Boundary State Generation for Testing and Improvement of Autonomous Driving Systems

Matteo Biagiola
Università della Svizzera italiana
Lugano, Switzerland
matteo.biagiola@usi.ch

Paolo Tonella
Università della Svizzera italiana
Lugano, Switzerland
paolo.tonella@usi.ch

***Abstract*—Recent advances in Deep Neural Networks (DNNs) and sensor technologies are enabling autonomous driving systems (ADSs) with an ever-increasing level of autonomy. However, assessing their dependability remains a critical concern. State-of-the-art ADS testing approaches modify the controllable attributes of a simulated driving environment until the ADS misbehaves. Such approaches have two main drawbacks: (1) modifications to the simulated environment might not be easily transferable to the in-field test setting (e.g., changing the road shape); (2) environment instances in which the ADS is successful are discarded, despite the possibility that they could contain hidden driving conditions in which the ADS may misbehave.**

In this paper, we present GENBO (GENerator of BOundary state pairs), a novel test generator for ADS testing. GENBO mutates the driving conditions of the ego vehicle (position, velocity and orientation), collected in a failure-free environment instance, and efficiently generates challenging driving conditions at the behavior boundary (i.e., where the model starts to misbehave) in the same environment. We use such boundary conditions to augment the initial training dataset and retrain the DNN model under test. Our evaluation results show that the retrained model has up to 16× higher success rate on a separate set of evaluation tracks with respect to the original DNN model.

***Index Terms*—Software Testing, Autonomous Driving**

I. INTRODUCTION

While the dream of fully-autonomous vehicles (the so-called *Level 5*, as defined by the Society of Automotive Engineers (SAE) [1]) is still to come, there exist deployed systems that exhibit impressive levels of automation in the driving task (e.g., the Tesla's autopilot is considered an advanced Level 2 system [2]). Hence, testing of autonomous driving systems (ADSs) has become a critical research topic, of vital importance for today's and for future ADSs.

The literature on testing ADS systems is quite rich. A recent survey on ADS testing [3] analyzed 181 papers published in peer-reviewed venues between June 2015 and June 2022. Most of the papers target vision-based ADSs that use Deep Neural Networks (DNNs) to process the input images and decide the actions to take. Such systems are typically trained in a supervised fashion using a dataset of labeled images, where the labels determine the desired actions (e.g., steering angle and throttle). DNN vision-based ADSs are tested offline and/or online [4, 5, 6]. Online testing usually involves a simulator (hence, it is often called simulation-based testing), where the DNN model drives the vehicle with the objective of keeping it within the driving lane [7, 8, 9, 10, 11, 12].

Existing simulation-based testing approaches modify the driving environment in which the DNN model operates. A test case is a set of values representing all (or a subset of) the controllable attributes of the environment. Such attributes determine the environment in which the ADS operates. In particular, a test case can be a road network [8], a sequence of waypoints determining the track shape [9, 11, 12] or a mixture of dynamic and static elements of the simulation [13, 14, 15, 16]. Examples of static elements that do not change during the simulation, are the track shape, or the positions of the buildings and trees; dynamic elements are the weather condition and the dynamics, including position and velocity, of the other actors in the environment (e.g., pedestrians and other vehicles).

Modifications to the simulated environment are not always easy to transfer from simulation to the in-field testing setting. In fact, after simulation testing and before deployment, an ADS is typically tested in the field on a limited number of private test tracks [17, 18, 19, 20]. Changing the environment in such setting is expensive (e.g., building new testing tracks) and, sometimes, impossible (e.g., changing the weather condition). Another drawback of current test generators is that they generate new environment scenarios until the ADS under test misbehaves. Nonetheless, even in *failure-free* environment scenarios there might exist hidden driving conditions that are left unexplored and would potentially represent further opportunities to challenge the ADS. This calls for new ADS testing approaches able to generate challenging driving conditions within resource-constrained testing settings, where only a limited number of environment configurations (e.g., driving tracks) are available.

We propose GENBO (**GEN**erator of **BO**undary state pairs), a novel test generator for online ADS testing. GENBO works by mutating the driving conditions of the ego vehicle the DNN model is controlling, i.e., position, velocity and orientation (a *state* hereafter), within a fixed environment scenario (including the track shape, the weather condition, etc.). GENBO makes full use of an existing, failure-free environment instance, by uncovering challenging driving conditions typically neglected by state-of-the-art approaches. In particular, GENBO employs a novel search algorithm that evolves pairs of states in order to find *boundary state pairs*, i.e., pairs of states that are *close* to each other and that expose different (successful vs failing) behaviors of the ADS [9, 21, 22]. GENBO uses mutation

operators to generate a sequence of state pairs that crosses the behavioral boundary and then applies binary search, which executes a logarithmic number of driving simulations, to efficiently reach the target boundary state pairs.

Among the boundary state pairs, we are interested in the *recoverable* ones, i.e., those where an expert pilot can avoid the failure the ADS exhibits. GENBO uses such pairs to collect a dataset of challenging driving conditions, automatically labeled by an autopilot with global knowledge of the driving scenario (that we use as a proxy for the expert pilot). Our hypothesis is that by augmenting the original training dataset with such challenging driving scenarios we can increase the generalization capabilities of the ADS to related, but unseen, driving tracks, which contain similar boundary conditions.

We applied our approach to test the lane-keeping functionality of the Dave-2 model [23], a test subject widely used in the related literature on ADS testing [7, 9, 10, 11, 20, 24, 25, 26, 27, 28]. In our empirical evaluation, we tested driving models from different stages of training, and we show that GENBO can find boundary state pairs even for well-trained models. Our results also show a very strong discriminative capability of boundary state pairs, as boundary state pairs of well-trained models are significantly more challenging than those of poorly trained models. Moreover, we retrained a well-trained driving model after augmenting the original training dataset with labeled examples collected from its boundary state pairs. Results show that the success rate of the retrained model on a set of evaluation tracks is up to 16× higher with respect to the original model.

Our paper makes the following contributions:

Technique. A novel approach, implemented in the publicly available tool GENBO, which exploits a limited number of failure-free environment instances to generate challenging driving conditions for the driving model under test.

Evaluation. An empirical study showing that GENBO exposes challenging driving conditions even for well-trained driving models and that such conditions are useful to significantly improve the model.

II. APPROACH

Figure 2 shows an overview of our approach, which takes as input an ADS and performs two steps: step ❶ executes a search algorithm that extracts boundary state pairs from the driving model of the ADS, i.e., its decision-making component. In particular, the search algorithm evolves a pair of nearby states characterizing the driving conditions of the vehicle on a given environment scenario. In the initial pair of states $\langle s_1, s_2 \rangle$ (also called seed pair), the driving model succeeds, i.e., when it starts to drive the vehicle either in state s_1 or s_2, it is able to successfully drive it along the driving track of the given scenario (i.e., none of the two starting states causes a failure). The objective of the search algorithm is to find a pair of *boundary* states $\langle \hat{s}_1, \hat{s}_2 \rangle$, such that in one state (say $\hat{s}_1$) the driving model succeeds and in the other ($\hat{s}_2$) it fails. Once the search finds a boundary state pair, it samples a new seed until there is search budget, to find multiple boundary state pairs for the given driving model.

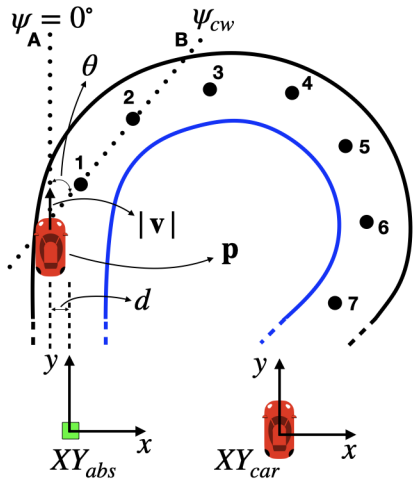

Fig. 1. Visualization of a vehicle state

Boundary state pairs identify challenging driving conditions for the driving model in a given environment scenario, which can help to improve its capabilities on other scenarios where similar challenging conditions may occur. In the second step ❷ of our approach, we place an autopilot with global knowledge of the given environment scenario on the boundary state pairs of the driving model. The autopilot automatically labels the new camera images with the ground-truth steering angle values and outputs a labeled dataset. We use such dataset to retrain and improve the driving model of the ADS.

A. Boundary State

The state of a vehicle driving along a track within an environment scenario is a subset of variables characterizing its motion. In particular, we define a vehicle state as follows:

*Definition 1 (**State**):* A state s of a vehicle in a track T is a tuple $\langle \mathbf{p}, \psi, v \rangle$ where:

1) $\mathbf{p}$ is the $\langle x, y \rangle$ absolute position of the vehicle inside T;
2) ψ is the orientation of the vehicle;
3) $v = |\mathbf{v}|$ is the magnitude of the velocity vector $\mathbf{v}$ of the vehicle.

Figure 1 shows an example of a vehicle state. The black dots in the figure indicate the waypoints of the track, while the green square indicates the origin of the track. The waypoints are evenly spaced along the track and they are placed at the center. There are two reference systems, i.e., the absolute reference system XY_{abs} and the reference system of the vehicle XY_{car}. All the quantities that follow are defined w.r.t. XY_{abs}, except for the components of the velocity that are defined w.r.t. XY_{car}.

The vehicle has position $\mathbf{p}$, where the vector $\mathbf{p}$ is centered at the center of the mass of the vehicle. It has a magnitude velocity of $v = |\mathbf{v}|$ and orientation ψ. The orientation is defined w.r.t. the y axis of XY_{abs} whose orientation is 0°.

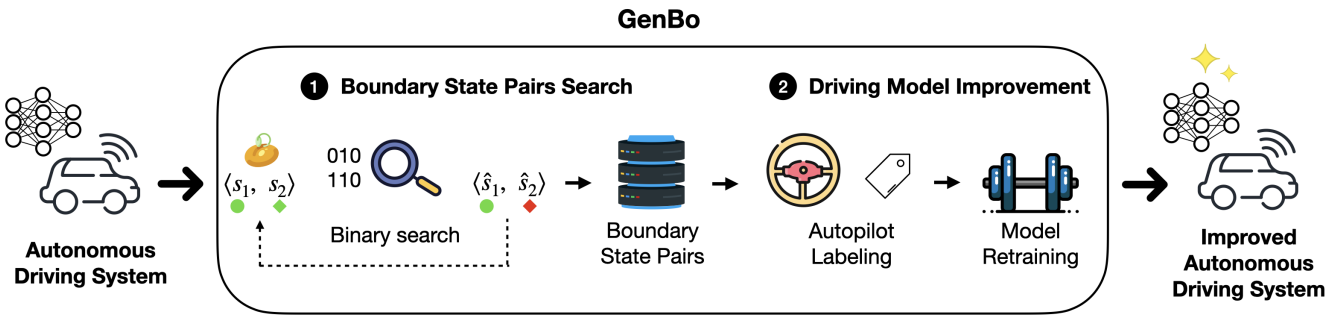

Fig. 2. Overview of our approach.

Given the state of a vehicle, we define a function $d(\mathbf{p}, T)$ that measures the distance d of the vehicle from the center of the driving lane of track T (also called cross-track error or XTE [20]). It is also convenient to define a function $\theta(\mathbf{p}, \psi, T)$ that measures the angle θ between the orientation of the vehicle ψ and the orientation of the closest waypoint ψ_{cw}. In the figure, the line crossing the vehicle is parallel to the y axis of XY_{abs}, i.e., $\psi = 0°$ (see the dotted line "A"). The next closest waypoint to the vehicle is waypoint "1" and its orientation is determined by the line going through the two successive waypoints (see the dotted line "B"). The angle between the orientation of the vehicle and the orientation of the closest waypoint is θ, the relative orientation of the vehicle.

*Definition 2 (**State Validity**):* A state $s = \langle \mathbf{p}, \psi, v \rangle$ of a vehicle in a track T is valid iff:

1) $d(\mathbf{p}, T) \leq \frac{W}{2}$, where W is the lane width of track T;
2) $v \leq v_{max}$;
3) $\theta(\mathbf{p}, \psi, T) \leq \theta_{max}$;

The first condition predicates that the center of the mass of the vehicle must stay within the bounding box of the track (i.e., the black and blue lines in Figure 1). The second and third validity conditions regard velocity and relative orientation. In both cases, such quantities must be less than a maximum value, to be configured based on the mechanical properties of the autonomous vehicle being simulated.

*Definition 3 (**Boundary State Pair**):* Given a driving model M and a track T, a boundary state pair is a pair of valid states $\langle s_i, s_j \rangle$ such that:

1) the driving model M, when placed in state s_i, *succeeds* at the task of driving the vehicle along track T;
2) on the other hand, the driving model M, when placed in state s_j, *fails* at the task of driving the vehicle along track T;
3) the two states of the pair, i.e., s_i and s_j, are distinct but close to each other, i.e., $s_i \neq s_j$ and $dist(s_i, s_j) \leq \boldsymbol{\epsilon}$.

This definition introduces the operation of *placing* the driving model, and hence the vehicle, in a state s. For instance, if one of the two states in a boundary state pair is $s = \langle \langle 10, 0.5 \rangle, 0, 25 \rangle$, the simulator would place the vehicle in position $\mathbf{p} = \langle 10, 0.5 \rangle$, with an orientation of $\psi = 0°$ w.r.t. XY_{abs} and an initial velocity $v = 25\,km/h$.

In each of the two states of a boundary state pair, we check if model M is able to keep the vehicle in lane for a non-negligible amount of time $t \geq t_{min}$. For instance, if the driving simulation is set at 20 *fps* and $t_{min} = 12.5\,s$, succeeding at the lane-keeping task means that the driving model M is able to keep the vehicle in lane for at least 250 simulation steps. In this case, we say that the state s is a *recoverable* state for the driving model M. On the other hand, failing at the lane-keeping task, means that the vehicle goes out-of-bound before reaching the time threshold. In this case, we say that the state s is a *non-recoverable* state for the driving model M. A boundary state pair consists of one recoverable and one non-recoverable state. In Figure 2, we indicate recoverable states with a green circle/diamond and non-recoverable states with a red circle/diamond.

The last condition for two states to form a boundary state pair is *closeness*:

$$dist(s_i, s_j) \leq \boldsymbol{\epsilon} \iff \begin{cases} \|\mathbf{p}_i - \mathbf{p}_j\| \leq \epsilon_p \\ |v_i - v_j| \leq \epsilon_v \\ |\psi_i - \psi_j|_{360°} \leq \epsilon_\psi \end{cases}$$

where $\boldsymbol{\epsilon} = \langle \epsilon_p, \epsilon_v, \epsilon_\psi \rangle$. The first component of the distance is the Euclidean distance between the two 2D position vectors of the two states. The second distance is the difference in absolute value between the magnitude velocity vectors of the two states. The third distance is the difference between two angles measured in degrees. To ensure that the difference stays within the interval $[0, 360]$, we take the modulo on 360° (we use the notation $|\cdot|_{360°}$ to indicate this). The epsilon values, defining how close two boundary states should be, are configured depending on the properties of the track and of the autonomous vehicle being simulated.

B. Boundary State Pairs Search

Algorithm 1 shows the high level steps of the search algorithm. The algorithm evolves one individual at a time, where the individual is a pair of states.

Algorithm 1: Pseudocode of GENBO

```
   Input : M, Driving model under test;
           T, Track to drive;
           S, Simulator instance;
           Tr, Reference trace;
           R, Number of restarts;
           N, Number of iterations;
           L, Length of the sequence when mutating a seed state.
   Output: A, archive of boundary states.
1  A ← ∅
2  for num_restarts = 1 to R do
3      /* Initialize seed state */
4      s1 ← SAMPLEVALIDSTATE(Tr)
5      repeat
6          s2, valid ← s1.MUTATE()
7      until valid
8      b ← ⟨s1, s2⟩
9      success1, success2 ← EXECUTE(b, S, T, M)
10     if success1 ⊗ success2 ∧ b ∉ A then
11         A ← A ∪ b                    ▷ Store collateral boundary state
12     end
13     if ¬success1 ∨ ¬success2 then
14         continue                     ▷ Restart if any execution failed
15     end
16     /* Evolve seed state */
17     pairs ← [b]
18     num_iterations ← 1
19     while num_iterations ≤ N do
20         for seq_length = 1 to L do
21             b̂, valid ← GETLAST(pairs).MUTATE()
22             if ¬ valid then
23                 break
24             end
25             pairs.APPEND(b̂)
26         end
27         idx, it ← BINARYSEARCH(pairs, S, T, M)
28         num_iterations ← num_iterations + it
29         if idx > 0 ∧ pairs[idx] ∉ A then
30             A ← A ∪ pairs[idx]
31             break
32         end
33     end
34 end
35 return A
```

The main parameters of the algorithm are the number of iterations N, the number of restarts R and the sequence length L. The number of restarts R is the number of seed states that the algorithm samples and evolves. The number of iterations N corresponds to the maximum number of executions of each individual for each restart. The sequence length L is the maximum number of mutations of each individual in each iteration. The output of the search algorithm is a set of boundary state pairs stored in the archive A.

The search algorithm consists of two main phases, i.e., the initialization of a seed state (lines 4–15) and its evolution (lines 17–33). In the first phase, the algorithm starts by sampling a state from a reference trace Tr, by calling the SAMPLEVALIDSTATE function at line 5. The function also has a state validity filter, to ensure that the initial states are valid. A reference trace is a trajectory of states that we collect by running an autopilot with global knowledge of the track and of the state of the vehicle. The autopilot drives by following nominal trajectories i.e., it drives the vehicle at the center of the lane along the waypoints of the track. Alternatively, if the reference trace is not available, seed states can be sampled at random or can be obtained from a driving model M, with a downstream validity check that filters out invalid states.

The loop at lines 5–7 mutates the seed state s_1 until the resulting state s_2 is valid. As described below in Section II-B1, the mutation operator ensures closeness of the boundary states by construction. At line 8, the algorithm builds the individual b consisting of the two states s_1 and s_2. The EXECUTE function at line 9 places the vehicle in the two states of b, executes the two simulations, and collects the results. In particular, it returns two boolean values, $success_1$ and $success_2$, indicating whether the driving model M succeeds at the lane-keeping task. The **if** statement at lines 10–12, checks if the boolean values of $success_1$, $success_2$ have opposite values (XOR logical operator, indicated as $\otimes$), which indicates a boundary state pair that is stored into the archive A (line 11) after checking for duplicates (i.e., $b \notin A$). When either of the two executions, or both, fail, we restart the search (lines 13–15), while the evolution loop (lines 17–33) is performed only when in both states of the pair b the driving model succeeds.

The evolution phase starts by creating a list of pairs (line 17), initialized with the seed, and by assigning the counter variable for the following while loop (line 18). Such loop (lines 19–33) iteratively mutates the last pair in *pairs* for L times. The MUTATE function at line 21 outputs a new individual for every execution (i.e., $\hat{b}$), mutating both states. The function also returns a boolean variable (i.e., *valid*), indicating whether it was possible to obtain a valid state pair by mutation. If at any point during the sequence an invalid pair is generated, the algorithm breaks the **for** loop (**if** statement at lines 22-24) and goes directly to line 27. The latter performs a binary search operation on the *pairs* list to find a boundary state pair. The function BINARYSEARCH executes the last individual and if in both its states the driving model fails, it chooses an individual in the middle of the list. Then, it proceeds recursively with the head or the tail of the list depending on whether the middle individual consists of two states where the driving model fails or succeeds, respectively. When in one state the driving model fails and in the other it succeeds, a boundary state pair is found. The function BINARYSEARCH returns the number of executions of individuals into the variable *it*. Moreover, it returns a positive index into the variable *idx* if it found a boundary state pair. In such a case, the algorithm adds the corresponding boundary state pair (*pairs*[*idx*]) to the archive after checking for duplicates (**if** statement at lines 29–32) and restarts the search. If no boundary state is found, the **for** loop at lines 19–33 restarts by mutating the last individual of the *pairs* list until the budget of N iterations expires.

1) State Mutation: In Algorithm 1 there are two calls at the MUTATE function. The first one (line 6), mutates a single state into a new one, while the second one (line 21) mutates a pair of states. Both ensure closeness of the two resulting states.

*Definition 4 (**State Mutation Function**):* Given two valid states $s_i, s_j \in \mathbb{S}$ that satisfy closeness (i.e., $dist(s_i, s_j) \leq \epsilon$), a mutation is a function $\mu : \mathbb{S} \times \mathbb{S} \rightarrow \mathbb{S}$ that modifies the state

$s_i = \langle \mathbf{p}, \psi, v \rangle$ such that the resulting state $\hat{s}_i = \mu(s_i, s_j) = \langle \hat{\mathbf{p}}, \hat{\psi}, \hat{v} \rangle$ has the following properties:

1) $d(\hat{\mathbf{p}}, T) \geq d(\mathbf{p}, T)$;
2) $\hat{v} \geq v$;
3) $|\theta(\hat{\mathbf{p}}, \psi, T)| \geq |\theta(\mathbf{p}, \psi, T)|$;
4) at least one of the three inequalities 1), 2), 3) is strict;
5) $dist(\hat{s}_i,\ s_j) \leq \boldsymbol{\epsilon}$;
6) $\hat{s}_i$ is valid.

The mutation function μ changes the position $\mathbf{p}$, the velocity v or the orientation ψ of the given state s_i, such that the resulting state has higher values, in absolute terms, of d, v or $|\theta|$, while preserving closeness and validity. Changes to the velocity magnitude can be made directly on the given state, while in order to change the distance from the center d and the relative orientation θ, the mutation function μ needs to change the position $\mathbf{p}$ and/or the orientation ψ of the vehicle.

In the following, we describe the mutation function for each component of the state, i.e., position, orientation and velocity. After mutating one of these three components, chosen at random, a second and then a third mutation is applied to each of the two remaining components with probability 0.3.

Mutate position. The position vector $\mathbf{p}$ of the state s_i being mutated has two components, i.e., p_x and p_y. The position mutation operator changes either of the two components, chosen with equal probability (e.g., p_x). In order to respect the closeness condition, $\hat{p}_x$ is chosen in the interval $(p_x - \epsilon_p, p_x + \epsilon_p)$. Then, we determine the other component of $\hat{\mathbf{p}}$ (e.g., $\hat{p}_y$) by satisfying the following Euclidean distance equation:

$$\sqrt{(\hat{p}_x - q_x)^2 + (\hat{p}_y - q_y)^2} \leq \epsilon_p \tag{1}$$

where q_x and q_y are the X and Y components of the position vector of the other state in the boundary pair, s_j. To make sure that the mutation increases the current XTE d, we randomly sample position components from their respective ranges and compute the XTE $\hat{d}$ for each resulting position vector $\hat{\mathbf{p}}$. We use a finite budget to determine whether it is possible to obtain a higher XTE, while also checking that the resulting state $\hat{s}_i$ is still valid. If the outcome of both checks is positive, we deem the mutation successful and change the position components of s_i accordingly.

Mutate orientation. Given the orientation ψ_i of state s_i, the orientation mutation operator applies a mutation such that the closeness and validity constraints hold.

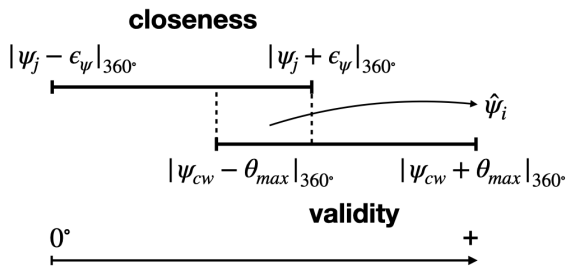

Fig. 3. Orientation constraints overlap.

The closeness constraint regards the orientation of the other state of the pair, i.e., ψ_j, limiting the new orientation $\hat{\psi}_i$ to be ϵ_ψ-close to ψ_j. On the other hand, the validity constraint on the mutated orientation $\hat{\psi}_i$ depends on the orientation of the closest waypoint ψ_{cw}, whose angle w.r.t. the new orientation must not be greater than θ_{max}. In Figure 3 there is overlap between the two ranges, hence we choose the new orientation $\hat{\psi}_i$ in the interval determined by the higher of the two interval lower bounds (i.e., $|\psi_{cw} - \theta_{max}|_{360°}$ for the example in Figure 3) and the lower of the two interval upper bounds (i.e., $|\psi_j + \epsilon_\psi|_{360°}$ for the example in Figure 3).

In computing the intersection between closeness and validity intervals, we have to carefully consider that angles are defined modulo 360°. For instance, with $\epsilon_\psi = 7.2$, (i.e., $360 \cdot 2\%$) $\theta_{max} = 20°$, $\psi_j = 350°$ and $\psi_{cw} = 5°$, the resulting overlap range would be $[355°, 35°]$, and we would deem that there is no overlap as the upper bound is lower than the lower bound. To properly handle such cases, we have to represent the overlap range as $[355°, 360°] \cup [0°, 35°]$. Only by doing this we correctly find that in such case an overlap between validity and closeness ranges does exist, being the range $[355°, 357.2°]$.

When it is not possible to determine an overlap range, the mutation operator fails. We use a finite budget to randomly sample orientation values within the overlap range until the new relative orientation $\hat{\theta}$ is greater than the current relative orientation θ. If the search finds such a value, we replace the orientation ψ_i with new orientation $\hat{\psi}_i$ as it is respects all the constraints.

Mutate velocity. The velocity mutation operator mutates the magnitude of the velocity vector. In particular the mutation needs to satisfy the following conjunctions of constraints:

$$|v_i - v_j| \leq \epsilon_v \wedge v_i \leq v_{max} \wedge \hat{v}_i > v_i \tag{2}$$

where the first constraint represents closeness, the second validity, and the third the increased velocity requirement. The constraint system is solvable if $v_i < v_{max}$; we randomly choose a solution that satisfies all the constraints, $\hat{v}_i$, which is the mutated magnitude velocity. We then compute the components of the velocity vector $\mathbf{v}$ on the reference system of the vehicle XY_{car}. We set $v_x = 0$, $v_y = \hat{v}_i$, since only the vertical component of the velocity is non-zero in the coordinate system of the vehicle. To compute the absolute velocity vector on the reference system XY_{abs} we carry out a rotation of axis using the orientation of the vehicle ψ [29].

2) Binary Search: In Algorithm 1 at line 27, the search algorithm calls the binary search function on the sequence of mutations *pairs* generated from the seed b. Figure 4 illustrates the process on a sequence of $L = 4$ mutations. Applying consecutive increasing mutations (i.e., mutations that increase either d or v or θ) to a starting pair of states makes the pair increasingly more challenging for the driving model. Depending on the length of the sequence, the resulting pairs will move away from the initial seed and potentially far from the boundary. The idea is to apply a binary search to look for

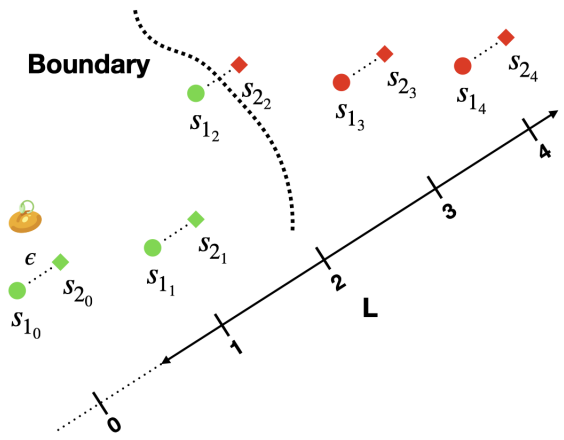

Fig. 4. Binary search function to select boundary states.

the pair that sits in the boundary, given that the pairs in the sequence are sorted by their difficulty [1].

In Figure 4, the first state ($idx = 0$) is the seed, where in both states the model M is successful by construction. The last state ($idx = 4$) is the most challenging and the driving model fails in both states. Then, the binary search function executes the pair at index $idx = 2$. In this example, the model succeeds in $\hat{s}_1$ and fails in $\hat{s}_2$. The two states in the pair are close and valid by construction, forming a boundary state pair.

3) Individual-level Mutation: In the previous sections we described how to mutate a state, but Algorithm 1 evolves individuals that are state pairs, rather than single states. Given a state pair $b = \langle s_1, s_2 \rangle$, the individual-level mutation function starts by mutating s_2, producing a new valid state $\hat{s}_2$ close to s_1. Since the pair $\langle s_1, s_2 \rangle$ consists by construction of a less challenging state s_1 and a more challenging state s_2 (see Algorithm 1), during the evolution of an individual we want to preserve such a relation between the elements of the pair, which makes it easier to cross the failure boundary with the most challenging state ($\hat{s}_2$) while keeping the other state ($\hat{s}_1$) within the boundary (see Figure 4).

Hence, we compute the difference operator $\delta_s = \hat{s}_2 - s_2 = \langle \boldsymbol{\delta}_p, \delta_\psi, \delta_v \rangle$ and apply it to $s_1 = \langle \mathbf{p}_1, \psi_1, v_1 \rangle$, obtaining $\hat{s}_1 = \langle \mathbf{p}_1 + \boldsymbol{\delta}_p, \psi_1 + \delta_\psi, v_1 + \delta_v \rangle$. Then, we check whether $\hat{s}_1$ is valid. In particular, the position is the most critical parameter since it also affects the relative orientation. If $\hat{s}_1$ is not valid, e.g., because adding the deltas resulted in a position that is out of the lane or the orientation is invalid, the individual-level mutation operator discards both states $\hat{s}_2$ and $\hat{s}_1$, and re-attempts to mutate the pair until either both resulting states are valid or the budget expires.

C. Driving Model Improvement

The second step of our approach (i.e., step ❷) consists of improving the driving model of the ADS. We first place the autopilot on the non-recoverable states of the boundary state pairs of the driving model. The autopilot automatically labels each frame of the simulation with the right steering angle, as it can avoid the failures that affect the ADS under test.

In the next phase, we retrain the driving model from scratch by using the union of the initial dataset used for training the model, and the labeled dataset resulting from the placing the autopilot on the boundary state pairs of the driving model.

[1]In the presence of non-monotonic behaviors, the algorithm always returns a boundary state pair, although it might not be the most challenging one.

III. Empirical Evaluation

To assess the existence and the practical benefits of boundary state pairs, we consider the following research questions:
RQ_1 (Existence): *Do boundary state pairs exist in the training track for well-behaving driving models?*

In our analysis we are interested in evaluating *well-behaving* driving models, i.e., models that, in nominal conditions, drive well in the driving scenario they have been trained on (in our evaluation, a closed driving track with asphalt roads, surrounded by green grass and sunny weather [9, 11, 12]). For nominal conditions, we mean that the vehicle is placed in the starting position of the driving track, with zero velocity and oriented in the direction of the road. We say that a driving model drives well in nominal conditions if it is able to keep the lane for 1200 simulation steps (which correspond to two consecutive laps and 1 minute of driving at 20 *fps*).

RQ_1 aims at assessing whether such models exhibit boundary state pairs. A positive answer would imply that challenging driving scenarios exist even in a failure-free track where the driving model is well-behaving. Therefore, testing driving models does not necessarily require the manipulation of the environment (e.g., by generating of new driving tracks).
Metrics. To assess the existence of boundary state pairs given a well-behaving driving model, we simply measure the number of boundary state pairs resulting from the search process.
RQ_2 (Comparison): *How do boundary state pairs of driving models of different qualities compare with each other? Do boundary state pairs discriminate between them?*

In RQ_2, we compare boundary state pairs of driving models with different degrees of performance. All the driving models respect the requirement of being able to complete the training track in nominal conditions (i.e., they are well-behaving) but they differ in terms of the amount of time they have been trained (i.e., they have different validation losses). We aim to study the extent to which boundary state pairs discriminate between high-quality and low-quality driving models. We expect boundary state pairs of high-quality driving models to be more challenging and extreme than those of low-quality driving models.
Metrics. We use two metrics for comparing boundary state pairs. Such metrics are system-level since the search process executes the driving model within the system (i.e., the vehicle driving along a track).

The first system-level metric we consider is *recoverability*, that measures to which extent the driving model M_A is able to recover from the boundary state pairs of the driving model M_B. For instance, let us suppose that the driving model M_B has N boundary state pairs in the training track. We

measure the recovery percentage of the driving model M_A in each of them, as the number of times M_A succeeds when placed in a certain state, divided by the number of runs the model is executed in each state (to take into account the randomness of the simulation [30]). The recoverability of M_A on the boundary state pairs of M_B is the average recovery percentage across all N boundary state pairs. We consider the recoverability of M_A on both the recoverable and non-recoverable states of M_B. Hence, recoverability is a measure of how challenging the boundary state pairs of a certain driving model are for another driving model.

The second system-level metric we consider is the *radius*. We define the radius at the state level. In particular, we consider the non-recoverable state of a boundary state pair, since it is more challenging. Given a state s and the training track T we consider three quantities: (1) the distance from the center $d(\mathbf{p}, T)$, (2) the velocity v, and (3) the relative orientation $\theta(\mathbf{p}, \psi, T)$. Our hypothesis is that such quantities determine how difficult a state is. In order to compare boundary state pairs of different driving models we need to both normalize each quantity for each state and to consider a point of reference to compare with. We define the point of reference as $\mathbf{\Omega} : (d(\mathbf{p}, T) = 0\,m,\, v = 0\,km/h,\, \theta(\mathbf{p}, \psi, T) = 0°)$, i.e., the origin. We then normalize distance, velocity and relative orientation based on their maximum values, i.e., respectively $\frac{W}{2}$ for the distance, v_{max} for the velocity, and θ_{max} for the relative orientation. Given a state s and the vector of normalized quantities $\mathbf{q}$, we compute the radius as the distance between $\mathbf{q}$ and the reference (i.e., $||\mathbf{q} - \mathbf{\Omega}||$) and normalize by dividing by the maximum distance. If a driving model has N boundary state pairs, the radius of the driving model is the average of all the radii for each boundary state pair. The closer the resulting radius is to 1, the better the driving model. The rationale is that a boundary state pair consists of two states that are close to each other and such that one of them is recoverable by the driving model. If the radius is high, then the boundary state pair is challenging, but nonetheless the driving model is able to recover in one of the states of the pair.

We use the Mann-Whitney rank test [31] to assess the statistical significance of the difference between the radii of different models and the Vargha-Delaney effect size (i.e., $\hat{A}_{12}$) to assess the magnitude of such difference [32], as previous literature suggests [33].

RQ_3 (Retraining): *How effective are boundary state pairs in improving the performance of a high-quality driving model?*

In RQ_3, we investigate the usefulness of boundary state pairs for improving a well-behaving driving model. In particular, we consider the boundary state pairs of the best model according to both model-level (i.e., validation loss) and system-level metrics (i.e., recoverability and radius). Our hypothesis is that such boundary state pairs identify challenging driving scenarios that improve the generalization capabilities of the model on a separate set of evaluation scenarios. A positive answer to this question would imply that a fixed and failure-free environment scenario contains hidden states that are useful not only to test but also to improve the ADS.

Metrics. To assess the usefulness of the boundary state pairs of a given model, we measure the success rates of the original model on a set of evaluation tracks different from the training one. We then compare them with the success rates of the model retrained with the boundary state pairs dataset on the same evaluation tracks. We use the Wilcoxon signed-rank test [34] to assess the statistical significance of the difference between the success rates of the two models on the same evaluation tracks. As for the radius, we use the Vargha-Delaney effect size (i.e., $\hat{A}_{12}$) to assess the magnitude of the difference.

A. Procedure

Our experimental procedure consists of: (1) searching for boundary state pairs of different driving models (RQ_1 and RQ_2) and computing recoverability and radius for the boundary state pairs of each of them (RQ_2); (2) improving the best driving model according to validation loss and recoverability/radius, by using its generated boundary state pairs (RQ_3).

1) Implementation and Test Object: We implemented our approach in a Python tool called GENBO. The test object of our study is the popular DNN-based model Dave-2 [23]. Dave-2 is a robust lane-keeping model that has been previously used in several testing works in the literature [7, 9, 10, 11, 20, 24, 25, 26]. We use the Donkey Car™ open-source framework [35, 36] built with Unity [37]. The simulator has been used in previous work to test both supervised learning and reinforcement learning models [20, 38, 39, 40, 41]; it is also featured in a recent survey as one of the prevalent open-source simulation platforms for online ADS testing [3].

2) Model Training: The first step to train a DNN-based driving model is to collect a dataset of labeled images captured by the camera of the vehicle driving along a track. We used the default closed track provided by the simulator, and we resorted to an autopilot with global knowledge of the track to automatically label the images with the right steering angles. We decide the throttle command at runtime, both for the DNN-based driving model and the autopilot, via a linear interpolation between the minimum velocity ($10\,km/h$) and the maximum velocity ($30\,km/h$) so that the vehicle decreases its velocity when the steering angle increases (e.g., in a curve).

As autopilot, we used a PID controller [42] and tuned the Proportional, Integral and Derivative constants for the specific track. We made the PID controller drive the vehicle for three laps of the track, collecting approximately 5*k* labeled images. We used an 80/20 train/validation split to train the Dave-2 model until convergence (we stopped the training after 10 epochs of no improvements of the validation loss). Furthermore, we saved a checkpoint every epoch of validation loss improvement. Of those models we kept four models we used for testing, at decreasing validation losses. The first model, i.e., M_1, is the first well-behaving model. The last model, i.e., M_4, is the model with the best validation loss. We selected the two models in between, i.e., M_2 and M_3, by looking at the validation loss. In particular, we kept a model whenever its validation loss decreased by at least 10% w.r.t. the previously considered model (i.e., M_1 and M_2 respectively).

3) Boundary State Pairs Search: We executed GENBO for the driving models obtained in the previous step, plus the autopilot, which we expect to perform better than the DNN-based models given its global knowledge of the track. We used the autopilot to collect the reference trace Tr we use in Algorithm 1 to sample valid states. As hyperparameters of the algorithm, we used $R = 40$ number of restarts, $N = 10$ iterations for each restart and $L = 3$ as sequence length. When placing the driving model on a certain state, we consider the state recoverable if the model is able to keep the vehicle in lane for at least $12.5s$ which, at 20 *fps*, corresponds to 250 simulation steps. In the track we use, the width of the lane is $W = 4\,m$, hence we chose $\epsilon_p = W \cdot 10\%$. We set the maximum velocity $v_{max} = 30\,km/h$, and $\epsilon_v = v_{max} \cdot 10\%$. Regarding the orientation we chose $\epsilon_\psi = 360° \cdot 2\%$ and $\theta_{max} = 20°$. To account for the randomness of the search algorithm [33] and of the driving simulation [30], we executed GENBO for each driving model 9 times.

To specifically target the randomness of the simulation during the search, when GENBO finds a boundary state pair, we re-execute the driving model on each state of the pair three times. If the state pair is a boundary state pair the majority of the times (i.e., 2 out of 3 times), then we consider the state pair a boundary state pair and store it in the archive. At the end of the search process, we iterate over the boundary state pairs in the archive, and we compute a replication percentage across 10 repetitions, i.e., the number many times a state pair is actually a boundary state pair for the driving model under test out of 10 repetitions. For the subsequent analysis we only consider the boundary state pairs which have a replication percentage greater than zero.

Each run of GENBO consists of at most $N \times R = 400$ driving simulations, plus a variable number of simulations due to replications depending on the number of boundary state pairs in the archive. If we were to discard the replications, we would have 400×5 driving models $\times$ 9 repetitions, consisting of a total of $\approx 18k$ driving simulations.

In order to measure recoverability, we executed each driving model on the boundary state pairs of the others (both recoverable and non-recoverable), keeping track of the number of times a given model is able to recover from a certain state. Also in this case we consider a state recoverable, if the driving model is able to keep the vehicle in lane for at least 250 simulation steps.

4) Driving Model Improvement: Table II shows the list of tracks along with their features (Track Features macro-column). In particular, we measured the curvature of the track (Column 1), i.e., the inverse of the minimum radius of the circles going through each sequence of three consecutive road points [11], and the number of turns (Column 2). The first track, i.e., T_1, is the training track, with a curvature of 0.29 and 4 turns; the second track, i.e., T_2 is the specular version of T_1. To generate the other evaluation tracks, i.e., T_3—T_9, we mutated the control points of T_1 until one of the following condition is true: (1) the distance between T_1 and the candidate track T_c (i.e., the Euclidean distance between the control points of the two tracks being compared) must be lower than a threshold; (2) the curvature of T_c must be greater than the curvature of T_1; (3) the number of turns of T_c must be greater than the number of turns of T_1. The distance constraint imposes that the resulting track is similar to T_1, while the curvature and the number of turns constraints ensure that it is more challenging to drive.

We executed the best driving model on the evaluation tracks, T_2—T_9, to collect the success rates. We used 100 simulations, and we considered a simulation a success if the driving model is able to keep the vehicle in lane for 600 simulation steps (which corresponds to one lap of the track and $30s$ of driving at 20 *fps*). For each search repetition of the best driving model (9 in total), we took the corresponding boundary state pairs in the archive and placed the autopilot in the non-recoverable state of each pair, in order to collect a labeled dataset for retraining. We executed the autopilot for 100 simulation steps per state and discarded the cases where the autopilot was not able to keep the vehicle in lane for that amount of time. Then, we merged the original training dataset and the labeled boundary state pairs dataset for each repetition, splitting the resulting dataset in such a way that the validation set is a superset of the validation set used for the initial training, to mitigate the possibility of regressions on the training track. We carried out retraining with the same hyperparameters of the initial training (random seed included) until convergence, obtaining 9 driving models. Finally, we executed each of them on the 8 evaluation tracks to collect the success rates.

B. Results

1) Existence (RQ_1): Table I, macro-column RQ_1 (Existence), shows the number of boundary state pairs the search process found for each driving model, averaged across the 9 repetitions. We notice that the number of boundary state pairs decreases with the model quality, as measured by the validation loss. The only exception occurs between M_3 and M_{last}, but with a small difference in terms of number of boundary state pairs found. As expected, it is challenging for the search algorithm to find boundary state pairs for the autopilot (on average 1.33), since it is, by construction, robust against changes of the initial conditions of the vehicle.

> **RQ_1 (Existence)**: Boundary state pairs exist for driving models with different performance. In particular, the number of pairs found by the search decreases with the quality of the driving model.

2) Comparison (RQ_2): Table I, macro-column RQ_2 (Comparison), shows the two system-level metrics, i.e., *Radius* (Column 3) and *Recoverability* (Columns 3—14), we used to compare the different driving models under test. All the values in the table (except the Avg column and row), are averages across the 9 repetitions. Regarding the radius, we observe that the metric reaches a plateau after M_2, suggesting that it is able to discriminate low-quality driving models (i.e., M_1) from

TABLE I
RESULTS FOR RQ_1 (EXISTENCE) AND RQ_2 (COMPARISON). BOLD-FACED VALUES INDICATE A STATISTICAL SIGNIFICANT DIFFERENCE W.R.T. THE RADIUS OF M_1; UNDERLINED VALUES INDICATE A LARGE $\hat{A}_{12}$.

	RQ_1 (Existence)	**RQ_2 (Comparison)**											
	# Boundary State Pairs	Radius	Recoverability (%)										
			M_{1_R}	$M_{1_{NR}}$	M_{2_R}	$M_{2_{NR}}$	M_{3_R}	$M_{3_{NR}}$	M_{last_R}	$M_{last_{NR}}$	autopilot$_R$	autopilot$_{NR}$	Avg
M_1	10.78	0.63	–	–	30.41	19.69	33.59	25.54	16.58	4.82	0.00	0.00	16.33
M_2	7.78	**0.72**	98.99	95.86	–	–	90.16	48.04	64.51	36.15	3.57	0.00	54.66
M_3	4.78	**0.75**	100.0	98.99	97.65	84.03	–	–	94.17	54.16	28.57	17.86	71.93
M_{last}	5.89	**0.73**	100.0	97.88	94.02	75.07	97.22	61.77	–	–	32.14	3.57	70.21
autopilot	1.33	**0.75**	100.0	100.0	100.0	100.0	100.0	98.61	100.0	89.44	–	–	98.51
Avg	6.11	–	99.75	98.18	80.52	69.70	80.24	58.49	68.82	46.14	16.07	5.36	–

good to high-quality driving models (i.e., M_2—M_{last}). The radius of the autopilot is in line with the DNN-based driving models M_2—M_{last} while it is statistically different with a large effect size w.r.t. the radius of M_1 (as the radius of the other driving models).

Regarding recoverability, the table shows the average recovery percentage for each driving model on the boundary state pairs of the remaining models across the 9 repetitions. For instance, model M_1 has an average recovery percentage of 30.41% on the recoverable states of M_2 (Column 6, M_{2_R}), while it has an average recovery percentage of 19.69% on the non-recoverable states of M_2 (Column 7, $M_{2_{NR}}$). Looking at the Avg row, we observe that the average recovery percentage decreases when the model quality increases. Indeed, we go from a recovery percentage of 99.75% on the recoverable states of M_1 (i.e., M_{1_R}) to a recovery percentage of 46.14% on the non-recoverable states of M_{last} (i.e., $M_{last_{NR}}$). The recovery percentage further decreases when we consider the autopilot driving model. Its boundary state pairs are challenging for the DNN-based driving models, especially the non-recoverable states of the pairs, as the DNN-based driving models recover only on 5.36% of them on average.

The last column (i.e., Column 14, Avg), shows the average recovery percentage of each driving model on the boundary state pairs of the remaining models. We observe that the recovery percentage increases from 16.33% of M_1 to 98.51% of the autopilot. The recovery percentage of M_2 is more than twice as much that of M_1 (i.e., 54.66%). The recovery percentage further increases when moving from M_2 to M_3 and then plateaus.

It is interesting to notice how our novel recoverability metric behaves when assessing the relative quality of two models. For instance, let us compare M_1 with M_2: M_1 has a recoverability of 30.41% (resp. 19.69%) on the recoverable (resp. non-recoverable) states of M_2, while M_2 has a recoverability of 98.99% (resp. 95.86%) on the recoverable (resp. non-recoverable) states of M_1. This indicates a clear superiority of M_2 over M_1. The same pairwise comparison can be conducted for all pairs of driving models, consistently showing a very strong discriminative capability of the recoverability metric.

RQ_2 (Comparison): Boundary state pairs discriminate driving models with different qualities. In particular, the radius discriminates low-quality from high-quality models at a gross granularity, while the recoverability metric is more fine-grained and discriminates very accurately the driving quality of each pair of models considered in our study.

TABLE II
RESULTS FOR RQ_3 (RETRAINING). BEST SUCCESS RATES ARE HIGHLIGHTED IN BOLD. "NA" INDICATE A RATIO WITH ZERO DENOMINATOR.

	Track Features			Success Rate (%)		Impr.
	Curvature	# Turns	Distance	M_{last}	$M_{retrained}$	
T_1*	0.29	4	–	100.0	100.0	–
T_2†	0.29	4	–	0.000	0.000	NA
T_3	0.34	5	20.13	56.00	**72.00**	1.3×
T_4	0.32	5	19.08	0.000	0.000	NA
T_5	0.28	5	29.31	70.00	**93.00**	1.3×
T_6	0.31	5	33.13	1.000	**16.00**	16×
T_7	0.28	5	31.43	23.00	**78.00**	3.4×
T_8	0.39	7	29.28	0.000	**1.000**	NA
T_9	0.33	5	31.18	0.000	0.000	NA
Avg	–	–	–	18.75	**32.64**	1.7×

* *Training track* † *Specular version of the training track*

3) Retraining (RQ_3): Table II shows the results of retraining the best driving model, M_{last}, using its boundary state pairs. In particular, we show the *Success Rate (%)* on the evaluation tracks of the original and retrained models (Column 5—6), as well as the ratio between the two (*Improv.*). The success rates of $M_{retrained}$ for each driving track are averaged across the 9 repetitions.

We observe the absence of regressions of the retrained models on the training track T_1. In fact, the success rate on T_1 remains 100%. Moreover, the success rate of the retrained models improves in all the evaluation tracks where the original model M_{last} has a non-zero success rate (i.e., T_3, T_5—T_7). On average, the boundary state pairs consistently cause an improvement of the success rate, going from a minimum of 1.3× to a maximum of 16×, considering each evaluation track. The difference between the success rates of the two models is statistically significant with a small $\hat{A}_{12}$ (if we were to remove the zeros for T_2, T_4, T_8 and T_9, the effect size would be large). Indeed, some of the testing tracks are very challenging for M_{last}, which has a success rate of 0%, and on them retraining with the boundary state pairs can only slightly improve the outcome (only on T_8 the success rate of the retrained model is slightly higher). Overall, the success rate of the retrained model $M_{retrained}$ is on average, across all evaluation tracks, 1.7 times the success rates of the original model M_{last}.

RQ$_3$ (Retraining): The boundary state pairs of our best driving model are effective at improving the model through retraining. On average, the success rate of the retrained model on a set of evaluation tracks is 1.7× higher than the original success rate, with a maximum improvement of 16×.

C. Threats to Validity

External Validity. Using one simulator and one ADS poses an external validity threat. To mitigate this issue we selected a widely used ADS from the testing literature [7, 9, 10, 11, 20, 24, 25, 26] (i.e., Dave-2 [23]) and we considered it at different training levels. As driving simulator we selected the popular open-source driving simulator Donkey Car™, also used in previous studies in ADS testing [20, 38, 39, 40, 41]. Both the simulator and the ADS we selected represent the state-of-the-art in ADS testing as a recent survey suggests [3]. Another external validity threat is the limited number of evaluation tracks we used to measure the improvement of the retrained driving model. We addressed this threat by generating a diverse set of evaluation tracks where the original driving model displayed a wide range of success rates, i.e., ranging from high (70%) to low (0%).

Internal Validity. We compared all the driving models under identical settings of the hyperparameters of the search algorithm and we evaluated the improvement of the retrained model on the same evaluation tracks.

Conclusion Validity. We executed the search process multiple times with different random seeds to account for the randomness of the search algorithm. We also ran each driving simulation under the same conditions multiple times to both assess the reliability of a boundary state pair and to measure the success rate. We used rigorous statistical tests to draw our conclusions.

IV. Related Work

Simulation-based Testing. Simulation-based testing is a prominent approach in the literature to test the reliability of autonomous driving systems (ADSs) [3]. Several search-based techniques have been proposed by researchers. AsFault combines procedural content generation and search-based algorithms to generate virtual roads. Abdessalem et al. [15] use decision trees to guide the search towards the critical features of the environment. Calò et al. [16] introduce the concept of avoidable collisions and proposed two search-based algorithms to generate them. Riccio et al. propose DeepJanus [9] that use multi-objective optimization to generate challenging and diverse virtual roads; moreover, they propose DeepHyperion [11, 12] that outputs an explanatory map of the road features that cause a misbehavior. Samota [14] uses a surrogate model to predict the output of the driving simulator in order to efficiently generate critical scenarios. Recently, Morlot [13] uses reinforcement learning (RL) to manipulate the large input space of the driving simulation. Our approach GenBo differs from the above in three distinct ways: (1) GenBo starts from a failure-free driving scenario to extract hidden and challenging driving conditions; (2) GenBo mutates the driving conditions of the ego vehicle instead of acting on the environment; (3) GenBo uses the boundary state pairs associated with challenging driving conditions to improve the ADS under test.

Boundary Input Generation. Instead of generating individual driving scenarios that induce a failure of the ADS under test, researchers proposed search-based approaches to find boundary inputs [43], i.e., inputs that trigger different behaviors of the ADS under test. For instance, Mullins et al. [21] use an adaptive search algorithm to discover performance boundaries of the ADS. Tuncali et al. [22] employ an approach called rapidly-exploring random trees to generate a pair of configurations where a collision is avoidable/unavoidable. Riccio et al. [9] use a multi-objective search algorithm to generate a road shape where the ADS starts to misbehave. In the approach by Biagiola et al. [44], a test case is a pair of environment configurations and the objective is to find boundary pairs such that in one of them the RL agent under test can adapt to the new environment, while in the other it cannot. While such approaches search for boundary pairs of the environment, our approach focuses on the boundary pairs of the vehicle the driving model controls.

Similarly to us, Tappler et al. [45] define a boundary state as a state that precedes another non-terminal state in which the RL agent under test misbehaves. However, their algorithm to search for boundary states relies on symbolic reasoning and symbolic exploration of the execution tree, which does not scale to a complex context such as ADS testing, for which we designed a novel, ad-hoc state exploration algorithm.

V. Conclusion and Future Work

GenBo extracts challenging driving conditions in a failure-free driving scenario by mutating the initial state of the ADS. Experimental results show that boundary state pairs exist even

for high-quality and well-trained driving models. By retraining the ADS under test with examples collected from its boundary state pairs, we significantly improved the success rate of the retrained model on a separate set of evaluation tracks. In our future work, we plan to extend our study to multiple driving simulators and experiment with alternative search algorithms to look for boundary state pairs.

REFERENCES

[1] Society of Automotive Engineers, "Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles," https://www.sae.org/standards/content/j3016_201806, 2016, online; accessed 18 August 2019.

[2] Dan Croutch, "The State of Self-Driving Cars: Autonomous Advances," https://www.techspot.com/article/2644-the-state-of-self-driving-cars/, 2016, online; accessed 23 April 2023.

[3] S. Tang, Z. Zhang, Y. Zhang, J. Zhou, Y. Guo, S. Liu, S. Guo, Y.-F. Li, L. Ma, Y. Xue, and Y. Liu, "A survey on automated driving system testing: Landscapes and trends," *ACM Trans. Softw. Eng. Methodol.*, feb 2023, just Accepted. [Online]. Available: https://doi.org/10.1145/3579642

[4] F. U. Haq, D. Shin, S. Nejati, and L. Briand, "Comparing offline and online testing of deep neural networks: An autonomous car case study," in *Proceedings of 13th IEEE International Conference on Software Testing, Verification and Validation*, ser. ICST '20. IEEE, 2020.

[5] ——, "Can offline testing of deep neural networks replace their online testing? a case study of automated driving systems," *Empirical Software Engineering*, vol. 26, no. 5, p. 90, 2021.

[6] A. Stocco, B. Pulfer, and P. Tonella, "Model vs system level testing of autonomous driving systems: A replication and extension study," *preprint github.io*, 2023.

[7] A. Stocco, M. Weiss, M. Calzana, and P. Tonella, "Misbehaviour prediction for autonomous driving systems," in *Proceedings of 42nd International Conference on Software Engineering*, ser. ICSE '20. ACM, 2020.

[8] A. Gambi, M. Mueller, and G. Fraser, "Automatically testing self-driving cars with search-based procedural content generation," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019. New York, NY, USA: ACM, 2019, pp. 318–328. [Online]. Available: http://doi.acm.org/10.1145/3293882.3330566

[9] V. Riccio and P. Tonella, "Model-based exploration of the frontier of behaviours for deep learning system testing," in *Proceedings of ESEC/FSE*, 2020.

[10] G. Jahangirova, A. Stocco, and P. Tonella, "Quality metrics and oracles for autonomous vehicles testing," in *Proceedings of 14th IEEE International Conference on Software Testing, Verification and Validation*, ser. ICST '21. IEEE, 2021.

[11] T. Zohdinasab, V. Riccio, A. Gambi, and P. Tonella, "Deephyperion: exploring the feature space of deep learning-based systems through illumination search," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 79–90.

[12] ——, "Efficient and effective feature space exploration for testing deep learning systems," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 2, pp. 49:1–49:38, 2023. [Online]. Available: https://doi.org/10.1145/3544792

[13] F. U. Haq, D. Shin, and L. Briand, "Many-objective reinforcement learning for online testing of dnn-enabled systems," *arXiv preprint arXiv:2210.15432*, 2022.

[14] ——, "Efficient online testing for dnn-enabled systems using surrogate-assisted and many-objective optimization," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 811–822.

[15] R. Ben Abdessalem, S. Nejati, L. C. Briand, and T. Stifter, "Testing vision-based control systems using learnable evolutionary algorithms," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, May 2018, pp. 1016–1026.

[16] A. Calò, P. Arcaini, S. Ali, F. Hauer, and F. Ishikawa, "Generating avoidable collision scenarios for testing autonomous driving systems," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2020, pp. 375–386.

[17] L. BGR Media, "Waymo's self-driving cars hit 10 million miles," https://techcrunch.com/2018/10/10/waymos-self-driving-cars-hit-10-million-miles, 2018, online; accessed 23 April 2023.

[18] M. Borg, R. B. Abdessalem, S. Nejati, F.-X. Jegeden, and D. Shin, "Digital twins are not monozygotic–cross-replicating adas testing in two industry-grade automotive simulators," in *ICST '21*. IEEE, 2021.

[19] V. G. Cerf, "A comprehensive self-driving car test," *Communications of the ACM*, vol. 61, no. 2, 2018.

[20] A. Stocco, B. Pulfer, and P. Tonella, "Mind the gap! a study on the transferability of virtual vs physical-world testing of autonomous driving systems," *IEEE Transactions on Software Engineering*, 2022.

[21] G. E. Mullins, P. G. Stankiewicz, R. C. Hawthorne, and S. K. Gupta, "Adaptive generation of challenging scenarios for testing and evaluation of autonomous vehicles," *J. Syst. Softw.*, vol. 137, pp. 197–215, 2018. [Online]. Available: https://doi.org/10.1016/j.jss.2017.10.031

[22] C. E. Tuncali and G. Fainekos, "Rapidly-exploring random trees for testing automated vehicles," in *2019 IEEE Intelligent Transportation Systems Conference (ITSC)*. IEEE, 2019, pp. 661–666.

[23] M. Bojarski, D. D. Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba, "End to end learning for self-driving cars." *CoRR*, vol. abs/1604.07316, 2016. [Online]. Available: http://arxiv.org/abs/1604.07316

[24] A. Stocco, P. J. Nunes, M. d'Amorim, and P. Tonella, "Thirdeye: Attention maps for safe autonomous driving systems," in *Proceedings of 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '22. IEEE/ACM, 2022.

[25] A. Stocco and P. Tonella, "Towards anomaly detectors that learn continuously," in *Proceedings of 31st International Symposium on Software Reliability Engineering Workshops*, ser. ISSREW 2020. IEEE, 2020.

[26] ——, "Confidence-driven weighted retraining for predicting safety-critical failures in autonomous driving systems," *Journal of Software: Evolution and Process*, 2021.

[27] S. Panichella, A. Gambi, F. Zampetti, and V. Riccio, "Sbst tool competition 2021," in *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*. IEEE, 2021, pp. 20–27.

[28] A. Gambi, G. Jahangirova, V. Riccio, and F. Zampetti, "Sbst tool competition 2022," in *2022 IEEE/ACM 15th International Workshop on Search-Based Software Testing (SBST)*. IEEE, 2022, pp. 25–32.

[29] Wikipedia, "Rotation of axes," https://en.wikipedia.org/wiki/Rotation_of_axes, online; accessed April 2023.

[30] A. Afzal, D. S. Katz, C. Le Goues, and C. S. Timperley, "Simulation for robotics test automation: Developer perspectives," in *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2021, pp. 263–274.

[31] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The annals of mathematical statistics*, pp. 50–60, 1947.

[32] A. Vargha and H. D. Delaney, "A critique and improvement of the cl common language effect size statistics of mcgraw and wong," *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.

[33] A. Arcuri and L. Briand, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Software Testing, Verification and Reliability*, vol. 24, no. 3, pp. 219–250, 2014.

[34] F. Wilcoxon, *Individual comparisons by ranking methods*. Springer, 1992.

[35] "Donkey Car," https://www.donkeycar.com/, 2021.

[36] M. E. Tawn Kramer and contributors, "Self driving car sandbox," https://github.com/tawnkramer/sdsandbox, 2021.

[37] "Unity3d." https://unity.com, 2019.

[38] H. Zhou, X. Chen, G. Zhang, and W. Zhou, "Deep Reinforcement Learning for Autonomous Driving by Transferring Visual Features," in *2020 25th International Conference on Pattern Recognition (ICPR)*, 2021.

[39] A. Viitala, R. Boney, Y. Zhao, A. Ilin, and J. Kannala, "Learning to drive (l2d) as a low-cost benchmark for real-world reinforcement learning," *arXiv e-prints*, pp. arXiv–2008, 2020.

[40] A. Verma, S. Bagkar, N. V. S. Allam, A. Raman, M. Schmid, and V. N. Krovi, "Implementation and Validation of Behavior Cloning Using Scaled Vehicles," in *SAE WCX Digital Summit*. SAE International, 2021.

[41] Q. Zhang, T. Du, and C. Tian, "Self-driving scale car trained by deep reinforcement learning," *arXiv preprint arXiv:1909.03467*, 2019.

[42] W. Farag, "Complex trajectory tracking using pid control for autonomous driving," *International Journal of Intelligent Transportation Systems Research*, vol. 18, no. 2, pp. 356–366, 2020.

[43] M. Pezzè and M. Young, *Software testing and analysis: process, principles, and techniques*. John Wiley & Sons, 2008.

[44] M. Biagiola and P. Tonella, "Testing the plasticity of reinforcement learning-based systems," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 4, pp. 1–46, 2022.

[45] M. Tappler, F. C. Córdoba, B. K. Aichernig, and B. Könighofer, "Search-based testing of reinforcement learning," *arXiv preprint arXiv:2205.04887*, 2022.