Restricted Natural Language and Model-based Adaptive Test Generation for Autonomous Driving

Yize Shi*, Chengjie Lu*, Man Zhang†, Huihui Zhang‡, Tao Yue* and Shaukat Ali§
*Nanjing University of Aeronautics and Astronautics
†Kristiania University College
‡Weifang University
§Simula Research Laboratory
Email: * {yizeshi, chengjielu, taoyue}@nuaa.edu.cn, †man.zhang@kristiania.no, ‡huihui@wfu.edu.cn, §shaukat@simula.no

Abstract—**With the ultimate goal of reducing car accidents, autonomous driving attracted a lot of attentions these years. However, recently reported crashes indicate that this goal is far from being achieved. Hence, cost-effective testing of autonomous driving systems (ADSs) has become a prominent research topic. The classical model-based testing (MBT), i.e., generating test cases from test models followed by executing the test cases, is ineffective for testing ADSs. This is mainly because ADSs are constantly exposed to ever-changing operating environments, and their uncertain internal behaviors due to the employed AI techniques. Thus, MBT must be adaptive to guide test case generation based on test execution results in a step-wise manner. To this end, we propose a natural language and model-based approach, named LiveTCM, to automatically execute and generate test case specifications (TCSs) by interacting with an ADS under test and its environment. LiveTCM is evaluated with an open-source ADS and two test generation strategies: Deep Q-Network (DQN)-based and Random. Results show that LiveTCM with DQN can generate TCSs with 56 steps on average in 60 seconds, leading to 6.4 test oracles violations and covering 14 APIs per TCS on average.**

Index Terms—**Natural Language and Model-based Testing, Adaptive Test Generation, Autonomous Driving.**

I. Introduction

Due to the application of artificial intelligence (AI) in autonomous driving systems (ADSs) together with these system's dynamic environments, some behaviors of such systems can only be discovered at run time. Therefore, a model-based testing (MBT) technique for such systems shall consider the dynamic evolution of such systems and their environments. Consequently, the test model of a system for generating test cases must be kept updated by constantly interacting with the system under test (SUT) during test execution.

To support automated testing of continuously evolving ADSs, we propose a novel methodology, named LiveTCM. LiveTCM is based on RTCM proposed in [4]. However, RTCM is not effective for testing ADSs since it statically generates executable test cases from test case specifications (TCSs), similar to use case specifications. Other related existing works such as the work by Wang et al. [3] also faces the same challenge as RTCM for testing ADSs.

An overview of LiveTCM is shown in Figure 1. It supports the full life cycle testing of evolving systems. LiveTCM is equipped with two types of TCS editors. One is for manually specifying TCSs (i.e., the *TCS Editor*) and the other is for dynamically completing TCSs (i.e., the *Executable TCS Editor*). Both editors implement the template of the Restricted Use Case Modeling (RUCM) approach proposed in [5]. The RUCM template has been applied to solve various software problems, e.g., requirements configuration for product lines [6], automated test case generation from use cases [3], and security and privacy requirements modeling [7]. *Executable TCS Editor* of LiveTCM has extra features such as highlighting the current being-executed step in a TCS and providing runtime execution information.

Test Generator of LiveTCM explores the behavior of the SUT with equipped test strategies, which adaptively determine which API to call, and then gets it executed via *Execution Engine* by interacting with the SUT and its environment at runtime. *TCS Generator* then transforms the executed API call into steps in the TCS. Moreover, LiveTCM is equipped with *TCS Verifier and Recommender* for automated verifying manually developed TCS steps and recommend them to testers when possible. LiveTCM also provides two key extension mechanisms: integrating different *API Specifications* to connect LiveTCM to various SUTs and their (simulated) environments, and adopting different *Environment Configuration Strategies* implementing with different techniques such as Deep Q-Networks (DQN) [33].

In general, the paper has the following contributions: 1) a web-based and executable TCS editor with the features of manually specifying TCSs, displaying generated TCSs and execution logs, and highlighting executing steps at runtime; 2) test and TCS generators, which together with an execution engine, supports five application contexts, such as step-wise or automatically executing already specified TCSs, and generating TCSs while execution, by interacting with the SUT and its operating environment via plugged-in REST APIs; and 3)an integrated and extensible framework for supporting automated testing of ADSs.

For evaluation, we choose open source Baidu Apollo [28] as the SUT, which supports ADSs development and testing. We set up a test bed with the Open Software platform of Apollo and LGSVL simulator [30], and used LiveTCM (integrate with two environment configuration strategies: a DQN model

Identify applicable funding agency here. If none, delete this.

and random) to generate and execute TCSs. Results showed that with the DQN model 400 TCSs were generated in 60 seconds, having 56 steps on average. We also observed 6.4 test oracle violations on average and covering 14 APIs per TCS on average.

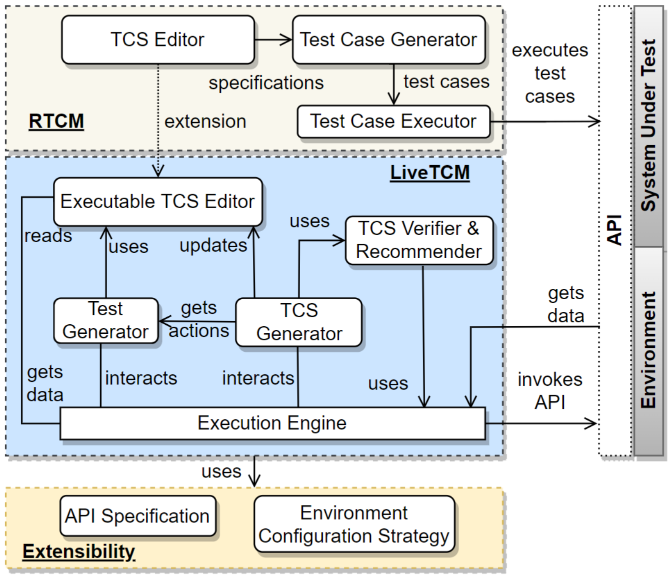


Fig. 1. Overview of LiveTCM

We organize the paper as: Section II introduces the background and running example, whereas the application contexts are presented in Section III. We introduce the LiveTCM methodology in Section IV. Section V presents the empirical evaluation and we report the experience and lessons learnt in Section VI. The related work is discussed in Section VII. Section VIII concludes the paper and discusses future work.

II. Background and Running Example

A. Static Test Case Generation with RTCM

To transit from textual TCSs to executable test cases, a natural language (NL)-based test case generation framework (named as RTCM) was proposed in our previous work [4]. RTCM defines an NL based template and keywords for the manual specification of TCSs, and supports the static and automated test generation from TCSs.

RTCM inherits from RUCM [5] and defines keywords such as VALIDATES THAT and INVOKES API. When specifying TCSs, these keywords are used with the RTCM template to structure textual TCSs and facilitate test generation. For example, keyword INVOKES API supports test generation. An API of the SUT specified with a sentence containing this keyword can be extracted and eventually executed on the SUT. VALIDATES THAT allows specifying a condition check performed by either the SUT or the test system, and the condition typically refers to accessible status of the SUT or the environment. The VALIDATES THAT sentence is handled as an assertion that retrieves such statuses and further assess if they satisfy the specified condition. In testing, a tester might also need to check some conditions manually, especially the test execution requires the SUT is operated together with unpredictable physical environment. Keyword VERIFIES THAT can then be applied for specifying such conditions with the subject of the sentence being a tester. RTCM also defines Oracle Verification Flows (OVFs) as specialized alternative flows, for providing a way to specify test oracles and connect them to specific steps such that one can easily understand what is expected after executing a TCS step.

RTCM's test generator takes TCSs as the input and statically generates executable test scripts with pre-defined coverage strategies such as *All Branch Coverage* and *All Condition Coverage*. This assumes that given TCSs, which are required to be manually developed, are correct and complete. Our understanding of the behavior of an ADS, however, gradually increases during the operation of the ADS. Hence, specifying TCSs and testing ADSs are naturally incremental. Hence, we, in this paper, propose the LiveTCM methodology (with tool support) to generate and execute TCSs by interacting with the SUT and its environment.

B. Running Example

We select a key ADS feature, *Maintain Lane* to illustrate LiveTCM. This feature should correctly handle the vehicle control input, react timely to avoid obstacles, and maintain driving in the lane, as described in *Brief Description* of the TCS (Figure 2). The LiveTCM interface (Figure 2) mainly contains two parts: a *Test Case Specification* pane and a *Console* view to visualize runtime test execution information, displayed on the left and right of Figure 2, respectively.

To specify a TCS, in *Test Data Specification*, a tester needs to ensure that the control, 3D geometrical transformations (i.e., transform), traffic light, perception, prediction, planning, localization, and routing of the ADS under test are all turned on, and the start and destination locations are defined. Configurations of these parameters are through the Configuration APIs such as *Configuration.Perception.Mode = On*. LiveTCM automatically checks whether the existence of these APIs and validate their values. Results of this verification step are provided in the console window (as illustrated in the console pane: *Configure Success!*, Figure 2).

A tester is also required to specify or select (if one already exists) a test setup, which itself is a flow of events including a list of steps. For instance, as shown in Figure 2, test setup *Load Scenario (San Francisco)* has five steps and one postcondition, which is for loading the SUT, simulator, etc. Whether a test setup is properly specified and consequently, and whether the SUT (e.g., *Apollo_V5*), the virtual vehicle (e.g., *lincoln_mkz*), and the operating environment (e.g., the LGSVL simulator) are properly configured can be verified via information provided in the *Console* view (Figure 2). Note that a TCS can plug in different test setups. In other words, pre-defined test setups can be reused across TCSs. Details are discussed in Section V.

In Figure 2, the *Maintain Lane* TCS has a basic flow and six alternative flows, i.e., a specific OVF (*O1*), a bounded alternative flow (*B1*), and four global OVFs (*G1-G4*). The basic flow specifies the main flow execution path of the TCS, which consists of a sequence of test actions (INVOKES API sentences) or assertions (VALIDATES THAT sentences). An alternative flow, e.g., *O1*, can branch out from a specific step of another flow of events. For instance, *O1* is a specific OVF for handling manual verification (i.e., its Step 1 containing keyword VERIFIES THAT) after Step 3 of the basic flow is executed. A bounded alternative flow (e.g., *B1*) branches out from more than one steps of another flow of events (i.e., Steps 2 and 5 of the basic flow). A global alternative flow branches out from any step of the basic flow as long as its condition is satisfied. For example, the condition of *G1*, *Status.Detection.Collision == True*, checks if the SUT has a collision with any obstacle in the environment. If yes, the steps in *G1* shows that the execution and therefore simulation are terminated, and the tester is notified. More details about Figure 2 are provided in Section IV.

III. Application Contexts

There are five application contexts of LiveTCM.

1) Manual Specification and Automated Execution (Manual-Spec-Auto-Exe): First, a tester manually specifies TCSs with the *TCS Editor* and executable test cases are statically generated by its *Test Case Generator*. Generated test cases can then be executed together with the SUT. This is a typical model-based test generation process, which requires that a test model, i.e., TCSs in this context, is available, and assumes that the test model is complete and correct in terms of describing the expected behavior of the SUT. Such an MBT approach assumes that tester fully know the expected behavior of the SUT and can specify them correctly with a proposed test modeling method, which is however very unlikely for a constantly-evolving systems. Correctness and completeness of such test models are often challenged when applied in practice [8].

2) Interactively Specification and Execution (Intera-Spec-Exe): To compare with *Manual-Spec-Auto-Exe*, the second application context doesn't require a complete and correct TCSs. Instead, one can interactively work with the *Executable TCS Editor*, which further interacts with *Test Generator*, *TCS Generator*, *Execution Engine*, and the SUT. This scenario can be initiated even when there is only one step (specified as an English sentence in our current implementation), and subsequent steps can be automatically generated based on a given strategy and the execution results of the previous step.

LiveTCM also supports the flexible generation of TCSs. A tester can import a TCS to *Executable TCS Editor* and select one of its steps to pause/resume the execution. The tester can also choose to change any already-specified TCS steps manually or automatically.

3) Automated Generation and Execution (Auto-Gen-Exe): LiveTCM can also automatically generate TCSs by interacting with *Test Generator*, *TCS Generator*, *Execution Engine* and the SUT. This is possible because LiveTCM can be connected to an environment configuration strategy, which automatically generates values to environment configuration parameters such as introducing pedestrians or NPC vehicles, changing the weather conditions. Such a strategy can be a trained a specialised algorithm, e.g., reinforcement learning model (e.g., [31]), or even a random strategy. Generated environment configurations are realized via REST APIs, which are plugged in LiveTCM and embedded as part of generated flow steps. Details are presented in Section IV-B.

4) Automated Verification and Recommendation (Auto-Veri-Recom): LiveTCM's *TCS Verifier* verifies TCSs via execution. A given TCS can be executed via *Execution Engine* by interacting with the SUT. During the execution, the tool can help testers to verify whether she/he specifies correct TCS steps and recommend generated solutions when needed. More details will be provided in Section IV-E.

5) Specification Replaying (Spec-Replaying): This context allows replaying TCSs step by step. A tester can ease and expedite diagnoses by selecting a specific system violation invoking step from *Executable TCS Editor* and inputting it into *Execution Engine* to replay. Moreover, the tester can classify system violations according to pre-defined criteria (e.g., collisions with obstacles) then replay specific categories of system violations.

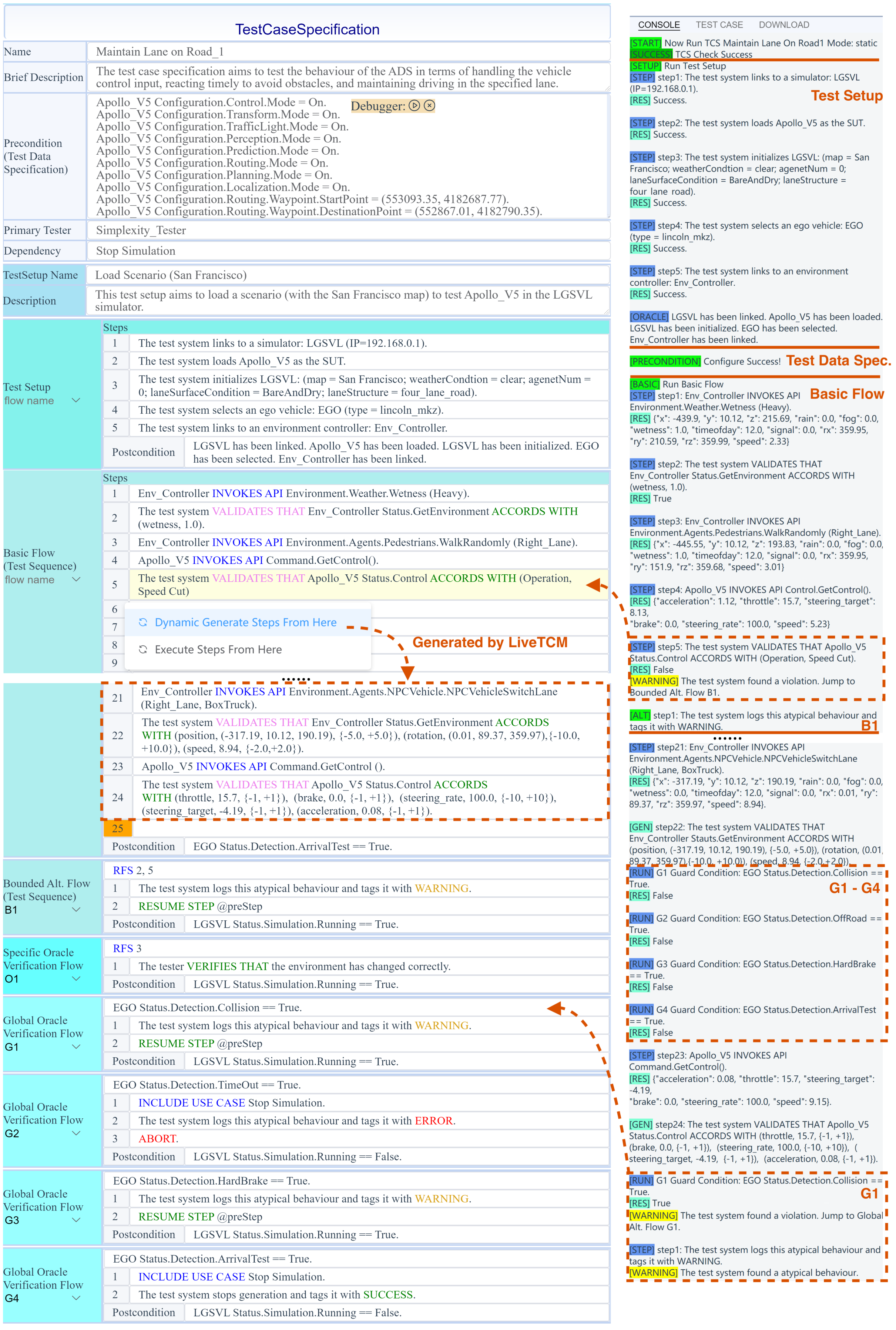
In summary, LiveTCM supports the five application contexts. In principle, a SUT can be a virtual ADS situated in a virtual operating environment such as Apollo_V5 deployed on the Lincoln MKZ model in the LGSVL simulator, or a physical vehicle. Both can be controlled with REST APIs invoked in TCSs via *Executable TCS Editor* of LiveTCM, with different sets of APIs equipped in *Execution Engine*.

IV. The LiveTCM Methodology

As shown in Figure 1, *TCS Editor*, *Test Case Generator* and its accompanying *Test Case Executor* of LiveTCM are already supported by RTCM (Section II). The other five components are newly introduced to support the dynamic execution and adaptive generation of TCSs. *Executable TCS Editor* is an extension to *TCS Editor* (Section IV-A). Section IV-B presents the automated and dynamic test execution. LiveTCM's adaptive test generation will be discussed in Section IV-C. With the support of the editor, test execution and generation, LiveTCM can automatically generate TCS steps/sentences, i.e., *TCS Generation* (IV-D). LiveTCM can also verify manually developed or modified TCSs, and provide recommendations when possible (Section IV-E).

A. Executable TCS Editor

Figure 2 shows the interface for testers interacting with LiveTCM. *Executable TCS Editor* implements the RTCM template (Section II) having fields to specify test data (as the *Precondition* of the TCS), test setup, and test steps (structured in flows of events). LiveTCM also introduces the following new keywords: ACCORDS WITH (used in a generated VALIDATES THAT sentence to check the status of the SUT or its environment, for handling verification with uncertainty measurements, e.g., Step 22, Figure 2), and SUCCESS/WARNING/ERROR (for the convenience of tagging



TestCaseSpecification

Name	Maintain Lane on Road_1
Brief Description	The test case specification aims to test the behaviour of the ADS in terms of handling the vehicle control input, reacting timely to avoid obstacles, and maintaining driving in the specified lane.
Precondition (Test Data Specification)	Apollo_V5 Configuration.Control.Mode = On. Apollo_V5 Configuration.Transform.Mode = On. Apollo_V5 Configuration.TrafficLight.Mode = On. Apollo_V5 Configuration.Perception.Mode = On. Apollo_V5 Configuration.Prediction.Mode = On. Apollo_V5 Configuration.Routing.Mode = On. Apollo_V5 Configuration.Planning.Mode = On. Apollo_V5 Configuration.Localization.Mode = On. Apollo_V5 Configuration.Routing.Waypoint.StartPoint = (553093.35, 4182687.77). Apollo_V5 Configuration.Routing.Waypoint.DestinationPoint = (552867.01, 4182790.35).
Primary Tester	Simplexity_Tester
Dependency	Stop Simulation
TestSetup Name	Load Scenario (San Francisco)
Description	This test setup aims to load a scenario (with the San Francisco map) to test Apollo_V5 in the LGSVL simulator.

Test Setup	Steps	
	1	The test system links to a simulator: LGSVL (IP=192.168.0.1).
	2	The test system loads Apollo_V5 as the SUT.
	3	The test system initializes LGSVL: (map = San Francisco; weatherCondtion = clear; agenetNum = 0; laneSurfaceCondition = BareAndDry; laneStructure = four_lane_road).
	4	The test system selects an ego vehicle: EGO (type = lincoln_mkz).
	5	The test system links to an environment controller: Env_Controller.
	Postcondition	LGSVL has been linked. Apollo_V5 has been loaded. LGSVL has been initialized. EGO has been selected. Env_Controller has been linked.
Basic Flow (Test Sequence)	Steps	
	1	Env_Controller INVOKES API Environment.Weather.Wetness (Heavy).
	2	The test system VALIDATES THAT Env_Controller Status.GetEnvironment ACCORDS WITH (wetness, 1.0).
	3	Env_Controller INVOKES API Environment.Agents.Pedestrians.WalkRandomly (Right_Lane).
	4	Apollo_V5 INVOKES API Command.GetControl().
	5	The test system VALIDATES THAT Apollo_V5 Status.Control ACCORDS WITH (Operation, Speed Cut)
	6	
	7	
	8	
	9	
	……	
	21	Env_Controller INVOKES API Environment.Agents.NPCVehicle.NPCVehicleSwitchLane (Right_Lane, BoxTruck).
	22	The test system VALIDATES THAT Env_Controller Status.GetEnvironment ACCORDS WITH (position, (-317.19, 10.12, 190.19), {-5.0, +5.0}), (rotation, (0.01, 89.37, 359.97),{-10.0, +10.0}), (speed, 8.94, {-2.0,+2.0}).
	23	Apollo_V5 INVOKES API Command.GetControl ().
	24	The test system VALIDATES THAT Apollo_V5 Status.Control ACCORDS WITH (throttle, 15.7, {-1, +1}), (brake, 0.0, {-1, +1}), (steering_rate, 100.0, {-10, +10}), (steering_target, -4.19, {-1, +1}), (acceleration, 0.08, {-1, +1}).
	25	
	Postcondition	EGO Status.Detection.ArrivalTest == True.
Bounded Alt. Flow (Test Sequence) B1	RFS 2, 5	
	1	The test system logs this atypical behaviour and tags it with WARNING.
	2	RESUME STEP @preStep
	Postcondition	LGSVL Status.Simulation.Running == True.
Specific Oracle Verification Flow O1	RFS 3	
	1	The tester VERIFIES THAT the environment has changed correctly.
	Postcondition	LGSVL Status.Simulation.Running == True.
Global Oracle Verification Flow G1	EGO Status.Detection.Collision == True.	
	1	The test system logs this atypical behaviour and tags it with WARNING.
	2	RESUME STEP @preStep
	Postcondition	LGSVL Status.Simulation.Running == True.
Global Oracle Verification Flow G2	EGO Status.Detection.TimeOut == True.	
	1	INCLUDE USE CASE Stop Simulation.
	2	The test system logs this atypical behaviour and tags it with ERROR.
	3	ABORT.
	Postcondition	LGSVL Status.Simulation.Running == False.
Global Oracle Verification Flow G3	EGO Status.Detection.HardBrake == True.	
	1	The test system logs this atypical behaviour and tags it with WARNING.
	2	RESUME STEP @preStep
	Postcondition	LGSVL Status.Simulation.Running == True.
Global Oracle Verification Flow G4	EGO Status.Detection.ArrivalTest == True.	
	1	INCLUDE USE CASE Stop Simulation.
	2	The test system stops generation and tags it with SUCCESS.
	Postcondition	LGSVL Status.Simulation.Running == False.

CONSOLE TEST CASE DOWNLOAD

```
[START] Now Run TCS Maintain Lane On Road1 Mode: static
[SUCCESS] TCS Check Success
[SETUP] Run Test Setup
[STEP] step1: The test system links to a simulator: LGSVL
(IP=192.168.0.1).
[RES] Success.

[STEP] step2: The test system loads Apollo_V5 as the SUT.
[RES] Success.

[STEP] step3: The test system initializes LGSVL: (map = San
Francisco; weatherCondtion = clear; agenetNum = 0;
laneSurfaceCondition = BareAndDry; laneStructure =
four lane road).
[RES] Success.

[STEP] step4: The test system selects an ego vehicle: EGO
(type = lincoln_mkz).
[RES] Success.

[STEP] step5: The test system links to an environment
controller: Env_Controller.
[RES] Success.

[ORACLE] LGSVL has been linked. Apollo_V5 has been loaded.
LGSVL has been initialized. EGO has been selected.
Env_Controller has been linked.

[PRECONDITION] Configure Success!

[BASIC] Run Basic Flow
[STEP] step1: Env_Controller INVOKES API
Environment.Weather.Wetness (Heavy).
[RES] {"x": -439.9, "y": 10.12, "z": 215.69, "rain": 0.0, "fog": 0.0,
"wetness": 1.0, "timeofday": 12.0, "signal": 0.0, "rx": 359.95,
"ry": 210.59, "rz": 359.99, "speed": 2.33}

[STEP] step2: The test system VALIDATES THAT
Env_Controller Status.GetEnvironment ACCORDS WITH
(wetness, 1.0).
[RES] True

[STEP] step3: Env_Controller INVOKES API
Environment.Agents.Pedestrians.WalkRandomly (Right_Lane).
[RES] {"x": -445.55, "y": 10.12, "z": 193.83, "rain": 0.0, "fog": 0.0,
"wetness": 1.0, "timeofday": 12.0, "signal": 0.0, "rx": 359.95,
"ry": 151.9, "rz": 359.68, "speed": 3.01}

[STEP] step4: Apollo_V5 INVOKES API Control.GetControl().
[RES] {"acceleration": 1.12, "throttle": 15.7, "steering_target":
8.13,
"brake": 0.0, "steering_rate": 100.0, "speed": 5.23}

[STEP] step5: The test system VALIDATES THAT Apollo_V5
Status.Control ACCORDS WITH (Operation, Speed Cut).
[RES] False
[WARNING] The test system found a violation. Jump to
Bounded Alt. Flow B1.

[ALT] step1: The test system logs this atypical behaviour and
tags it with WARNING.
                        ......
[STEP] step21: Env_Controller INVOKES API
Environment.Agents.NPCVehicle.NPCVehicleSwitchLane
(Right_Lane, BoxTruck).
[RES] {"x": -317.19, "y": 10.12, "z": 190.19, "rain": 0.0, "fog": 0.0,
"wetness": 0.0, "timeofday": 12.0, "signal": 0.0, "rx": 0.01, "ry":
89.37, "rz": 359.97, "speed": 8.94}.

[GEN] step22: The test system VALIDATES THAT
Env_Controller Stauts.GetEnvironment ACCORDS WITH
(position, (-317.19, 10.12, 190.19), {-5.0, +5.0}), (rotation, (0.01,
89.37, 359.97),{-10.0, +10.0}), (speed, 8.94, {-2.0,+2.0}).
[RUN] G1 Guard Condition: EGO Status.Detection.Collision ==
True.
[RES] False

[RUN] G2 Guard Condition: EGO Status.Detection.OffRoad ==
True.
[RES] False

[RUN] G3 Guard Condition: EGO Status.Detection.HardBrake
== True.
[RES] False

[RUN] G4 Guard Condition: EGO Status.Detection.ArrivalTest
== True.
[RES] False

[STEP] step23: Apollo_V5 INVOKES API
Command.GetControl().
[RES] {"acceleration": 0.08, "throttle": 15.7, "steering_target":
-4.19,
"brake": 0.0, "steering_rate": 100.0, "speed": 9.15}.

[GEN] step24: The test system VALIDATES THAT Apollo_V5
Status.Control ACCORDS WITH (throttle, 15.7, {-1, +1}),
(brake, 0.0, {-1, +1}),  (steering_rate, 100.0, {-10, +10}),  (
steering_target, -4.19,  {-1, +1}),  (acceleration, 0.08, {-1, +1}).

[RUN] G1 Guard Condition: EGO Status.Detection.Collision ==
True.
[RES] True
[WARNING] The test system found a violation. Jump to Global
Alt. Flow G1.

[STEP] step1: The test system logs this atypical behaviour and
tags it with WARNING.
[WARNING] The test system found a atypical behaviour.
```

Fig. 2. Running Example: *Maintain Lane*

test scenarios in execution results, e.g., Step 1 of *B1*, Figure 2). In addition, *@preStep* is introduced to combine with keyword RESUME STEP to indicate the previous step from where an alternative flow (e.g., *G3*) branches out from a reference flow of events (e.g., the basic flow). This keyword is needed as LiveTCM dynamically generates TCSs and the step to resume back is not static anymore.

Moreover, the editor serves as the window for displaying generated steps (e.g., Steps 21-24, Figure 2), the channel to initiate an execution process by right-clicking the TCS to be executed directly on the editor, and the window for monitoring the TCS execution progress, i.e., indicating the currently-executing sentence highlighted (e.g., Step 5, Figure 2).

LiveTCM also has a right-clicking menu for two execution modes: *Execute Steps From Here* and *Dynamic Generate Steps From Here* (Figure 2). The former requires that the tester manually specifies at least one step in the basic flow, while with the later LiveTCM can automatically generate steps for the current flow since its first step. Moreover, one can switch between these two modes.

During the execution, the debugger widget floats on the editor (top of Figure 2). The tester can click on the *Pause* or *Play* icons to pause or resume the execution. In addition, the *Close* icon can be used to terminate the execution. The console window records the output of every module (the right-hand side of Figure 2). The tester can monitor the execution process and download it as a log file after each execution.

B. Test Execution

One of the key contributions of LiveTCM is its automated and dynamic test execution, which is the foundation to support all the five application contexts (Section III).

1) APIs: To enable the dynamic test execution in LiveTCM, a list of APIs must be imported (Figure 1) and embedded as part of the *Executable TCS Editor*, *TCS Generator* and *Execution Engine* for manually specifying TCS steps and enabling automated test execution. There are four types of APIs defined : *Configuration*, *Status Checking*, *Command*, and *Environment*. The first three are from RTCM and reused in LiveTCM. The *Environment* API category is new in LiveTCM for manipulating the operating/testing environment of the SUT. In our running example, we have defined 52 REST APIs for supporting the testing of an ADS with the LGSVL simulator. For example, "Environment.NPCVehicle.SwitchLane" is an environment API for adding an NPC vehicle on one side of the ADS under test and switching to the lane that the ADS under test is following. Then we can use the *Status* APIs, such as "Status.GetEnvironment", to obtain the current environmental state and calculate the collision probability.

2) Process: From the methodology perspective, first, testers must use *Executable TCS Editor* to complete *Test Setup* when specifying a TCS for testing a specific feature/functionality of the ADS under test such as *Maintain Lane on Road_1* (Figure 2). It is required, as shown in Figure 2, to: 1) link the TCS to a specific simulator (e.g., LGSVL) via the IP address where the simulator is installed; 2) load the SUT; 3) initialize the simulator with a specific scene defining such as the weather conditions and map (e.g., *San Francisco*); 4) select a vehicle, which can be a virtual one operating in the simulator, or a physical vehicle; and 5) link to a environment configuration strategy, which is only needed in *Intera-Spec-Exe* and *Auto-Gen-Exe*. The TCS is ready to be generated and executed as long as all the conditions defined in the *Postcondition* field are satisfied. Moreover, a test setup can be reused across TCSs since *Executable TCS Editor* allows plugging in any existing test setup as long as it fits the needs of TCSs. Second, testers need to prepare test data in field *Test Data Specification*. For instance, as shown in Figure 2, the state of Apollo (the SUT) is initialized with statements such as *Configuration.Routing.Mode=On*.

While completing the test setup and test data specification, *Execution Engine* is triggered to automatically generate test actions containing Configuration APIs, which subsequently interact with the SUT and its operating environment to complete the actual configuration of the SUT and its operating environment, including the simulator (e.g., LGSVL), the SUT (e.g., Apollo_V5), the virtual vehicle (e.g., Lincoln_MKZ), and/or environment controller (e.g., a trained DNQ model)).

To execute an INVOKES API step in a TCS (e.g., Step 1 of the basic flow, Figure 2), LiveTCM translates the sentence of the step into a test action containing an API (e.g., translating *Environment.Weather.Wetness(Heavy)* into API "/LGSVL/Control/Weather/Wetness?wetness_level=Heavy", and then execute it (e.g., invoking the API to change the weather of the simulated operating environment (in LGSVL) of Apollo_V5). The execution results are then displayed in the console pane of the editor. For instance, as shown in the *Basic Flow* part of the right hand side of Figure 2, after executing Step 1 of the basic flow, the status of both the environment and the ADS under test is displayed with details, including the weather, location and speed of the vehicle.

C. Test Generation

Another key contribution of LiveTCM is its adaptive test generation supporting *Intera-Spec-Exe*, *Auto-Gen-Exe*, and *Auto-Veri-Recom* (Section III). Test generation generates test actions that are effective to push the SUT into unsafe situations such as collisions. Generated test actions can be executed via *Execution Engine* of LiveTCM (Section IV-B). The implementation of the test generation functionality is *TCS Generator* of LiveTCM (Figure 1).

Considering that ADSs are naturally exposed to complex and changing environment, in addition to their uncertainty of internal behaviors which employ AI techniques, the environment has a high dimensional space with a large number of unique configurations, efficiently generating befitting steps is challenging. Thus, we implemented and trained a DQN model for choosing an action based on the Q-network with the policy π. The input of DQN is an array of parameters for characterizing the environment and the ADS under test, such as parameters for its position, rotation and speed, parameters about the weather, the time of day and traffic light. The reward is the collision probability, i.e., if the distance between the ADS under test and other objects in the environment is near to the safe distance, the collision probability is closer to 1.0. The output of DQN is an action. Such an action is one of the *Environment* APIs for changing the environment to increase the collision probability. For instance, as shown in Figure 2, Steps 21 corresponds to the action automatically generated by the DQN model, which controls an NPC vehicle switching on the right lane of the road.

Note that we do not claim the DQN model as one of our contributions, because 1) its effectiveness has not been sufficiently assessed, doing which is a separate work, and 2) LiveTCM can be integrated with other strategies for automatically generating test actions, even including a simple random strategy. In our experiments, we include two test action strategies: one DQN model and one random strategy. Details are reported in Section V.

A test generation terminates when: 1) the ADS under test reaches the destination (defined in *Test Data Specification*, Figure 2), 2) the execution flow terminates with an ABORT sentence (e.g., Step 3 of *G2*), or 3) the ADS under test stays in one place for 12 seconds or more. Note that we do not stop a test execution, when a collision occurs, as the overall goal is to observe as many collision occurrences as possible.

D. TCS Generation

This section focuses on TCS artifacts generation, such as steps of flows of events. *Test Generator* (Section IV-C) gets the current status of the environment and ADS under test and inputs the state into the DQN model followed by recommending a test action, i.e., one Restful API to be executed by *Execution Engine* to configure/change the environment of the ADS under test. To obtain a realistic observation of the environment state change and the ADS, we let the ADS under test drive in the simulator for a few seconds[1], after one environment configuration API is invoked to change the environment. *TCS Generator* then translates the test action generated by *Test Generator* and its impact (i.e., status changes of the environment and ADS under test) into an order of steps.

1) Generating TCS steps: The first generated step is an *INVOKES API* sentence, based on a pre-defined template for mapping test actions to English sentences. Consequently, the resulting sentences are displayed properly in *Executable TCS Editor*. For instance, Step 21 (Figure 2) is a generated sentence based on the template of *Env_Controller INVOKES API X_API*, in which *Env_Controller* is the environment controller (i.e., essentially the DQN model for generating test actions) registered for testing the *Maintain Lane* TCS (see *Test Setup* in Figure 2); *INVOKES API* is a keyword defined in RTCM and inherited in LiveTCM to indicate that a sentence is about invoking an API; *X_API* corresponds to the test action generated by *Test Generator*.

Followed by an INVOKES API sentence, *TCS Generator* also generates a VALIDATES THAT sentence. Originally in RTCM, a VALIDATES THAT sentence is used to validate a condition, which is subsequently considered as a branching condition for statically generating test cases if there exists an alternative flow corresponding to the sentence. In LiveTCM, a generated VALIDATES THAT sentence displays values of a set of parameters relevant to the invoked API in the INVOKES API sentence. For instance, as shown in Figure 2, after executing Step 21, Step 22 is generated to check whether the current position, rotation and speed of the ADS under test are within ranges (which are predefined in the plugged-in API XML file). We only display information of these three parameters because, based on our experience and as also defined in the API schema (e.g., XML file), they are closely related to the API invoked in Step 21: *Environment.Agents.NPCVehicle.SwitchLane (Right_Lane, BoxTruck)*. The tester, however, can check the execution logs for the complete status information, which is also displayed in the console window of the editor. In the future, we plan to introduce an approach based on AI techniques to learn what parameters to display.

As discussed in Section IV-A, LiveTCM introduces a new keyword, named ACCORDS WITH, which should be used in an VALIDATES THAT sentence, to check the status of the SUT or environment. Following the ACCORDS WITH keyword, there can be one to more tuples. Each tuple (e.g., *(speed, 8.94, {-2.0, +2.0})* in Step 22) consists of three arguments, indicating the name of a parameter (e.g., *speed*), its value (e.g., *8.94*), and its tolerance including a lower limit and a upper limit (e.g., {*-2.0, +2.0*}). For categorical or Boolean parameters, there is no need to have tolerances. For instance, wetness is defined as a categorical parameter that can take three valid values: light, moderate and heavy. Note that such a tolerance, in the current implementation of LiveTCM, is pre-defined in the API XML file. However, we see opportunities to introduce, e.g., deep learning, to learn the tolerance along the time, especially considering that it might be unknown or only partially known during testing.

Furthermore, *TCS Generator* generate two steps (e.g., Steps 23 and 24) to obtain and display control decisions made by the ADS. For example, the ACCORDS WITH part of Step 24 displays the current control information of the ADS such as throttle being 15.7, no brake, steering_rate being 100.

2) Process: It is, first, important to define a clear scope (e.g., *Maintain Lane*) for a TCS. Then, in addition to define the test setup and test data specification, a tester is also required to manually specify OVFs, which can be either specific, bounded or global alternative flows, as defined in RTCM (Section II). We divide OVFs into two categories: OVFs specific to one or more steps in the TCS (e.g., *B1*, Figure 2), and OVFs specifying actions to take when a specified oracle is violated, which should be hold, otherwise, during any execution of the TCS. For instance, *G1* in Figure 2 can be defined in TCS for testing most of ADS features such as *Maintain Lane* and *Change Lane*. After executing each test action (i.e., an INVOKES API step), each of these global OVFs (e.g., G1) is invoked to check their conditions (e.g., *EGO*

[1]In our experiment, we used 4 seconds as with this configuration LiveTCM was able to produce good results.

Detection.Collision == True.). If any of their conditions is verified to be true, then the whole execution flow branches to the global OVF. For instance, if a *TimeOut* occurred, LGSVL is requested to stop the simulation (i.e., Step 1 of *G2*), then the execution flow is terminated (i.e., Step 3: ABORT).

E. TCS Verification and Recommendation

First, LiveTCM verifies all the test data configuration of a TCS (e.g., *Apollo_V5 Configuration.Control.Mode = On.* in Figure 2) to ensure that, before the execution, the SUT is configured properly. This is based on a given configuration file where valid options for Boolean and enumeration parameters, and valid ranges for integer and real parameters, are defined.

In *Manual-Spec-Auto-Exe* and *Intera-Spec-Exe* contexts, testers have the option of manually specifying steps in the flows of events of a TCS. A manually specified step can be verified (*Auto-Veri-Recom*) with *Executable TCS Editor* for syntactic checking, and *Execution Engine* for semantic checking. Syntactic checking verifies that RTCM restrictions (Section II), especially keywords, are correctly applied in TCS statements. For instance, misspelled keyword or missing parentheses when specifying a statement with an API invoked are considered syntax errors and can be detected by the editor. For VALIDATES THAT and INVOKES API sentences, after passing the syntactic checking, *Execution Engine* can be used to automatically verify their semantic correctness by executing the APIs together with the simulator and SUT.

In the *Auto-Veri-Recom* context, when the tester finishes writing an INVOKES API sentence, then starts writing a VALIDATES THAT sentence, LiveTCM can auto-fill the condition of the VALIDATES THAT sentence. For instance, as shown in Figure 2, the text after keyword VALIDATES THAT of Step 2 of the basic flow can be recommended by LiveTCM, as the status parameter that needs to be checked is *wetness*, which is the only relevant parameter to the *Environment.Weather.Wetness (Heavy)* API (embedded in Step 1 of the basic flow), according to the plugged-in API XML file. Moreover, the value of the wetness should be verified to be *1.0(Heavy)*.

If each step in the TCS is successfully executed, we consider that the whole TCS is verified. Automatically generated steps are verified when they are generated, since the generation is performed via execution. In case an exception occurs during an execution, testers are notified to manually determine whether the exception is caused because the TCS step being executed is incorrect or the SUT does have a bug.

All the execution results are recorded in LiveTCM. Also, each execution is labelled with information such as the number of associated faults. Each execution scenario (i.e., a sequence of steps in a TCS) is also labelled with information such as the total number of faults detected by executing the scenario, and the total number of executions of the scenario.

With an existing (specified/generated) TCS, LiveTCM supports to execute it from its beginning or any specified step, and stop the execution at any step (see *Debugger* toolkit in Figure 2). With the *Console* view, the verification and execution information are shown at runtime with labels, e.g., WARNING, SUCCESS. Note that, here, with the runtime execution and verification with the SUT, if a behavior of SUT is not as specified, LiveTCM allows a tester to check whether it is a bug in the SUT or a mistake in the specified TCS.

V. Evaluation

Section V-A presents our experimental design, and Section V-B presents results and discussion. Supplementary material is available online[2].

A. Experimental Design and Execution

1) Research Questions (RQs): Our RQs are:

RQ1: How effective is LiveTCM in terms of violating test oracles? We investigate whether LiveTCM generates effective TCSs to violate test oracles.

RQ2: How effective is LiveTCM in terms of covering environment configuration APIs? We investigate whether LiveTCM generates TCS steps covering diverse APIs.

RQ3: How efficient is LiveTCM in generating TCSs? We study LiveTCM's performance regarding time and steps required to violate test oracles.

2) Subject System, Experimental Settings and Procedure: Our subject system (i.e., the SUT) is the Apollo Open Platform 5.0 [28]. It has deep learning models for handling complex and uncertain scenarios. For automated driving, we integrated the LGSVL simulator 5.0 [29] in Apollo Open Platform. The simulator provides data from the simulated sensors such as camera, LiDAR, and GPS. On top of LGSVL, we implemented 52 REST APIs plugged in LiveTCM to facilitate testing. In this experiment, we defined our test setup with the default San Francisco map in LGSVL, which has traffic light intersections and multi-lane streets. The virtual vehicle is Lincoln2017MKZ. We implemented two test generation strategies: the DQN model (Section IV-C) and a random strategy that selects an API randomly each time.

We tested the maintain lane feature of Apollo_V5 on four roads, each of which has different test data specifications and consequently different TCSs. Figure 2 partially shows one of these four TCSs. Considering the inherent uncertainty of the SUT and randomness in the employed strategies, for each test setup/road, we repeated TCS generation 100 times. Consequently, with two different strategies, LiveTCM generated 800 TCSs (i.e., $4 \times 100 \times 2$).

3) Metrics: For RQ1, we define **effectiveness metric** $\#V_{TO}$, where $TO \in \{CL, TM, HB, ALL\}$. *CL*, *TM* and *HB* represent the *collision*, *not-arriving-destination and timeout* and *hard brake* types of test oracle violations, respectively; *ALL* combines all types of test oracle violations together. This classification is adapted from [34]. Given a TCS, $\#V_{TO}$ measures the number of violations of one type of TO.

To answer RQ2, we define **effectiveness metrics** below:

- $\#APIs$ *and* $\#U\_APIs$ measure the total number of APIs and unique APIs invoked in the generated steps of one TCS, respectively;

[2]https://github.com/simplexity-lab/LiveTCM

- $\#\overline{APIS}$ is the average of $\#APIs$ among different TCSs;
- $\#APIs(ST, RD)$, where $ST \in \{DQN, Random\}$ and $RD \in \{R1, R2, R3, R4\}$, indicating the numbers of APIs covered by TCSs generated by LiveTCM with each test generation strategy, and for each road, respectively;
- $\#CoU\_APIs$ is the unique API coverage: $\#U\_APIs$ divided by 52 (the total number of APIs in LiveTCM).

To answer RQ3, we define **efficiency measures** below:

- $\#Steps$: the total number of generated steps in one TCS;
- T_{avg}^{step} is the average time of generating a TCS step;
- OST is the overall time required to execute a TCS;
- $VDR_{time} = \frac{\#V_{ALL}}{OST}$ and $VDR_{step} = \frac{\#V_{ALL}}{\#Steps}$ are the violation detection rates based on time and steps.

B. Results and Analyses

1) RQ1: From Figure 3 (a), we can see that both environment configuration strategies (i.e., the DQN model and random) generated TCSs that led to three different types of test oracle violations, among which $\#V_{TM}$ is the type that both strategies achieved the least numbers of violations. Overall, the DQN model achieved a higher $\#V_{ALL}$ than the random strategy: the DQN model and random achieved 6.4 and 4.33 test oracle violations per TCS, respectively. This is expected since the DQN model is trained with the collision probability as reward function and tested to configure the driving environment of the ADS. Notice that overall violations achieved by the DQN model has a higher variance than results with random. Note that in this evaluation we only aim to demonstrate that LiveTCM can integrate different environment configuration strategies, but not assess/compare them. More optimized strategies can be applied in the future when needed.

2) RQ2: In terms of the average number of APIs covered in each TCS, LiveTCM with the DQN model (random), on average, covered 14.09 (13.68) APIs in each TCS (Figure 3 (b)). One can also observe that the unique number of APIs covered in each TCS, on average is 2.32 for LiveTCM with the DQN model and 11.78 for LiveTCM with random.

To further analyze the API coverage across the four roads, we present the total number of APIs covered by generated steps of TCSs for each road, in Table I. Recall from Section IV-B that we implemented 52 APIs and embedded them in LiveTCM. To better interpret the results, we classify these 52 APIs into five categories: 3 APIs for introducing pedestrians in the environment, 18 APIs about NPC switching lanes, 18 for NPC maintaining lanes, 10 for setting weather conditions, and 3 APIs for changing the time of day.

As shown in column $\#\overline{APIS}$ of the table, both strategies generated steps covering comparable average numbers of APIs in each TCS. LiveTCM with the DQN model achieved higher $\#\overline{APIS}$ than random for all the roads except for R1. In terms of the unique API coverage (i.e., $\#CoU\_APIs$), LiveTCM with random obviously generates steps with more diverse APIs than with the DQN model. This is reasonable as the DQN model has been trained to generate test actions that lead to test oracle violations; therefore from the results we can observe that introducing pedestrians to the environment is more effective in leading to test oracle violations.

From the table, we can also see that LiveTCM with the DQN model generates sentences invoking much higher numbers of pedestrian APIs than the other API categories for all the four roads except for R4. The plausible reason is that Apollo_V5 possibly has limitations on perceiving pedestrians when they stand still. We observed this by analysing LiveTCM execution logs, further checked it by repeating the experiment multiple times while listening to the channel of the Apollo's perception module. We also observed that the DQN model prefers to invoke NPC MaintainLane (*ML*) APIs than the other three API categories in R4 (with three traffic lights). This is because when the SUT approached an intersection, a number of NPC vehicles stopped at the traffic light and caused a traffic jam, and consequently increased the collision probability. This scenario eventually led to the DQN model to generate more MaintainLane APIs.

3) RQ3: Figure 3 (c) shows that the 400 TCSs (100 runs for each road) generated by LiveTCM with the DQN model have 56.37 steps, on average. Figure 3 (d) shows that it took LiveTCM around 58 seconds, on average, to generate a TCS, for both strategies. Consequently, it is easy to understand, as shown in Figure 3 (e), that the average time required to generate a step is around 1 second. These data are very comparable, across the two strategies, because the number of steps generated in a TCS is largely correlated to a selected road. In our experiment, the same four roads were used for both of the strategies.

Most of time spent by LiveTCM to generate steps is on simulation. Time spent on taking a test action produced by the environment configuration strategy and translating it into four steps of a TCS (Section IV-D) is negligible. In total, as shown in Figure 3 (e), generating each step, on average, is roughly 1 second. Therefore, we can conclude that LiveTCM itself is very efficient and a faster simulation can further improve the overall performance of the test generation.

From Figure 3 (f), one can observe that on average LiveTCM with the DQN (random) can lead to 0.11 (0.08) test oracle violations per time (VDR_{time}) and also per step (VDR_{step}). This again shows that LiveTCM is very efficient on generating TCSs leading to test oracle violations.

VI. Experience, Lessons Learnt, and Limitations

A. Applicability of LiveTCM

When using LiveTCM in the *Manual-Spec-Auto-Exe* context, it inherits the same benefits as RTCM, i.e., TCSs can be specified more precisely than free text. At the same time, TCSs are more understandable to practitioners as compared to formal specifications. The rest of the four application contexts (Section III) are designed to significantly improve the applicability of LiveTCM compared to RTCM for dealing with testing challenges imposed by ADSs. In particular, *Intera-Spec-Exe* and *Auto-Gen-Exe* help to interactively develop TCSs and automatically generate TCSs, respectively. This is important since it is practically very challenging to specify test

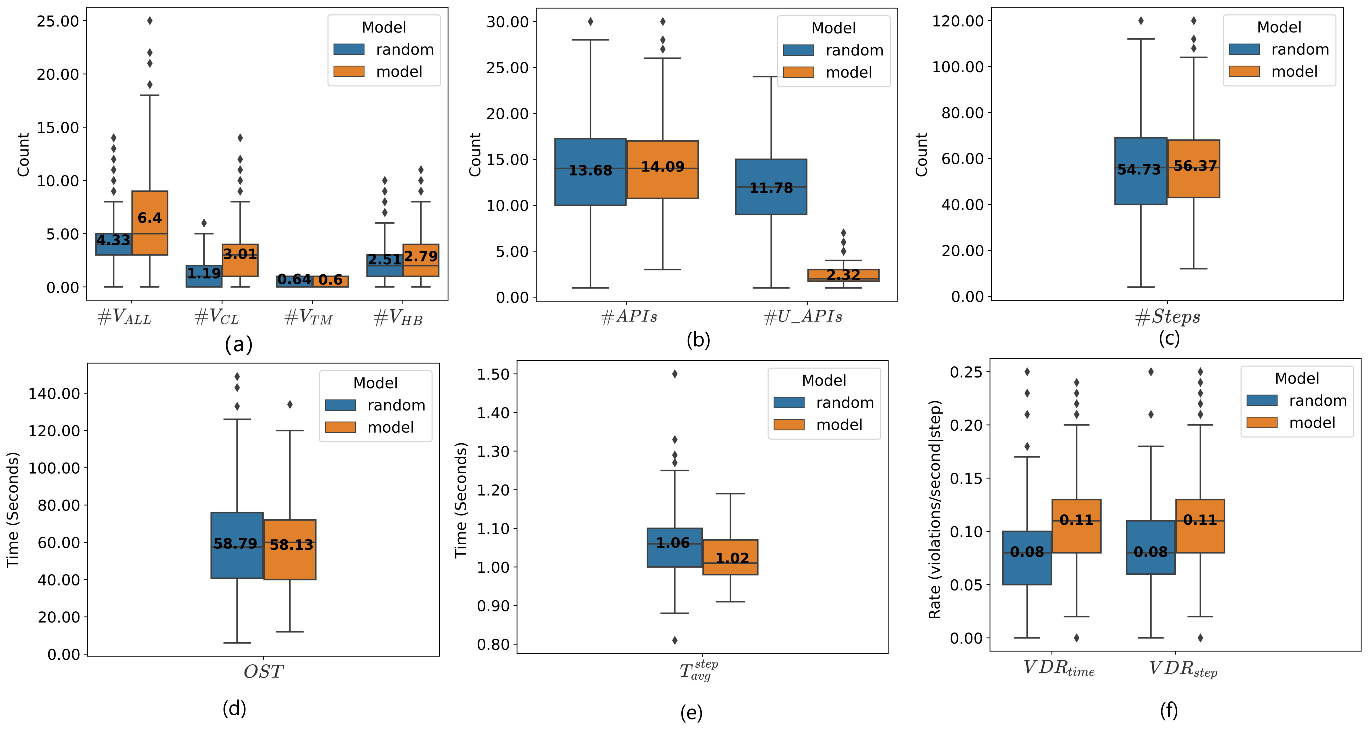


Fig. 3. Results in Box-plots for Effectiveness and Efficiency Measures

TABLE I
API COVERAGE OF GENERATED TCSS ACROSS VARIOUS ENVIRONMENT CONFIGURATIONS*

Road	**Strategy**	**Pedestrians (3)**		**SL (18)**		**ML (18)**		**Weather (10)**		**Time of Day (3)**		$\#\overline{APIS}$	**#CoU_APIs**
		Total	PA	Total	PA	Total	PA	Total	PA	Total	PA		
R1	Random	91	30.3	457	25.4	434	24.1	270	27.0	67	22.3	13.2	22%
	Model	907	302.3	27	1.5	37	2.1	25	2.5	7	2.3	10.0	4%
R2	Random	78	26.0	421	23.4	430	23.9	265	26.5	57	19.0	12.5	21%
	Model	1352	450.7	50	2.8	50	2.8	25	2.5	7	2.3	14.8	5%
R3	Random	91	30.3	519	28.8	524	29.1	266	26.6	86	28.7	14.9	24%
	Model	1499	499.7	55	3.1	66	3.7	30	3.0	11	3.7	16.6	5%
R4	Random	80	26.7	471	23.2	520	28.9	264	26.4	82	27.3	14.2	23%
	Model	10	3.3	60	3.3	1399	77.7	14	1.4	6	2.0	14.9	5%

* SL and ML denote NPC SwitchLane and MaintainLane; PA denotes the average number of APIs per each API type, e.g., averagely 30.3 APIs for each of the three Pedestrians API types for Random.

specifications of complex systems such as ADSs manually. Furthermore, with *Auto-Veri-Recom*, LiveTCM ensures that TCSs are syntactically and semantically correct; otherwise, one may think that there is a bug in the SUT, when it is not. Finally, *Spec-Replaying* helps a tester replay TCSs by the analysis purpose such as finding causes of a collision.

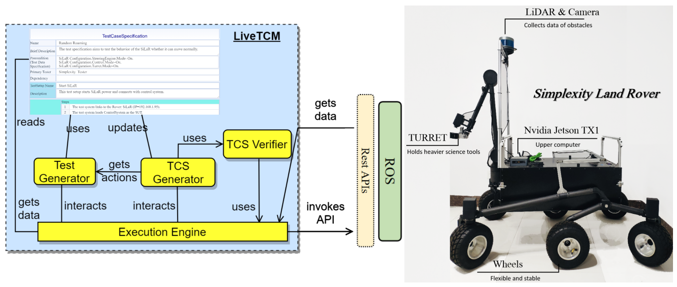
B. TCS Maintainability and Reusability

LiveTCM is designed explicitly with the maintainability and reusability of TCSs in mind. In terms of maintainability, once the SUT's implementation is changed, one can either rerun the generated TCSs, or let LiveTCM generate the TCSs automatically. In contrast, with RTCM, one has to manually change TCSs to reflect changes in TCSs, which is a time-consuming and error prone task. Regarding reusability, TCSs generated with LiveTCM can potentially be reused for the other versions of the SUT and, together with *Intera-Spec-Exe* could be tailored easily. Thus, LiveTCM improves over RTCM in terms of maintainability and reusability.

C. Extensibility of the LiveTCM Framework

LiveTCM is extensible from three perspectives: 1) plugging in different API Specifications, 2) adopting various environment configuration strategies; and 3) being applicable to different SUTs. In terms of API specifications, one can easily update existing APIs or upload new schemes that define accessible APIs with e.g., .XML files. Regarding environment configuration strategies, we demonstrated the integration of two different strategies, i.e., random and DQN-based. Other strategies can easily be integrated into LiveTCM.

To further investigate if LiveTCM is applicable to various SUTs, we connected LiveTCM with a physical land rover, named SiLaR (Figure 4), which uses open-source software, has sensors such as cameras and LiDAR. We implemented 10 APIs: 5 *Command* APIs for movement and 5 *Status* APIs for getting the status of SiLaR and validating test oracles. Note that for SiLaR, we only have *Command* and *Status* APIs, since we are not yet having the access to a fully equipped sandbox to operate it. Hence, we only conducted a simple experiment to assess the extensibility of LiveTCM. The experiment was repeated 10 times with a random strategy to invoke *Command* APIs and generate TCSs with LiveTCM. The stopping criteria were that SiLaR couldn't move or the number of steps exceeds 50. We observed that frequently sending commands from LiveTCM via the *Command* APIs may cause the memory leak of the upper computer and hence its reboot. This experiment demonstrates that it is feasible and relatively easy to use LiveTCM for testing other types of SUTs.



TestCaseSpecification

Name	Random Roaming
Brief Description	The test specification aims to test the behavior of the SiLaR whether it can move normally.
Precondition (Test Data Specification)	SiLaR Configuration.SteeringEngine.Mode=On. SiLaR Configuration.Control.Mode=On. SiLaR Configuration.Turret.Mode=On.
Primary Tester	Simplexity Tester
Dependency	
TestSetup Name	Start SiLaR
Description	This test setup starts SiLaR power and connects with control system.
Steps	1 The test system links to the Rover: SiLaR (IP=192.168.1.95).
	2 The test system loads ControlSystem as the SUT.

Fig. 4. SiLaR connected to LiveTCM

D. Abstraction of Generated TCSs

Currently, LiveTCM lacks a systematic way to abstract generated concrete test steps. Thus, many steps can be generated that affect the readability of TCSs. Our future work is to implement various TCS abstraction mechanisms. Abstracted TCSs will further enhance the reusability of TCSs since we can instantiate them for testing similar SUTs easily. Moreover, abstracted TCSs will provide a better understanding of automatically generated TCSs. This is important in our case, since we employ machine learning algorithms (e.g., reinforcement learning) to generate TCSs. Such algorithms are inherently uncertain, and it isn't easy to determine which decisions were made by them and why. With our abstracted TCSs, in the future, we would be able to understand decisions made by such algorithms and assess whether they are correct.

VII. Related Work

Dynamic test case generation approaches are related to our work. Such approaches guide test design with information obtained during test execution. Example works include the work by Miller and Spooner [13], a feedback-based framework integrating specification-based test case and dynamic inferences by Xie and Notkin [14], and a test case generation approach with dynamic specification mining by Dallmeier et al. [15]. LiveTCM shares the dynamic test case generation with these works, i.e., guiding test case generation based on information from test execution. However, the novelty of LiveTCM lies in automatically deriving TCSs based on interactions with SUT and its environment.

Executable Model-Based Testing (EMBT) is also related to LiveTCM., which focuses on executing test models in parallel to a SUT to guide test generation and execution. Ma et al. [20] proposed an EMBT approach for testing self-healing behaviors under uncertainty consisting of a modeling framework to create executable models and a test model executor for executing the models together with the SUT. Ma et al. further extended their work with a reinforcement learning-based approach to improve the effectiveness of their approach. In comparison, LiveTCM automatically generates executable test cases by reusing RTCM technique, in addition to generating TCSs using the feedback of previous test execution.

Keyword Driven Testing (KDT) is also related to LiveTCM, which is defined in the ISO/IEC/IEEE 29119-5 standard [22]. The KDT specifies test cases with predefined keywords associated with a set of actions in a test case. Comparing to TCSs described with free NL, KDT makes TCSs easy to understand, maintain and automate. LiveTCM is a type of KDT since LiveTCM partially follows the standard. Some related KDT works include the work by Rwemalika et al. [23] for test code evolution, the work by Tang et al. [24] to transform keyword-based test cases into different kinds of test scripts, and the work by Hametner et al. [25] to enable non-programmers in reading and writing test cases, and enable generation of executable test cases. As compared to these works, LiveTCM intelligently creates automated test steps meanwhile automatically generating executable test cases by following the KDT standard [22]. LiveTCM is a smart KDT approach in terms of capturing and using test execution feedback to generate TCSs.

VIII. Conclusion and Future Work

Testing Autonomous driving systems (ADSs) is challenging since they operate in highly dynamic environments, thereby facing unavoidable uncertainty in their operation. Thus, testing ADSs requires a dynamic and adaptive approach. To this end, we proposed LiveTCM, a restricted natural language, model-based adaptive testing approach. LiveTCM can dynamically generate test scenarios by dynamically interacting with an ADS and its environment to deal with dynamic and continuously changing ADS and its environment. LiveTCM was evaluated with an open source ADS in simulation to demonstrate its effectiveness in terms of violating test oracles and API coverage, whereas efficiency regarding time and steps needed to violate test oracles. Moreover, we demonstrate its extensibility by integrating LiveTCM with a physical rover.

Our future plans are to: 1) conduct a controlled experiment to compare LiveTCM with related approaches in terms of extensibility, and applicability; 2) introduce uncertainty concepts; and 3) introduce abstraction mechanisms to generate a high-level view of generated TCSs; and 4) provide a set of cost-effective adaptive test strategies.

REFERENCES

[1] Yuan, X., M. B. Cohen and A. M. Memon (2010). ”GUI interaction testing: Incorporating event context.” IEEE Transactions on Software Engineering 37(4): 559–574.

[2] Fourneret, E., F. Bouquet, F. Dadeau and S. Debricon (2011). ”Selective test generation method for evolving critical systems.” In 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, pp. 125–134.

[3] Wang, C., F. Pastore, A. Goknil and L. Briand (2020). ”Automatic generation of acceptance test cases from use case specifications: an nlp-based approach.” IEEE Transactions on Software Engineering. In press.

[4] Yue, T., S. Ali and M. Zhang (2015). ”RTCM: a natural language based, automated, and practical test case generation framework.” Proceedings of the 2015 international symposium on software testing and analysis, pp. 397–408.

[5] Yue, T., L. C. Briand and Y. Labiche (2013). ”Facilitating the transition from use case models to analysis models: Approach and experiments.” ACM Transactions on Software Engineering and Methodology (TOSEM) 22(1): 1–38.

[6] Hajri, I., A. Goknil, L. C. Briand and T. Stephany (2018). ”Configuring use case models in product families.” Software and Systems Modeling 17(3): 939–971.

[7] Mai, P. X., A. Goknil, L. K. Shar, F. Pastore, L. C. Briand and S. Shaame (2018). ”Modeling security and privacy requirements: a use case-driven approach.” Information and Software Technology (IST) 100: 165–182.

[8] Anand, S., E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. McMinn and A. Bertolino (2013). ”An orchestrated survey of methodologies for automated software test case generation.” Journal of Systems and Software 86(8): 1978–2001.

[9] NASA. 2020. Open-Source-Rover: A Build-It-Yourself, 6-Wheel Rover Based on the Rovers on Mars. Jet Propulsion Laboratory, NASA (2020). https://github.com/ nasa-jpl/open-source-rover.

[10] Microsoft. 2020. AirSim: a simulator for drones, cars and more. AirSim, Microsoft (2020). https://microsoft.github.io/AirSim/.

[11] Li, L., W.-L. Huang, Y. Liu, N.-N. Zheng and F.-Y. Wang (2016). ”Intelligence testing for autonomous vehicles: A new approach.” IEEE Transactions on Intelligent Vehicles 1(2): 158–166.

[12] Kalelkar, M., A. Kelkar and S. Pamarthi (2013). ”Autonomous Vehicle: Obstacle Detection and Decision-Based Navigation.” International Journal of Scientific and Research Publications 3(6): 1–2.

[13] Miller, W. and D. L. Spooner (1976). ”Automatic generation of floating-point test data.” IEEE Transactions on Software Engineering(3): 223–226.

[14] Xie, T. and D. Notkin (2003). ”Mutually enhancing test generation and specification inference.” International Workshop on Formal Approaches to Software Testing, pp. 60–69.

[15] Dallmeier, V., N. Knopp, C. Mallon, G. Fraser, S. Hack and A. Zeller (2011). ”Automatically generating test cases for specification mining.” IEEE Transactions on Software Engineering 38(2): 243–257.

[16] Kesserwan, N., R. Dssouli, J. Bentahar, B. Stepien and P. Labrèche (2019). ”From use case maps to executable test procedures: a scenario-based approach.” Software and Systems Modeling 18(2): 1543–1570.

[17] Buhr, R. J. A. (1998). ”Use case maps as architectural entities for complex systems.” IEEE Transactions on Software Engineering 24(12): 1131–1155.

[18] TDL. 2020. Test Description Language. (2020). $http://www.etsi.org/deliver/etsi_es/203100_203199/20311901/01.03.01_60/es_20311901v010301p.pdf$

[19] TTCN-3. 2020. Test Control Notation. (2020). http://www.ttcn-3.Org/index.php/downloads/standards

[20] Ma, T., S. Ali and T. Yue (2019). ”Modeling foundations for executable model-based testing of self-healing cyber-physical systems.” Software and Systems Modeling 18(5): 2843–2873.

[21] Ma, T., S. Ali, T. Yue and M. Elaasar (2019). ”Testing self-healing cyber-physical systems under uncertainty: a fragility-oriented approach.” Software Quality Journal 27(2): 615-649.

[22] ISO/IEC/IEEE. 2016. ISO/IEC/IEEE 29119-5: 2016 Software and systems engineering-Software testing-Part 5: Keyword Driven Testing.

[23] Rwemalika, R., M. Kintis, M. Papadakis, Y. Le Traon and P. Lorrach (2019). ”On the evolution of keyword-driven test suites.” IEEE Conference on Software Testing, Validation and Verification (ICST 2019), pp. 335–345.

[24] Tang, J., X. Cao and A. Ma (2008). ”Towards adaptive framework of keyword driven automation testing.” IEEE International Conference on Automation and Logistics, pp. 1631–1636.

[25] Hametner, R., D. Winkler and A. Zoitl (2012). ”Agile testing concepts based on keyword-driven testing for industrial automation systems.” IECON 2012-38th Annual Conference on IEEE Industrial Electronics Society, pp. 3727–3732.

[26] Thummalapenta, S., S. Sinha, N. Singhania and S. Chandra (2012). ”Automating test automation.” International Conference on Software Engineering (ICSE 2012), pp. 881-891.

[27] Little, G. and R. C. Miller (2006). ”Translating keyword commands into executable code.” Proceedings of the 19th annual ACM symposium on User interface software and technology, pp. 135–144.

[28] Baidu. ”Apollo data open platform.” Baidu, Beijing, China, 2020, http://data.apollo.auto.

[29] Rong, Guodong and Shin, Byung Hyun and Tabatabaee, Hadi and Lu, Qiang and Lemke, Steve and Možeiko, Mārtiņš and Boise, Eric and Uhm, Geehoon and Gerow, Mark and Mehta, Shalin and others. ”LGSVL Simulator: A High Fidelity Simulator for Autonomous Driving.” arXiv preprint arXiv:2005.03778, 2020.

[30] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. Carla: An open urban driving simulator, 2017.

[31] Margus Veanes, Pritam Roy, and Colin Campbell. Online testing with reinforcement learning. In Klaus Havelund, Manuel Núñez, Grigore Roşu, and Burkhart Wolff, editors, *Formal Approaches to Software Testing and Runtime Verification*, pages 240–253, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[32] Bugzilla: https://bugs.eclipse.org/bugs/ [Last Access-6/21/2015]

[33] Mnih, Volodymyr, et al. ”Human-level control through deep reinforcement learning.” nature 518.7540 (2015): 529-533.

[34] Ana Nora Evans, Mary Lou Soffa, and Sebastian Elbaum. A language for autonomous vehicles testing oracles, 2020.

[35] Vargha, A., Delaney, H.D. (2000). ”A critique and improvement of the cl common languageeffect size statistics of mcgraw and wong”, Journal of Educational and Behavioral Statistics, 25(2): 101–132.

[36] Kruskal, W.H., Wallis, W.A. (1952). ”Use of ranks in one-criterion variance analysis”, J.Amer. Stat. Assoc, 47(260): 583–621.