Automatic Generation of Test-cases of Increasing Complexity for Autonomous Vehicles at Intersections

Abolfazl Karimi
University of North Carolina at Chapel Hill
ak@cs.unc.edu

Parasara Sridhar Duggirala
University of North Carolina at Chapel Hill
psd@cs.unc.edu

ABSTRACT

This paper presents a new framework for generating test-case scenarios for autonomous vehicles. We address two challenges in automatic test-case generation: first, a formal notion of test-case *complexity*, and second, an algorithm to generate more-complex test-cases. We characterize the complexity of a test-case by its set of solutions, and compare two complexities by the subset relation. The novelty of our definition is that it only relies on the pass-fail criteria of the test-case, rather than indirect or subjective assessments of what may challenge an ego vehicle to pass a test-case. Given a test-case, we model the problem of generating a more complex test-case as a constraint-satisfaction search problem. The *search variables* are the changes to the given test-case, and the *search constraints* define a solution to the search problem. The constraints include steering geometry of cars, the geometry of lanes, the shape of cars, traffic rules, bounds on longitudinal acceleration of cars, etc. To conquer the computational challenge, we divide the constraints to three categories and satisfy them with simulation, answer set programming, and SMT solving. We have implemented our algorithm using the Scenic libraries and the CARLA simulator and generate test-cases for several 3-way and 4-way intersections with different topologies. Our experiments demonstrate that both CARLA's autopilot and autopilot-plus-RSS (Responsibility-Sensitive Safety) can fail as the complexity of test-cases increase.

KEYWORDS

test-case complexity, test-case generation, traffic rules, logic programming, Answer Set Programming, Bezier curves, SMT, CARLA, Scenic

1 INTRODUCTION

There are uncountably many traffic scenarios! This is a challenge for scenario-based approaches for specifying the requirements and Operational Design Domain (ODD), development, testing, falsification, and certification of Autonomous Vehicles (AVs). A variety of methods are proposed in the literature to explore different subsets of the infinite space of test-case scenarios. Riedmaier et al. [22] survey a category of test-case generation methods that are based on some notion of *complexity*. Here, *complexity* is a property of a test-case, i.e. determinable before execution, and so independent of the performance of a particular AV. The idea is that the probability of a failure increases as the complexity of the test-cases increase. Complexity is distinguished from *criticality*, which is an assessment of the performance of an AV in a scenario, and is determinable only after test-case execution. Complexity is similar to *difficulty* in game design [1], where a more difficult level should be more challenging with respect to all possible players.

A *scenario* is a sequence of *scenes*. A *test-case* is a scenario in which the description of *ego* (i.e. vehicle-under-test) in the scenes is unspecified, and has some *pass-fail* criteria that determine what ego behaviors would pass the test-case [25]. Bagschik et al. [2] give a five-layer ontology for a scenario: roads, traffic infrastructure, temporary changes in roads and infrastructure, objects, and weather. One aspect of complexity concerns the perception task, e.g. detecting the road boundaries and traffic signs, tracking objects, etc. For example, faded lane markings, occlusions and rainy weather can make perception more complex. On the other hand, an AV with complete information about its environment still has the challenge of navigating around to pass a test-case, e.g. avoiding collisions, reaching its goal, and following traffic rules. That is, another aspect of complexity concerns the planning and control task. In this paper we address the latter.

First, we propose a formal definition of test-case complexity. Our definition is objective in the sense that it does not rely on subjective assessments of what features may challenge an AV to pass a test-case. Instead, we rely directly and only on the pass-fail criteria.

Second, we propose an algorithm to generate more-complex test-case scenarios. Our technique can handle pass-fail criteria that regard the traffic rules and right-of-way at an intersection, in addition to goal-reach and collision avoidance. Similar to [13], we expect the traffic rules to be provided in a logic program (more precisely an Answer Set Program [18]). Our algorithm takes as input the geometry of the traffic intersection, traffic rules to be followed at the intersection, and the routes of the vehicles and generates several test-case scenarios with increasing order of complexity. Our algorithm gives full coverage over some subspaces of the possible test-cases: after the set of lane events of two cars are fixed, there are only a finite number of relative temporal order of these events that may result in a more complex test-case, and our algorithm uses an ASP solver to do an exhaustive search over this subspace.

Third, we generate test-cases for a four-way stop, a T-intersection, and an uncontrolled Y-intersection. Then we execute these test-cases to test CARLA's autopilot and autopilot-plus-RSS in the CARLA simulator. We incrementally increased the complexity of test-cases and discovered instances where the CARLA autopilot failed test-cases by violating a traffic rule or colliding with a non-ego vehicle. Also we observed that restricting the behavior of CARLA's autopilot with RSS improved its rate of success, but did not guarantee passing a test-case.

1.1 Related work

In [10, 27, 28], a *complexity index* is defined based on some *influence factors* and their relative importance. The influence factors are based on the functionality of the AV that is going to be tested

and may include a mixture of perception-related and planning-and-control-related factors such as weather, rapid changes in light, lane line clarity, road curvature, road congestion, etc. These factors are derived from technical specifications, naturalistic traffic data, etc. The contribution of each factor to the complexity index is derived with a quantitative and subjective evaluation, the Analytic Hierarchy Process (AHP). In contrast to subjective assessments, we rely objectively on the pass-fail criteria alone to automatically decide if a test-case is more complex than another. We give sufficient conditions that guarantee that a test-case B is at least as complex as a test-case A, and a certificate to why B is more complex than A.

Wang et al. [26] give a measure of complexity of scenarios gathered from an AV. They first define a measure of scene complexity, then calculate its distribution across a scenario as a measure of its complexity. That is, the spatio-temporal relations between two consecutive scenes are ignored. Therefore, two scenarios with a completely different spatio-temporal development but the same distribution of scene complexity would be treated the same. The *scene complexity* is a weighted sum of *road semantic complexity* and *traffic element complexity*. *Road semantic descriptors* are features that are subjectively selected to contribute to the complexity measure, and obtained via manual annotations for a small subset of the data and generalized with supervised learning. *Traffic element complexity* is quantified in terms of non-egos' distance and orientation relative to the AV (ego), which makes their definition fall under criticality rather than complexity in our terminology above (adopted from [22]).

Qi et al. [21] propose a manual process to characterize a test-case based on a finite number of failing trajectories of ego. The causes of failure are analyzed to make a list of Scenario Character Parameters (SCPs) which could be related to perception, planning or control aspect of the driving task. In contrast, our characterization of a test-case is with respect to all (infinitely many) failing ego behaviors. Furthermore, our algorithm decides the cause of failure of a behavior automatically from the pass-fail criteria.

Apart from a few papers reviewed above that generate complex test-case scenarios, most literature on scenario generation/selection focus on other qualities and quantities of scenarios, such as *similarity* [12], *criticality* [15, 29], *corner cases* [20], etc. These techniques include a variety of algorithmic paradigms such as knowledge-based methods [17], data-driven methods [20], optimization-based search [6, 7, 16], genetic algorithms [4, 15, 29], synthesis from formal specification [16, 23], probabilistic search [9, 24], combinatorial search [10, 23, 28], etc. Only a few of the proposed techniques handle traffic rules, especially right-of-way, at intersections. In a survey on performing safety assessment of autonomous vehicles [22], the authors stress the importance of scenario generation at intersections. Here, we mention a few examples that are more related to our work.

Fremont et al. [9] use Scenic as a probabilistic programming DSL to generate scenarios. In contrast, we use Scenic only as an interface to CARLA, and generate the scenarios by solving logic programs. An advantage of logic programming over probabilistic programming is that a logic solver gives coverage guarantee over its search space. For example, in Step 2 of our algorithm, the ASP solver covers all possible order relations between events, where ASP's search space is restricted by the order relations that are fixed in Step 1.

Tuncali et al. [23], also Klischat and Althoff [16] generate test-cases for intersections from requirements and formal specifications. However, they do not include right-of-way traffic rules.

Calo et al. [4] use genetic algorithms to find *avoidable collision scenarios*. These are scenarios in which a particular AV crashes but it could have avoided the collision. This is somewhat similar to our complexity and solvability certificates: a behavior that fails a test-case and a behavior that passes the same test-case. However, our goal is different, i.e. finding a test-case that is more complex than a given test-case. Also, our pass-fail criteria are not limited to collisions.

In [14] SMT solving is used to generate test-cases. To generate trajectories, they use a predetermined geometric class, say piecewise linear or spline curve. Therefore, the generated trajectory is not necessarily feasible for the steering geometry of a vehicle, say with Ackermann steering. In contrast, we satisfy the steering constraints using simulation and closed-loop control, and use SMT only for generating the longitudinal speed.

2 TEST-CASE COMPLEXITY

We follow Ulbrich et al. [25] for the definitions of *scene*, *scenario* and *test-case*. We clarify their definition of a test-case by adding the terminology of a *partial* scenario and *execution* of a test-case. Then, we propose our definition of *test-case complexity*. Finally, we present some of the details of our formal representations.

2.1 Scenario

A *scene* defines the state of the environment at an instant in time. The environment includes the *scenery*, *dynamic elements*, and *actors*. The *scenery* are the geo-spatially stationary elements and all the metric, semantic and topological information they entail, such as a road, its width, an intersection and its intersecting lanes. A *dynamic element* is movable (moving or able to move), and an *actor* is an element that acts on its own, which are both only cars in our generated scenarios.

A *scenario* is a sequence of scenes, starting with an *initial* scene and spanning over a certain amount of time. *Actions and events* constitute any changes between consecutive scenes. These changes include movement of a car, change in its speed, activating a turn signal, entering or exiting a region, the speed reaching a threshold, etc. A scenario may specify a *goal* for an actor, and the actor may use it to select what action(s) to take. For example, reaching the end of a particular lane may be the goal of a car.

A scenario can be specified *partially*. Technically, a partial scenario corresponds to a set of scenarios that share the specified information. For example, the state of an actor in some scenes may be left unspecified. If the initial scene is fully specified, we can *execute* the (partial) scenario by starting with the initial scene, and applying the actions and events at each scene to get the next scene, until the specified duration has passed. The outcome of the execution is a fully specified scenario.

In this paper, we generate partial scenarios for simulation-based testing. Each generated scene specifies all non-egos completely, whereas ego is specified in the initial scene only. There is a fixed *time*

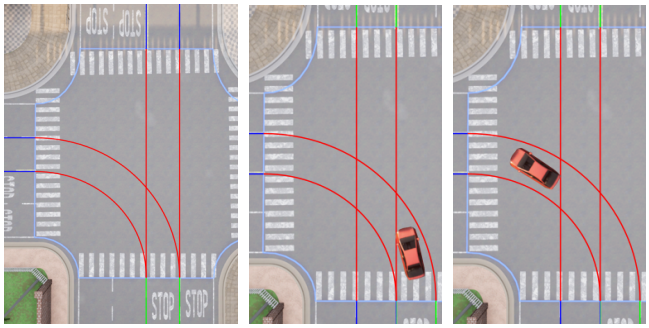
step between each pair of consecutive scenes. The execution of the scenario only determines the description of the ego in each scene. In the initial scene, ego is on an incoming lane to an intersection. The goal of ego is to reach a designated outgoing lane.

2.2 Test-cases and their complexity

A *test-case* is a partial scenario together with *pass-fail criteria*. Our pass-fail criteria consist of ego reaching its goal, avoiding collisions, and respecting the traffic rules. The pass-fail criteria partitions a partial scenario (i.e. its corresponding set of fully specified scenarios) into two sets: scenarios that pass and scenarios that fail. This is equivalent to partitioning the set of ego behaviors into those that pass and those that fail, since our partial scenarios specify everything in a scene except the state of ego. A passing behavior of ego is a *solution* of the test-case. A test-case is *solvable* if its solution set is nonempty.

The *complexity* is characterized by the set of solutions to a test-case. This induces a partial order on complexity via the subset relation. In particular, if the solutions of test-case B is a proper subset of the solutions of test-case A, then B is more complex that A. The difference of the two solution sets characterizes the increase in complexity. Note that solution sets and their difference are uncountably infinite sets due to the continuous parameters such as location, velocity, etc. Therefore, most of these sets cannot be represented explicitly, and the subset relation cannot be checked directly. Discretization of continuous variables does not help, for two reasons: First, complexity would not be unique e.g. under one discretization the solution sets of two test-cases may be in a subset relation, but under another discretization the solution sets may be incomparable. Second, a coarse discritization is unrealistic, while a fine discritization creates an enormous state space and does not help with representing the solution sets explicitly or computing the subset relation between them. Therefore, we take a different approach to ensure that a generated test-case is more complex than a given test-case.

We ensure that the complexity of a generated test-case is equal to or more than (i.e. a subset of) the complexity of a given test-case, by not changing the behaviors of the old non-egos. That is, keeping the behaviors of the old non-egos ensures that the old failing behaviors fail the new test-case as well. That is, all colliding behaviors of ego remain colliding, and all right-of-way-violating behaviors of ego remain violating. Therefore, we only add new non-egos. New non-egos should not collide with the old non-egos since a collision may alter an old non-ego's trajectory. Note that a new non-ego does not relieve ego from any right-of-way obligation that it already had to existing non-egos. This is because each right-of-way rule refers only to a pair of vehicles and is indifferent to other vehicles. In general, traffic rules also include *precedence rules* where a higher priority rule makes a lower priority rule inapplicable to the traffic context. In this situation, addition of a non-ego may not result in a subset relation between solution sets since it may introduce new solutions, in addition to possibly removing some solutions. For example, a new non-ego ambulance may make a no-stopping sign inapplicable to ego. Precedence rules are a challenge for controlling test-case complexity and are not addressed in this paper.



(a) Incoming, connecting, and outgoing lanes (green, red, blue). **(b) A car entering a lane that intersects its route (left turn).** **(c) A car exiting a lane that intersects its route (left turn).**

Figure 1: Lane events.

We ensure that the complexity of a generated test-case is more than (i.e. a proper subset of) the complexity of a given test-case, as follows. First we do not change the behaviors of old non-egos to ensure that the new complexity is a subset. Second, we add one or more non-egos such that at least one solution to the old test-case fails the new test-case because it violates the right-of-way of one or more of the new non-egos.

2.3 Formalization

We use Python objects to represent a scenario, and reuse the data structures in Scenic's driving domain.[1] The traffic rules are modeled as first-order logic sentences over the elements, attributes, states, actions, events, etc. in a scenario. These sentences are represented in a logic program, more specifically an Answer Set Program. In this section we explain some of the details.

The scenery is modeled as a road network in Scenic's driving domain. A road *network* is a collection of *roads* and *intersections*. An *intersection* connects two or more *roads* together. Each *road* is a collection of *lanes* and posted *signs*. A *lane* is a polygonal region of the road together with an associated driving direction. A lane is either a part of a road or a part of an intersection. The *incoming* lanes of an intersection are the road lanes that end at that intersection. The *outgoing* lanes of an intersection are the road lanes that start at that intersection. The *connecting* lanes of an intersection are intersection lanes that connect an incoming lane to an outgoing lane. A *route* is a triple of (incoming, connecting, outgoing) lanes at an intersection. See Figure 1 for (a) two routes from the same incoming lane, and (b) two routes from different incoming lanes. The connecting lanes are polygonal regions even though their boundaries appear smooth.

A car has attributes such as shape, wheelbase, maximum steering angle, etc. Scenic models the *shape* of a car as a rectangle, which simplifies computing the lane events (entrance and exit). The *location* of a car is the location of the center of its rectangle. The *state* of a car in a scene specifies its *pose* (location and orientation) and turn signal. The *behavior* of a car is the sequence of its states at each scene. When the shape of a car starts overlapping a lane, it

[1] https://scenic-lang.readthedocs.io/en/latest/libraries.html#driving-domain

enters the lane, and when it stops overlapping, it *exits* the lane. See Figure 1.

The traffic rules describe right-of-way rules and abiding by traffic signs. We model the traffic rules in an Answer Set Programming (ASP) language similar to [13]. Before presenting our formulation, we need a quick introduction to ASP. An ASP *program* is a collection of *rules* of the form:

```
h :- p_1, ..., p_n, not p_{n+1}, ..., not p_{n+m}.
```

We call `h` the *head* of the rule, and the list after `:-` is the *body* of the rule. The intended meaning of the above rule is

$$p_1 \wedge \cdots \wedge p_n \wedge \neg p_{n+1} \wedge \cdots \wedge \neg p_{n+m} \rightarrow h$$

Here each *condition* p_i, as well as h, is an *atomic predicate* over constants and variables. An atomic predicate or a rule is *ground* if it has no variables. A *fact* is a ground rule with no conditions. A rule is used to infer more facts, except if the head is empty. If the head is empty, we get a *constraint* which means that the conjunction of predicates $p_1 \ldots \neg p_{n+m}$ should not hold true. A *choice rule*, which is a syntactic sugar, is a rule where the head is a list of predicates to choose from. In particular,

```
{ h_1, ..., h_k } = N :- p_1, ..., not p_{n+m}.
```

specifies that exactly `N` predicates from the list `h_1, ..., h_k` must be inferred if the body of the rule is true. The domains of variables are determined using Herbrand semantics [19], that is from all the constants mentioned in the program. Given an ASP program, the ASP solver finds a set of facts (i.e. an *answer set*) that is closed under the rules of the program. If no such set exists, the ASP solver returns that the program is *unsatisfiable*.

Relations, actions, events, etc. in a scenario are included in an ASP program as facts. For example, if the incoming lane `road8_lane0` has a stop sign, we represent this with a fact `hasStopSign(road8_lane0)`. The event of `car1` arriving at the intersection at time `t1` from the incoming lane `road8_lane0` is represented by the fact

```
arrivedAtForkAtTime(car1, road8_lane0, t1).
```

Here, *fork* is an incoming lane together with its connecting lanes. Traffic rules are added to a program to infer the violations. For example, to infer the violations of the traffic rule *"the driver of any vehicle approaching a stop sign at the entrance to an intersection shall stop"* we add

```
violatedRule(V, stopAtSign) :-
  arrivedFromIncomingLane(V, L), hasStopSign(L),
  not stoppedAtIncomingLane(V, L).
```

Here, `violatedRule(V, stopAtSign)` is an atomic predicate that can be inferred to be true or false based on the grounding of variables `V` and `L`, and the truth of the rule's conditions. Traffic rule violations are found simply by checking whether the answer set includes a corresponding fact. In our formalization, we have two types of violations: `violatedRule(V, r)` and `violatedRightOfForRule(V1, V2, r)`. The first violation involves one vehicle `V` violating a rule named `r`, and the second is a violation where vehicle `V1` violated the right-of-way of vehicle `V2` according to the traffic rule named `r`. For example, the rule above is named with the term `stopAtSign`. Note that upper-case terms are variables and lower-case terms are constants. In the next section, we will see examples of choice rules used to search through possible orders between events, and constraints used to restrict the search space.

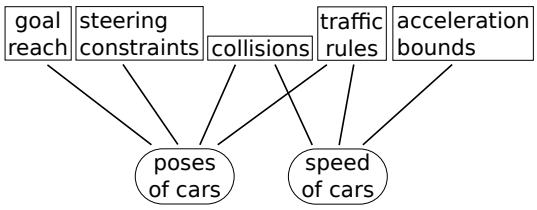


Figure 2: Test-case CSP problem.

We use the notion of *perceptible order* to model ordering of events in traffic rules. The parameter *minimum perceptible time difference* is a threshold beyond which events are considered simultaneous from the perspective of traffic rules. Suppose that m represents this parameter, and let s, t be two moments in time. Then s, t are *perceptibly simultaneous* if $|s - t| < m$. Otherwise, s *perceptibly precedes* t or vice versa, if $m \leq t - s$ or $m \leq s - t$, respectively. Perceptible simultaneity is not transitive since $|r - s| < m$ and $|s - t| < m$ do not imply $|r - t| < m$. Therefore, perceptible order is not a partial order. In contrast, *temporal order*, i.e. the standard real number order of time points, is a partial order. Consequently, we use separate binary relations to represent perceptible order and temporal order.

3 TEST-CASE GENERATION

Given a solvable test-case (referred to as the *old* test-case), the problem is to find a *new* solvable test-case that is more complex. We model this search problem as a constraint-satisfaction problem (CSP). First we explain this search problem, then we present our search algorithm.

3.1 Test-case search problem

The *search variables* are the behaviors of the new non-egos, a *complexity certificate*, and a *solvability certificate*. The *complexity certificate* is an ego behavior that solves the old test-case, but fails the new test-case. The *solvability certificate* is an ego behavior that solves the new test-case. The set of new non-egos and their routes through the intersection are given as parameters rather than as search variables.[2] As discussed in the previous section, we do not change the behaviors of the old non-egos so they are fixed parameters. The rest of the parameters, such as the scenery and traffic rules, are given by the old test-case.

There are three types of constraints imposed by the pass-fail criteria, to generate a more-complex and solvable test-case. First is collision avoidance, which regards the shape of each car, and its location and orientation at each time relative to other cars. The locations and orientations are themselves constrained by physics, such as limited forces available to accelerate or brake, and bounded range for steering angles. Second is goal-reach, which regards overlap of a trajectory with a region in the scenery. This in turn is constrained by physics of car motion, and also the scenery. Third is traffic rules, which regards the lane events, stop events, etc. This

[2]The reason we did not extend our algorithm to automatically select values for these parameters is that it is not clear how to do significantly better than a brute-force approach. Furthermore, a manual parameter assignment gives more control over the search.

depends on the logic of traffic rules, geometry of lanes, location and orientation of the cars at each time, also each car's shape. The space of possible complexity and solvability certificates depends on the choice of new non-ego behaviors. Therefore, the search for the new non-egos has to be done simultaneously as the search for the certificates.

Our proposed approach is to decompose the CSP problem into three subproblems, where each subproblem is suitable to a specialized technique. In particular, we use simulation and closed-loop control to find a sequence of poses that satisfies the steering constraints and goal-reach. We use ASP to find possible orderings of events that satisfy the required traffic rules violations or compliance. Finally, we use SMT to find timing of poses that satisfy the analytical constraints on the longitudinal motion e.g. smoothness of speed, bounded longitudinal acceleration, negligible speed at stop events, etc. Our decomposition of the problem is shown in Figure 2. The first subproblem finds the poses of cars, while the second and third subproblems find the speed of cars.

The new non-egos should not collide with the old non-egos since a collision may alter an old non-ego's trajectory which violates our guarantee of comparability of test-case complexities.

The complexity certificate is an ego behavior that passes the old test-case, but fails the new test-case. The constraints that guarantee such a behavior are as follows. The behavior should avoid colliding with the old non-egos, respect traffic signs and the right-of-way of old non-egos, and reach ego's goal, to pass the old test-case. The behavior should avoid colliding with the new non-egos, so that it is a physically valid behavior in the old test-case where the collision forces of the new non-egos are absent. Since the behavior respects traffic signs and does not collide with the new non-egos, the only way for it to fail the new test-case is to violate the right-of-way of at least one of the new non-egos.

The solvability certificate is an ego behavior that passes the new test-case. That is, the behavior does not collide with the old or new non-egos, respects traffic signs and the right-of-way of both the old and new non-egos, and reaches ego's goal.

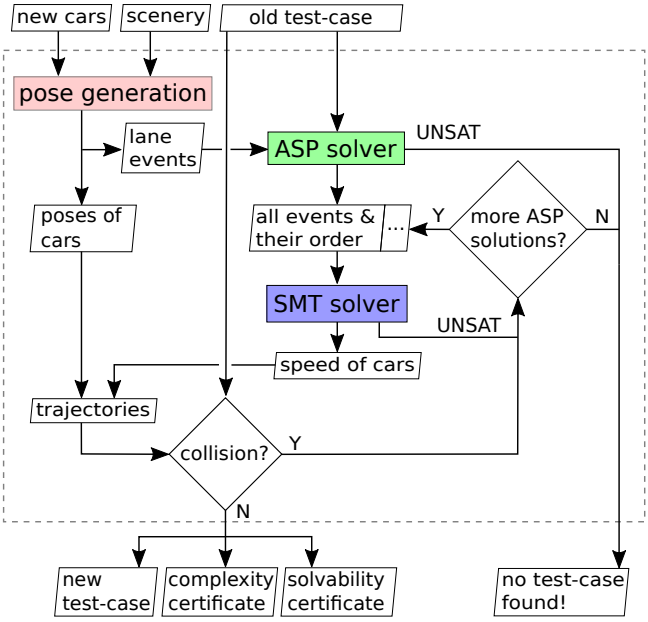


Figure 3: The test-case generation algorithm.

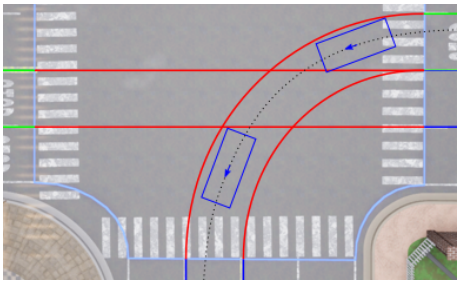


Figure 4: Two poses at which lane events happen are shown with arrows, and the car's shape with rectangles.

3.2 Test-case generation algorithm

Our algorithm has three main steps. These steps are highlighted with red, green and blue in Figure 3.

3.2.1 Step 1: pose generation. The first step is to determine the poses of each car such that they satisfy the car's *steering constraints*. Two consecutive poses p_1, p_2 at times t_1, t_2 respectively, satisfy the constraint if and only if there exists a steering input as a function of time over the interval $[t_1, t_2]$ that drives the car from p_1 to p_2. Note that a pose specifies both the location and the orientation of the car. The constraint depends on the steering mechanism of each car e.g. steering axles, maximum steering angles, wheelbase, etc.

The algorithm drives each car using a steering controller in a physics simulation, so the generated pose-sequence satisfies the steering constraints. The steering controller tracks the centerline of the car's route. A speed controller is used to track a constant speed. See Figure 4 for a pose sequence. A location is shown by a dot and the orientation is implied by the tangent to the dot sequence.

If the two ego trajectories (candidates for complexity and solvability certificates) reach ego's goal, our algorithm proceeds to Step 2. Note that the new non-egos do not need to reach their goal.

Otherwise, the algorithm stops and returns with failure. If we have a solvability certificate for the old test-case, we can simply use its poses and skip the simulations of ego, so Step 1 would not fail.

3.2.2 Step 2: deciding order of events. This step finds all possible order of events such that one of the ego behaviors would qualify as a complexity certificate, and the other as a solvability certificate. The set of events and their perceptible order affect the traffic rules violations. Some of this information is already fixed by the behavior of old non-egos. The lane events of each new car are computed from its pose sequence from Step 1, the car's shape, and scenery's lane geometry. Furthermore, for each car, the temporal order between its lane events are implied by the poses at which they happen and the order between poses in the sequence. Each event is assigned a symbolic time such that simultaneous lane events are assigned the

same time symbol. It remains to determine what stop events should happen and what is the perceptible order between all events.

Therefore, this step is a CSP where the *search variables* are the stop events and the perceptible order between events. The search space is finite since there are only a finite number of events and possible order between them. The *constraints* include the old events and their temporal and perceptible order, the new lane events and their temporal order, and the complexity and solvability criteria. Our algorithm specifies this search problem as an ASP program and solves it with an ASP solver.

The perceptible order between time symbols is represented by two binary relations `equal()` and `lessThan()`. If we instruct the ASP solver to make a choice about the perceptible order of every pair of time symbols, we make the search space unnecessarily big. Instead, we manually go through the set of traffic rules and add a choice rule only for each pair of events that their perceptible order matters to the right-of-way violations. This manual derivation has to be done only once for each traffic rule, say after modeling the traffic rules for an intersection. For example, consider the traffic rule

> "The driver of a vehicle approaching an intersection shall yield the right-of-way to any vehicle which has entered the intersection from a different highway,"

and its ASP encoding

```
violatesRightOfForRule(V1, V2, yieldToInside):-
  enteredForkAtTime(V1, F1, Te1),
  enteredForkAtTime(V2, F2, Te2),
  onDifferentHighways(F1, F2),
  requestedLane(V1, L1), requestedLane(V2, L2),
  overlaps(L1, L2),
  lessThan(Te2, Te1),
  leftLaneAtTime(V2, L1, T1),
  enteredLaneAtTime(V1, L2, T2),
  lessThan(T2, T1).
```

For this rule, we only need a choice for `lessThan(Te2, Te1)`:

```
{equal(Te1,Te2); lessThan(Te1,Te2); lessThan(Te2,Te1)}=1 :-
  enteredForkAtTime(V1, F1, Te1),
  enteredForkAtTime(V2, F2, Te2),
  onDifferentHighways(F1, F2),
  requestedLane(V1, L1), requestedLane(V2, L2),
  overlaps(L1, L2).
```

and a choice for `lessThan(T2, T1)`:

```
{equal(T1, T2); lessThan(T1, T2); lessThan(T2, T1) } = 1 :-
  enteredForkAtTime(V1, F1, Te1),
  enteredForkAtTime(V2, F2, Te2),
  onDifferentHighways(F1, F2),
  requestedLane(V1, L1), requestedLane(V2, L2),
  overlaps(L1, L2),
  lessThan(Te2, Te1),
  leftLaneAtTime(V2, L1, T1),
  enteredLaneAtTime(V1, L2, T2).
```

Temporal order is different from perceptible order as discussed earlier. The temporal order between time symbols is represented using a binary relation `realLTE()`. The 'LTE' stands for Less-Than-or-Equal-to and emphasizes that the binary relation is a partial order (and so transitive). The 'real' emphasizes the intended meaning of this relation as the order between real numbers. The order between real numbers is total but `realLTE()` is partial, since the temporal order between lane events of two different cars are not physically constrained and may not matter to traffic rules violations either. In particular, time symbols `s,t` are simultaneous if and only if `realLTE(s,t)` and `realLTE(t,s)`, `s` precedes `t` if and only if `realLTE(s,t)` and `not realLTE(t,s)`, and the order is unknown (and unimportant) if and only if `not realLTE(s,t)` and `not realLTE(t,s)`. For each new car, the temporal order between its lane events are already determined by Step 1, and are added as facts to the ASP program.

The new lane events, determined from Step 1, are added to the ASP program as facts. The time symbol of each event is used as a parameter of the fact. For example, the term `t1` in the fact `enteredLaneAtTime(car1, road258_lane0, t1)` is its time symbol.

Each old event is also added to the ASP program as a fact. Even though the real number value of time is known for old events, time symbols are instantiated so that old events can be represented as facts in the ASP program. The temporal and perceptible order of old events are determined from the numerical time of those events. The temporal and perceptible order between an old event and a new event is not a known fact and will be determined by the choice rules.

The occurrence of a stop event (at a stop sign) is a search variable, so we instruct the ASP solver to make a choice:

```
{stoppedAtForkAtTime(V, F, @time(V, stop))} = 1 :-
  arrivedAtFork(V, F), hasStopSign(F),
  not violatedRule(V, stopAtSign).
```

The term `@time(V, stop)` is simply the time symbol assigned to the stop event of car `V`. If a stop event happens, then it must be perceptibly after the car has arrived at the intersection and perceptibly before it entered the intersection.

```
lessThan(T1, T) :-
  arrivedAtForkAtTime(V, F, T1),
  stoppedAtForkAtTime(V, F, T).
lessThan(T, T2) :-
  stoppedAtForkAtTime(V, F, T),
  enteredForkAtTime(V, F, T2).
```

The ego behavior that is a candidate for a complexity certificate must satisfy the following constraints. Let `illegal` be the name of the car, `v1,...,vN` be the list of new non-egos, and `old()` be true of the old non-egos. The first constraint says that `illegal` does not violate the right-of-way of old non-egos:

```
:- old(V), violatesRightOf(illegal, V).
```

The second constraint says that `illegal` does not violate any other traffic rules, say stopping at a stop sign:

```
:- violatesRule(illegal, _).
```

These two constraints require `illegal`'s behavior to be a solution for the old test-case. The third constraint says that `illegal` violates the right-of-way of at least one of the new non-egos:

```
:- not violatesRightOf(illegal, v1), ...,
   not violatesRightOf(illegal, vN).
```

Hence `illegal`'s behavior fails the new test-case.

The ego behavior that is a candidate for a solvability certificate must satisfy the following constraints. Let `ego` be the name of the car, and `nonego()` be true of both the old and the new non-egos. The first constraint says that `ego` does not violate the right-of-way of any non-egos:

```
:- nonego(V), violatedRightOf(ego, V).
```

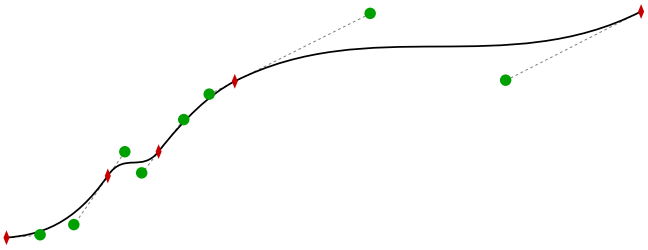
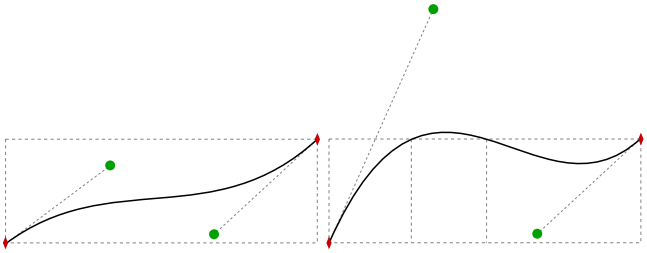


Figure 5: A composite cubic Bezier curve.



(a) Curve bounded between the heights of endpoints. **(b) Curve created a new lane event at a vertical dotted line.**

Figure 6: No-new-lane-event constraint.

The second constraint says that `ego` does not violate any other traffic rules either:

```
:- violatedRule(ego, _).
```

Hence `ego`'s behavior solves the new test-case.

If there are no ASP solutions, the ASP solver returns 'unsatisfiable' and our algorithm stops with failure.

If the ASP program is satisfiable, the ASP solver enumerates all the solutions, which are finitely many. This step passes the ASP solutions one-by-one to the next step to check if each yields a final solution. If a final solution is found (in Step 3), the algorithm either stops and returns the result, or it tries to find more solutions by continuing through the list of ASP solutions, depending on which option the user wants.

3.2.3 Step 3: generating the speed of cars. In this step we determine how fast each car moves along its sequence of poses fixed in Step 1. The speed profile determines the time at which each lane event or stop event happens, which must preserve the order relations fixed in Step 2. This step is again a CSP where the *search variables* are numerical values of the new time symbols, and a time-to-distance-travelled mapping for each car. The speed of a car is the slope of this mapping. The *constraints* include occurrence of stop events, the relations `realLTE()`, `lessThan()`, and `equal()` between time symbols, numerical values of old time symbols, and bounds on longitudinal accelerations. Our algorithm specifies this search problem as a system of polynomial equations and inequalities and solves it with an SMT solver.

A real-valued variable T is assigned for each new time symbol t that appears in `equal()`, `lessThan()`, or a stop event. These are the only new time symbols that are potentially consequential to traffic rules, due to our custom choice rules (in Step 2) for perceptible order.[3] The rest of the new time symbols are for lane events (that were not consequential to the right-of-way violations,) and their numerical value can be determined from their car's time-to-distance-travelled mapping (after it is found) and their distance.

We use piecewise cubic polynomial functions in the form of composite Bezier curves to represent the distance travelled by a car as function of time. The coefficients of the polynomials are in the form of *control points*. The crucial property for our purposes is the *local control property*. That is, changing a control point, affects the function only locally. For example, if an SMT solver has fixed the first few control points to control the speed of a car before entering the intersection, it can change later control points, say for inside the intersection, without invalidating the speed before entrance. *Cubic* Bezier curves have four control points. The first and the last control points are *interpolating* points, meaning that the curve passes through those points. The second and the third points determine the slope of the curve at the first and the last point, respectively. See Figure 6 for examples. A *composite* Bezier curve is a sequence of Bezier curves where the end of each segment is the start of the next segment. See Figure 5 for an example. The red diamonds and green dots indicate the interpolating and intermediate control points of the segments, respectively. See [5] for more on Bezier curves.

The representation of a time-to-distance mapping is as follows. For each pair of time variables T_i, T_{i+1} of a car that are consecutive (with respect to the `realLTE()` order between their time symbols), a cubic Bezier segment is instantiated. The four control points of the segment are (T_i, d_i), $(T_i + \frac{T_{i+1}-T_i}{3}, d_{i,1})$, $(T_i + \frac{2(T_{i+1}-T_i)}{3}, d_{i,2})$, and (T_{i+1}, d_{i+1}). Here $d_{i,1}$ and $d_{i,2}$ are real-valued variables; d_i is the travelled distance for the event corresponding to T_i which is a determined number for a lane event, and a real-valued variable otherwise; and d_{i+1} is defined similarly. Furthermore, the first segment has the control points $(0, 0)$, $(\frac{T_1}{3}, d_{0,1})$, $(\frac{2T_1}{3}, d_{0,2})$, (T_1, d_1); and the last segment has the control points (T_n, d_n), $(T_n + \frac{T_M-T_n}{3}, d_{n,1})$, $(T_n + \frac{2(T_M-T_n)}{3}, d_{n,2})$, (T_M, d_M), where is T_M is the duration of the scenario, and d_M is the total travelled distance of the car as determined in Step 1.

The perceptible order constraint is specified as follows. For a time symbol t, let $v(t)$ be its numerical value if t is for an old event, otherwise let $v(t)$ be its assigned time variable T. Now for any time variable S, if `equal(s,t)` or `equal(t,s)` then we require $|S - v(t)| < m$, if `lessThan(s,t)` then we require $m \leq v(t) - S$, and if `lessThan(t,s)` then we require $m \leq S - v(t)$, where m is the minimum perceptible time difference. As for temporal order, if `realLTE(s,t)` and `not realLTE(t,s)` then we require $S < v(t)$. Similarly if `realLTE(t,s)` and `not realLTE(s,t)` then we require $v(t) < S$.

We must ensure that the Bezier interpolation does not create new lane events consequential to right-of-way. In Figure 6(b), suppose that the right endpoint of the curve represents entering a lane. The height of the curve reaches the height of the right endpoint at the vertical dotted lines which implies that the car enters the lane sooner than intended. We prevent new lane events by requiring that the heights of the intermediate control points be between the

[3] This restriction reduces the number of search variables and constraints, thus simplifies the SMT search problem significantly.

heights of the endpoints.

$$d_i \le d_{i,1} \le d_{i+1}, \quad d_i \le d_{i,2} \le d_{i+1}.$$

Then the convex hull property implies that the whole interpolated Bezier segment stays within its endpoint heights, e.g. as in Figure 6(a).

The longitudinal acceleration constraints are as follows. The second derivative of a cubic polynomial is linear so its extrema over a closed interval are achieved at the endpoints. For an interval $[T_i, T_{i+1}]$, the second derivatives at endpoints are

$$a_i = \frac{6(d_i - 2d_{i,1} + d_{i,2})}{(T_{i+1} - T_i)^2}, \quad a_{i+1} = \frac{6(d_{i,1} - 2d_{i,2} + d_{i+1})}{(T_{i+1} - T_i)^2}.$$

Therefore, if a_m and a_M are respectively the minimum and maximum bounds on longitudinal acceleration, we require

$$a_m \le a_i \le a_M, \quad a_m \le a_{i+1} \le a_M.$$

These inequalities are quadratic since we can remove the fractions by multiplying the inequalities by $(T_{i+1} - T_i)^2$.

Our scenarios do not include collision events, as discussed in §3.1, so the speed of a car, i.e. the slope of the time-distance curve, must be continuous. Each segment of a composite Bezier curve is a smooth function since it is a polynomial. At interpolating points between two consecutive segments, we force the slopes on both sides to be equal with the quadratic equation

$$\frac{d_i - d_{i-1,2}}{T_i - T_{i-1}} = \frac{d_{i,1} - d_i}{T_{i+1} - T_i}.$$

A stop event implies a constraint on the instantaneous speed of the car. If T_i is a time variable for a stop event, we require the instantaneous speed at T_i to be within a threshold $v_{stop} \ge 0$:

$$\frac{d_{i,1} - d_i}{\frac{1}{3}(T_{i+1} - T_i)} \le v_{stop}$$

If a car violates a stop sign, we require the instantaneous speed between arrival at and entrance to the intersection to be at least a threshold v_{run} where $v_{run} > v_{stop}$. It is sufficient to force the slope to not have a local minimum over the interval, and the slopes at the endpoints to be at least v_{run}. For the slope of the function to not have a local minimum over the interval, the second derivative should not change from negative to positive:

$$\neg(a_i < 0 \ \wedge \ a_{i+1} > 0)$$

which is equivalent to the linear inequalities

$$d_i - 2d_{i,1} + d_{i,2} \ge 0 \ \vee \ d_{i,1} - 2d_{i,2} + d_{i+1} \le 0.$$

The lower bound on endpoint slopes are specified with the linear inequalities

$$\frac{d_{i,1} - d_i}{\frac{1}{3}(T_{i+1} - T_i)} \ge v_{run}, \quad \frac{d_{i+1} - d_{i,2}}{\frac{1}{3}(T_{i+1} - T_i)} \ge v_{run}.$$

If the SMT solver returns 'unsatisfiable', Step 3 is repeated with the next ASP solution, if any. If no ASP solution remains, the algorithm returns with no solutions.

If the SMT solver returns a solution, i.e. coordinates of the control points, we can sample points on the time-distance curves using the standard de Casteljau algorithm for Bezier curve computation. This gives the time-to-distance-travelled mapping. Composing this with the travelled distance at each pose, we get the pose at each time.

4 EVALUATION

We demonstrate the capabilities of our approach by generating test-cases for three different situations:

(1) left turn at a multi-lane 4-way stop,
(2) left turn at a T-intersection from the continuing highway to the terminating highway,
(3) left turn at a 3-way uncontrolled Y-intersection.

For each of these static environments, we generate a sequence of increasingly complex test-cases. The videos and logs of the test-cases, also the traffic rules' encoding in Clingo [11], are available on the web[4]. The solvability and complexity certificates are visualized in the videos with green and red egos, respectively.

CARLA has several built-in autopilot software that have access to the full simulation state. We subject two specific autopilot agents to the test suite which are automatically generated by our algorithm. We observe that autopilot can fail as we increase the complexity of the test-cases. Finally, we test CARLA's autopilot when Intel's Responsibility Sensitive Safety (RSS) is added to safeguard autopilot's actions. We observe fewer failures when RSS is enabled, but still autopilot-plus-RSS doesn't pass all the test-cases.

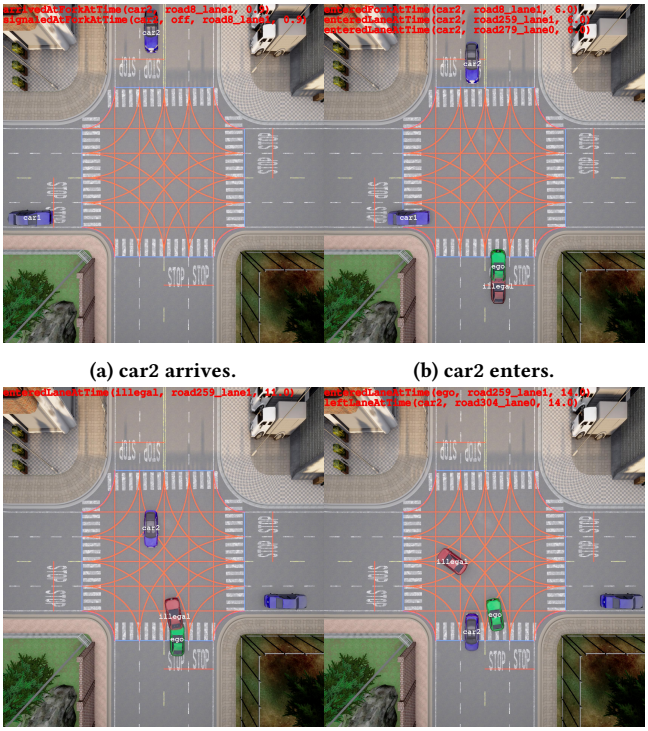
Our straightforward attempt using Scenic's probabilistic programming failed to find solutions to the search problem. The search space is uncountably infinite so there are too many ways to modify a failed sample. On the other hand, the solution space is highly restricted with continuous and discrete constraints and intertwined dependencies, as explained in the previous section. More specifically, traffic rules depend on only a few discrete events, so they partition the search space into big equivalence classes. Probabilistic mutation of trajectories is searching for a target class by walking within big classes and looking for their boundaries. In contrast, our ASP approach is identifying all the target classes by looking from the above, then searching for a sample within each of them.

4.1 Results

We use Scenic [8] as an interface to CARLA to execute a generated test-case, e.g. to test CARLA's autopilot (playing the role of ego).

Four-way Stop: The first example demonstrates making incrementally more complex test-cases by adding more non-egos. The goal of ego is to make a left turn from a four-way stop. Ego starts from the bottom side of the intersection in Figure 7 and must exit the intersection from the left side. We generate three test-cases, each more complex than the previous one. In the first test-case a non-ego is added that enters the intersection from the left and passes straight through the intersection. In the second test-case (visualized in Figure 7), a non-ego is added that approaches from the top, the opposite side of ego, and passes straight through the intersection. In the third test-case, a non-ego is added that approaches also from the left but the other lane of the road, and passes straight through the intersection. Both CARLA's autopilot and autopilot-plus-RSS pass the first test-case. Autopilot fails the second test-case while autopilot-plus-RSS passes it. Finally, both autopilot and autopilot-plus-RSS fail the third test-case. See the videos for even more complex extensions of this test-case.

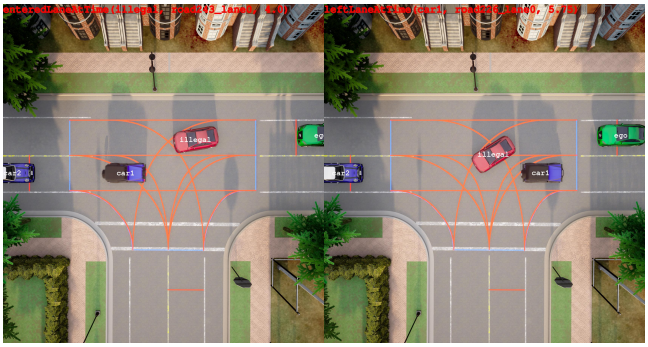
[4]https://www.dropbox.com/sh/e6q4hw98ert5yj7/AAAgo7IlYRKg-rYs7Q9GyMx9a?dl=0



(a) car2 arrives. (b) car2 enters.

(c) illegal enters car2's route. (d) ego enters car2's route.

Figure 7: Left turn from a 4way-stop.



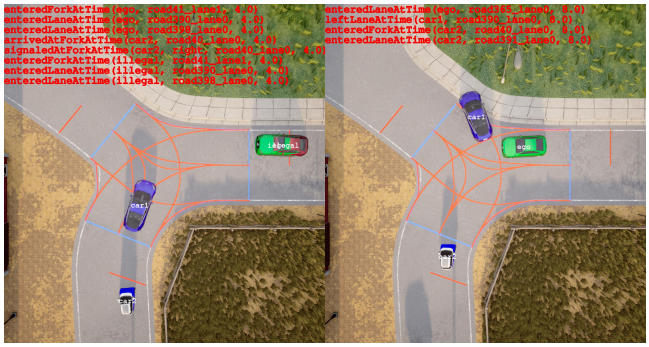
(a) car2 approaches (passes the red line) before illegal enters car2's route

(b) illegal does not yield to car2

Figure 8: Unprotected left turn from continuing highway.

T-intersection: The goal of ego is to perform an unprotected left turn from the continuing highway to the terminating highway of the T-intersection shown in Figure 8. Ego approaches the intersection from the right side and must make a left turn to enter the road on the bottom. This example demonstrates that we can add multiple non-egos to a test-case simultaneously. Starting with an empty test-case, our test-case generator adds multiple non-egos that pass straight through the intersection from left to right. CARLA's autopilot fails this test-case but autopilot-plus-RSS passes it.

Y-intersection: The third example demonstrates that we can add a non-ego whose route does not conflict ego's route, as long as we add also a non-ego whose route has a conflict so that the test-case can become more complex. Autopilot fails this test-case. Running Autopilot-plus-RSS results in a segmentation fault which seems to be due to a bug in CARLA's map or integration of RSS. Ego's goal is to approach from right and make a left turn to the bottom of the intersection. The conflicting non-ego, `car1`, approaches from the bottom, makes a left turn and exits from the top left corner of the intersection. The non-conflicting non-ego, `car2`, also approaches from the bottom, but makes a right turn and exits from the right side of the intersection.



(a) car1 enters earlier than ego and illegal.

(b) illegal already exited from the bottom but ego waited for car1.

Figure 9: Unprotected left turn from an uncontrolled Y-intersection.

5 CONCLUSIONS

In this paper we proposed a formal definition of test-case *complexity* and an algorithm for generating test-case scenarios that progressively become more complex. In particular, our algorithm can use traffic rules (modeled in ASP) to generate a more complex test-case. We use Scenic for executing a test-case in a CARLA simulation. We demonstrated the applicability of our approach by generating test-cases for three different types of intersections with different topologies and vehicle behaviors.

In this work, the order of events leading to a violation was solved by ASP, using the traffic rules and symbolic constants for event timings. Then, using an SMT solver, the symbolic times were instantiated with real numbers, and smooth speeds were generated using Bezier parametrization of their time-distance functions.

ACKNOWLEDGMENT

We thank the anonymous reviewers who provided valuable feedback on earlier drafts of this paper. The authors would like to acknowledge the support of the Air Force Office of Scientific Research under award number FA9550-19-1-0288, National Science Foundation (NSF) under grant numbers CNS 1935724, 2038960, and Amazon Research Award. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Air Force, National Science Foundation, or Amazon.

REFERENCES

[1] Maria-Virginia Aponte, Guillaume Levieux, and Stephane Natkin. 2011. Measuring the level of difficulty in single player video games. *Entertainment Computing* 2, 4 (2011), 205–213.

[2] Gerrit Bagschik, Till Menzel, and Markus Maurer. 2018. Ontology based scene creation for the development of automated vehicles. In *2018 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 1813–1820.

[3] California law. [n. d.]. Rules of the road. http://leginfo.legislature.ca.gov/faces/codes_displayexpandedbranch.xhtml?tocCode=VEH&division=11.&title=&part=&chapter=&article=. Accessed: 2021-01-28.

[4] Alessandro Calò, Paolo Arcaini, Shaukat Ali, Florian Hauer, and Fuyuki Ishikawa. 2020. Generating avoidable collision scenarios for testing autonomous driving systems. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 375–386.

[5] Gerald E Farin. 2002. *Curves and surfaces for CAGD: a practical guide*. Morgan Kaufmann.

[6] Shuo Feng, Yiheng Feng, Haowei Sun, Shan Bao, Yi Zhang, and Henry X Liu. 2020. Testing scenario library generation for connected and automated vehicles, part II: Case studies. *IEEE Transactions on Intelligent Transportation Systems* (2020).

[7] Shuo Feng, Yiheng Feng, Chunhui Yu, Yi Zhang, and Henry X Liu. 2020. Testing scenario library generation for connected and automated vehicles, Part I: Methodology. *IEEE Transactions on Intelligent Transportation Systems* 22, 3 (2020), 1573–1582.

[8] Daniel J. Fremont, Edward Kim, Tommaso Dreossi, Shromona Ghosh, Xiangyu Yue, Alberto L. Sangiovanni-Vincentelli, and Sanjit A. Seshia. 2020. Scenic: A Language for Scenario Specification and Data Generation. arXiv:2010.06580 [cs.PL]

[9] Daniel J Fremont, Edward Kim, Yash Vardhan Pant, Sanjit A Seshia, Atul Acharya, Xantha Bruso, Paul Wells, Steve Lemke, Qiang Lu, and Shalin Mehta. 2020. Formal scenario-based testing of autonomous vehicles: From simulation to the real world. In *2020 IEEE 23rd International Conference on Intelligent Transportation Systems (ITSC)*. IEEE, 1–8.

[10] Feng Gao, Jianli Duan, Yingdong He, and Zilong Wang. 2019. A test scenario automatic generation strategy for intelligent driving systems. *Mathematical Problems in Engineering* 2019 (2019).

[11] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Patrick Lühne, Philipp Obermeier, Max Ostrowski, Javier Romero, Torsten Schaub, Sebastian Schellhorn, and Philipp Wanko. 2018. The potsdam answer set solving collection 5.0. *KI-Künstliche Intelligenz* 32, 2 (2018), 181–182.

[12] Aron Harder, Jaspreet Ranjit, and Madhur Behl. 2021. Scenario2Vector: scenario description language based embeddings for traffic situations. In *Proceedings of the ACM/IEEE 12th International Conference on Cyber-Physical Systems*. 167–176.

[13] Abolfazl Karimi and Parasara Sridhar Duggirala. 2020. Formalizing traffic rules for uncontrolled intersections. In *2020 ACM/IEEE 11th International Conference on Cyber-Physical Systems (ICCPS)*. IEEE, 41–50.

[14] Baekgyu Kim, Takato Masuda, and Shinichi Shiraishi. 2019. Test specification and generation for connected and autonomous vehicle in virtual environments. *ACM Transactions on Cyber-Physical Systems* 4, 1 (2019), 1–26.

[15] Moritz Klischat and Matthias Althoff. 2019. Generating critical test scenarios for automated vehicles with evolutionary algorithms. In *2019 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 2352–2358.

[16] Moritz Klischat and Matthias Althoff. 2020. Synthesizing traffic scenarios from formal specifications for testing automated vehicles. In *2020 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 2065–2072.

[17] Yihao Li, Jianbo Tao, and Franz Wotawa. 2020. Ontology-based test generation for automated and autonomous driving functions. *Information and software technology* 117 (2020), 106200.

[18] Vladimir Lifschitz. 2010. Thirteen definitions of a stable model. In *Fields of logic and computation*. Springer, 488–503.

[19] Michael Genesereth and Eric Kao. 2019. Herbrand Semantics. http://logic.stanford.edu/herbrand/herbrand.html.

[20] Matthew O'Kelly, Aman Sinha, Hongseok Namkoong, John Duchi, and Russ Tedrake. 2018. Scalable end-to-end autonomous vehicle testing via rare-event simulation. *arXiv preprint arXiv:1811.00145* (2018).

[21] Yunlong Qi, Yugong Luo, Keqiang Li, Wei Kong, and Yongsheng Wang. 2019. *A trajectory-based method for scenario analysis and test effort reduction for highly automated vehicle*. Technical Report. SAE Technical Paper.

[22] Stefan Riedmaier, Thomas Ponn, Dieter Ludwig, Bernhard Schick, and Frank Diermeyer. 2020. Survey on scenario-based safety assessment of automated vehicles. *IEEE Access* 8 (2020), 87456–87477.

[23] Cumhur Erkan Tuncali, Georgios Fainekos, Danil Prokhorov, Hisahiro Ito, and James Kapinski. 2019. Requirements-driven test generation for autonomous vehicles with machine learning components. *IEEE Transactions on Intelligent Vehicles* 5, 2 (2019), 265–280.

[24] Cumhur Erkan Tuncali, Theodore P Pavlic, and Georgios Fainekos. 2016. Utilizing S-TaLiRo as an automatic test generation framework for autonomous vehicles. In *2016 IEEE 19th International Conference on Intelligent Transportation Systems (ITSC)*. IEEE, 1470–1475.

[25] Simon Ulbrich, Till Menzel, Andreas Reschka, Fabian Schuldt, and Markus Maurer. 2015. Defining and substantiating the terms scene, situation, and scenario for automated driving. In *2015 IEEE 18th International Conference on Intelligent Transportation Systems*. IEEE, 982–988.

[26] Jiajie Wang, Chi Zhang, Yuehu Liu, and Qilin Zhang. 2018. Traffic sensory data classification by quantifying scenario complexity. In *2018 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 1543–1548.

[27] Qin Xia, Jianli Duan, Feng Gao, Tao Chen, and Cai Yang. 2017. *Automatic generation method of test scenario for ADAS based on complexity*. Technical Report. SAE Technical Paper.

[28] Qin Xia, Jianli Duan, Feng Gao, Qiuxia Hu, and Yingdong He. 2018. Test scenario design for intelligent driving system ensuring coverage and effectiveness. *International Journal of Automotive Technology* 19, 4 (2018), 751–758.

[29] Ziyuan Zhong, Gail Kaiser, and Baishakhi Ray. 2021. Neural Network Guided Evolutionary Fuzzing for Finding Traffic Violations of Autonomous Vehicles. *arXiv preprint arXiv:2109.06126* (2021).

A TRAFFIC RULES

In this section, we list the traffic rules used in our experiments, taken from California Law[5]:

(1) *"The driver of a vehicle approaching an intersection shall yield the right-of-way to any vehicle which has entered the intersection from a different highway." [3, section 21800 (a)]*

(2) *"When two vehicles enter an intersection from different highways at the same time and the intersection is controlled from all directions by stop signs, the driver of the vehicle on the left shall yield the right-of-way to the vehicle on his or her immediate right." [3, section 21800 (c)]*

(3) *"This section does not apply to any of the following: (1) Any intersection controlled by an official traffic control signal or yield right-of-way sign. (2) Any intersection controlled by stop signs from less than all directions. (3) When vehicles are approaching each other from opposite directions and the driver of one of the vehicles intends to make, or is making, a left turn." [3, section 21800 (e)]*

(4) *"The driver of a vehicle intending to turn to the left or to complete a U-turn upon a highway, or to turn left into public or private property, or an alley, shall yield the right-of-way to all vehicles approaching from the opposite direction which are close enough to constitute a hazard at any time during the turning movement, and shall continue to yield the right-of-way to the approaching vehicles until the left turn or U-turn can be made with reasonable safety." [3, section 21801 (a)]*

(5) *"A driver having yielded as prescribed in subdivision (a), and having given a signal when and as required by this code, may turn left or complete a U-turn, and the drivers of vehicles approaching the intersection or the entrance to the property or alley from the opposite direction shall yield the right-of-way to the turning vehicle." [3, section 21801 (b)]*

(6) *"The driver of any vehicle approaching a stop sign at the entrance to, or within, an intersection shall stop at a limit line, if marked, otherwise before entering the crosswalk on the near side of the intersection." [3, section 22450 (a)]*

(7) *"The driver of any vehicle approaching a stop sign at the entrance to, or within, an intersection shall stop as*

[5] sections 21800-21802 from Vehicle Code (VEH), Division 11 (Rules of the road), Chapter 4 (Right-of-Way)

required by Section 22450. The driver shall then yield the right-of-way to any vehicles which have approached from another highway, or which are approaching so closely as to constitute an immediate hazard, and shall continue to yield the right-of-way to those vehicles until he or she can proceed with reasonable safety." [3, section 21802 (a)]

(8) *"A driver having yielded as prescribed in subdivision (a) may proceed to enter the intersection, and the drivers of all other approaching vehicles shall yield the right-of-way to the vehicle entering or crossing the intersection." [3, section 21802 (b)]*

(9) *"This section does not apply where stop signs are erected upon all approaches to an intersection." [3, section 21802 (c)]*