From Scenario Selection to Simulation: Safety Testing of an Automated Driving System

Fauzia Khan[1 [0000−0001−9942−8709]], Ali Ihsan Güllü[2 [0000−0001−5911−2726]], Hina Anwar[3 [0000−0002−4725−4636]], and Dietmar Pfahl[4 [0000−0003−2400−501X]]

University of Tartu, Narva mnt 18, 51009, Tartu, Estonia
{fauzia.khan,ali.ihsan.gullu,hina.anwar,dietmar.pfahl}@ut.ee

3.2 Scenario Implementation

To implement the *Follow Lead Vehicle* scenario, we followed the standard procedure provided in the CARLA `ScenarioRunner`[1] for creating custom scenarios. We developed a Python class that inherits from the `BasicScenario` base class. The base class defines how the lead vehicle behaves, controls the flow of the scenario, and sets up the evaluation criteria. We consider three input parameters that could affect the safety behavior of the ego vehicle: (i) ego vehicle speed v_r, (ii) lead vehicle speed v_f, and (iii) initial distance between the vehicles d_i. We use different combinations of speed and distance to assess how these parameters could affect the ego vehicle's ability to maintain a safe following distance and avoid collisions when the lead vehicle stops suddenly. The scenario implementation includes the following key methods:

- **`_initialize_actors()`**: This method sets up the lead vehicle. The lead vehicle is created using the `vehicle.nissan.patrol` blueprint. The initial distance d_i is set using the `get_waypoint_in_distance()` function to position the lead vehicle ahead of the ego vehicle. A vertical offset of 0.5 m is added to prevent ground collision.
- **`_create_behavior()`**: This method sets up the behavior of the lead vehicle using the `py_trees` library. To simulate the behavior of the lead vehicle, we model three phases: acceleration, constant velocity, and braking.

  1. Acceleration Phase: In the first phase, the lead vehicle starts from rest and accelerates gradually until it reaches the target speed (i.e., v_f). The acceleration is applied at a constant rate, following the equation:

$$v(t) = \min(v_{\text{target}}, a \cdot t) \tag{1}$$

  where a is the predefined acceleration and t is the time step. $v(t)$ is the desired speed at time t, and v_{target} is the maximum target speed the lead vehicle aims to reach. To ensure a smooth transition and reduce the impact of simulation discretization, we apply exponential smoothing using the following equation:

$$v_{\text{smoothed}} = v_{\text{current}} + \alpha \cdot (v(t) - v_{\text{current}}) \tag{2}$$

[1] CARLA ScenarioRunner

Here, α is a smoothing factor that controls how quickly the vehicle's speed adjusts toward the desired speed $v(t)$. A smaller α results in slower, smoother changes. v_{current} is the vehicle's current smoothed speed, which updates incrementally based on the difference from $v(t)$.

2. Constant Speed Phase: After reaching the target speed (i.e., v_f), the lead vehicle enters the constant-speed phase and continues moving with v_f. We use a 0.1 m/s tolerance to avoid unnecessary speed changes caused by minor fluctuations. During this phase, the lead vehicle maintains a steady speed for consistent motion by continuously setting its forward speed to the target speed using a control function.

3. Braking Phase: In this phase, the lead vehicle's braking behavior is controlled by a configurable parameter `braking_trigger_distance`. The parameter specifies how far the vehicle must travel before braking starts. Once the distance is reached, the lead vehicle decelerates smoothly to a stop through two stages:

- *Reaction Delay:* A reaction delay of t_r is introduced to simulate human-like behavior. During this delay, the lead vehicle travels at the target speed v_{target}.
- *Deceleration Phase:* After the delay, the lead vehicle starts braking at a constant deceleration rate, which is calculated using the equation:

$$a = \mu \cdot g \tag{3}$$

where μ is the road friction coefficient and g is the gravitational acceleration. Using this deceleration, the vehicle's speed decreases linearly over time, following:

$$v(t) = v_0 - a \cdot t \tag{4}$$

Here, v_0 is the velocity at the beginning of braking, and a is the deceleration value. Braking continues until the vehicle's speed drops below 0.2 m/s, which is considered stopped.

– **`_create_test_criteria()`**: This method sets up the evaluation metrics. To assess the safety of the *Follow-Lead-Vehicle* scenario, we define a single test criterion using the `CollisionTest` class provided by the `ScenarioRunner` suite. This test monitors the ego vehicle during the entire scenario execution and detects any collisions with the lead vehicle.
In addition, we perform a separate safety distance analysis to assess the ego vehicle's compliance with the Responsibility-Sensitive Safety (RSS) model, which is explained in Section 0.4.

3.3 Scenario Configuration

The scenario configuration file specifies simulation parameters such as the map, vehicle positions, and environmental conditions. We use the CARLA map `Town01`, positioning the ego vehicle on a straight 220-meter road. In addition, we configure the environmental settings for clear skies, no wind, and daylight with sun altitude at 75 degrees as shown below:

Scenario Configuration

```
<scenarios>
<scenario name="FollowLeadVehicle_1" type="FollowLeadVehicle" town="Town01
    ">
<ego_vehicle x="107" y="133" z="0.5" yaw="0" rolename="ego_vehicle"/>
<weather cloudiness="0" precipitation="0" precipitation_deposits="0"
    wind_intensity="0" sun_azimuth_angle="0" sun_altitude_angle="75"/>
</scenario>
```

3.4 Simulation Setup and Execution

We set up the simulation environment using the CARLA simulator (v0.9.13)[2], ScenarioRunner[3], and the ego vehicle (UT-ADS)[4], which uses the Autoware Mini autonomy stack based on ROS 1 Noetic. The ego vehicle operates in a controlled urban environment, following a closed-loop route through the city center of Tartu[5] under clear weather conditions.

We integrate the ego vehicle with CARLA through the `ros_bridge` interface, which enables real-time, bidirectional communication between the simulator and ROS. Sensor data from CARLA (e.g., camera, LiDAR, GPS) is published to ROS topics, which the ego vehicle uses for perception and planning. In response, the ego vehicle generates control commands (throttle, brake, steering) sent back to CARLA through the bridge. This setup allows the ego vehicle to perceive its environment, make driving decisions, and control its motion as it would in the real world.

We run simulations following these steps. First, the CARLA server is launched to provide the virtual driving environment. Next, the ego vehicle autonomy stack is started, which establishes the ROS bridge and configures adjustable parameters such as the map and the ego vehicle's speed. ScenarioRunner is then used to run the `Follow Lead Vehicle` scenario, spawning the ego and lead vehicles with the specified parameters.

Furthermore, we run all simulations on the hardware, which includes an 11th Gen Intel Core i7-11800H CPU (2.30 GHz, 16 threads), 32 GB of RAM, an NVIDIA GeForce RTX 3070 Laptop GPU with 8 GB of VRAM (CUDA 12.2), and 2.5 TB of storage. The simulation environment runs on Ubuntu 20.04.6 LTS (GNOME 3.36.8, X11) with Python 3.8.10. We installed all required packages via `pip` using CARLA's official dependency list and ran simulations at 20 frames per second.

3.6 Safety Evaluation Metrics

We evaluate two safety metrics: (i) collisions, and (ii) maintaining the minimum safe distance. We use the `CollisionTest()` method that the ScenarioRunner

[2] CARLA
[3] ScenarioRunner
[4] UT-ADS
[5] Tartu Loop: A demonstration track that passes through the city center of Tartu, Estonia.

framework provides to detect collisions. The method class generates a binary output: 0 if no collision occurs and one if at least one collision is detected. However, we use the RSS model to calculate the minimum safe distance. According to RSS, if the ego vehicle maintains at least d'_{min} distance from the lead vehicle, it will have enough time and space to stop without a collision, even in a worst-case emergency (see Figure **??**). The RSS formula is defined as:

$$d'_{\text{min}} = \max\left(0,\ v_r\rho + \frac{1}{2}a_{\text{max,accel}}\rho^2 + \frac{(v_r + \rho a_{\text{max,accel}})^2}{2a_{\text{min,brake}}} - \frac{v_f^2}{2a_{\text{max,brake}}}\right) \quad (5)$$

where: d'_{min} is the minimum required distance between the ego and the lead vehicle, v_f is the longitudinal velocity of the front (lead) vehicle,v_r is the longitudinal velocity of the ego vehicle, ρ is the response time delay of the ego vehicle before it starts braking, $a_{\text{max,accel}}$ is the maximum acceleration of the ego vehicle, $a_{\text{min,brake}}$ is the minimum braking capability of the ego vehicle, $a_{\text{max,brake}}$ is the maximum braking capability of the lead vehicle.

We then calculate the actual longitudinal distance between the ego vehicle and the lead vehicle using Eq. 6 at each time step to see whether the ego vehicle maintains a safe following distance as per the RSS model.

The actual longitudinal distance between the ego vehicle and the lead vehicle is defined by the formula:

$$\text{d} = \sqrt{(x_e - x_l)^2 + (y_e - y_l)^2 + (z_e - z_l)^2} \quad (6)$$

Where (x_e, y_e, z_e) and (x_l, y_l, z_l) represent the 3D coordinates of the ego and lead vehicles, respectively, at time t. We calculate the distance between both vehicles at every frame, with a sampling rate of 20 Hz (i.e., every 0.05 seconds).

Finally, we compare the actual distance d with the theoretically calculated RSS safety distance d'_{min} at each time step to see whether the ego vehicle maintains a safe following distance from the lead vehicle throughout the scenario.