Science of Computer Programming 242 (2025) 103253

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

System-level simulation-based verification of Autonomous Driving Systems with the VIVAS framework and CARLA simulator ☆

Srajan Goyal [a,b,*], Alberto Griggio [a], Stefano Tonetta [a]

[a] Fondazione Bruno Kessler, via Sommarive 18, Trento, 38123, Italy
[b] University of Trento, via Sommarive 9, Trento, 38123, Italy

ARTICLE INFO

Keywords:
Formal verification
Autonomous Driving Systems
CARLA
Scenario generation

ABSTRACT

Ensuring the safety and reliability of increasingly complex Autonomous Driving Systems (ADS) poses significant challenges, particularly when these systems rely on AI components for perception and control. In the ESA-funded project VIVAS, we developed a comprehensive framework for system-level, simulation-based Verification and Validation (V&V) of autonomous systems. This framework integrates a simulation model of the system, an abstract model describing system behavior symbolically, and formal methods for scenario generation and verification of simulation executions. The automated scenario generation process is guided by diverse coverage criteria.
In this paper, we present the application of the VIVAS framework to ADS by integrating it with CARLA, a widely-used driving simulator, and its ScenarioRunner tool. This integration facilitates the creation of diverse and complex driving scenarios to validate different state-of-the-art AI-based ADS agents shared by the CARLA community through its Autonomous Driving Challenge. We detail the development of a symbolic ADS model and the formulation of a coverage criterion focused on the behaviors of vehicles surrounding the ADS. Using the VIVAS framework, we generate and execute various highway-driving scenarios, evaluating the capabilities of the AI components. The results demonstrate the effectiveness of VIVAS in automating scenario generation for different off-the-shelf AI solutions.

1. Introduction

The rapid evolution of Autonomous Driving Systems (ADS) poses significant challenges in ensuring their safety and reliability. The Verification and Validation (V&V) of these systems must address diverse, dynamic and complex real-world scenarios. Addressing these challenges has driven the integration of advanced verification and simulations methods and tools [1,2].

In the ESA-funded project VIVAS [3], a generic framework was developed for system-level simulation-based V&V of autonomous systems. This framework employs a combination of a simulation model of the system, an abstract symbolic model of system behavior, and formal methods to generate and verify simulation scenarios. It permits the specification of diverse coverage criteria, thereby directing the automated creation of scenarios, and formal properties to be verified on the simulation runs. The framework has been

☆ This work has been supported by: the "AI@TN" project funded by the Autonomous Province of Trento; the PNRR project FAIR - Future AI Research (PE00000013), under the NRRP MUR program funded by the NextGenerationEU; and the PNRR MUR project VITALITY (ECS00000041), Spoke 2 ASTRA - Advanced Space Technologies and Research Alliance.

* Corresponding author.
E-mail addresses: sgoyal@fbk.eu (S. Goyal), griggio@fbk.eu (A. Griggio), tonettas@fbk.eu (S. Tonetta).

https://doi.org/10.1016/j.scico.2024.103253
Received 15 June 2024; Received in revised form 7 November 2024; Accepted 26 November 2024

Available online 6 December 2024

created for space applications and applied to two use cases employing AI for resource prediction and opportunistic science. The system under test is based on the robotic digital twin developed in ROBDT [4].

This paper presents the application of the VIVAS framework to Autonomous Driving Systems (ADS), utilizing the CARLA simulator [5], a widely-used platform for simulating intricate driving scenarios in a controlled virtual environment. Among the various features provided by CARLA, its ScenarioRunner tool [6] enables the specification of diverse and complex scenarios based on the reuse of various predefined car behaviors. Moreover, various works proposed AI-based solutions (e.g., [7–9]) for the perception and control components of cars that are integrated with the CARLA simulator for their validation. The CARLA community also organized a competition to compare and rank such solutions [10]. These autonomous driving agents, grounded in AI methodologies, serve as crucial components in achieving the autonomy of cars. Their integration in CARLA allows evaluating the ADS behavior across different scenarios. However, such validation is so far based on a few manually crafted scenarios.

This paper describes the instantiation of the VIVAS framework for V&V of ADS, aimed at automatically generating and verifying scenarios simulated with CARLA. The VIVAS instantiation comprises: 1) the abstract ADS model specified in the extended SMV language of nuXmv [11], capturing essential aspects of its functionality and behavior in highway settings with varying traffic situations; 2) the formulation of a coverage criterion based on the abstract model, focusing on the interactions between ADS vehicle under verification (hereafter called *ego*) and surrounding vehicles (hereafter called *non-egos*); 3) the translation of the abstract traces generated from the abstract model to the ScenarioRunner specification and 4) a mapping of the simulation runs back to the abstract traces for runtime verification of formal properties. The experimental evaluation demonstrates how VIVAS is able to generate interesting scenarios effectively evaluating the behavior of the AI-based agents.

This work builds on the one presented at FMAS'23 [12], extending it by providing a detailed formal approach—particularly the abstract model—and expanding the experimental evaluation to include different AI agents and weather conditions.

The remainder of the paper is organized as follows: Section 2 summarizes related works and compares them with our approach. Section 3 provides a detailed description of the VIVAS framework and its components. Section 4 focuses on its instantiation for the ADS application. Section 5 presents the results, and Section 6 concludes the paper with key findings and directions for future work.

2. Related work

Over the last decade, we have witnessed significant efforts in the verification of AI-based autonomous systems using formal methods. Many works focus on formal verification of neural networks, for example encoding them into constraint solving (e.g., [13–15]) or using abstraction (e.g., [16,17]), just to name a few approaches. Our approach instead is rooted in the line of research (e.g., [18,19,1]) that tackles the verification at the system level using a simulator. This approach integrates AI components, potentially using machine learning (ML) models, for perception or control, in the context of a closed-loop cyber-physical system. As in VerifAI [1], the simulation traces are then formally analyzed with monitoring and runtime verification techniques.

Differently from the mentioned approaches, we exploit an abstract symbolic model to generate automatically the scenarios and define a coverage criterion for the generated test cases. While previous approaches focus on the automated synthesis of the simulation parameters for a specific scenario (e.g., different car movements to change lanes in front of the ego car), we concentrate on generating different functional scenarios (e.g., sequences of scenes with different change lanes of non-ego cars). Moreover, in this paper, we map such abstract symbolic scenarios to the scenario specification language of CARLA to verify ADS with different available AI solutions. So, the case study is based on available benchmarks for AI-based ADS taken as is.

There are in fact a variety of scenario specification languages that can be used in this context. VerifAI uses the Scenic language [20,21] to model the abstract feature space defining the scenarios, which can be instantiated to test cases. Scenic is a probabilistic programming language for scenario generation specifically designed to test the robustness of systems containing AI and ML components by allowing the generation of rare events. It allows the specification of spatial and temporal relationships between objects of a scenario as well as composing several scenarios into more complex ones. By the use of distributions for encoding interesting parameters, Scenic will perform automatic test case generation through the use of sampling.

Similarly, the Paracosm [2] framework is a programmatic interface that can be used to create various automotive driving simulation scenarios through the design of parameterized environments and test cases. The parameters control the environment in the scenario including the behavior of the actors and can include things such as pedestrians, lanes, and light conditions. Parameters are specified using either discrete or continuous domains and test cases are instantiated from the domain using random sampling and Halton sampling respectively. A coverage criterion is then defined over the coverage of the domains, where k-wise combinatorial coverage is used for the discrete domains and dispersion is used for continuous domains. Although Paracosm can provide output using the OpenDRIVE format, it is primarily coupled to be used with the Unity game engine, and as such scenarios are modeled using the C# programming language.

The Measurable Scenario Description Language (M-SDL) [22] is another scenario description language similar to Scenic. In M-SDL, one captures the behavior of identified actors in scenarios. M-SDL makes use of pre-defined basic building blocks such as actors (including the AV) along with some pre-defined behaviors, sets of possible routes, and environmental conditions. Libraries then use the basic building blocks to implement more complex behaviors such as cars overtaking, running red lights, driving on a highway, etc. Since M-SDL scenarios are abstract and parameterized, a single scenario can map onto many concrete ones through the use of sampling.

ScenarioRunner [6] is a module of CARLA that allows traffic scenario definition and execution for the CARLA simulator. The scenarios can be defined through a Python interface or using the OpenSCENARIO standard [23]. ScenarioRunner is used to validate

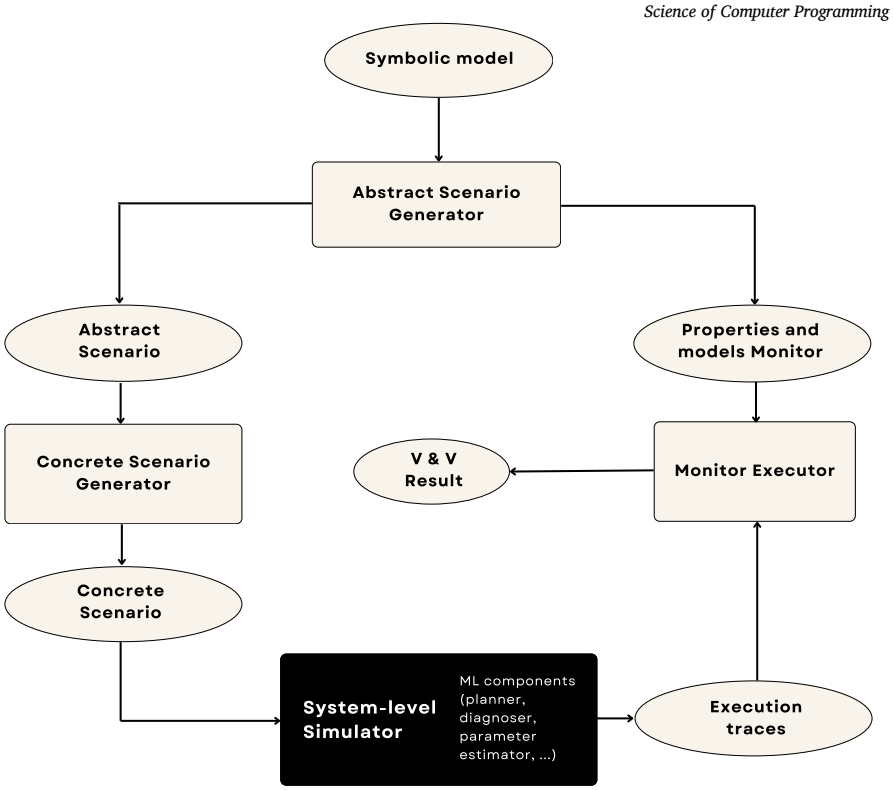


Fig. 1. Top-level architecture of VIVAS framework (rectangles represent the software components, ellipses represent the artifacts).

AI solutions for ADS. These results can be validated and shared in the CARLA Autonomous Driving Leaderboard [10], an open platform for the community to fairly compare their progress, evaluating agents in realistic traffic situations.

For all the above languages, the scenario must be specified manually, to then derive the test cases automatically. VIVAS instead provides a model-based approach to generate the scenarios automatically based on a coverage criterion that defines the interesting combinations of situations. In this paper, we focus on the integration with CARLA, because it allows the verification of the solutions shared by the CARLA community. However, the approach can also work with different specification languages, and we have, for example, a prototype integration with Scenic interfaced with CARLA. We have not presented the results of this integration in this paper since the ego model is based on Newtonian physics, with no AI models involved in the autonomous driving pipeline.

Although not specifically focused on AI-based systems, another very relevant work is described in [24], which proposes an optimization-based approach to synthesize ADS scenarios from formal specifications and a given map. Their formal specification of scenarios corresponds to our abstract scenario and is also synthesized from a symbolic model. However, test case generation does not follow any coverage criteria but enumerates specifications starting from an initial scene. In principle, our coverage-driven generation of scenarios can be combined with various techniques to concretize the scenario with different trajectories and sampling of the different environment parameters.

TAF [25] is another tool for automated test case generation of autonomous systems. Their abstract model is defined in an XML-based domain-specific language. It includes semantic constraints on the initial conditions of the environment and its agents (unlike the additional state transition systems in our work), which are solved using SMT solvers to generate abstract test cases. Random sampling is combined with these solvers to diversify the test cases, with an expert given coverage of data values. Their coverage criteria are based on covering parameter values to instantiate the scenarios. Although constraints on time can be expressed, more generic temporal specification on the sequences of actions and the related coverage criteria are not supported as in our approach. On the other side, our framework can be extended to constraints with quantifiers and complex data structure as in [26], which are currently not supported in VIVAS.

3. The VIVAS framework

VIVAS is a V&V framework that generates test cases for autonomous systems, including those with AI/ML components, by integrating system-level simulation with symbolic model checking. The framework employs formal, symbolic models of both the environment and system components to generate *abstract test scenarios* using model checking techniques. These abstract scenarios are then instantiated by concretizing the abstract parameters, resulting in concrete scenarios that are executed on a system-level simulator incorporating AI/ML models. The resulting execution traces are analyzed by an automatically generated monitor. VIVAS provides V&V results that include coverage statistics of the executed traces relative to the symbolic models, as well as detailed quantitative

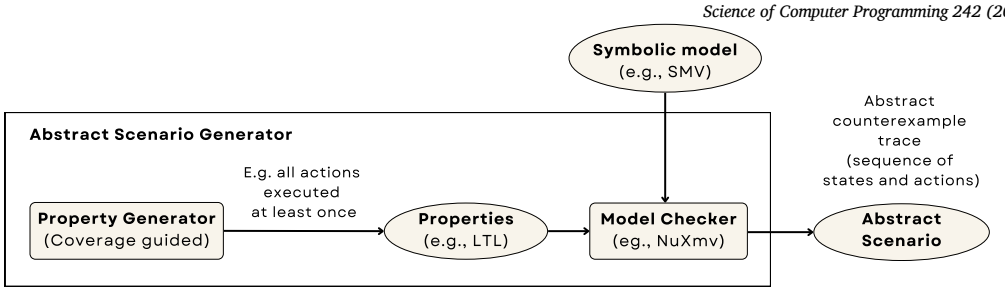


Fig. 2. Abstract scenario generation.

and qualitative information for each use case. The entire VIVAS workflow is fully automated. The domain-specific components are provided as inputs to the framework through a generic Python API, detailed in §4. An overview of the architecture is illustrated in Fig. 1, highlighting the key components of the VIVAS framework. These are the symbolic model, the abstract scenario generator, the concrete scenario generator, the simulator, and the executor monitor.

3.1. Symbolic model

The starting point of our methodology is a formal, symbolic model of the system, which provides an abstract view of the environment and the components under test (including AI/ML components). The abstraction is meant to provide an approximation of the behavior of the real system using a model that can be analyzed by means of symbolic model checking techniques. More specifically, we shall adopt infinite-state synchronous symbolic transition systems, possibly augmented with temporal aspects and continuous dynamics (as in timed and hybrid systems) as our formal modeling language, using first-order logic with background interpreted theories to describe the system's behavior. The symbolic model is then used to drive the generation of test cases for evaluating a set of formal properties of interest. The symbolic model will consist of the following three components:

1. Abstract model of environment It describes the behavior of the environment in which the autonomous system operates. Few examples from the automotive domain include: lanes and intersections on a highway, weather conditions, behavior of agents like lane changing and collision avoidance maneuvers, and other relevant aspects of the operating environment. We assume that the model of the environment provides an abstraction of the real environment, characterizing the assumptions we have on the environment's behavior. The quality of this environment model directly influences the validity of the system operating within it. The more precise the environment model is, the more effectively it defines the context in which the system operates, thereby enhancing the validation of the system under the assumptions we have proposed.

2. Autonomous system model This is an approximation of the behavior of the autonomous system under analysis (e.g., an autonomous vehicle driving on a highway). It is not crucial that the autonomous system model is a conservative abstraction of its real components, since, as explained below in §3.5, the correctness of the approach does not rely on this assumption. Detailed models can however provide further guidance for the generation of meaningful and/or interesting tests, thus allowing to improve the coverage by focusing on the more relevant simulation executions for stimulating interesting system behaviors.

3. Formal properties of interest These are specified in a language such as linear temporal logic (LTL), that capture the system-level requirements under test. An example from automotive domain would include, an AI-based autonomous vehicle under test never collides with any other obstacle in the environment.

3.2. Abstract scenario generation

Abstract test scenarios are generated from the formal system model using symbolic model checking techniques by the abstract scenario generator. Abstract scenarios are defined as combinations of values of predicates, which are inherently a collection of constraints describing interesting behaviors of the abstract system. Each constraint is defined by a domain (given as a list of predicates) and a type stating when the predicate must hold in the generated scenario (which can be initial, final, intermediate, or intermediate-ordered). From the technical point of view, as shown in Fig. 2, each abstract scenario is encoded as a formal property that is expected to be violated by the system (i.e. a property specifying that "the scenario cannot occur in the abstract system"). For each such property defined by the property generator, a model checker will be executed on the system model, with the goal of finding a counterexample to the property. By construction, each such counterexample corresponds to an execution trace witnessing the realization of the abstract scenario of interest. The enumeration of different scenarios is driven by a (input) coverage criterion, according to which the properties for the model checker are generated.

3.3. Concrete scenario generation

Each of the traces produced by the model checker is then refined into (a set of) concrete scenarios in order to produce the quantitative inputs required by the system-level (concrete) simulator. Due to uncertainties and abstractions in the abstract scenarios, a one-to-many mapping is defined where a single abstract scenario can be instantiated to many, possibly infinite, concrete scenarios. This is achieved by mapping abstract values to corresponding sets of concrete values based on simulation-specific inputs, followed by sampling from these sets. We employ sampling algorithms, e.g., Bayesian optimization, uniform random sampling, etc., to sample from input parameter probability distributions. These distributions are either defined by an expert or derived from historic simulator data or a real-time asset. For instance, the abstract model may discretize the time of day into broad categories like dusk, midnight, dawn, and midday. The corresponding sets of concrete time ranges could be [16:00, 22:00] hrs for dusk, [22:00, 04:00] hrs for midnight, [04:00, 10:00] hrs for dawn, and [10:00, 04:00] hrs for midday. The concretizer then samples an exact time from within these sets for simulation.

3.4. Simulation

The system-level simulator represents both the autonomous system with its control logic as well as the environment it is immersed in. The simulator represents the “ground truth” with respect to which the properties of the autonomous system are verified. The objective of the system level simulator is to run a simulation of the target asset under the requested conditions, by configuring the system, its environment, and its inputs as specified in the concrete scenario produced by VIVAS. Upon completion, the simulator provides the corresponding execution trace of the system, containing all the state changes and continuous variables to evaluate the properties of interest.

From the architectural point of view, the simulator is treated as a black box by the verification and validation methodology. The interaction with the rest of the VIVAS workflow is abstracted through a generic Python API for specifying simulation inputs (produced by the concrete scenario generator as mentioned above), executing the simulation, and producing a concrete execution trace to be analyzed by the runtime monitor.

3.5. Execution monitor

Each concrete scenario produced is executed by the simulator, which generates a corresponding concrete execution trace. This trace is then used to determine whether:

1. the concrete execution of the system satisfies the property of interest,
2. the concrete execution of the system complies with the input abstract scenario (which defines the situation of interest for the current test).

This is done by formally evaluating the trace with a runtime monitor that is automatically generated from the formal specification of the property and the abstract system model. The trace evaluation can have four possible outcomes:

1. The trace complies with the abstract scenario (defining the situation under test), and it also satisfies the property: the test execution is relevant and the test passes.
2. The trace complies with the abstract scenario, but it does not satisfy the property: this corresponds to a test failure on a relevant scenario, and it should be reported to the user.
3. The trace satisfies the property, but it does not comply with the abstract scenario: this corresponds to a (good) execution in an unexpected situation, in which some of the assumptions defining the scenario might be violated. This might be due to imprecisions/abstractions in the symbolic model and in the concretizer, which might prevent the realization of the abstract scenario under analysis. This situation might be reported to the user, as it might suggest that a revision/refinement of the symbolic model might be needed.
4. The trace violates the property and it does not comply with the abstract scenario: this corresponds to a test failure in an unexpected situation. Similarly to the above, it might be a warning that the symbolic model of the system is not precise enough to capture the situations of interest defined by the abstract scenario.

Importantly, since the monitor is generated only from the formal properties and the environment model, its correctness does not depend on the additional models for the system components, which therefore need not be conservative abstractions of the corresponding real components. This is very important for the practical feasibility of the approach: ensuring that the system models are correct over-approximations is in general extremely challenging in practice, particularly for components making heavy use of advanced machine learning techniques; therefore, not imposing such requirement is one of the key aspects of VIVAS.

From the implementation point of view, the simulation execution traces are adapted to a format specific to the monitor. A use case-specific mapping is provided to internally translate and manipulate these execution traces, ensuring compatibility with the execution monitor. This mapping varies among different use cases, depending on the trace formats directly available from the autonomous system under analyzes. To ensure efficient monitoring, only the necessary trace information is output directly in a target format compatible with the execution monitor, emphasizing the usefulness of the monitoring properties. This translation mechanism can be

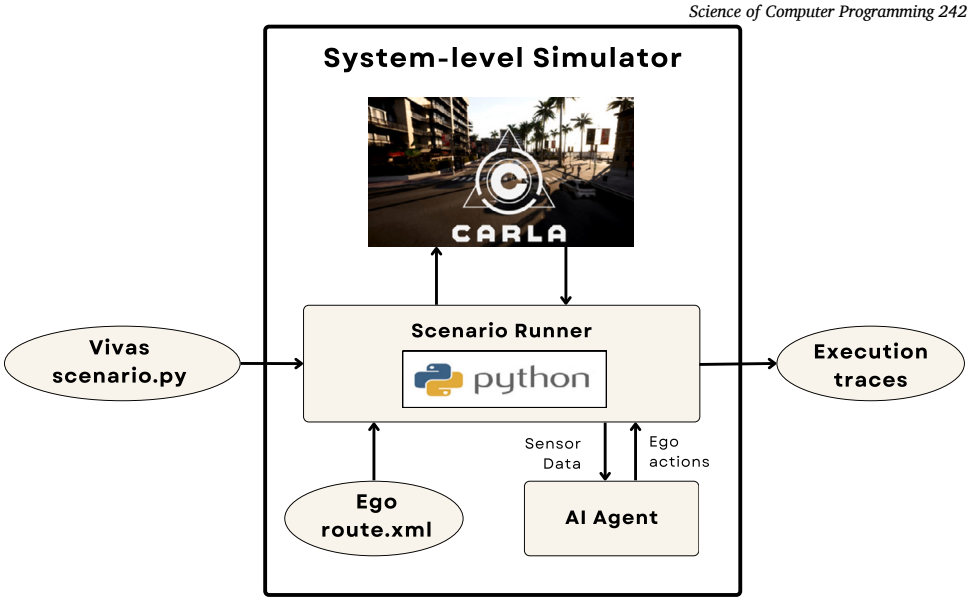


Fig. 3. VIVAS interface with CARLA simulator and AI based agents.

seamlessly integrated into the simulator itself (e.g., as an output filter) that pre-processes the data for analysis. The Monitor Executor then serves as a simple wrapper that invokes the runtime monitor to perform the necessary monitoring tasks.

3.5.1. Abstract and concrete coverage

Ensuring an adequate level of coverage is one of the primary goals of a good set of tests. In VIVAS, coverage is defined with respect to a domain-specific notion of "interesting situations", which are those that are (implicitly) defined by the possible combinations of values of predicates that are used by the abstract scenario generator to produce abstract traces. By construction, therefore, VIVAS tries to enumerate abstract scenarios that ensure a 100% degree of coverage of the *abstract* situations of interest.[1] Each abstract scenario is then refined into one or more concrete simulation inputs, leading to corresponding concrete simulation traces. In order to determine the *concrete* coverage (i.e., the degree of coverage of interesting situations at the concrete level), the VIVAS monitor analyzes the execution traces. It checks for compliance with the property of interest and the corresponding abstract scenario's specification (i.e., the "interesting situation") from which the concrete executions originate. Only executions that satisfy the abstract scenario specification contribute to the coverage at the concrete level: if an execution does not comply with its abstract specification, it represents an unexpected situation from which no coverage information can be drawn.[2]

4. Autonomous driving application

In this section, we instantiate the VIVAS framework on an ADS application. Specifically, we define an abstract model that focuses on highway scenarios where the ego is surrounded by other vehicles in various dynamic situations, guided by a coverage criterion we define explicitly. We choose CARLA as the system-level simulator as it is widely used in the automotive domain and supported by an active community offering various AI-based solutions for perception and control. We integrate the VIVAS framework with CARLA through automated processes within the "Concrete Scenario Generator" and "Execution Monitor" components. In the concrete scenario generator, abstract scenarios generated by the model checker are first parsed to extract relevant information, which is then automatically translated into CARLA-specific input files, resulting in concrete scenarios for simulation. In the execution monitor, we implement mechanisms to map CARLA simulation runs back to abstract traces for coverage analysis, as well as to perform runtime verification of formal properties. In the following, we provide detailed information on these domain-specific aspects of the VIVAS framework.

4.1. CARLA simulator

CARLA [5] is a high-fidelity, open-source simulator designed for the development, testing, and validation of autonomous driving systems. Developed in C++ as a plug-in for Unreal Engine, it offers a standalone package with pre-defined maps featuring 3D meshes that replicate various real-world environments, including city roads with intersections and highways. CARLA includes multiple sensor

[1] Note that a 100% degree of coverage might not be reached, either because some situations are not feasible already at the abstract level, or because the model checker cannot find a witness trace for the scenario specification within the given resource budget (time and/or memory).

[2] Note that in principle such a situation might still provide *some* information (e.g. it might still cover a different but still interesting situation); therefore, the test result is still reported to the user. However, determining this might not be obvious in general, and therefore we opted for the conservative choice of excluding the test from the computation of the degree of coverage in such cases.

Table 1
Details of AI agents used in the benchmarking.

AI agent	Carla Leaderboard Rank	Sensors	Perception and Planning	Low-level Control
Interfuser	2	1 Lidar, 3 RGB Cameras	Machine Learning based	PID
TCP	3	1 RGB Camera	Machine Learning based	PID
TF++ WP Ensemble	4	1 Lidar, 1 RGB Camera	Machine Learning based	MPC
LAV	6	1 Lidar, 3 RGB Cameras	Machine Learning based	PID

models, such as cameras, Lidar, radar, GPS, and IMU, to collect environmental data. The simulator supports a wide range of vehicle models, from small cars to large trucks, each with unique properties like mass, dynamics, and controls. Fig. 3 shows the system-level simulator with its interface with VIVAS framework. A simulation in CARLA comprises (i) the CARLA Simulator, which handles physics computations, scene rendering, and actor properties, and (ii) client scripts using a Python API to control actors, sensors, and environmental conditions. Additionally, ScenarioRunner, a CARLA module, provides a Python interface for specifying ego routes and complex traffic scenarios by defining non-ego agent behavior. This module allows users to run CARLA on specified maps and implement their own AI-based ego agents to process sensor data and determine the ego's actions.

4.1.1. AI-based components

The CARLA community through its leaderboard competition [10] provides various state-of-the-art AI solutions for end-to-end autonomous driving. However, only a few of them provide the necessary code and well-trained models for their methodologies to be evaluated and built upon. We have specifically benchmarked Interfuser [7], TCP [8], TF++ WP Ensemble [27], and LAV [9], all four currently in the top 6 of the leaderboard 1.0 (sensors track) at the time of writing (top 4 in terms of the solutions with open-source code and well-trained models). All four AI agents that we benchmark share the following main characteristics:

1. The maximum driving speed is restricted to 5 m/s, which is quite conservative for highway driving;
2. The ego vehicle always remains in its designated lane and requires an external route to follow. It may only change lanes according to the waypoints provided by this route on the map, meaning it does not overtake slower cars in the same lane. Instead, the ego vehicle follows them while maintaining a safe distance or coming to a complete stop.
3. Standard overtaking rules, such as overtaking only on the left (or right), are not enforced.

Note that our V&V methodology is agnostic to the AI solution chosen for the simulator. Since the abstract test scenarios are generated from the symbolic model of the system, abstract coverage would be the same for different AI solutions, although the concrete coverage may vary.

Here is a brief summary of the AI solutions we have benchmarked with our methodology. In Table 1, we provide some further details on the type of sensors, planner and low-level control algorithms utilized by these AI solutions.

Interfuser This solution primarily focuses on the safety of AD systems by generating interpretable semantic features of the environment through multi-model sensor fusion, for constraining the agent's low-level control actions in real-time within safe sets. The perception system processes the data gathered by 3 RGB cameras and one Lidar sensor.

TCP It stands for Trajectory-guided Control Prediction for End-to-end Autonomous Driving. The approach combines trajectory planning and direct control in a single learning pipeline. It features a multi-step control prediction branch with a temporal module and trajectory-guided attention, allowing for temporal reasoning. Outputs from both branches are fused to leverage their complementary advantages. The perception system processes data from a single monocular camera input.

TF++ WP ensemble The methodology tackles two critical biases in end-to-end driving systems: lateral recovery via target point following and longitudinal averaging of multimodal waypoint predictions for slowing down. It predicts network uncertainty through target speed classification, whereby uncertainty-weighted speed interpolation reduces collisions by leveraging the continuous nature of waypoint output representations to reduce collisions.

LAV The approach, Learning from All Vehicles, trains driving policies by leveraging data from all observed vehicles to generate diverse scenarios. It addresses the absence of direct sensor data by employing supervised tasks to create viewpoint-invariant representations. This framework separates perception and action challenges, distilling a perception model for invariant representations and a motion planner using privileged computer-vision labels. LAV then integrates these components into a unified model for robust driving policy learning from raw sensor inputs.

```
-- number of lanes
#define MAX_LANE 2
-- discrete time step in seconds
#define TIME_STEP 1
-- safety distance between two cars
#define SAFE_DISTANCE 7
-- initial distance between any two cars
#define INIT_DISTANCE 5
-- physical constraints on the maximum braking and acceleration
-- that are realistic (expressed in m/s^2). Averaged (eyeballed)
-- from literature.
#define MAX_BRAKING -4.6
#define MAX_ACCELERATION 5.6
-- maximum speed for cars in m/s
#define MAX_SPEED 12
-- maximum cruising speed of ego vehicle in m/s
#define EGO_CRUISE_SPEED 5
-- minimum interval between two lane changes of the same car
#define LANE_CHANGE_DURATION 6
```

Fig. 4. Defined global constants for symbolic abstraction of the ADS model.

4.2. Abstract model

We specify our abstract model as a synchronous symbolic transition system written in the language of the nuXmv [11] model checker. The model consists of 3 vehicles (one "ego" car, representing the autonomous system under test, and two other cars) moving on a highway with 3 lanes. The vehicles all drive in the same direction. The ego is constrained to stay in the middle lane and tries to maintain a given cruise speed, braking when necessary to avoid collisions with other cars, and possibly accelerating to reach the target speed. The other two "non-ego" cars can move freely on the highway, with arbitrary accelerations, braking, and lane change maneuvers (subject to physical constraints about min/max acceleration rates and speed limits, taken from publicly available online car databases), but are not allowed to crash into each other or the ego. We use a discrete model of time, in which each transition of the system corresponds to a time elapse of 1 second. A shorter time step would improve the precision of generated abstract scenarios, however, at the cost of computational overhead. We found the choice of 1 second time step to be adequate enough for an effective concretization of the abstract scenarios on the real system. We use the theory of real arithmetic to encode the transition relation of the system, using mostly linear constraints to compute the updates to the speed and locations of the vehicles (thanks to the discretization of time).

We first define here some fixed parameters for simple, but meaningful scenario generation. A list of macros is given in Fig. 4. These parameters are fixed during the compilation time, shared by all the modules of the SMV program during execution. Our symbolic model is composed of three modules: Car, Ego, and main. Below we explain these modules in detail.

4.2.1. Module: Car

An excerpt of the symbolic model of the `Car` module is shown in Fig. 5. This is a generic module, describing different transition constraints and constraints on speed, acceleration, and position, which need to hold for any car in the environment. This module takes as input an identifier (*id*), which can be utilized to instantiate its multiple instances within a program.

Lines 2-4 define frozen variables, including the maximum number of lane changes and initial position for a car. These variables retain their initial values throughout the evolution of the state machine. Their initial values can either be arbitrarily chosen by the model checker, or specified using an INIT constraint. Lines 5-12 define the state variables, whose values are chosen freely by the model checker during execution, except for their initial states, if provided. Here, we represent the vehicle's navigation on a 2D coordinate plane using $(pos, lane)$. The longitudinal position is represented by, *pos*, while the lateral position is abstracted to one of the lanes, given by the integer enumerated type state variable, *lane*. Additional predicates *speed*, *acceleration*, and *target_speed* are provided for the longitudinal direction.

In lines 13-15, the position of a car is initialized to the constant *initial_pos*, with the initial state of number of lane changes assigned to 0. Lines 18-31 describe different transition constraints that need to hold for longitudinal movements. The vehicle reaches its target speed following Newtonian physics, with the acceleration being constrained to be within its bounds which are defined globally. Line 31 adds the invariant constraints on the position and speed to constrain the movement to only forward direction, and limit the maximum speed that could be reached by a car.

Lines 34-52 specify different constraints that need to hold for the lateral and longitudinal movement of a car while changing lanes. A macro, *changing_lane* (line 34), evaluates to true when the next lane differs from the current one. A car is not allowed to accelerate while changing lanes (line 35), and can change only one lane at a time (line 37), with each lane change taking a single time step. The total number of lane changes by a car during an execution is limited by the constant *max_lane_changes* (line 38), following the transition in lines 39-41. The longitudinal position update of the car follows Newtonian physics (lines 42-45). To account for the lateral movement while changing lanes, the *pos* update is reduced by 5%, a value chosen heuristically as the symbolic model assumes lanes with no width. Lines 46-52 impose a minimum interval of 6 time steps between consecutive lane changes for any car. Shorter intervals prevent realistic execution of lane-change maneuvers in the real system.

```
1  MODULE Car(id)
2  FROZENVAR
3    max_lane_changes : integer;
4    initial_pos : real;
5  VAR
6    pos : real;
7    acceleration : real;
8    target_speed : real;
9    lane : 0 .. MAX_LANE;
10   speed : real;
11   num_lane_changes : integer;
12   change_lane_countdown : 0 .. LANE_CHANGE_DURATION;
13 INIT
14   pos = initial_pos;
15   num_lane_changes = 0;
16
17 -- updates to speed
18 TRANS
19   MAX_BRAKING <= acceleration & acceleration <= MAX_ACCELERATION;
20 TRANS
21   next(speed) = max(speed + acceleration * TIME_STEP, 0);
22 TRANS
23   target_speed >= 0;
24 TRANS
25   case
26     speed < target_speed : next(speed) >= speed;
27     speed > target_speed : next(speed) <= speed;
28     TRUE : next(speed) = speed;
29   esac;
30 INVAR
31   speed >= 0 & pos >= 0 & speed <= MAX_SPEED;
32
33 -- lane change
34 DEFINE changing_lane := next(lane) != lane;
35 TRANS changing_lane -> (acceleration = 0);
36 TRANS
37   (next(lane) = lane) | (next(lane) = lane + 1) | (next(lane) = lane - 1);
38 INVAR (num_lane_changes <= max_lane_changes);
39 TRANS
40   changing_lane ? (next(num_lane_changes) = num_lane_changes + 1) :
41   (next(num_lane_changes) = num_lane_changes);
42 TRANS
43   changing_lane ?
44   (next(pos) = pos + (speed + next(speed)) / 2 * TIME_STEP * 0.95) :
45   (next(pos) = pos + (speed + next(speed)) / 2 * TIME_STEP);
46 TRANS
47   case
48     changing_lane : next(change_lane_countdown) = LANE_CHANGE_DURATION;
49     change_lane_countdown = 0 : next(change_lane_countdown) = 0;
50     TRUE : next(change_lane_countdown) = change_lane_countdown - 1;
51   esac;
52 TRANS changing_lane -> change_lane_countdown = 0;
```

Fig. 5. Excerpt of the nuXmv code for the Car module.

In principle, *lane* could be defined as a real-valued variable, accompanied by a more detailed vehicle dynamics model, to compute both longitudinal and lateral motion more accurately. However, introducing nonlinear constraints would significantly increase the computational overhead for the model checker. We argue that the chosen level of abstractions is adequate enough to generate meaningful scenarios which can be effectively concretized for real-world systems through a mapping from the symbolic model to physical implementation.

4.2.2. Module: Ego

Ego module models the key behaviors required for the autonomous vehicle to navigate safely in the environment. In real-world systems, these behaviors are typically implemented as black-box AI components, with each expert employing different methods. Here, we provide a high-level abstraction of the ego vehicle, designed to reflect how AI generally models its behavior in the real world. This abstraction is sufficiently general to encompass a variety of AI-based methods. An excerpt of the nuXmv code for the Ego module is given in Fig. 6. The module takes as input the states of all the non-ego vehicles in the environment, specifically *car1* and *car2* in this instance. This setup assumes a "perfect" perception system, where the ego vehicle has complete and accurate knowledge of its environment.

To inherit the methods and behaviors defined in the generic Car module, the Car module is instantiated as a state variable *me* within Ego (line 3), and is identified by the value 0. Lines 15-16 define the collision condition (*collision_next*) with non-ego vehicles. Following Newtonian physics, the time required for the ego to come to a complete stop with maximum braking (*time_to_stop*) is calculated (line 14). If this time exceeds or equals the time to collision with a non-ego vehicle in front at the current speed, the

```
1  MODULE Ego(car1, car2)
2
3  VAR me : Car(0);
4
5  DEFINE
6    pos := me.pos;
7    x := me.pos;
8    lane := me.lane;
9    speed := me.speed;
10   acceleration := me.acceleration;
11   target_speed := me.target_speed;
12   initial_pos := me.initial_pos;
13   max_lane_changes := me.max_lane_changes;
14   time_to_stop := speed / (-MAX_BRAKING);
15   collision_next := (car1.lane = lane & car1.pos >= pos & speed > 0 & (car1.pos - pos) / speed <= time_to_stop) |
16                     (car2.lane = lane & car2.pos >= pos & speed > 0 & (car2.pos - pos) / speed <= time_to_stop);
17
18 -- collision avoidance and highway cruising
19 TRANS
20   collision_next ?
21   (acceleration = MAX_BRAKING & target_speed = 0) :
22   (target_speed = EGO_CRUISE_SPEED &
23   ((speed < target_speed) -> (next(speed) = min(target_speed, speed + MAX_ACCELERATION * TIME_STEP))));
24
25 -- lane keeping
26 INVAR me.max_lane_changes = 0;
```

Fig. 6. Excerpt of the nuXmv code for the Ego module.

```
1  MODULE main
2  VAR
3    car1 : Car(1);
4    car2 : Car(2);
5    ego : Ego(car1, car2);
6  DEFINE
7    distance_ego_car1 := abs(ego.pos - car1.pos);
8    distance_ego_car2 := abs(ego.pos - car2.pos);
9    distance_car1_car2 := abs(car1.pos - car2.pos);
10 INIT -- the cars are not overlapping with each other initially
11   ((ego.lane = car1.lane) -> (distance_ego_car1 > INIT_DISTANCE)) &
12   ((ego.lane = car2.lane) -> (distance_ego_car2 > INIT_DISTANCE)) &
13   ((car1.lane = car2.lane) -> (distance_car1_car2 > INIT_DISTANCE)) &
14   TRUE;
15 INVAR -- the cars do not crash into each other on purpose
16   ((distance_ego_car1 > SAFE_DISTANCE) | (ego.lane != car1.lane)) &
17   ((distance_ego_car2 > SAFE_DISTANCE) | (ego.lane != car2.lane)) &
18   ((distance_car1_car2 > SAFE_DISTANCE) | (car1.lane != car2.lane)) &
19   TRUE;
20 TRANS
21   ((ego.lane = car1.lane & next(ego.lane = car1.lane)) ->
22         ((ego.pos >= car1.pos) <-> next(ego.pos >= car1.pos))) &
23   ((ego.lane = car2.lane & next(ego.lane = car2.lane)) ->
24         ((ego.pos >= car2.pos) <-> next(ego.pos >= car2.pos))) &
25   ((car1.lane = car2.lane & next(car1.lane = car2.lane)) ->
26         ((car1.pos >= car2.pos) <-> next(car1.pos >= car2.pos))) &
27   TRUE;
28 INIT
29   agent_ego.x = 0 &
30   agent_car1.x = 0 &
31   agent_car2.x = 0 &
32   agent_car1.lane = 0 &
33   agent_car2.lane = 2 &
34   agent_ego.lane = 1 &
35   agent_ego.speed = 0 &
36   agent_car1.speed = 0 &
37   agent_car2.speed = 0;
```

Fig. 7. Excerpt of the nuXmv code for the Main module.

collision_condition is set to true. In such cases, the ego applies maximum braking (MAX_BRAKING) until it stops; else it continues with (or reach towards) its *target_speed* using maximum acceleration (MAX_ACCELERATION), where *target_speed* is assigned the value of EGO_CRUISE_SPEED (lines 19-23). Since lane changes are not supported for the ego vehicle in the real simulator, we enforce a constraint that the ego must never change its lane. This is implemented as an invariant (line 26).

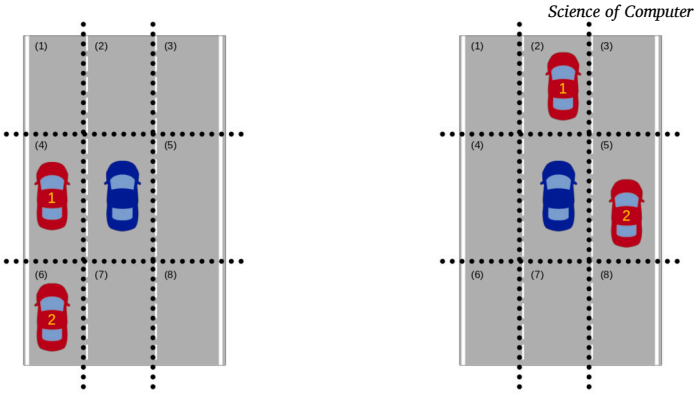


Fig. 8. Example traffic situations for constructing abstract scenarios, specifying positions of non-ego cars (in red) in terms of the occupation of cells of an abstract 3x3 grid centered on the ego car (in blue). (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

4.2.3. Module: main

The `main` module serves as the top-level module, defining the entire transition system. An excerpt of the nuXmv code for this module is shown in Fig. 7. All previously defined modules must be instantiated as variables within this module to be included during execution. In this setup, we instantiate two non-ego vehicles, *car1* and *car2*, inheriting from the `Car` module with identifiers 1 and 2, respectively. In line 5, the *ego* vehicle is instantiated, inheriting from the `Ego` module. Lines 6–9 calculate the current absolute distances between all vehicles in the environment. Lines 10–14 ensure that no two vehicles overlap in the initial state, in case they are instantiated in the same lane. Lines 15–19 introduce an invariant constraint to prevent collisions between vehicles during any execution, thereby generating only safe executions. The transition constraint in lines 20–27 ensures that vehicles do not "jump" past each other in a single transition if they are in the same lane and remain in the same lane in the subsequent step. Lines 28–37 define the initial positions and speeds for all vehicles. Here, all vehicles start stationary next to each other, with the *ego* vehicle in the center lane ($lane = 1$) and the non-ego vehicles occupying the adjacent lanes.

4.3. Coverage criterion

In order to enumerate abstract scenarios encoding potentially-interesting traffic situations, we define for each non-ego car a set of predicates specifying its position relative to the ego, in terms of occupation of cells of an abstract 3x3 "grid" centered on the ego. Examples of the possible configurations that can be expressed in this way are shown in Fig. 8. We then define an *abstract scenario* as a combination of constraints about the different positions of the non-ego cars on the grid at different points in time. More specifically, each abstract scenario is specified as an LTL property of the following form:

$$\begin{aligned} &\neg\mathbf{F}(\text{car1\_grid\_pos} = \text{CELL\_A1} \wedge \text{car2\_grid\_pos} = \text{CELL\_A2} \wedge \\ &\quad \mathbf{X}(\mathbf{F}(\text{car1\_grid\_pos} = \text{CELL\_B1} \wedge \text{car2\_grid\_pos} = \text{CELL\_B2}))), \end{aligned} \tag{1}$$

(where car$i$_grid_pos encodes the position of the i-th non-ego car in the grid and CELL_$*$ represent possible target positions for the cars). By asking the model checker to find a counterexample to Eq. (1), we generate traces in which the non-ego cars first reach the configuration with *car1* in position *A1* and *car2* in position *A2*, and then subsequently move to the configuration with *car1* in position *B1* and *car2* in position *B2*, performing the necessary maneuvers while avoiding collisions with each other or with the ego.

Dynamic coverage predicates The cells in the abstract 3x3 "grid" represent the 2D space for the possible target locations of the non-egos. This space is dynamic, i.e., the dimensions of these cells may vary during the course of simulation, depending on the minimum braking distance of ego vehicle at each time step. In Eq. (2), we show examples of mapping the positions of non-egos to the dynamic cell locations 1 and 4, and static cell location 8 of the abstract grid.

$$\begin{aligned} &\textbf{CELL\_1}: (\text{car\_i.lane} < \text{ego.lane}) \wedge \\ &\qquad (\text{ego.braking\_dist} + \text{safe\_dist} \leq \text{car\_i.pos - ego.pos} \leq \text{ego.braking\_dist} + 3 * \text{safe\_dist}) \\ &\textbf{CELL\_4}: (\text{car\_i.lane} < \text{ego.lane}) \wedge \\ &\qquad (\text{car\_i.pos} \leq \text{ego.pos} + \text{ego.braking\_dist} + \text{safe\_dist}) \wedge \\ &\qquad (\text{car\_i.pos} \geq \text{ego.pos} - \text{safe\_dist}) \\ &\textbf{CELL\_8}: (\text{car\_i.lane} > \text{ego.lane}) \wedge \\ &\qquad (\text{safe\_dist} \leq \text{ego.pos} - \text{car\_i.pos} \leq 3 * \text{safe\_dist}) \wedge \\ &\qquad (\text{car\_i.pos} < \text{ego.pos}), \end{aligned} \tag{2}$$

where car_i.pos is the longitudinal position (in meters) of the i-th car in the abstract trace (and similarly for ego.pos). In this way, we define the boundaries of the cells on the abstract grid. Only the cell boundaries ahead of the ego vary at each time-step depending on ego's braking distance, as inferred from the equation above. Here, safe_dist = SAFE_DISTANCE (from Fig. 4), braking_dist is the distance covered by ego while braking with MAX_BRAKING (from Fig. 4) from its current position, computed using Newtonian physics in Eq. (3):

$$\text{braking\_dist} = \frac{(\text{ego.speed})^2}{2 * \text{MAX\_BRAKING}} \tag{3}$$

The space of scenarios that is being explored therefore consists of all the possible combinations of transitions from configurations of the non-ego cars in terms of their position in the grid defined above. Enumerating all of them would give 4096 scenarios. We define our coverage criterion by selecting a subset of *abstract scenarios of interest*, consisting of various combinations of the traffic situations that can be modeled by positioning the non-ego cars in the grid around the ego. In total, for the experiments, we defined 144 such interesting scenarios. Specifically, we considered breaking the symmetry in non-egos' positions to choose these interesting scenarios.

In our previous work [12], the coverage criteria consisted of static coverage predicates, independent of ego's braking distance. To improve the coverage results we presented in that paper, we introduce here a refined notion of coverage criteria as described above, linking it more tightly with the dynamics of ego vehicle.

4.4. Concrete scenario generator

This section details the integration of the VIVAS framework with the CARLA simulator, focusing on the automated generation of CARLA-specific input files tailored for the simulation environment. We utilize the CARLA module, ScenarioRunner, as the primary interface with the simulator. As illustrated in Fig. 3, ScenarioRunner accepts inputs such as the route for the AI-driven ego vehicle and a traffic scenario that includes the behaviors of non-ego actors. In the following, we provide an in-depth discussion on how VIVAS interfaces with the CARLA simulator to automatically generate concrete scenarios.

4.4.1. Traffic scenario

In order to generate a traffic scenario for the simulator, the abstract counterexample trace generated by the model checker is parsed for relevant information to be fed as input to the simulator. Every state of the abstract scenario trace is concretized into the corresponding behavior of every non-ego agent. Each behavior is then specified in Python to generate a sequential behavior tree for each corresponding non-ego vehicle, using the *py_trees* library. The behavior trees of all the non-egos present in the environment are then run in parallel during the simulation. The entire process of parsing the abstract scenario and translating it into concrete scenarios is fully automated.

We first parse the initial coordinates and lanes of all the non-egos relative to the ego to instantiate them on the map. In each behavior of a behavior tree, the corresponding non-ego has to drive at a certain speed for a certain distance, following the waypoints given by the map on the same lane it is instantiated on. While each state transition in the abstract trace lasts for 1 second, actual travel times in the CARLA simulator may differ due to discrepancies between the symbolic and simulator models. In the case in which a vehicle remains stationary for n states in the abstract trace, it remains stationary for n seconds in the concrete scenario upon halting.

In the symbolic model, as described in §4.2.1, lane changes occur in one time step, with zero lateral distance traveled (since lanes have no width in the symbolic world). However, we enforce a constraint where successive lane changes by the same vehicle must be separated by N steps apart[3] to model the fact that lane changes are not instantaneous in reality. When concretizing such state transitions, a non-ego vehicle traverses 9 m during a lane change, with this behavior terminating after it has traveled a total distance of 12 m before the next behavior in the tree is instantiated. Lane changes with smaller travel distances were not feasible in CARLA given the speed ranges of the vehicles in our scenarios. Similar to the symbolic model, a non-ego can change only one lane at a time, with inputs {left, right} representing a lane change to the left or right, respectively.

We leverage the behavior library of ScenarioRunner to implement these atomic behaviors and trigger conditions. Specifically, we use the atomic behaviors: *WaypointFollower* and *LaneChange*; and the atomic trigger conditions: *DriveDistance* and *StandStill*, from the ScenarioRunner library. An example behavior tree for one non-ego vehicle (*car1*), incorporating the three behaviors described above, is shown in Fig. 9. In line 2, we instantiate a sequential behavior tree for *car1* using the *composites* class of *py_trees*. In lines 6-7, a parallel behavior tree, *drive_car1*, is created to execute its children concurrently. Line 8 adds the atomic behavior *WaypointFollower* as a child of *drive_car1*, enabling *car1* to drive straight at a speed of 3 m/s. The atomic trigger condition, *DriveDistance* is added as a child to *drive_car1* in line 10, terminating this behavior once *car1* has traveled for 2.6 m. In line 11, *drive_car1* is added as the first child in the sequential behavior tree of *car1_behavior*. Similarly, lines 14-20 define a lane change behavior for *car1*, which is added to the sequence in *car1_behavior*. This behavior allows *car1* to change lanes to the left at a speed of 2 m/s, traveling 9 m during the lane change. The behavior terminates once the vehicle has traveled a total of 12 m. Lines 23-26 specify a standstill behavior for *car1*, where the vehicle remains stationary for 1 second. This behavior is also added as a child in the *car1_behavior* tree.

We do not extract the ego's behavior from the abstract trace, since it is expected to make decisions autonomously in the simulator. Only initial spawn position and destination are extracted for the AI-based agent to navigate along the specified route.

[3] We used $N = 6$ in our experiments.

```python
# instantiate car1's sequential behaviors
car1_behavior = py_trees.composites.Sequence("car1_behavior")

# first behavior in the sequence for car1: to drive forward.
##   Atomic behavior WaypointFollower runs in parallel with an atomic trigger condition DriveDistance
drive_car1 = py_trees.composites.Parallel("Drive forward",
                policy=py_trees.common.ParallelPolicy.SUCCESS_ON_ONE)
drive_car1.add_child(WaypointFollower(car1, speed=3, avoid_collision=False))
## drive_car1 terminates when car1 reaches the given distance
drive_car1.add_child(DriveDistance(car1, distance=2.6))
car1_behavior.add_child(drive_car1)

# second behavior in the sequence for car1: to change lane
lane_change_car1 = py_trees.composites.Parallel("lane change",
                  policy=py_trees.common.ParallelPolicy.SUCCESS_ON_ONE)
## change lanes to the left with a speed of 2 m/s
lane_change_car1.add_child(LaneChange(car1, speed=2, direction='left',
            distance_same_lane=1.3, distance_other_lane=60, distance_lane_change=9))
lane_change_car1.add_child(DriveDistance(car1, distance=12))
car1_behavior.add_child(lane_change_car1)

# third behavior in the sequence for car1: to stand still for 1 second.
still_car1 = py_trees.composites.Parallel("standstill",
                policy=py_trees.common.ParallelPolicy.SUCCESS_ON_ONE)
still_car1.add_child(WaypointFollower(car1, speed=0.0, avoid_collision=False))
still_car1.add_child(StandStill(car1, name="StandStill", duration=1.0))
car1_behavior.add_child(still_car1)
```

Fig. 9. Excerpt of automatically generated Scenario runner code (in Python3) for a non-ego (car1) behavior. Comments (in green) explain the behavior tree.

4.4.2. Ego's route

Alongside the traffic scenario, as described above, we need to provide as input an ego route file (in XML format) in order to instantiate a scenario in CARLA. Specifically, we need to: (a) load a specific map; (b) provide initial spawn locations on that map for an ego vehicle; (c) provide a goal location for the ego to navigate to, which also serves as a scenario termination condition in CARLA; and (d) provide intermediate locations (if needed) which must be visited by the ego. These locations are defined in terms of waypoints, which are the global 3D coordinates on the specified map. We use a map with a long straight highway section, coherently with our abstract model. We instantiate ego at the beginning of one section of a highway with 5 straight lanes. Due to the approximations in mapping between the abstract and concrete scenarios, specifically in lane changing behaviors, non-egos may travel longer distances at the concrete level compared to the abstract scenario. Hence, irrespective of the distance traveled in the abstract scenario, we fix the goal location of ego to be 200 meters straight ahead of its initial position. Since the abstract scenarios we generate may involve a maximum of four lane change maneuvers by each non-ego vehicle, this goal distance is adequate enough for the scenario to be fully executed at the concrete level which we confirmed in our evaluations. The concrete scenario in CARLA is terminated once any of the following two conditions are met, (a) ego reaches its goal location, (b) scenario reaches a timeout, i.e., the "Game time" (duration of simulation trace) of CARLA simulator.

4.4.3. Weather conditions

CARLA also provides the capability to specify different weather conditions for scenarios within the same XML file alongside the ego vehicle's route. The WeatherParameters class can be instantiated with various variables to define the levels of lighting and other environmental conditions, such as cloudiness, precipitation, precipitation deposits, rain, sun angle, altitude, and more. Each abstract scenario can theoretically be instantiated into an infinite number of concrete scenarios with varying weather parameters. However, in practice, this is not feasible. To effectively challenge the AI-based perception and decision-making components of the ego vehicle, we specify four distinct weather conditions, detailed in Table 2. As a result, each abstract scenario is concretized into four unique concrete scenarios corresponding to these weather conditions.

While it is theoretically possible to model a weather component in our symbolic model, linking it with the ego vehicle's dynamics to create different abstract scenarios (e.g., reduced MAX_BRAKING, leading to longer braking times in slippery conditions), this would allow us to monitor the AI-based ego vehicle's behavior in greater detail against our abstract model. However, in CARLA, these weather conditions only affect the RGB cameras and do not influence the physics of the actors or other sensors. Therefore, we concretize each abstract scenario for different weather conditions instead.

4.5. Execution monitor

The monitor component of VIVAS is used to determine whether the concrete system (simulator) satisfies the system-level formal specification. For the automotive application, the simulation output traces include sequences of all states and actions executed by the ego vehicle, along with the time evolution of other observable parameters, which must be checked for property satisfaction/violation. These simulation traces are mapped back to the abstract trace for evaluating the corresponding predicates for property evaluation and measuring the concrete coverage.

An illustrative example of mapping simulation execution traces to abstract traces for property evaluation and coverage measurement is shown in Fig. 10. In this example, the simulation trace captures key data such as the positions and lanes of both the ego vehicle

Table 2
Weather conditions for different concrete scenarios.

Variables	Ranges	Weathers			
		Wet Morning	Sunset	Clear Noon	Night Rain
cloudiness	Cloud cover: [0, 100]	5	0	0	90
precipitation	Rain intensity: [0, 100]	0	0	0	100
precipitation deposits	Puddles on road: [0, 100]	80	0	0	100
sun azimuth angle	[0, 360]	0	180	90	225
sun altitude angle	[-90, 90]	10	20	70	-90
wetness	[0, 100]	30	0	0	0
Challenge to Ego:		Sun right in front of ego; sun's reflection in the puddles.	Sun right behind ego, reflecting hard on non-egos' in front.	Sun straight up: ideal condition.	Hard rain at night, visibility through headlamps and streetlights.

and car1, ego's braking distance, and collision sensor output, all recorded at various timestamps. For clarity, the details of car2 and intermediate timestamps have been omitted to focus on the main scenario: $\neg\mathbf{F}(\text{car1\_grid\_pos} = 6 \wedge \mathbf{X}(\mathbf{F}(\text{car1\_grid\_pos} = 2)))$. This scenario reflects the requirement that car1 should first reach grid position 6, followed by eventually reaching grid position 2 in future time steps. The variables from the simulation trace are systematically mapped to the predicates in the abstract trace using logical formulas, as defined in the Mapping block. These abstract predicates serve as inputs to the runtime monitor, which evaluates them offline (once the simulation output has been recorded completely) to detect any violations of the specified properties and calculate concrete coverage. Further details on how the runtime monitor processes these predicates to check property violations and ensure sufficient coverage are provided in the subsequent sections.

4.5.1. Monitoring of properties

In this study, we primarily need to check whether the ego vehicle crashes with another vehicle in the environment. Since ego always travels in its own lane, we limit the check for the case when ego crashes with any non-ego in front of it in its own lane. We do not consider rear-end collisions involving the ego vehicle, as the vehicle in front is generally not considered at fault under typical road rules. Furthermore, any scenario following such a crash would no longer conform to the intended abstract scenario. We leverage the continuous data stream from the collision sensor mounted on the ego to detect these collisions. We define a predicate, *crash*, which is True when the collision sensor detects a collision. We further need to check whether the ego reaches its goal destination, as discussed in §4.4.2. We define a predicate, *goal*, which is True as soon as ego reaches it's destination, i.e., traveling more than 200 m. A runtime monitor based on NuRV [28,29] is utilized to check standard LTL properties on ego behavior, to check for route completion and collisions. The LTL properties are given in Eq. (4).

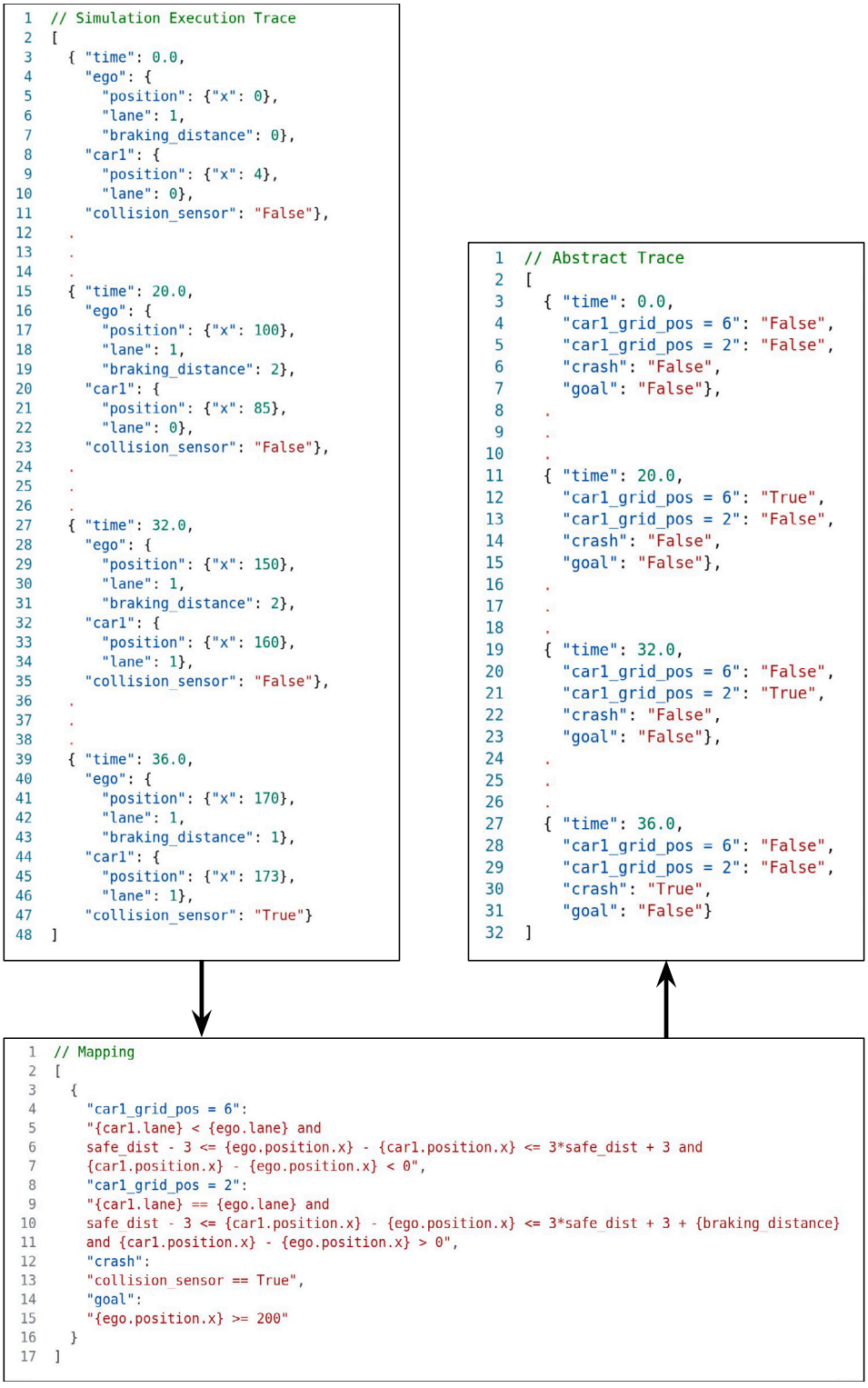
$$\begin{aligned} \phi_1 &:= \mathbf{F}(goal) \\ \phi_2 &:= \mathbf{G}(\neg crash) \end{aligned} \tag{4}$$

4.5.2. Abstract and concrete coverage

The concrete simulation traces are mapped back to the abstract trace to measure coverage, as illustrated in Fig. 10, to check if the same sequence of scenes was encountered in the concrete scenarios or not. The predicate map shown in the mapping block translates the absolute positions of the non-egos in the concrete simulation trace to the abstract 3x3 grid shown in Fig. 8.

In the symbolic model, vehicles are represented as points, whereas in the simulator, they are modeled as boxes. Although we consider only the center positions of vehicles to map back concrete simulation executions, the ego vehicle's decision-making module takes into account the bounding-box dimensions of both the ego and non-ego vehicles for collision-avoidance maneuvers. To mitigate the error arising from this difference in representation, we expand the size of each cell in the abstract grid by 3 meters (intuitively, less than one car length) in both the longitudinal directions. This adjustment, as incorporated in the mapping block of Fig. 10, ensures a more accurate correspondence between the concrete simulation executions and their abstract counterparts.

For the scenario depicted in Fig. 10, the predicates car1_grid_pos = 6 and car1_grid_pos = 2 in the resulting abstract trace are checked by the monitor at every time instant, producing boolean values for each abstract condition. Based on the predicate mapping, car1_grid_pos = 6 : $\top$ and car1_grid_pos = 2 : $\bot$ at time = 20 s. At time = 32 s, car1_grid_pos = 6 : $\bot$ and car1_grid_pos = 2 : $\top$, hence verifying the concrete coverage of the abstract scenario.



```
// Simulation Execution Trace
[
  { "time": 0.0,
    "ego": {
      "position": {"x": 0},
      "lane": 1,
      "braking_distance": 0},
    "car1": {
      "position": {"x": 4},
      "lane": 0},
    "collision_sensor": "False"},
  .
  .
  .
  { "time": 20.0,
    "ego": {
      "position": {"x": 100},
      "lane": 1,
      "braking_distance": 2},
    "car1": {
      "position": {"x": 85},
      "lane": 0},
    "collision_sensor": "False"},
  .
  .
  .
  { "time": 32.0,
    "ego": {
      "position": {"x": 150},
      "lane": 1,
      "braking_distance": 2},
    "car1": {
      "position": {"x": 160},
      "lane": 1},
    "collision_sensor": "False"},
  .
  .
  .
  { "time": 36.0,
    "ego": {
      "position": {"x": 170},
      "lane": 1,
      "braking_distance": 1},
    "car1": {
      "position": {"x": 173},
      "lane": 1},
    "collision_sensor": "True"}
]
```

```
// Abstract Trace
[
  { "time": 0.0,
    "car1_grid_pos = 6": "False",
    "car1_grid_pos = 2": "False",
    "crash": "False",
    "goal": "False"},
  .
  .
  .
  { "time": 20.0,
    "car1_grid_pos = 6": "True",
    "car1_grid_pos = 2": "False",
    "crash": "False",
    "goal": "False"},
  .
  .
  .
  { "time": 32.0,
    "car1_grid_pos = 6": "False",
    "car1_grid_pos = 2": "True",
    "crash": "False",
    "goal": "False"},
  .
  .
  .
  { "time": 36.0,
    "car1_grid_pos = 6": "False",
    "car1_grid_pos = 2": "False",
    "crash": "True",
    "goal": "False"}
]
```

```
// Mapping
[
  {
    "car1_grid_pos = 6":
    "{car1.lane} < {ego.lane} and
    safe_dist - 3 <= {ego.position.x} - {car1.position.x} <= 3*safe_dist + 3 and
    {car1.position.x} - {ego.position.x} < 0",
    "car1_grid_pos = 2":
    "{car1.lane} == {ego.lane} and
    safe_dist - 3 <= {car1.position.x} - {ego.position.x} <= 3*safe_dist + 3 + {braking_distance}
    and {car1.position.x} - {ego.position.x} > 0",
    "crash":
    "collision_sensor == True",
    "goal":
    "{ego.position.x} >= 200"
  }
]
```

Fig. 10. Example of mapping a simulation trace to the resulting abstract trace.

5. Results

In this section, we report on our experimental evaluation of our instantiation of VIVAS for the ADS application using the CARLA simulator. We first describe the experimental setup in §5.1, including the choice of parameters for the vehicles and environment in the symbolic model and in the CARLA simulator, necessary to generate meaningful scenarios. The details of computational performance are reported in §5.2. We then present the results of the evaluation in §5.3 and discuss them in §5.4.

5.1. Experimental setup

5.1.1. Abstract scenario generation

All abstract scenarios that we generate consist of 2 non-ego agents and one ego agent, all of which start from the same longitudinal position, 0 m, with ego in the center lane and 2 non-egos on each side of it. Note that fixing the initial positions would not make a difference to the abstract scenario generation since the acceleration, braking, and speed for the non-egos are picked non-deterministically by the model checker for every time step, while respecting the above bounds. Since all four AI-based agents we benchmarked in CARLA can only drive at maximum speed of 5 m/s, we limit the ego agent cruising speed to the same. Since the benchmarked solutions do not provide the vehicle dynamics data, we use acceleration values based on the literature. All the agents start from 0 m/s, with ego reaching its cruising speed with MAX_ACCELERATION. To avoid collisions, it brakes with MAX_BRAKING to maintain at minimum the SAFE_DISTANCE with all the non-egos.

5.1.2. Concrete scenario generation

To mimic the symbolic environment model, we instantiate the CARLA simulator on a straight, five-lane highway section from its Town06 map. The ego vehicle is positioned in the center lane, with two non-ego vehicles in the adjacent lanes, corresponding to the symbolic model setup. Although the symbolic model was designed with three lanes, we chose the five-lane highway in Town06 because it offers the longest straight stretch among all available CARLA maps, which is essential for executing extended scenarios. To better align with the symbolic model, we excluded the left-most and right-most lanes from use.

As mentioned in §4.4.3, each abstract scenario is concretized into 4 different weather conditions: wet morning, sunset, clear noon, and night rain. To compensate for the mismatch between the vehicle dynamics in the symbolic model and CARLA simulator, we further concretize the abstract scenarios with three different initial positions of the non-egos with respect to ego at:

$$x = \{-4, 0, 4\}\ m \tag{5}$$

i.e., both the non-egos start at 4 m behind, same level, and 4 m ahead of the ego in their respective lanes in different simulation runs. 4 m is the average length of the non-ego vehicles in our simulations. All the vehicles start from 0 m/s, as parsed from the abstract trace.

5.2. Performance

We ran the experiments on an Intel i7 with NVIDIA GeForce RTX 2080 8 GB GPU. These are the minimum hardware requirements to run the simulations on the CARLA simulator with AI models. We used a timeout of 1000 seconds for each abstract scenario generation. We use the Bounded Model Checking (BMC) algorithm of the nuXmv [11] symbolic model checker to produce abstract counterexamples. Due to the presence of non-linear constraints in our symbolic model, specifically in computing ego's braking distance, we leverage the z3 SMT solver [30] for checking the satisfiability of the BMC queries, as it provides better performance than the default solver of NuXmv, MathSAT [31]. Out of 144 abstract scenarios, the model checker reached the timeout for 2 scenarios. With an increased timeout value, these two scenarios could be generated by the model checker within 1300 seconds.

Simulation executions in CARLA took 5-10 minutes on average depending on AI agent, which includes time to instantiate a scenario in CARLA and performing the simulations. We used a timeout of 100 seconds in terms of the Game Time (simulation length) to terminate the scenario execution, which is reached in cases when ego is stuck behind a static non-ego in front in its lane. Different AI-agents we have benchmarked have different computational performances. On an average, the ratio of System time (real time) and Game time of different agents is given in Table 3.

The code and data necessary for reproducing our experiments are available at https://es-static.fbk.eu/people/sgoyal/afmas23.

Table 3
Performance of different AI agents in Carla simulator.

AI Agent	Interfuser	TCP	TF++WP Ensemble	LAV
Average (Game time / System time)	0.19	0.32	0.43	0.51

5.3. Evaluation

The model checker produces a total of 144 abstract scenarios based on the coverage criteria given in §4.3. Each abstract scenario is concretized into 12 concrete scenarios, consisting of 4 different weather conditions and three different initial positions of the non-egos according to Eq. (5), which gives us $144 * 4 * 3 = 1728$ concrete scenario outputs from the simulator for each AI agent. The evaluation results are shown in Table 4. The columns have the following meanings:

1. **Property status**: We check the violation of system-level properties, ϕ_1 and ϕ_2, defined in Eq. (4). In particular, we check the following cases:
   (a) $\neg\phi_1$: the autonomous ego agent does not reach its goal position.
   (b) $\neg\phi_2$: the autonomous ego agent collides with at least one non-ego in front. Situations where non-egos crash into each other or hit the ego from behind are not taken in account, as mentioned in §4.5.1.
   (c) $\phi_1 \wedge \neg\phi_2$: the autonomous ego agent collides with at least one non-ego in front, but still reaches the goal.
2. **Coverage OK**: Each point of the grid of coverage criteria represents a scenario with a fixed order of scenes. The concrete simulation run passes ("OK") if the abstract scenario generated by the model checker could indeed be generated on the simulator as well.
3. **Coverage OK** $\cap \neg\phi_2$: These are the set of "interesting" cases (along with the other cases where ego crashed), where the coverage criteria passed, but the ego crashed with a non-ego in front.
4. **Set Union**: This operation aggregates the results of all concrete scenarios corresponding to each abstract scenario. Given that one abstract scenario is concretized into 12 different concrete scenarios, the set union of their results is taken. If the union contains at least one passing concrete scenario, the corresponding abstract scenario is considered to have passed the test.

5.4. Analysis

As we see from the obtained results, not all the abstract scenarios generated by the model checker could be covered in the simulator by any of the AI-based agents that we have benchmarked. This is primarily due to the mismatch in (a) behavior models and (b) vehicle dynamics, between the symbolic model and the simulator.

Behavior model mismatch The ego vehicle's non-deterministic behavior, driven by AI models, is not fully captured in the symbolic model. Specifically, the ego's speed fluctuates within ± 1 m/s, unlike the constant cruising speed assumed in the symbolic model. The bounds of cells in the predicate map, as defined in Eq. (2), rely on the relative position of the non-egos with respect to the ego. Consequently, in some cases, the non-egos fail to reach the required regions within these bounds because the ego is traveling either too fast or too slow. In principle, this issue could be addressed by conditioning the behavior of non-ego on the ego's relative motion—using distance traveled with respect to the ego instead of absolute lane positions when translating abstract scenarios to concrete ones. However, ScenarioRunner currently lacks the atomic behaviors or conditions needed to implement such functionality.

Vehicle dynamics mismatch The vehicle dynamics models in the simulator are based on OEM data and are hard-coded into CARLA. The physical constraints for maximum acceleration and braking in the symbolic model are instead averaged (eyeballed) from literature. However, during simulations, we observed significant discrepancies: vehicle models in CARLA exhibited maximum acceleration values as high as 12 m/s^2 and maximum braking values as low as -15 m/s^2. Additionally, these parameter values varied between different vehicle models in CARLA, leading to instances where non-ego vehicles collided with each other during lane-change maneuvers.

5.4.1. AI-based agents

In general, we observe that scenario coverage varies across agents due to different weather conditions and initial positions of non-egos relative to the ego vehicle. As anticipated, decision-making processes of these agents also vary with weather conditions, influencing their abstract scenario coverage. The perception systems of these agents face challenges in adverse weather conditions, occasionally leading to collisions between the ego vehicle and non-egos positioned ahead. Below, we evaluate the performance of each AI-based agent in detail.

Interfuser This agent achieved the highest coverage of abstract scenarios compared to the other agents evaluated in this study, with particularly higher coverage when non-egos start 4 m ahead of the ego. Scenario coverage and property failure results varied with weather conditions. Notably, more crashes were observed during clear noon and wet morning weather conditions. Additionally, when

Table 4
Property and coverage evaluation results for various AI-based autonomous driving agents in the CARLA simulator on highway scenarios with different weather conditions, generated by the VIVAS framework.

AI agents	Weather	Non-ego Initial Position w.r.t. Ego (m)	Property Status			Coverage OK	Coverage OK $\cap \neg\phi_2$	(Set Union) Concrete → Abstract coverage (Max = 144)	
			$\phi_1 \wedge \neg\phi_2$ (goal and crash)	$\neg\phi_1$ (not goal)	$\neg\phi_2$ (crash)			Coverage OK	Coverage OK $\cap \neg\phi_2$
		-4	36	20	44	54	13		
	Wet Morning	**0**	8	19	12	73	4		
		+4	10	18	11	78	6		
		-4	27	19	33	51	5		
	Sunset	**0**	10	18	13	72	3		
		+4	5	19	9	76	4		
Interfuser		**-4**	34	20	44	54	16	**90**	**32**
	Clear Noon	**0**	10	20	15	73	4		
		+4	8	18	10	77	6		
		-4	28	21	39	57	11		
	Night Rain	**0**	11	19	15	76	4		
		+4	7	18	8	75	6		
	Total Concrete Coverage (Max: 144*4*3 = 1728)		**194**	**229**	**253**	**816**	**81**		
		-4	1	93	1	19	0		
	Wet Morning	**0**	1	114	1	6	0		
		+4	0	86	0	10	0		
		-4	1	55	4	32	0		
	Sunset	**0**	0	112	0	9	0		
		+4	0	86	0	14	0		
TCP		**-4**	1	84	2	40	0	**64**	**1**
	Clear Noon	**0**	2	105	2	24	0		
		+4	0	86	0	14	0		
		-4	5	55	8	36	1		
	Night Rain	**0**	0	109	1	16	0		
		+4	0	86	0	15	0		
	Total Concrete Coverage (Max: 144*4*3 = 1728)		**11**	**1020**	**19**	**235**	**1**		
		-4	42	18	51	58	22		
	Wet Morning	**0**	3	19	5	52	0		
		+4	5	23	5	43	0		
		-4	41	18	49	60	23		
	Sunset	**0**	3	18	3	51	0		
TF++		**+4**	7	20	7	44	1		
WP		**-4**	41	19	51	58	21	**86**	**24**
Ensemble	**Clear Noon**	**0**	3	18	3	51	0		
		+4	3	21	4	43	0		
		-4	38	18	44	59	17		
	Night Rain	**0**	3	18	3	46	1		
		+4	3	20	3	45	0		
	Total Concrete Coverage (Max: 144*4*3 = 1728)		**192**	**230**	**228**	**610**	**85**		
		-4	8	55	18	31	2		
	Wet Morning	**0**	2	74	24	19	4		
		+4	1	61	13	9	0		
		-4	6	59	21	30	2		
	Sunset	**0**	7	65	23	18	3		
		+4	7	59	16	9	0		
LAV		**-4**	6	56	18	32	1	**35**	**5**
	Clear Noon	**0**	8	66	20	16	2		
		+4	3	62	13	9	0		
		-4	9	58	20	31	2		
	Night Rain	**0**	7	67	25	18	2		
		+4	2	61	11	9	0		
	Total Concrete Coverage (Max: 144*4*3 = 1728)		**66**	**743**	**222**	**231**	**18**		

non-egos started 4 m behind the ego, there were more crashes, as the non-egos cut into the ego's lane closer than in other scenarios. The agent failed to reach the goal 229 times out of 1,728 concrete scenarios, primarily when non-egos blocked its route. However, in 194 cases, the ego reached its goal despite crashing with a non-ego earlier.

TCP In our tests with the TCP agent, we observed that the ego vehicle braked to a standstill whenever another vehicle approached in an adjacent lane, a behavior deemed too conservative for testing with our verification methodology. As a result, only 19 crashes occurred across 1,728 concrete scenarios, but the ego failed to reach its goal in 1,020 of them. Despite this, 64 abstract scenarios were covered in the simulations. Coverage was higher when non-egos started 4 m behind the ego, but this performance declined in wet morning conditions, where failure to reach the goal was slightly higher than in other weather scenarios. Most crashes occurred during rainy night conditions, suggesting that the perception system is particularly vulnerable to performance degradation in non-ideal weather conditions.

TF++ WP ensemble This agent covered 86 abstract scenarios out of 144 in its execution within CARLA. Higher coverage was observed when non-egos started 4 m behind the ego. At this concretization level, we observed a significantly higher number of crash situations. The perception system appears to perform consistently across all weather conditions in terms of the differences in coverage and property failures, with only marginal improvement under night rain conditions. However, the total number of concrete scenarios covered is notably lower compared to the Interfuser agent (610 vs. 816), despite similar abstract coverage. This suggests that the decision-making of this agent varies significantly across different weather conditions. The total number of property violations is very similar to those observed for the Interfuser agent.

LAV The LAV agent exhibited erratic behavior, making non-deterministic lane changes upon scenario instantiation. As a result, it covered only 35 abstract scenarios during simulations, primarily when non-egos started 4 m behind the ego. Although it experienced fewer crashes than both Interfuser and TF++ WP Ensemble, adverse weather conditions led to higher crash rates compared to the ideal clear noon weather setting. In 743 concrete scenarios, the ego failed to reach its goal position due to the route being obstructed by non-egos. In summary, the LAV agent's performance proved unreliable, primarily due to its non-deterministic lane changes at scenario start, frequent failure to reach its goal, and decision-making being highly sensitive to weather conditions.

Summary Interfuser demonstrates the highest overall performance in terms of total coverage, though it also experiences a significant number of crash scenarios. TF++ WP Ensemble follows closely, achieving good abstract coverage with slightly fewer crash scenarios compared to Interfuser. LAV shows weaker performance, struggling to reach its goal and exhibiting lower overall coverage. TCP performs the poorest, with the ego failing to reach its destination in the majority of scenarios due to its overly conservative behavior, indicating a need for significant improvements to enhance its reliability.

5.4.2. Dynamic coverage predicates

In this work, we aimed to enhance abstract scenario coverage by introducing a more refined notion of coverage criteria and scenario concretization for various weather conditions. Compared to the results in [12], which were executed under "clear noon" weather with the Interfuser AI agent, the coverage results in this paper did not show improvement when non-egos started at a 0 m longitudinal distance from the ego in the concrete scenarios. However, when non-egos were instantiated 4 m ahead or behind the ego, the coverage results significantly improved with the use of dynamic coverage predicates. Additionally, in contrast to [12], the abstract coverage is better in cases where non-egos start ahead of the ego rather than at the same level.

5.4.3. Interesting scenarios

Even though not all abstract scenarios were covered in the concrete simulator, many scenarios involved the AI-based ego vehicle colliding with a non-ego agent in front. Specifically, we identified 32 such "interesting" scenarios with the Interfuser agent and 24 with the TF++ WP Ensemble, where the scenarios met their abstract specifications but resulted in a front collision for the ego vehicle. Fig. 11 shows 5 scenes (in temporal order of 1-5) extracted from one such scenario for Interfuser agent under "Night Rain" weather conditions. This concrete execution corresponds to the abstract scenario specified by the LTL property in Eq. (6) (referencing Eq. (1)).

$$\neg\mathbf{F}(\text{car1\_grid\_pos} = 1 \wedge \text{car2\_grid\_pos} = 5 \wedge \mathbf{X}(\mathbf{F}(\text{car1\_grid\_pos} = 2 \wedge \text{car2\_grid\_pos} = 2))) \tag{6}$$

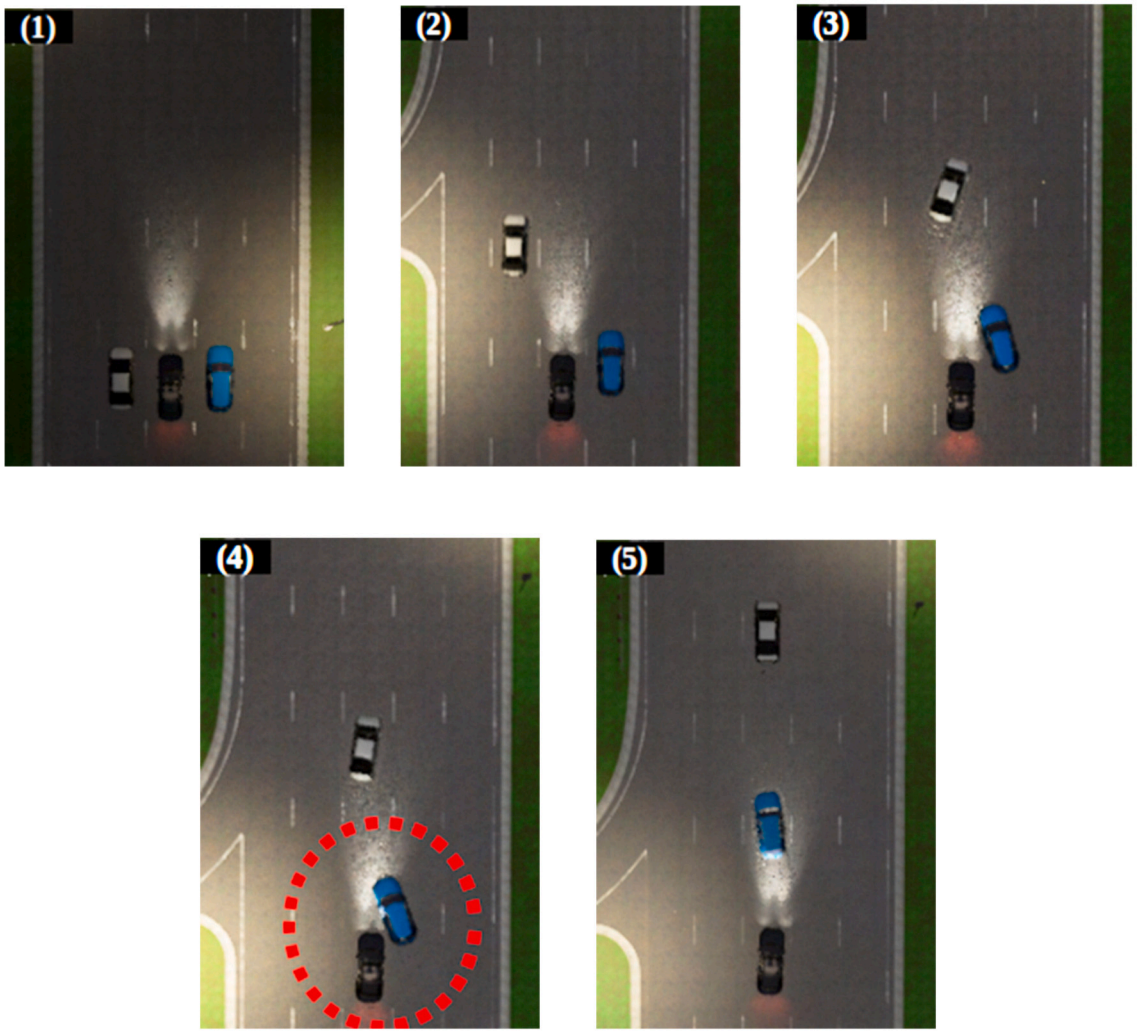


Fig. 11. Scenario defined in Eq. (6), with Coverage OK ∩ Property FAIL. Ego crashing with a non-ego in front (scene 4).

Here, the grid positions correspond to the cell numbers indicated in Fig. 8 for the abstract grid space. Note that the remaining concrete scenarios for other three weather conditions, based on this specific abstract scenario specification, do not result in a crash. We now describe the scenario in Fig. 11.

- All the agents are initialized at 0 m/s, with ego in the center lane, car1 to its left, and car2 to its right, both starting at the same longitudinal level as the ego (scene 1). In this configuration, the predicate "car1_grid_pos $=4 \wedge$ car2_grid_pos $=5$" holds on the abstract grid.
- The non-egos travel faster than the ego, resulting in the configuration (scene 2) where the first predicate "car1_grid_pos $=1 \wedge$ car2_grid_pos $=5$" is satisfied.
- Car1 then changes lane to the right, ahead of ego, while car2 initiates a lane change maneuver to enter ego's lane from the right at a close distance (scene 3).
- Car2 continues its lane change and crashes into the ego (marked by the red ellipse in scene 4).
  Fig. 12 further shows the front view of the Interfuser AI agent at the instant leading up to the crash with car2. Real-time telemetry for scene 1 and 2 indicates that ego's brake $=0$ and throttle $=0.75$ when car2 has already entered its lane. While the perception system correctly detects car1's position, it still computes a safe trajectory to drive forward. In scene 3, the ego applies brakes, but only after the crash has already occurred.
- Car1 and car2 complete their lane change maneuvers, and the ego vehicle is not stationary when the predicate "car1_grid_pos $=2 \wedge$ car2_grid_pos $=2$" is satisfied (scene 5).

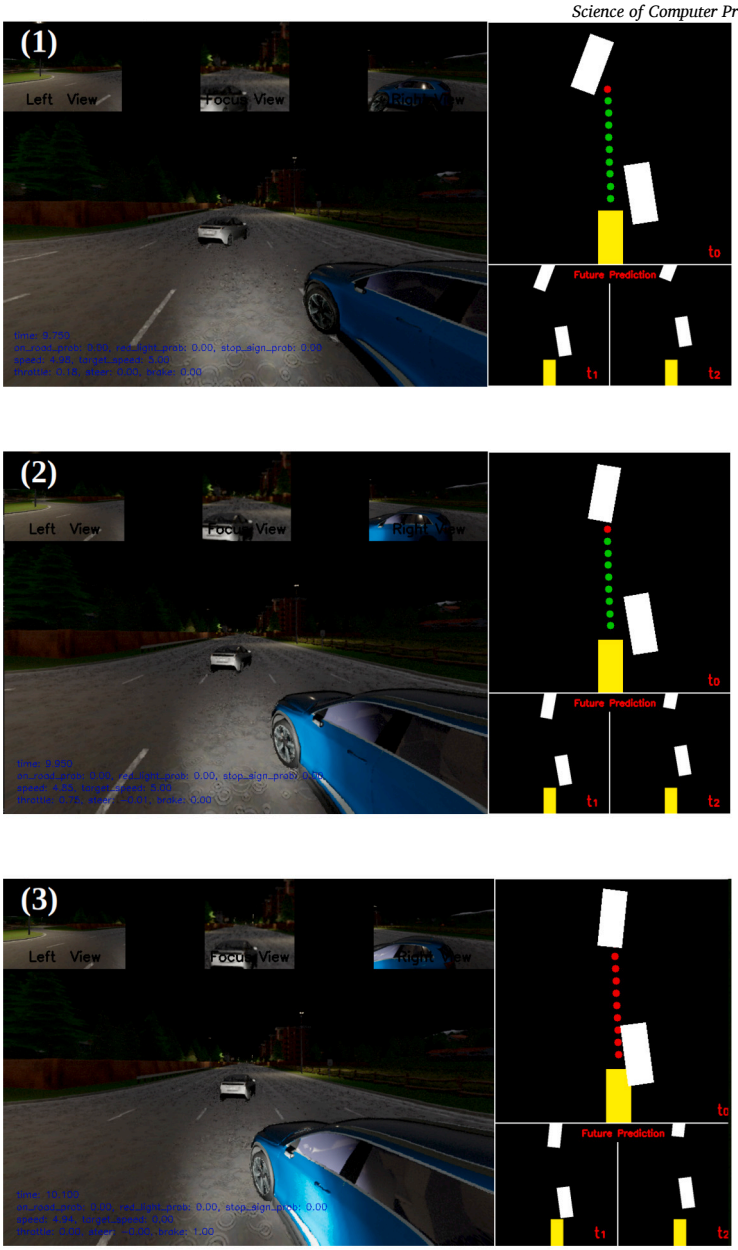


Fig. 12. Sequence leading to crash showing different camera views of Interfuser agent, with the perception component's output; (1, 2): car2 entering ego's lane; (3) ego crashing with car2.

6. Conclusions

This paper demonstrated the application of the VIVAS framework for system-level simulation-based verification of Autonomous Driving Systems (ADS). The framework employs formal methods to generate abstract scenarios and specify formal properties for verifying simulation traces. The ADS instantiation integrates VIVAS with CARLA, a widely-used driving simulator, and its ScenarioRunner tool to create diverse and complex driving scenarios. We detailed the abstract ADS model and the coverage criterion used to generate abstract scenarios focused on hazardous behaviors of vehicles surrounding the ADS. Using the VIVAS framework, we generated and executed a variety of concrete driving scenarios, testing different state-of-the-art AI-based ADS agents from the CARLA Autonomous Driving Challenge.

There are several potential future research directions to enhance the proposed verification approach. These include: developing more efficient techniques for generating abstract scenarios to reduce the number of model checking runs required to achieve a specific coverage level; integrating effective sampling techniques to synthesize various simulation parameters for the same abstract scenario;

extending the abstract model to incorporate uncertainty in ego behavior; providing a more precise representation of continuous-time behavior using timed or hybrid versions of SMV [32,33]; and enhancing the concrete scenario specification with conditional behaviors of non-ego vehicles that react to the choices of the ego vehicle.

CRediT authorship contribution statement

Srajan Goyal: Writing – original draft, Visualization, Validation, Software, Methodology, Investigation, Formal analysis, Conceptualization. **Alberto Griggio:** Writing – review & editing, Supervision, Software, Methodology, Investigation, Formal analysis, Conceptualization. **Stefano Tonetta:** Writing – review & editing, Supervision, Project administration, Methodology, Funding acquisition, Formal analysis, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] T. Dreossi, D.J. Fremont, S. Ghosh, E. Kim, H. Ravanbakhsh, M. Vazquez-Chanlatte, S.A. Seshia, Verifai: a toolkit for the formal design and analysis of artificial intelligence-based systems, in: I. Dillig, S. Tasiran (Eds.), Computer Aided Verification, Springer International Publishing, Cham, 2019, pp. 432–442.
[2] R. Majumdar, A. Mathur, M. Pirron, L. Stegner, D. Zufferey, Paracosm: a test framework for autonomous driving simulations, in: E. Guerra, M. Stoelinga (Eds.), Fundamental Approaches to Software Engineering, Springer International Publishing, Cham, 2021, pp. 172–195.
[3] S. Fratini, P. Fleith, N. Policella, A. Griggio, S. Tonetta, S. Goyal, T.T.H. Le, J. Kimblad, C. Tian, K. Kapellos, C. Tranoris, Q. Wijnands, Verification and validation of autonomous systems with Embedded AI: The VIVAS Approach, in: ASTRA, ESA, 2023, https://az659834.vo.msecnd.net/eventsairwesteuprod/production-atpi-public/070740b67e5b4a32a9be94228c9ac40d.
[4] M. Bozzano, R. Bussola, M. Cristoforetti, S. Goyal, M. Jonáš, K. Kapellos, A. Micheli, D. Soldà, S. Tonetta, C. Tranoris, A. Valentini, Robdt: Ai-enhanced digital twin for space exploration robotic assets, in: The Use of Artificial Intelligence for Space Applications, Springer Nature, Switzerland, 2023, pp. 183–198.
[5] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, V. Koltun, CARLA: an open urban driving simulator, in: S. Levine, V. Vanhoucke, K. Goldberg (Eds.), Proceedings of the 1st Annual Conference on Robot Learning, in: Proceedings of Machine Learning Research, PMLR, vol. 78, 2017, pp. 1–16.
[6] C. Team, CARLA ScenarioRunner, https://carla-scenariorunner.readthedocs.io. (Accessed 30 August 2023).
[7] H. Shao, L. Wang, R. Chen, H. Li, Y. Liu, Safety-enhanced autonomous driving using interpretable sensor fusion transformer, in: K. Liu, D. Kulic, J. Ichnowski (Eds.), Proceedings of the 6th Conference on Robot Learning, in: Proceedings of Machine Learning Research, PMLR, vol. 205, 2023, pp. 726–737.
[8] P. Wu, X. Jia, L. Chen, J. Yan, H. Li, Y. Qiao, Trajectory-guided control prediction for end-to-end autonomous driving: a simple yet strong baseline, in: S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, A. Oh (Eds.), Advances in Neural Information Processing Systems, vol. 35, Curran Associates, Inc., 2022, pp. 6119–6132.
[9] D. Chen, P. Krahenbuhl, Learning from all vehicles, in: 2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), IEEE Computer Society, Los Alamitos, CA, USA, 2022, pp. 17201–17210.
[10] C. Team, CARLA autonomous driving leaderboard, https://leaderboard.carla.org/leaderboard/. (Accessed 30 August 2023).
[11] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, S. Tonetta, The nuxmv symbolic model checker, in: CAV, in: Lecture Notes in Computer Science, vol. 8559, Springer, 2014, pp. 334–342.
[12] S. Goyal, A. Griggio, J. Kimblad, S. Tonetta, Automatic generation of scenarios for system-level simulation-based verification of autonomous driving systems, in: FMAS@iFM, in: EPTCS, vol. 395, 2023, pp. 113–129.
[13] X. Huang, M. Kwiatkowska, S. Wang, M. Wu, Safety verification of deep neural networks, in: R. Majumdar, V. Kunčak (Eds.), Computer Aided Verification, Springer International Publishing, Cham, 2017, pp. 3–29.
[14] G. Katz, C.W. Barrett, D.L. Dill, K. Julian, M.J. Kochenderfer, Reluplex: an efficient SMT solver for verifying deep neural networks, in: CAV (1), in: Lecture Notes in Computer Science, vol. 10426, Springer, 2017, pp. 97–117.
[15] G. Katz, D.A. Huang, D. Ibeling, K. Julian, C. Lazarus, R. Lim, P. Shah, S. Thakoor, H. Wu, A. Zeljic, D.L. Dill, M.J. Kochenderfer, C.W. Barrett, The marabou framework for verification and analysis of deep neural networks, in: CAV (1), in: Lecture Notes in Computer Science, vol. 11561, Springer, 2019, pp. 443–452.
[16] G. Singh, T. Gehr, M. Püschel, M. Vechev, An abstract domain for certifying neural networks, in: Proc. ACM Program. Lang., 3 (POPL), Jan. 2019.
[17] C.S. Păsăreanu, R. Mangal, D. Gopinath, S. Getir Yaman, C. Imrie, R. Calinescu, H. Yu, Closed-loop analysis of vision-based autonomous systems: a case study, in: C. Enea, A. Lal (Eds.), Computer Aided Verification, Springer Nature, Switzerland, Cham, 2023, pp. 289–303.
[18] H. Winner, K. Lemmer, T. Form, J. Mazzega, Pegasus—first steps for the safe introduction of automated driving, in: G. Meyer, S. Beiker (Eds.), Road Vehicle Automation 5, Springer International Publishing, Cham, 2019, pp. 185–195.
[19] C.E. Tuncali, G. Fainekos, H. Ito, J. Kapinski, Sim-atav: simulation-based adversarial testing framework for autonomous vehicles, in: Proceedings of the 21st International Conference on Hybrid Systems: Computation and Control (Part of CPS Week), HSCC '18, Association for Computing Machinery, New York, NY, USA, 2018, pp. 283–284.
[20] D.J. Fremont, E. Kim, T. Dreossi, S. Ghosh, X. Yue, A.L. Sangiovanni-Vincentelli, S.A. Seshia, Scenic: a language for scenario specification and data generation, Mach. Learn. (Feb. 2022), https://doi.org/10.1007/s10994-021-06120-5.
[21] E. Vin, S. Kashiwa, M. Rhea, D.J. Fremont, E. Kim, T. Dreossi, S. Ghosh, X. Yue, A.L. Sangiovanni-Vincentelli, S.A. Seshia, 3d environment modeling for falsification and beyond with scenic 3.0, in: C. Enea, A. Lal (Eds.), Computer Aided Verification, Springer Nature, Switzerland, Cham, 2023, pp. 253–265.
[22] O. foretellix, Open m-sdl, https://releases.asam.net/OpenSCENARIO/2.0-concepts/M-SDL_LRM_OS.pdf. (Accessed 7 August 2023).
[23] A. for Standardization of Automation, M. S. (ASAM), OpenSCENARIO, https://www.asam.net/standards/detail/openscenario/. (Accessed 30 August 2023).
[24] M. Klischat, M. Althoff, Synthesizing traffic scenarios from formal specifications for testing automated vehicles, in: IV, IEEE, 2020, pp. 2065–2072.
[25] C. Robert, J. Guiochet, H. Waeselynck, L.V. Sartori, Taf: a tool for diverse and constrained test case generation, in: 2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS), 2021, pp. 311–321.
[26] L.V. Sartori, H. Waeselynck, J. Guiochet, Pairwise testing revisited for structured data with constraints, in: ICST, IEEE, 2023, pp. 199–209.
[27] B. Jaeger, K. Chitta, A. Geiger, Hidden biases of end-to-end driving models, arXiv:2306.07957, 2023.
[28] A. Cimatti, C. Tian, S. Tonetta, Nurv: a nuxmv extension for runtime verification, in: B. Finkbeiner, L. Mariani (Eds.), Runtime Verification, Springer International Publishing, Cham, 2019, pp. 382–392.
[29] A. Cimatti, C. Tian, S. Tonetta, Assumption-based runtime verification with partial observability and resets, in: RV, in: Lecture Notes in Computer Science, vol. 11757, Springer, 2019, pp. 165–184.

[30] L. de Moura, N. Bjørner, Z3: an efficient smt solver, in: C.R. Ramakrishnan, J. Rehof (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 337–340.

[31] A. Cimatti, A. Griggio, B.J. Schaafsma, R. Sebastiani, The mathsat5 smt solver, in: N. Piterman, S.A. Smolka (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 93–107.

[32] A. Cimatti, A. Griggio, E. Magnago, M. Roveri, S. Tonetta, Extending nuxmv with timed transition systems and timed temporal properties, in: I. Dillig, S. Tasiran (Eds.), Computer Aided Verification, Springer International Publishing, Cham, 2019, pp. 376–386.

[33] A. Cimatti, A. Griggio, S. Mover, S. Tonetta, HyComp: an SMT-based model checker for hybrid systems, in: TACAS, in: Lecture Notes in Computer Science, vol. 9035, Springer, 2015, pp. 52–67.