Testing Vision-Based Control Systems Using Learnable Evolutionary Algorithms

Raja Ben Abdessalem, Shiva Nejati,
Lionel Briand
SnT Centre, University of Luxembourg
{benabdessalem,nejati,briand}@svv.lu

Thomas Stifter
IEE S.A., Luxembourg
thomas.stifter@iee.lu

ABSTRACT

Vision-based control systems are key enablers of many autonomous vehicular systems, including self-driving cars. Testing such systems is complicated by complex and multidimensional input spaces. We propose an automated testing algorithm that builds on learnable evolutionary algorithms. These algorithms rely on machine learning or a combination of machine learning and Darwinian genetic operators to guide the generation of new solutions (test scenarios in our context). Our approach combines multiobjective population-based search algorithms and decision tree classification models to achieve the following goals: First, classification models guide the search-based generation of tests faster towards *critical test scenarios* (i.e., test scenarios leading to failures). Second, search algorithms refine classification models so that the models can accurately characterize *critical regions* (i.e., the regions of a test input space that are likely to contain most critical test scenarios). Our evaluation performed on an industrial automotive vision-based control system shows that: (1) Our algorithm outperforms a baseline evolutionary search algorithm and generates 78% more distinct, critical test scenarios compared to the baseline algorithm. (2) Our algorithm accurately characterizes critical regions of the system under test, thus identifying the conditions that likely to lead to system failures.

KEYWORDS

Search-based Software Engineering, Evolutionary algorithms, Software Testing, Automotive Software Systems

ACM Reference Format:
Raja Ben Abdessalem, Shiva Nejati, Lionel Briand and Thomas Stifter. 2017. Testing Vision-Based Control Systems Using Learnable Evolutionary Algorithms. In *Proceedings of the 40th International Conference on Software Engineering, Gothenburg, Sweden, May 27 - 3 June 2018 (ICSE'18),* 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Autonomous or self-driving vehicles are just around the corner. Increasingly more companies are ramping up their self-driving technologies and teams. Many automotive companies take on-road test initiatives to drive their fleets of autonomous vehicles on real roads. However, there is a large difference between building a few cars to drive under controlled conditions versus large scale production of millions of vehicles that have to operate under realistic and sometimes critical conditions [21]. On-road testing of autonomous cars is typically restricted to a small number of vehicles driven by professional safety drivers during specific hours on some designated roads with specific speed limits. Such testing is often expensive and time-consuming. It is further impractical to perform a full-fledged on-road vehicle-level testing after every change to self-driving software systems. To ensure safety of self-driving technologies, vehicle-level testing alone is neither enough nor practical. Therefore, it needs to be complemented by testing methods performed on computer software simulators [4, 19].

In this paper, we focus on simulation-based testing of vision-based control systems. In the automotive domain, they are referred to as *Advanced Driver Assistance Systems (ADAS)*, and are main enablers of self-driving cars. Examples of ADAS include automatic parking, night vision and collision avoidance systems. Simulation platforms for ADAS [19] allow engineers to run a much larger number of test scenarios compared to vehicle-level testing without being limited by conditions enforced during on-road testing.

The main difficulty with simulation-based testing of ADAS is that the space of test input scenarios is complex and multidimensional. Engineers require techniques that allow them to explore complex test input spaces and to identify *critical test scenarios* (i.e., failure-revealing test scenarios).

Similar to existing work [5, 9, 10, 29], we rely on *evolutionary search techniques* [23] to help engineers explore the complex input space of ADAS and to identify critical test scenarios. It is argued that for testing at the system level, search-based techniques are best suited [37]. They provide effective and flexible guidance for test generation, going beyond test generation based on structural coverage that is not often effective or scalable for system testing.

Evolutionary algorithms work by iteratively sampling the input space, selecting the fittest scenarios (critical test scenarios in our work), and evolving the fittest using genetic search operators to generate new scenarios [23]. The scenarios are expected to eventually move towards the fittest regions in the input space. These algorithms are able to effectively guide the generation of test scenarios towards the most critical ones and can provide useful results regardless of specific time constraints and the size of the input space.

Even though evolutionary search algorithms often scale well to large input spaces, their ability to effectively identify critical test scenarios may diminish as the search space increases in size and dimensions. This is mostly because the search may be stuck in local optima in less critical parts of the input space [23].

ICSE'18, May 27 - 3 June 2018, Gothenburg, Sweden
© 2017 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

In this paper, we provide an algorithm to improve effectiveness of the evolutionary search for large and multidimensional input spaces. Our algorithm builds on *learnable evolution models*, a machine learning-guided form of evolutionary computation [25, 36]. Specifically, we propose to use the set of scenarios generated at intermediary search iterations to build *decision tree classification models* [34]. Decision trees learn the characteristics of the critical test scenarios and identify critical regions in an input space (i.e., the regions of a test input space that are likely to contain most critical test scenarios). We then focus the subsequent search iterations on the critical regions, generating and evolving more critical test scenarios within those regions using genetic operators. We iteratively build decision trees followed by search iterations focused on critical regions identified by the trees. The process stops when we run out of our search time budget. Our algorithm, in addition to guiding the search towards the critical test scenarios faster, produces a decision tree model that identifies the critical regions of the system under test. The critical region characterizations help engineers understand the conditions on input variables that may lead to failures.

The contributions of this paper are summarized below:

- We propose a lightweight formalism for ADAS (i.e., vision-based control systems used in self-driving cars). Our formalism specifies ADAS input and output variables and their critical behaviors. Our formalism is developed based on our analysis of ten different ADAS examples [7] as well as the characteristics of a widely-used, industrial ADAS simulation tool [19].

- We propose a system testing algorithm that combines evolutionary search algorithms and decision tree classification models. Our algorithm has two main objectives, which are important in the context of testing ADAS systems: First, classification models guide the search-based generation of tests faster towards critical test scenarios. Second, search algorithms refine classification models so that the models can accurately characterize critical regions.

- Our evaluation performed on an industrial ADAS shows that: (1) Our algorithm outperforms a baseline evolutionary search algorithm, and generates 78% more distinct, critical test scenarios compared to the baseline algorithm. (2) Based on our interviews with three engineers at our partner company (IEE[1]), the critical region characterizations obtained by our algorithm, while being understandable and intuitive, help engineers debug their systems, identify hardware changes to increase ADAS safety, and specify conditions that may lead to ADAS failures.

The paper is structured as follows: Section 2 motivates our work. Section 3 provides an ADAS formalization. Section 4 describes our approach. Section 5 evaluates our approach. Section 6 compares our work with the related work, and Section 7 concludes the paper.

2 MOTIVATING CASE STUDY

Figure 1 shows an overview of an ADAS example referred to as the Automated Emergency Braking (AEB) system. Its main function is to identify pedestrians in front of a vehicle and to avoid collision by applying the brake when it is necessary. AEB has three main components: (1) *The Sensor Component.* This component identifies the position and speed of objects in a cone-shaped area in front of a vehicle (i.e., the field of view). It uses a radar sensor that is particularly effective in identifying the speed, distance and azimuth angle of objects, but is not able to identify the type and shape of the objects. It also computes *time to collision (TTC)* that measures the time required for a vehicle to hit an object if both continue with the same speed and do not change their paths [30]. When an object is detected in front of a vehicle and when the TTC is below a defined threshold, the object position is sent to the vision component. (2) *The Vision (Camera) Component.* This component detects object types and shapes after receiving their positions from the sensor component. Cameras are complementary to sensors. They cannot effectively determine objects' positions and speeds. But given the position of the detected object, cameras can determine whether the object is a pedestrian (human or animal), a car, a traffic-sign, etc. After identifying the speed, position and type of the object, the system is able to decide whether braking is needed and sends a command to the brake control component when braking is necessary. This command determines the required braking force to stop the car and avoid accidents. (3) *The Braking Control Component.* This component applies the braking request.

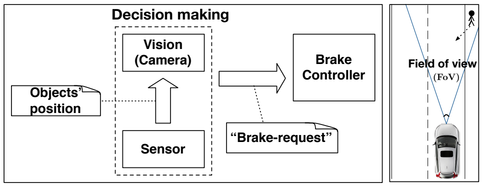

Figure 1: An example of a vision-based control system: Automated Emergency Braking (AEB) system.

To simulate AEB, we use the PreScan simulator [19]. PreScan is a widely-used, commercial ADAS simulator in the automotive sector and has been used by IEE. It allows us to define and execute scenarios capturing various road traffic situations and different pedestrian-to-vehicle and vehicle-to-vehicle interactions. In addition, using PreScan, one can vary road-topologies, weather conditions and infrastructures in test scenarios.

Figure 2 shows a domain model capturing the test input space and the output of AEB. Based on our analysis, we categorize the AEB input variables into two categories:

I. *Static input variables.* The values of these variables are fixed during ADAS simulation and they include: (1) Different road types (e.g., straight, curved or ramped). For the curved and ramped roads, we specify the curve radius and the ramp height, respectively. (2) Different weather types: normal, rainy and snowy. For each of the snowy and rainy weather types, we specify the level of precipitation. For each weather type, we may or may not have fog with different density levels. Finally, we specify a visibility range, i.e., the distance at which the objects can be clearly seen. As Figure 2 shows, we have defined enumerations for the road radius and height, the level of precipitation for rain and snow, fog density, and visibility range. According to the domain experts in IEE, these enumerations provide a desired level of granularity for analysis, and hence, static variables do not need to be real or integer.

II. *Dynamic (mobile) objects.* They indicate objects that change their positions during ADAS simulation, i.e., pedestrians and vehicles. For AEB, we consider two mobile objects: one pedestrian and one vehicle, and assume linear trajectories for them. These assumptions are meant to reduce the complexity of test scenarios and

[1] IEE (International Electronics & Engineering) s.a., https://www.iee.lu/.

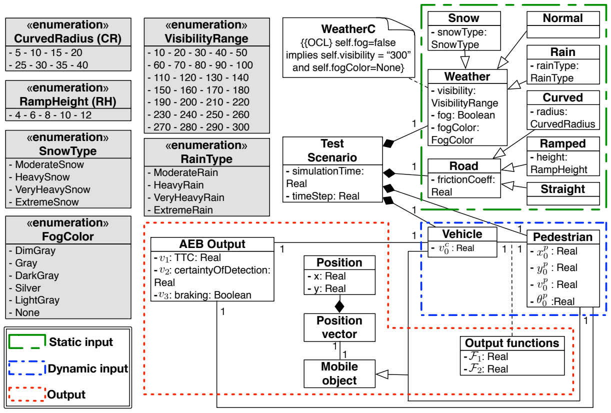

Figure 2: The AEB domain model.

Curved road	Ramped road	Straight road
Range θ_0^p = [120..250]	Range θ_0^p = [40..160]	Range θ_0^p = [40 ..160]
Range x_0^p= [32..50]	Range x_0^p= [60..95]	Range x_0^p= [30 ..85]
Range y_0^p= [50..76]	Range y_0^p= [2..16]	Range y_0^p= [24 ..36]

Figure 3: The ranges of the pedestrian position (x_0^p, y_0^p) and orientation (θ_0^p) for different road topologies.

were suggested by the domain experts. For the vehicle, we require to know its initial speed (v_0^c). The pedestrian has four variables characterizing its initial position along x and y axes and relative to the position of the vehicle (x_0^p, y_0^p), its orientation angle (θ_0^p) and its initial speed (v_0^p). The dynamic objects variables are float. Figure 3 shows the ranges for the pedestrian initial position and orientation variables when the road is curved, ramped and straight, respectively. The ranges for vehicle and pedestrian speed variables are $[1km/h..90km/h]$ and $[1km/h..18km/h]$, respectively.

In addition to variable ranges, the valid inputs of ADAS are determined by constraints defined over the input variables. These constraints are either defined on static input variables, or they specify how value assignments to static variables impact the ranges of the mobile object variables. An example of the constraints defined over AEB static variables is shown using the OCL language [16] in Figure 2 (see WeatherC-OCL). The constraint states that when there is no fog, the visibility range is set to maximum. The constraints that relate static variables of AEB to ranges of mobile object variables are captured in Figure 3. Specifically, the figure specifies the valid ranges for pedestrian position and orientation variables corresponding to different road topologies.

ADAS simulations have two outputs: **I.** Position vectors for mobile objects (i.e., position vectors for the vehicle and the pedestrian in the AEB case study). The position vector related to each mobile object stores the position of that object at each simulation time step. **II.** Function specific output variables: Each ADAS, depending on its function, produces some outputs. For example, AEB produces three outputs corresponding to its three main components: (1) *Time to collision (TTC)* generated by the sensor component and discussed earlier. (2) *certaintyOfDetection* generated by the vision component which is a percentage value indicating the probability that the detected object is a pedestrian. (3) *Braking* that indicates whether a braking request has been triggered.

The following describes the main AEB critical (or failure) behavior extracted from the AEB requirements: "*AEB detects a pedestrian in front of the car with a high degree of certainty, but an accident happens where the car hits the pedestrian with a relatively high speed (i.e., more than* $30km/h$)". We denote this critical behavior by *CB*, and refer to any AEB simulation scenario exhibiting this behavior as a *critical test scenario* of AEB.

The test input space of AEB is large and multidimensional. As we will specify in Section 3, it consists of four enumeration (static) and five float (dynamic) variables. Considering only the static AEB variables, their total number of value assignments is 11, 242. Further, AEB simulations (and in general ADAS simulations) are computationally expensive. This is because the underlying simulator (e.g., PreScan) builds on high-fidelity mathematical models and takes a relatively large amount of time to run (e.g., on average, each AEB simulation takes 1.2 min). Our goal is to provide an effective algorithm that, within a reasonable testing time budget: (1) generates AEB critical test scenarios (i.e., those exhibiting *CB*), and (2) identifies under what conditions on the AEB input variables such critical scenarios are more likely to occur. The latter will provide engineers with critical region characterizations, allowing them to better understand the conditions under which AEB fails to behave correctly.

3 ADAS FORMALIZATION

In this section, we provide a generic formalism to capture an ADAS system and its environment. Our formalization is meant to help define our algorithm precisely, and to demonstrate how our work can be applied to other ADAS systems. Our formalization is developed based on our analysis of ten ADAS examples as well as the input and configuration variables of the PreScan tool [19]. Generic descriptions of our ADAS examples can be found on the Bosch website [7].

Definition 3.1. We define an ADAS as a tuple (S, O, I, D, C), where

- $S = \{s_1, \ldots, s_n\}$ is an ordered set of *variables* specifying (immobile) static environment aspects.

- O is a set of *mobile objects* (pedestrians and vehicles).

- $I = \{i_1, \ldots, i_m\}$ is an ordered set of variables specifying *initial states* of the mobile objects in O. Each variable in I is related to one mobile object in O, while each mobile object $o \in O$ is related to one or more variables in I.

- D is a set of (non-empty) domains of values for variables in $S \cup I$. In particular, D is partitioned into D_S and D_I ($D = D_S \cup D_I$) such that $D_S = \{D_1, \ldots, D_n\}$ is an ordered set of finite value sets to variables in S, while $D_I = \{D'_1, \ldots, D'_m\}$ is an ordered set of infinite value sets to variables in I. Specifically, D_j is the set of values for $s_j \in S$, and D'_j is an interval $[min...max]$ of real values specifying the values that $i_j \in I$ can take.

- C is a set of boolean propositional constraints over $S \cup I$. The set C is partitioned into C_S and C_I such that constraints in C_S are defined on finite-domain variables in S, and constraints in C_I relate finite-domain variables in S to infinite-domain variables in I.

Example 3.1. We formalize AEB in Figure 1 as follows:

- Static variables (S): s_1 (precipitation), s_2 (fogginess), s_3 (road shape) and s_4 (visibility range).

- Mobile objects (O): o_1 (vehicle) and o_2 (pedestrian).

- Dynamic variables (I): v_0^c (initial speed of vehicle), v_0^p (initial speed of pedestrian), x_0^p (initial position of pedestrian on the x-axis), y_0^p (initial position of pedestrian on the y-axis) and θ_0^p (the orientation of pedestrian).

- The domain of s_1 is the union of RainType and SnowType enumerations in Figure 2 as well as a value for normal weather. Variable s_2 takes values from the FogColor enumeration. The domain of s_3 is the union of RampedHeight and CurvedRadius enumerations and a value for the straight road. Variable s_4 takes values from the VisibilityRange enumeration. The ranges for dynamic variables were discussed in Section 2.

- The constraints over static variables (C_S) relate the level of fog (s_2) to the visibility range (s_4). An example of a C_S constraint is: (s_2 = "$DimGray$" $\Rightarrow s_4 = 10 \vee \ldots \vee s_4 = 100$). The constraints over static and dynamic variables (C_I) relate the shape of the road (s_3) to different ranges for x_0^p, y_0^p and θ_0^p (see Figure 3). An example of a C_I constraint is: (s_3 = "$RH4$" $\vee \ldots \vee s_3$ = "$RH12$" $\Rightarrow D_{x_0^P} = [60..95] \wedge D_{y_0^P} = [2..16] \wedge D_{\theta_0^P} = [40..160]$).

We denote by $Z \subseteq D_1 \times \ldots \times D_n \times D'_1 \times \ldots \times D'_m$ the set of value assignments to variables in $S \cup I$ satisfying all the constraints in C. An ADAS simulation function Σ takes as input a value assignment $z \in Z$ and a value $T \in \mathbb{N}$ indicating the simulation duration (i.e., the number of simulation steps). The output of Σ is (1) a set U of *output vectors* indicating the position and speed of mobile objects at each simulation time step, and (2) a set V of (time-independent) output variables. Specifically, U captures the dynamic behavior of ADAS and the environment (i.e., how mobile objects move over time). Each position vector $u \in U$ corresponds to one and only one mobile object $o \in O$ and is a function $u : \{0, 1, \ldots, T\} \rightarrow \mathbb{R}^3$ where $u(t)$ ($t \in \{0, \ldots, T\}$) is a triple (x, y, v) indicating the position (x, y) and the speed v of the mobile object related to u at time t. The set V determines the function-specific outputs produced by decision-making components of an ADAS.

Example 3.2. AEB generates two position vectors (U): u_1 (for vehicle) and u_2 (for pedestrian); and three decision-making outputs (V): (1) TTC denoted by v_1, (2) *certaintyOfDetection* denoted by v_2, and (3) *braking* denoted by v_3 (see Figure 2).

To specify critical behaviors of ADAS, we define (auxiliary) functions over the dynamic system outputs U. For example, let u_1 and u_2 be position vectors generated for AEB over simulation time T. We define two functions: (1) $\mathcal{F}_1(u_1, u_2)$ that computes the minimum distance between the pedestrian (u_2) and the field of view of the vehicle (u_1), and (2) $\mathcal{F}_2(u_1, u_2)$ that computes the speed of the car at the time of collision, and returns -1 if collision does not occur.

We formalize the AEB critical behavior CB described in Section 2. Given AEB outputs $U = \{u_1, u_2\}$ and $V = \{v_1, v_2, v_3\}$, we define $CB(U, V)$ as follows:

$$CB(U,V) = (\mathcal{F}_1(u_1,u_2) < 50cm) \wedge (v_2 > 0.5) \wedge (\mathcal{F}_2(u_1,u_2) > 30km/h) \quad (1)$$

The CB property states that: a pedestrian is in front of a car ($\mathcal{F}_1(u_1, u_2) < 50cm$), is detected by AEB with a high certainty ($v_2 > 0.5$), and the car hits the pedestrian with a speed higher than $30km/h$ ($\mathcal{F}_2(u_1, u_2) > 30km/h$). The constant values $50cm$, 0.5 and $30km/h$ are taken from the AEB specification. An AEB test scenario generating U and V is critical if and only if $CB(U, V)$ is true.

4 SEARCH GUIDED BY CLASSIFIERS

In this section, we describe our ADAS testing algorithm that combines multi-objective search and decision tree classification models.

4.1 Multi-objective search

The formalization of ADAS critical behaviors depends on several ADAS outputs. For example, formalizing the CB behavior (see equation (1)) relies on three AEB outputs $\mathcal{F}_1$, $\mathcal{F}_2$ and v_2. We cast the problem of computing ADAS critical test scenarios as a *multi-objective search optimization problem* [23] where the ADAS outputs specifying its critical behaviors act as the search fitness functions. We use the Non-dominated Sorting Genetic Algorithm version 2 (NSGAII) [15, 23], which has been previously applied to several software engineering problems including ADAS testing [5]. The NSGAII algorithm generates a set of solutions forming a *Pareto nondominated front* [15, 23]. A dominance relation over solutions is defined as follows: A solution x dominates another solution y if x is not worse than y in all fitness values, and x is strictly better than y in at least one fitness value. The output of NSGAII is a non-dominating (equally viable) set of solutions, representing best-found trade-offs among fitness functions. In our work, NSGAII generates a number of ADAS critical test scenarios by maximizing or minimizing the ADAS outputs characterizing its critical behavior.

We do not present the details of the NSGAII algorithm due to lack of space. To learn more about this widely-used algorithm, see [23]. Here, we discuss how we tailor NSGAII to ADAS testing:

Representation. A feasible solution is a vector of values to static variables $s_1, \ldots, s_n$ and dynamic variables $i_1, \ldots, i_m$ of the ADAS under analysis such that each vector satisfies the constraints in C. Each such vector defines an ADAS test scenario. Simulating each vector generates outputs U and V that can be used to compute fitness functions.

Initial population. An initial population for our search algorithm is a set P consisting of vectors of ADAS test scenarios. We aim to generate P by selecting a diverse set of vectors from the input space. We generate P with size q as follows: First, we generate q vectors of value assignments to static variables $s_1, \ldots, s_n$ using t-wise combinatorial testing [22] such that (1) the C_S constraints hold, and (2) the pairwise coverage of variables s_1 to s_n is maximized. We use the PLEDGE tool [18] for this purpose. Second, we use an adaptive random search algorithm [23] to generate a large number ($> q$) of value assignments to dynamic variables $i_1, \ldots, i_m$. Adaptive random search is an extension of the naive random search that attempts to maximize the Euclidean distance between the points selected in the input space. Third, for each static variable vector, we select a dynamic variable vector such that the constraints C_I (i.e., constraints between static and dynamic variables) hold. If for some static variable vector v we cannot find such dynamic variable vector among the existing randomly generated pool, we perform some more iterations of (adaptive) random search within the value ranges accepted by the C_I constraint for v. The initial population set P is complete when every static variable vector is matched to one dynamic variable vector. Note that in our ADAS formalization, we do not have any constraint among the dynamic variables.

Fitness Functions. Fitness functions are defined based on the ADAS outputs specifying its critical behavior. For the AEB case study, fitness functions are the two functions $\mathcal{F}_1$ and $\mathcal{F}_2$, and the output variable v_2. These are used to formalize the critical behavior of AEB (the *CB* behavior in Section 3). To generate critical test scenarios, we maximize $\mathcal{F}_2$ and v_2, and minimize $\mathcal{F}_1$. This is because for scenarios exhibiting *CB*, the values of $\mathcal{F}_2$ and v_2 should be larger than a threshold, and $\mathcal{F}_1$ should be smaller than a threshold.

Genetic operators. The genetic operators of NSGAII should be defined such that the generated test scenario vectors satisfy the C_S and C_I constraints. Here, we provide crossover and mutation operators that respect pairwise C_S constraints, and C_I constraints relating one static variable to one or more dynamic variables. The constraints of the AEB case study as well as those of the ten ADAS systems we have studied in our work conform to these conditions. Specifically, in all of these systems, the C_S constraints relate the weather properties (e.g., fog-level (s_2) to visibility range (s_4)), and the C_I constraints relate different road shape types (s_3) to the ranges of dynamic variables x_0^p, y_0^p, and θ_0^p (see Example 3.1).

Selection: We use a binary tournament selection with replacement that has been used in the original implementation of NSGAII [15].

Crossover: To avoid violating the C_S constraints, crossover is not applied to the static segments of the vectors. That is, our crossover operator is applied to dynamic segments of the vectors only (i.e., $(i_1, \ldots, i_m)$). To avoid violating the C_I constraints, we match pairs of vectors with the same value for the static variables participating in C_I (e.g., the same value for s_3 in the AEB case study). If we do not find any match for some parent vector, we match two vectors with the smallest Euclidean distance between the variables participating in the C_I constraints. We then use Simulated Binary Crossover operator (SBX) [6, 13] that has been previously applied to vectors of float variables. The difference between offsprings generated by SBX and their parents is controlled by a distribution index (η): The offsprings are closer to the parents when η is large, while with a small η, the difference between offsprings and parents will be larger [14]. In this paper, we chose a high value for η (i.e., $\eta = 20$) based on existing guidelines [13]. Given that η is large, even when parents do not have the same values for the static variables in C_I, the values of the dynamic variables in each of the two offsprings are likely to fall within the valid ranges of their respective parent vector. Hence, the C_I constraints are likely to still hold after applying SBX in such situations. If the resulting values are out of variable ranges after crossover, we cap them at the max or min of the ranges when they are closer to the max or min, respectively.

Mutation: Mutation is applied after crossover to static and dynamic variables with a probability (mutation rate). To avoid violation of the C_I constraints, we do not mutate static variables participating in the C_I constraints. Note that since the initial population is generated by maximizing pairwise coverage of static variables, different value combinations of the static variables in C_I are already present in the initial population. Except for static variables in C_I, all other static and dynamic variables can be mutated. We mutate a static variable not appearing in C_S by randomly changing its value within its valid range. For a pair s_i and s_j of static variables appearing in a C_S constraint we define a closed mutation operator as follows: after mutating s_i (respectively s_j), we identify the set of values for s_j (respectively s_i) consistent with the new value of s_i (respectively s_j), and randomly change s_j (respectively s_i) to one of those values. To mutate a dynamic variable, we shift the variable by a value selected from a normal distribution with mean $\mu = 0$ and a small variance. Similar to the crossover operator, if the resulting values are out of variable ranges, we cap them at the max or min of the ranges when they are closer to the max or min, respectively. For the AEB case study, our tweak operator is able to (potentially) generate any valid value combinations of the AEB static and dynamic variables.

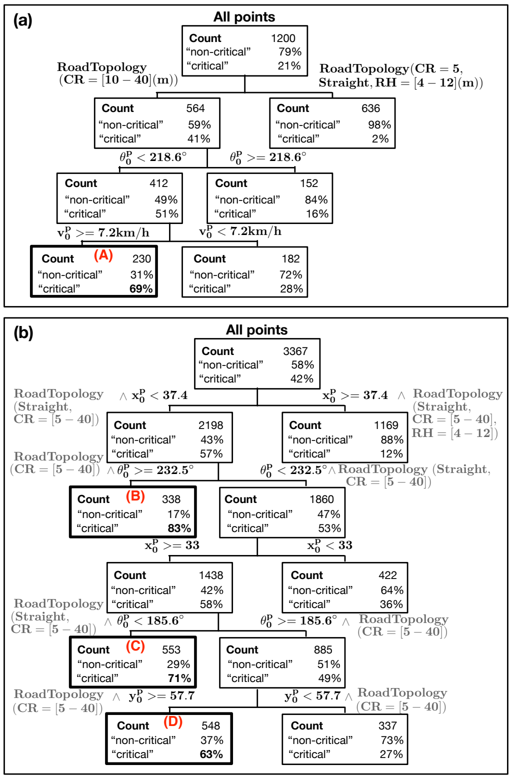

Figure 4: Decision trees generated our approach for the AEB system: (a) An initial decision tree, and (b) A decision tree obtained after some iterations of the NSGAII-DT algorithm.

4.2 Decision tree learning

Decision tree learning is a supervised learning classification technique [1, 34]. Supervised learning techniques are trained based on labeled data, and are divided into *regression* and *classification* techniques where the goal is to predict real-valued and categorical outputs, respectively. In this paper, we use classification decision trees. In this paper, we use boolean functions such as *CB* (see equation (1) in Section 3) to label each ADAS test scenario as critical or non-critical. Alternatively, we could characterize the critical behavior as a real-valued function and use regression trees instead.

In contrast to other learning techniques (e.g., SVM), decision tree boundaries are parallel to the dimensions of the input space and expressible in terms of linear conditions over input variables. This makes decision tree boundaries understandable by practitioners, and has been a main reason why we selected them in our work.

Figure 4 shows two decision trees generated for the AEB case study. The input data for building decision trees is a set of AEB test scenario vectors. The label for each scenario is computed by first simulating the scenario and then labeling it either as critical or non-critical by applying the *CB* function to the scenario simulation outputs. A decision tree model is built by partitioning the set of labeled test scenarios in a stepwise manner aiming to create partitions with increasingly more homogeneous labels (i.e., partitions in which the majority of scenarios are labeled either as critical or non-critical). For example, the tree in Figure 4(a) shows that out of the 636 scenarios that use a straight, ramped or curved (with **CR = 5**) road, 98% were not critical (did not exhibit *CB*).

The tree leaves containing more than 50% critical scenarios (i.e., leaves **A** to **D** in Figure 4) are critical regions. For example, out of the total of 1200 scenarios, 230 of them are classified in the critical region (**A**), and 69% of them are critical. Each critical region is specified by conjoining the conditions appearing on the path from the root to the critical region. For example, the critical region **A** is characterized as follows: $v_0^P \geq 7.2\text{km/h} \wedge \theta_0^P < 218.6° \wedge (s_3 = \text{“}CR10\text{”} \vee \ldots \vee s_3 = \text{“}CR40\text{”})$.

At each (non-leaf) node, a decision tree partitions the data in that node based on a condition on only one variable. However, due to the C_S and C_I constraints, a decision tree condition on a variable v may additionally constrain variables other than v but related to v via C_S or C_I. As discussed in Section 4.1, our genetic operators respect the C_S and C_I constraints. Hence, when our search algorithm applies these operators to a specific critical region (as we will discuss in Section 4.3), the operators automatically handle both the constraints explicitly identified by the tree and the additional constraints implied by C_S or C_I. However, as critical region characterizations are among outputs of our approach, we explicate these additional constraints when the regions are presented to engineers. For example, in Figure 4(b), the conditions in gray color are not directly identified by the decision tree but are implied based on the AEB constraints.

We note that, in this paper, we do not use decision trees to predict whether a given ADAS scenario is critical or not (i.e., the decision trees are not used as predictor models). We exclusively use the decision trees: (1) to better guide the search, and (2) to characterize the critical regions of the ADAS input space. Further, to avoid overfitting in the trees generated by our approach, in Section 5.3, we define a stopping criterion to control the tree expansion such that the number of vectors in each tree leaf does not fall below a certain threshold.

4.3 NSGAII guided by decision trees

Algorithm 1 shows our proposed algorithm, NSGAII-DT, that generates critical test scenarios and critical regions for ADAS. NSGAII-DT receives as input an ADAS specification, a set of (quantitative) fitness functions, a boolean *label* function indicating whether a test scenario is critical or not, and a parameter g indicating the number of search iterations we perform in each critical region. The output of NSGAII-DT is a set of the critical test scenarios and the critical regions $R_1, \ldots, R_k$ of the ADAS input space.

Algorithm 1: NSGAII-DT

```
Input:  - $(S, O, I, D, C)$: An ADAS specification
        - $F_1, \ldots, F_l$: Search fitness functions
        - label: A boolean function to label scenarios as critical/non-critical
        - $g$: Number of search iterations to be applied at each critical leaf
Result: - criticalScenarios: A set of critical test scenarios
        - $R_1, \ldots, R_k (\subseteq D_1 \times \ldots \times D_n \times D'_1 \times \ldots \times D'_m)$: A set of critical regions
1  begin
2      Select an initial population set $P$ randomly.
       /*Each $p \in P$ is a vector of values for $(s_1, \ldots, s_n, i_1, \ldots, i_m)$ */
3      $k \leftarrow 1$; $Best \leftarrow \emptyset$
4      $R_1 \leftarrow D_1 \times \ldots \times D_n \times D'_1 \times \ldots \times D'_m$
       /*$R_1$ is the entire search space and includes all elements in $P$*/
5      repeat
6          for $i = 1$ to $k$ do
7              $Q \leftarrow P \cap R_i$
8              $B, Q' \leftarrow$ NSGAII$(g, Q, F_1, \ldots, F_l, R_i, C)$
               /*Inputs passed to NSGAII:
               $g$: the number of search iterations applied to each critical leaf;
               $Q$: the set of scenarios used as the initial population of NSGAII;
               $F_1, \ldots, F_l$: search fitness functions;
               $R_i$: the critical leaf in which we want to run NSGAII; and
               $C$: the ADAS constraints.
               Outputs received from NSGAII:
               $Q'$: all the solutions generated during search; and
               $B$: best solutions generated by NSGAII.*/
9              $P \leftarrow P \cup Q'$
10             $Best \leftarrow B \cup Best$
11         $rank_1, \ldots, rank_t \leftarrow ComputeRanks(Best)$
12         $criticalScenarios \leftarrow rank_1$
13         $(P^+, P^-) \leftarrow ComputeLabel(label, P)$ /*$P^+$ : non-critical, $P^-$ : critical*/
14         Build a decision tree Tree based on $(P^+, P^-)$
15         Let $R_1, \ldots, R_k$ characterize the leaves of Tree where $P^-$ has a higher
           probability than $P^+$
           /* For each region $R_i = d_1 \times \ldots \times d_n \times d'_1 \times \ldots \times d'_m$,
           we have: $\forall j \in \{1, \ldots, n\} \Rightarrow d_j \subseteq D_j$, and $\forall j \in \{1, \ldots, m\}$,
           $\exists min \in D'_j, \exists max \in D'_j$ s.t. $min < max \wedge d'_j = [min..max]$*/
16     until search time has run out
```

NSGAII-DT starts with an initial and randomly selected population set P (line 2). Each iteration of NSGAII-DT consists of the following main steps: *First*, it performs a number of (genetic) search iterations using NSGAII in critical regions of the input space (lines 6–10). Specifically, for each critical region R_i, the set Q of the elements inside R_i is passed as the initial population to NSGAII along with the parameter g (i.e., the number of search iterations to be applied in R_i). We further pass R_i and C to NSGAII. In particular, R_i specifies the ranges of input variables valid for the critical region under search. Provided with the variable ranges and the constraints (i.e., the set C), our mutation and crossover operators described in Section 4.1 can generate new vectors within the region R_i. Note that in the first iteration, the only critical region is the entire input space (R_1 in line 4). NSGAII returns two sets: Q' and B where Q' is the set of all scenarios and B is the set of most critical scenarios computed by NSGAII.

Second, NSGII-DT identifies the scenarios on the best Pareto front rank computed so far (lines 11–12). In particular, it identifies the best Pareto front rank in set *Best* (i.e., the set of all best solutions generated by all invocations of NSGAII). *Third*, NSGII-DT builds a decision tree based on the labeled set of all the scenarios generated up to that point (lines 13–14). Specifically, it computes the label for each $p \in P$, partitions P into P^+ (the non-critical set) and P^- (the critical set), and builds a decision tree based on the labeled data. *Fourth*, it updates the set of desired input space regions in which subsequent search iterations are performed (line 15). Specifically, it identifies the *critical* leaves $R_1, \ldots, R_k$ of the tree such that the probability of failure P^- is higher than that of non-failure P^+. Each R_i is a sub-region of the ADAS input space and is specified as the conjunction of the conditions on the tree paths leading to the leaves containing more elements from P^- than from P^+.

NSGAII-DT can be stopped when we run out of time. Alternatively, we can stop NSGAII-DT when all the tree leaves classify critical scenarios with a high probability (e.g., more than 95%) or when the fitness functions do not improve for the scenarios in the *criticalScenarios* set compared with the previous iteration.

For example, the decision trees in Figure 4 are computed by applying NSGAII-DT to the AEB case study. Figure 4(a) shows an initial tree, and Figure 4(b) shows a tree after a few search iterations. The tree in Figure 4(b) contains more conditions, and identifies three critical regions B, C and D, instead of one such region in Figure 4(a). Further, the regions B, C and D are considerably more specific than region A as they prune the domains of the input variables more.

We note three important aspects of NSGAII-DT: (1) In ADAS testing, the most time-consuming part of the search is running simulations to compute fitness functions. NSGAII-DT does not increase the number of simulations compared to NSGAII. The fitness values computed by the NSGAII search (line 8) are reused at line 13 to label the new elements. (2) To rebuild the tree in line 14, we use all the scenarios generated and simulated by NSGAII (i.e., Q'). Since computing simulations is expensive and to build more accurate trees, we try to reuse as much as possible the simulation outputs computed by NSGAII. (3) In our work, we run NSGAII in leaves that classify critical scenarios with a probability lower than 95%. This is to use search time budget exploring the critical parts in the input space about which we have less certainty regarding criticality. These are parts of the space where the tree may need to be refined.

5 EVALUATION

In this section, we present the result of our evaluation performed on the AEB case study.

5.1 Research Questions

RQ1. *Does the decision tree technique help guide the evolutionary search and make it more effective?* The most important criterion for a search algorithm to be effective in the context of ADAS testing is that it should be able to generate critical test scenarios, in particular, in large and multidimensional search spaces. To answer this question, we determine whether NSGAII-DT (i.e., our proposed algorithm that is guided by both decision trees and genetic operators) is able to generate scenarios that are more critical compared to those obtained by the NSGAII algorithm (i.e., the baseline evolutionary search algorithm).

RQ2. *Does our approach help characterize and converge towards homogeneous critical regions?* After evaluating the ability of NSGAII-DT in generating critical test scenarios in **RQ1**, we evaluate the critical regions. In particular, in **RQ2**, we investigate whether the decision trees generated by NSGAII-DT are able to precisely characterize critical regions in ADAS input spaces and increasingly do so better over NSGAII-DT iterations.

In Section 5.4, in addition to answering **RQ1** and **RQ2**, we provide qualitative insights into the benefits of our approach from the perspective of practitioners.

5.2 Metrics

To answer **RQ1**, we compare the Pareto fronts generated by NSGAII-DT and NSGAII using three well-known quality indicators for evaluating multi-objective search results [20]: Hypervolume (HV), Generational Distance (GD), and Spread (SP). To compute the quality indicators, following existing guidelines in the literature [33], we compute a *reference* Pareto front as the union of all the non-dominated solutions obtained from all runs of NSGAII-DT and NSGAII. The HV quality indicator [38] measures the size of the space covered by the members of a Pareto front generated by a search algorithm. The higher this size, the better the results of the algorithm. The GD quality indicator [31] measures the Euclidean distance between members of a Pareto front and the nearest solutions on a reference Pareto front. The lower the value of GD, the more optimal the Pareto front solutions. The SP quality indicator [15] measures the extent of spread among the members of a Pareto front generated by a search algorithm [15]. The lower the SP values, the better spread out the search outputs.

To answer **RQ2**, we use the *RegionSize*, the *GoodnessOfFit* and the *GoodnessOfFit-crt* metrics defined below.

RegionSize measures the size of the critical regions as a percentage of the size of the entire input space. It is used to determine whether the critical regions become smaller and more specific over NSGAII-DT iterations. Let D_1 to D_n and D'_1 to D'_m be the dimensions of the input space (as defined in Section 3). Recall from the NSGAII-DT algorithm (line 15 in Algorithm 1) that the dimensions of a region R_i are characterized by $d_1 \times \ldots \times d_n \times d'_1 \times \ldots \times d'_m$ such that d_1 to d_n are respectively (finite) subsets of D_1 to D_n, and d'_1 to d'_n are respectively sub-intervals of the intervals D'_1 to D'_m. We define *RegionSize*(R_i) as follows:

$$RegionSize(R_i) = \prod_{j=1}^{n} \frac{|d_j|}{|D_j|} \times \prod_{j=1}^{m} \frac{max(d'_j)-min(d'_j)}{max(D'_j)-min(D'_j)}$$

RegionSize for the entire input space is equal to one, and the lower *RegionSize*(R), the smaller the region R. For example, for the tree in Figure 4(a), we have *RegionSize*(**A**) = 0.25, and for that in Figure 4(b), we have *RegionSize*(**B**) = 0.02, *RegionSize*(**C**) = 0.03 and *RegionSize*(**D**) = 0.03, implying that the size of critical regions are reduced over subsequent iterations of NSGAII-DT. As discussed in Section 4.2, in our work, input variable domains are reduced in two ways: by the explicit conditions on tree edges and due to ADAS constraints, i.e., the gray conditions in Figure 4(b). In order to accurately compute *RegionSize* (e.g., for **B**-**D** in Figure 4(b)), we consider both explicit and implicit domain reductions.

GoodnessOfFit is used to determine how well the trees generated during the search fit to the set of scenarios sampled during the search. Similarly, *GoodnessOfFit-crt* determines goodness of fit for critical scenarios only. Each decision tree is built based on a labeled set $P^+ \cup P^-$ of elements (see line 14 in Algorithm 1). The *GoodnessOfFit* of each decision tree is the number of elements in $P^+ \cup P^-$ that are correctly classified by the tree (either as critical or non-critical) divided by $|P^+ \cup P^-|$. Similarly, the *GoodnessOfFit-crt* for each tree is the number of elements in P^- that are correctly classified by the tree (as critical) divided by $|P^-|$. Note that since we do not use the classification trees as prediction models, we do not evaluate them based on cross validation with test sets. Instead, we assess how well the trees characterize critical scenarios, while avoiding overfitting as discussed in Sections 5.3 and 5.4.

5.3 Experiment Design

We applied both NSGAII-DT and NSGAII to the AEB case study introduced in Section 2. For both algorithms, we set the (initial) population size to 100, the mutation rate to 0.11, and the crossover rate to 0.6. Specifically, the mutation rate is $1/l$ where l is the chromosome size (nine in our work). The search parameter values are consistent with existing guidelines [3]. We could further tune the parameters to improve the performance of our approach. However, as shown in Section 5.4, the results obtained based on the current setting clearly support our analysis. Hence, we do not report experiments targeted at tuning population size, and crossover and mutation rates.

We set the search time to 24 hours. Based on our experiments, the HV, GD and SP quality indicator values for both NSGAII and NSGAII-DT start to stabilize and reach a plateau within the search time budget of 24h. Further, according to domain experts, longer search time budgets are not practical in the context of ADAS testing.

To build NSGAII-DT decision trees, we use the Classification and Regression Trees (CART) [8] algorithm. We control the decision tree size (depth) by setting the value of *minimum split parameter (msp)* to 10% of the size of the underlying data set. Our goal is thus to avoid overfitting and obtain reasonable estimates in ADAS critical regions captured by the tree leaves labeled critical. Moreover, we require that splitting a node reduces the miss-classification error of decision trees by at least 1%.

Within the 24h search time budget, NSGAII performed, on average, 22 search iterations (generations). Note that the NSGAII-DT algorithm consists of two nested loops: The outer loop that generates decision trees (lines 5–16 in Algorithm 1), and the inner loop that invokes NSGAII for critical input space regions (lines 6–10 in Algorithm 1). We refer to each iteration of the outer loop as *tree generation*. Corresponding to each tree generation, NSGAII is invoked one or more times depending on the number critical tree leaves. We set the number of search iterations performed by each NSGAII invocation to five (i.e., we set $g = 5$ in Algorithm 1). By setting $g = 5$, NSGAII-DT performed between five to seven *tree generations* in 24h (i.e., each run of NSGAII-DT generated between five to seven trees). Further, on average, NSGAII-DT performed 30 search iterations (i.e., NSGAII iterations) in 24h. Note that in our experiments, each run of NSGAII-DT performed more search iterations than each run of NSGAII. This is because, in our experiments and within the 24h search time budget, most search iterations of NSGAII-DT are applied to population sets smaller than the initial population set, while all search iterations of NSGAII are applied to a fixed-size population set equal to the size of the initial population.

We reran each of the NSGAII and NSGAII-DT algorithms for 20 times to account for their randomness. Due to lack of support for parallelizing the execution of the PreScan tool, we ran all the experiments on a single laptop with a 2.5 GHz CPU and 16GB of memory.

5.4 Results

RQ1. Figure 5 shows the HV, GD and SP values computed based on the outputs of NSGAII-DT and NSGAII. We show the results at every four-hour time interval starting at 2h as well as the results at the end of the search time limit (i.e., at 24h). Note that, on average, simulating the elements in the initial population takes about 2h. Hence, the results at 2h are those obtained for the randomly selected initial population and prior to any search iteration. As shown in the figure, the HV, GD and SP values for NSGAII-DT are consistently better than those for NSGAII. Further, after executing the algorithms for about 22h, both NSGAII-DT and NSGAII converge towards their respective Pareto optimal solutions (i.e., for each algorithm, the differences in HV, GD and SP average values between 22h and 24h are negligible).

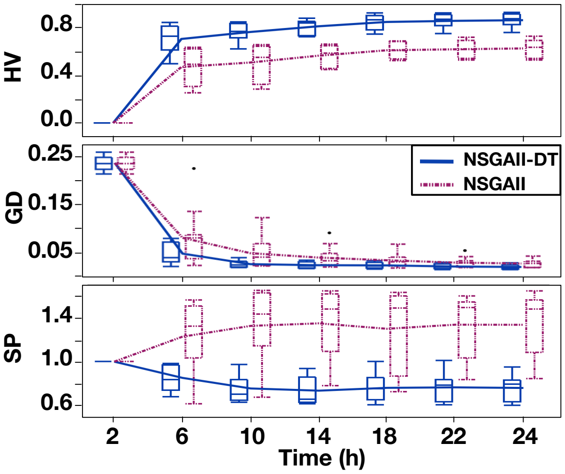

Figure 5: Comparing HV, GD and SP values obtained by NSGAII and NSGAII-DT.

Table 1: Statistical test results for NSGAII-DT and NSGAII at 24h

Algo.	Metric	p-value	$\hat{A}_{12}$
NSGAII-DT vs. *NSGAII*	HV	0.01	0.9
	GD	0.07	0.3
	SP	0.01	0.1

Following existing guidelines [2], to statistically compare HV, GD and SP values, we use the nonparametric pairwise Wilcoxon rank sum test [12] and the Vargha-Delaney's $\hat{A}_{12}$ effect size [32]. The level of significance (α) is set to 0.05. Table 1 reports the statistical test results comparing NSGAII-DT and NSGAII at 24h. For HV and SP comparisons, the p-values are less than 0.05, and the $\hat{A}_{12}$ values show large effect sizes. The differences between the GD distributions of NSGAII-DT and NSGAII are not statistically significant, although the effect size value is in the medium range. However, as shown in Figure 5, the medians and averages of the GD values obtained by NSGAII-DT are lower (i.e., better) than the medians and averages of the GD values obtained by NSGAII.

Finally, we evaluate the results of NSGAII-DT and NSGAII based on the number of *distinct, critical* test scenarios generated by each algorithm. Recall that an AEB test scenario is a vector in the AEB input space (i.e., a vector of values to four static and five dynamic variables). Also, an AEB test scenario is critical if its simulation outputs satisfy the *CB* property (equation (1) in Section 3). Two AEB test scenarios are distinct if they differ in the value of at least one static variable or in the value of at least one dynamic variable with a significant margin.

Over the 20 runs, NSGAII generates 708 distinct AEB test scenarios among which 411 are critical. In contrast, over the 20 runs, NSGAII-DT generates 1045 distinct AEB test scenarios among which 731 are critical. This result shows that, within the same search time budget, on average, NSGAII-DT provides 78% more

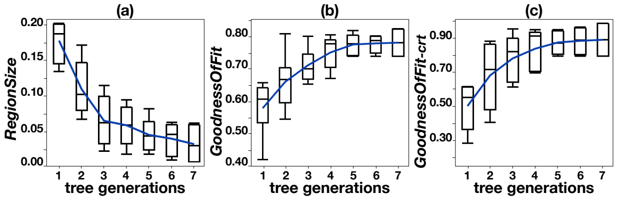

Figure 6: Evaluating the critical regions: (a) the *RegionSize*, (b) the *GoodnessOfFit*, and (c) the *GoodnessOfFit-crt* values.

distinct, critical test scenarios compared to NSGAII, enabling the engineers to better identify the limitations of AEB.

The answer to **RQ1** is that, NSGAII-DT significantly outperforms NSGAII. Further, on average, NSGAII-DT generates 78% more distinct, critical test scenarios compared to NSGAII.

RQ2. To answer this question, we focus on assessing the critical regions characterized by NSGAII-DT (i.e., the algorithm that is shown, in **RQ1**, to outperform NSGAII). We further note that NSGAII, or any search algorithm for that matter, has never been used to characterize critical regions and cannot be used as a baseline of comparison for this research question.

Figure 6(a) shows the *RegionSize* values computed for the critical regions obtained from the decision trees generated across different NSGAII-DT runs. Recall that NSGAII-DT performs five to seven *tree generations* within the 24h search time limit. In our experiments, each decision tree generated by NSGAII-DT had between one and three critical leaves (i.e., critical regions). As shown in the figure, the critical regions generated by NSGAII-DT become monotonically smaller (i.e., more specific) over successive tree generations. In particular, the critical regions obtained from the first decision trees are on average 17.2% of the size of the entire input space, while the final trees generated by NSGAII-DT are on average 3.5% of the input space.

Figures 6(b) and 6(c) show the *GoodnessOfFit* and the *GoodnessOfFit-crt* values for NSGAII-DT decision trees, respectively. As shown in the figure, *GoodnessOfFit* increases from 57% to 77%, and *GoodnessOfFit-crt* increases from 50% to 89% over the maximum seven tree generations of NSGAII-DT. These results show that the decision trees generated by NSGAII-DT, compared to those generated based on random initial populations, accurately classify on average 20% more critical and non-critical scenarios, and almost 40% more *critical* scenarios. Hence, NSGAII-DT, over its successive tree generations, produces decision trees that fit noticeably better to *critical* scenarios.

The answer to **RQ2** is that the *RegionSize*, *GoodnessOfFit* and *GoodnessOfFit-crt* values monotonically improve across different tree generations, confirming that the generated critical regions consistently become smaller, more homogeneous and more precise over successive tree generations of NSGAII-DT. In particular, the trees generated by NSGAII-DT, compared to those generated based on the initial randomly selected populations, fit on average to 40% more *critical* AEB test scenarios.

Benefits from a practitioner's perspective. Here, we investigate whether practitioners are able to use and benefit from our approach. In particular, we intend to know whether the critical regions computed by our approach are understandable, informative, and useful

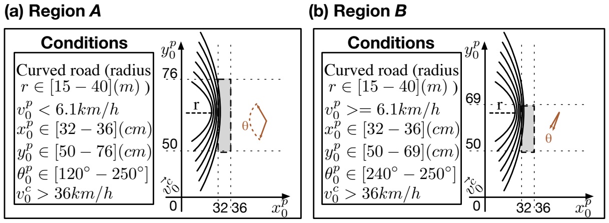

Figure 7: Examples of critical regions for the AEB case study

to practitioners. To do so, we draw on the qualitative reflections of two *semi-structured interview* [35] sessions that we conducted with three senior engineers at IEE. The reflections are based on the comments the engineers made in two two-hour meetings with the researchers. The engineers were selected from three different groups at IEE working on different aspects of ADAS development and testing. We have been collaborating with one of these engineers on the research and the case study presented in this paper. The two other engineers, however, did not have any interaction with the researchers prior to the interview sessions. Further, they were not involved in our research nor in the development of the AEB case study or any of our other ADAS examples.

To perform the interviews, we selected, among the decision trees generated by NSGAII-DT in our experiments, the one with the highest goodness of fit. The selected tree characterized three critical regions in the AEB input space. We created visually-enhanced representations of the three regions, showing the regions individually without any reference to the tree structure. Figures 7(a) and (b) illustrate the representations for two of the regions. The conditions specifying each region are shown on the left side of each region diagram. Furthermore, some of these conditions (i.e., those on the road type, and the initial position and orientation of the pedestrian) are visually shown on the right side of each diagram. Region **A** specifies the AEB input scenarios where a car (speed > 36.6km/h) drives on a curved road with a radius between 15m to 40m, while a pedestrian starts walking from a point inside the dashed gray rectangle with a trajectory between 120° and 250° and crosses the road with a low speed (< 6.1km/h). Region **B** specifies similar scenarios as those in **A** except that the pedestrian walks with a high speed (>= 6.1km/h) within a much narrower trajectory and starts crossing the road from a slightly smaller area compared to the one in **A**.

During the meetings, we presented the critical regions to the engineers, and asked the following questions: (1) Are you able to understand the conditions specifying the regions? (2) Based on your domain knowledge, do you think the regions specify situations where AEB is more likely to fail (i.e., exhibits *CB*)? (3) How can you utilize the knowledge you gain from the characterizations of the regions to analyze AEB? These questions aim to assess, respectively, comprehension, intuitiveness and usefulness of the critical region characterizations generated by our approach.

Regarding comprehension and intuitiveness, the engineers noted that the characterizations of the regions are understandable and consistent with their intuition. For example, regions **A** and **B** indicate that scenarios containing curved roads are more likely to exhibit *CB*. This is because, on such roads, pedestrians appear relatively late in the camera's field of view and will be detected late by AEB, hence leaving little time to apply the brake. The regions further show that the probability of *CB* is higher when the car speed is higher than 36.6km/h. Finally, the regions show that, in addition to the road and vehicle characteristics, *CB* likely happens due to pedestrian dynamics. Specifically, critical scenarios are more likely when pedestrians walk from particular areas on the sidewalk, or as shown in **B**, running pedestrians with a particular trajectory (θ) are more likely to escape accidents if they do not run towards the car.

Regarding the usefulness of our approach, the engineers noted that the information captured by these regions can help them in the following ways: (1) *Debugging the system or the simulator.* The region characterizations, particularly when they do not match the domain knowledge, may point to errors in the system or the simulator. For example, in our early results, curved roads did not appear as critical regions. Our investigation showed that due to an error in the AEB sensor output (i.e., the TTC output), which resulted in some scenarios that actually led to collision to be wrongly labeled as non-critical. Further, the pedestrian dynamic situations identified as critical may point to weaknesses in pedestrian tracking algorithms [27] typically used in ADAS. (2) *Identifying changes to hardware components to help increase ADAS safety.* For example, in our work, we assume AEB contains one camera located at the front center of the car with a specific value for its field of view. Regions **A** and **B** indicate that a different type of camera with a larger field of view or two cameras, although more expensive, may help detect pedestrians faster and better on curved roads. (3) *Identifying proper warnings to drivers.* Some of ADAS critical behaviors may not be avoidable due to real world and physical constraints. Nevertheless, our approach enables car makers to be aware of such situations and consider mitigation strategies. For example, regions **A** and **B** indicate that AEB may not be fully trusted on curved roads in residential zones where it is more likely for pedestrians to cross roads. In such situations, a warning message may be shown to drivers to reduce their speed (e.g., to lower than 30km/h).

5.5 Threats to validity

To mitigate the *Internal validity* risks caused by confounding factors, we compared NSGAII-DT and NSGAII under identical parameter settings. Further, we present a detailed formal description of our case study and search algorithm, and provide all the parameter settings to facilitate reproducibility. Our case study is a real ADAS. The simulation data is obtained based on an industrial and widely-used ADAS simulation tool. To assess usefulness of our approach, we conducted two semi-structured interview sessions with three engineers from different groups at IEE with varying types of expertise related to ADAS development.

Conclusion validity is related to random variations and inappropriate use of statistics. To mitigate these threats, we have followed standard guidelines in search-based software engineering [3] and ran the search algorithms 20 times. Further, we use the nonparametric Mann-Whitney U-test and Vargha and Delaney's $\hat{A}_{12}$ for statistical testing and effect sizes.

The main threat to *construct validity* concerns unsuitable or incorrect metrics. To compare multi-objective search algorithms we use standard quality indicators (i.e., HV, GD, SP). Further, we assess the decision trees generated by our approach using our formally defined *RegionSize* and the standard *GoodnessOfFit* metrics.

Regarding the *external validity* threats, we note that we provide in Section 3 a precise formalization of the ADAS systems to which our testing approach is applied. Our ADAS formalism builds on our experiences of studying ten ADAS systems as well as the characteristics of the PreScan simulator. Our testing approach applies to any ADAS system that conforms to our formalism presented in Section 3. Finally, we note that ADAS systems comprise an important and growing industry sector with pressing needs regarding testing and verification.

6 RELATED WORK

Search-based testing has received significant attention in recent years [17, 24]. However, due to the functional complexity of most real-world systems, it has been less applied to system testing and more often to unit testing [11, 24]. Search-based system testing has been previously used to automate test generation for ADAS [11]. For example, it has been applied to a vehicle-to-vehicle braking assistance [9], an autonomous parking [10] and a pedestrian detection system [5]. Bühler and Wegener [9, 10] base their testing on a single-objective search algorithm, while Ben Abdessalem et. al. [5] use a multi-objective search algorithm. In contrast to our work, none of these approaches consider (static) environment variables in the test input space, and they vary only mobile objects' variables in test scenarios. Hence, these approaches are not able to automatically explore different environment conditions (e.g., different road types and weather conditions). Further, the above-cited work focuses on identifying individual critical simulation scenarios only. In our work, we deal with a considerably larger test input space that includes environment variables. Further, we provide a novel search-based testing algorithm that, in addition to identifying individual critical scenarios, characterizes critical regions of the ADAS test input space. Finally, our formalism in Section 3 is able to capture the ADAS systems used by Bühler and Wegener [9, 10] and Ben Abdessalem et. al. [5] as they are among the ten ADAS used as a basis of our formalism.

In the context of machine learning, multi-objective optimization algorithms have been used to improve supervised learning techniques such as SVM, artificial neural networks and decision trees [26, 28] where the aim is to improve prediction accuracy of the resulting classifiers. Our work is related to software testing and uses decision trees to guide the search-based generation of tests faster towards the most critical regions.

Our NSGAII-DT algorithm can be seen as an instance of the *Learnable Evolution Model* algorithms [25]. These algorithms rely on machine learning, instead of the Darwinian genetic operators, to generate new populations (i.e., to guide evolutions). Our algorithm employs a combination of genetic operators and guidance through classifiers to evolve populations and is the first such algorithm applied to ADAS testing.

7 CONCLUSIONS

We proposed a simulation-based testing algorithm for vision-based control systems such as ADAS. Our algorithm builds on learnable evolution models and uses classification decision trees to guide the generation of new test scenarios within complex and multidimensional input spaces. Our approach is evaluated on an industrial ADAS. The results indicate that our classification-guided search algorithm outperforms a baseline evolutionary search algorithm and generates 78% more distinct, critical test scenarios compared to the baseline algorithm. Our approach, further, characterizes critical regions of the ADAS input space. Based on our interviews with domain experts, such characterizations are accurate and help engineers debug their systems. They further help engineers identify environment conditions that are likely to lead to ADAS failures as well as hardware changes that can increase ADAS safety.

REFERENCES

[1] Ethem Alpaydin. 2010. *Introduction to Machine Learning* (2nd ed.). MIT Press, Cambridge, Massachusetts, USA.

[2] Andrea Arcuri and Lionel Briand. 2014. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability* 24, 3 (2014), 219–250.

[3] Andrea Arcuri and Gordon Fraser. 2011. On Parameter Tuning in Search Based Software Engineering. In *Proceedings of the International Symposium on Search Based Software Engineering (SSBSE'11)*. Springer, Berlin, Heidelberg, 33–47.

[4] Assia Belbachir, Jean-Christophe Smal, Jean-Marc Blosseville, and Dominique Gruyer. 2012. Simulation-driven validation of advanced driving-assistance systems. *Procedia-Social and Behavioral Sciences* 48 (2012), 1205–1214.

[5] Raja Ben Abdessalem, Shiva Nejati, Lionel C. Briand, and Thomas Stifter. 2016. Testing advanced driver assistance systems using multi-objective search and neural networks. In *Proceedings of the International Conference on Automated Software Engineering (ASE'16)*. IEEE, Singapore, 63–74.

[6] Hans-georg Beyer and Kalyanmoy Deb. 2001. On self-adaptive features in real-parameter evolutionary algorithms. *IEEE Transactions on Evolutionary Computation* 5, 3 (2001), 250–270.

[7] Bosch. 2017. Driving safety systems for passenger cars. (Aug. 2017). Retrieved August 24, 2017 from https://goo.gl/4LSl3H

[8] Leo Breiman, Jerome Friedman, Charles J Stone, and Richard A Olshen. 1984. *Classification and regression trees*. CRC press.

[9] Oliver Buehler and Joachim Wegener. 2005. Evolutionary functional testing of a vehicle brake assistant system. In *Proceedings of the Metaheuristics International Conference (MIC'05)*. Vienna Austria, 157–162.

[10] Oliver Bühler and Joachim Wegener. 2004. *Automatic testing of an autonomous parking system using evolutionary computation*. Technical Report. SAE Technical Paper.

[11] Oliver Bühler and Joachim Wegener. 2008. Evolutionary functional testing. *Computers & Operations Research* 35, 10 (2008), 3144–3160.

[12] J. Anthony Capon. 1991. *Elementary Statistics for the Social Sciences: Study Guide*. Wadsworth Publishing Company.

[13] Kalyanmoy Deb and Ram Bhushan Agrawal. 1995. Simulated binary crossover for continuous search space. *Complex systems* 9, 2 (1995), 115–148.

[14] Kalyanmoy Deb and Hans-georg Beyer. 2001. Self-Adaptive Genetic Algorithms with Simulated Binary Crossover. *Evolutionary Computation* 9, 2 (2001), 197–221.

[15] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. 2002. A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation* 6, 2 (2002), 182–197.

[16] Object Management Group. 2017. Object Constraint Language (OCL). (Aug. 2017). Retrieved August 24, 2017 from http://www.omg.org/spec/OCL/

[17] Mark Harman, S. Afshin Mansouri, and Yuanyuan Zhang. 2012. Search-based software engineering: Trends, techniques and applications. *Comput. Surveys* 45, 1 (2012), 11.

[18] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, and Yves Le Traon. 2013. PLEDGE: a product line editor and test generation tool. In *Proceedings of the International Software Product Line Conference co-located workshops (SPLC'13)*. ACM, New York, NY, USA, 126–129.

[19] TASS International. 2017. PreScan simulation of ADAS and active safety. (Aug. 2017). Retrieved August 24, 2017 from https://www.tassinternational.com/prescan

[20] Joshua D. Knowles, Lothar Thiele, and Eckart Zitzler. 2006. *A Tutorial on the Performance Assessment of Stochastic Multiobjective Optimizers*. Technical Report. Computer Engineering and Networks Laboratory of Zurich.

[21] Philip Koopman and Michael Wagner. 2016. Challenges in autonomous vehicle testing and validation. *SAE International Journal of Transportation Safety* 4, 1 (2016), 15–24.

[22] D. Richard Kuhn, Dolores R. Wallace, and Albert M. Gallo. 2004. Software Fault Interactions and Implications for Software Testing. *IEEE Transactions on Software Engineering* 30, 6 (2004), 418–421.

[23] Sean Luke. 2013. *Essentials of Metaheuristics* (second ed.). Lulu, Fairfax, Virginie, USA. Available for free at http://cs.gmu.edu/~sean/book/metaheuristics/.

[24] Phil McMinn. 2004. Search-based software test data generation: A survey. *Software Testing Verification and Reliability* 14, 2 (2004), 105–156.

[25] Ryszard S Michalski. 2000. Learnable evolution model: Evolutionary processes guided by machine learning. *Machine learning* 38, 1 (2000), 9–40.

[26] Anirban Mukhopadhyay, Ujjwal Maulik, Sanghamitra Bandyopadhyay, and Carlos Artemio Coello Coello. 2014. A Survey of Multiobjective Evolutionary Algorithms for Data Mining: Part I. *IEEE Transactions on Evolutionary Computation* 18, 1 (2014), 4–19.

[27] Vasanth Philomin, Ramani Duraiswami, and Larry Davis. 2000. Pedestrian tracking from a moving vehicle. In *Proceedings of the IEEE Intelligent Vehicles Symposium (IV'2000)*. IEEE, Dearborn, MI, USA, 350–355.

[28] Thorsten Suttorp and Christian Igel. 2006. Multi-objective optimization of support vector machines. In *Multi-objective machine learning*. Springer, 199–220.

[29] Mugur Tatar. 2016. Test and Validation of Advanced Driver Assistance Systems Automated Search for Critical Scenarios. *ATZelektronik worldwide* 11, 1 (2016), 54–57.

[30] Richard van der Horst and Jeroen Hogema. 1993. Time-to-collision and collision avoidance systems. In *Proceedings of the workshop of the International Cooperation on Theories and Concepts in Traffic Safety (ICTCT'93)*. Salzburg, Austria, 109–121.

[31] David A. Van Veldhuizen and Gary B. Lamont. 1998. *Multiobjective evolutionary algorithm research: A history and analysis*. Technical Report. Air Force Institute of Technology.

[32] András Vargha and Harold D. Delaney. 2000. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132.

[33] Shuai Wang, Shaukat Ali, Tao Yue, Yan Li, and Marius Liaaen. 2016. A Practical Guide to Select Quality Indicators for Assessing Pareto-Based Search Algorithms in Search-Based Software Engineering. In *Proceedings of the International Conference on Software Engineering (ICSE'16)*. ACM, New York, NY, USA, 631–642.

[34] Ian H. Witten, Eibe Frank, and Mark A. Hall. 2011. *Data Mining: Practical Machine Learning Tools and Techniques* (3rd ed.). Morgan Kaufmann Publishers Inc.

[35] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in software engineering*. Springer Science & Business Media.

[36] Janusz Wojtusiak and Ryszard S Michalski. 2004. The LEM3 implementation of learnable evolution model: user's guide. In *Proceedings of the Machine Learning and Inference Laboratory, George Mason University, (MLI'04)*. Citeseer, Fairfax, Virginie, USA, 04–05.

[37] Andreas Zeller. 2017. Search-Based Testing and System Testing: A Marriage in Heaven. In *Proceedings of the International Workshop on Search-Based Software Testing (SBST'17)*. IEEE, Piscataway, NJ, USA, 49–50.

[38] Eckart Zitzler and Lothar Thiele. 1999. Multiobjective evolutionary algorithms: a comparative case study and the strength Pareto approach. *IEEE Transactions on Evolutionary Computation* 3, 4 (1999), 257–271.