Self-driving Simulator Test Scenario Framework with Event-triggered Functionality

Takuma Yabe[1,a)] Yuto Koyanagi[1] Keita Miura[1] Takuya Azumi[1]

Abstract: This paper proposes a testing framework with an event-triggered functionality to validate flexible test scenarios of self-driving systems in a virtual environment. Autonomous vehicles have been developed and tested worldwide, and virtual environments testing using simulators has become the mainstream driving test method for self-driving systems. However, the virtual environment testing is difficult to set up complex movements of vehicles and pedestrians, and to determine whether an ego vehicle is in dangerous states is difficult. The proposed framework provides the event-triggered functionality to verify the movements of vehicles and pedestrians flexibly. With the event-triggered functionality, the vehicle and pedestrian movement can be controlled by developers. The proposed framework consists of SVL simulator, and the simulator can be linked to Autoware (open-source self-driving software) and configured to move vehicles and pedestrians according to the position and speed of an ego vehicle. This relationship is used to obtain the position and speed values of the ego vehicle. Experimental results demonstrate the degree to which the event-triggered functionality affects simulation runtime.

Keywords: Self-driving System, Test Scenario, SVL Simulator, ROS, Autoware

1. Introduction

In recent years, the automotive industry has focused on planning, developing, and testing autonomous vehicles. The realization of autonomous vehicles is expected to contribute to the reduction of traffic congestion and accidents. In addition, deploying autonomous vehicles in rural areas where public transport is lacking is expected to ensure that older people have access to transportation, which will reduce the number of traffic accidents. Autonomous vehicles; therefore, have the potential to resolve complex problems in society.

Widespread use of autonomous vehicles requires a high-level of safety; thus, a large number of sensors are used in self-driving systems based on Robot Operating System (ROS) [1], which is a widely used middleware robot development tool. ROS is used to process sensor data from many sensors and is suitable for developing self-driving systems, e.g., Autoware [2], [3], open-source self-driving software.

To ensure the safety of self-driving systems, self-driving tests must be performed. Self-driving systems can be tested using two different approaches. One is to drive an actual vehicle in a real environment, (e.g., a public roadway or inside a facility). Real-world testing can be conducted under the same operating conditions as in production. However, real-world testing is expensive and time-consuming, and it is impossible to manipulate the external environment, such as non-ego vehicles, pedestrians, and the weather. The other is a testing method that exploits a virtual environment using a simulator such as SVL [4], CARLA [5], and CAT Vehicle Testbed [6]. Assuming developers have all required testing tools, e.g., a simulator, the developers can reduce testing costs and easily test infrequent or outlier traffic scenarios, e.g., accidents. Testing in the virtual environments with self-driving systems has become the norm in the automotive industry today.

Testing self-driving systems in a simulator have two problems. The first problem is the inability to define flexible movement settings for non-ego vehicles and pedestrians. This problem is caused by the fact that simulators do not consider the interrelationships among an ego vehicle, non-ego vehicles, and pedestrians. The second problem is the difficulty in visually assessing whether the ego vehicle is in a dangerous situation, e.g., a collision. These problems make it difficult to validate a wide range of test cases and obtain the desired feedback.

This paper proposes a test scenario framework with the event-triggered functionality to solve these problems. With the proposed framework, developers can test self-driving systems while controlling the non-ego vehicle and pedestrian movements using the SVL simulator. Currently, when developers simulate the movements of non-ego vehicles and pedestrians using the SVL simulator, the start and end points of their paths are set. The simulator automatically starts moving from the start point to the end point when the simulation begins. However, these paths cannot be changed or stopped after the simulation has begun, and thus developers can only configure simple movements, e.g., straight lines. Using the event-triggered functionality, these problems can be solved, and the paths can be changed along the way.

To realize the event-triggered functionality, the position and velocity states of an ego vehicle need to be incorporated into the movement conditions of non-ego vehicles and pedestrians. Controlling the movement of non-ego vehicles and pedestrians using the position and speed information of the ego vehicle is referred to as a "trigger." The proposed framework allows developers to verify flexible test scenarios using the SVL simulator, thus ex-

[1] Saitama University, Saitama-shi, Saitama-ken 338–8570, Japan
[a)] t.yabe.240@saitama-u.ac.jp

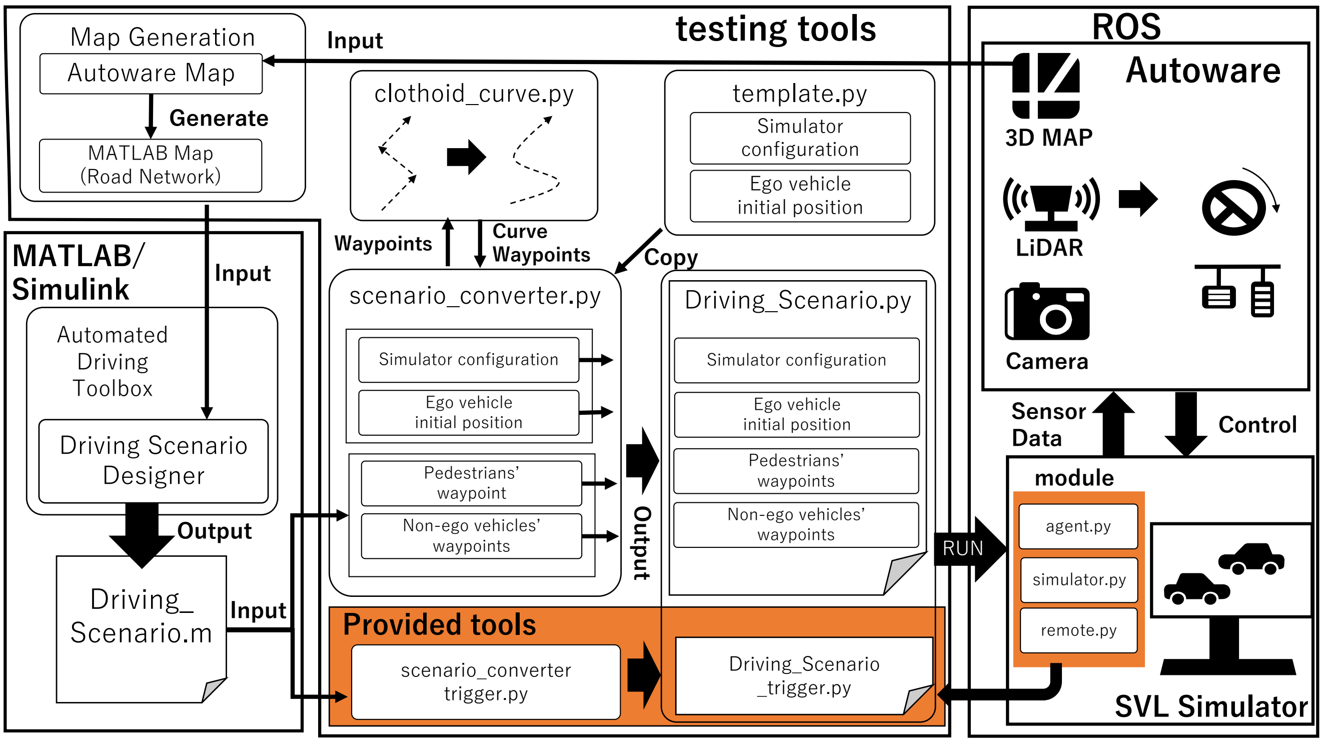


Fig. 1 Detail system model

panding the scope of verification. The SVL simulator represents the movements of non-ego vehicles and pedestrians using Python APIs.

The primary contributions of this paper are summarized as follows.

- The scope of scenarios that can be verified using the SVL simulator is expanded.
- Feedbacks are provided to the user in the event of a crash, thereby making it easier to confirm safety.
- The impact of the event-triggered functionality on the processing speed of Autoware and simulation runtime is reduced.

The remainder of this paper is organized as follows. Section 2 describes the proposed framework's system model. Section 3 presents the proposed framework and event-triggered functionality in detail. Section 4 describes an evaluation of the proposed framework, and Section 5 describes related work. Finally, Section 6 explains a brief conclusion.

2. System Model

This section introduces the proposed framework's system model with the event-triggered functionality as shown in **Fig. 1**. The event-triggered functionality is realized by allowing the trigger configuration to be described for non-ego vehicles and pedestrians in a scenario file. After setting the trigger configuration, the scenario file runs in the SVL simulator, and Autoware controlling the ego vehicle movement to reflect the behavior of the non-ego vehicles and pedestrians in a natural manner. The following introduces ROS, Autoware, the SVL simulator, and the Python APIs used to control the SVL simulator.

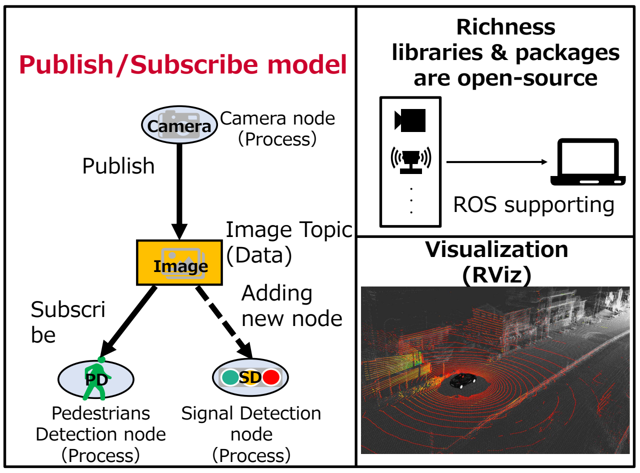


Fig. 2 Advantages of Robot Operating System (ROS)

2.1 Robot Operating System (ROS)

ROS is open-source middleware used for the robot development as shown in **Fig. 2**. ROS utilizes a *Publish/Subscribe communication model*. In this model, data are referred to as a *topic*, and a process is referred to as a *node*. *Topics* are pushed by publishing and received by subscribing. ROS handles large-scale processing by separating processes into *nodes* and communicating between *nodes*, which enables distributed processing.

In addition, ROS includes many development libraries and packages for the sensors and actuators, e.g., LiDAR, cameras, and inertial measurement units, which are used in self-driving systems. ROS includes RViz, a visualization tool that can display three-dimensional (3D) maps of the vehicle, and *topic* data such as the *image topic* and *point cloud topic*. As a result, real-time

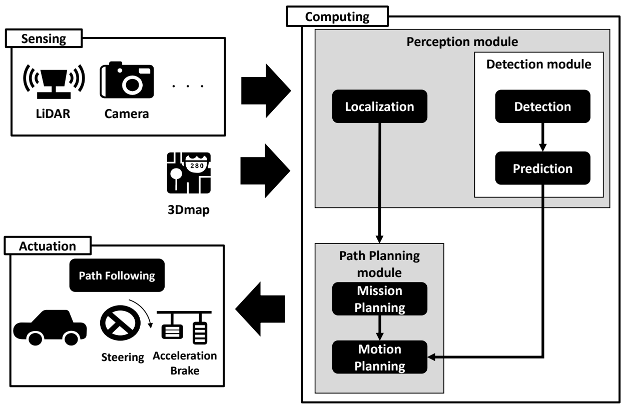


Fig. 3 Autoware flow

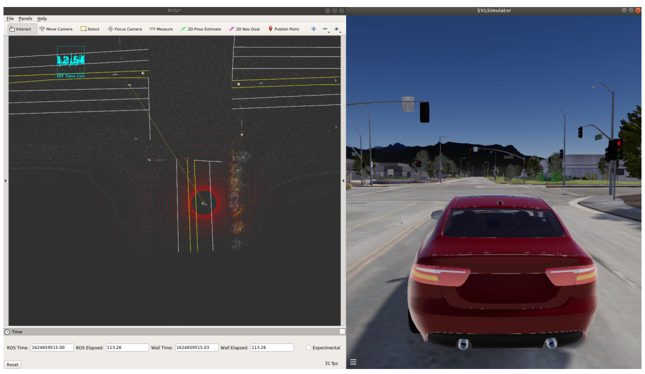


Fig. 4 Cooperation between SVL simulator and Autoware

evaluation of vehicle behavior and sensor data is possible.

2.2 Autoware

Autoware [2] is open-source self-driving software based on ROS. Autoware provides modules required for self-driving systems. In addition, Autoware can drive a real vehicle and perform virtual simulations with excellent visualization graphics.

Autoware comprises sensing, computing, and actuation modules, as shown in **Fig. 3**. The sensing module processes sensor data from LiDAR and cameras. This module also publishes *sensor topics*, e.g., *point-cloud topic* and *image topics*. The computing module comprises localization, detection, and planning modules. Here, the localization module compares a pre-stored 3D map to *point-cloud data* from the sensing module to determine and publish a matching point as the current position. The detection module detects surrounding objects, e.g., pedestrians, vehicles, and obstacles. The planning module generates a path from the current position to the destination, and the actuation module calculates the velocity and angular rate required to move along a path generated by the planning module. Autoware supports self-driving systems by processing these modules repeatedly.

2.3 SVL simulator

The SVL simulator is an open-source simulator provided by LG Electronics. The SVL simulator graphically recreates the virtual environment described in a scenario file. The SVL simulator

```
sim = lgsvl.Simulator(os.environ.get("
    SIMULATOR_HOST", "127.0.0.1"), 8181)
if sim.current_scene == "BorregasAve":
  sim.reset()
else:
  sim.load("BorregasAve")
spawns = sim.get_spawn()
ego_state = lgsvl.AgentState()
ego_state.transform = spawns[0]
ego = sim.add_agent("Lincoln2017MKZ (Apollo 5.0)",
    lgsvl.AgentType.EGO, ego_state)
npc_state = lgsvl.AgentState()
npc_state.transform.position = spawns[0].position +
    10 * forward
npc = sim.add_agent("Sedan", lgsvl.AgentType,
    npc_state)
waypoints = []
z_delta = 12
for i in range(20):
  speed = 24
  px = 0
  pz = (i + 1) * z_delta
  angle = spawns[0].rotation
  hit = sim.raycast(spawns[0].position + pz *
      forward, lgsvl.Vector(0, -1, 0), 1)
  wp = lgsvl.DriveWaypoint(hit.point, speed, angle,
      0)
  waypoints.append(wp)
npc.follow(waypoints)
sim.run()
```

Fig. 5 Sample Python file

sends sensor data to the self-driving software (i.e., Autoware) and receives control values to test the movement of the ego vehicle in the scenario.

The integration of the SVL simulator and Autoware is illustrated in **Fig. 4**. The SVL simulator (Fig. 4, right) shows the simulated environment and the ego vehicle with LiDAR. Sensor data from LiDAR is sent to Autoware to calculate the current position. RViz (Fig. 4, left) shows the output from Autoware, including the current position, upcoming the angular rate, and velocity calculated by the computing module.

2.4 Python APIs to control SVL simulator

The SVL simulator uses Python APIs to build a virtual simulation environment. For example, developers can manipulate the object placement and vehicle movements in a scenario, obtain sensor configuration and data, and control the weather and time state. These Python APIs can be divided into the following main types.

Simulator
The main object for connecting to the simulator and creating other objects.

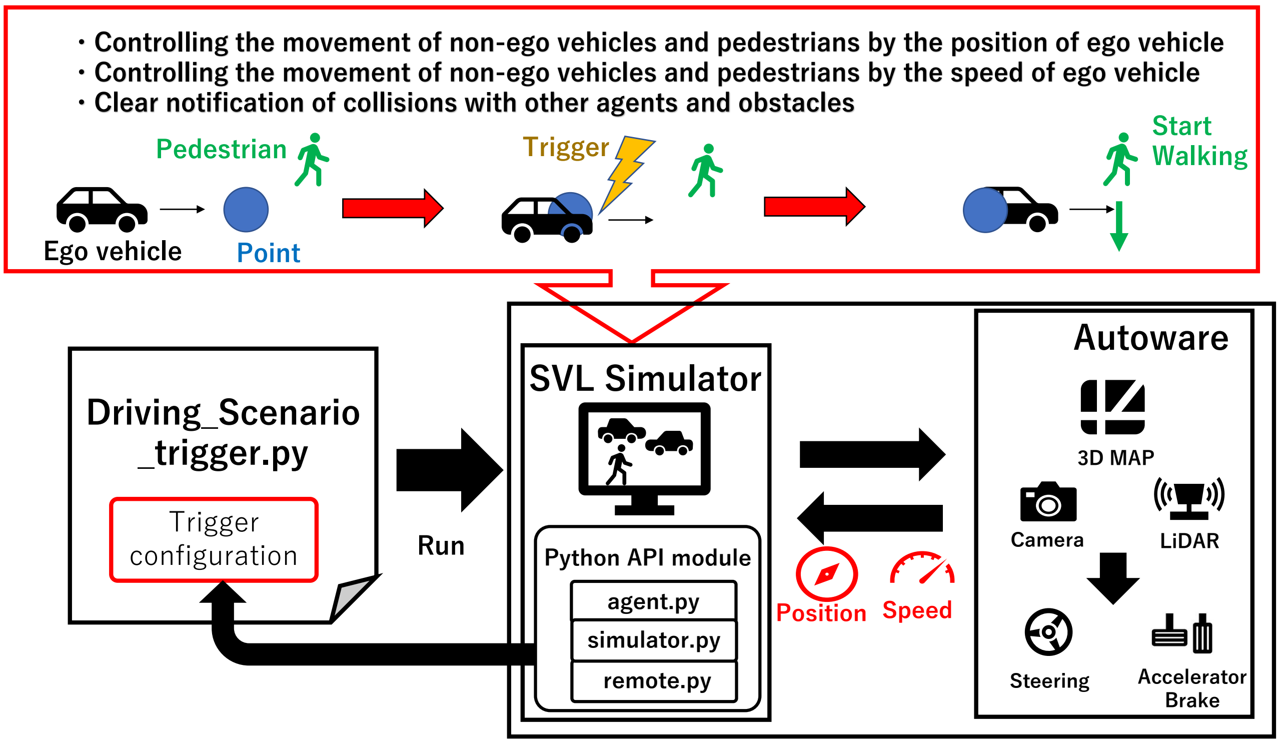
Agent
The superclass of an ego vehicle, a non-ego vehicle, and a pedestrian.

Ego vehicle
The ego vehicle with accurate physics simulation and sensors.

Non-ego vehicle
The non-ego vehicle with simplified physics (useful for creating many background vehicles.)

Pedestrian
A pedestrian walking on sidewalks.

The ego vehicle, non-ego vehicle, and pedestrian are subclasses of the agent, and these have common properties, e.g., con-



・Controlling the movement of non-ego vehicles and pedestrians by the position of ego vehicle
・Controlling the movement of non-ego vehicles and pedestrians by the speed of ego vehicle
・Clear notification of collisions with other agents and obstacles
Pedestrian
Ego vehicle
Point
Trigger
Start Walking
Driving_Scenario_trigger.py
Trigger configuration
Run
SVL Simulator
Python API module
agent.py
simulator.py
remote.py
Position
Speed
Autoware
3D MAP
Camera
LiDAR
Steering
Accelerator Brake

Fig. 6 The workflow of a trigger scenario linking Autoware and SVL simulator

version, position, and speed. One functionality of the ego vehicle is the *connect_bridge*, which is used to activate a bridge that establishes the communication between Autoware and the SVL simulator. Through this connection, the movement of the ego vehicle can be controlled by Autoware.

A sample description of a virtual environment using Python APIs is shown in **Fig. 5**. This code is output as a Python file and is reflected in the SVL simulator as the vehicle and pedestrian movements and the environment. Here, lines 1-6 describe the connection with the simulator, loading map data, and setting initial variable values. Lines 7-9 represent the generation of an ego vehicle, and lines 10-12 show the generation of a non-ego vehicle. The *npc* is a variable representing a non-ego vehicle. In this code, the non-ego vehicle with the same orientation as the ego vehicle is placed 10 m in front of the ego vehicle. Lines 13-22 set the waypoints on the path traveled by the non-ego vehicle. Then, *follow()* (Line 23) defines that the non-ego vehicle passes through a specific waypoint, and *run()* (Line 24) indicates simulator execution.

2.5 Testing tools

A previous study [7] proposed a framework to convert MATLAB files created in the Driving Scenario Designer (DSD) [8] to Python file format that can be run in SVL simulations. This existing framework allows developers to test scenarios created in MATLAB/Simulink on the SVL simulator. The DSD creates an original scenario file (*Driving Scenario.m*); then the original scenario is converted to a scenario file that can be run in the SVL simulator, and the scenario file can be verified by executing it in the SVL simulator. The conversion to scenario file flow includes a smoothing function for vehicle movements. The *clothoid_curve.py* file generates a smooth path along the waypoints in the *Driving Scenario.m* file that is created by the DSD tool.

The DSD tool is included in the Automated Driving Toolbox (ADT) within MATLAB/Simulink. MATLAB/Simulink software is commonly used in the automotive industry because it allows developers to analyze data, design algorithms, generate models, and supports model-based development. MATLAB/Simulink also supports ROS toolbox, a set of tools to communicate with ROS-based systems, e.g., Autoware. ADT provides tools to design, simulate, and test advanced driver assistance systems (ADAS) and self-driving systems. The DSD creates a virtual environment by placing the environmental elements (e.g., roads, vehicles, and pedestrians), and developers can set the parameters of these elements.

3. EVENT TRIGGER

This section presents the proposed framework as shown in Fig. 1 and **Fig. 6**. Firstly, this section describes the testing flow of the proposed framework. Secondly, this section explains the method used to realize the event-triggered functionality. Finally, this section describes the use case that can be achieved with the proposed framework.

3.1 Auto scenario converter of event-triggered functionality

The proposed tool to convert trigger scenarios is shown in Fig. 1. A trigger is a condition that a non-ego vehicle moves. The *scenario_converter_trigger.py* file can convert the trigger object and output a python scenario file that differs from the *sce-*

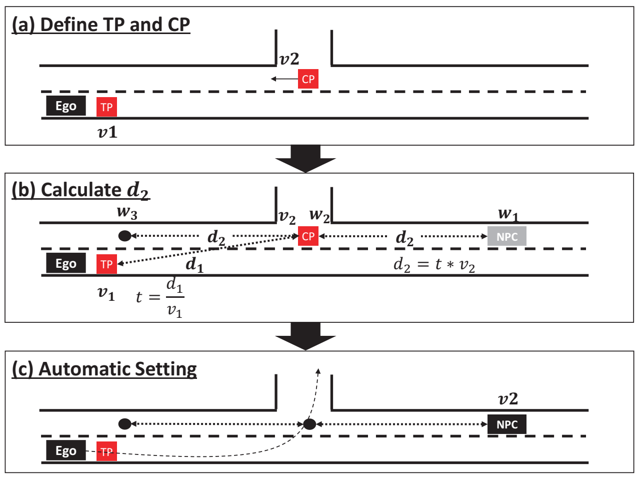


Fig. 7 Automatic settings

nario_converter.py output. The *scenario_converter_trigger.py* file has two different functionalities from *scenario_converter.py*.

The first functionality is a trigger position. Previously, developers must calculate the proper position of the non-ego vehicle when the trigger occurs, and transfer the non-ego vehicle; therefore, creating the trigger scenarios for many test cases manually is difficult for developers. In the proposed tool, the DSD can automatically set the trigger position in scenarios. A trigger object is added as a new object, and the proposed framework extracts the object's position as the trigger position.

The second functionality is automatic settings of a non-ego vehicle. The scenario test has an issue, which is necessary for developers to adjust the start position of a non-ego vehicle repeatedly. To reduce developer workload, the proposed framework determines a non-ego vehicle's position automatically. **Fig. 7** shows an example. TP is the trigger position to start a non-ego vehicle when the ego vehicle reaches the position. TP also has a speed parameter (v_1), i.e., the speed of the ego vehicle. CP is the crossover position of the ego vehicle's path and the non-ego vehicle's path. CP has the orientation and speed parameter (v_2) of the non-ego vehicle. The proposed framework calculates the distance d_1 between TP and CP, and then estimates the time t to move d_1 at v_1. To determine distance d_2, time t and v_2 (i.e., the non-ego vehicle's speed) are multiplied. The points that are d_2 distant from CP in a direction toward the opposite CP's orientation (w_1), CP's position (w_2), and d_2 distant from CP in a direction toward the CP's orientation (w_3) are set as the non-ego vehicle's waypoints. This allows developers to create the scenario as shown in Fig. 7 by only setting TP, CP, CP's orientation, v_1, and v_2. This reduces that the developers adjust the position of the non-ego vehicle repeatedly.

3.2 Driving scenario for event-triggered functionality

To realize the event-triggered functionality using the SVL simulator, linking Autoware and the SVL simulator is necessary. The workflow of a trigger scenario linking Autoware and SVL simulator is shown in Fig. 6. The trigger scenario file makes the SVL simulator receive the ego vehicle's position published by Autoware and determines whether the ego vehicle is at the trigger position. When Autoware operates the ego vehicle in the SVL

```
 1 sim = lgsvl.Simulator(LGSVL__SIMULATOR_HOST,
       LGSVL__SIMULATOR_PORT)
 2 ...
 3 def waypoint_reached(agent, index):
 4     ...
 5 def collision(agent1, agent2, contact):
 6     ...
 7 def trigger(agent):
 8     ...
 9 npc.on_waypoint_reached(waypoint_reached)
10 ego.on_collision(collision)
11 ego.on_trigger(trigger)
12 npc.follow(npc_waipoint)
13 sim.run()
```

Fig. 8 Trigger scenario file

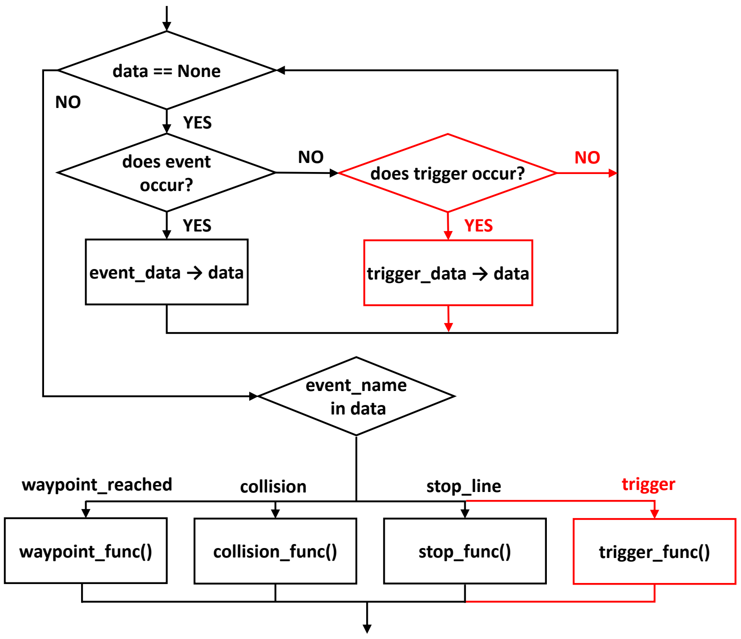


Fig. 9 Event processing flowchart for SVL simulator with the event trigger functionality

simulator, a time delay occurs from the time that the simulator launches until the time Autoware can control the ego vehicle. The non-ego vehicle starts running during this delay. The trigger scenario does not allow a non-ego vehicle to move before the ego vehicle reaches the trigger position; therefore, using the trigger scenario, the non-ego vehicle waits for the ego vehicle to reach the trigger position, and the test is performed as expected.

The event-triggered functionality is performed in a scenario file that can be validated in the SVL simulator. An example of a trigger scenario file as shown in **Fig. 8** is described as follows. This scenario file is a trigger scenario file that is written without the codes shown in Fig. 5. This trigger scenario file contains three callback functions, which are called when an event occurs. The first is the *waypoint_reached* function in Line 3, which is called when the non-ego vehicle reaches a waypoint. The second is the *collision* function in Line 5, which is called when the ego vehicle collides with another object. This function allows to help developers find out that the ego vehicle is in danger, e.g., an accident. As well as these two callback functions, the provided tool also can define a new callback function for when a trigger occurs. In this trigger scenario file, the *trigger* function corresponds to this

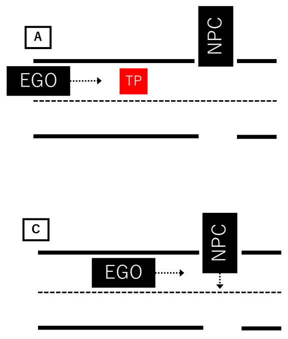


Fig. 10 Use case

callback function. The callback function can set the path of the non-ego vehicle after the trigger condition is satisfied. This allows developers to increase the scope of scenarios. To execute a callback function, a function that calls the callback function, such as the *on_waypoint_reached* function (Line 9) and the *on_collision* function (Line 10), is required. This function is defined in the Python APIs module file, such as *agent.py* and *simulator.py* in Fig. 1. The Python APIs module file contains the definitions, e.g., the Agent class, Pedestrian class and the functions that will help developers to execute the scenario file. The *on_waypoint_reached* function and *on_collision* function are already defined in the module file, and we defined the *on_trigger* function (Line 11) in the same way as them.

The flow after the *on_trigger* function has been called is shown in **Fig. 9**. The process written in red is the new process that we added. First of all, the process enters a branch to check whether *data* is stored or not. If the *data* does not exist, the process will continue to loop. This *data* is a structure in a JSON format, which contains the information, such as the name of the event and the agent. If the *data* is not stored, it enters a conditional branch to check whether the event has occurred or not. If an event (e.g., the collision the objects or reach the waypoint) has occurred, the transition is made in the "YES" direction, and the data of the event are assigned to *data*. If no event has occurred, it checks whether the trigger condition is satisfied or not. If the trigger condition is satisfied, the data of the trigger are assigned to *data*; otherwise, the loop is executed again. If the event occurs or the trigger condition is satisfied, the *data* is stored, and the process can exit the first conditional branch. After exiting the first conditional branch, the name of the event in the *data* is checked, and a different function is executed for each event. In the proposed tool, we add the event of the trigger to the event name.

To check if the trigger condition is satisfied, the system receives the position and speed information of the ego vehicle from Autoware. The *current_pose* topic, which is obtained using ROS communication, is used to obtain the position of the vehicle, and the *current_velocity* topic is used to obtain the velocity information.

3.3 Use case

This section explains the use case of the proposed framework. The proposed framework allows developers to create and conduct the flexible test scenarios to verify a self-driving system using the SVL simulator. In addition, when verifying dangerous scenarios, the proposed framework provides easy-to-understand feedback about whether an ego vehicle has collided with non-ego vehicles, pedestrians, or obstacles.

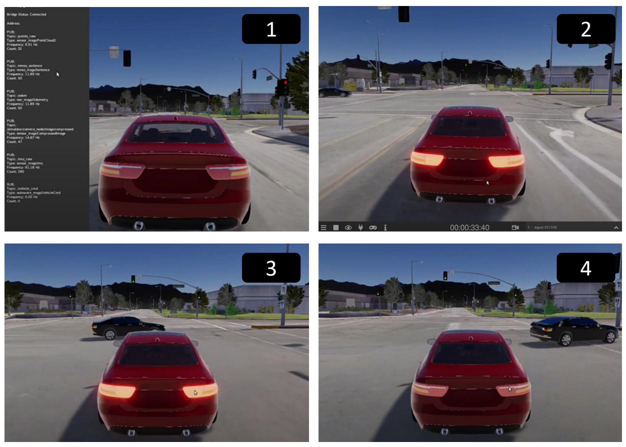


Fig. 11 Evaluation a trigger scenario in SVL simulator

Table 1 Software versions

Software	Version
ROS	Melodic
Ubuntu (OS)	18.04
Autoware	Autoware.AI 1.14.0
SVL simulator	2021.1

Using the proposed framework with the event-triggered functionality, a scenario such as shown in **Fig. 10**, can be verified easily using the SVL simulator. Fig. 10 shows a scenario in which a non-ego vehicle makes a sudden start, which is triggered by the ego vehicle reaching TP. We uploaded a demo movie using the proposed framework on this use case (`https://youtu.be/yCTYmegdYpc`). This scenario is designed with a single NPC to clarify the capabilities of the proposed framework, but event-trigger functionality is also available for scenarios with multiple NPCs.

If developers want to realize a scenario without the event-triggered functionality, the distance from the location of the non-ego vehicle to TP must be calculated when configuring the non-ego vehicle. In contrast, if using the event-triggered functionary, the distance from the location of the non-ego vehicle to TP is not needed because the event-triggered functionary is activated when TP is reached. The event-triggered functionality, therefore, realizes intuitive scenario creation and validation.

In addition to the conditions in the scenario shown in Fig. 10, the event-triggered functionality can be used to verify a scenario in which the non-ego vehicle makes a sudden start when it reaches TP exceeds a certain value. The event-triggered functionality allows developers to easily create and verify a wider range of the complex scenarios. In addition, by linking the simulator to self-driving systems, e.g., Autoware, and using collision feedback, developers can verify the safety of the self-driving system.

4. EVALUATION

This section examines how much the event-triggered functionality affects a simulation runtime. The environment versions used for the evaluation are summarized in **Table 1**. The evaluation of this paper used the scenario shown in Fig. 10, which is described in Section 3.3. This scenario includes a trigger event that could not be simulated in existing studies. In the SVL simulator, this scenario is executed as shown in **Fig. 11**. The first step is to connect the SVL Simulator to the Autoware bridge. The second step is to control the ego vehicle from Autoware, and the ego vehicle starts. Next, when the ego vehicle reaches the trigger position, the non-ego vehicle starts moving and passes in front of the ego vehicle. Finally, the ego vehicle moves forward.

For comparison with the proposed tool, this paper implemented the event trigger function using only the scenario file. In this implementation, the *trigger* function was placed in the *waypoint_reached* function in Line 3 of Fig. 8, and the waypoint was constantly updated to check if the trigger was satisfied. The *current_pose* topic and the *current_velocity* topic for obtaining the position and velocity from Autoware were also written in the scenario file.

The simulation runtime of the provided tools and when only the scenario file was changed is shown in **Fig. 12**. The simulation runtime of only the scenario file changed is 36 seconds, while the simulation runtime with the proposed tool is 17 seconds. The results show that the proposed tool is superior in terms of speed to the case using only modified scenario files. The reason for the delay in the case of changing only the scenario file is that the *waypoint_reached* function loops repeatedly until the trigger is satisfied. During the loop, objects such as non-ego vehicles are redrawn in the *waypoint_reached* function. This redrawing process is executed many times, which causes a large delay problem in the case of changing only the scenario file.

To solve this problem, this paper will use the *time.sleep* function within the *waypoint_reached* function. The *time.sleep* function can stop the process for a certain amount of time; thus, this function is expected to reduce the number of times the *waypoint_reached* function loops unnecessarily. The simulation runtime of the only scenario file using the *time.sleep* function is also shown in Fig. 12. Using the *time.sleep* function, the simulation runtime is shorter than without the *time.sleep* function; however, compared to the provided tools, it is more than 1.8 times slower.

In this paper, the event-triggered functionality is implemented using the callback function of the trigger to achieve both a small impact on simulation runtime and natural vehicle behavior. The proposed framework allows developers to verify previously unfeasible scenarios or create scenarios in a more intuitive manner at an increased time cost.

5. RELATED WORK

This section summarizes work related to simulators and test environments for self-driving systems. **Table 2** describes the comparison of the proposed framework and related work on simulator and test environments. The proposed framework supports multiple test environments including event-triggered functionality.

CARLA [5]: CARLA is an open-source simulator that supports the development, training, and validation of self-driving systems. CARLA provides open-source code and protocols, as well as digital assets, e.g., urban layouts, buildings, and vehicles. The simulation-executable agents in CARLA include Autoware agents and conditional imitation learning agents.

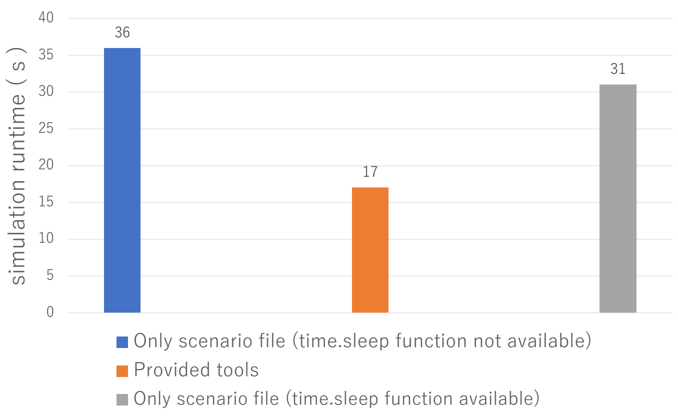


Fig. 12 Comparison of the simulation runtime

Table 2 Comparison of previous work on simulator and test environments

Test cases	CARLA [5]	3xD simulator [9]	AutonoVi-Sim [10]	AsFault [11]	MontiSim [12]	AD-EYE [13]	Force-based Concept [14]	Automatically generating test scenarios [15]	Self-driving Vehicle verification [16]	Scenario-Framework [7]	Proposed Framework
Weather	✓		✓			L		✓		✓	✓
Time	✓							✓		✓	✓
Machine learning			✓								
The behavior of other actors	L						✓			✓	✓
The study of driver's behavior		✓									
Automatically generating the road networks				✓				✓			
Detail and unit test					✓				✓	✓	✓
Event-triggered functionality											✓

∗"L" means "Limited."

Warwick 3xD [9]: The Warwick 3xD simulator tests real vehicles weighing up to 3,000 kg on a six-degree-of-freedom hexapod system. This simulator considers communication with both the front and rear of the vehicle, and the developers test co-adaptive cruise control systems. This simulator offers a wider range of testing options for autonomous vehicles compared to other physical testing methods.

AutonoVi-Sim [10]: AutonoVi-Sim is a new algorithm for autonomous vehicle navigation that satisfies traffic codes and norms. AutonoVi-Sim improves learning with its new algorithm, allowing it to simulate critical situations, e.g., jaywalking pedestrians or non-ego vehicles suddenly moving into the roadway.

AsFault [11]: AsFault automatically creates a virtual test environment for a lane-keeping system, which is a function of self-driving systems. It allows for more complex and challenging tests than randomly creating a road network. However, the only strength of the AsFault is the creation of the road network; thus, sensor information and surrounding buildings must be generated separately.

MontiSim [12]: MontiSim is a framework for testing the behavior of self-driving systems using a model environment that simplifies the real world. Map data are generated from OpenStreetMap, and road signs and traffic lights are generated on a map in an intersection-by-intersection manner. However, this simulator only supports either high-level simulations performed in large-scale environments, e.g., urban areas, or low-level simulations performed on individual components.

Force-based concept [14]: Force-based concept is a simulator that determines how an ego vehicle and other actors, e.g., pedestrians, and non-ego vehicles, move. It uses the force factor to represent the impact of the ego vehicle on other actors. Here, each parameter is represented by a force parameter to mimic a real-world scenario.

Architectural implications [17]: Architectural implications of autonomous driving is a study of self-driving tests. This study introduced the physical testing of self-driving platforms and summarized the results of real-world experiments on design constraints, e.g., performance, predictability, storage, temperature, and power, associated with self-driving systems. However, this study focused on hardware testing, which differs from the objective of the current study.

6. Conclusion

This paper has proposed a testing framework with an event-triggered functionality to verify test cases while controlling the movement of non-ego vehicles and pedestrians according to the position and speed information of an ego vehicle. High-safety standards have been required for the widespread use of autonomous vehicles, and accurate driving tests must be performed. Tests using conventional simulation techniques could not consider the flexible movements of non-ego vehicles and pedestrians; thus, determining whether the ego vehicle is in a dangerous state is difficult.

The proposed framework realized three primary contributions: the proposed framework increased the number of easily verifiable scenarios, showed collision feedback, and reduced the negative impact on the verification time by implementing the event-triggered functionality. The proposed framework allowed developers to verify previously unfeasible scenarios and create scenarios in a more intuitive manner at increased time cost. Specifically, developers could create scenarios intuitively, e.g., the scenario described in Section 3.3. In addition, the collision feedback made it easier to determine if the ego vehicle was in a dangerous state. The simulation time when using the proposed framework was verified in Section 4. The verification results showed that the proposed framework took at least 1.8 times faster than changing only the conventional scenario file. Note that the methodology of the proposed framework does not depend on ROS; therefore, the proposed framework can be applied to ROS 2 [18] such as Autoware.Auto [3], [19].

Acknowledgments This work was partially supported by JST PRESTO Grant Number JPMJPR1751.

References

[1] Open Robotics: Robot Operating System, `https://www.ros.org/`.
[2] Kato, S., Tokunaga, S., Maruyama, Y., Maeda, S., Hirabayashi, M., Kitsukawa, Y., Monrroy, A., Ando, T., Fujii, Y. and Azumi, T.: Autoware on Board: Enabling Autonomous Vehicles with Embedded Systems, *Proceeding of IEEE conference on International Conference on Cyber-Physical Systems (ICCPS)* (2018).
[3] Azumi, T., Maruyama, Y. and Kato, S.: ROS-lite: ROS Framework for NoC-Based Embedded Many-Core Platform, *Proceeding of International Conference on Intelligent Robots and Systems (IROS)* (2020).
[4] LG Electronics: SVL simulator, `https://www.svlsimulator.com/`.
[5] Alexey, D., German, R., Felipe, C., Antonio, L. and Vladlen, K.: CARLA: An Open Urban Driving Simulator, *Proceeding of Conference on Robot Learning (CoRL)* (2017).
[6] Bhadani, R. K., Sprinkle, J. and Bunting, M.: The CAT Vehicle Testbed: A Simulator with Hardware in the Loop for Autonomous Vehicle Applications, *Proceeding of International Workshop on Safe Control of Autonomous Vehicles (SCAV)* (2018).
[7] Miura, K. and Azumi, T.: Scenario-Framework: Converting Driving Scenario Framework for Testing Self-Driving Systems, *Proceedings of IEEE International Conference on Embedded and Ubiquitous Computing (EUC)* (2020).
[8] The MathWorks: Driving Scenario Designer, `https://jp.mathworks.com/help/driving/ref/drivingscenariodesigner-app.html`.
[9] Siddartha, K., Stewart, B., Gunwant, D. and Paul, J.: Identifying a Gap in Existing Validation Methodologies for Intelligent Automotive Systems Introducing the 3xD Simulator, *Proceeding of IEEE Intelligent Vehicles (IV)* (2015).
[10] Andrew, B., Sahil, N., Lucas, P., Daniel, B. and Dinesh, M.: AutonoVi-Sim: Autonomous Vehicle Simulation Platform with Weather, Sensing, and Traffic Control, *Proceeding of IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)* (2017).
[11] Alessio, G., Marc, M. and Gordon, F.: Automatically testing self-driving cars with search-based procedural content generation, *Proceeding of International Symposium on Software Testing and Analysis (ISSTA)*, pp. 318–328 (2019).
[12] Grazioli, F., Kusmenko, E., Roth, A., Rumpe, B. and Von, Wenckstern, M.: Simulation Framework for Executing Component and Connector Models of Self-Driving Vehicles, *Proceeding of International Conference on Model Driven Engineering Languages and Systems (MODELS)* (2017).
[13] Mohan, N. and Torngren, M.: AD-EYE: A Co-Simulation Platform for Early Verification of Functional Safety Concepts, *Proceeding of WCX SAE World Congress Experience (WCX)* (2019).
[14] Filippo, G., Evgeny, K., Alexander, R., Bernhard, R. and Michael, Von, W.: Force-based Heterogeneous Traffic Simulation for Autonomous Vehicle Testing, *Proceeding of IEEE conference on International Conference on Robotics and Automation (ICRA)* (2019).
[15] Feng, G., Jianli, D., Yingdong, H. and Zilong, W.: A Test Scenario Automatic Generation Strategy for Intelligent Driving Systems, *Mathematical Problems in Engineering (MPE)*, Vol. 2019, p. 10 (2019).
[16] Nima, R., Ramneet, K., James, W., Oleg, S. and Insup, L.: Self-Driving Vehicle Verification Towards a Benchmark, arXiv 1806.08810 (2018).
[17] Shih-Chieh, L., Yunqi, Z., Chang-Hong, H., Matt, S. M. E. H., Lingjia, T. and Jason, M.: The Architectural Implications of Autonomous Driving: Constraints and Acceleration, *Proceeding of ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 751–766 (2018).
[18] Maruyama, Y., Kato, S. and Azumi, T.: Exploring the performance of ROS2, *Proceeding of International Conference on Embedded Software (EMSOFT)* (2016).
[19] The Autoware Foundation: Autoware.Auto, `https://www.autoware.auto/`.