Machine learning testing in an ADAS case study using simulation-integrated bio-inspired search-based testing

Mahshid Helali Moghadam¹,² | Markus Borg³ | Mehrdad Saadatmand¹ | Seyed Jalaleddin Mousavirad⁴ | Markus Bohlin² | Björn Lisper²

¹Smart Industrial Automation, RISE Research Institutes of Sweden, Västerås, Sweden
²School of Innovation, Design and Engineering, Mälardalen University, Västerås, Sweden
³Humanized Autonomy, RISE Research Institutes of Sweden, Lund, Sweden
⁴Universidade da Beira Interior, Covilhã, Portugal

Summary
This paper presents an extended version of Deeper, a search-based simulation-integrated test solution that generates failure-revealing test scenarios for testing a deep neural network-based lane-keeping system. In the newly proposed version, we utilize a new set of bio-inspired search algorithms, genetic algorithm (GA), \((\mu+\lambda)\) and \((\mu,\lambda)\) evolution strategies (ES), and particle swarm optimization (PSO), that leverage a quality population seed and domain-specific crossover and mutation operations tailored for the presentation model used for modeling the test scenarios. In order to demonstrate the capabilities of the new test generators within Deeper, we carry out an empirical evaluation and comparison with regard to the results of five participating tools in the cyber-physical systems testing competition at SBST 2021. Our evaluation shows the newly proposed test generators in Deeper not only represent a considerable improvement on the previous version but also prove to be effective and efficient in provoking a considerable number of diverse failure-revealing test scenarios for testing an ML-driven lane-keeping system. They can trigger several failures while promoting test scenario diversity, under a limited test time budget, high target failure severity, and strict speed limit constraints.

KEYWORDS
advanced driver assistance systems, deep learning, evolutionary computation, lane-keeping system, machine learning testing, search-based testing

1 | INTRODUCTION

Machine learning (ML) nowadays is used in a wide range of application areas such as automotive¹,² health care³, and manufacturing⁴. Many of the ML-driven systems in these domains present a high level of autonomy⁵ and meanwhile are subject to rigorous safety requirements⁶. In 2018, the European Commission (EC) published a strategy for trustworthy artificial intelligence (AI) systems⁷. In this strategy, AI systems are defined as “systems that display intelligent behavior by analyzing their environment and taking actions—with some degree of autonomy—to achieve specific goals.” The EC states that a trustworthy AI system must be lawful, ethical, and robust. Self-driving cars are examples of safety-critical AI systems, which leverage various ML techniques such as deep neural networks (DNN), machine vision, and sensor data fusion. Meanwhile, in the context of automotive software engineering, there is always a set of strict safety requirements to meet.
The quality assurance methodology for AI systems is quite different from the conventional software systems since the included ML components in those systems are not explicitly programmed, they are intended to learn from data and experience instead—called Software 2.0. In addition, in AI systems, a part of the requirements is mainly seen as encoded implicitly in the data, and the challenge of under-specificity is common in requirements definitions. However, it is still highly expected to assure the ability of the AI system to control the risk of hazardous events in particular in safety-critical domains.

Self-driving cars, as one of the pioneering examples of safety-critical AI systems, are one of the focus points for research studies on quality assurance of ML-driven systems. There is also a vigorous need for the integration verification and validation (V&V) of ML models that are deployed in self-driving cars to make sure that they are safe and dependable. Many of the failures basically emerge in the interplay between software containing ML components, hardware, and remote sensing devices, for example, sensors, cameras, RADAR, and LiDAR technologies. Hardware-in-the-loop (HIL), simulation-based, and field testing are common approaches for system-level verification of deployed ML models. System-level testing mainly targets defining a set of operational scenarios that could lead to failures. In this regard, in the ISO/PAS 21448 Safety of the Intended Function (SOTIF) standard—which addresses complementary aspects of functional safety in ISO 26262—simulation-based testing has been considered a proficient approach and a proper complementary solution to on-road testing. Testing on real-world roads is costly and does not scale to cover all the needed scenarios, and in addition, it is dangerous to create and execute critical scenarios. The use of virtual prototyping allows testing and verification at the early stages of development and offers the possibility of efficient and effective testing. It can capture the whole of the operational environment to a great extent using high-fidelity autonomous driving simulators.

Research challenge. In this study, we target an advanced driver-assistance system (ADAS) that provides lane-keeping assistance. Effective and efficient system-level testing in simulation environments requires sophisticated approaches to generate critical test scenarios. The critical test scenarios are those that break or are close to breaking the safety requirements of the ADAS under test, which hence result in safety violations. Generating effective test scenarios involves sampling from a large and complex set of test inputs. Several authors have shown the potential of search-based software test generation techniques to address this challenge. Various system-level testing techniques using different search-based testing approaches for different types of ADAS, relying on simulators, have been proposed in recent years.

Research contribution. In this paper, we present a bio-inspired computation-driven test generator, called Deeper, for effective and efficient generation of failure-revealing test scenarios to test a deep neural network (DNN)-based lane-keeping system in the BeamNG driving simulator. The test subject is BeamNG.AI, the built-in ML-driven driving agent in the BeamNG simulator. In this study, a failure is defined in terms of episodes in which the ego car—driven by the BeamNG.AI agent—drives partially outside the lane w.r.t a certain tolerance threshold. The tolerance threshold determines the percentage of the car’s bounding box needed to be outside the lane to be regarded as a failure.

Deeper in its current version in this paper benefits from the genetic algorithm (GA), \((\mu + \lambda)\) and \((\mu, \lambda)\) evolution strategies (ES), and the particle swarm optimization (PSO) to generate failure-revealing test scenarios, which are test roads in our study. The problem is basically regarded as an optimization problem, and in order to generate the test scenarios that are of interest, we evaluate the quality of the test scenarios using a fitness (objective) function that guides the search process to maximize the detected distance of the car from the center of the lane during driving of the car on the lane. The initial version of Deeper contained a test generator based on NSGA-II. In this paper, we extend Deeper with four additional test generators based on GA, \((\mu + \lambda)\) and \((\mu, \lambda)\) ESs, and PSO. In the newly proposed test generators, we leverage an initial quality population seed to boost the search process and also customize common crossover and mutation operations for the presentation model used for the test scenarios in the search algorithms—develop crossover and mutation operations tailored for the model. We rely on the presentation model used by DeepJanus based on Catmull-Rom cubic splines.

Empirical evaluation. In order to carry out an empirical evaluation, we use the setup provided by the cyber-physical systems (CPS) testing competition at the IEEE/ACM International Workshop on Search-Based Software Testing (SBST). Our experiments are designed to answer three main research questions which are as follows:

- How effective and efficient do the test generators perform? It has three parts, which are as follows: Given a certain test budget, (3.1) how many test scenarios are generated, (3.2) what proportion of the scenarios is valid, and finally, (3.3) what proportion of the valid test scenarios leads to triggering failures?

We provide a comparative analysis on the performance of the proposed bio-inspired test generators in Deeper and five counterpart tools all integrated into the BeamNG simulator. In this regard, we compare the results of the proposed test generators in Deeper with five other test generator tools, Frenetic, GABExploit and GABExplore, Swat, and also the earlier version of the Deeper (based on NSGA-II)—all participating tools in the CPS competition at SBST 2021. In order to do a fair comparison, we consider the same experimental evaluation procedures as the original CPS tool competition. Our experimental results show that first, the newly proposed test generators in Deeper present a considerable improvement on the previous version, and second, they perform as effective and efficient test generators that can provoke a considerable number.
of diverse failure-revealing test scenarios w.r.t different target failure severity (i.e., in terms of tolerance threshold), available test budget, and driving style constraints (e.g., setting speed limits). For instance, in terms of the number of triggered failures within a given test time budget and with less strict driving constraints, the \((\mu + \lambda)\) ES-driven test generator in Deeper considerably outperforms other tools while keeping the level of promoted failure diversity quite comparable to the counterpart tool with the highest number of detected failures in the competition. Then, with respect to the test effectiveness and efficiency in non-strict conditions, again Deeper \((\mu + \lambda)\) ES-driven test generator results in the highest effectiveness in terms of the ratio of the number of detected failures to the generated valid test scenarios, while PSO- and GA-driven are the next effective ones, which show performance comparable to the top counterparts. Meanwhile, as a distinctive feature, none of the newly proposed test generators leaves the experiment without triggering any failures, and in particular, they act as more reliable test generators than most of the other tools for provoking diverse failures under a limited test budget and strict constraints.

The rest of this paper is organized as follows: Section 2 presents background information on bio-inspired search techniques including evolutionary and swarm intelligence techniques and simulation-based ADAS testing. Section 3 presents the problem formulation and the technical details of our proposed test generators in Deeper. Section 4 elaborates on the empirical evaluation, including the research method and experiments setup. Section 5 discusses the results, answers to the RQs, and the threats to the validity of the results. Section 6 provides an overview of the related work, and finally, Section 7 concludes the paper with our findings and the potential research directions for future work.

2 | BACKGROUND

2.1 | Evolutionary and swarm intelligence

These algorithms are two main classes of random search techniques, which are widely used in many different optimization problems. Genetic algorithms (GA) and evolution strategies (ES) are two of the main categories inside the family of evolutionary algorithms (EAs). Particle swarm optimization (PSO) is one of the primary representatives of swarm intelligence algorithms.

Genetic algorithms are one of the most common nature-inspired optimization techniques. It starts with a random population of individuals—each called a chromosome—representing a potential solution for the problem. The objectives to be optimized in the problem are defined in an objective function and the quality of the solutions is measured via this function. It shows how “well” each solution satisfies the objective. The quality of each individual, which is also referred to as “fitness,” is a main factor during the evolution process. At each generation, a new population is formed based on the selected individuals from the previous generation. Three operations are involved in forming the new generation, which are as follows:

- Crossover and mutation operations are applied w.r.t user-set probabilities, and these two operations might be used, either independently or jointly, to create new individuals to form a new population. The resulting individuals are added to the new population. The fitness values are calculated and stored for each individual in this population. This process iterates each generation until stopping criteria are met; for example, a user-set number of generations or an allowed time budget is exhausted.

Evolution strategy is another common class of EAs. It is commonly used in almost all fields of optimization problems including discrete and continuous input spaces. ES also involves applying selection, recombination, and mutation to a population of individuals over various generations to get iteratively evolved solutions. Two canonical versions of ES are \((\mu/\rho + \lambda)\) and \((\mu/\rho, \lambda)\) evolution strategies. If \(\rho = 1\), we have ES cases without recombination, which are denoted by \((\mu + \lambda)\) and \((\mu, \lambda)\) ESs. In the case of \(\rho = 1\), the recombination is simply making a copy of the parent. In these notations, \(\lambda\) and \(\mu\) indicate the size of the offspring and population, respectively. The main difference between GA and ES is related to the selection step. In GA, at each iteration, the next generation is formed by selecting highly valued individuals, while the size of the population is kept fixed. In ES, a temporary population with the size of \(\lambda\) is created, and the individuals in this temporary population undergo crossover and mutation at user-set probabilities regardless of their fitness values. In \((\mu + \lambda)\) ES, then, both parents and the generated offspring resulting from the temporary population are copied to a selection pool—with size \((\mu + \lambda)\)—and a new population with size \(\mu\) is formed by selecting the best individuals, while, in \((\mu, \lambda)\) ES, the new generation with size \(\mu\) is selected only from the offspring (with size \(\lambda\)). Therefore, a convergence condition as \(\mu < \lambda\) is required to guarantee an optimal solution.

Particle swarm optimization is one of the most common representatives of swarm intelligence (SI) algorithms, which form a big class of nature-inspired optimization methods alongside evolutionary algorithms. The SI algorithms present the concept of collective intelligence, which is mainly defined as a collective behavior in a group of individuals that seem intelligent. SI algorithms have been inspired by collective behavior and self-organizing interactions between living agents in nature, for example, ant colonies and honey bees.

PSO is an optimization method simulating the collective behavior of certain types of living species. In PSO, cooperation is an important feature of the system as each of the individuals changes its searching pattern based on its own and others’ experiences. PSO starts with a swarm of random particles. Each particle has position and velocity vectors, which are updated w.r.t the local and global best values. The best values get updated at each iteration. In the application of PSO, each particle (individual) represents a potential solution and is often modeled as a vector containing \(n\) elements, in which each element represents a variable of the problem that is being optimized. Like GA, PSO searches for the optimal
solution through updating solutions and creating subsequent generations, though without using evolution operators. The position (the elements) and velocity of each particle are updated as follows:

\[
P_{t+1} = p_t + v_{t+1}
\]

\[
v_{t+1} = w v_t + c_1 r_1 (p_{t_{\text{best}}} - p_t) + c_2 r_2 (g_{t_{\text{best}}}/C_0 - p_t)
\]

where \( p_t \) and \( v_t \) are the position and velocity of the particle at iteration \( t \), respectively; \( p_{t_{\text{best}}} \) and \( g_{t_{\text{best}}}/C_0 \) are the local best position of the particle and the global best one up to the iteration \( t \). The first part of (2) is perceived as inertia, which indicates the tendency of the particle to keep moving in the same direction, while the second part—which reflects a cognitive behavior—indicates the tendency toward the local best position discovered by the particle and the last part—which is the social knowledge—reflects the tendency to follow the best position found so far by other particles.

Therefore, at each iteration, the position of each particle is updated based on its velocity, and the velocity is controlled by the inertia and accelerated stochastically toward the local and global best values. \( r_1 \) and \( r_2 \) are random weights from range \([0,1]\), which adjust the cognitive and social acceleration. In (2), \( w \) is inertia weight, which adjusts the ability of the swarm to change the direction and makes a balance between the level of exploration and exploitation in the search process. A lower \( w \) leads to more exploitation of the best solutions found, while a higher value of \( w \) facilitates more exploration around the found solutions. \( c_1 \) and \( c_2 \) are the acceleration hyperparameters defining to what extent the solutions are influenced by the local best solutions and global best solution. These hyperparameters and the inertia weight could be static or changed dynamically over the iterations. For instance, \( w = 0.72984 \) and \( c_1 + c_2 > 4 \) is a common setup for a static configuration of these parameters.

2.2 Simulation-based ADAS testing

System-level testing for systems with ML-driven components in safety-critical domains such as automotive industry introduces new challenges for system quality certification and verification. In this regard, new safety complementary standards and guides are always developed to cover the gaps and ensure the safety of ML components in those domains. Standards such as ISO 26262 Functional Safety \(^{11}\) and ISO 21448 Safety of the Intended Functionality \(^{10}\) often provide a high-level view of the requirements that are required to be satisfied in a safety case for an ML-driven system. However, how it shall be implemented and argued for being adequate and sufficient is highly up to the specific stakeholder organization. \(^{34}\) In this regard, Hawkins et al. \(^{8}\) introduced a methodology for the assurance of ML in autonomous systems, called AMLAS. It presents a systematic process for integrating safety assurance into the development of ML systems and introduces verification procedures at different stages, for example, learning model verification and system-level (integration) verification that happens after integrating the ML model into the system. Figure 1 provides an overview of the proposed AMLAS framework.

Assurance activities on ML safety requirements, data management, and model learning alongside the model verification and model deployment \(^{8,32}\) are different levels proposed by AMLAS. In this regard, model verification/testing and model deployment assurance—which is typically satisfied by integration and system-level testing—are considered different test levels for the quality assurance of ML systems. Various testing or formal verification techniques could be used to assure that the ML model satisfies the ML requirements. It is important to note that the ML testing is performed on the verification data set that is not involved and used in the training process in anyhow. \(^{8}\) Model deployment assurance
involves assuring the satisfaction of system safety requirements after the integration of the model into the system w.r.t. the specified operational design domain (ODD) for the system. Model testing could be regarded as unit testing for ML components and integration testing focuses on the issues emerging after the integration of the ML model into the system. Another view on testing ML systems is based on access to the test subject. In this regard, there are black-box and white-box testing approaches analogous to traditional non-ML systems. Black-box testing involves access only to the ML inputs and outputs, while white-box testing implies access to the internal architecture of the test ML subject, code, hyper-parameters, and training/test data. In this regard, Riccio et al. introduced another type of ML testing called data-box, which requires access to data plus everything that a black-box test requires.

The use of virtual testbeds, for example, simulation-based testing, has been considered a feasible, non-expensive, effective, and proper complement to field testing. In addition to the possibility of early-stage verification, virtual testing provides the possibility to create a huge number of test scenarios and examine autonomous driving functions in critical and unsafe situations. Moreover, the test could be reproducible and scalable. Recently, a growing number of commercial and open-source simulators have been developed to support the need for realistic simulation of self-driving cars—we refer interested readers to a review by Rosique et al. In this regard, for instance, various high-fidelity simulators such as the ones using physics-based models (e.g., SVL simulator, Pro-SiVIC, and PreScan) and the ones based on game engines (e.g., BeamNG. tech and CARLA) have considerably contributed to this area by providing the possibility of realistic simulations of functionalities in autonomous driving. Kaur et al. also provide a comparison of some of the common simulators w.r.t. some essential features expected from the autonomous driving simulators such as perception, multi-view geometry, path planning, 3D virtual environment, traffic infrastructure, and some other relevant features that can be crucial to assure the fidelity of the virtual environment to reality.

3 | DEEPER: A BIO-INSPIRED SIMULATION-INTEGRATED TESTING FRAMEWORK

This section presents the technical details of Deeper and shows how it challenges a DNN-based lane-keeping system in a self-driving car trained and tested in the BeamNG simulator environment. The subject system is a built-in AI driving agent encompassing a steering angle predictor (ML model) that receives images captured by an onboard camera in the simulation environment. Then, the test inputs (test cases) generated by Deeper are defined as scenarios in which the car drives. Our target is to generate diverse test scenarios triggering the misbehavior of the subject system. In this regard, we benefit from bio-inspired search-based techniques to explore the input space and generate highly-valued failure-revealing test scenarios.

3.1 | Test scenario and failure specification

Test scenarios are defined as combinations of roads, the environment including, for example, the weather and illumination, and the driving path, that is, starting and end points and the lane to keep. Hereafter, we consider scenarios involving a single asphalt road surrounded by green grass where the car is to drive on the right lane, and the environment is set to a clear day with the overhead sun (see Figure 2). Therefore, the focus of Deeper is to generate diverse roads that trigger failures in the system under test. In this system, failure is defined in terms of an episode, in which the car drives partially outside the lane meaning that \( X\% \) of the car's bounding box gets outside the lane. \( X \) is a configurable tolerance threshold for Deeper.

In the test scenarios, the road is composed of two fixed-width lanes with a yellow center line and two white lines separating the lanes from the non-drivable area. In BeamNG, each road is mainly described by a set of points that are used by the simulation engine to render the road. The simulation engine accomplishes the rendering by interpolating the points and creating a sequence of polygons on the points—provided by the SBST competition setup. Notably, not every sequence of road points results in valid roads, so each sequence of points is also validated against

FIGURE 2  A test scenario executed in the BeamNG simulator.
some initial geometrical constraints related to the road polygons and some other domain-specific constraints. The main constraints are as follows:

1. the start and end points of the road shall be different,
2. the road shall be completely contained in the map used in the simulation,
3. the road shall not self-intersect, and
4. the road shall not contain too sharp turns that force the vehicle to invade the opposite lane. To assure the satisfaction of these constraints, Deeper validates the generated roads before getting executed and consequently, the invalid roads are not counted as failed test scenarios.

Road representation model: In order to convert the abstract road model into a proper set of points that can be rendered by the simulation engine, we rely on the representation model used by DeepJanus based on Catmull-Rom cubic splines. The Catmull-Rom splines is a method that involves defining a series of points at intervals and a piecewise-defined function to calculate additional points within the intervals. The constructed curve will pass through the given set of points and feature a certain number of continuous derivatives. In Catmull-Rom interpolating splines, those given points defining the spline are called “control points.” Therefore, each road is represented by two sets of points, control points and sample points. First, control points are provided as input for a candidate road. Second, sample points are calculated using the Catmull-Rom calculation algorithm. Third, the simulation engine uses the sample points, if they are valid, to render the road. Figure 3 shows the representation model of a road in terms of control and sample points (Figure 3A,B) and the corresponding rendered road in the simulation (Figure 3C).

\[
CP = \{C_1, C_2, \ldots, C_m\}, \quad R_{\text{min}} \leq C_i \leq R_{\text{max}}
\]

\[
R_{\text{min}}, R_{\text{max}} \in \mathbb{R}
\]

\[
SP = \{S_1, S_2, \ldots, S_n\}, \quad SP = \text{Catmull}_\text{-Rom}_\text{Spline}(CP)
\]

\[
R_{\text{min}} \leq S_i \leq R_{\text{max}}
\]

\[
R_{\text{min}}, R_{\text{max}} \in \mathbb{R}
\]

3.2 Fitness function

As indicated above, the focus is the generation of diverse failure-revealing test scenarios w.r.t the intended tolerance threshold. The competition setup elaborates the positional data and detects the episodes in which the car breaks out of the lane bounds, that is, out-of-bound episodes (OBE). It computes the distance of the car from the center of the lane in those OBE episodes and reports an OBE failure each time that the car drives outside the lane if the percentage of the area of the car that is outside the lane is bigger than the intended threshold (referred to as \(X\) in Section 3.1).

The problem is regarded as an optimization problem, and in order to generate valuable test scenarios leading us to meet the target, we evaluate the quality of the test scenarios using a fitness (objective) function. In this regard, for each test scenario, the main objective of interest to be optimized is the maximum detected distance of the car from the center of the lane during driving of the car on the lane. So, more accurately the fitness function that we want to minimize is as follows:

![Figure 3](image-url)
where \(\text{lane}_\text{width}\) is the width of the lane and \(d(\text{spin}_\text{polyline},\text{car}_\text{position})\) indicates the distance of the car position from the central polyline (center) of the lane.

### 3.3 | Bio-inspired search algorithms

We are interested in sampling from the space of possible test scenarios in an effective way to generate those that lead to the emergence of failures. This can be achieved by using an optimization algorithm to guide the search by the fitness function. Therefore, in order to find the solutions of interest, we use evolutionary and swarm intelligence algorithms, that is, GA, \((\mu + \lambda)\) and \((\mu, \lambda)\) ESs, and PSO, guided by the fitness function introduced in Section 3.2. In Deeper, the roads—as the main focus of the test scenarios—are evolved through changes applied to the control points by the bio-inspired algorithms. Any changes in the set of control points do not lead to creating a valid road. Once a new set of control points is formed, the simulation engine performs the road rendering by first interpolating the points, then checking the satisfaction of the geometrical and domain-specific conditions.

The search algorithms are mainly modeled on the basis of population evolution over time and they usually get started with the creation of a random population of solutions. In Deeper, we leverage an initial quality population seed to boost the search process regarding the fact that the search is done at a fixed test budget. Throughout the development, the impact of different initial population seeds was investigated. For instance, starting from an initial random population seed was not quite effective to lead the search to find the failure-revealing solutions w.r.t high tolerance thresholds within a reasonable test budget. We created a quality population seed mainly based on the use of an improved random road generation technique, which besides tweaking the control points, changes the geometrical parameters of the roads such as the maximum allowed angle and the length of road segments as well. We formed 90% of the quality population seed through the generated set of roads from the improved random generation technique. Then, in order to augment the population seed, we filled the remaining 10% of the population with solutions randomly extracted from a prior standalone 5-h execution of Deeper NSGA-II, which had started from a random population seed—generated through just tuning the control points. For this latter part, the generated solutions that could cause OBEs w.r.t a tolerance threshold \(\tau \geq 0.5\) were only considered. We use this quality population seed for running Deeper NSGA-II in the empirical evaluation as well. Basically, the same population seed is used for running all the test generators involved in Deeper.

#### 3.3.1 | Genetic algorithm

The GA-driven test generator in Deeper starts with forming an initial population by sampling from the quality population seed. Over various generations, new populations of test scenarios are formed through applying crossover and mutation operations to the best ones selected from the previous generations.

**Selection:** We use tournament selection for identifying the promising test scenarios, which have a high probability to lead to failures and safety violations. In tournament selection, a subset of the population is selected at random in each tournament and the best test scenario of the subset is picked. The number of individuals participating in each tournament indicates the size of the tournament.

**Crossover:** We develop a domain-specific one-point crossover operation tailor-made to our representation model. The proposed crossover operation performs the segment exchange at the level of control points, which means that a random point is selected as the crossover point in the sets of control points in the parent roads, then the parts of the sets beyond the crossover point are swapped between the parents, and accordingly, two new sets of control points for two child roads are formed. The corresponding sample points for the generated child roads are calculated using the Catmull-Rom calculation algorithm (see Figure 4).

However, still, these resulting sets of points might not represent valid roads w.r.t the geometrical constraints, so before adding these new test scenarios to the offspring, we also check their validity and let only the resulting valid roads be added to the offspring. If both generated child roads are valid, then both of them will be transferred to the offspring, while if one of them is valid, we keep the valid child and another crossover point is tried to breed the parent roads and generate the second valid child. Likewise, if none of the child roads are identified as valid roads, another crossover point is tried. All in all, in order to generate two valid child roads from a crossover operation, at most, five attempts to try with different crossover points are done. If in the end, the attempts do not lead to valid child roads, the whole process will be rolled back and the original parents will be added to the offspring.

**Mutation:** The mutation operation also targets the coordinate values of the control points. We use polynomial bounded mutation, a bounded mutation operation for real-valued individuals which was used in NSGA-II.\(^{21}\) It features using a polynomial function for the probability distribution and a user-set parameter, \(\eta\), presenting the crowding degree of the mutation and adjusting the diversity in the resulting mutant. A high value for \(\eta\) results in a mutant resembling the original solution, while a low \(\eta\) leads to a more divergent mutant from the original. This domain-specific
polynomial bounded mutation operation selects randomly a point—mutation point—in the set of control points and mutates randomly the x or y coordinate of the selected control point (see Figure 5). Accordingly, the sample points are re-calculated for the mutated set of control points, and their validity w.r.t. the geometrical constraints are checked. In case the mutant does not represent a valid road, another control point is tried.

It is also worth noting that the rates of the crossover and mutation have been set empirically and with regard to the characteristics of individuals in the problem. According to our empirical assessment, due to the existing inherent similarity between the individuals, the crossover operation could not lead to a big impact. Basically, since the test roads, as the individuals, are subject to geometrical and domain-specific constraints, the valid generated roads over the generations do not show a high level of variance commonly, so the resulting roads from the crossover operation are very likely to be either similar to the parents or not valid at all. Then, in this regard, the mutation can take a more prominent role and have
a better impact, and based on these observations, we set the crossover and mutation rates to 0.3 and 0.7, respectively. The GA-driven test generator of Deeper is configured as presented in Algorithm 1.

**Algorithm 1** GA-driven test generator in Deeper

1. Initialize a population of test scenarios (with size = 70) from the quality population seed
2. Evaluate the test scenarios through rendering them in the simulation and computing the fitness values
   
   repeat
   
   3. Select highly-valued scenarios using tournament selection (Tournament size = 3)
   4. Create offspring by using crossover and mutation operations
      
      4.1. Apply the domain-specific crossover operation (Crossover rate = 0.3)
      4.2. Apply the domain-specific polynomial Bounded mutation operation (Mutation rate = 0.7)
   5. Evaluate the offspring
   until reaching the end of the test budget (e.g., given time)
6. Collect the test scenarios revealing OBE failures

3.3.2 | Evolution strategies

The ES-driven test generators in Deeper use two canonical \((μ + λ)\) and \((μ, λ)\) ES algorithms. Algorithm 2 presents the procedure of \((μ + λ)\) ES-driven test generator, where \(μ = 70\) and \(λ = 30\). The procedure of \((μ, λ)\) ES test generator is almost the same as \((μ + λ)\) ES, and the main difference is in the selection step, where the \(μ\) highly valued individuals are selected only from the offspring in \((μ, λ)\) ES. However, with regard to the required convergence condition \((μ < λ)\) in this case, then those parameters in \((μ, λ)\) ES test generator are configured as \(μ = 70\) and \(λ = 100\) (the size of the main population is fixed at both ES-driven test generators).

Moreover in order to draw a better distinction between GA and ES algorithms in detail, it is worth noting that in Deeper, both GA and ES use the same size for the main (original) population. However, in ES, we have a temporary population with size \(λ\), which forms the offspring—it is the pool where the mutation and crossover operations are applied, while in GA, the original population (with size \(μ\)) is used to form the offspring. Additionally, the way the mutation and crossover operations are applied to the offspring’s individuals are different. For instance, in GA, each individual of the offspring will be considered to undergo both mutation and crossover operations; that is, both mutation and crossover will be separately tried for each individual, and their occurrences will be based on the user-defined probabilities. While, in our ES-based algorithms, either crossover or mutation is applied to each individual of the offspring; that is, each offspring member is given one chance, and it would undergo either crossover or mutation, though in ES-based algorithms, mutation takes a more prominent role.

**Algorithm 2** \((μ + λ)\) ES-driven test generator in Deeper

1. Initialize a population \(P\) with size \(μ\) of test scenarios sampled from the quality population seed \(μ = 70\)
2. Evaluate the test scenarios in the population \(P\) through simulation and computing the fitness values
   
   repeat
   
   3. Create a temporary population \(P_F\) with size \(λ\) by reproduction of test scenarios from the population \(P\)
   4. Create an offspring by applying the crossover or mutation operation to the test scenarios in the population \(P_F\) (with crossover probability \(C_{xp} = 0.3\) and mutation probability \(M_{up} = 0.7\))
      
      4.1. choice = Random.random(), \(0 < \text{choice} < 1\)
      4.2. If \(\text{choice} < C_{xp}\) then
          
          Apply the domain-specific crossover operation
          
          else if \(\text{choice} < C_{xp} + M_{up}\)
          
          Apply the domain-specific polynomial Bounded mutation operation
      5. Evaluate the offspring
   6. Select \(μ\) highly-valued test scenarios using tournament selection (Tournament size = 3) from the original population \(P\) and the offspring
   until reaching the end of the test budget (e.g., given time)
7. Collect the test scenarios revealing OBE failures
3.3.3 | Particle swarm optimization

The PSO-driven test generator in Deeper starts with initializing a population of particles from the quality population seed. Each test scenario is modeled as a particle, and the set of control points represents the position vector of the particle that is updated according to Equations (1) and (2) over various generations. After each updating, the corresponding sample points for the updated set of control points are calculated and the validity of the new set of road points is checked against the geometrical constraints. The PSO-driven test generator is configured as presented in Algorithm 3 and based on the following setting for the parameters, which is \( w = 0.8, c_1 = 2.0, \) and \( c_2 = 2.0. \)

Algorithm 3 PSO-driven test generator in Deeper

1. Initialize a swarm of test scenario particles (with size = 70) from the quality population seed
2. Evaluate the particles of the swarm through simulation and computing the fitness values
3. Select the global best particle w.r.t the fitness value \((G_{best})\)

repeat
    for each test scenario particle P in the swarm do
        4. Calculate the particle’s velocity according to eq. (2)
        5. Update particle’s position according to eq. (1)
        6. Evaluate the particle based on the fitness function
        if fitness value of P is better than the local best of P, \((P_{best})\), then
            7. Update \(P_{best}\) with \(P\)
        end if
        if fitness value of P is better than the global best, \((G_{best})\), then
            8. Update \(G_{best}\) with \(P\)
        end if
    end for
until reaching the end of the test budget (e.g., given time)
9. Collect the test scenario particles revealing OBE failures

4 | EMPIRICAL EVALUATION

We conduct an empirical evaluation of the proposed simulation-integrated bio-inspired test generators in Deeper, by running experiments on an experimental setup based on a PC with 64-bit Windows 10 Pro, Intel Core i7-8550U CPU @ 1.80GHz, 16GB RAM, Intel UHD Graphics 620, and BeamNG.tech driving simulator together with the software requirements for running Deeper.

Test subject: The system under test is BeamNG’s built-in driving agent, BeamNG.AI. It is an autonomous agent utilizing optimization techniques to plan the driving trajectory according to the speed limit while keeping the ego car inside the road lane. It is equipped with a DNN-based lane-keeping ADAS. The DNN-based lane-keeping system learns a mapping from the input of the onboard camera in the simulated environment to the steering angle. It is based on the DAVE-2 architecture including a normalization layer, five convolutional layers followed by three fully connected layers. This test subject has been used in previous research and also in the SBST 2021 cyber-physical tool competition for evaluating test scenario generators and moreover does not require manual training, which can mitigate the threats to the validity of the results.

4.1 | Research method

We design and implement a set of experiments to answer the research questions:

1. RQ1: How capable are these test generators to trigger failures?
2. RQ2: How diverse are the generated failure-revealing test scenarios?
3. RQ3: How effectively and efficiently do the test generators perform? That is, given a certain test budget, (3.1) how many test scenarios are generated, (3.2) what proportion of the scenarios is valid, and finally (3.3) what proportion of the valid test scenarios leads to triggering failures?
The experiments are simulation scenarios generated by a Python test scenario generator and executed by the simulation engine. BeamNG.AI is the autonomous driving agent controlling the ego car in the simulation (Figure 6). In order to provide quantitative answers to the RQs, we use the following quality criteria to assess the bio-inspired test scenario generators in Deeper:

- Detected failures: The number of generated test scenarios that lead to failures, w.r.t the given tolerance threshold.
- Failure diversity: The dissimilarity between the test scenarios that lead to the failures. Generating diverse failure-revealing test scenarios is of interest, since triggering the same failures multiple times results in wasting the test budget, for example, computation resources. In order to measure the failure diversity, we rely on a two-step strategy adopted by the SBST 2021 tool competition. It extracts, first, the road segments related to the failures and then computes the sparseness, which is considered as the average of the maximum Levenshtein distance between those road segments.

The failure-related road segments are referred to as the parts of the road 30 m before the OBE and 30 m after it, and accordingly, the sparseness is calculated as follows:

\[ \text{Sparseness} = \frac{\sum_{i \in \text{OBEs}} \max_{j \in \text{OBEs}} \text{Lev}_\text{dist}(i,j)}{|\text{OBEs}|} \]  

where \( \text{Lev}_\text{dist}(i,j) \) indicates the weighted Levenshtein distance between the road segments.
- Test generation efficiency and effectiveness: It indicates how the test generator uses the given test budget to generate the test scenarios; in particular, how many test scenarios are generated in total, what fraction of them are valid, and what fraction of the valid ones triggers failures.

**Experiments:** We design two sets of experiments as implemented in the SBST 2021 CPS testing tool competition. In order to provide a comparative analysis, we compare the results of the proposed test generators in Deeper with the presented test generators in the tool competition, that is, Frenetic, GABExploit and GABExplore, Swat, and also the earlier version of Deeper. We run the test generators on the test subject based on the same two experiment configurations as in the competition, which are shown in Table 1. The SET1 of experiments provides a 5-h test generation budget, meanwhile sets the failing tolerance threshold to a high value, 0.95, and does not consider any speed limit. This experiment configuration might lead to a more careless style of driving. The SET2 of the experiments allocates a shorter time budget for the test generation.

**Figure 6** An overview of the experimental setup.

**Table 1** Experiment configurations.

| Name                      | Test budget (h) | Map size (m²) | Speed limit (Km/h) | Failing tolerance threshold (%) |
|---------------------------|-----------------|---------------|--------------------|---------------------------------|
| SET1 (careless driving)   | 5h              | 200 × 200     | None               | 0.95                            |
| SET2 (cautious driving)   | 2h              | 200 × 200     | 70 km/h            | 0.85                            |
and considers a lower tolerance threshold, 0.85, while imposing a speed limit of 70 km/h—promoting a more careful driving style. To ensure a fair comparison, we run each tool the same number of times; we run each test generator five times in the experiment configuration SET1 and 10 times in SET2 and report distributions of the results. It is to be noted that due to the long execution time of each simulation run and more importantly in order to make a fair comparison between the newly proposed test generators and the counterpart tools—w.r.t addressing the same problem, under the same evaluation conditions, and for answering the same research questions—the number of runs for each tool in the experiment configurations has been tied to the setup used in SBST Tool Competition 2021 and the results for the counterpart tools in the empirical evaluation also rely on the published results by the competition.

Furthermore, in order to analyze and provide a better insight into the statistics of the results, we run the Kruskal-Wallis test\(^4\) on the results in each category, that is, number of detected failures, sparseness, and effectiveness plus. Here, for each analysis, we have nine groups of data, each of which represents a test generator. Kruskal-Wallis is a non-parametric test and an extension of the Mann-Whitney U test\(^5\) (also known as Wilcoxon rank-sum test) for comparing more than two groups of data—to evaluate the hypothesis that the populations, from which the groups are sampled, are equivalent. This implies that none of the group populations is dominant over the others. A group is considered dominant if it has a higher probability of containing the largest element when one element is randomly selected from each of the group populations. The Kruskal-Wallis test is essentially used when the sizes of the data groups are small and no assumption about the underlying data distribution is made. Kruskal-Wallis test reports the test statistic, H statistic, and the corresponding \(p\)-value.

Kruskal-Wallis test detects a significant difference between the data groups overall but does not state which group is different. Then, a pairwise comparison can be utilized to identify the specific differences between the data groups. Pairwise Mann-Whitney U test is one of the appropriate follow-up pairwise tests to be done after the Kruskal-Wallis test. It does not rely on assumptions about the population parameters, for example, normality or equal variances, either. It is based on the ranks of the data rather than the actual values. In the Mann-Whitney U test, the null hypothesis is that there is no big difference between the two intended data groups and the distributions of the data groups are equivalent. The test is performed by ranking the observations from lowest to highest, combining the ranks from both groups, and calculating the sum of the ranks per each group. The Mann-Whitney U test reports the U statistics corresponding with each data group and the \(p\)-value accordingly. The U statistics are the sums of the ranks of the observations in each group, and the smaller U statistic is chosen as the Mann-Whitney test statistic and is used for calculating the \(p\)-value. Here, the associated significance level (\(\alpha\)) is chosen as \(\alpha = 0.05\). A smaller \(p\)-value indicates stronger evidence against the null hypothesis, so if the \(p\)-value is less than or equal to the significance level, the null hypothesis is rejected; otherwise, the test fails to reject it.

### 5 | RESULTS AND DISCUSSION

This section reports results corresponding to the three RQs.

#### 5.1 | Detected failures (RQ1)

In order to gain a statistical overview of the results related to the number of detected failures (Figure 7 and Table 2), first we run a Kruskal-Wallis statistical test on the groups of collected data from the test generators (see Table 3). The \(p\)-values resulting from the Kruskal-Wallis test on SET1 and SET2 indicate significantly different behavior—in terms of the number of triggered failures—between the test generators in both experiment configurations. Moreover, in pursuance of achieving a deeper view of the difference between the test tools w.r.t number of detected failures, we perform the Mann-Whitney U test to show the pair-wise comparison between the test generators. According to the Mann-Whitney test results—shown in Table 4—in a normal experimental condition such as SET1, Deeper \((\mu + \lambda)\) ES clearly behaves differently from all other tools (Table 4a).

Meanwhile, the test does not prove a considerable difference in the comparison of Deeper PSO- and GA-driven test generators with either Frenetic or GABExploit—the top performers in the competition. In particular, the further boxplots (see Figure 7a) for Deeper PSO and GA actually show comparable performance—according to the median of the values—to Frenetic. Regarding the behavior of the test tools in SET2—under strict conditions—we see that all the new test generators, Deeper PSO, \((\mu + \lambda), (\mu, \lambda),\) and GA, show totally different behavior compared to Swat, GABExplore, and Deeper NSGA-II. Meanwhile, regarding Deeper PSO, \((\mu + \lambda),\) and GA, no considerably different behavior compared to either of Frenetic and GABExploit has been proved according to the Mann-Whitney test (see Table 4b). Similarly, the boxplots in Figure 7b, w.r.t the median of the number of triggered failures, for Deeper PSO and GA, show quite analogous performance to Frenetic.

Along with the statistical tests, boxplots(Figure 7) draw a picture of the distribution of the number of failures triggered by the test tools through indicating minimum, first quartile (Q1), median, third quartile (Q3), and maximum of the result data. Figure 7a reports the number of triggered failures by each of the tools in experiment configuration SET1. In this regard, as shown in Figure 7a, \((\mu + \lambda)\) ES-driven test generator in Deeper could successfully trigger at least \(2 \times\) more OBEs than the highest record—held by GABExploit—this significantly different behavior has been also indicated by the results of the statistical test. At the same time, Deeper \((\mu + \lambda)\) ES showed more consistent performance over the runs,
that is, with lower standard deviation, compared to GABExploit, which revealed highly varying behavior across the runs (e.g., returning over 100 OBEs in some runs, but failed to trigger any failure in other runs). More importantly, the PSO- and GA-driven approaches were able to trigger the failures in which the ego car invades the opposite lane of the road—a type of failure that has been typically considered difficult to trigger in most of the test generator tools.

Similarly, Figure 7B presents the number of triggered failures in experiment configuration SET2. With regard to the limited time budget, the speed limit of 70 km/h and the tolerance threshold 0.85, in contrast to Deeper NSGA-II, GABExplore, GABExploit, and Swat, none of the newly proposed test generators in Deeper left an experiment without triggering any failure or with a very low number of failures (i.e., less than 3). It means that they are able to detect failures even within a limited test budget and strict constraints such as setting a speed limit (see Table 2). It is noted that Deeper NSGA-II, GABExplore, and Swat triggered just equal or less than 1 failure in a considerable number of experiments in SET2.

| Test Generator | Exp.1 | Exp.2 | Exp.3 | Exp.4 | Exp.5 | Exp.6 | Exp.7 | Exp.8 | Exp.9 | Exp.10 |
|----------------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| Deeper_NSGA-II | 0     | 0     | 0     | 1     | 1     | 0     | 0     | 0     | 3     | 0     |
| Frenetic       | 12    | 8     | 9     | 11    | 19    | 10    | 15    | 20    | 6     | 23    |
| GABExplore     | 0     | 1     | 1     | 0     | 1     | 3     | 0     | 0     | 3     | 0     |
| GABExploit     | 84    | 18    | 126   | 38    | 0     | 11    | 47    | 0     | 28    | 5     |
| Swat           | 0     | 0     | 0     | 0     | 5     | 1     | 1     | 2     | 1     | 2     |
| Deeper_PSO     | 54    | 8     | 13    | 13    | 10    | 9     | 10    | 15    | 18    | 12    |
| Deeper_{μ,λ} ES | 8     | 4     | 3     | 5     | 7     | 7     | 3     | 5     | 7     | 9     |
| Deeper_{μ + λ} ES | 29    | 5     | 9     | 4     | 4     | 7     | 19    | 3     | 14    | 5     |
| Deeper_GA      | 11    | 7     | 11    | 15    | 14    | 9     | 25    | 9     | 14    | 20    |

Note: The colored cells show the “low” performance of the tools.

| Test Generator | Exp.1 | Exp.2 | Exp.3 | Exp.4 | Exp.5 | Exp.6 | Exp.7 | Exp.8 | Exp.9 | Exp.10 |
|----------------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| Deeper_NSGA-II | 0     | 0     | 0     | 1     | 1     | 0     | 0     | 0     | 3     | 0     |
| Frenetic       | 12    | 8     | 9     | 11    | 19    | 10    | 15    | 20    | 6     | 23    |
| GABExplore     | 0     | 1     | 1     | 0     | 1     | 3     | 0     | 0     | 3     | 0     |
| GABExploit     | 84    | 18    | 126   | 38    | 0     | 11    | 47    | 0     | 28    | 5     |
| Swat           | 0     | 0     | 0     | 0     | 5     | 1     | 1     | 2     | 1     | 2     |
| Deeper_PSO     | 54    | 8     | 13    | 13    | 10    | 9     | 10    | 15    | 18    | 12    |
| Deeper_{μ,λ} ES | 8     | 4     | 3     | 5     | 7     | 7     | 3     | 5     | 7     | 9     |
| Deeper_{μ + λ} ES | 29    | 5     | 9     | 4     | 4     | 7     | 19    | 3     | 14    | 5     |
| Deeper_GA      | 11    | 7     | 11    | 15    | 14    | 9     | 25    | 9     | 14    | 20    |

Note: The colored cells show the “low” performance of the tools.
TABLE 4  Mann-Whitney U test on the detected failures in SET1 and SET2.

| Group1          | Group2          | U1  | U2  | p-value |
|-----------------|-----------------|-----|-----|---------|
| **(a) SET1**    |                 |     |     |         |
| Deeper_PSO      | Frenetic        | 13.0| 12.0| 1.0     |
| Deeper_PSO      | GABExplore      | 25.0| 0.0 | 0.0079  |
| Deeper_PSO      | GABExploit      | 17.0| 8.0 | 0.4206  |
| Deeper_PSO      | Swat            | 25.0| 0.0 | 0.0079  |
| Deeper_PSO      | Deeper_NSGAII   | 25.0| 0.0 | 0.0079  |
| Deeper_PSO      | Deeper_(μ + λ)  | 0.0 | 25.0| 0.0079  |
| Deeper_PSO      | Deeper_(μ,λ)    | 25.0| 0.0 | 0.0079  |
| Deeper_PSO      | Deeper_GA       | 9.0 | 16.0| 0.5476  |
| Deeper_(μ + λ)  | Frenetic        | 25.0| 0.0 | 0.0079  |
| Deeper_(μ + λ)  | GABExplore      | 25.0| 0.0 | 0.0079  |
| Deeper_(μ + λ)  | GABExploit      | 25.0| 0.0 | 0.0079  |
| Deeper_(μ + λ)  | Swat            | 25.0| 0.0 | 0.0079  |
| Deeper_(μ + λ)  | Deeper_NSGAII   | 25.0| 0.0 | 0.0079  |
| Deeper_(μ + λ)  | Deeper_(μ,λ)    | 25.0| 0.0 | 0.0079  |
| Deeper_(μ + λ)  | Deeper_GA       | 8.5 | 16.5| 0.5476  |
| Deeper_(μ,λ)    | Swat            | 25.0| 0.0 | 0.0079  |
| Deeper_(μ,λ)    | Deeper_NSGAII   | 22.5| 2.5 | 0.0556  |
| Deeper_(μ,λ)    | Deeper_GA       | 5.0 | 20.0| 0.1508  |
| Deeper_GA       | Frenetic        | 20.0| 5.0 | 0.1508  |
| Deeper_GA       | GABExplore      | 24.0| 1.0 | 0.0159  |
| Deeper_GA       | GABExploit      | 14.0| 11.0| 0.8413  |
| Deeper_GA       | Swat            | 24.0| 1.0 | 0.0159  |
| Deeper_GA       | Deeper_NSGAII   | 24.0| 1.0 | 0.0159  |
| Deeper_NSGAII   | Frenetic        | 0.0 | 25.0| 0.0079  |
| Deeper_NSGAII   | GABExplore      | 8.5 | 16.5| 0.5476  |
| Deeper_NSGAII   | GABExploit      | 4.5 | 20.5| 0.1508  |
| Deeper_NSGAII   | Swat            | 7.5 | 17.5| 0.4206  |
| Frenetic        | GABExplore      | 25.0| 0.0 | 0.0079  |
| Frenetic        | GABExploit      | 15.0| 10.0| 0.6905  |
| Frenetic        | Swat            | 25.0| 0.0 | 0.0079  |
| GABExplore      | GABExploit      | 5.0 | 20.0| 0.1508  |
| GABExplore      | Swat            | 11.5| 13.5| 1.0     |
| GABExploit      | Swat            | 20.0| 5.0 | 0.1508  |
| **(b) SET2**    |                 |     |     |         |
| Deeper_PSO      | Frenetic        | 52.0| 48.0| 0.9118  |
| Deeper_PSO      | GABExplore      | 100.0| 0.0| 0.0     |
| Deeper_PSO      | GABExploit      | 40.5| 59.5| 0.5288  |
| Deeper_PSO      | Swat            | 100.0| 0.0| 0.0     |
| Deeper_PSO      | Deeper_NSGAII   | 100.0| 0.0| 0.0     |
| Deeper_PSO      | Deeper_(μ + λ)  | 73.5| 26.5| 0.0892  |
| Deeper_PSO      | Deeper_(μ,λ)    | 98.0| 2.0 | 0.0     |
| Deeper_PSO      | Deeper_GA       | 49.5| 50.5| 1.0     |
Besides the wide range of number of detected failures by GABExploit, the PSO- and GA-driven test generators result in a comparable number of detected failures to Frenetic. Additionally, in the experiment configuration SET2, the GA-driven test generator was also able to trigger the failures showing the invasion of the car to the opposite lane of the road.

5.2 Diversity of failures (RQ2)

Tables 3 and 5 present the results of the Kruskal-Wallis and pair-wise Mann-Whitney U statistical tests on the data showing the promoted failure diversity by the test generators (Figure 8) respectively. It is important to note that in the statistical tests on the sparseness data, the tools that did not report sparseness value in more than 20% of the runs—due to the very low number of triggered failures, that is, zero or one—were excluded from the tests, since the basic assumptions for the possibility of doing the test would be distorted. In particular, Deeper NSGA-II and GABExplore were excluded in the statistical analysis of sparseness data in SET1, and in addition to these tools, Swat was also kept out in the statistical tests on the sparseness data from SET2. As shown by the Mann-Whitney U test on the sparseness data from SET1 (see Table 5a), Frenetic and Swat perform differently from the rest of the tools and notable differences between the behavior of the Deeper test generators and either Frenetic or Swat have been reported by the test. However, regarding the comparison between the new test generators in Deeper and GABExploit, the test does not prove a significant difference in their behavior. Later, the boxplots (see Figure 8A) actually, for new Deeper test generators, show comparable performance, w.r.t the median of the sparseness values, to GABExploit.
In strict experimental conditions (SET2), the Mann-Whitney test (see Table 5b) indicates that Frenetic show different behavior in terms of promoted diversity from the Deeper test generators. It is also worth noting that regarding SET2, all the new Deeper test generators are included in the statistical analysis, since they all triggered enough failures in all runs. Though, among the rest of the tools, only Frenetic and GABExploit were eligible, and the other ones were excluded due to the fact that in a considerable number of runs under SET2, they did not trigger enough failures to calculate the sparseness.

| Group1          | Group2          | U1  | U2  | p-value |
|-----------------|-----------------|-----|-----|---------|
| (a) SET1        |                 |     |     |         |
| Deeper_PSO      | Frenetic        | 0.0 | 25.0| 0.0079  |
| Deeper_PSO      | GABExploit      | 15.0| 5.0 | 0.2857  |
| Deeper_PSO      | Swat            | 0.0 | 20.0| 0.0159  |
| Deeper_PSO      | Deeper_\((\mu + \lambda)\) | 14.0| 11.0| 0.8413  |
| Deeper_PSO      | Deeper_\((\mu, \lambda)\) | 16.5| 8.5 | 0.5476  |
| Deeper_PSO      | Deeper_GA       | 12.0| 13.0| 1.0     |
| Deeper_\((\mu + \lambda)\) | Frenetic        | 0.0 | 25.0| 0.0079  |
| Deeper_\((\mu + \lambda)\) | GABExploit      | 14.0| 6.0 | 0.4127  |
| Deeper_\((\mu + \lambda)\) | Swat            | 0.0 | 20.0| 0.0159  |
| Deeper_\((\mu + \lambda)\) | Deeper_GA       | 10.5| 14.5| 0.8413  |
| Deeper_\((\mu, \lambda)\) | Frenetic        | 0.0 | 25.0| 0.0079  |
| Deeper_\((\mu, \lambda)\) | GABExploit      | 12.0| 8.0 | 0.7302  |
| Deeper_\((\mu, \lambda)\) | Swat            | 0.0 | 20.0| 0.0159  |
| Deeper_\((\mu, \lambda)\) | Deeper_GA       | 8.0 | 17.0| 0.4206  |
| Deeper_GA       | Frenetic        | 0.0 | 25.0| 0.0079  |
| Deeper_GA       | GABExploit      | 16.0| 4.0 | 0.1905  |
| Deeper_GA       | Swat            | 0.0 | 20.0| 0.0159  |
| Frenetic        | GABExploit      | 20.0| 0.0 | 0.0159  |
| Frenetic        | Swat            | 20.0| 0.0 | 0.0159  |
| GABExploit      | Swat            | 0.0 | 16.0| 0.0286  |
| (b) SET2        |                 |     |     |         |
| Deeper_PSO      | Frenetic        | 0.0 | 100.0| 0.0    |
| Deeper_PSO      | GABExploit      | 34.0| 46.0| 0.6334  |
| Deeper_PSO      | Deeper_\((\mu + \lambda)\) | 53.0| 47.0| 0.8534  |
| Deeper_PSO      | Deeper_\((\mu, \lambda)\) | 72.0| 28.0| 0.1051  |
| Deeper_PSO      | Deeper_GA       | 46.0| 54.0| 0.7959  |
| Deeper_\((\mu + \lambda)\) | Frenetic        | 0.0 | 100.0| 0.0    |
| Deeper_\((\mu + \lambda)\) | GABExploit      | 34.0| 46.0| 0.6334  |
| Deeper_\((\mu + \lambda)\) | Deeper_\((\mu, \lambda)\) | 68.5| 31.5| 0.1903  |
| Deeper_\((\mu + \lambda)\) | Deeper_GA       | 44.0| 56.0| 0.6842  |
| Deeper_\((\mu, \lambda)\) | Frenetic        | 0.0 | 100.0| 0.0    |
| Deeper_\((\mu, \lambda)\) | GABExploit      | 25.0| 55.0| 0.2031  |
| Deeper_\((\mu, \lambda)\) | Deeper_GA       | 26.0| 74.0| 0.0753  |
| Deeper_GA       | Frenetic        | 0.0 | 100.0| 0.0    |
| Deeper_GA       | GABExploit      | 38.0| 42.0| 0.8968  |
| Frenetic        | GABExploit      | 70.0| 10.0| 0.0062  |

Note: The shaded rows (colored in grey) highlight the proposed test tools that have shown significantly different behavior.
Figure 8A depicts the diversity of the detected failures in SET1 in terms of the distribution of the failures’ sparseness for all the test tools—including the ones that reported the sparseness in a limited number of runs, that is, GABExplore and Deeper NSGA-II. It is noted that GABExplore and Deeper NSGA-II in 40% of the runs under SET1 did not report any sparseness figure, since they just triggered one or zero failure in each run. According to Figure 8A, all the new test generators in Deeper result in a considerable improvement in failure sparseness compared to the first version, Deeper NSGA-II, under normal experimental conditions (SET1). The newly proposed test generators also show comparable performance to GABExploit. Figure 8B shows the distribution of the failures’ sparseness in SET2. In the limited test budget and strict driving constraints, all the new test generators again show a big improvement in promoting failure sparseness in comparison to the first version, Deeper NSGA-II, under SET2. In the meantime, Deeper PSO, $(\mu + \lambda)$, and GA promote comparable levels of failure diversity to GABExploit—though in a more consistent way. At the same time, GABExplore and Swat in around 70% of the experiments in SET2 did not provide any sparseness figure, since they triggered just one or zero failure (see Table 2).

### 5.3 Test effectiveness and efficiency (RQ3)

In the SBST competition, test effectiveness and efficiency were indicated by how many test scenarios are generated and what proportion of the scenarios is valid, given a certain test budget. They basically show how well the test generator is able to utilize the test budget. Figure 9A,B report
the average number of total test scenarios, as well as the number of valid and invalid scenarios generated by each tool in SET1 and SET2, respectively. Generally, the new test generators in Deeper utilize the test budget more efficiently than the competition tools and generate a higher number of test scenarios within the given test time. In this regard, Deeper PSO results in the highest efficiency (e.g., generates more than 650 scenarios on average within 5 h) among all the test generators in both experimental configurations. Regarding the number of valid test scenarios, all the Deeper test generators along with Swat lead to an almost comparable number of valid scenarios. However, with respect to the ratio of the valid test scenarios to the total generated ones—called test effectiveness according to the competition evaluation—Swat, Deeper NSGA-II, and GABExploit are the ones showing the highest result.

To answer RQ3, in addition to the metrics defined and used by the competition, we define an additional metric, an aggregated test effectiveness metric called effectiveness plus, which indicates what proportion of the valid test scenarios leads to triggering failures. It is defined as the ratio of the triggered failures to the number of valid test scenarios and is intended to present the effectiveness of the test generators w.r.t meeting the target—detecting failures. We conduct Kruskal-Wallis and pair-wise Mann-Whitney U statistical tests on the result data showing effectiveness plus as well (see Tables 3 and 6). According to the results of the Mann-Whitney U test on the data, in non-strict experimental conditions (SET1), Deeper ($\mu + \lambda$) ES-driven test generator shows a significantly different performance than other tools; meanwhile, Deeper PSO and GA also prove a majorly different behavior than GABExploit, Swat, and Deeper NSGA-II (see Table 6a). Additionally, the Mann-Whitney U test does not prove a significant difference for the comparison of DeeperPSO or GA with either Frenetic or GABExploit. In SET2, the statistical test shows that GABExploit performs differently from others (see Table 6b), while again regarding the comparison of DeeperPSO, GA, and Deeper ($\mu + \lambda$) ES with Frenetic, no big difference has been proved by the Mann-Whitney U test.

Figure 10A,B report the test effectiveness plus for the test generators in SET1 and SET2, respectively. In SET1, Deeper ($\mu + \lambda$) ES-driven test generator results in the highest target-based effectiveness, and then Deeper GA, PSO, GABExploit, and Frenetic are the next effective tools. In this regard, DeeperPSO and GA also show a very comparable behavior to GABExploit. In SET2, GABExploit shows the highest effectiveness plus, while Frenetic, Deeper PSO, and GA—with comparable performance—are the next ones leading to the highest ratio of triggered failures to the number of valid test scenarios.

5.4 Discussion

The conclusion from the experimental results for the number of detected failures presents the following remarks about the performance of the test generators: The similarity between Frenetic and Deeper GA in terms of the number of detected failures and also resulting effectiveness plus comes from the common use of GA in both test generators. However, they have used quite different representation models in the formulation of the problem. Frenetic uses curvature values to represent roads and augments the test generation with a model-based heuristic to promote a higher level of road diversity.

Besides Frenetic and Deeper GA, GABExploit, and GABExplore also use GA, as the evolutionary mechanism, for the generation of test scenarios. They use a parametric approach to represent the roads; that is, the roads are presented as parametric curves, Bezier Curves, based on a set of control points. GABExploit uses a randomly generated seed population over the whole search, while GABExplore restarts the search with a new random seed population, once it finds a failure-revealing test scenario. Generally, GABExploit shows better performance in terms of the number of detected failures and also effectiveness plus compared to GABExplore, while GABExplore could promote failure diversity, given enough test time budget, which can be expected due to renewing the seed population throughout the search process. In comparison to other tools, due to the common use of GA again, GABExploit shows almost similar behavior, w.r.t all aspects, to Deeper GA in non-strict experimental conditions (e.g., SET1). Though, in strict experimental configuration, it led to better performance, in terms of the number of triggered failures and effectiveness plus, which could be due to the specific representation model, and accordingly the customization of the genetic operations used in GABExploit.

Furthermore, excluding the impact of additional heuristics for diversity promotion, PSO as a swarm intelligence optimization algorithm in Deeper PSO also leads to fairly comparable performance to the GA-driven ones, in particular, Deeper GA, GABExploit—w.r.t different aspects—in this problem, though it still shows a more consistent behavior due to its specific continual evolution mechanism. The PSO-driven test generator never left an experiment without triggering any failure, while it was not valid for the GA-based tools, such as GABExploit. Deeper PSO shows quite similar behavior in terms of the number of detected failures and effectiveness plus to the GA-based tools, in particular, Deeper GA and Frenetic. However, due to the lack of an extra diversity-promoting heuristic, it did not result in triggering a very high level of failure diversity.

Finally, regarding the role of evolutionary strategies, it is worth noting that due to the pivotal difference between GA and ES in the selection process (Section 3.3.2), and the impact of this difference within a non-restricted time budget, Deeper ($\mu + \lambda$) ES could outperform all other test tools in terms of the number of detected failures and effectiveness plus, once it gave enough time. While in a restricted test time budget, it still works well and shows comparable performance to the PSO-based generator and most of the GA-based tools such as Frenetic and Deeper GA.
TABLE 6  Mann-Whitney U test on the test generation effectiveness plus in SET1 and SET2.

| Group1                              | Group2                              | U1     | U2     | p-value |
|-------------------------------------|-------------------------------------|--------|--------|---------|
| (a) SET1                            |                                     |        |        |         |
| Deeper_PSO                          | Frenetic                            | 12.0   | 13.0   | 1.0     |
| Deeper_PSO                          | GABExplore                          | 25.0   | 0.0    | 0.0079  |
| Deeper_PSO                          | GABExploit                          | 17.0   | 8.0    | 0.4206  |
| Deeper_PSO                          | Swat                                | 25.0   | 0.0    | 0.0079  |
| Deeper_PSO                          | Deeper_NSGAII                       | 25.0   | 0.0    | 0.0079  |
| Deeper_PSO                          | Deeper_(μ+λ)                        | 25.0   | 0.0    | 0.0079  |
| Deeper_PSO                          | Deeper_(μ,λ)                        | 0.0    | 25.0   | 0.0079  |
| Deeper_PSO                          | Deeper_GA                           | 9.0    | 16.0   | 0.5476  |
| Deeper_(μ+λ)                        | Frenetic                            | 25.0   | 0.0    | 0.0079  |
| Deeper_(μ+λ)                        | GABExplore                          | 25.0   | 0.0    | 0.0079  |
| Deeper_(μ+λ)                        | GABExploit                          | 25.0   | 0.0    | 0.0079  |
| Deeper_(μ+λ)                        | Swat                                | 25.0   | 0.0    | 0.0079  |
| Deeper_(μ+λ)                        | Deeper_NSGAII                       | 25.0   | 0.0    | 0.0079  |
| Deeper_(μ+λ)                        | Deeper_(μ,λ)                        | 25.0   | 0.0    | 0.0079  |
| Deeper_(μ+λ)                        | Deeper_GA                           | 5.0    | 20.0   | 0.1508  |
| Deeper_(μ,λ)                        | Frenetic                            | 22.0   | 3.0    | 0.0056  |
| Deeper_(μ,λ)                        | GABExplore                          | 7.0    | 18.0   | 0.3095  |
| Deeper_(μ,λ)                        | Swat                                | 25.0   | 0.0    | 0.0079  |
| Deeper_(μ,λ)                        | Deeper_NSGAII                       | 22.0   | 3.0    | 0.0056  |
| Deeper_(μ,λ)                        | Deeper_GA                           | 5.0    | 20.0   | 0.1508  |
| Deeper_(μ,λ)                        | Frenetic                            | 5.0    | 20.0   | 0.1508  |
| Deeper_GA                           | GABExplore                          | 15.0   | 10.0   | 0.6905  |
| Deeper_GA                           | Swat                                | 24.0   | 1.0    | 0.0159  |
| Deeper_GA                           | Deeper_NSGAII                       | 24.0   | 1.0    | 0.0159  |
| Deeper_NSGAII                       | Frenetic                            | 0.0    | 25.0   | 0.0079  |
| Deeper_NSGAII                       | GABExplore                          | 7.0    | 18.0   | 0.3095  |
| Deeper_NSGAII                       | GABExploit                          | 4.0    | 21.0   | 0.0952  |
| Deeper_NSGAII                       | Swat                                | 6.0    | 19.0   | 0.2222  |
| Frenetic                            | GABExplore                          | 25.0   | 0.0    | 0.0079  |
| Frenetic                            | GABExploit                          | 15.0   | 10.0   | 0.6905  |
| Frenetic                            | Swat                                | 25.0   | 0.0    | 0.0079  |
| GABExplore                          | GABExploit                          | 5.0    | 20.0   | 0.1508  |
| GABExplore                          | Swat                                | 14.0   | 11.0   | 0.8413  |
| GABExploit                          | Swat                                | 20.0   | 5.0    | 0.1508  |
| (b) SET2                            |                                     |        |        |         |
| Deeper_PSO                          | Frenetic                            | 41.0   | 59.0   | 0.5288  |
| Deeper_PSO                          | GABExplore                          | 100.0  | 0.0    | 0.0     |
| Deeper_PSO                          | GABExploit                          | 36.0   | 64.0   | 0.315   |
| Deeper_PSO                          | Swat                                | 100.0  | 0.0    | 0.0     |
| Deeper_PSO                          | Deeper_NSGAII                       | 100.0  | 0.0    | 0.0     |
| Deeper_PSO                          | Deeper_(μ+λ)                        | 74.0   | 26.0   | 0.0753  |
| Deeper_PSO                          | Deeper_(μ,λ)                        | 94.5   | 5.5    | 0.0003  |
| Deeper_PSO                          | Deeper_GA                           | 44.0   | 56.0   | 0.6842  |
| Method                  | Set         | Effectiveness Plus |
|------------------------|-------------|--------------------|
| Deeper\(_{\mu + \lambda}\) | Frenetic    | 24.0               |
|                        | GABExplore  | 99.0               |
|                        | GABExploit  | 27.0               |
|                        | Swat        | 95.0               |
|                        | Deeper\(_{\mu + \lambda}\) | 100.0              |
|                        | Deeper\(_{\mu, \lambda}\) | 59.5               |
|                        | Deeper\(_{\mu, \lambda}\) | 6.0                |
|                        | Deeper\(_{\mu, \lambda}\) | 96.0               |
|                        | Deeper\(_{\mu, \lambda}\) | 23.0               |
|                        | Deeper\(_{\mu, \lambda}\) | 95.0               |
|                        | Deeper\(_{\mu, \lambda}\) | 98.0               |
|                        | Deeper\(_{\mu, \lambda}\) | 5.0                |
|                        | Deeper\(_{\mu, \lambda}\) | 44.5               |
|                        | Deeper\(_{\mu, \lambda}\) | 100.0              |
|                        | Deeper\(_{\mu, \lambda}\) | 35.0               |
|                        | Deeper\(_{\mu, \lambda}\) | 100.0              |
|                        | Deeper\(_{\mu, \lambda}\) | 100.0              |
|                        | Deeper\(_{\mu, \lambda}\) | 0.0                |
|                        | Deeper\(_{\mu, \lambda}\) | 100.0              |
|                        | Deeper\(_{\mu, \lambda}\) | 35.5               |
|                        | Deeper\(_{\mu, \lambda}\) | 13.0               |
|                        | Deeper\(_{\mu, \lambda}\) | 33.0               |
|                        | Deeper\(_{\mu, \lambda}\) | 36.0               |
|                        | Deeper\(_{\mu, \lambda}\) | 100.0              |
|                        | Deeper\(_{\mu, \lambda}\) | 100.0              |
|                        | Deeper\(_{\mu, \lambda}\) | 84.0               |

**Note:** The shaded rows (colored in grey) highlight the proposed test tools that have shown significantly different behavior.

**FIGURE 10** Test generation effectiveness plus in SET1 and SET2.
At the end, in addition to the number of detected failures and effectiveness, and concerning the failure diversity, it is worth remarking that in order to promote a considerably high level of failure diversity, it is essential to augment the search process with an additional diversity-promoting heuristic, for example, based on the representation model. Otherwise, given the use of the same representation model, almost all these swarm and evolutionary algorithms lead to comparable levels of failure diversity.

5.5 | Threats to validity

The evaluation of Deeper comes with a set of threats to construct, internal, and external validity of the results.

5.5.1 | Construct validity

The choices of the fitness function—the distance of the car position from the center of the lane—and also the metrics used for calculating the sparseness and indicating the diversity of the test scenarios (weighted Levenshtein distance) in this study are domain-specific. However, we have based our choices on the sound metrics adopted by other research works in the literature. Moreover, we also used the same algorithm settings, for example, population size, crossover, and mutation probabilities, in the proposed test generators to alleviate the potential non-comparability between the algorithms.

5.5.2 | Internal validity

The randomized nature of the used bio-inspired algorithms could be a source of threats to the internal validity of the results. In this regard, we tried to follow the guidelines given by Arcuri and Briand for the evaluation and analysis of the results and mitigate this threat by reporting the distribution and statistics of the results, for example, using Box plots to show the results, conducting non-parametric statistical tests, that is, Kruskal-Wallis and pairwise Mann-Whitney U tests, on different parts of the results, that is, detected failures, promoted diversity, and the resulting effectiveness. Furthermore, the limited number of runs for each tool in the experimental configurations—due to the long execution time and also being compliant with the applied experimental settings at the SBST (2021) tool competition—is another source of threat to the internal validity of the results. However, in order to mitigate the effects of this potential threat, we have applied non-parametric statistical tests, that is, Kruskal-Wallis and Mann-Whitney U tests, which are appropriate for the situations in which the sizes of the data groups are small and no assumption about the underlying data distribution is available.

5.5.3 | External validity

The choice of the test subject system is a potential threat to the external validity of the results. However, the test system in this study is one of the commonly used systems in self-driving cars, and furthermore, different ML models with various quality levels (i.e., different accuracy levels) could be deployed within the BeamNG.AI agent, and in this regard, the proposed test scenario generation techniques can still be used. Nonetheless, it still offers one type of DL-based systems in self-driving cars, and further studies are required to address the testing of other DL-driven systems; meanwhile, we also keep the tool open for extensions, for example, to support the execution of test scenarios in other simulators.

6 | RELATED WORK

Test input data that can reveal failures in the test subject is the most common generated test artifact in the literature related to testing of automotive AI systems. Depending on the test level and the test subject, the inputs could be images, for instance, as used in DeepTest, or test scenario configurations as used in Ebadi et al. A brief overview of the most common techniques used to generate the test data is as follows.

6.1 | Input data mutation

This type of mutation involves generating new inputs based on the transformation of the existing input data. For instance, DeepXplore uses such input transformations to find the inputs triggering different behaviors between similar autonomous driving DNN models, while also striving
to increase the level of neuron coverage. Moreover, in many studies, those transformations are based on metamorphic relations. DeepTest\textsuperscript{50} applies different transformations to a set of seed images with the aim of increasing neuron coverage and uses metamorphic relations to find the erroneous behaviors of different Udacity DNN models for self-driving cars. DeepRoad\textsuperscript{52} uses a GAN-based metamorphic testing technique to generate input images to test three autonomous driving Udacity DNN models. It defines the metamorphic relations such that the driving behavior in a new synthesized driving scene is expected to be consistent with the one in the corresponding original scene.

### 6.2 Test scenario manipulation

Another major category of the methods to generate test input data is based on the manipulation and augmentation of the test scenarios. Most of the works in this category use search-based techniques to go through the search space of the scenarios to find the failure-revealing or collision-provoking test scenarios. In this regard, simulators as a form of digital twins have played a key role to generate and capture those critical failure-revealing test scenarios. Simulation-based testing can act as an effective complementary solution to field testing since exhaustive field testing is expensive, meanwhile inefficient, and even dangerous, in some cases.

Accordingly, various testing approaches relying on the simulators have been presented in the literature, and in this regard, search-based techniques have been frequently used to address the generation of failure-revealing test scenarios. Abdessalem et al. utilize multi-objective search algorithms such as NSGA-II\textsuperscript{53} along with surrogate models to find critical test scenarios with fewer simulations and then at less computation time for a pedestrian detection system. In a following study,\textsuperscript{14} they use MOSA\textsuperscript{54}—a many-objective optimization search algorithm—along with objectives based on the branch coverage and some failure-based heuristics to detect undesired and failure-revealing feature interaction scenarios for integration testing in a self-driving car. Further, in another study,\textsuperscript{55} they leverage a combination of multi-objective optimization algorithms (NSGA-II) and decision tree classifier models—referred to as a learnable evolutionary algorithm—to guide the search-based process of generating critical test scenarios and also to refine a classification model that can characterize the failure-prone regions of the test input space for a pedestrian detection and emergency braking system. Haq et al\textsuperscript{56} use many-objective search algorithms to generate critical test data resulting in severe mispredictions for a facial key-points detection system in the automotive domain. Ebadi et al\textsuperscript{57} benefit from GA along with a flexible data structure to model the test scenarios and a safety-based heuristic for defining the objective function to test the pedestrian detection and emergency braking system of the Baidu Apollo (an autonomous driving platform) within the SVL simulator.

Regarding the impacts of the simulators in this area, Haq et al\textsuperscript{57} provide a comparison between the results of testing DNN-based ADAS using online and offline testing. Their results clearly motivate an increased focus on online testing as it can identify faults that never would be detected in offline settings—whereas the opposite does not appear to be likely. Our current study responds to this call, and motivates our work on systems testing in simulated environments. With regard to a different perspective, Borg et al\textsuperscript{58} discuss the consistency between the test results obtained from running the same experiments based on two different simulators and investigate the reproducibility of the results in both simulators. When running the same testing campaign in PreScan and ESI Pro-SiVIC, the authors found notable differences in the test outputs related to revealed safety violations and the dynamics of cars and pedestrians.

Furthermore, with regard to the use of BeamNG.AI as the test subject and BeamNG.Tech as the simulation environment in ML testing, in particular, ADAS testing, Nguyen et al\textsuperscript{37} propose a test generator that uses accident sketches and with the help of image processing identifies the road segments, lane markings, and vehicle positions and generates the corresponding test scenarios in BeamNG simulator. Birchler et al\textsuperscript{48} develop a test scenario selection (SDC-Scissor) that uses ML techniques, that is, logistic regression, random forest, and naive Bayes, to train ML models to identify safe and unsafe test scenarios before their execution. In their study, they also use lane keeping system in BeamNG.AI as the test subject and employ AsFault\textsuperscript{40} as a GA-based test tool to generate the simulation-based test scenarios. In another study, they use single and multi-objective GA to create a test prioritization technique that can prioritize the test scenarios before their execution.\textsuperscript{59} Similar to the previous studies, the lane keeping system in BeamNG is used as the test subject, the experimentation is done within the BeamNG simulator, and the scenarios are generated by using AsFault and the ML-based SDC-Scissor test scenario selection. Generally, in this regard, besides genetic algorithms, which are very commonly used in this area, utilizing evolution strategies and particle swarm optimization to generate failure-revealing test scenarios is a quite novel study.

## 7 Conclusion

Deeperin its extended version utilizes a set of bio-inspired algorithms, genetic algorithm (GA), \((\mu+\lambda)\) and \((\mu,\lambda)\) evolution strategies (ES), and particle swarm optimization (PSO), to generate failure-revealing test scenarios for testing a DL-based lane-keeping system. The test subject is an AI agent in BeamNG.tech’s driving simulator. The extended Deeper contains four new bio-inspired test generators that leverage a quality seed population and domain-specific crossover and mutation operations tailored for the presentation model used for modeling the test scenarios. Failures are defined as episodes where the ego car drives partially out of the lane w.r.t a certain tolerance threshold. In our empirical evaluation, we
focused to answer three main questions: First, how many failures the test generators can detect, second, how much diversity they can promote in the failure-revealing test scenarios, and third, how effectively and efficiently they can perform, w.r.t different target failure severity (i.e., in terms of tolerance threshold), available test budget, and driving style constraints (e.g., speed limits). Our results show that the newly proposed test generators in Deeper present a considerable improvement on the previous version and they are able to act as effective and efficient test generators that provoke a considerable number of diverse failure-revealing test scenarios for testing an ML-driven lane-keeping system. They show considerable effectiveness in meeting the target, that is, detecting diverse failures, with respect to different target failures intended and constraints imposed. In particular, they act as more reliable test generators than most of the counterpart tools for provoking diverse failures within a limited test budget and with respect to strict constraints.

As some directions for future work, we plan to apply the proposed approaches to testing further types of ML-based lane-keeping systems, that is, more industrial ones and also in other state-of-the-art simulation platforms. We also plan to extend the approaches by applying machine learning-based techniques such as reinforcement learning or generative adversarial networks (GANs) for empowering the discovery of failure-revealing test scenarios.

ACKNOWLEDGMENTS

This work has been funded by Vinnova through the ITEA3 European IVVES (https://itea3.org/project/ivves.html) and H2020-ECSEL European AIDOaRT (https://www.aidoart.eu/) and InSecTT (https://www.insectt.eu/) projects. Furthermore, the project received partially financial support from the SMILE III project financed by Vinnova, FFI, Fordonsstrategisk forskning och innovation under the grant number: 2019-05871.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are available from the corresponding author upon reasonable request.

ENDNOTES

* https://documentation.beamng.com/tutorials/ai/.
† https://github.com/se2p/tool-competition-av.git.
‡ see requirements at https://github.com/mahshidhelali/Deeper_ADAS_Test_Generator.git

REFERENCES

1. Borg M, Englund C, Wnuk K, et al. Safely entering the deep: a review of verification and validation for machine learning and a challenge elicitation in the automotive industry. J Autom Softw Eng. 2019;1(1):1-19.
2. Moghadam MH, Borg M, Mousavirad SJ. Deeper at the SBST 2021 tool competition: ADAS testing using multi-objective search. In: 2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST). IEEE; 2021:40-41.
3. Abdessalem RB, Nejati S, Briand LC, Stifter T. Testing vision-based control systems using learnable evolutionary algorithms. In: 2018 IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE; 2018:143-154.
4. Flaxman A, Deogun J. Anomaly detection and isolation: a comparison of techniques. Int J Softw Eng Knowl Eng. 2010;20(7):1015-1033.
5. Hawkins R, Paterson C, Picardi C, Jia Y, Calinescu R. A technique for automated testing of software with a large number of test cases. In: Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE; 2019:318-328.
19. Back T. Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming. Genetic Algorithms. Oxford University Press; 1996.
20. Kennedy J, Eberhart R. Particle swarm optimization. In: Proceedings of ICNN’95-international conference on neural networks. IEEE; 1995:1942-1948.
21. Deb K, Pratap A, Agarwal S, Meyarivan TAMT. A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Trans Evol Comput. 2002;6(2): 182-197.
22. Riccio V, Tonella P. Model-based exploration of the frontier of behaviours for deep learning system testing. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering ACM; 2020:876-888.
23. Catmull E, Rom R. A class of local interpolating splines. Computer Aided Geometric Design. Elsevier; 1974:317-326.
24. Castellano E, Cetinkaya A, Thanh CH, Klikovits S, Zhang X, Arcaini P. Frenetic at the SBST 2021 tool competition. In: 2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST). IEEE; 2021:36-37.
25. Mavrovouniotis M, Li C, Yang S. A survey of swarm intelligence for dynamic optimization: algorithms and applications. Swarm Evol Comput. 2017;33: 1-17.
26. Slowik A, Kwasnicka H. Nature inspired methods and their application to engineering problems. Neural Comput Appl. 2020;2020:1-17.
27. Beyer H-G, Schwefel H-P. Evolution strategies—a comprehensive introduction. Natural Comput. 2002;1(1):3-52.
28. Mavrovouniotis M, Li C, Yang S. A survey of swarm intelligence for dynamic optimization: algorithms and applications. Swarm Evol Comput. 2017;33: 1-17.
29. Slowik A, Kwasnicka H. Nature inspired methods and their industry applications swarm intelligence algorithms. IEEE Trans Ind Inf. 2017;14(3): 1004-1015.
30. Clerc M, Kennedy J. The particle swarm-explosion, stability, and convergence in a multidimensional complex space. IEEE Trans Evol Comput. 2002;6(1): 58-73.
31. Borg M, Henriksson J, Socha K, Lennartsson O, Lönegren ES, Bui T, Moghadam MH. Ergo, smirk is safe: a safety case for a machine learning component in a pedestrian automatic emergency brake system. arXiv preprint arXiv:220407874; 2022.
32. Riccio V, Jahangirova G, Stocco A, Humbatova N, Weiss M, Tonella P. Testing machine learning based systems: a systematic mapping. Empir Softw Eng. 2020;25(6):5193-5254.
33. Scharke H, Schneider H, Kinalzyk D. Virtual validation of ADAS/AD functions. ATZelectronics Worldwide. 2022;17(7):8-13.
34. Rosique F, Navarro PJ, Fernández C, Padilla A. A systematic review of perception system and simulators for autonomous vehicles research. Sensors. 2019;19(3):648.
35. LG Electronics. SVL Simulator. https://www.svlsimulator.com/; Retrieved, July 2022.
36. Belbichir A, Smajl J-C, Blosseville J-M, Gruyer D. Simulation-driven validation of advanced driving-assistance systems. Procedia-Social Behav Sci. 2012; 48:1205-1214.
37. TASS International. PreScan Simulator. https://tass.plm.automation.siemens.com/prescan-overview; Retrieved July, 2022.
38. Panichella A, Gambi A, Zampetti F, Riccio V. SBST tool competition 2021. In: 2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST). IEEE; 2021:20-27.
39. Levenshtein VI, et al. Binary codes capable of correcting deletions, insertions, and reversals. In: Soviet Physics Doklady Soviet Union; 1966:707-710.
40. Gambi A, Müller M, Fraser G. Asfault: Testing self-driving car software using search-based procedural content generation. In: 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion). IEEE; 2019:27-30.
41. Bojarski M, Del Testa D, Dworakowski D, et al. End to end learning for self-driving cars. arXiv preprint arXiv:160407316; 2016.
42. Panichella S, Gambi A, Zampetti F, Riccio V. SBST tool competition 2021. In: 2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST). IEEE; 2021:20-27.
43. Levenshtein VI, et al. Binary codes capable of correcting deletions, insertions, and reversals. In: Soviet Physics Doklady Soviet Union; 1966:707-710.
44. Kaur P, Taghavi S, Shi W. A survey on simulators for testing self-driving cars. In: 2021 Fourth International Conference on Connected and Autonomous Driving (MetroCAD). IEEE; 2021:62-70.
45. Wilcoxon F. Individual Comparisons by Ranking Methods. J Am Stat Assoc. 1952;47(260):583-621.
46. Arcuri A, Briand L. A hitchhiker’s guide to statistical tests for assessing randomized algorithms in software engineering. Empir Softw Eng. 2014;19(4):219-250.
47. Nguyen V, Gambi A, Ahmed J, Fraser G, Crispe: towards generating test cases from accident sketches. In: 2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion). IEEE; 2022:339-340.
48. Börsch C, Czank N, Khatiri S, Gambi A, Panichella S. Cost-effective simulation-based test selection in self-driving cars software with SDC-Scissor. In: 29th IEEE International Conference on Software Analysis, Evolution, and Reengineering, Honolulu, USA (online), 15-18 March 2022 ZHAW Zürcher Hochschule für Angewandte Wissenschaften; 2022.
49. Börsch C, Khatiri S, Derakhshanfar P, Panichella S. Panichella A. Single and multi-objective test cases prioritization for self-driving cars in virtual environments. Proceedings of the ACM on Measurement and Analysis of Computing Systems; 2022.
50. Tian Y, Pei K, Jana S, Ray B. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In: Proceedings of the 40th international conference on software engineering. IEEE/ACM; 2018:303-314.
51. Kao C, Yang J. Jana S. Deepexplore: Automated whitebox testing of deep learning systems. In: Proceedings of the 26th Symposium on Operating Systems Principles. ACM; 2017:1-18.
52. Zhang M, Zhang Y, Zhang L, Liu C, Khurshid S. Deeproad: Gan-based metamorphic testing and input validation framework for autonomous driving systems. In: 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE; 2018:132-142.
53. Abdesselem R, Nejati S, Briand LC, Stifter T. Testing advanced driver assistance systems using multi-objective search and neural networks. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. IEEE/ACM; 2016:63-74.
54. Panichella A, Kiftelew FM, Tonella P. Reformulating branch coverage as a many-objective optimization problem. In: 2015 IEEE 8th international conference on software testing, verification and validation (ICST). IEEE; 2015:1-10.
55. Abdesselem RB, Nejati S, Briand LC, Stifter T. Testing vision-based control systems using learnable evolutionary algorithms. In: 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE). IEEE; 2018:1016-1026.
56. Haq FU, Shin D, Briand LC, Stifter T. Wang J. Automatic test suite generation for key-points detection DNNS using many-objective search (experience paper). In: Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis. ACM; 2021:91-102.
57. Haq FU, Shin D, Nejati S, Briand L. Can offline testing of deep neural networks replace their online testing? Empir Softw Eng. 2021;26(5):1-30.
58. Borg M, Abdessalem RB, Nejati S, Jegeden F-X, Shin D. Digital twins are not monozygotic–cross-replicating ADAS testing in two industry-grade automotive simulators. In: 2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST). IEEE; 2021:383-393.

How to cite this article: Moghadam MH, Borg M, Saadatmand M, Mousavirad SJ, Bohlin M, Lisper B. Machine learning testing in an ADAS case study using simulation-integrated bio-inspired search-based testing. J Softw Evol Proc. 2024;36(5):e2591. doi:10.1002/smr.2591