# Verifying Collision Risk Estimation using Autonomous Driving Scenarios Derived from a Formal Model

Jean-Baptiste Horel[1*], Philippe Ledent[2*], Lina Marsso[3*], Lucie Muller[2*], Christian Laugier[1], Radu Mateescu[2*], Anshul Paigwar[1], Alessandro Renzaglia[4*] and Wendelin Serwe[2*]

[1]Univ. Grenoble Alpes, Inria, 38000 Grenoble, France.
[2]Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP[‡], LIG, 38000 Grenoble, France.
[3]Dept. of Computer Science University of Toronto, Canada.
[4]Univ. Lyon, Inria, INSA Lyon, CITI, 69100 Villeurbanne, France.

*Corresponding author(s). E-mail(s): jean-baptiste.horel@inria.fr; philippe.ledent@inria.fr; lina.marsso@utoronto.ca; lucie.muller@inria.fr; radu.mateescu@inria.fr; alessandro.renzaglia@inria.fr; wendelin.serwe@inria.fr;

**Abstract**

Autonomous driving technology is safety-critical and thus requires thorough validation. In particular, the probabilistic algorithms employed in perception systems of autonomous vehicles (AV) are notoriously hard to validate due to the wide range of possible critical scenarios. Such critical scenarios cannot be easily addressed with current manual validation methods, thus there is a need for an automatic and formal validation technique. To this end, we propose a new approach for perception component verification that, given a high-level and human-interpretable description of a critical situation, generates relevant AV scenarios and uses them for automatic verification. To achieve this goal, we integrate two recently proposed methods for the generation and the verification that are based on formal verification tools. First, we

[‡]Institute of Engineering Univ. Grenoble Alpes

use formal conformance test generation tools to derive, from a verified formal model, sets of scenarios to be run in a simulator. Second, we model check the traces of the simulation runs to validate the probabilistic estimation of collision risks. Using formal methods brings the combined advantages of an increased confidence in the correct representation of the chosen configuration (temporal logic verification), a guarantee of the coverage and relevance of automatically generated scenarios (conformance testing), and an automatic quantitative analysis of the test execution (verification and statistical analysis on traces).

**Keywords:** Behavior tree, CADP, CARLA, CMCDOT, Model-based testing, Model checking, Quantitative analysis

# 1 Introduction

Safety and reliability are key factors for success and acceptance by the broad public of autonomous vehicles (AV). In general, the aim for an AV is to be safer than a human driven vehicle, especially in critical situations, which could lead to costly or even fatal accidents.[1] Thus, AVs, their key components (e.g., perception with a LiDAR and/or with cameras), and algorithms (e.g., for scene interpretation and decision making) are subjected to increasing regulatory and certification demands, requiring particular attention to validation, as witnessed by several international standards under discussion [2, 3].

Because hardware and software components are more and more deeply intertwined and operate on different spatial and temporal scales, the validation of AVs is still an open research topic. Formal methods, in particular mathematical logic, have for long been studied for the verification and certification of algorithms [4], and are gaining popularity in the aerospace and railways [5]. However, the validation of AVs is still challenging, mainly due to the immersion in a less controlled environment [6] and the resulting complexity of key components. These challenges call for probabilistic approaches to perception and prediction [7], with algorithms involving multiple states and complex transitions between them. Hence, the most common validation method is currently still testing, with a particular focus on critical situations, e.g., those emerging from road accident data [8].

Since such critical scenarios are (fortunately) unlikely to happen and costly in real road traffic, a common practice is to reproduce these critical scenarios in an autonomous driving simulator, such as CARLA [9]. Building such scenarios manually is too cumbersome to be scalable, and automatic random generation does not necessarily guarantee relevance of the generated test suite (e.g., coverage of hazards and critical situations), and might present a high level of hard-to-detect redundancies, also strongly limiting coverage [10].

[1] See, e.g., [1] or https://www.tesladeaths.com/ for a list of fatal accidents involving an AV.

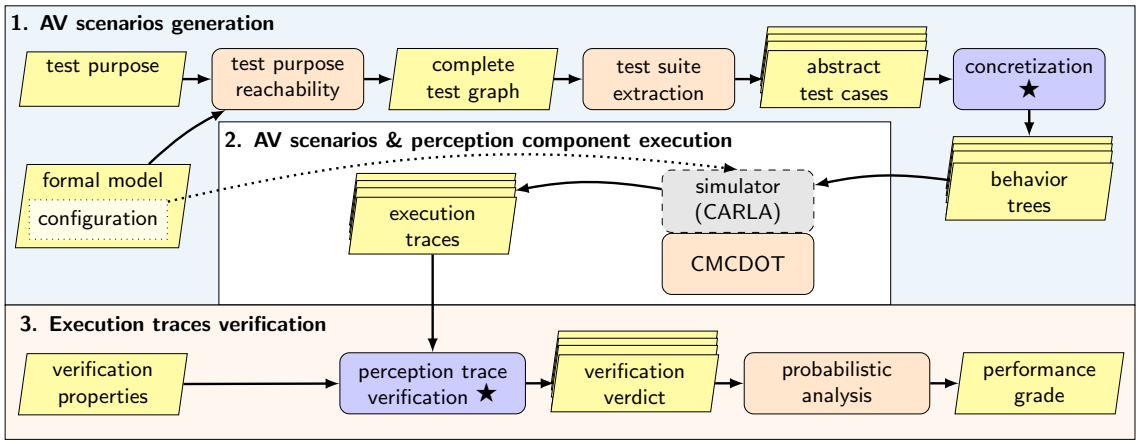



**Fig. 1**: Overview of the proposed approach to verify AV perception components. Our approach generates for a formal model all possible AV scenarios (behavior trees) addressing a specific situation to simulate (test purpose). The generated AV scenarios are then executed on an AV simulator (e.g., CARLA) connected to a perception component (e.g., CMCDOT) to obtain execution traces, on which to perform formal verification and probabilistic reliability analysis. New components are in violet with a ★. Components reused from our previous work are in orange. Other existing components reused are in gray. Artifacts are in yellow rhombus.

An autonomous vehicle has the three classic robotic components that are perception, decision, and action. The role of the perception component is to read the environment using one or more sensors and interpret the data into a model that will be used by the decision component. Generating test scenarios is not enough to evaluate the accuracy of the perception of the environment and the pertinence of the perception model [11]. To evaluate the perception component with a test scenario, one must check whether the perception component outputs have been accurate and reliable to decide about the success of a test execution. Since perception components return probabilistic predictions, their validation must use probabilistic methods. Previously a new technique was proposed, using formal verification to validate systems with probabilistic predictions [12].

In this paper, we combine the technique to formally evaluate a perception component by analysis of execution traces [12] with a technique for the automatic generation of scenarios guaranteed to be relevant [13]. The link between the two techniques is the execution of the scenarios generated by the latter technique using the CARLA simulator to obtain the execution traces analyzed by the former technique.

The complete flow is shown in Figure 1. The starting point is a formal behavioral model of the AV in its environment (see Section 3), specifying a scene map (or ground truth) and all actors (AV, other cars, pedestrians, etc.) with their initial positions and constraints on their trajectories. From this formal model and a test purpose, we derive a complete test graph (see Section 4.1), from which we extract a collection of abstract test cases (with

relevance and coverage guarantees), automatically inducing concrete trajectories for the actors (see Section 4.2). Concretely, using a test purpose specifying a particular situation (e.g., collision, near miss, etc), conformance testing theory [14] ensures that the generated abstract test cases reach the test purpose. These test cases are then transformed into behavior trees (see Section 4.3), which are run in a simulator to obtain execution traces including sensor measurements (see Section 5). This requires the simulator to feature a faithful representation of the dynamic physical environment (e.g., urban landscape, vehicles, pedestrians, etc.) and the perception system under study. In the last step, these execution traces are then formally validated by a combination of model checking (see Section 6.2) and statistical analysis (see Section 6.3) to assess the collision-risk estimation of a perception component. Note that in this paper, we focus on the software part of a perception component taking as input sensor data (e.g., LiDAR, radar, camera) and computing an occupancy grid. Concretely, for each trace, model checking several properties (invariants, safety, liveness) produces quantitative verdicts (with explaining diagnostics), which are used to compute grades, which are in turn aggregated using statistical analyses. The steps illustrated in Figure 1 are all automated, except for the two steps indicated by dotted lines (building a formal model and a CARLA configuration), which have to be carried out together in a consistent way. These two steps are facilitated by means of libraries grouping the generic parts of the formal model and simulator configuration.

Besides connecting the two techniques [12, 13] in order to combine their benefits, we bring the following new contributions:

1. validation of the formal AV and scene model by devising and checking three generic temporal logic properties that should be satisfied by all configurations (see Section 3.3);
2. refined translation of test cases (execution sequences leading to the fulfillment of a test purpose) into behavior trees for the AV simulator (see Section 4.3) with control nodes monitoring the AV scenario execution;
3. analysis of more complex scenarios, with a non-linear behavior of actors (varying speeds and curved trajectories), in order to evaluate the output of the perception system in more realistic situations (see Section 6).

We instantiate the flow in Figure 1 using the CADP (Construction and Analysis of Distributed Processes) toolbox [15] for the analysis of asynchronous systems (in particular the LNT [16] modeling language, the MCL property language [17], and the EVALUATOR [17] and XTL [18] model checkers), the TESTOR tool [19, 20] for the generation of abstract test cases, the CARLA simulator [9] for vehicles, CMCDOT (Conditional Monte Carlo Dense Occupancy Tracker) [21] for collision-risk estimation based on input of perception components, and R-Studio [22] for the statistical analysis. Last, but not least, we illustrate the approach on six representative situations, generating over 1700 complex simulation traces to assess the collision-risk estimation of CMCDOT.

The rest of the paper is organized as follows. Section 2 compares our approach to related work. Each of the Sections 3 to 6 details one step of the approach: Section 3 presents the formal model and its validation by model checking; Section 4 presents the generation of a set of scenarios from the formal model; Section 5 presents the simulation of the generated scenarios with a simulator and the generation of execution traces including the output of a perception component. Section 6 presents the assessment of the collision risk estimation based on the model checking and statistical analysis of the generated execution traces. Section 7 reports about experiments with our approach. Finally, Section 8 gives some concluding remarks, alternative applications of our approach, and directions of future work.

## 2 Related Work

In this section, we compare our work with existing approaches for generating AV scenarios, as well as with approaches that address the verification of perception components and existing methods enabling probabilistic verification. As a general remark, the approach we propose is complementary to most existing techniques, which could benefit from a combination with our techniques.

**AV scenarios generation.** Automatic generation of AV scenarios has been the focus of intensive work [10]. However, most of the existing methods are restricted to one configuration, e.g., highway overtaking or a crossroad. With such restrictions, one cannot specify the complete ODD (Operational Design Domain), which removes the ability to completely cover the ODD and therefore to test the full safety of the AV (also called ego vehicle). In the following, we compare our scenario generation approach to those most closely related (see [13] for a more detailed comparison).

Existing methods use as starting point an abstract but parametric scenario, and generate from it several AV scenarios by instantiating the values of the parameters. Relevant parameter values representing real-world and critical scenarios (i.e., the corner cases of an abstract scenario) are usually searched using stochastic optimization methods as in [23–25]. Another approach [26] uses an evolutionary algorithm to find the abstract scenario parameters (e.g., initial positions or obstacle speeds) corresponding to scenarios that do not lead to a collision or uses other optimization techniques by modeling the generation problem as a quadratic problem with constraints to find such parameters [27]. In contrast, our approach does not rely on an abstract scenario with predefined and parametric behaviors as input, but rather uses high-level test purposes to directly generate several scenarios with different actor behaviors (e.g., trajectories of the ego vehicle and the obstacles), which cover all of the possible outcomes of an abstract scenario. Note that our approach could be used to generate parametric scenarios usable as an input for these other approaches.

Other approaches rely on machine learning (ML) for producing realistic and critical AV scenarios. Starting from abstract scenarios involving urban intersections, neural networks are used in [8] to generate safety-critical AV

scenarios as series of probability distributions, from which one is extracted by sampling and executed on the CARLA simulator. Other works [28] focus on highway overtaking scenarios, from which realistic lane change trajectories are produced using generative adversarial networks previously trained using datasets of highway vehicle trajectories. In comparison, our approach enables to generate both abstract and concrete (i.e., simulator readable) AV scenarios and is versatile enough to account for various scene configurations. Moreover, the ability to extract automatically all test cases matching a test purpose enables to cover all scenarios relevant for the AV behavior in a given situation.

An approach for testing an AEB (Autonomous Emergency Braking) system combining semantic web and ML techniques was proposed in [29]. The ODD of an AEB system serves as basis for defining an ontology, from which AV scenarios are produced using combinatorial testing. The criticality level of scenarios is then raised by applying ML techniques to the outcomes produced by simulating the scenarios. In comparison, our approach enables to play with various abstract scenarios and focuses on behavioral aspects. The complexity of AV scenarios generated by conformance testing is parametric by the configurations considered (number of obstacles and their trajectories) and the test purposes, indicating the critical situations under study.

**Perception component validation.** A significant number of perception components use ML techniques to perform a vision task given sensor data [30]. For instance, ML models are used for object detection and image segmentation using 2D camera sensor data [31–33] or for object detection in 3D LiDAR point clouds [34–36]. Most of the attempts to validate ML-based perception components focus on verifying robustness and reliability. Existing work evaluates the robustness of computer-vision based perception components against adversarial examples, either by generating these examples using testing techniques [37–39], or by verifying their presence in a given range of image data modifications [40, 41]. Another approach similar to adversarial examples is to generate corner cases using testing techniques [42]. Alternatively, other works tested the reliability of a perception component against changes that could occur in the operating environment [43, 44]. In contrast, our approach is focused on behavior: instead of verifying the robustness of a perception component against perturbations or its reliability against visual changes that can occur in deployment, we verify the reliability of the perception component against an environment involving interactions with static and dynamic actors.

**Probabilistic verification.** Perception components (e.g., CMCDOT) compute a probabilistic prediction of the environment given sensor data. In the following, we compare our verification approach with other existing probabilistic verification approaches. Existing work proposes formal verification techniques for Probabilistic Model Checking [45]. While such techniques are ideal for verifying systems where the system behavior and the environment constraints can be accurately formalized [46], it is more challenging to apply it to autonomous systems in dynamic environments. For systems or components in dynamic environment in general, exhaustively checking probabilistic

properties using model-checking does not scale, mostly because the states of such systems and their environment (e.g., other actors) evolve at every time step, increasing the complexity and the cost of verification. As an alternative, existing work uses statistical model checkers [47, 48] to verify if a perception component meets a required accuracy. This verification is done by estimating the component accuracy from a list of its execution traces. In contrast, our approach is using XTL, which is both a model checker and a programming language, and can therefore perform any computation on an execution trace; this enables us to verify not only the accuracy of the perception component outputs, but also other metrics, such as the coherence of its prediction estimation and whether its predictions have an appropriate progression.

# 3 Formal Model of an AV in its Environment

The starting point of our approach is a formal model of an autonomous vehicle (AV)—also called ego vehicle—interacting with its environment comprising static and dynamic obstacles. As shown in Figure 1, the formal model describes a specific configuration for which we verify the perception component. In this section, we first present the use of the formal modeling language LNT [16, 49] for the specification of an IOLTS (Input-Output Labeled Transition System) [50], the semantic model underlying our approach. Then, we give an overview of our formal model of an ego vehicle interacting in diverse scenarios, corresponding to the configuration input of the processing flow shown on Figure 1. Finally, we describe how we validated our formal model by model checking.

## 3.1 Representation of an IOLTS using LNT

To describe formally the behavior of the AV and the obstacles, we use IOLTSs. An IOLTS is a state-transition graph, where all information is contained in the actions labeling the transitions—the states do not provide information, except for the indication of the initial state. Formally, an IOLTS is a four-tuple $\langle Q, A, T, q_0 \rangle$, containing a set of *states* $Q$, a set of *labels* (or *actions*) $A$, a transition relation $T \subseteq Q \times A \times Q$, and an initial state $q_0 \in Q$. The set of labels is partitioned into $A = A_{\mathrm{I}} \cup A_{\mathrm{O}} \cup \{\tau\}$, where $A_{\mathrm{I}}$ is the set of *controllable inputs*, $A_{\mathrm{O}}$ the set of *observable outputs*, and $\tau$ is the internal (unobservable) label, representing the internal behavior that is not visible in the system's interactions with the environment. An example of IOLTS is shown in Figure 6.

To conveniently describe an IOLTS we advocate the LNT language [16, 49], a user-friendly modeling language supported by the CADP verification toolbox [15]. Aiming to make concurrency theory accessible to a wider community, LNT provides the salient features of process calculi in a user-friendly Ada-like syntax. Thus, the vast majority of LNT constructs (modifiable variables, conditionals, loops, functions, pattern matching, etc.) are borrowed from imperative and functional programming languages and thus familiar to most programmers, which only have to learn the constructs dedicated to the concurrency,

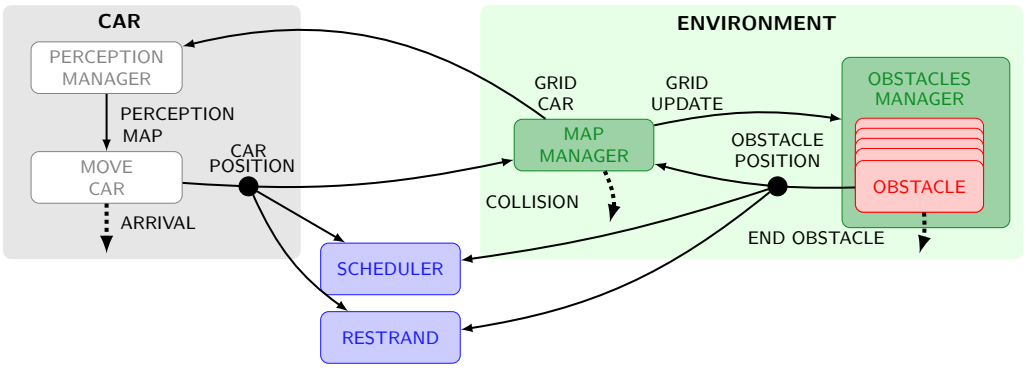



**Fig. 2**: Architecture of the LNT model taken from [53]

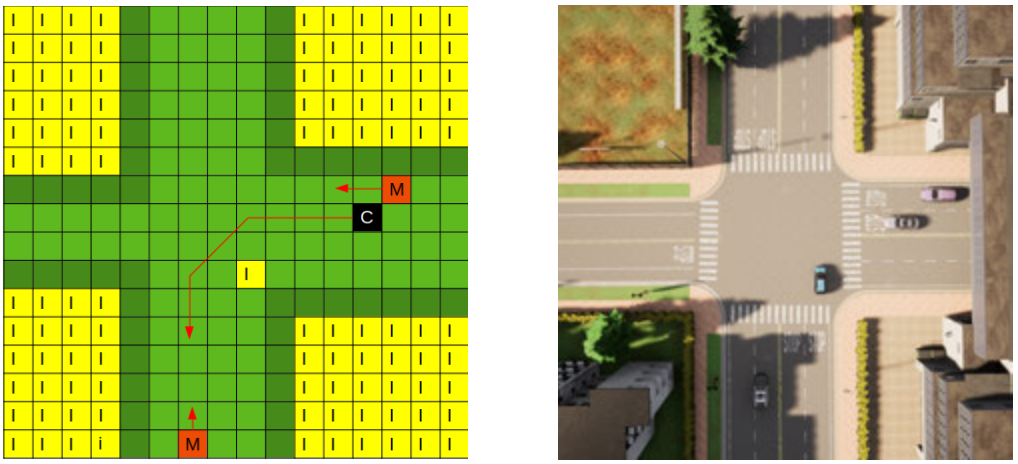
such as parallel composition, non-deterministic choice, and the multiway rendezvous [51, 52]. In a nutshell, an LNT model is decomposed in modules, each containing definitions of (inductive) data types, functions, and processes. LNT comes with the usual set of predefined data types (Booleans, numbers, characters, and strings).

LNT is equipped with a formal semantics, associating to an LNT model the corresponding behavior as an LTS (Labeled Transition System) [49]. Roughly speaking, each rendezvous (involving one, two, or more processes) yields a correspondingly labeled transition in the LTS; data types and functions are used to express values (called offers) exchanged during rendezvous. Processes may have two kinds of parameters. Gate parameters specify the gates on which the process can participate in rendezvous, and value parameters allow to create several instances differing only in the manipulated data, e.g., various vehicles with different initial position, speed, and trajectory. A detailed presentation of LNT constructs is available in [49].

The CADP toolbox provides automatic tools to generate the LTS corresponding to a particular instance of a process. This LTS is available explicitly in the BCG (Binary Coded Graph) format or implicitly for an on-the-fly exploration, such as model checking (see Section 3.3) or test case generation (see Section 4). To transform an LTS in an IOLTS, it is sufficient to indicate (in a separate file), which of the actions are inputs.

## 3.2 Ego Vehicle Interacting with its Environment

We present below the formal model of an ego vehicle moving around in a scene, towards a goal or destination position, and interacting with its environment, i.e., a given set of moving obstacles (pedestrians, cyclists, other cars, etc.) to avoid collisions (if possible). To model the ego vehicle in LNT, we describe its perception (e.g., LiDAR or camera) and action (i.e, controlling the ego vehicle) components and how they interact with the environment. The architecture of the LNT model is shown in Figure 2. The behavior of the ego vehicle is modeled as a sequence of moves by specifying the speed and direction of each move. The ego vehicle arrives at its destination when his sequence of moves




(a) Ground truth grid (b) CARLA simulation

**Fig. 3**: Scene map of a urban road crossing

has been entirely executed. The ego vehicle can also collide with obstacles in the scene and, consequently, will not execute its remaining moves.

**Obstacles.** We distinguish between static obstacles (such as buildings or vegetation) and dynamic ones (such as other vehicles, pedestrians, and cyclists). Both kinds of obstacles can be transparent, in the sense that the perception component of the ego vehicle can perceive the status of the scene behind the obstacle—this is the case for instance for a pedestrian, or a ball, but not for a building. Dynamic obstacles have predefined trajectories, specified as sequences of moves. To provide room for various interaction scenarios with the ego vehicle, an obstacle may perform random moves and, at each step, randomly choose to move or not. The behavior of dynamic obstacles is similar to the ego vehicle, with three extensions: (1) an obstacle may not move (whereas the ego vehicle always advances); (2) an obstacle may perform a random move with an arbitrary choice of direction, while avoiding collisions; and (3) an obstacle may restart the execution of its sequence of moves (cyclic behaviors). An obstacle can also leave the map if its trajectory leads out of the scene border and will also stop instead of colliding with another obstacle if the two obstacles would move to the same cell of the map.

**Geographical map.** We represent the scene map (ground truth) as a grid, i.e., a discrete two-dimensional array composed of cells. Figure 3 shows an example of a grid, together with a screenshot of a corresponding simulation in CARLA. Each cell of the grid can be either free, occupied by the ego vehicle, or occupied by an obstacle. The grid is updated upon each move of an actor (ego vehicle or dynamic obstacle) to reflect its position change in the scene. The grid on Figure 3(a) represents an X-shaped crossroad delimited by static obstacles corresponding to buildings (cells marked "I"). The dark green cells (surrounding the buildings) are also static obstacles and correspond to sideways. There are four actors in the scene: the ego vehicle (cell marked "C") and three other cars, among which two are dynamic (cells marked "M") and

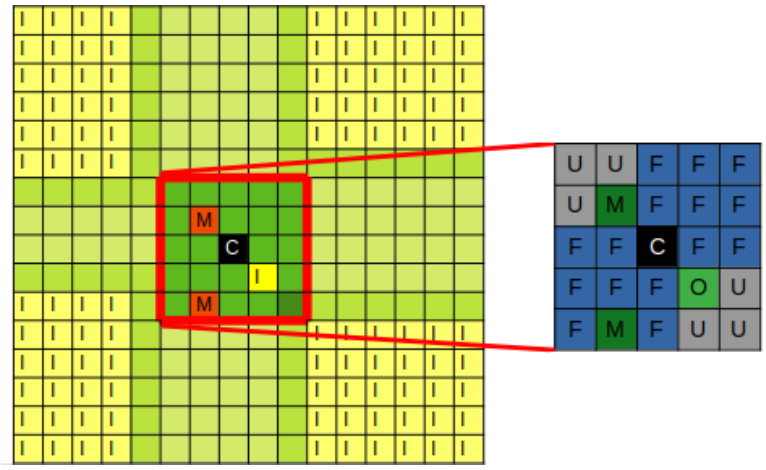



**Fig. 4**: Perception grid corresponding to the configuration in Figure 3

one is static (cell marked "I"). The ego vehicle advances on the left lane, in parallel with another car moving at the same pace, then takes a left turn at the intersection, while the other car continues straight ahead. Then, the ego vehicle encounters the last obstacle that moves up on the wrong way, leading to a collision.

Although the representation of the scene map as a grid is an abstraction, it has been shown to be appropriate for the purpose of generating simulation scenarios [13]. Using a grid also facilitates the trade-off between the precision of the representation and the computing resources needed for processing the IOLTS model, by varying the grid resolution (the finer the resolution, the larger the IOLTS size).

**Perception grid (simplified 3D model).** Based on the scene map, the LNT model computes the correct perception grid shown in Figure 4, corresponding to the ideal expected output (occupancy grid) of the perception algorithm generated from the ego vehicle's perception component input. The perception grid is modeled as a square of cells (zone of the scene map) with a given size, centered around the ego vehicle and indicating whether its cells are free, occupied, or unknown (e.g., hidden by an opaque obstacle or out of the map if the ego vehicle is near the scene border). The perception grid detects the obstacle moves by inspecting if a cell that was free on the previously computed grid became occupied on the current grid. Even though the perception grid in our formal model is 2D, it includes a notion of transparency to represent 3D information. This simplified representation is sufficient for controlling actor positions, i.e., ensure that actor trajectories are in the perceivable area of the perception component in the AV scenario.

**Global system composition.** Actors are modeled as concurrent LNT processes interacting by multiway rendezvous, as illustrated on Figure 2. The process `MAIN` defines the overall behavior as the parallel composition of the ego vehicle and environment processes (`CAR` and `ENVIRONMENT`) and two additional processes used to optimize the size of the model (`SCHEDULER` and `RESTRAND`), as

```
process MAIN [GRID_UPDATE:G, OBS_POS:O,
              PERCEPTION_MAP:L, CAR_POS:C, GRID_CAR:CG,
              COLLISION, ARRIVAL, END_OBSTACLE, TICK:none]
is
  -- parallel composition enabling model optimization
  par CAR_POS, OBS_POS in
    -- parallel composition of ego vehicle and environment
    par CAR_POS, GRID_CAR in
      CAR [PERCEPTION_MAP, CAR_POS, GRID_CAR, ARRIVAL]
    ||
      ENVIRONMENT [GRID_UPDATE, OBS_POS, GRID_CAR,
                   CAR_POS, COLLISION, END_OBSTACLE]
    end par
  ||
    SCHEDULER [OBS_POS, CAR_POS, TICK]
  ||
    RESTRAND [OBS_POS, CAR_POS]
  end par
end process
```

**Fig. 5**: Global system composition (`MAIN` process)

shown in the LNT fragment given in Figure 5 (the gate parameters of a process are specified between square brackets and the gate names were abbreviated for conciseness).

**Map management.** The global status and the updates of the scene map are handled by a map manager process (a sub-process of `ENVIRONMENT`), in charge of maintaining a consistent ground truth. The ego vehicle and each obstacle process has a local copy of the map, which is used to perform moves respecting the laws of physics. Each actor, after moving, sends its new position to the map manager, which updates the actor position on the scene map and then broadcasts the map to all actors to inform them about the change. The execution of the scenario terminates when either the ego vehicle arrives at its destination (action `ARRIVAL`) or a collision occurs between the ego vehicle and an obstacle (action `COLLISION`). Consequently, an execution sequence of the model cannot contain both a collision and a successful arrival.

**Model realism and optimization.** LNT has an interleaving semantics, which basically amounts to represent the parallel execution of two actions $a$ and $b$ as the choice between their two possible orderings $a.b$ or $b.a$. Therefore, the combined behavior of concurrent actors in the LNT model is represented in the corresponding IOLTS by the interleaving of all their execution sequences. Although semantically equivalent, some of these interleaved executions are unrealistic w.r.t. the physical scene (e.g., an obstacle suddenly stops, the other actors perform several moves, then the obstacle suddenly restarts and performs several moves while the other actors do not move). To obtain realistic executions of concurrent actors for the simulator, we introduce a notion of discrete

time enabling the specification of concurrent moves (i.e., observed simultaneously) by adding a special `TICK` action. Between two `TICK`s in the LNT model, an actor can only move once, and therefore in the simulator all moves between two `TICK`s can be executed in parallel, leading to realistic simulations. The execution order of actor moves between two consecutive `TICK`s is imposed by an auxiliary process `SCHEDULER`, which removes redundant interleaved sequences of actor moves and also reduces the size of the IOLTS.

Another optimization of the model stems from the observation that an obstacle far from the ego vehicle is not relevant for testing the perception component, and hence the model should focus only on relevant scenarios, in which obstacles are perceivable by the perception component. This is achieved by introducing an auxiliary process `RESTRAND` in charge of constraining the random moves of the obstacles and disturb the ego vehicle neighborhood. Based on the ego vehicle trajectory and position, the `RESTRAND` process constrains the next obstacle moves by restricting their random moves such that obstacles are brought closer to the perceivable area of ego vehicle (a parameter of the scenario). An obstacle close enough to the ego vehicle is able to move randomly without restrictions. This optimization also enables to reduce the size of the IOLTS.

**Scenario configuration.** The LNT model has parameters to specify a configuration, which consists of the scene map, the initial positions of the actors (ego vehicle and obstacles), and constraints on their trajectories. Note that the concrete trajectories of the actors will be automatically induced by the generated AV scenarios. To facilitate the modeling of various configurations, the LNT model of a configuration is divided in two parts: a common part defining types, functions, and processes shared between all configurations, and a specific part defining the constants specific to a configuration. More precisely, the specific part must contain the definitions of: the scene map (with the static obstacle positions and initial position of the dynamic obstacles), the number of obstacles, the size of the perception grid, the behavior of all actors, and the size of the area around the ego vehicle where an obstacle is allowed to move randomly.

Figure 6 shows an excerpt of the IOLTS generated from the LNT model of the configuration in Figure 3. Transitions colored in blue represent inputs to control the simulation (the positions of the ego vehicle and obstacles), all the other transitions being outputs to observe the progress of the simulation (scene map, perception grid, arrival, and collision). For clarity, the contents of the scene map and perception grid is not shown on the `GRID_UPDATE`, `GRID_CAR` and `PERCEPTION_MAP` transition labels. The IOLTS fragment contains the transition sequences going out of the initial state (numbered 0) until the first `TICK`.

The complete LNT model of a simple configuration is published [53], together with an alternative model including a more detailed description of the car components (in particular, a decision component in charge of computing the car trajectory).

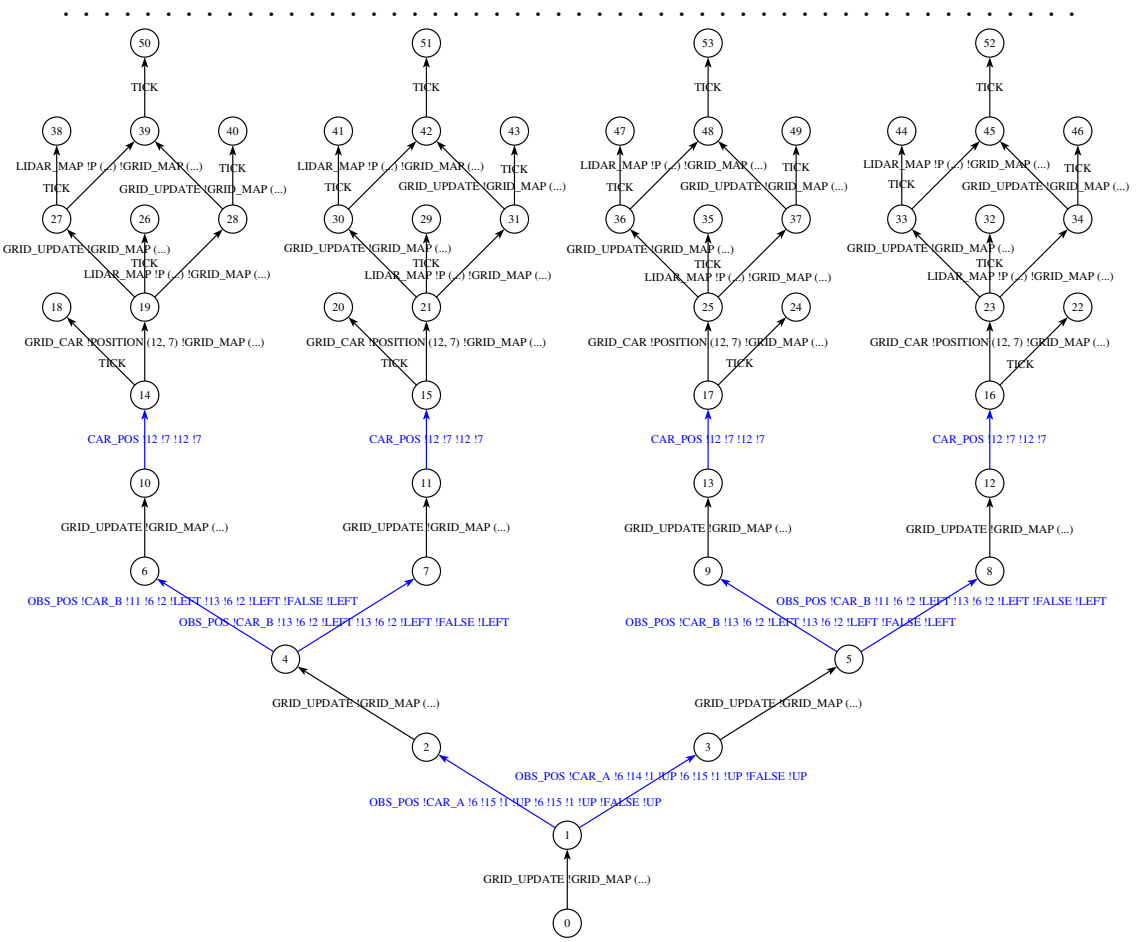



**Fig. 6**: Graphical illustration of an IOLTS for an excerpt of the configuration in Figure 3. Blue edges correspond to the moves of the ego vehicle and the obstacles (the obstacles have the choice to move or to stay in their position, the car always moves: thus there are four different possible behaviors in the upper part of the figure). Black edges correspond to the position updates in the geographical map (`GRID_UPDATE`), and the car map (`GRID_CAR`) from which the perception grid (`LIDAR_MAP`) is computed.

## 3.3 Validation by Model Checking

To assess the correctness of the LNT model, we validated it using the CADP tools, first by interactive step-by-step simulation (back and forth navigation along the transition sequences of the corresponding IOLTS) and then by model checking [54] (verification of temporal logic properties on the IOLTS). We formulated several properties to ensure that, in a given configuration, the LNT model correctly represents the expected behaviors of the actors in the scene map defined by the configuration. Each property was specified in MCL (Model Checking Language) [17], the data-handling, action-based, branching-time temporal logic supported by CADP.

The considered properties belong to the classes of liveness, fairness, and safety properties, traditionally used for concurrent systems [55]. A liveness

property expresses that something good eventually happens. A fairness property expresses that, under certain conditions, something should always happen. A safety property expresses that something bad never happens. We describe below the three most important properties and their formulation in MCL.

*Property 1:* "*The model is free of deadlocks (states without successors).*"
This liveness property can be expressed in MCL as follows:

```
[ true* ] < true > true
```

This modal formula specifies that any state reachable from the initial state via a sequence of zero or more transitions (necessity modality "`[ ... ]`") must have at least one outgoing transition (possibility modality "`< ... >`").

*Property 2:* "*Inevitably, the system should reach a state where either the car arrived, or a collision occurred between the car and an obstacle, or all obstacles have finished their list of moves.*"

Given that the model is free of deadlocks (as ensured by Property 1), this fairness property can be expressed in MCL by forbidding the presence of infinite transition sequences not containing one of the three terminal actions `ARRIVAL`, `COLLISION`, and `END_OBSTACLE`:

```
not < not (ARRIVAL or END_OBSTACLE or { COLLISION ... }) >@
```

The infinite looping modality "`< ... >@`" specifies the existence of (unfair) infinite sequences containing only transitions labeled by actions other than the three terminal ones. The action predicate "`{ COLLISION ... }`" characterizes the actions (or transition labels) consisting of the gate name `COLLISION` followed by zero or more data fields.

*Property 3:* "*An obstacle does not cause a collision with the car.*"
This safety property can be expressed in MCL by a necessity modality:

```
[ true* .
  { CAR_POS ?x:Nat ?y:Nat } .
  (not { CAR_POS ... })* .
  { OBS_POS ?c:String ?x1:Nat ?y1:Nat ?x2:Nat ?y2:Nat } .
  (not { CAR_POS ... })* .
  { CAR_POS !x !y } .
  (not { CAR_POS ... })* .
  { OBS_POS !c ?x1p:Nat ?y1p:Nat ?x2p:Nat ?y2p:Nat where
    ((x1 <> x1p) or (x2 <> x2p) or
     (y1 <> y1p) or (y2 <> y2p)) and
    (x >= x1p) and (x <= x2p) and
    (y >= y1p) and (y <= y2p) } .
  (not { CAR_POS ... })* .
  { COLLISION ... }
] false
```

The necessity modality forbids the presence of undesirable transition sequences specified by the enclosed regular formula, namely sequences containing, in order: (1) an update of the car position ("`CAR_POS ?x ?y`"), where `x` and `y`

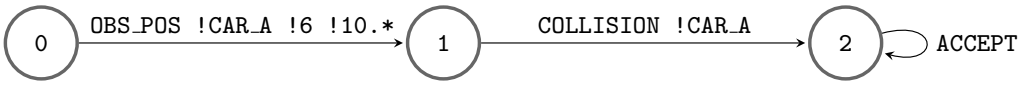



**Fig. 7**: Example of test purpose specifying that, after `CAR_A` has moved to a given position (first transition "`OBS_POS ...`"), a collision of the ego vehicle with obstacle `CAR_A` should occur (second transition "`COLLISION !CAR_A`")

are the car coordinates; (2) an obstacle move ("`OBS_POS ?c ?x1 ?y1 ?x2 ?y2`"), where `c` is the name of the obstacle and `x1`, `y1`, `x2`, and `y2` are its rectangle coordinates; (3) a non modification of the car position ("`CAR_POS !x !y`"); (4) a change in the position of the same obstacle ("`OBS_POS ... where ...`"); and (5) a collision occurring before any car move. Note the extraction of data values contained in transition labels and their reuse in subsequent action predicates.

Further properties have been verified for a GALS (Globally Asynchronous, Locally Synchronous) variant of the model [56].

# 4 Test Suite Extraction

To extract a test suite, i.e., a set of AV test cases focusing on particular hazards or situations, from the formal LNT model, we use a test generation tool, which computes the reachability of a test purpose to generate scenarios interacting with the AV simulator as shown in Figure 1. Concretely, we use ioco [50] conformance testing with a test purpose (TP) [14] as implemented in the TESTOR tool [19] to compute in a first step a complete test graph (CTG). The CTG generated in this way is guaranteed to contain all possible abstract test cases (in the formal model) leading to the goal of the chosen TP [14]. By construction, each test case (TC) of the test suite can be used to control a system under test (SUT) so as to drive it towards the goal of the TP, while checking conformance of the SUT with the formal model. In this work, we are not interested in the conformance checks, but rather reuse the extraction algorithms of conformance testing to generate simulation sequences. It is worth noting that in our particular setting, each TC is guaranteed to be a sequence, because all outputs of the model correspond to observations of the ground truth and are thus deterministic in the sense that they either depend only on the sequence of inputs (i.e., the moves of the actors), or their order is enforced by the scheduler (e.g., a `COLLISION` transition occurs always before an update of the ground truth `GRID_UPDATE`).

## 4.1 Computation of a Complete Test Graph

Technically, a TP is an IOLTS that is composed in parallel with the formal model, so as to mark the goal states of the model to be reached with a self-loop labeled with the particular action `ACCEPT`. For convenience, the parallel composition can complete parts left unspecified in the TP, which can thus focus on specifying only a partial pattern of the expected sequence to be extracted.

As an example, Figure 7 shows a TP to extract scenarios ending with a collision of the ego vehicle with another car named `CAR_A` after that `CAR_A` has been at the precise position `(6, 10)`, hence forcing `CAR_A` to move up to this position. This TP has only three states and three transitions, requesting to reach (after an arbitrary number of transitions) a collision with a car, represented by a transition labeled with "`COLLISION !CAR_A`", but not before `CAR_A` reaches the position `(6, 10)`.

A CTG is an IOLTS containing *all* transition sequences of the model that lead to the `ACCEPT` states. Roughly speaking, the CTG corresponds to the part of the model such that from all states of the CTG an `ACCEPT` state is reachable. The more precise the TP, the smaller the CTG, because fewer sequences of the model match the TP. Figure 8 shows an excerpt of the CTG generated from the IOLTS in Figure 6 and the TP in Figure 7. Some actions were hidden for readability, keeping only the interactions of the ego vehicle with `CAR_A`.

In our analysis flow (see Figure 1), in the CTG and all artifacts derived from it (TCs, behavior trees, etc.), the only interesting actions are the controllable inputs or observable outputs of the SUT. Thus, all other actions that were kept in the model for expressing and verifying temporal logic properties (e.g., the broadcast of the ground truth map, perception grid, etc.) can be hidden, yielding after minimization a smaller IOLTS and improving the performance of the CTG computation and also of the subsequent steps.

## 4.2 Test Suite Extraction

In general, a CTG may contain several sequences of the model that match the TP: as seen before, the more generic the TP, the more sequences of the model will be included in the CTG. In our setting, this manifests as states of the CTG with several outgoing transitions labeled by inputs of the SUT (i.e., in terms of conformance test theory, the CTG is not controllable).

To obtain a set of sequences that contain at least once every transition of the CTG, we apply an approach to compute a set of TCs covering the CTG [20]. In a first step, we compute the set of *candidate* transitions, i.e., transitions such that there exists another transition in the CTG with the same source state but a different (input) label. In a second step, we extract for each candidate transition a TC (i.e., a sequence) containing that transition. To reduce the size of the test suite (set of TCs generated), we start by extracting sequences for the candidate transitions in decreasing order of the distance from the initial state to their source state (since longer sequences are more likely to cover other candidate transitions), and extract only sequences for candidate transitions not yet covered by the test suite.

In the TC shown in red in Figure 8, the transition labeled with `:PASS:` indicates that the TP has been reached. Any observation of an output not foreseen in the TC would indicate that the SUT is not conform to the model, pointing to an error in the model or the SUT.

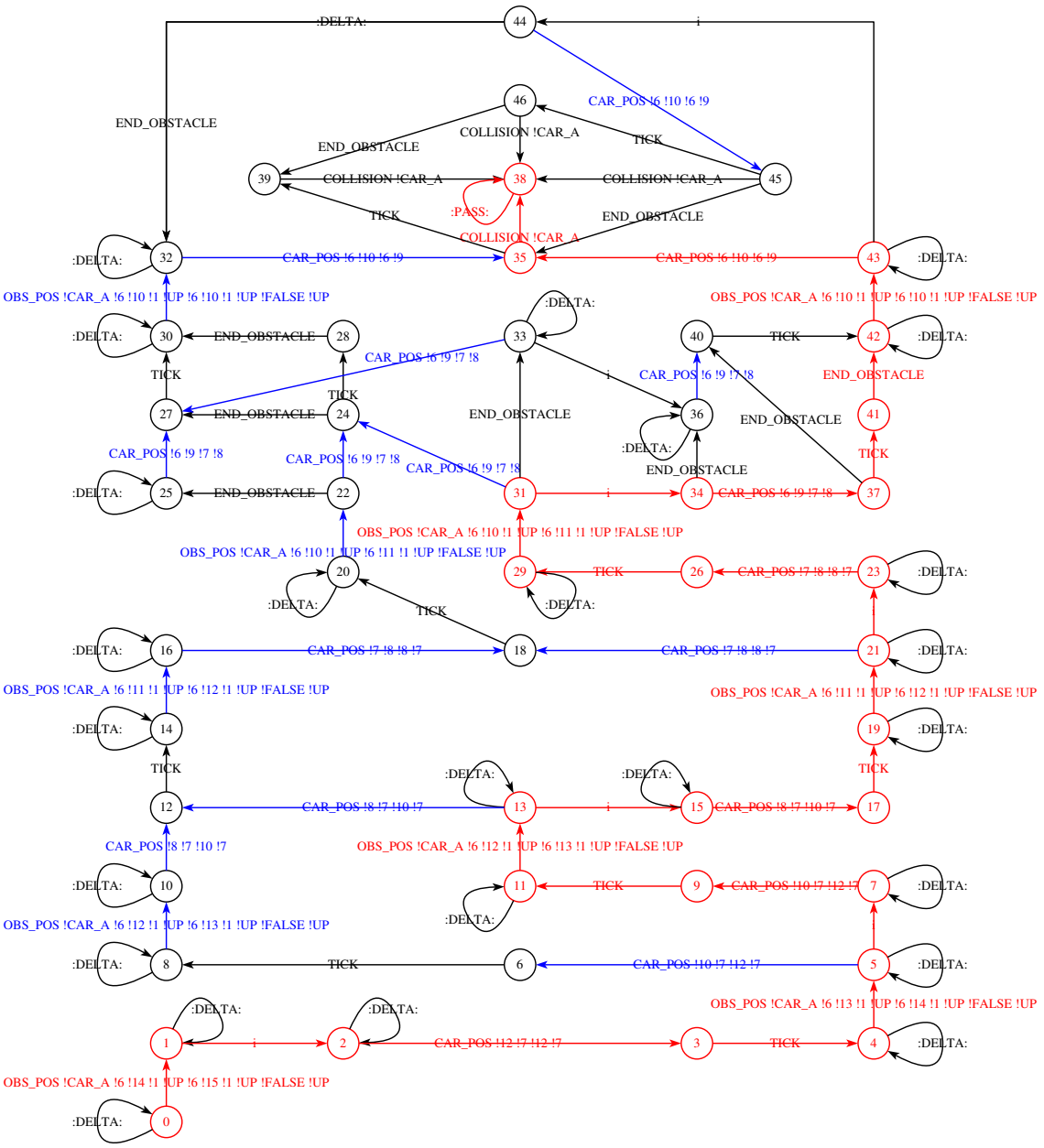



**Fig. 8**: CTG (excerpt) generated from the IOLTS in Figure 6 and the TP in Figure 7. The transitions in red correspond to one TC contained in the CTG. As for Figure 6, transitions of the CTG (but not the TC) corresponding to inputs of the SUT are colored in blue.

## 4.3 Translation into Behavior Trees

The TCs generated from the LNT model and a TP are transition sequences denoting executions of the various actors present in the configuration. To use these TCs for synthesizing simulation scenarios, they must first be translated into a format suitable for the targeted AV simulator (in our case, CARLA), which corresponds to the concretization step shown on Figure 1.

**Setup to simulate an AV scenario.** To set up the simulation, one starts by configuring CARLA according to the configuration defined in the LNT model. This includes the definition of the map (called "town" in CARLA) and instantiating all actors with their type (e.g., cars, cyclists, and pedestrians) and initial position. Then, one must describe the behavior (i.e., the moves) of the actors during the simulation scenario. This behavior description is obtained by translating automatically a TC into a behavior tree (BT), which is the CARLA representation closest to our transition sequences. To execute simulation scenarios defined by BTs, we adapted the scenario_runner feature of CARLA to fit our method.

**Translation of a TC to a behavior tree.** The translation from a TC to a BT consists of three steps:

1. Conversion of the TC from the transition sequence format to a textual format, which is processed by a shell script to retrieve the information necessary for its execution (e.g., actor names, initial positions, and moves).
2. Conversion of the retrieved information into a JSON description, which provides a human-readable representation, as well as a storage format for TCs. JSON files are also easy to handle in Python, the programming language used by scenario_runner.
3. Instantiating the BT just before the execution of the TC in CARLA. The scenario_runner extracts from the JSON file the list of actors, their initial position, and their behavior (the sequence of their moves).

**Behavior tree extended with monitoring nodes.** Figure 9 shows the BT obtained by translating the TC shown in Figure 8. The BT is composed of several types of nodes:

- The blue rounded rectangles are behavior nodes, each one denoting the execution of a specific task (e.g., move an actor during a tick or detect a collision between two actors). A behavior node succeeds if it successfully performs its task, otherwise it fails.
- The yellow parallelograms are parallel nodes, which denote the simultaneous execution of their child nodes (e.g., several actors moving simultaneously). A parallel node can be configured to succeed when either one of its child nodes succeeds, or all of them succeed (join).
- The green arrows are sequence nodes, which denote the sequential execution of their child nodes in order (from left to right on Figure 9). A sequential node waits for the currently executed child to succeed before executing the next one, and it succeeds when its last child node succeeds.

As specified in the LNT model (see Section 3.2), the actors' moves occurring between two ticks in the TC sequence must be performed simultaneously in the simulator. In the BT on Figure 9, this is described by the subtree rooted at the sequential node *Moves Sequence*. The move actions between two consecutive ticks are grouped together as children of a parallel node (the nodes *Step k* on Figure 9), and thus the actors of the TC perform their actions simultaneously

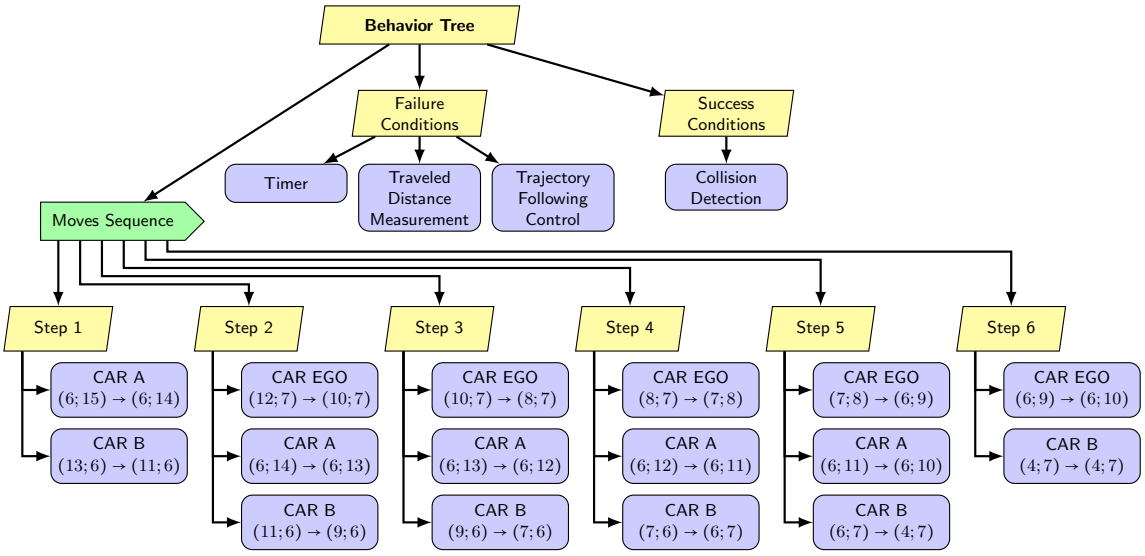



**Fig. 9**: Behavior tree corresponding to the TC outlined in Figure 8

during the current tick period. Then, the parallel nodes are placed, in order, as children of the sequential node. The execution starts by the first tick (*Step 1*), and each node *Step k* is executed when the previous node *Step k* − 1 succeeded, i.e., all the move behaviors of the previous tick period succeeded.

We enriched the BT with two extra nodes named *Failure Conditions* and *Success Conditions*, in charge of monitoring the proper execution of the simulation scenario and the fulfillment of the TP, respectively.

*Failure Conditions* is a parallel node, whose children are behavior nodes monitoring that the actors or the scenario itself respect a property and failing if the property is broken. *Failure Conditions* fails when any of its children fails (i.e., a property is broken during the execution of the scenario) and it never succeeds because its children are designed not to succeed. The child nodes of *Failure Conditions* are the following:

- *Timer* specifies that the execution time of a scenario must be smaller than a specified time, which is typically chosen long enough for this property to be permissive. This behavior node will mostly fail when the scenario faces an unexpected deadlock situation, and prevents an automatic execution of many (wrong) scenarios of being indefinitely stuck.
- *Traveled Distance Measurement* specifies that an actor must not travel a distance greater than an expected distance, which is calculated by integrating the moves from the TC. If an actor exceeds the expected distance above a threshold, we consider that it did not respect its behavior as described by the TC.
- *Trajectory Following Control* specifies that an actor must not deviate from its given trajectory, which is calculated by putting the actor's moves together to form a list of segments, then by computing the distance of the actor to the closest segment and comparing it to a threshold. This node ensures that the actors do not deviate from the behavior described by the TC.

*Success Conditions* is a parallel node, whose children are behavior nodes waiting for a property to be satisfied, e.g., the TP to be fulfilled. *Success Conditions* succeeds when all its children succeed (i.e., when all the expected properties the scenario are satisfied) and it never fails because its children are designed not to fail. For a TC ending with a collision between the ego vehicle and an obstacle, we introduced the behavior node *Collision Detection*, which succeeds when a collision occurs between two actors.

The root of the BT is the node *Behavior Tree*, which is configured to succeed only when *Success Conditions* succeeds and to fail when any of its child nodes fails, in particular *Failure Conditions*.

The experiments we carried out required to automatically execute many scenarios, each of them several times. The monitoring nodes *Failure Conditions* and *Success Conditions* were very useful in automating this process, by avoiding a human monitoring of each execution. An execution of a scenario not respecting the conditions, which means not respecting the TC, is ignored and restarted.

Note that our approach to generate these AV scenarios is versatile enough to account for various scene configurations. To do so, one must adapt the inputs, namely the TP and the configuration (as shown in Figure 1). For instance, one can update the number and the positions of the static obstacles in the geographical map, or even select a completely different map.

# 5 Simulation and Collision Risk Estimation

The purpose of being able to generate scenarios was to have meaningful simulator executions to test perception algorithms and determine the reliability of their outputs for making decisions that will drive AVs. In this work, we demonstrate our approach with CMCDOT [21] as shown in Figure 1.

Given sensor data, CMCDOT computes a representation of the perceived environment in the form of a probabilistic occupancy grid (see Figure 12) [57, 58]. Occupancy grids—whether probabilistic or not—provide an efficient representation of the environment of an AV, and are therefore widely used in AV perception research. Different configurations of input data have been studied so far: LiDAR [21][59], cameras [60][61], Radar [62], and fusion of LiDAR and camera [63]. The CMCDOT perception system, which we use to illustrate our approach, uses clouds of LiDAR points as main sensor inputs.

As shown in Figure 10, CMCDOT converts the sensor data into probabilistic estimations representing the static and dynamic occupied areas, and also the free and unknown spaces of the perceived environment. More importantly for our work, CMCDOT also computes for each cell of the occupancy grid a collision risk estimation. Given the trajectory of the ego vehicle and the obstacles linearly projected in the future, CMCDOT estimates the position of every static and dynamic cell of the occupancy grid [64]. These estimations are periodically (small-time periods) computed, until a potential collision is detected.

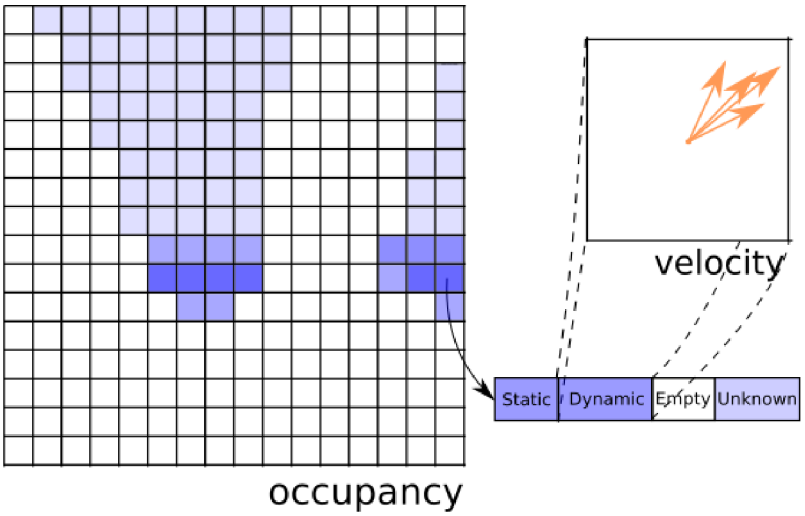



**Fig. 10**: CMCDOT represents the environment as a grid, to whose cells are associated static, dynamic, empty and unknown coefficients. Weighted particles, which sample the velocity space, are then associated to the dynamic part (taken from [21]).

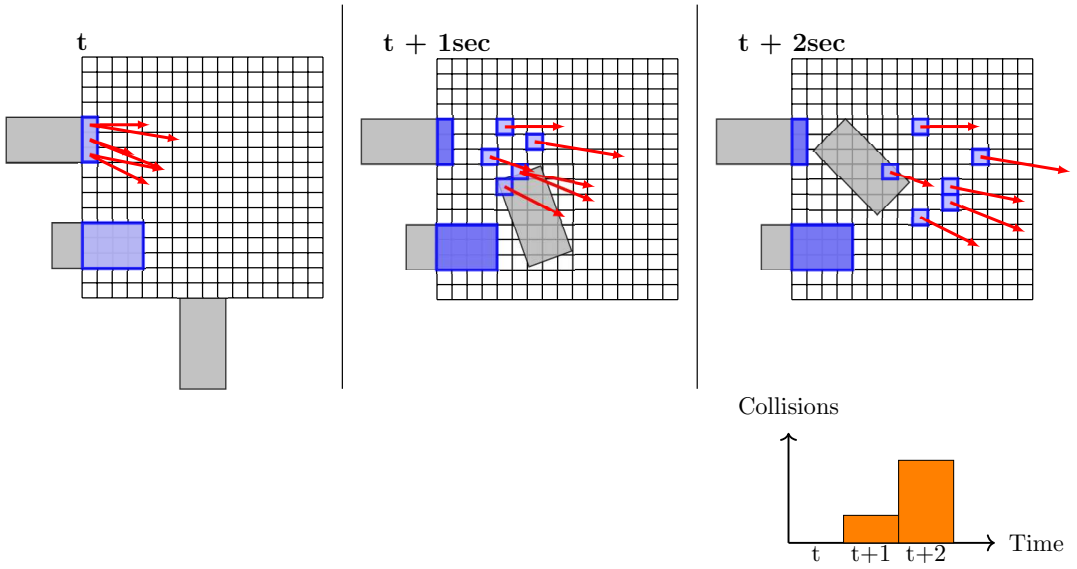
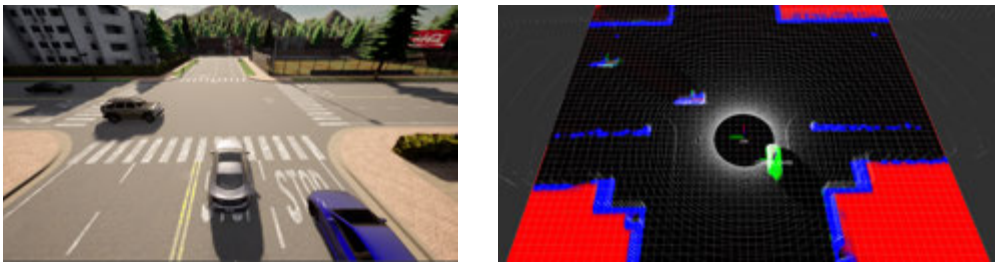



**Fig. 11**: Collision risk estimation of a specific cell. Both the future cell and the ego-vehicle positions are predicted according to their estimated velocity. The risk of every cell is used to integrate over time the total collision risk.

When a potential collision is detected, a time to collision (TTC) [65] is associated to the cell from which the colliding actor came from (see Figure 11). The probabilistic estimation for different TTCs is then mapped to each cell to get a collision risk profile.



(a) View from ego vehicle (b) CMCDOT occupancy grid

**Fig. 12**: Simulated scenario in CARLA (a) and output of CMCDOT (b). Colors in the CMCDOT output represent different states of each grid cell: occupied static (blue), occupied dynamic (green), empty (black), and unknown area (red).

## 5.1 Simulation for Perception

The system we use to execute the scenarios relies on two frameworks: the AV simulator CARLA [9] and the communication framework of ROS (Robot Operating System) [2]. We use CARLA to create a complex virtual environment. CARLA provides several complex urban environments with complex road topology, buildings, objects related to road infrastructure, and decorative objects. CARLA also provides many models of fully controllable actors (e.g., cars, pedestrians, motorbikes), which can be equipped with typical AV sensors (e.g., LiDAR, cameras). For our experiments, we used the urban environment *Town03*, which is composed of several intersections and road segments well-suited to the configurations of our formal model. ROS is a communication framework dedicated to robotics that is highly used to design autonomous systems composed of processes running in parallel and communicating by messages via a network. CMCDOT receives the data of the LiDARs and the other sensors of the vehicle through ROS and also publishes its output, namely the occupancy grids, through ROS. The connection between ROS and CARLA takes place via the CARLA-ROS bridge [3], which monitors the sensors instantiated in CARLA and publishes their measurements on the ROS network.

The collision risk for a specific actor (i.e., a car the ego vehicle is supposed to collide with) can be computed by monitoring the risk grid output of CMCDOT and, knowing the exact position, orientation, and bounding boxes of that actor, extracting the risk from the grid cells corresponding to the actor's footprint.

## 5.2 CMCDOT Execution Trace

The CMCDOT execution traces can be logged while executing the scenarios in the simulator. Table 1 shows a fragment of an execution trace corresponding to the running example scenario in Figure 12. A trace is a sequence of events,

[2] https://www.ros.org/
[3] https://github.com/carla-simulator/ros-bridge

| Timestamp (Real) | Risk 1 sec (Probability) | Risk 2 sec (Probability) | Risk 3 sec (Probability) | Collision status (Boolean) | Segment (Int) |
|---|---|---|---|---|---|
| *... the ego car moves toward the car obstacle ...* | | | | | |
| 28.2946 | 0 | 0 | 0 | `false` | 3 |
| 27.7946 | 0 | 0.11 | 0.11 | `false` | 3 |
| *... the ego car slows down ...* | | | | | |
| 28.3946 | 0 | 0 | 0 | `false` | 4 |
| 28.4946 | 0 | 0 | 0.102 | `false` | 4 |
| *... collision ...* | | | | | |
| 30.9946 | 0.996 | 0.996 | 0.996 | `false` | 6 |
| 31.0946 | 1.0 | 1.0 | 1.0 | `false` | 6 |
| 31.1946 | 1.0 | 1.0 | 1.0 | `true` | 6 |

**Table 1**: Fragment of a simulation trace from the scenario in Figure 12

produced at a 10Hz rate. Each event contains the current simulation timestamp, CMCDOT's current estimations of the probability of a collision of the ego vehicle with the other vehicle (in the next 1, 2, and 3 seconds), the current collision status reported by the simulator, and the segment of the trace. Until the final collision (as requested by the TP of the running example) eventually occurs, the collision status is `false` and the collision risks are estimations of the likelihood of the collision status to change to `true` within the next 1, 2, or 3 seconds following the current timestamp. The segment field of the trace is an extension of our work [12], necessary to handle non linear actor behavior as discussed in the next section.

# 6 Formal Validation of the Collision Risk Estimation

Our objective is the evaluation of a perception component (here, CMCDOT's collision risk estimation) in order to determine its reliability for decision-making, taking advantage of the automated generation of complex scenarios and their execution in a simulator, as shown in Figure 1. The evaluation focuses particularly on the evolution and meaningfulness of the collision *risk*, expressed as a triple of the likelihoods that the ego vehicle will collide with a cell within the next 1, 2, and 3 seconds, respectively.

Three properties are checked on the execution traces to ensure *estimation coherence*, *prediction safety*, and *appropriate progression*. Previously [12], we tested these properties on simple scenarios, where two vehicles collide at a right angle in an X-junction. Our goal was to find near miss configurations to check CMCDOT's proximity accuracy. In this article, taking advantage of the scenario generation approach (see Section 4.3 and [13]), we can evaluate perception component outputs in more complex scenarios mirroring more realistic situations with more dynamic vehicles. In these more complex scenarios, vehicles may accelerate, slow down, and even turn. This means that a collision predicted with high risk at one timestamp with given speeds and directions may never occur because of the actors' non-linear behavior. For this reason, the events of the trace are grouped into segments, corresponding to time windows

| Coherent combinations of $R(e)$ | Interpretation |
|---|---|
| (0, 0, 0) | Not within 3 |
| (0, 0, .5) | Not within 2 |
| (0, 0, 1) | After 2, before 3 |
| (0, .5, 1) | After 1, before 3 |
| (0, 1, 1) | After 1, before 2 |
| (.5, 1, 1) | Within 2 |
| (1, 1, 1) | Within 1 |
| (0, .5, .5) | Not before 1 |
| (.5, .5, 1) | Within 3 |
| (.5, .5, .5) | Undecided |

The interpretation indicates when the collision is to be expected. Incoherent combinations are not presented. The order and grouping of combinations corresponds to the penalty in the quantitative analysis (see Section 6.3).

**Table 2**: Coherent combinations of $R(e)$ and their interpretation

during which the actors' behavior is stable enough to evaluate the properties. These segments change when the speed or direction of vehicles vary too much. We observed that these changes occur along the `TICK` actions that we provide in the scenarios (see Section 3.2) : Between two `TICK`s, actors behave linearly, and segments may span over several `TICK`s. In the following, we explain the process to verify the three properties on the logged execution traces of the perception component.

## 6.1 Risk Interpretation and Estimation Coherence

The collision risk is estimated as the probability of observing a collision event, which in itself has no meaning for making decisions. A low probability expresses the idea that a collision is not expected, while a high probability expresses that a collision will almost certainly be observed. However, these meanings can hardly be derived from the probabilistic values alone. That is why it is important to classify the probabilities by interpretation. We consider a 10% threshold around 0 and 1 to take into account the slightly fluctuating behavior of CMCDOT's risks. Thereby, probabilities close to 0 are classified as 0 and interpreted as expressing that no collision will occur in the respective 1, 2, or 3 seconds. Likewise, probabilities close to 1 are classified as 1 and interpreted as announcing a collision. The safe interpretation of values in between is that they predict neither a collision nor the absence thereof. The status is unknown at that time with the expectation that they will converge to 0 or 1. Until then, probabilities between 0.1 and 0.9 are classified as 0.5 with the *unknown* interpretation.

From these individual interpretations, it is interesting to consider the three risks together as a triple $R(e) = \big(R_1(e), R_2(e), R_3(e)\big)$, and derive a meaning for the entire triple. $R_i(e)$ with $i \in \{1, 2, 3\}$ denotes the collision risk within $i$ seconds expressed on trace event $e$. For instance, $R(e) = (0, .5, 1)$ means that a collision is expected within 3 seconds but not within the first one.

Intuitively, there must be some coherence within each triple. For instance, $R(e) = (1, .5, 0)$ is not coherent, because it is impossible to observe a collision during the next second, and none during the next three seconds. Table 2 displays the ten coherent configurations out of the possible 27 ($= 3^3$). The *estimation coherence* property consists therefore in checking that within each triple $\big(R_1(e), R_2(e), R_3(e)\big)$, the risks follow the order relation $R_1(e) \leq R_2(e) \leq R_3(e)$. This property is an *invariant* because it must always hold and can be checked separately on each event of the trace. It is also a sanity property because if it fails, the risk is meaningless for other properties as well as for deciding how to drive the AV.

## 6.2 Perception Trace Verification

We assess the reliability of CMCDOT's predictions by checking several safety and liveness properties with the XTL (eXecutable Temporal Language) model checker [18] of CADP. XTL can perform operations at a lower level than classical temporal logic, in particular the parsing of traces in the BCG (Binary-Coded Graph) format [66], an explicit representation of an LTS. Additionally, because XTL is also a programming language, it can also be used to define non-standard temporal operators and to perform any computation on an LTS, also involving the data values carried by the events. Because XTL requires a BCG graph as an input, the traces are converted into BCG format having each event of the trace as a separate transition label.

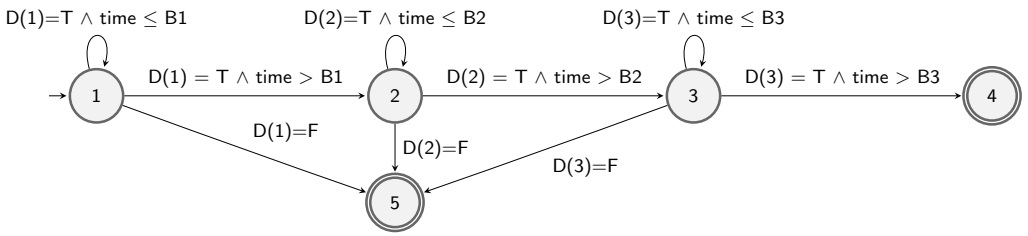
We present below the three considered properties in natural language, together with the corresponding commented XTL code for the first one, and the observer automaton illustrated and explained for the two others.

**Coherence (sanity).** This property can be verified using an invariant. We parse each trace and evaluate the coherence property on every event by applying the following XTL formula. Since the risk is given as a probability, in XTL we use the notation `P_col_`$i$ with $i \in \{1, 2, 3\}$.

```
def PROP_INV_COHERENCE (event:edge) : boolean =
   let (P_col_1:real, P_col_2:real,
        P_col_3:real) = Get_Risk (event) in
      (P_col_1 <= P_col_2) and
      (P_col_2 <= P_col_3) and
      (Round_num (P_col_1) <= Round_num (P_col_2)) and
      (Round_num (P_col_2) <= Round_num (P_col_3))
   end_let
end_def
```

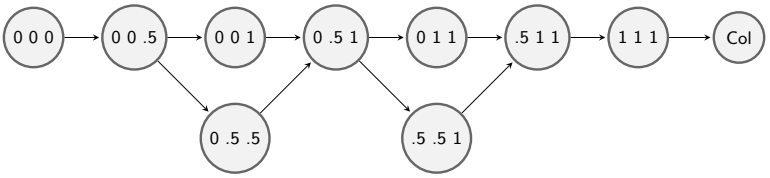
The function `Get_Risk()` extracts the values of the three risks from an event, and the function `Round_num()` converts the risks into their classifications so as to check that the order of the classifications is coherent as well.

**Prediction safety.** As stated in Section 3.3, safety is the notion that nothing bad will happen. A *bad* prediction foresees that a collision will not occur but it does. Similarly, it is also arguably bad to predict that a collision will occur when




This automaton combines the behavior of a deciding and a timed automaton. The initial state 1 indicates that the property is being checked for second 1. `B1` is a bound identifying the timestamp after 1 second. Until `B1` is reached, the function `D(1)` decides whether the prediction currently holds. If it fails, the transition to the rejecting state 5 is taken. If the prediction was safe until the end, it moves onto state 2. If the automaton finishes in the accepting state 4, the property passes and the prediction is declared safe.

**Fig. 13**: Observer automaton checking prediction safety




The arrows indicate a gain of information. It contains the first nine of the combinations given in Table 2. The last one, (.5, .5, .5), is omitted because it does not provide a gain of information.

**Fig. 14**: Expected progression of predictions as a collision approaches

it actually does not. This safety property detects bad predictions, and if there are none, CMCDOT's behavior over the traces is considered safe. Concretely, the property checks for each event `e`, the accuracy of the prediction `P_col` with respect to the collision field "`col (e)`". To do so, it reads the prediction given on event `e` and parses the trace for the next 3 seconds verifying the presence or absence of the collision event according to the 3 predictions. This temporal property is specified using an observer automaton [67] illustrated in Figure 13. The complexity of the scenarios requires that we extend our approach to take into account the major changes in the context due to very dynamic actors. This is the reason for the use of segments shown in Table 1. All violating events are logged along with the prediction they made and the timestamp of the actual collision.

**Proper prediction progression.** Predictions do not only need to be coherent and accurate (safe), the evolution of successive predictions must be coherent as well. To be reliable, as the collision is approaching, the whole risk needs to progress from (0, 0, 0) to (1, 1, 1). Figure 14 indicates the intended evolution of the risk using the coherent combinations of Table 2. This property checks the

evolution of the prediction `P_col` throughout the whole trace. In our previous work [12], we expected the trace to only advance forward through this expected evolution and rejected any other transition backwards or too fast forwards. Here, we once again adapt the approach to consider the progress over the segments reported in the trace, because it is normal for the prediction to suddenly drop when a vehicle turns (to avoid a collision). This property is a *response* property according to the classification of temporal properties [55] because if nothing happens, the evolution is expected to remain always at (0, 0, 0), yet as soon as it departs from that initial prediction, the progression must follow the one that is expected. The progressing part of the property is a mix of safety and liveness elements. Expecting a precise order is a *safety* element as any other order is considered a *bad* evolution. Each step of the progression is a *liveness* element itself because the *good* evolution is expected to *eventually* make progress (at least for all scenarios ending with a collision). This last property is checked using another observer automaton that either expects (0, 0, 0) the whole time or to observe the progress expressed in Figure 14. The diagnostic counterexample (or proof) that is produced for events that violate the property contains the prediction read on the event and the previous one.

The next section will give more details on how the properties were checked and how the set of violating events with violation proofs are used to grade a trace according to each property.

## 6.3 Probabilistic analysis

Since we seek to evaluate CMCDOT's risk estimation, when checking the properties on the traces, instead of just giving a binary `true` or `false` verdict, we generate for each trace the set of events that violate each property. Each of these events is accompanied with a proof of why it violates the property. These proofs of violation and the number of violating events are then used to give a quantitative grade to each trace according to each property.

**Querying the trace.** All XTL functions necessary to analyze the events of the trace can be automatically generated, knowing the position of the data as a field of an event, its type, and its name. For example, the three generated XTL functions below enable to get, set, and test (`Get_P_Col1`, `set_where_P_Col1_is`, and `has_P_Col1`) the field named `P_col_1` of type `real` contained in an event `e`.

```
(* Read P_col_1 *)
def Get_P_Col1 (e:edge) : real =
    if e -> [ G _ _ _ ?p1:real ... ]
       then p1 else -(1.0)
    end_if
end_def

(* Test if P_col_1 has value p1 *)
def _ has_P_Col1 _ (e:edge, p1:real) : boolean =
    e -> [ G _ _ _ !p1 ... ]
end_def
```

```
(* Generate the set of edges where P_col_1 has value p1 *)
def set_where_P_Col1_is (p1:real) : edgeset =
    { e:edge where e has_P_Col1 p1 }
end_def
```

Note that in XTL, an underscore ("_") corresponds to a field wildcard, matching (and disregarding) any value. Likewise, an ellipsis ("...") indicates that all following fields are irrelevant in the pattern matching of the current event, and that counting them is not necessary. Additionally, note that the function `has_P_Col1()` is defined with underscores around its name, which allows it to be used with infix notation in the function `set_where_P_Col1_is()`.

**Property grades.** After verifying the properties on the traces, it is interesting to quantify how bad the violations were. We use the violation proofs to give each trace a penalty with respect to each property. These penalties take into account the fact that the closer a mistake is to an actual collision, the more critical the mistake. We give grades as real numbers from 0 to 1. When a trace never violates a property it receives the maximal grade of 1, otherwise a penalty is deducted for every violating event. Thus, if a major part of the trace is bad, the grade is close to 0, but if only a few events are locally bad, the grade is close to 1 without reaching it. Grades and penalties are first applied to the individual events, and then the global grade of the trace is the average of grades of all its events.

For *prediction coherence*, the property checked that the risks expressed on $R(e) = (R_1(e), R_2(e), R_3(e))$ are in order. When they are not, e.g., $R_2(e) > R_3(e)$, the penalty is the difference of probabilities $R_2(e) - R_3(e)$.

For *prediction safety*, the severity of the error increases depending on which of the three predictions was wrong. A wrong prediction of $R_1(e)$ in less than 1 second of the collision is worse than a wrong prediction of $R_3(e)$ at a little less than 3 seconds before the collision. We give as individual grade for each event, $1 - \frac{1}{k}$ with $k$ in $\{1, 2, 3\}$ designating which of the $R_k(e)$ was wrong. The penalties are therefore, *full, half, third*.

For *proper prediction progress*, all combinations are numbered (from 0 to 6) in the order (from left to right) of the top group of Figure 14. The combinations on the bottom group get the same number as the combination above it, i.e., 2 for (0, .5, .5) and 4 for (.5, .5, 1). When an event violates this property, it attempted a progression through a transition that was not in the figure. The penalty is of $1 - \|\frac{k}{6}\|$ where $k$ is the number of combinations backwards or too far forwards the evolution has jumped. Recall that we allow for such jumps when the segment changes.

# 7 Experiments & Results

In contrast to existing work (listed in Section 2), our AV generation approach does not rely on an abstract scenario with predefined parametric behaviors,

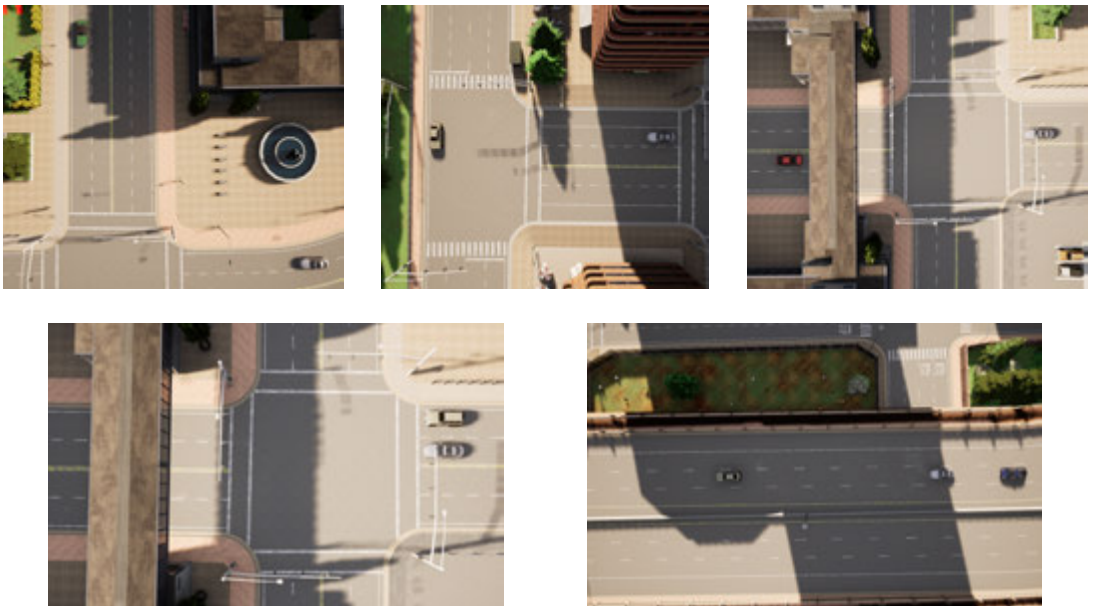


**Fig. 15**: Scenes of the configuration in CARLA, from top left to bottom right: X-crossroad, static-collision, cross-no-collision, car-obst-par, and highway

but instead has the ability to generate such abstract scenarios with the guarantee to be relevant with respect to a given high-level test purpose. Thus, our approach is complementary to existing work and not comparable. Consequently, in this section we cannot provide a quantitative comparison with existing work, but report instead the results of experiments demonstrating the effectiveness of our approach on six realistic and autonomous-driving relevant configurations for verifying the estimation results of a perception component (here, CMCDOT). We measure effectiveness in terms of the ability to generate relevant autonomous vehicle scenarios and use them to assess the probabilistic estimation of collision risks of CMCDOT, and performance, i.e., time and memory usage.

All experiments were carried out using a computer running Linux with an NVidia RTX 3090 GPU, two Intel Xeon Silver 4110 CPUs, and 46 GB of RAM. GPU load during the simulation averaged around 35%. The traces were generated by executing the following processes: CARLA, the scenario_runner (execution of the behavior tree), CMCDOT, and the process generating traces from the risk grid. The computation of the test suite and the formal analysis using CADP, TESTOR, and R-Studio require less resources, and are feasible on a standard laptop with less than 16 GB of RAM.

**Configurations.** To illustrate the genericity of our approach, we considered six different configurations, shown on Figures 3b and 15, obtained by varying the map (town area of a given height and width), the number and kind of obstacles (size, position, static or dynamic), and the behavior of actors (directions and speed). The chosen configurations are inspired from real accidents that occurred on the French roads. An analysis of the accident tickets from French police was carried out in [68], resulting in a large set of abstract

**Fig. 16**: Execution of the behavior tree in Figure 9, corresponding to the configuration in Figure 3, starting from top-left to bottom-right

| model | #obstacles | #lines | #states | #transitions |
|---|---|---|---|---|
| X-crossroad | 1 | 96 | 252 | 371 |
| static-collision | 1 | 98 | 18 | 23 |
| car-obst-par | 1 | 96 | 250 | 377 |
| cross-no-collision | 1 | 96 | 258 | 392 |
| highway | 2 | 97 | 3559 | 6909 |
| running-example | 2 | 106 | 964 | 1638 |

**Table 3**: Information about the formal models of the considered configurations

scenarios that cover the various configurations found in the original accident database. The configurations, the test purposes and scenarios we defined for our experiments (e.g., rear collision, right-angled collision, left turn collision at a crossroad) correspond to the configurations in [68] that led to more than 50% of the yearly deadly accidents involving cars (*véhicule léger* in french). To create a new scenario, one can easily change one or more of the aforementioned parameters (see Section 4.3).

Each of the six configurations contains a different map and obstacle behaviors. More precisely, we consider the following configurations, CARLA screenshots of which are shown in Figure 15 and Figure 3b: (1) an X-shaped crossroad with and without collision; (2) collision with a static obstacle; (3) drive toward a point paralleled with another car; (4) crossing another vehicle

| model | TP | CTG | | test suite | | | | behavior |
|---|---|---|---|---|---|---|---|---|
| | | states | trans. | nb | states | trans. | $\sigma$ | JSON lines |
| X-crossroad | collision | 30 | 52 | 3 | 44 | 46 | 1 | 213 |
| static-collision | collision | 30 | 14 | 4 | 55 | 58 | 1 | 252 |
| car-obst-par | arrival | 601 | 1127 | 64 | 2396 | 2485 | 4 | 11944 |
| cross-no-collision | arrival | 522 | 991 | 69 | 2548 | 2624 | 2 | 12711 |
| highway | collision | 1368 | 2836 | 589 | 104294 | 120008 | 88 | 87722 |
| highway | arrival | 3230 | 6639 | 342 | 79275 | 92556 | 100 | 83790 |
| running-example | collision | 564 | 1100 | 249 | 7665 | 8030 | 7 | 45456 |

**Table 4**: Statistics about the scenario extraction

going in the opposite direction without leading to a collision; (5) the ego vehicle and two other cars on a highway; and (6) the running example used in the previous sections. A collection of videos illustrating examples of scenarios for each of these six configurations is available online[4]. The statistics of the corresponding formal models are shown in Table 3, which includes the number of (moving) obstacles, the number of LNT lines specific to the configuration and the number of states and transitions of the corresponding IOLTS, minimized for strong bisimulation. Note that the generic part of the model (common to all configurations) consists of 1130 lines of LNT.

The first four configurations in Table 3 are basic ones, with one obstacle and no turn: this is reflected by the size of the LTS. The configuration where the ego vehicle collides with a static obstacle has the smallest LTS: this is because only the ego vehicle moves, and does so in a predefined and deterministic way. The last two configurations have more obstacles and more moves carried out by the actors, and hence yield larger LTSs.

**Test purposes.** For each considered configuration, we specified one test purpose (TP), either ensuring that the AV scenarios lead to a collision of the ego vehicle with an obstacle or that the ego vehicle arrives at the destination (in which case no collision occurs). For the highway configuration, we specified both TPs. Information about the corresponding complete test graph (CTG), test suite, and generated AV scenarios is shown in Table 4, which gives the size of the CTG, number of test cases (corresponding also to the number of scenarios run in the simulator), sum of states and transitions of the test suite (i.e., the total size), the standard deviation ($\sigma$) for the number of test-case states (showing that all the test cases are of comparable size), and the total number of JSON lines of the generated behavior trees.

The size of the CTG of a configuration is correlated with the size of the LTS. Thus, the first four (basic) configurations of Table 3 also have a smaller CTG than the two other configurations. We can also observe that the size of the CTG for the TP "arrival" is significantly larger, because there are more possibilities to walk through a scenario without a collision than with a collision. Indeed, for a collision to happen it is necessary that the trajectory of the ego vehicle crosses the trajectory of another obstacle at a precise moment, and because obstacles are free to choose not to move, there is a lot of leeway to

[4] https://doi.org/10.5281/zenodo.7225366

| scenario name | speeds (km/h) | traces | lines | avg. lines/trace |
|---|---|---|---|---|
| X-crossroad collision | 20, 25, 30 | 90 | 8,814 | 97 |
| car-obst-par | 20, 25, 30, 35, 40 | 183 | 27,495 | 150 |
| cross-no-collision | 30, 35, 40 | 205 | 26,436 | 128 |
| highway arrival | 30, 35, 40 | 342 | 55,129 | 161 |
| highway collision | 30, 35, 40 | 306 | 37,638 | 123 |
| running-example | 20, 25, 30 | 666 | 86,997 | 130 |
| static-collision | 30, 35, 40 | 120 | 8,740 | 72 |
| *total* | *30, 35, 40* | *1,703* | *227,459* | *134* |

**Table 5**: Statistics about the generated traces

avoid collisions. Notice that the number of generated test cases depends not only on the size of the CTG, but also on the diversity of possible sequences. For instance, whereas the CTG of highway with TP "arrival" is larger, there are fewer test cases than for the same configuration with TP "collision".

**AV scenarios and CMCDOT traces.** Based on the 8 AV scenarios shown in Table 4, we generated a total of 1703 traces of CMCDOT outputs. As shown in Table 4, the number of TCs per scenario ranges from 3 to 589. To obtain a number of traces more balanced among the AV scenarios, we varied the ratio of traces per TC for each AV scenario. The diversity of traces was increased by executing AV scenarios in CARLA with different target speeds (20, 25, 30, 35, and 40 km/h depending on the scenario) for the vehicles, which accelerate to these speeds when performing their moves. Statistics on the generated traces are shown in Table 5: the number of traces per scenario ranges from 90 to 666. For AV scenarios with few TCs (i.e., AV scenarios X-crossroad collision and static-collision), we executed each TC 10 times at each speed. For AV scenarios with many TCs (i.e., cross-no-collision, both highway scenarios, and running-example), we did not execute all TCs at every speed, but we chose instead a sufficient number of TCs and randomly distributed them among the speeds. Finally, for AV scenarios with enough TCs (i.e., car-obst-par), we executed each TC at every speed.

**Verification of CMCDOT traces.** To obtain and analyze verification results, we used CADP/XTL observers to parse the execution traces. These observers identified for each property, which trace events violated the properties and provided proofs of violation formally called *certificates*. Certificates contain all necessary information to indicate why a trace event violated a property.

All certificates contain for each violating event the trace name and the timestamp of the event in the trace. More information is given depending on the property. For the *coherence* property, a certificate is just the risk triple (i.e., the three risks at 1, 2, and 3 seconds) that was found to be illegal. For the *safe prediction*, a certificate is the prediction given by the risk triple together with the timestamp of the collision. Since there is also the timestamp of the event, it is easy to identify using the timestamps which part of the risk prediction triple is wrong. For the *proper progress*, a certificate is the risk-triple pair of the current and previous events. It is then easy to identify that the risk estimation

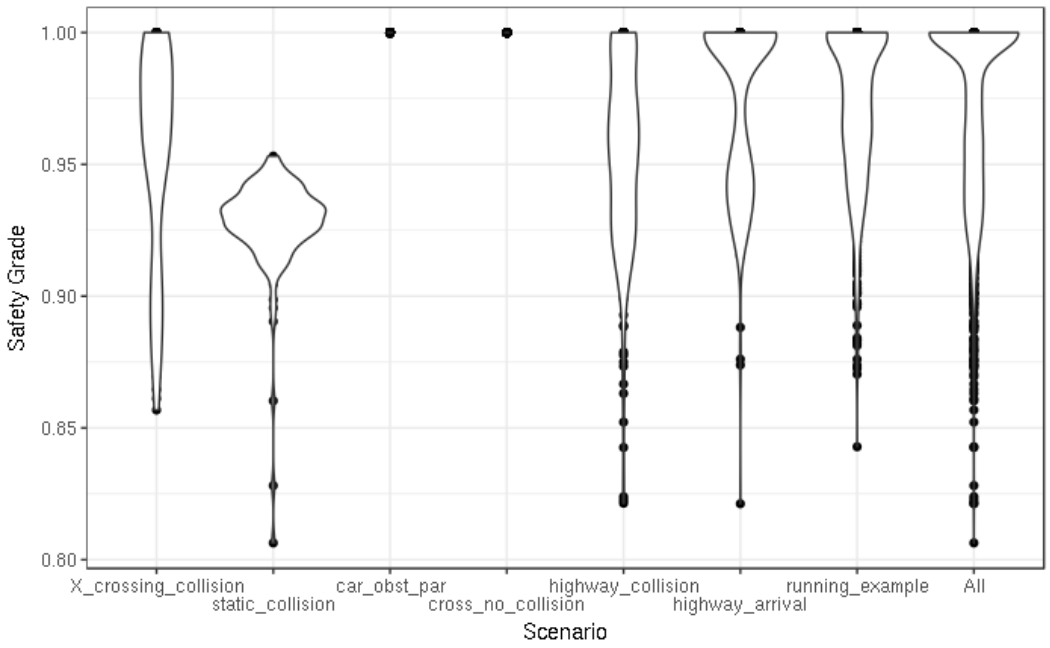



**Fig. 17**: Safe prediction grades for each scenario type

evolved the wrong way. To make sure that we do not catch false negatives, it is also interesting to give the segment numbers.

All these certificates are collected in CSV (comma-separated values) files (one for each trace and then one regrouping all CSV files for each scenario type). The analysis is done with the open source tool R-Studio. Using the certificates, we compute a penalty for each trace based on the *property grades* paragraph of Section 6.3. This gives us a grade (between 0 and 1) per property for each trace and by extension for each scenario type. The analysis results are shown in Figures 17 and 18 for *safe prediction* and the *proper progression* properties, respectively. Since the *coherence* property is a sanity property, we only allowed ourselves to work with traces where all predictions were coherent. Therefore, there is no need to show them as all traces have perfect grades for coherence.

The executions illustrated in Figures 17 and 18 are grouped by scenario. Each point on the vertical axis is the grade for one execution. Since many grades are very close and overlap on the plot, we use a violin shape to indicate as well the density of the points with similar grades. When the violin shape is wide, many points have the same grade, whereas when the violin shape is just a line, only few to no executions have this grade. In those situations, it is possible to see the dots. In each figure, the rightmost column shows the general distribution of all the grades. The sorting of the traces by scenario helped us to identify and debug some outliers among our results.

Figure 17 shows the grades for the *safe prediction* property. The majority of the grades are above 0.90 and the lowest one is at 0.81. Overall these are good grades, but rigorously one would expect safety grades to be perfect. Hence it is important to understand the reason for these grades.

The scenario that seems to be the worst is static-collision. Here, a car is sitting in the middle of the road and the ego vehicle drives straight into it. This is the simplest scenario possible and yet the grades seem bad. This is actually because CMCDOT is known to predict collisions in advance. For example, if a trace contains a collision on an event $e_{col}$, all events during the three seconds preceding $e_{col}$ should be predicting it properly, indicating that it will respectively happen within three, two, and one seconds. In practice, CMCDOT will start converging to the certainty of a collision before the time-frame that it is indicating. This is done to be able to help the decision by giving the warnings slightly in advance. However, from a formal perspective CMCDOT is arguably wrong in its early predictions. The fact that the grades are high anyway indicates that only a few predictions are early and the impact of these early predictions on decision-making is beneficial.

On the contrary, the two scenarios without collision (car-obst-par where two cars advance in parallel without colliding and cross-no-collision) only have traces with a perfect grade. This indicates that CMCDOT maintained the entire time that collisions would not occur.

The other scenarios contain collisions in more dynamic contexts. Part of the grades can be explained by early predictions. Since the contexts are more dynamic, there are less opportunities to have an early prediction because of the context changes expressed in our segments. These segments are necessary, because when the context changes too much (vehicle turning or significantly changing speed), the predictions become irrelevant. However, halting the exploration for the presence or absence of collisions at the end of the segment, can give some traces better grades than the traces should have, because sometimes the event with collision is the only event of the last segment. Some events that might have been early predictions are not reported as such and are not penalized for it. This is why apart from the static-collision scenario with grades accumulating around 0.94, the other grades are pulled out more and even reaching 1. In general, CMCDOT seems to be quite accurate with its important predictions, especially when estimating that no collision will occur.

Figure 18 shows the grades for the *proper progression* property. The scale of the plot is worth noting: the lowest point corresponds to an execution trace with an excellent grade slightly higher than 0.994. The *proper progression* property ensured that during a same segment, the risk estimation can only advance forward according to Figure 14. In the traces, we were able to observe that the evolution of the collision prediction is usually forward as expected. The only times where it would move backwards are when segments changed. The results show that the observations match what we expected from this property. Most of the traces have a perfect score of 1. Only one execution has a grade slightly lower than 1, which is caused by the strictness of our interpretation thresholds and minor hesitations for CMCDOT. We chose 10% and 90% thresholds in our interpretation. The penalized event is in a context where the vehicles are still some distance away and more than three seconds apart. CMCDOT moved some of its predictions to 12% risk and then back

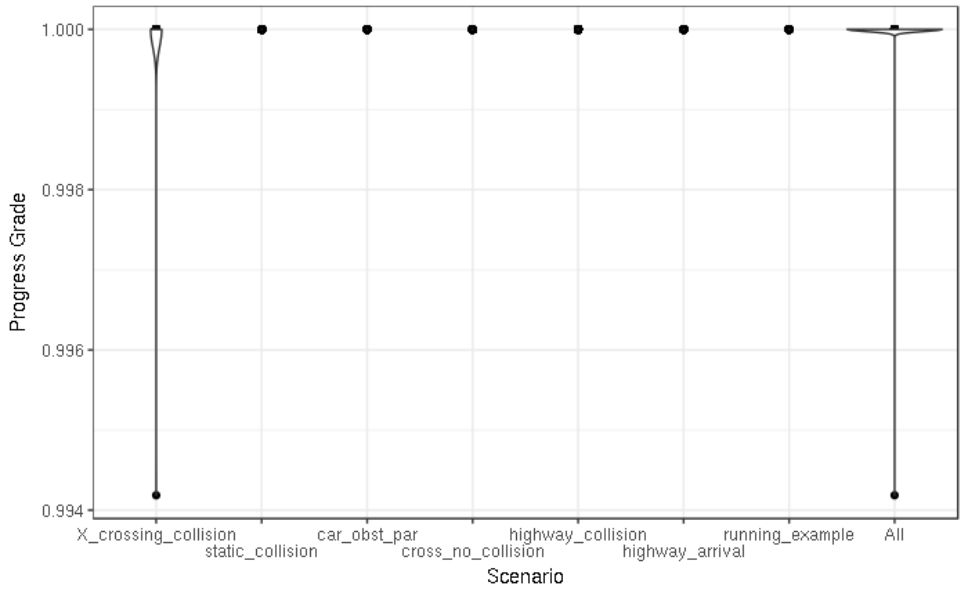



**Fig. 18**: Safe prediction grades for each scenario type.

to under 10% risk. We are quite confident about our interpretation thresholds because only one event is caught not respecting the property. The fact that only a few events have the incorrect evolution does not question our approach. On the contrary, it increases the confidence in CMCDOT in terms of risk evolution. Moreover, CMCDOT is very rarely hesitating, and when so, it is just for a few events that are long in advance.

It is worth mentioning that the results presented here have not always been as good. During the experiments, some traces had drastic outliers compared to other traces from the same scenario. This triggered us to adjust our simulator usage. In some scenarios, when the test case required the vehicles to drive close to each other without colliding (near miss situations), it could happen that the way we positioned the cars in the simulator led the cars to barely touch. In very close situations, sometimes the resolution of CMCDOT's grid did not exactly match the position of the car in the simulator. Also, premature collisions happened a few times, causing the trace observers to indicate that all predictions during the three seconds before the premature collision were wrong. The grading system we use makes sure that in these circumstances, the trace would get a very low grade close to zero. Because we were able to replay those scenarios ten times without the premature collision happening again, we cleaned up our data set (replacing each outlying trace by one generated during the replay). We consider that those were technical issues with the simulation rather than problems with our methodology.

# 8 Conclusion

In this paper we proposed an automated approach to statistically validate a perception component using scenarios automatically generated from a high-level description of situations. Concretely, our approach first automatically

generates from a formal model describing a configuration (e.g., a scene and constraints on actors trajectories) and high-level description of a situation (e.g., a collision or a near miss must occur), relevant scenarios in the form of behavior trees for an AV simulator. The formal method used to generate these scenarios guarantees the generation of all possible scenarios relevant to the given high-level description of the situation and feasible in the specified configuration. Then, our approach runs the AV scenario generated in a simulator comprising the perception component, generating simulation traces featuring the probabilistic estimation perceptions outputs. Finally, our approach verifies the accuracy of these perception estimations using a combination of model checking the execution traces with statistical analysis. In this paper, we instantiated our approach with the CADP and TESTOR formal tools, the CARLA AV simulator, the perception algorithm CMCDOT, and the R-Studio statistical computing environment. The experiments assessing the collision-risk estimation of CMCDOT that we carried out on more than 1700 complex simulation traces demonstrated the effectiveness of our approach and also enabled us to automatically detect punctual simulator bugs (outliers).

Our approach combines two steps, namely AV scenario generation and trace verification, and provides contributions to each of these steps.

Regarding AV scenario generation, the test suite generated by our approach could be used to generate abstract parametric scenarios usable as an input for existing methods [23–27], which can generate from each abstract parametric AV scenario several concrete AV scenarios by instantiating the values of the parameters. In [26] an evolutionary algorithm is used to reduce the reachable space of the ego vehicle in a driving scenario. This method takes as input the trajectory of the actors of the scenario and the road topology (drivable space), all of which are contained in the test cases generated with our method. A similar approach could be used with the work in [27], which is similar to [26] but uses an optimization technique instead of an evolutionary algorithm. In [23–25] scenarios are generated by optimizing parametric abstract scenarios. By allowing modifications of the trajectories restricted to given intervals, the test cases generated by our method could be used as inputs for these optimizing methods, enabling to obtain AV scenarios with more refined trajectories, which can be used to test control components. To do so, one must update the configuration and the test purpose with the desired parametric arguments, which is straightforward by using the configuration template and the user-friendly formal LNT language [49]. Also, because our AV generation approach uses high-level test purposes to directly generate several scenarios with different actor behaviors, which cover all of the possible outcomes of an abstract scenario, it could also be used for building datasets for training autonomous driving models or control components.

Regarding trace verification, our approach—even if we applied it to validate a specific perception component, namely CMCDOT—is generic and can be applied to other kinds of perception components. This is possible because

traces are checked as testimonies of past executions by generating automatically the necessary functions for querying the traces, such as the ones of other perception components computing risk estimations [69]. More precisely, given a trace structure, the functions to get, check, and obtain sets of edges presented in traces are automatically generated for each field of the trace. The trace structure consists at giving, for each of its fields, a name (`P_col_1`), variable name (`p1`), type (`real`), and a value to return when the pattern matching failed (`-(1.0)`). The trace format can therefore be easily adapted to accommodate any events and XTL can be used to describe observer automata that can parse the traces. Although XTL might seem a verbose language, its semantics combining graph theory and mathematics guarantees the generated observer automata to be correct by construction. More generally, our verification approach is appropriate for any offline trace verification, through the concepts of observer automata advocated for runtime verification [70].

Currently, the verification of the perception component is limited to one actor at a time: we plan to extend our approach to verify several actors at a time as future work. We also plan to evaluate the applicability of our approach for verifying other autonomous vehicle software components, such as the control component, firstly by adapting our formal model [53] and incorporating traffic-sign rules [71] and secondly by specifying the related verification properties. Concerning the selection of scenarios, our approach permits to formally express requirements defined in standards, such as ISO 26262, as test purposes (similarly to [72]) and thus to generate corresponding test cases for the validation of an AV.

## Acknowledgements

A part of this work was supported by the project ArchitectECA2030 that has been accepted for funding within the Electronic Components and Systems for European Leadership Joint Undertaking in collaboration with the European Union's H2020 Framework Programme (H2020/2014-2020) and National Authorities, under grant agreement No. 877539. A part of this work was supported by the PRISSMA project, co-financed by the French Grand Défi on Trustworthy AI for Industry.

# References

[1] Boudette, N.: 'It Happened So Fast': Inside a Fatal Tesla Autopilot Accident. https://www.nytimes.com/2021/08/17/business/tesla-autopilot-accident.html (2021)

[2] Fagnant, D.J., Kockelman, K.: Preparing a nation for autonomous vehicles: opportunities, barriers and policy recommendations. Transportation Research Part A: Policy and Practice **77**, 167–181 (2015)

[3] Redmond, A.M.: A Critical Review of Photonic Opportunities within Autonomous Vehicles Transport System. In: Proceedings of the 6th International Forum on Research and Technology for Society and Industry (RTSI'2021), Naples, Italy, pp. 188–193. IEEE, Italy section (2021). https://doi.org/10.1109/RTSI50628.2021.9597361

[4] McCarthy, J.: In: Colburn, T.R., Fetzer, J.H., Rankin, T.L. (eds.) Towards a Mathematical Science of Computation, pp. 35–56. Springer, Dordrecht (1993). https://doi.org/10.1007/978-94-011-1793-7_2

[5] Garavel, H., Graf, S.: Formal Methods for Safe and Secure Computer Systems - BSI Study 875. BSI German Federal Office for Information Security, Bonn (2013)

[6] Urmson, C., Anhalt, J., Bagnell, D., Baker, C., Bittner, R., Clark, M., Dolan, J., Duggins, D., Galatali, T., Geyer, C., *et al.*: Autonomous driving in urban environments: Boss and the urban challenge. Journal of Field Robotics **25**(8), 425–466 (2008)

[7] Leonard, J., How, J., Teller, S., Berger, M., Campbell, S., Fiore, G., Fletcher, L., Frazzoli, E., Huang, A., Karaman, S., *et al.*: A perception-driven autonomous urban vehicle. Journal of Field Robotics **25**(10), 727–774 (2008)

[8] Ding, W., Chen, B., Xu, M., Zhao, D.: Learning to collide: An adaptive safety-critical scenarios generating method. In: International Conference on Intelligent Robots and Systems (IROS), pp. 2243–2250 (2020). IEEE

[9] Dosovitskiy, A., Ros, G., Codevilla, F., Lopez, A., Koltun, V.: CARLA: An open urban driving simulator. In: Proceedings of the 1st Annual Conference on Robot Learning, pp. 1–16 (2017)

[10] Riedmaier, S., Ponn, T., Ludwig, D., Schick, B., Diermeyer, F.: Survey on scenario-based safety assessment of automated vehicles. IEEE Access **8**, 87456–87477 (2020). https://doi.org/10.1109/ACCESS.2020.2993730

[11] Bishop, P., Bloomfield, R.: A Methodology for Safety Case Development. In: Redmill, F., Anderson, T. (eds.) Proceedings of the Sixth Safety-critical Systems Symposium on Industrial Perspectives of Safety-critical Systems, Birmingham, UK, pp. 194–203. Springer, London (1998). https://doi.org/10.1007/978-1-4471-1534-2_14

[12] Ledent, P., Paigwar, A., Renzaglia, A., Mateescu, R., Laugier, C.: Formal Validation of Probabilistic Collision Risk Estimation for Autonomous Driving. In: CIS-RAM 2019 - 9th IEEE International Conference on Cybernetics and Intelligent Systems (CIS) Robotics, Automation and Mechatronics (RAM), pp. 1–6. IEEE, Bangkok, Thailand (2019). https:

//hal.inria.fr/hal-02355551

[13] Horel, J.-B., Laugier, C., Marsso, L., Mateescu, R., Muller, L., Paigwar, A., Renzaglia, A., Serwe, W.: Using Formal Conformance Testing to Generate Scenarios for Autonomous Vehicles. In: DATE/ASD 2022 - Design, Automation and Test in Europe - Autonomous Systems Design. IEEE, Antwerp, Belgium (2022). https://hal.inria.fr/hal-03516799

[14] Jard, C., Jéron, T.: TGV: Theory, Principles and Algorithms – A Tool for the Automatic Synthesis of Conformance Test Cases for Non-Deterministic Reactive Systems. Springer International Journal on Software Tools for Technology Transfer (STTT) **7**(4), 297–315 (2005)

[15] Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2011: A Toolbox for the Construction and Analysis of Distributed Processes. Springer International Journal on Software Tools for Technology Transfer (STTT) **15**(2), 89–107 (2013)

[16] Garavel, H., Lang, F., Serwe, W.: From LOTOS to LNT. In: ModelEd, TestEd, TrustEd – Essays Dedicated to Ed Brinksma on the Occasion of His 60th Birthday. LNCS, vol. 10500, pp. 3–26. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68270-9_1

[17] Mateescu, R., Thivolle, D.: A model checking language for concurrent value-passing systems. In: Cuéllar, J., Maibaum, T.S.E., Sere, K. (eds.) Proceedings of the 15th International Symposium on Formal Methods (FM'08), Turku, Finland. Lecture Notes in Computer Science, vol. 5014, pp. 148–164. Springer, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-68237-0_12

[18] Mateescu, R., Garavel, H.: XTL: A Meta-Language and Tool for Temporal Logic Model-Checking. In: Margaria, T. (ed.) Proceedings of the International Workshop on Software Tools for Technology Transfer (STTT'98), Aalborg, Denmark, pp. 33–42 (1998). BRICS

[19] Marsso, L., Mateescu, R., Serwe, W.: TESTOR: A Modular Tool for On-the-Fly Conformance Test Case Generation. In: 24th Int. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'18). LNCS, vol. 10806, pp. 211–228. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89963-3_13

[20] Marsso, L., Mateescu, R., Serwe, W.: Automated Transition Coverage in Behavioural Conformance Testing. In: 32nd IFIP Int. Conference on Testing Software and Systems (ICTSS'20), Naples, Italy, pp. 219–235. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-64881-7_14

[21] Rummelhard, L., Négre, A., Laugier, C.: Conditional monte carlo dense

occupancy tracker. In: IEEE 18th International Conference on Intelligent Transportation Systems, pp. 2485–2490 (2015)

[22] Grolemund, G., Wickham, H.: R for Data Science. O'Reilly Media, Inc (2016)

[23] Tuncali, C.E., Pavlic, T.P., Fainekos, G.E.: Utilizing s-taliro as an automatic test generation framework for autonomous vehicles. In: 19th IEEE International Conference on Intelligent Transportation Systems (ITSC), Rio de Janeiro, Brazil, pp. 1470–1475 (2016). https://doi.org/10.1109/ITSC.2016.7795751

[24] Gangopadhyay, B., Khastgir, S., Dey, S., Dasgupta, P., Montana, G., Jennings, P.A.: Identification of test cases for automated driving systems using bayesian optimization. In: 22nd IEEE Intelligent Transportation Systems Conference (ITSC), Auckland, New Zealand, pp. 1961–1967 (2019). https://doi.org/10.1109/ITSC.2019.8917103

[25] Khastgir, S., Dhadyalla, G., Birrell, S., Redmond, S., Addinall, R., Jennings, P.: Test scenario generation for driving simulators using constrained randomization technique. Technical report, SAE Technical Paper (2017)

[26] Klischat, M., Althoff, M.: Generating critical test scenarios for automated vehicles with evolutionary algorithms. In: IEEE Intelligent Vehicles Symposium (IV), pp. 2352–2358 (2019). https://doi.org/10.1109/IVS.2019.8814230

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

[28] Krajewski, R., Moers, T., Nerger, D., Eckstein, L.: Data-driven maneuver modeling using generative adversarial networks and variational autoencoders for safety validation of highly automated vehicles. In: Zhang, W., Bayen, A.M., Medina, J.J.S., Barth, M.J. (eds.) 21st IEEE International Conference on Intelligent Transportation Systems (ITSC), Maui, HI, USA, pp. 2383–2390 (2018). https://doi.org/10.1109/ITSC.2018.8569971

[29] Li, Y., Tao, J., Wotawa, F.: Ontology-based test generation for automated and autonomous driving functions. Information and software technology **117**, 106200 (2020)

[30] Singh, V., Pitale, M.: Impact of automotive system safety design on machine learning based perception systems. In: Proceedings of the 4th IEEE International Conference on Industrial Cyber-Physical Systems, (ICPS'2021), Victoria, BC, Canada, pp. 591–596 (2021). https://doi.org/10.1109/ICPS49255.2021.9468225

[31] Redmon, J., Divvala, S.K., Girshick, R.B., Farhadi, A.: You only look once: Unified, real-time object detection. In: Proceedings of the 29th IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Las Vegas, NV, USA, pp. 779–788 (2016). https://doi.org/10.1109/CVPR.2016.91

[32] Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S.E., Fu, C., Berg, A.C.: SSD: single shot multibox detector. In: Leibe, B., Matas, J., Sebe, N., Welling, M. (eds.) Proceedings of the 14th European Conference on Computer Vision (ECCV'2016), Amsterdam, The Netherlands. Lecture Notes in Computer Science, vol. 9905, pp. 21–37. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46448-0_2

[33] Ren, S., He, K., Girshick, R.B., Sun, J.: Faster R-CNN: towards real-time object detection with region proposal networks. IEEE Trans. Pattern Anal. Mach. Intell. **39**(6), 1137–1149 (2017). https://doi.org/10.1109/TPAMI.2016.2577031

[34] Zhou, Y., Tuzel, O.: Voxelnet: End-to-end learning for point cloud based 3d object detection. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 4490–4499 (2018)

[35] Lang, A.H., Vora, S., Caesar, H., Zhou, L., Yang, J., Beijbom, O.: Pointpillars: Fast encoders for object detection from point clouds. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 12697–12705 (2019)

[36] Shi, S., Guo, C., Jiang, L., Wang, Z., Shi, J., Wang, X., Li, H.: PV-RCNN: point-voxel feature set abstraction for 3d object detection. In: Proceedings of the 2020 IEEE/CVF International Conference on Computer Vision and Pattern Recognition (CVPR), Seattle, WA, USA, pp. 10526–10535 (2020). https://doi.org/10.1109/CVPR42600.2020.01054

[37] Wicker, M., Huang, X., Kwiatkowska, M.: Feature-Guided Black-Box Safety Testing of Deep Neural Networks. In: Proceedings of the 24th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'2018), Thessaloniki, Greece. Lecture Notes in Computer Science, vol. 10805, pp. 408–426. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89960-2_22

[38] Melis, M., Demontis, A., Biggio, B., Brown, G., Fumera, G., Roli, F.: Is Deep Learning Safe for Robot Vision? Adversarial Examples against the Icub Humanoid. In: Proceedings of the IEEE International Conference on Computer Vision Workshops, pp. 751–759 (2017)

[39] Serban, A., Poll, E., Visser, J.: Adversarial Examples on Object Recognition: A Comprehensive Survey. ACM Computing Surveys (CSUR) **53**(3),

1–38 (2020)

[40] Huang, X., Kwiatkowska, M., Wang, S., Wu, M.: Safety verification of deep neural networks. In: Majumdar, R., Kuncak, V. (eds.) Proceedings of the 29th International Conference on Computer Aided Verification (CAV'2017), Heidelberg, Germany. Lecture Notes in Computer Science, vol. 10426, pp. 3–29. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_1

[41] Shekar, A.K., Gou, L., Ren, L., Wendt, A.: Label-free robustness estimation of object detection cnns for autonomous driving applications. Int. J. Comput. Vis. **129**(4), 1185–1201 (2021). https://doi.org/10.1007/s11263-020-01423-x

[42] Wu, W., Xu, H., Zhong, S., Lyu, M.R., King, I.: Deep validation: Toward detecting real-world corner cases for deep neural networks. In: Proceedings of the 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'2019), Portland, OR, USA, pp. 125–137 (2019). https://doi.org/10.1109/DSN.2019.00026

[43] Zhang, M., Zhang, Y., Zhang, L., Liu, C., Khurshid, S.: Deeproad: Gan-based metamorphic testing and input validation framework for autonomous driving systems. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE'2018), Montpellier, France, pp. 132–142 (2018). https://doi.org/10.1145/3238147.3238187

[44] Hu, B.C., Marsso, L., Czarnecki, K., Salay, R., Shen, H., Chechik, M.: If a Human Can See It, So Should Your System: Reliability Requirements for Machine Vision Components. In: Proceedings of the 44th International Conference on Software Engineering (ICSE'22), Pittsburgh, PA, USA. ACM, USA (2022)

[45] Zhao, X., Robu, V., Flynn, D., Dinmohammadi, F., Fisher, M., Webster, M.: Probabilistic model checking of robots deployed in extreme environments. arXiv preprint arXiv:1812.04128 (2018)

[46] Calinescu, R., Ghezzi, C., Johnson, K., Pezzé, M., Rafiq, Y., Tamburrelli, G.: Formal verification with confidence intervals to establish quality of service properties of software systems. IEEE transactions on reliability **65**(1), 107–125 (2016)

[47] Barbier, M., Renzaglia, A., Quilbeuf, J., Rummelhard, L., Paigwar, A., Laugier, C., Legay, A., Ibañez-Guzmán, J., Simonin, O.: Validation of perception and decision-making systems for autonomous driving via statistical model checking. In: IEEE Intelligent Vehicles Symposium (IV), Paris, France, pp. 252–259 (2019). https://doi.org/10.1109/IVS.2019.

8813793

[48] Paigwar, A., Baranov, E., Renzaglia, A., Laugier, C., Legay, A.: Probabilistic collision risk estimation for autonomous driving: Validation via statistical model checking. In: IEEE Intelligent Vehicles Symposium (IV), Las Vegas, NV, USA, pp. 737–743 (2020). https://doi.org/10.1109/IV47402.2020.9304821

[49] Champelovier, D., Clerc, X., Garavel, H., Guerte, Y., McKinty, C., Powazny, V., Lang, F., Serwe, W., Smeding, G.: Reference Manual of the LNT to LOTOS Translator (Version 7.0). INRIA, Grenoble (2021)

[50] Tretmans, J.: Testing Concurrent Systems: A Formal Approach. In: Baeten, J.C.M., Mauw, S. (eds.) Proceedings of the 10th International Conference on Concurrency Theory (CONCUR'99), Eindhoven, The Netherlands. Lecture Notes in Computer Science, vol. 1664, pp. 46–65. Springer, Berlin, Heidelberg (1999). https://doi.org/10.1007/3-540-48320-9_6

[51] Charlesworth, A.: The Multiway Rendezvous. ACM Transactions on Programming Languages and Systems **9**(3), 350–366 (1987). https://doi.org/10.1145/24039.24050

[52] Garavel, H., Serwe, W.: The Unheralded Value of the Multiway Rendezvous: Illustration with the Production Cell Benchmark. In: 2nd Workshop on Models for Formal Analysis of Real Systems (MARS'17). EPTCS, vol. 244, pp. 230–270 (2017). https://doi.org/10.4204/EPTCS.244.10

[53] Marsso, L., Mateescu, R., Muller, L., Serwe, W.: Formally modeling autonomous vehicles in lnt for simulation and testing. In: Proceedings of the 5th Workshop on Models for Formal Analysis of Real Systems (MARS@ETAPS'2022), Munich, Germany. EPTCS (2022)

[54] Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge, Massachusetts (2001)

[55] Chang, E., Manna, Z., Pnueli, A.: Characterization of temporal property classes. In: Proceedings of the 19th ICALP (Vienna). Lecture Notes in Computer Science, vol. 623, pp. 474–486. Springer, Berlin (1992)

[56] Marsso, L., Mateescu, R., Parissis, I., Serwe, W.: Asynchronous testing of synchronous components in GALS systems. In: Proceedings of the 15th International Conference on Integrated Formal Methods (IFM'2019), Bergen, Norway. LNS, vol. 11918, pp. 360–378. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-34968-4_20

[57] Elfes, A.: Using occupancy grids for mobile robot perception and navigation. Computer **22**(6), 46–57 (1989)

[58] Moravec, H.: Sensor fusion in certainty grids for mobile robots. AI magazine **9**(2), 61 (1988)

[59] Fei, J., Peng, K., Heidenreich, P., Bieder, F., Stiller, C.: Pillarsegnet: Pillar-based semantic grid map estimation using sparse lidar data. In: 2021 IEEE Intelligent Vehicles Symposium (IV), pp. 838–844 (2021). IEEE

[60] Saha, A., Mendez, O., Russell, C., Bowden, R.: Translating images into maps. In: 2022 International Conference on Robotics and Automation (ICRA), pp. 9200–9206 (2022). https://doi.org/10.1109/ICRA46639.2022.9811901

[61] Philion, J., Fidler, S.: Lift, splat, shoot: Encoding images from arbitrary camera rigs by implicitly unprojecting to 3d. In: European Conference on Computer Vision, pp. 194–210 (2020). Springer

[62] Zhou, T., Yang, M., Jiang, K., Wong, H., Yang, D.: Mmw radar-based technologies in autonomous driving: A review. Sensors **20**(24) (2020). https://doi.org/10.3390/s20247283

[63] Hendy, N., Sloan, C., Tian, F., Duan, P., Charchut, N., Xie, Y., Wang, C., Philbin, J.: Fishing net: Future inference of semantic heatmaps in grids. arXiv preprint arXiv:2006.09917 (2020)

[64] Rummelhard, L., Nègre, A., Perrollaz, M., Laugier, C.: Probabilistic grid-based collision risk prediction for driving application. In: Springer (ed.) International Synposium on Experimental Robotics (2014)

[65] Kaempchen, N., Schiele, B., Dietmayer, K.: Situation assessment of an autonomous emergency brake for arbitrary vehicle-to-vehicle collision scenarios. IEEE Transactions on Intelligent Transportation Systems **10**(4) (2009)

[66] Garavel, H.: Binary coded graphs: Definition of the bcg format. Rapport SPECTRE C28, Laboratoire de Génie Informatique – Institut IMAG, Grenoble (January 1991)

[67] Alpern, B., B. Schneider, F.: Verifying temporal properties without temporal logic. ACM Transactions on Programming Languages and Systems (TOPLAS) **11**, 147–167 (2001). https://doi.org/10.1145/59287.62028

[68] Ledoux, V., Krishnakumar, R., Hervé, V.: Livrable L2.8 Situations d'interactions accidentogènes : enjeux. financé par la FSR et la DSR (2019). https://surca.univ-gustave-eiffel.fr/livrables-et-publications/

wp2-etat-de-lart-donnees-accidentologiques/

[69] Lefèvre, S., Vasquez, D., Laugier, C.: A survey on motion prediction and risk assessment for intelligent vehicles. ROBOMECH Journal **1**(1) (2014). https://doi.org/10.1186/s40648-014-0001-z

[70] Bartocci, E., Falcone, Y., Francalanza, A., Reger, G.: Introduction to runtime verification. In: Lectures on Runtime Verification - Introductory and Advanced Topics. Lecture Notes in Computer Science, vol. 10457, pp. 1–33. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-75632-5_1

[71] Bagschik, G., Menzel, T., Maurer, M.: Ontology based scene creation for the development of automated vehicles. In: IEEE Intelligent Vehicles Symposium (IV), pp. 1813–1820 (2018). https://doi.org/10.1109/IVS.2018.8500632

[72] Makartetskiy, D., Marchetto, G., Sisto, R., Valenza, F., Virgilio, M., Leri, D., Denti, P., Finizio, R.: (User-friendly) formal requirements verification in the context of ISO26262. Engineering Science and Technology, an International Journal **23**, 494–506 (2020). https://doi.org/10.1016/j.jestch.2019.09.005