones. Each of them is encoded as a discrete vector. The category of weather is a dictionary variable whose keys and values are the types of weather and the quantification of the weather type, respectively. Note that if the value of a weather type is described as a predefined level label (e.g., small, middle), it will be transferred to a real value in $[0, 1]$. Hence, the weather can be encoded as a vector $[0, 1]^{|wk|}$, where $|wk|$ is

TABLE 1: Encoding of Vehicles

	ego	npc1	npc2	npc3	npc4	npc5
offset	[10.27,	40.27, 1, 10, 100	60.27,	10,	50, 100,	40]
speed		[6, 2, 6,	6,	5.8,	8,	0]
type		[1,	3,	5,	6,	4]

TABLE 2: Encoding of The Pedestrian *ped*

position	[(553011, 4182689.7), (553011,4182681.5), (553011, 4182673.18)]
speed	[1, 1]
type	[6]

the number of weather kinds defined in a simulator.

Example of population encoding in `AVUnit`. As an example, we show the population encoding of the motivating example. In this example, suppose all possible NPC vehicle types and pedestrians are [Sedan, SUV, Jeep, Hatchback, SchoolBus, BoxTruck] and [Bob, EntrepreneurFemale, Howard, Johny, Pamela, Presley, Robin, Stephen, Zoe], respectively, and the available weather types are [rain, fog, wetness]. Hence, the encoding of the vehicles and the pedestrian is shown in Tables 1 and 2. Weather and time are encoded as $[0.5, 0.1, 0.8]$ and $[12, 0]$, respectively.

Selection. Since our target is to generate test cases with negative robustness, we first sort the population according to their robustness in ascending order. The selection of an individual is described as follows. We first select an individual from the first half of the population and then select an individual from the whole population. Finally, the individual with lower robustness is selected from the two individuals. Repeating the selection process, we can select a parent population with the given number of individuals.

Crossover. For each pair of test cases, the crossover can be done only in the same category. In theory, we can perform crossover for each category. However, crossover on positions may cause infeasible scenarios which violate the constraint described in Equation 4 in Definition 2. Hence, to guarantee the feasibility of the new individual, we do not perform crossover on the vehicle and pedestrian position chromosomes.

Mutation. We can do mutation for each category. For continuous categories, i.e., position, speed, and weather, we apply Gaussian mutation. To guarantee the feasibility of motion, we need the clipping function to guarantee that all speeds are positive, no backward motion in the waypoints of NPC vehicles, and each weather quantification is in $[0, 1]$. For discrete categories, we apply random mutation, i.e., randomly select one value from the corresponding set.

5.2 Failure-Coverage Fuzzing

Given an `AVSpec` assertion ϕ, there may be different ways of violating it. We thus define a specification-based coverage, failure coverage, to measure the test sufficiency based on the number of predicates that could violate ϕ. The existing coverage-based methods usually concentrate on the behavior sequence of a scenario, such as [35]. Instead, our goal is to produce scenarios that can provoke various ways of violating the specification when failure scenarios occur.

Let $\Gamma(\phi)$ be the set of all such predicates, called *failure predicates*, whose satisfaction will lead to the violation of ϕ. Based on the syntax of `AVSpec`, each assertion can be written as one of the following formulas: $q_1 \& q_2$, $q_1 | q_2$, $q_1 \rightarrow q_2$, $\neg q_1$, $G\ q_1$, $F\ q_1$, $q_1\ U\ q_2$, and $X\ q_1$. Hence, Γ can be defined iteratively as follows.

Definition 6. *Given an assertion, the set of failure predicates can be computed as follows.*

- $\Gamma(p) = \{\neg p\}$ *if* p *is an atomic predicate,*
- $\Gamma(q_1 \& q_2) = (\Gamma(q_1) \& q_2) \bigcup (q_1 \& \Gamma(q_2)) \bigcup (\Gamma(q_1) \& \Gamma(q_2))$,
- $\Gamma(q_1 | q_2) = \Gamma(q_1) \& \Gamma(q_2)$,
- $\Gamma(q_1 \rightarrow q_2) = q_1 \& \Gamma(q_2)$,
- $\Gamma(\neg q_1) = q_1$,
- $\Gamma(G\ q_1) = F\ \Gamma(q_1)$,
- $\Gamma(F\ q_1) = G\ \Gamma(q_1)$,
- $\Gamma(q_1 U q_2) = (\Gamma(q_1) \& \Gamma(q_2)) \bigcup ((q_1 \& \Gamma(q_2)) U (\Gamma(q_1) \& \Gamma(q_2)))$,
- $\Gamma(X q_1) = X(\Gamma(q_1))$,

where the operators on two sets A and B can be described as follows: $A \& B = \{q_1 \& q_2 : \forall q_1 \in A, \forall q_2 \in B\}$, $F\ A = \{F\ q : \forall q \in A\}$, $G\ A = \{G\ q : \forall q \in A\}$, $X\ A = \{X\ q : \forall q \in A\}$, and $A\ U\ B = \{q_1\ U\ q_2 : \forall q_1 \in A, \forall q_2 \in B\}$.

Example for the computation of failure predicates. We illustrate the generation of failure predicates of the following assertion $q = p_1 \& p_2 \& p_3$, where p_1 - p_3 are atomic predicates. Let $q_1 = p_1 \& p_2$, we have:

$$\begin{aligned}
\Gamma(q_1) &= (\Gamma(p_1) \& p_2) \cup (p_1 \& \Gamma(p_2)) \cup (\Gamma(p_1) \& \Gamma(p_2)) \\
&= \{\neg p_1 \& p_2\} \cup \{p_1 \& \neg p_2\} \cup \{\neg p_1 \& \neg p_2\} \\
&= \{\neg p_1 \& p_2, p_1 \& \neg p_2, \neg p_1 \& \neg p_2\} \\
\Gamma(q) &= (\Gamma(q_1) \& p_3) \cup (q_1 \& \Gamma(p_3)) \cup (\Gamma(q_1) \& \Gamma(p_3)) \\
\Gamma(q_1) \& p_3 &= \{\neg p_1 \& p_2 \& p_3, p_1 \& \neg p_2 \& p_3, \neg p_1 \& \neg p_2 \& p_3\} \\
q_1 \& \Gamma(p_3) &= \{p_1 \& p_2 \& \neg p_3\} \\
\Gamma(q_1) \& \Gamma(p_3) &= \{\neg p_1 \& p_2 \& \neg p_3, p_1 \& \neg p_2 \& \neg p_3, \neg p_1 \& \neg p_2 \& \neg p_3\}
\end{aligned}$$

Given an assertion q, $\Gamma(q)$ contains all formulas that violate q, and our failure coverage measures how many formulas in $\Gamma(q)$ are covered. Given a scenario S whose generated trace is π, let $V(S) = \{\widetilde{q} \in \Gamma(q) : (\pi, t) \vDash \widetilde{q}\}$. Hence, the failure coverage of a set of scenarios $\mathcal{S}$ can be defined as follows.

Definition 7. *Given a set of scenarios $\mathbb{S}$ and a testing assertion q, the failure coverage, denoted as $cov(\mathbb{S}, q)$, is defined as $cov(\mathbb{S}, q) = |\cup_{S \in \mathbb{S}} V(S)| / |\Gamma(q)|$.*

Clearly, a larger failure coverage will result in diverse violated test cases, which has a higher probability to generate more issues for the ADSs under test. Hence, our failure-coverage fuzzing algorithm aims to generate a test suite that can cover as many failure predicates as possible, as given in Algorithm 2. Given an assertion ϕ, we first compute its failure predicates $\Gamma(\phi)$ (Line 1) and then generate a set of initial test cases randomly (Line 2). At each generation, the execution of each test case s_i generates a local robustness vector ρ_Γ^i containing the robustness of the failure predicates with respect to π_i (Line 8) and updates the global robustness vector ρ_Γ, which indicates the covered, as well as the maximal robustness, and uncovered failure predicates (Line 9) until now. Hence, the test cases are divided into two subsets, i.e., T^α and T^β. The test cases in T^α cover new failure predicates (i.e., Lines 11 and 12), while the rest test cases are associated with the corresponding fitness values $f_\rho(s_i)$. The fitness value of each test s_i in T^β is defined as

Algorithm 2: Failure-Coverage Fuzzing Algorithm.

Input: A SCENEST script S; an assertion ϕ; a population size n, and the maximal number of generations G
Output: A test suite

1 Generate the set of failure predicates of ϕ: $\Gamma(\phi)$;
2 Randomly generate the initial population $T = \{s_1, \ldots, s_n\}$;
3 Initialize the set of satisfied failure predicates $\Gamma_c(\phi) = \emptyset$ and the global robustness vector $\rho_\Gamma = [\rho_\Gamma(\phi_i) = -\infty : \forall \phi_j \in \Gamma(\phi)]$;
4 **repeat**
5 $T^\alpha = T^\beta = T_n^\beta = \emptyset$;
6 **for** *each* s_i *in* T **do**
7 Execute s_i via simulation and obtain trace π_i ;
8 Compute the local robustness vector: $\rho_\Gamma^i = [\rho_\Gamma^i(\phi_j) = \rho(\phi_j, \pi_i) : \phi_j \in \Gamma(\phi)]$;
9 Update $\rho_\Gamma(\phi_j)$ to $\rho_\Gamma^i(\phi_j)$ if $\rho_\Gamma^i(\phi_j) > \rho_\Gamma(\phi_j) > 0$;
10 Obtain the satisfied predicates in $\Gamma(\phi)$: $\Gamma_c^i(\phi, s_i) = \{\phi_j : \rho(\phi_j, \pi_i) > 0\}$;
11 **if** $\Gamma_c^i(\phi, s_i) \setminus \Gamma_c(\phi) \neq \emptyset$ **then**
12 $T^\alpha = T^\alpha \cup \{s_i\}$; $\Gamma_c(\phi) = \Gamma_c(\phi) \cup \Gamma_c^i(\phi, s_i)$
13 **else**
14 compute the fitness: $f_\rho(s_i) = \max\{\rho_\Gamma^i(\phi_j) : \rho_\Gamma^i(\phi_j) < 0 \wedge \rho_\Gamma(\phi_j) < 0\}$;
15 $T^\beta = T^\beta \cup \{(s_i, f_\rho(s_i))\}$
16 **end**
17 **end**
18 **if** $T^\beta \neq \emptyset$ **then**
19 Sort T^β according to $f_\rho(s_i)$ in descending order;
20 **repeat**
21 Randomly select two test cases, one from the first half of T^β and another from T^β;
22 Add the one with a larger fitness to T_n^β;
23 **until** $|T_n^\beta| = |T^\beta|$;
24 **end**
25 $T' = T^\alpha \cup T_n^\beta$;
26 Apply the crossover and mutation operators described in Section 5.1 to T' to generate the next population, denoted as T ;
27 **until** timeout *or the coverage criteria has been satisfied*;

the maximal robustness of the uncovered failure predicates in ρ_Γ^i (Line 14). Then, we select the same number of test cases, denoted as T_n^β, from T^β if $T^\beta \neq \emptyset$ (Lines 18–24). The selection process will select test cases with larger negative robustness for the uncovered predicates. Finally, we apply the crossover and mutation operators described in Section 5.1 to the new test cases T' and generate the population of the next generation.

6 Evaluation

We implement `AVUnit` with LGSVL+Apollo and aim to ask the following research questions.
RQ1: Expressiveness: What scenarios and specifications can `AVUnit` describe?
RQ2: Effectiveness: What ADS issues can `AVUnit` discover?
RQ3: Efficiency: How efficiently can `AVUnit` discover ADS issues?

To answer **RQ1**, based on the standards, such as OpenScenario [6] and the framework from US Department of Transportation [36], for the description of autonomous driving scenarios, we summarize the main features to describe a scenario and describe how `SCENEST` can satisfy these features. We also compare it against other state-of-the-art languages.

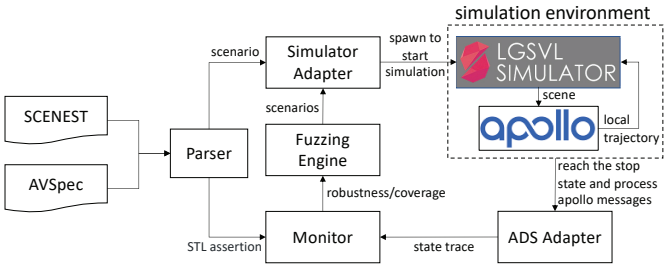

Fig. 6: The implementation of `AVUnit` with the LGSVL+Apollo simulation environment.

To answer **RQ2** and **RQ3**, we apply `AVUnit` to conduct testing on Baidu Apollo, one of state-of-the-art ADSs. Apollo is the only production-scale ADS with an open-source version. It has been deployed in real vehicles to drive on some restricted public roads, e.g., autonomous driving Lincoln MKZ[1] and Baidu's Apollo robotaxi[2]. Fig. 6 shows our modularised implementation on the LGSVL+Apollo simulation platform. It contains five modules: (1) the parser module to parser the textual input and extract the scenario, (2) the simulation adapter to preprocess the scenario so to spawn the scenario to the specific simulation environment, (3) the ADS middleware for communication between `AVUnit` and the simulation environment, (4) monitor to check whether the execution of a test case violates the assertion, and (5) a fuzzing engine to generate new test cases for checking in order the identify counterexamples.

The simulation starts by invoking the simulator adapter to spawn a given scenario s in the simulator. It will set the simulation environment (like time and weather), the initial scene of s, and the trajectories of the NPC vehicles and pedestrians described in s, and then start the simulation. The detailed execution can be described as follows. At any time instant, the simulator sends the current scene to the ADS, based on which the ADS computes a local trajectory for the ego vehicle and sends it back to the simulator. Thus, according to the dynamic equations, the simulator ensures the ego vehicle and other agents move along their trajectories. Note that the trajectories of other agents are predefined in s, and the simulator will send scenes at a given rate. When the received scene triggers the requirement of re-computation, the ADS computes a new local trajectory, and the simulator controls the movement of the ego vehicle based on the new trajectory. Once the ego vehicle reaches a stop state (e.g., arrives at the destination, causes a collision, or reaches the maximal motion time), the execution is completed. The ADS adapter collects and processes the ADS messages to generate the state trace for specification checking.

To illustrate the usage of `AVUnit` and evaluate its effectiveness, we extract eight general scenarios from the report of the U.S. Department of Transportation [36] according to

1. https://medium.com/apollo-auto/baidu-apollo-receives-driverless-vehicle-test-permit-in-california-c332ad5976de
2. https://iot-automotive.news/baidu-apollo-robotaxi/

the availability of the San Francisco HD map provided by LGSVL. They are: *T-Junction* (S1), *intersection with single-direction roads* (S2), *intersection with double-direction roads* (S3), *intersection with mixed-direction roads* (S4), *intersection with stop signs* (S5), *lane changing within a road* (S6), *lane changing among different roads connected by an intersection* (S7), and *overtaking scenario* (S8). It is worth noting that the regions in these scenarios are selected such that the corresponding lane gaps are suitable for Apollo to perform lane changes, as due to the automatic HD map generation, some lane gaps are otherwise too large. In this paper, we consider two kinds of assertions: collision avoidance and task achievement.

- Collision avoidance assertion: statement1 = G (dis_1 >= rho1 & dis_2 >= rho1 & ... & dis_n >= rho1), where n is the number of NPC vehicles in a scenario, dis_i = dis(ego_traj, trace[npc_i]) is the distance expression, and rho1=0.5 meters is the threshold of safety distance.
- Task achievement: statement2 = F dis(ego_traj, ego_target_position) <= rho2, where dis(ego_traj, ego_target_position) is the distance expression for the distance between the current state of the ego vehicle and its destination, rho2=2 meters is a tolerance threshold.

Hence, the final assertion ϕ is described as "statement1 & statement2". The corresponding `SCENEST` scripts can be found at our website: https://avunit2021.github.io/. Note that Apollo's open-source version focuses more on safety-related requirements, and sometimes, it is hard for us to confirm whether a performance-related issue (e.g., a large jerk during the motion) is an issue caused by the ADS. Hence, we focus on two safety specifications in our experiments. However, the performance-related specifications can be used by the ADS's developers, testers, and users.

Even though there are some studies on testing of the perception and machine learning components of ADSs [26], [37], which focus more on generating different scenes rather than dynamic scenarios, there are no systematic methods for testing a whole ADS in the literature. Testing on a whole ADS needs to consider different motion behaviours of other agents, such as NPC vehicles around. Hence, we only compare our GA-based fuzzing algorithms with the random method.

6.1 Expressiveness of `AVUnit`

Currently, the concept of Operational Design Domain (ODD) is widely applied by tech companies and automotive manufacturers, such as Waymo, Ford, and TuSimple, to indicate where ADSs can operate safely [38]. An ODD specifies what operating parameters should be managed in a scenario for AVs. According to the technical report from the U.S. Department of Transportation [36] and the SAE J3016 [39], the ODDs can be divided into physical infrastructure (e.g., road type and geometry), operational constraints (e.g., speed limits, traffic conditions), agents (e.g., roadway users like vehicles and pedestrians, and non-roadway users like obstacles), connectivity (e.g., traffic density), environmental conditions (e.g., weather, illumination), and zones [36]. Focusing on ADS testing, they can be further classified into two categories: fixed ODDs and flexible ODDs. Fixed ODDs, such as the physical infrastructure, operational constraints, and zones, rely on the maps that an ADS is running on and cannot be changed by users. They are achieved in `AVUnit` by loading a required map. Flexible ODDs, such as agents and environmental conditions, can be customized by users. In `AVUnit`, users can describe different environments, stationary obstacles, NPC vehicles, and pedestrians, together with their motion behaviour. In detail, using the statement `Environment(time, weather)`, `AVUnit` can define different environmental conditions; for NPC vehicles and pedestrians, `AVUnit` describes their behaviours in a decoupled way by stating each agent's motion in terms of uniform motion or a detailed trajectory. Hence, `AVUnit` also supports flexible ODDs.

Moreover, `AVUnit` supports scene elements that can be described by other languages, such as OpenScenario [6], GeoScenario [7], Scenic [8], SceML [9], Paracosm [10] and M-SDL [11]. A detailed comparison of these languages is given in Table 3. From the table, we can find that one of the main characteristics of `AVUnit` is that it supports the ability to perform mutations directly on the variables defined in it.

`AVUnit` can describe various specifications with respect to perception error, distance, speed, velocity, and acceleration. In the sequel, we list some typical specifications supported by `AVUnit`. As the examples illustrated in Section 4.1, `AVUnit` can define different assertions for collision avoidance and perception accuracy. Here we illustrate another three kinds of assertions. The first one is for task achievement. With `AVSpec`, we can describe different kinds of task achievements, e.g., reaching the destination with or without time budgets, which are given as follows:

```
trace |= F[0,T] (dis(ego_state, target_position)<=
    rho);
trace |= F (dis(ego_state, target_position)<=rho);
```

where `target_position` is the destination of the ego vehicle defined in `SCENEST`, T is the time budget for the ego vehicle's motion, and `rho` is a predefined error tolerance. The second one is the assertions related to traffic rules. For example, the ego vehicle should stop before an intersection when the corresponding traffic light is red. It can be written as:

```
light_id = trace[truth][traffic][light][id];
light_color = trace[truth][traffic][light][color];
light_offset = trace[truth][traffic][light][offset];
dis_light = dis(ego_state, light_id);
trace |= G ((light_color==RED & light_offset<=
    dis_light<=light_offset+rho -> (spd(ego_state,
    0)==0 U light_color==GREEN));
```

where the status of a traffic light in a scene is described by its ID (`light_id`), color (`light_color`), and distance to the other side of the intersection (`light_offset`), the distance between the ego vehicle and the light is described by `dis(ego_state, light_id)`, and `rho` is a predefined threshold. The third one is for performance-related assertion. Sometimes, to guarantee motion smoothness and passenger comfort, a sharp changing on the motion is not encouraged. Hence, we can declare a performance assertion as follows:

```
acc_dis = acc(ego_state, (0, 0, 0));
trace |= G acc_dis<=threshold;
```

where `threshold` is a predefined requirement.

We acknowledge that incorporating triggers can enhance predictability and expedite testing [40]. However, adding

TABLE 3: Comparison of Main Features of `AVUnit` with Other DSLs

Feature		`AVUnit`	Scenic	OpenScenario	GeoScenario	SceML	Paracosm	M-SDL
File Format		script	script	XML	XML	graph	script	script
Time		√	√	√	√	−	√	√
Weather		√	√	√	√	−	p	√
Vehicle Model		√	×	√	√	p	√	√
Position	Coordinate Position	√	√	√	p	−	√	×
	Distance Position	√	×	√	×	×	√	√
Heading	absolute value	√	√	√	√	×	−	√
	agent-relative value	√	√	√	×	×	−	√
	lane-relative value	√	√	√	×	×	−	√
Motion Behaviour	Format	individual trajectory	simulator-related event triggers	action + trigger	position profile + speed profile + reactive triggers	abstract behaviour	reactive event trigger	reactive event trigger
	Coupled vs. Decoupled	decoupled	coupled	coupled	coupled	coupled	coupled	coupled
	Simulator Independence	√	×	√	√	×	√	√
	Mutation Supported	√	×	×	×	×	×	×
Customized Oracles		√	×	×	×	×	p	√

√: supported; ×: not supported; −: unknown; p: partially support.

triggers to the description may limit certain agent behaviors, thereby making it impossible to depict certain specific scenarios. Hence, in this work, we focus on the design the automatic ADS testing framework. In the future, this framework can be extended to integrate triggers as an option in the scenario description language, and facilitate trigger-based testing approaches to generating critical scenarios.

6.2 Effectiveness of `AVUnit`

Based on our experiments, we find that there are rendering delays in LGSVL during the generation of LiDAR's point clouds, which means the current point cloud of an object is its previous state rather than the current one. In this case, the ADS cannot make correct decisions, and collisions occur easily. Hence, in our experiments, we bypass the perception module and use the ground truth data provided by LGSVL's modular testing feature.

According to the target scenario described in each `SCENEST` script, we generate an initial population whose size is 20, and the number of generations is 25, resulting in 520 test cases. We run the algorithm 5 times for each scenario. Moreover, to avoid the influence of execution uncertainties of the computer, we repeat the violated test cases several times to confirm that the failures are reproducible. For each violated test case, we analyse its execution and identify the unreasonable behaviour during the motion of the ego vehicle. However, it is not a root cause analysis, which is a challenging task due to the complexity of the ADS. It will be our future work. In the sequel, we describe the issues discovered in our experiments. Note that there are issues caused by other aspects, such as the simulator (e.g., loss of traffic light signals) and HD map (too large lane gaps). Hence, we manually check the failure test cases and identify the following 19 issues caused by Apollo.

Issues in Intersection Crossing. First, we describe the issues discovered during the crossing of intersections.

1) Wrong overtaking action. When an NPC vehicle is passing through the same intersection, Apollo predicts that the ego vehicle can pass through the intersection, so it generates an overtaking action, and the ego vehicle does not decelerate, resulting in a collision. Actually, the ego vehicle should slow down its motion and give way to the NPC vehicle.
2) No motion prediction of low-speed vehicles. Apollo regards a low-speed NPC vehicle as a stationary one and has no motion prediction. Hence, Apollo cannot make a decision to avoid collisions with the vehicle near its junction lane, resulting in a collision.
3) Too soft braking in emergencies. When there is an NPC vehicle passing across the ego vehicle's junction lane near the ego vehicle, Apollo does not take a 100% brake and causes a collision with the NPC vehicle. If Apollo applied more hard braking, the ego vehicle would stop before collisions.
4) Collision with a temporary stopping vehicle in an intersection due to aggressive motion decisions. Apollo first stops the ego vehicle safely when it detects an NPC vehicle moving to its junction lane. However, the NPC vehicle may stop temporarily since its lane is occupied by another NPC vehicle. Even though the stopping time is short, Apollo restarts the ego vehicle immediately and causes a collision with the NPC vehicle.
5) Rear-end collision with a large vehicle (e.g., SchoolBus and BoxTruck). When a large NPC vehicle makes a sharp turn, it may occupy a part of the adjacent lane. However, Apollo does not detect such a vehicle and keeps moving forward, resulting in a rear-end collision.
6) Wrong motion prediction of NPC vehicles. Apollo applies deep models to predict the motion of other agents. When there are multiple paths for an NPC vehicle, the prediction module may incorrectly predict the vehicle's motion, which would not cause collisions with the ego vehicle. Hence, the ego vehicle keeps crossing the

intersection and collides with the NPC vehicle.

7) Blocked in an intersection due to accidents. Apollo cannot detect accidents in an intersection and controls the ego vehicle moving into the intersection. As a result, the ego vehicle is blocked in the intersection and then stops other vehicles. Hence, the intersection is completely jammed.

Issues in Lane Changing. Another common action for vehicles is lane-changing behaviours, including cut-in. Note that overtaking can be decomposed into lane changing, which starts the overtaking task, and cut-in, which aims to complete the overtaking task. In the sequel, we describe the lane-changing related issues discovered in Apollo.

8) No deceleration to wait for lane changing or cut-in. Since other vehicles are moving on the target lane during lane changing, the ego vehicle cannot perform lane changing directly. Instead, it then follows the NPC vehicles, rather than slow down to wait for a long distance to perform lane changing, failing to reach the destination.
9) Obstacles ahead of the destination blocking the ego vehicle. Due to the fixed configurations, the ego vehicle stops before the destination when there is a stationary vehicle ahead of the destination, even though it is safe enough for the ego vehicle to reach the destination.
10) Lane changing or overtaking terminated halfway. When the ego vehicle detects a slow-speed NPC vehicle on the target lane, the ego vehicle terminates lane changing and switches to following, resulting in failing to arrive at the destination. In these situations, the ego vehicle is expected to continue its current lane-changing action.
11) Stuck during lane changing or overtaking. When the ego vehicle performs a sharp turn for lane changing or overtaking, it is forever stuck between two adjacent lanes.
12) Aggressive lane following during lane changing. Before performing lane changing, the ego vehicle first performs lane following. However, the ego vehicle follows the NPC vehicle ahead too aggressively such that when the NPC vehicle stops, the ego vehicle moves too close to perform lane changing. Hence, the ego vehicle stops forever.
13) Performing lane changing or overtaking wrongly. The ego vehicle starts lane changing or overtaking too late such that the ego vehicle cannot finish lane changing or overtaking successfully but blocks the adjacent lane.
14) No lane changing before an intersection when the ego vehicle is required to go through an intersection. Apollo prefers lane following and cannot perform lane changing before crossing an intersection. Hence, when the destination is near the intersection or an NPC vehicle is stopping near the intersection, the ego vehicle cannot perform lane changing or overtaking after crossing the intersection, resulting in task failures.
15) Performing cut-in wrongly. Apollo selects the wrong position, where there is an NPC vehicle, for cut-in to complete overtaking actions, which will stop the ego vehicle or cause collisions.
16) Cut-in terminated halfway and a new overtaking action restarted. During the end of overtaking, the ego vehicle takes cut-in to complete overtaking. When it detects another vehicle ahead, the ego vehicle terminated cut-in and restarts to overtake. However, the current position is close to the destination, and the ego vehicle cannot finish overtaking successfully, resulting in task failures. In this case, the ego vehicle is expected to continue cut-in and reach the destination successfully.
17) Wrong direction to perform overtaking. In this issue, the ego vehicle executes overtaking by moving along the lane with an opposite lane direction, while the adjacent lane with the same lane direction is empty. Such motion is forbidden by traffic laws.

TABLE 4: Issue Distribution in Scenarios

Scenario	Issues	
	Alg. 1	Rand
S1	(1)–(5)	(1), (4), (5)
S2	(2)	∅
S3	(2), (3), (6), (7)	(6), (7)
S4	∅	∅
S5	(7)	∅
S6	(8), (10), (15)	(8), (10)
S7	(9)–(14), (19)	(9), (12), (14)
S8	(10)–(13), (15)–(19)	(12), (13), (15), (19)

The above issues are safety-related and discovered by the design test oracles. During our experiments, we also identify some performance issues that are not because they are encoded into the test oracles.

18) Performing overtaking failed and returning to lane following. Sometimes, the ego vehicle first tries to overtake and then goes back to perform lane following. Even though such behaviour may not cause collisions or task failures, it can block the vehicles moving along the adjacent lane temporally and degrade the throughput of the whole transport system.
19) No overtaking when there is a slow vehicle ahead. The ego vehicle follows a slow-speed ($<$0.5m/s) vehicle and does not perform overtaking, which takes a long time to complete the motion task and degrades performance.

For example, as described in the last issue, if the ego vehicle does not overtake a slow NPC vehicle ahead, it may not complete the motion tasks in time and cause traffic jams. A well-designed ADS is expected to perform overtaking in such a scenario. It is worth noting that different scenarios may show different issues. The distribution of these issues in the scenarios is shown in Table 4. The videos for these failures can be found at https://avunit2021.github.io/.

6.3 Efficiency of `AVUnit`

Currently, no existing approaches support all the features required to run these experiments. For example, the authors in [41] proposed a search-based method for ADS collision avoidance testing. However, their open-source code requires significant efforts on configuration and customization to execute our target scenarios on the simulation platform (which makes it practically infeasible to compare with). Hence, we apply the random fuzzing approach as a baseline comparison with our approach. For each scenario, we repeat

the random method 5 times and generate 520 test cases for each run. We compare the performance of the two methods from two aspects: the number of failure test cases and discovered issues, and the experimental time. The experimental time includes generation time and execution time, where the former is the time to generate a new scenario and the latter is the time to run the scenario in the simulation environment.

Analysis of Failure-Inducing Algorithm. First, we compare our failure-inducing algorithm (i.e., Algorithm 1), denoted as Alg. 1, with the random method in terms of the numbers of violated test cases, failed test cases, and discovered issues, and the experimental time. Here a violated test case is a test case that violates the defined assertion ϕ, which may not cause system failure but due to the infeasibility of the test case (e.g., the destination of the ego vehicle is occupied, the intersection is blocked due to accidents among NPC vehicles, etc.), while a failed test case is a test case that triggers ADS's failures. Note that to identify the failed test cases, we currently confirm the violated test cases manually. Table 5 shows the number of violated and failed test cases for each execution round of the eight general scenarios. Fig. 7 presents the total number of discovered issues in each general scenario, and the detailed issues are listed in Table 4. The results show that Algorithm 1 can identify a more diverse set of failure cases and cover all issues discovered by the random. Table 6 displays the generation time and execution time. The results show that there are no significant differences between the two methods in terms of generation time. Even though some invalid violated test cases may affect the execution of Algorithm 1, the results still show the efficiency of our method.

We also compare the reported issues by AV-Fuzzer and ComOpt in their respective publications [41] and [42]. In [41], the authors proposed a method that combines GA-based global search and local fuzzing search to discover five types of issues related to wrong motion prediction of NPC vehicles (similar to Issue (6)), incorrect cut-in behavior (similar to Issue (15)), and overly soft braking (similar to Issue (3)). However, the trajectories that led to the discovered issues are different. For example, AV-Fuzzer reports that the ADS cannot predict the cut-in behavior of an NPC vehicle, while in our paper, we found that the ADS cannot predict the motion direction of an NPC vehicle in an intersection. This observation motivates us to explore more diverse scenarios and improve our mutation operators in the future. In [42], the authors considered different value combinations of parameters to describe a scenario and reported 12 classes of failure test cases. After manually reviewing the videos, we found that some issues were caused by the simulator (such as loss of signal of a traffic light by the simulator) and map issues (such as lanes being too far apart to allow for a lane change), and we were able to identify issues such as failure at sharp turns, oscillating motion (similar to Issues (16)), and aggressive lane changing (similar to Issue (15)). We found that our method can discover more issues but misses the sharp turn issue, which is dependent on the map structure. This observation encourages us to run our algorithms on more diverse map regions.

Analysis of Failure-Coverage Fuzzing Algorithm. The comparison of the failure coverage of our GA-based fuzzing algorithm (i.e., Algorithm 2, denoted as Alg. 2) and the random algorithm is shown in Table 7, where the second column is the total number of failure predicates in the assertion, and the last five columns are the covered failure predicates in each run. The results show that our algorithm outperforms the random method as our method covers more failure predicates in each scenario. Note that in Table 7, we do not distinguish invalid violated scenarios as our algorithm takes the failure predicates covered by both valid and invalid violated scenarios. Table 6 shows the comparison of experimental time. From the results, we can find Algorithm 2 takes shorter time to generate a new test case on average. The reason is that Algorithm 2 spends less time in the selection process (Lines 18-24 in Algorithm 2).

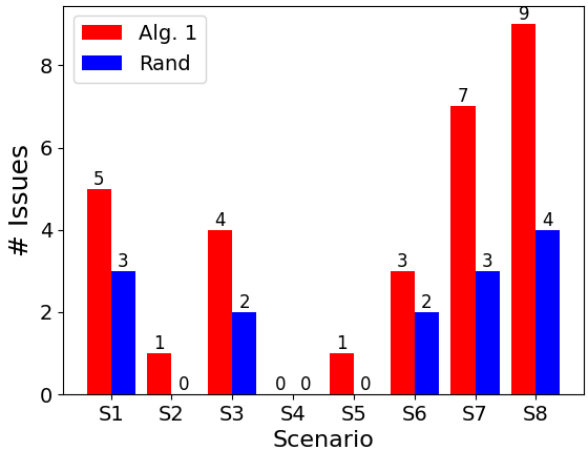

Fig. 7: Numbers of issues discovered in different scenarios.

6.4 Threats to Generality and Validity

Due to the nature of simulation-based ADS testing, `AVUnit` suffers some threats to generality and validity.

6.4.1 Threats to Generality

`AVUnit` is designed to be independent of simulation environments. However, we have to implement the corresponding simulator and ADS adapters and bridges if we want to use some specific simulators and ADSs, as different simulators and ADSs release different APIs and messages. On the one hand, as all elements defined in `AVUnit` (e.g., weather, time, and states of NPC vehicles and pedestrians) are general and should be supported by most of the current simulators and ADSs, the implementation of such adapters does not require much effort. On the other hand, the availability of the connection between a simulator and an ADS will limit the selection of simulation environments, which in turn will limit the application of `AVUnit`. For example, the simulator LGSVL provides bridges for both Apollo and Autoware, while CARLA only provides a bridge for CARLA. Therefore, if we want to set up a simulation environment with CARLA and Apollo, we must implement the corresponding bridge. However, it is out of the scope of `AVUnit`.

TABLE 5: Numbers of Violated Test Cases and Test Cases Causing Failures

Scenario	Method	# Failed/Violated Test Cases (out of 520 Test Cases)				
		R1	R2	R3	R4	R5
S1	Alg. 1	59/59	50/55	57/65	48/81	66/108
	Rand	3/7	2/2	5/5	13/13	5/6
S2	Alg. 1	1 /1	0/0	3/4	0/0	0/0
	Rand	0/0	0/0	0/0	0/0	0/0
S3	Alg. 1	57/57	48/48	69/69	54/54	62/62
	Rand	23/23	24/24	6/6	16/27	21/22
S4	Alg. 1	0/0	0/0	0/0	0/0	0/0
	Rand	0/0	0/0	0/0	0/0	0/0
S5	Alg. 1	19/20	14/45	6/39	4/37	29/100
	Rand	0/0	0/3	0/0	0/1	0/1
S6	Alg. 1	133/161	234/253	175/227	179/259	194/228
	Rand	18/27	16/26	17/31	23/34	12/17
S7	Alg. 1	116/156	149/316	131/392	66/125	99/135
	Rand	49/49	51/87	52/72	52/78	49/65
S8	Alg. 1	131/192	122/199	87/193	125/175	79/128
	Rand	58/172	49/160	65/165	66/174	52/160

TABLE 6: Average Generation and Execution Time for Each Scenario

Scenario	Time	Alg. 1	Random	Alg. 2
S1	Generation	0.34	0.3	0.19
	Execution	44.9	25.5	37.6
S2	Generation	0.32	0.23	0.18
	Execution	49.7	59.1	51.4
S3	Generation	0.34	0.29	0.19
	Execution	41.0	36.7	34.9
S4	Generation	0.32	0.29	0.19
	Execution	43.3	42.0	39.1
S5	Generation	0.35	0.31	0.2
	Execution	83.5	63.0	63.8
S6	Generation	0.31	0.29	0.18
	Execution	47.0	29.6	41.1
S7	Generation	0.3	0.29	0.19
	Execution	83.7	57.9	69.5
S8	Generation	0.3	0.28	0.17
	Execution	98.6	99.0	96.1

TABLE 7: Failure Coverage of Different Methods

Scenario	# predicates	Method	R1	R2	R3	R4	R5
S1	31	Alg. 2	9	9	8	9	9
		Rand	3	1	4	6	4
S2	31	Alg. 2	2	1	1	2	3
		Rand	0	0	0	0	0
S3	63	Alg. 2	3	4	5	4	6
		Rand	3	2	2	2	3
S4	31	Alg. 2	1	1	1	1	2
		Rand	0	0	0	0	0
S5	255	Alg. 2	5	4	4	5	7
		Rand	0	1	0	1	1
S6	63	Alg. 2	14	10	9	13	13
		Rand	4	3	3	4	3
S7	63	Alg. 2	7	9	6	10	6
		Rand	3	4	2	3	1
S8	15	Alg. 2	9	10	11	11	11
		Rand	4	5	5	7	6

6.4.2 Threats to Validity

External threats are mainly caused by simulators, HD maps, and execution computers. First, due to its design purposes, sometimes a simulator may not support all the features described in `AVUnit`. For example, we observed that the waypoint controller used in LGSVL caused NPC vehicles to move along straight lines to reach the next waypoints, which sometimes resulted in vehicles moving onto sidewalks instead of staying on the appropriate lane segments. To address this limitation, we modified the waypoint controller by adding additional waypoints on the lanes connecting any two consecutive waypoints. These new waypoints were placed such that they maintained a constant distance from one another, ensuring that NPC vehicles would always move along the lanes and stay on the correct path. Second, the quality of the HD maps will affect the evaluation of `AVUnit`. Currently, many lane gaps in the San Francisco map are not exact and too large for Apollo to perform lane changes. It results in failures for scenarios where lane changes are necessary. To mitigate this problem, we first check where the ego vehicle can perform a lane change between two lanes before creating scenarios in a region. Third, some modules of Apollo may suffer from high delays after long-time execution, which may result in the wrong execution of a scenario and cause AVUnit to conclude false-positive results. We implemented some solutions to mitigate this threat: (1) We used some high-performance computers to conduct our experiments; (2) During our experiments, we restarted the modules periodically to reduce such delays; (3) We replayed the failure scenarios several times to ensure that such failures can be reproduced.

The internal threats are generated by invalid failure test cases and the number of general scenarios. First, as `AVUnit` identifies violated test cases based on the last state of the ego vehicle, there may be failures caused by other reasons, such as simulator issues, infeasible scenarios, and NPC vehicles. They may affect the GA process of `AVUnit`. However, based on our experimental results, such effects do not cause serious consequences. In the future, we will design methods to filter out invalid failure test cases, which is a challenging task due to the unpredictable and complex executions of the scenarios. Second, we observed that the types of violations detected by our method are influenced by the topology and geometry characteristics of the map. In our experiments, we evaluated our method on a single map and instantiated each general scenario in a single region of that map. While our experimental results demonstrate the effectiveness of our method under these settings, we plan to test our method on more diverse map regions and maps to further validate its performance.

7 Related Work

Scenario Description Languages. There are some scenario description languages proposed to formulate complex scenarios [5]–[11].

CommonRoad [5] provides composable benchmarks for motion planning on roads, containing vehicle models, cost functions, and scenarios. Scenarios are described by an XML file, which includes the road network in terms of lanelets [43], static and dynamic obstacles, and the planning task of the ego vehicle. However, built on top of the predefined elements in the benchmarks, CommonRoad is hard to describe various simulation applications out of the benchmarks.

OpenScenario [6] is a standard, managed by the Association for Standardization of Automation and Measuring Systems, to describe autonomous driving scenarios. They define concepts such as driver behaviour, traffic, weather, environmental events, and other features, to construct scenarios, which are organized in an XML file format. The scenarios are described in terms of stories, which allow users to organize scenario elements into a higher-level hierarchy, but this can still be quite overwhelming. Besides, OpenScenario allows users to formulate the ADS behaviour in the scenario, which limits—even forbids—its applications to different ADSs.

GeoScenario [7] is a formal language designed according to OpenScenario. It has two basic concepts: Nodes and Ways. GeoScenario can be used to describe different agents in a scenario, such as vehicles and pedestrians, and their paths. GeoScenario focuses more on the planning and control modules, so it does not include the description of environmental elements, such as weather and time of a day. Hence, it can be applied only for the testing of planning and control parts of an ADS. Moreover, to describe a path, GeoScenario needs to define a set of nodes, which sometimes is overwhelming.

Scenic [8] is a probabilistic programming language that is primarily designed to generate a set of scenes for analysis of the ADS's perception module. Later, the authors extend it to express scenarios by adding behaviours for dynamic agents. A behaviour is a sequence of actions provided by a simulator, and users have to define a function to describe the transitions among the actions. Hence, the description of agent behaviours relies on simulators and is not user-friendly.

SceML [9] proposes a graphical scenario modeling language, where a scenario is described by a set of maneuver, condition, joint nodes. Each node is associated with a set of parameters to describe concrete scenarios. However, determining the nodes and their parameters is still a challenging task. Paracosm [10] provides a programmable language to describe scenarios where reactive event triggers define an object's dynamics. CRISCE analyzes the accident sketches and generates driving simulations that accurately reproduce the represented car crashes in a virtualization manner [44].

Unlike existing DSLs that mainly focus on describing complex scenarios, SCENEST prioritizes testing needs. It models each agent independently for the initialization of scenarios and ensures the ability to create new scenarios. Besides, the existing DSLs cannot customize oracles for ADS testing. In this paper, aiming to provide a platform for automatic ADS testing, we propose `AVUnit` to initialize scenarios following a description template, while leaving `AVUnit` to generate critical scenarios with customized specifications.

Scenario-Based ADS Testing. The main target for ADS testing is to evaluate whether an ADS can make proper decisions for different scenarios. However, due to the complex environments, a scenario consists of various configurations, resulting in the exponential increase of concrete scenarios since we need to parameterize all possible configurations to generate concrete scenarios. To reduce the number of scenarios, researchers focus on generating critical scenarios: in particular, the scenarios that can cause collisions.

The first group of methods is to generate critical scenarios from traffic accident reports [12]–[16] and real-world driving data [17]–[20]. For example, some methods have been proposed to extract collision scenarios from off-the-shelf datasets such as crash reports and human driving data [14], [15], [19]. Using deep learning technologies, we can increase data diversity and accelerate the generation of critical scenarios from existing datasets [17] or dashcam crash videos [12]. However, in these datasets, crash cases are rare, and many of them are highly similar, so it is costly to generate sufficient and diverse testing scenarios [20].

The second group of methods for critical scenario generation is based on ADSs' abstract models [21]–[25]. With the system models or surrogate models, one can generate critical scenarios via model evaluation using different technologies, such as optimization methods [21], [22]. However, in most cases, it is not easy to obtain the exact models of ADSs due to their complexity and confidentiality, while the surrogate models sometimes cannot describe the ADSs under test exactly.

Recently, simulation-based black-box testing technologies have been proposed to generate critical scenarios via the execution of an existing ADS [26], [27], [32], [41], [42], [45]–[47]. For example, Tuncali *et al* propose a framework Sim-ATAV to evaluate DNN-based ADSs with some predefined STL-based requirements [27]. The authors in [41] proposed a Genetic Algorithm-based method for ADS testing, where critical scenarios are generated by mutating the maneuvers of NPC vehicles, guided by the minimal feasible distance between the ego vehicle and other objects. In [42], the authors propose a method to generate diverse scenarios in terms of the value combinations of parameters, which, however, cannot generate critical scenarios efficiently. All these existing testing technologies focus on weak oracles and mutation operators without considering testing coverage. In this work, we propose `AVSpec` to customize specifications for ADS testing, and feedback-guided fuzzing algorithms to automatically generate failure scenarios with rich mutation operators and measure test sufficiency with different coverage measures (i.e., testing time and failure coverage).

8 Conclusion

In this paper, we proposed a new framework, `AVUnit`, for autonomous vehicle testing. `AVUnit` is equipped with two languages and two fuzzing algorithms. First, we proposed `SCENEST` to describe scenarios and `AVSpec` to describe different specifications to be monitored during the execution

of the scenarios. Second, we proposed a failure-inducing fuzzing algorithm to search for scenarios violating the defined assertions and a failure-coverage fuzzing algorithm to search for scenarios that can violate different predicates in the assertions. We implemented and evaluated `AVUnit` on LGSVL+Apollo, discovering 19 issues in Apollo, which indicate that the current open-source version of Apollo does not work well in complex intersections and lane changing maneuvers.

In the future, we will evaluate `AVUnit` on more regions in the San Francisco map and other maps, such as those generated by [48], [49]. We will integrate `AVUnit` with more simulators and ADSs and test more specifications of ADSs. We also design and implement more methods in `AVUnit` to generate critical scenarios; for example, we may investigate how to use the change rate of robustness to generate failure test cases. Finally, we will investigate the methods for root cause analysis of the discovered issues.

Acknowledgment

We would like to thank the editors and the reviewers for improving this manuscript. This work was supported in part by the Ministry of Education, Singapore under its Academic Research Fund Tier 3 (Award ID: MOET32020-0004) and Academic Research Fund Tier 2 (Grant No. MOE-T2EP20120-0004), the NRF Investigatorship (NRF-NRFI06-2020-0001), and the National Natural Science Foundation of China (No. 62032010). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of the Ministry of Education, Singapore.

References

[1] T. Menzel, G. Bagschik, and M. Maurer, "Scenarios for development, test and validation of automated vehicles," in *2018 IEEE Intelligent Vehicles Symposium (IV)*, 2018, pp. 1821–1827.

[2] C. Neurohr, L. Westhofen, T. Henning, T. de Graaff, E. Möhlmann, and E. Böde, "Fundamental considerations around scenario-based testing for automated driving," in *2020 IEEE Intelligent Vehicles Symposium (IV)*, 2020, pp. 121–127.

[3] 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)*, 2020, pp. 1–6.

[4] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, "CARLA: An open urban driving simulator," in *Conference on Robot Learning*, 2017, pp. 1–16.

[5] M. Althoff, M. Koschi, and S. Manzinger, "CommonRoad: Composable benchmarks for motion planning on roads," in *2017 IEEE Intelligent Vehicles Symposium (IV)*, 2017, pp. 719–726.

[6] Association for Standardization of Automation and Measuring Systems (ASAM). (2020) ASAM OpenSCENARIO. [Online]. Available: https://www.asam.net/standards/detail/openscenario/

[7] R. Queiroz, T. Berger, and K. Czarnecki, "GeoScenario: An open DSL for autonomous driving scenario representation," in *2019 IEEE Intelligent Vehicles Symposium (IV)*, 2019, pp. 287–294.

[8] D. J. Fremont, T. Dreossi, S. Ghosh, X. Yue, A. L. Sangiovanni-Vincentelli, and S. A. Seshia, "Scenic: a language for scenario specification and scene generation," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 63–78.

[9] B. Schütt, T. Braun, S. Otten, and E. Sax, "SceML: A graphical modeling framework for scenario-based testing of autonomous vehicles," in *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, 2020, pp. 114–120.

[10] R. Majumdar, A. Mathur, M. Pirron, L. Stegner, and D. Zufferey, "Paracosm: A test framework for autonomous driving simulations," in *International Conference on Fundamental Approaches to Software Engineering*. Springer, Cham, 2021, pp. 172–195.

[11] Foretellix Ltd, "Measurable Scenario Description Language," https://releases.asam.net/OpenSCENARIO/2.0-concepts/M-SDL_LRM_OS.pdf, Jan. 2020.

[12] S. K. Bashetty, H. B. Amor, and G. Fainekos, "DeepCrashTest: Turning dashcam videos into virtual crash tests for automated driving systems," in *2020 IEEE International Conference on Robotics and Automation, ICRA*, Paris, France, 2020, pp. 11 353–11 360.

[13] A. Gambi, T. Huynh, and G. Fraser, "Generating effective test cases for self-driving cars from police reports," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 257–267.

[14] W. G. Najm, S. Toma, J. Brewer *et al.*, "Depiction of priority light-vehicle pre-crash scenarios for safety applications based on vehicle-to-vehicle communications," National Highway Traffic Safety Administration, U.S. Department of Transportation, Washington, DC, Tech. Rep. DOT HS 811 732, Apr. 2013.

[15] P. Nitsche, P. Thomas, R. Stuetz, and R. Welsh, "Pre-crash scenarios at road junctions: A clustering method for car crash data," *Accident Analysis & Prevention*, vol. 107, pp. 137–151, 2017.

[16] F. Hauer, T. Schmidt, B. Holzmüller, and A. Pretschner, "Did we test all scenarios for automated and autonomous driving systems?" in *2019 IEEE Intelligent Transportation Systems Conference (ITSC)*, 2019, pp. 2950–2955.

[17] W. Ding, M. Xu, and D. Zhao, "CMTS: A conditional multiple trajectory synthesizer for generating safety-critical driving scenarios," in *2020 IEEE International Conference on Robotics and Automation (ICRA)*, 2020, pp. 4314–4321.

[18] C. Roesener, F. Fahrenkrog, A. Uhlig, and L. Eckstein, "A scenario-based assessment approach for automated driving by using time series classification of human-driving behaviour," in *2016 IEEE 19th international conference on intelligent transportation systems (ITSC)*, 2016, pp. 1360–1365.

[19] J.-P. Paardekooper, S. Montfort, J. Manders, J. Goos, E. d. Gelder, O. Camp, O. Bracquemond, and G. Thiolon, "Automatic identification of critical scenarios in a public dataset of 6000 km of public-road driving," in *26th International Technical Conference on the Enhanced Safety of Vehicles (ESV)*, 2019.

[20] D. Zhao, H. Lam, H. Peng, S. Bao, D. J. LeBlanc, K. Nobukawa, and C. S. Pan, "Accelerated evaluation of automated vehicles safety in lane-change scenarios based on importance sampling techniques," *IEEE Transactions on Intelligent Transportation Systems*, vol. 18, no. 3, pp. 595–607, 2017.

[21] M. Althoff and S. Lutz, "Automatic generation of safety-critical test scenarios for collision avoidance of road vehicles," in *2018 IEEE Intelligent Vehicles Symposium (IV)*, 2018, pp. 1326–1333.

[22] M. Klischat and M. Althoff, "Generating critical test scenarios for automated vehicles with evolutionary algorithms," in *2019 IEEE Intelligent Vehicles Symposium (IV)*, 2019, pp. 2352–2358.

[23] H. Beglerovic, M. Stolz, and M. Horn, "Testing of autonomous vehicles using surrogate models and stochastic optimization," in *2017 IEEE 20th International Conference on Intelligent Transportation Systems (ITSC)*, 2017, pp. 1–6.

[24] G. E. Mullins, P. G. Stankiewicz, and S. K. Gupta, "Automated generation of diverse and challenging scenarios for test and evaluation of autonomous vehicles," in *2017 IEEE international conference on robotics and automation (ICRA)*, 2017, pp. 1443–1450.

[25] G. E. Mullins, A. G. Dress, P. G. Stankiewicz, J. D. Appler, and S. K. Gupta, "Accelerated testing and evaluation of autonomous vehicles via imitation learning," in *2018 IEEE International Conference on Robotics and Automation (ICRA)*, 2018, pp. 1–7.

[26] T. Dreossi, D. J. Fremont, S. Ghosh, E. Kim, H. Ravanbakhsh, M. Vazquez-Chanlatte, and S. A. Seshia, "VerifAI: A toolkit for the formal design and analysis of artificial intelligence-based systems," in *International Conference on Computer Aided Verification*, 2019, pp. 432–442.

[27] C. E. Tuncali, G. Fainekos, D. Prokhorov, H. Ito, and J. Kapinski, "Requirements-driven test generation for autonomous vehicles with machine learning components," *IEEE Transactions on Intelligent Vehicles*, vol. 5, no. 2, pp. 265–280, 2019.

[28] Baidu, "Apollo 6.0," https://github.com/ApolloAuto/apollo/releases/tag/v6.0.0, 2019, online; accessed Oct 2020.

[29] O. Maler and D. Nickovic, "Monitoring temporal properties of continuous signals," in *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, 2004, pp. 152–166.
[30] J. V. Deshmukh, A. Donzé, S. Ghosh, X. Jin, G. Juniwal, and S. A. Seshia, "Robust online monitoring of signal temporal logic," *Formal Methods in System Design*, vol. 51, no. 1, pp. 5–30, 2017.
[31] D. Ničković and T. Yamaguchi, "RTAMT: Online robustness monitors from STL," in *International Symposium on Automated Technology for Verification and Analysis*, 2020, pp. 564–571.
[32] C. E. Tuncali, G. Fainekos, H. Ito, and J. Kapinski, "Simulation-based adversarial test generation for autonomous vehicles with machine learning components," in *2018 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 2018, pp. 1555–1562.
[33] C. E. Tuncali, J. Kapinski, H. Ito, and J. V. Deshmukh, "Reasoning about safety of learning-enabled components in autonomous cyber-physical systems," in *Proceedings of the 55th Annual Design Automation Conference*, 2018, pp. 1–6.
[34] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler, "The fuzzing book," 2019.
[35] P. Arcaini, X.-Y. Zhang, and F. Ishikawa, "Targeting patterns of driving characteristics in testing autonomous driving systems," in *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2021, pp. 295–305.
[36] E. Thorn, S. C. Kimmel, M. Chaka, B. A. Hamilton *et al.*, "A framework for automated driving system testable cases and scenarios," United States. Department of Transportation. National Highway Traffic Safety, Tech. Rep., 2018.
[37] S. Wang and Z. Su, "Metamorphic object insertion for testing object detection systems," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020, pp. 1053–1065.
[38] C. W. Lee, N. Nayeer, D. E. Garcia, A. Agrawal, and B. Liu, "Identifying the operational design domain for an automated driving system through assessed risk," in *2020 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 2020, pp. 1317–1322.
[39] Society of Automotive Engineers (SAE), "SAE J3016 – Taxonomy and definitions for terms related to driving automation systems for on-road motor vehicles," 2018.
[40] A. Arrieta, S. Wang, U. Markiegi, G. Sagardui, and L. Etxeberria, "Employing multi-objective search to enhance reactive test case generation and prioritization for testing industrial cyber-physical systems," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 3, pp. 1055–1066, 2017.
[41] G. Li, Y. Li, S. Jha, T. Tsai, M. Sullivan, S. K. S. Hari, Z. Kalbarczyk, and R. Iyer, "AV-Fuzzer: Finding safety violations in autonomous driving systems," in *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, 2020, pp. 25–36.
[42] C. Li, C.-H. Cheng, T. Sun, Y. Chen, and R. Yan, "ComOpT: Combination and optimization for testing autonomous driving systems," in *2022 International Conference on Robotics and Automation (ICRA)*. IEEE, 2022, pp. 7738–7744.
[43] P. Bender, J. Ziegler, and C. Stiller, "Lanelets: Efficient map representation for autonomous driving," in *2014 IEEE Intelligent Vehicles Symposium Proceedings*, 2014, pp. 420–425.
[44] A. Gambi, V. Nguyen, J. Ahmed, and G. Fraser, "Generating critical driving scenarios from accident sketches," in *2022 IEEE International Conference On Artificial Intelligence Testing (AITest)*. IEEE, 2022, pp. 95–102.
[45] A. Calò, P. Arcaini, S. Ali, F. Hauer, and F. Ishikawa, "Generating avoidable collision scenarios for testing autonomous driving systems," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, 2020, pp. 375–386.
[46] D. Kaufmann, L. Klampfl, F. Klück, M. Zimmermann, and J. Tao, "Critical and challenging scenario generation based on automatic action behavior sequence optimization: 2021 ieee autonomous driving ai test challenge group 108," in *2021 IEEE International Conference on Artificial Intelligence Testing (AITest)*. IEEE, 2021, pp. 118–127.
[47] A. Piazzoni, J. Cherian, M. Azhar, J. Y. Yap, J. L. W. Shung, and R. Vijay, "ViSTA: a framework for virtual scenario-based testing of autonomous vehicles," in *2021 IEEE International Conference on Artificial Intelligence Testing (AITest)*. IEEE, 2021, pp. 143–150.
[48] Y. Tang, Y. Zhou, K. Yang, Z. Zhong, B. Ray, Y. Liu, P. Zhang, and J. Chen, "Automatic map generation for autonomous driving system testing," *arXiv preprint arXiv:2206.09357*, 2022.
[49] Y. Zhou, G. Lin, Y. Tang, K. Yang, W. Jing, P. Zhang, J. Chen, L. Gong, and Y. Liu, "FLYOVER: A model-driven method to generate diverse highway interchanges for autonomous vehicle testing," *arXiv preprint arXiv:2301.12738*, 2023.

Yuan Zhou received his M.S. degree in computational mathematics from Zhejiang Sci-Tech University, Hangzhou, China, in March 2015 and received his Ph.D. degree in computer science from Nanyang Technological University, Singapore, in June 2019. He is currently a Research Fellow in School of Computer Science and Engineering at Nanyang Technological University, Singapore, where he was awarded the Research Scholarship Block Postdoctoral Fellow. His research interests focus on the safety and security of autonomous unmanned systems, including multi-robot systems and autonomous vehicles.

Yang Sun is currently a Ph.D. candidate in the School of Computing and Information System at Singapore Management University, supervised by Prof. SUN Jun. Yang's research focuses on evaluating and improving Autonomous Driving Systems.

Yun Tang received his B.Eng (1st class, Dean's list) from the School of Electrical & Electronic Engineering of Nanyang Technological University, Singapore in 2016 and his Ph.D. from the School of Computer Science and Engineering of Nanyang Technological University, Singapore in 2023. He is currently a Research Fellow in the Warwick Manufacturing Group, University of Warwick, United Kingdom since February 2023. His research interests focus on the safety verification and validation framework of cyber physical systems, such as intelligent vehicles.

Yuqi Chen is an Assistant Professor at the School of Information Science and Technology at ShanghaiTech University. He received his B.Sc. in computer science from the South China University of Technology in 2015 and his Ph.D. from the Singapore University of Technology and Design in 2019. Before joining ShanghaiTech, Yuqi was a Research Scientist in the System Analysis and Verification group at Singapore Management University. Yuqi's research interests lie at the intersection of software engineering and security. He employs a range of techniques, including testing, reverse engineering, program analysis, and formal methods, to develop practical solutions for securing critical cyber-physical systems.

Jun Sun is currently a professor at Singapore Management University (SMU). He received Bachelor and PhD degrees in computing science from National University of Singapore (NUS) in 2002 and 2006. In 2007, he received the prestigious LEE KUAN YEW postdoctoral fellowship. He has been a faculty member since 2010. He was a visiting scholar at MIT from 2011-2012. Jun's research interests include software engineering, formal methods, program analysis and cyber-security. He is the co-founder of the PAT model checker.

Christopher M. Poskitt is an Assistant Professor of Computer Science (Education) at Singapore Management University (SMU), where he is a member of the System Analysis and Verification Group. Prior to SMU, he held research and teaching positions at ETH Zürich, Switzerland, and the Singapore University of Technology and Design. His research broadly addresses the problem of engineering correct and secure software/systems, towards which he has co-developed techniques for testing/defending cyber-physical systems, tools for analysing concurrent programming models, and logics for reasoning about the correctness of graph-rewriting programs. His research interests span software engineering, formal methods, cybersecurity, and computer science education.

Yang Liu graduated in 2005 with a Bachelor's degree in Computer Science from the National University of Singapore (NUS). In 2010, he obtained his Ph.D. in Computer Science and continued with his post doctoral work at NUS.

He is currently a Full Professor at the School of Computer Science and Engineering, Nanyang Technological University, Singapore. His research focuses on software engineering, security, cyber-physical systems, and formal methods. Particularly, he specializes in software verification using model checking techniques, leading to the development of a state-of-the-art model checker, Process Analysis Toolkit (PAT).

Dr. Yang received his Ph.D. from the University of Pennsylvania. He is currently a professor and director of the Turing Interdisciplinary Information Science Research Center at Xi'an Jiaotong University. He founded Synkrotron Inc, a startup that focuses on autonomous driving software systems. Dr. Yang has published more than 100 papers and received the ACM SIGSOFT Outstanding Paper Award and the ACM TODAES Best Journal Paper Award. He is the co-chair of the IEEE Technical Committee on Electric and Autonomous Vehicles.