Automatically Testing Self-Driving Cars with Search-Based Procedural Content Generation

Alessio Gambi
alessio.gambi@uni-passau.de
University of Passau
Passau, Germany

Marc Mueller
mmueller@beamng.gmbh
BeamNG GmbH
Bremen, Germany

Gordon Fraser
gordon.fraser@uni-passau.de
University of Passau
Passau, Germany

ABSTRACT

Self-driving cars rely on software which needs to be thoroughly tested. Testing self-driving car software in real traffic is not only expensive but also dangerous, and has already caused fatalities. Virtual tests, in which self-driving car software is tested in computer simulations, offer a more efficient and safer alternative compared to naturalistic field operational tests. However, creating suitable test scenarios is laborious and difficult. In this paper we combine procedural content generation, a technique commonly employed in modern video games, and search-based testing, a testing technique proven to be effective in many domains, in order to automatically create challenging virtual scenarios for testing self-driving car software. Our AsFault prototype implements this approach to generate virtual roads for testing *lane keeping*, one of the defining features of autonomous driving. Evaluation on two different self-driving car software systems demonstrates that AsFault can generate effective virtual road networks that succeed in revealing software failures, which manifest as cars departing their lane. Compared to random testing AsFault was not only more efficient, but also caused up to twice as many lane departures.

CCS CONCEPTS

• **Theory of computation → Random search heuristics**; • **Software and its engineering → Empirical software validation**; **Search-based software engineering**; **Virtual worlds training simulations**; *Software safety*.

KEYWORDS

automatic test generation, search-based testing, procedural content generation, self-driving cars

ACM Reference Format:
Alessio Gambi, Marc Mueller, and Gordon Fraser. 2019. Automatically Testing Self-Driving Cars with Search-Based Procedural Content Generation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19), July 15–19, 2019, Beijing, China.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3293882.3330566

ISSTA '19, July 15–19, 2019, Beijing, China

ACM ISBN 978-1-4503-6224-5/19/07…$15.00
https://doi.org/10.1145/3293882.3330566

1 INTRODUCTION

Recently, autonomous driving has become an important part of the industry, and experts forecast this will profoundly impact society [6], in particular by increasing safety [8]. However, this is not yet the case: Even though the prototypes of famous companies like Waymo-Google [46], Tesla [39], or Uber [45] have reached high level of autonomy and can drive in everyday urban traffic, they have caused crashes and even fatal accidents (e.g.,[12, 37, 47]). Self-driving car software clearly needs to be better tested, but there is no standardized procedure to test automated vehicles yet [20], and neither methods for testing traditional cars, nor traditional software testing approaches translate well into the space of self-driving cars [21, 24, 36]. A common approach lies in naturalistic field operational tests (N-FOT), which put self-driving cars on trial in the real world, but these are not only expensive and dangerous, but also have limited effectiveness [18].

A more efficient and less risky alternative for testing self-driving car software lies in *virtual tests* [23]. Virtual tests follow the X-in-the-loop paradigm [41] and test self-driving car software by feeding them simulated sensory data [26], synthesized images [40, 50], or abstract test scenarios [1, 2]. Typically, these approaches require models of the hardware implementing the self-driving car and its sensors (e.g., [1, 2]). A more general approach consists in generating entire digital realities, i.e., virtual worlds, in which the self-driving car software is deployed [5, 33]. Generating such virtual environments for testing the self-driving car software comes with two main technical challenges: (i) generating environmental elements, e.g., terrain, weather, roads; and, (ii) assembling them into simulations which implement relevant test cases, i.e., test cases which challenge the self-driving car software, hence have the potential to expose problems in its implementation.

To address these challenges, we propose an approach for systematically testing self-driving car software which combines (1) procedural content generation (PCG) and (2) search-based testing (SBST). PCG is a core element of modern video games, and enables the automatic creation of photorealistic virtual environments [43]. SBST [25] describes the use of efficient meta-heuristic search algorithms to generate program inputs that achieve testing objectives such as achieving coverage or finding faults. PCG and SBST have previously been combined, for example, to generate new game elements driven by the feedback collected from the users playing the game [42]. In the context of racing games, Loiacono et al. [22] used generic algorithms to create racing tracks with various turns and speed profiles to increase players' enjoyment, while Georgiou et al. [17] automatically generated challenging tracks based on players bio-metric feedback, such as eye gaze and movement.

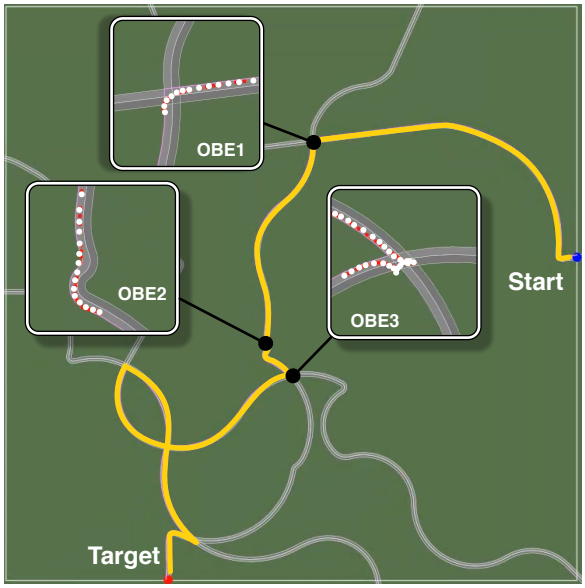

Figure 1: Example of a test case generated by AsFault

The figure reports one test case generated by AsFault during our experimental evaluation and the three safety-critical problems, called OBE (Out of Bound Episodes), it exposed. A test case is defined as a navigation path on the map (highlighted in yellow) that the ego-car must follow without causing any OBE.

In this paper, instead, we combine them for systematically creating virtual tests which effectively expose problems in self-driving car software. In particular, our AsFault prototype targets the *lane keeping* functionality of self-driving cars.

Lane keeping is the fundamental feature of autonomous driving according to SAE international [32], and self-driving cars should be able to always drive inside their lane no matter the weather conditions or the shape of the road. In order to test lane keeping, AsFault uses a genetic algorithm to iteratively refine virtual road networks towards those which cause the self-driving car software under test, i.e., the *ego-car*, to move away from the center of the lane. Eventually, AsFault generates virtual roads which cause the ego-car to drive off the lane. Figure 1 shows an example of a road network generated by AsFault which caused the ego-car to drive off the lane three times.

In detail, the contributions of this paper are as follows:

- We introduce an approach based on procedural content generation to generate virtual road networks (Section 3).
- We introduce a search-based approach to evolve virtual road networks towards safety-critical scenarios that cause ego-car to depart the road (Section 4).
- We evaluate our AsFault prototype implementation on two self-driving car software systems (Section 5).

Our extensive evaluation, which consists of executing more than 160,000 virtual tests against two test subjects shows that AsFault efficiently generates virtual roads which always cause the ego-car to drive off the lane in various occasions. Compared to random testing, AsFault is not only faster, but also generates more effective test suites which caused up to two times more out of bound episodes (OBE) than random testing for one of our test subjects.

2 BACKGROUND

In this paper, we describe a test generation technique targeting the lane keeping functionality of self-driving cars. Keeping the lane is one of the fundamental tasks in driving as well as autonomous driving. If self-driving cars cannot drive inside their lane, they might easily become a major safety-critical hazard. For instance, if a self-driving car runs over a sidewalk, it might hurt passers-by, and if it invades the opposite traffic lane, it might crash into oncoming traffic. This section describes lane keeping and introduces procedural content generation to provide context for this work.

Lane keeping systems work by continuously tracking the striped and solid lane markings of the road ahead using advanced image processing, deep learning, or machine learning techniques [49]. By elaborating images captured by forward facing cameras attached to the front of the vehicles the lane keeping systems derive lane data, such as the width of the lane, the current position of the car with respect to it ("in the middle", "between lanes", etc.), and the heading angle of the car, which are necessary to compute the driving actions to follow the lane or to regain its center.

Since lane keeping systems rely on visible lane markings, when these are faded or missing, then lane keeping systems might not operate correctly [15]. Therefore, state of the art approaches to test lane keeping systems feed them images of real roads suitably altered to simulate variable weather conditions [50] or to mimic distortions introduced by faulty cameras [40]. Those approaches aim to check if and how lane keeping becomes inconsistent; however, because they test lane keeping systems using only single frames, they cannot be used to evaluate *closed-loop properties* of lane keeping systems [44], including the ability to drive within a lane.

Instead of altering images of real roads, in this paper we generate virtual roads inside a driving simulator, which can generate photo-realistic, but synthetic, images of roads. By altering the properties of the virtual roads, for example by altering their geometry, we indirectly change how the road markings are rendered in the images fed to the lane keeping systems. This, in turn, lets us test both their ability to correctly elaborate the images to derive lane data in different conditions and the effects which different road configurations have on the systems acting on this input in order to keep the vehicle in the lane. In principle, we can include additional elements such as weather, obstacles, and traffic, in the simulation; however, we choose to focus only on generating virtual road networks with the reasoning that exposing problems in the lane keeping systems under "perfect" conditions likely identifies the presence of severe bugs. Simulating additional elements is part of our ongoing work.

Accurately simulating the physics of driving while rendering photorealistic images is currently achieved by game engines, and gaming technologies are already successfully employed for training and evaluating computer vision and other machine learning algorithms [5, 11, 28, 31], despite several theoretical concerns about their representativeness [29]. Game engines serve as a framework for the creation of video games and require accurate descriptions of the physical (e.g., mass, material) and visual (e.g., geometry, texture) properties of the various entities they simulate as well as the definition of the game rules and levels to correctly operate. Generating these models and configuring games is cumbersome and time consuming; therefore, the gaming industry uses procedural

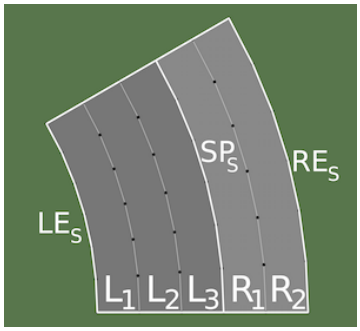

Figure 2: An example of multi-lane road segment represented as set of polylines.

The figure visualizes an example of a road segment which contains multiple lanes and identifies the points which define their polylines as black dots.

content generation (PCG) [42], i.e., the algorithmic creation of game content with limited or indirect user input.

Using PCG for generating roads automatically to test lane keeping is challenging. On the one hand, there are many different aspects of real roads that affect the ability of lane keeping systems to behave correctly, resulting in huge numbers of possible road configurations. Examples of such aspects are the length and steepness of the road segments, the shape of turns and the presence of intersections as well as random artifacts like potholes. On the other side, roads must be geometrically valid; for example, the various road segments must line up, and roads should not self-intersect or partially overlap.

3 PROCEDURAL GENERATION OF ROAD NETWORKS

The aim of the procedural generation of road networks is a precise description of the structure and the geometry of roads and their lanes in terms of *polylines* (Section 3.1). We generate these using an incremental approach: First, we build roads in isolation by procedurally generating their road segments (Section 3.2) and sealing road segments together to ensure gapless roads (Section 3.3). Road segments are modeled in term of predefined parametrized types such as "straight segment 30*m* long", "20° left turn", and so on, while roads are sequences of consecutive road segments which develop from a starting point on the map. Then, we combine roads together on the same maps to form road networks (Section 3.4).

3.1 Representing Roads as Polylines

Polylines are discrete sequences of points which define the boundaries of bi-dimensional shapes, i.e., lanes in our work, and are the basic geometrical representation of roads used by many game engines, driving simulators, and geographical information systems [9]. Figure 2 shows an example of a road segment represented as a set of polylines which define each of its five lanes (three on the left of the road spine SP_s, and two on its right), and the external edges of the road segment (i.e., LE_s and RE_s). The polyline-based representation of roads simplifies the implementation of common operations for rendering and manipulating roads as well as automatic checks on their geometrical properties (e.g., self-intersection, partial overlapping). Given their fundamentally discrete nature, polylines are not as precise as other geometrical constructs, like *clothoids*; despite this, as Althoff et al. [3] illustrate, in the context of self-driving car software validation using a polylines-based representation of roads has a comparable expressive power to other formats, such as OpenDrive [14] which is based on clothoids.

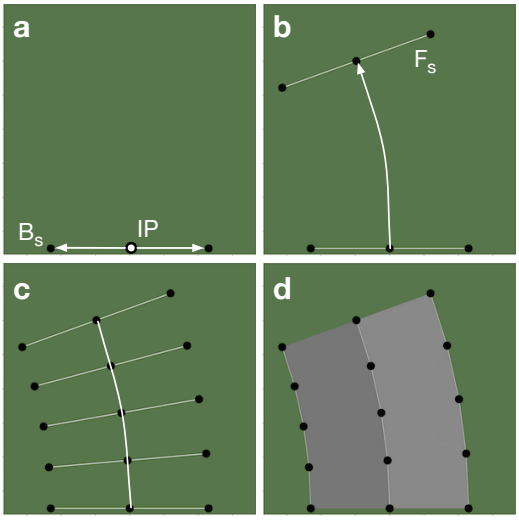

Figure 3: Road segment generation

The figure illustrates the procedural generation of a left turn with two lanes.

3.2 Generating Road Segments

The elementary activity when procedurally generating roads is the creation of road segments, and we define it as follows (Figure 3): For a road segment s, (a) we construct its *back line* (B_s) from the starting point IP of the road spine SP_s; B_s is the edge which defines where the road segment starts and contains the initial points of the polylines which define the lanes. (b) We construct the road segment *front line* (F_s) by applying affine transformations to all B_s points; we generate straight road segments by translation and turns by additional rotation. (c) We compute the position of all the internal points of the lane polylines by interpolation; and, finally (d) we obtain the geometrical representation of the road segment lanes.

3.3 Generating Roads

Roads are generated procedurally by stitching one road segment to the next one. To construct gapless roads, we constrain the road segment generation to use the front line of a road segment as the back line of the following one.

In the context of lane keeping testing, in order to avoid testing self-driving car software using impossible road configurations, procedural road generation must ensure that only *valid roads* are generated. In this work, we define valid roads as those which are not only gapless, but also do not self-intersect. Additionally, to fit with the capabilities of current driving simulators which cannot accurately simulate physics over extremely long distances and usually reason in terms of maps, we constrain the procedural road generation to generate roads within the boundaries of a fixed-size map by checking that roads start and end on the map boundary.

Given a sequence of road segments and the initial position of the road on the map boundary, we procedurally generate the road

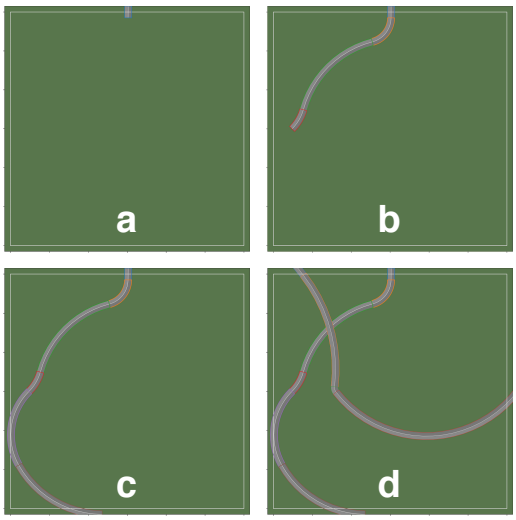

Figure 4: Roads and road networks generation

The figure illustrates the process of generating roads (a) to (c) by automatically appending road segments, and generating road networks (d) by stacking roads on the same map.

as follows (Figure 4): (a) we generate a small *straight* road segment from the given initial position on the map boundary towards the inside of the map to ensure by construction that the direction of the first back line is defined. (b) Next, we generate succeeding road segments as described above, and check that the road generated so far is valid. If the road becomes invalid, for example because it self-intersects, the road generation fails; otherwise, we continue with the generation of the next road segment. (c) Eventually, after generating all the road segments, we check if the road is valid, i.e., whether it crosses the map boundary; otherwise, the road generation fails.

3.4 Generating Road Networks

To procedurally generate road networks, we generate roads and place them on the same map (Figure 4 — d). As before, we need to ensure that either the resulting configuration is valid, or we fail the generation of the road network.

We define valid road networks as those networks which contain only valid *intersecting* roads, that is, they contain valid roads whose central polylines (SP_s) intersect at least once. Figure 5 illustrates this concept with an example of a valid configuration in which the roads correctly intersect (a), and an example of an invalid configuration in which the roads partially overlap (b). Additionally, in order to ensure that, given a road network, lane keeping can be tested in all the possible ways, we require that valid road networks allow a car to drive from any two road segments.

4 SEARCH-BASED TESTING FOR LANE KEEPING

4.1 Overview

AsFault uses a genetic algorithm to evolve road networks with the aim of finding errors in the lane keeping functionality of self-driving cars. The genetic algorithm is initialized with an initial population of random road networks. To evaluate the fitness of test

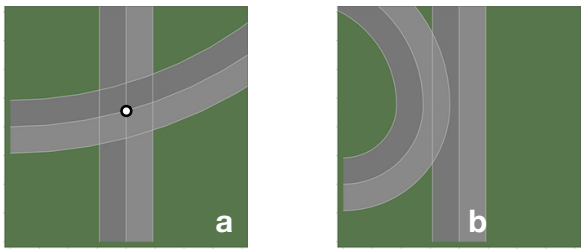

Figure 5: Valid and invalid road overlaps

The figure exemplifies a valid intersection (a) and an invalid partial overlap (b).

cases, the road networks are instantiated to driving simulations, in which the ego-car is instructed to reach a target location following a navigation path selected by AsFault. During the simulation, AsFault traces the position of the ego-car at regular intervals such that it can identify out of bound episodes (OBEs), i.e., lane departures. The distance between the center of the lane and the position of the ego-car is used to compute the fitness of individuals. This guides the genetic algorithm in evolving the test cases by recombining and mutating their road networks. The algorithm continues to execute and evolve test cases until a user defined ending condition is met, at which point AsFault returns the final test suite and stops.

4.2 Representation

Road networks are represented using an *operational* encoding, which is more amenable for automatically evolving the road networks than polylines. The encoding consists of a hierarchical data structure which associates road networks to roads, and roads to road segments. This enables AsFault to automatically generate the polylines of the roads when needed. In addition to the road network, a test case also requires the definition of a *driving task* for the ego-car. In the context of lane keeping, this essentially corresponds to the task of driving through a road network following a given navigation path. A valid test requires that navigation is possible along roads, a property that our procedural road network generation guarantees for all road networks it generates. AsFault identifies such navigation paths by (i) building a graph representation of the road network, in which edges model road segments and nodes model either road intersections (internal nodes) or intersections between roads and map boundaries (source and destination nodes), and (ii) selecting a navigation path between randomly chosen source and destination nodes. In doing so, AsFault relies on the fact that the road networks enable to drive between any two road segments, hence result in fully connected graphs.

Despite this simplification, selecting a navigation path is not trivial. On the one hand, there might be a large number of possible paths between source and destination nodes (possibly infinite if the graphs derived from the road networks contain cycles); on the other hand, different paths in the same network might correspond to roads which have very different geometrical properties and are likely to stress the ego-car in different ways. For example, to increase the chances of observing several OBEs, the roads on which the ego-car is tested should not be too short, and different paths should result in geometrically dissimilar roads. Additionally, to avoid subjecting the ego-car to the same set of stimuli, a path should not traverse the same kinds of road segments more often than necessary.

Since an analytic solution to the path selection problem cannot be efficiently computed in general [34], AsFault adopts a heuristic which aims to maximize the amount of road network traversed during test execution, while keeping the path selection fast despite its complexity. In particular, given a road network AsFault, first, randomly samples the set of *simple paths*[1] between the given source and destination nodes; then, it selects the longest navigation path among those. In the context of AsFault, the longest navigation path corresponds to the navigation path which contains the largest amount of road segments among the sampled ones.

4.3 Implementation and Execution of Tests

Converting the genotypic representation of tests to driving simulations is where procedural content generation takes place. AsFault uses code templates to generate the necessary simulation code, which instantiates the virtual roads in the driving simulation from the polylines, places the ego-car in the expected starting position, and instructs the ego-car about how to reach the target location. The ego-car then drives along the virtual roads following the chosen navigation paths.

The simulation code also contains timeout logic which fails the test execution and suspends the simulation if the ego-car cannot reach the target location fast enough. AsFault computes the timeout value based on the traveling distance to reach the end of the navigation path and a minimum constant cruise speed (i.e., 1 m/s).

Once the simulation code is ready, AsFault spawns the driving simulator and observes the driving behavior of the ego-car by sampling the car position at a constant rate (i.e., every 250 ms). Samples are stored into a trace which enables AsFault to identify the occurrence of OBE by finding sequences of observations for which the distance between the recorded position of the ego-car and the center of the lane was bigger than the half of the lane width. Figure 1 illustrates these concepts graphically over a few occurrences of OBE that AsFault identified during the experimental evaluation.

AsFault represents simulated cars by their center of mass, hence OBEs are identified only if a big enough part of the ego-car goes outside the lane; consequently, AsFault cannot identify very small infractions. Extending AsFault to consider the bounding box of the ego-car while checking for OBE is an engineering effort which does not affect the generality of the approach but only its precision.

AsFault checks the traces offline; hence, it does not halt the test execution after observing the first OBE. Instead, tests continue until either the ego-car reaches the target position within the given timeout or the timeout triggers. AsFault adopts this strategy to balance the cost of running expensive computer simulations with the benefit of collecting as many OBE occurrences as possible within the context of the same test execution. AsFault uses the execution traces not only for counting how many times the ego-car drove out of the lane, but also for evaluating the fitness of tests which guides the evolution process as we describe in the next section.

4.4 Fitness Function

In the context of lane keeping, effective tests are those which cause the self-driving car software to break out of the lane bounds, that is, cause OBEs. Therefore, AsFault uses as fitness function D_{LANE} (Eq. 1), which rewards those tests which cause the ego-car to move the furthest away from the lane centre. Given a test (T), the navigation path defining it (P_T), and the execution trace collected during its execution ($v = (v_1, ..., v_n)$), we define D_{LANE} as:

$$D_{LANE}(T, v) = \begin{cases} \max_{v_i \in v} D(v_i, P_T) & \text{if } \max_{v_i \in v} D(v_i, P_T) \leq W_{LANE}/2 \\ W_{LANE}/2 & \text{otherwise} \end{cases} \quad (1)$$

where $D(\bullet, P_T)$ is the shortest distance of a point ($\bullet$) to the centre of the lane in path P_T and W_{LANE} is the width of such lane.

D_{LANE} captures the intuition that tests which cause the ego-car to drive away from the lane center might contain road segments which stress the self-driving car software more, and this eventually leads the ego-car to drive out of the road. AsFault works under the assumption that OBEs are the exception, not the rule. Under this assumption, using the average distance to the lane center would smooth out, and effectively filter, short out of bound episodes; hence, we opted for the maximum value of the distance to the lane center to define D_{LANE}. Additionally, in order to promote the generation of more tests in a test suite which cause OBEs, instead of generating tests which try to expose more extreme problems, we cap the value of D_{LANE} to $W_{LANE}/2$ for tests which achieve their purpose.

4.5 Search Operators

AsFault evolves road networks by applying search operators which mutate and recombine roads and road networks according to configurable probabilities. Figures 6, 7, and 8 illustrate the application of those genetic operators on sample road networks.

AsFault mutates roads by randomly replacing their road segments with new ones (Figure 6), while it recombines roads using the *join* crossover operator (Figure 7). The join crossover operator splits roads from parent road networks at random points and re-joins their road segments shuffled such that no two road segments in the offsprings road networks come from the same parent road network; notably, this search operator does not change the number of roads in the road networks. To generate road networks with different roads in them, AsFault uses the *merge* crossover operator, which selects random, possibly empty, subsets of roads from parent road networks and re-distributes them into their offspring (Figure 8).

These search operators may produce invalid road configurations. For example, after a road mutation or the application of the join crossover roads might self-intersect or might not intersect the map boundary anymore; similarly, after the application of the merge crossover roads might partially overlap. When such a case is detected, AsFault retries the application of the same search operator with a probability of giving up which increases per failed attempt. This way, AsFault ensures that either a valid configuration is found quickly or the entire search operation is aborted, such that the test generation process can continue. The application of search operators is rather efficient, and many different configurations can be generated in a very short time; hence, we prefer to "generate and validate" configurations rather than using constraint solvers in order to generate valid configurations. We will investigate the use of constraints for generating valid roads in future work.

[1] A simple path between two nodes is the path which connects them by traversing each node at most once

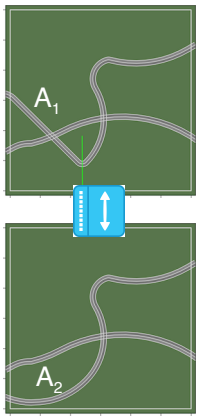

Figure 6: Road mutation

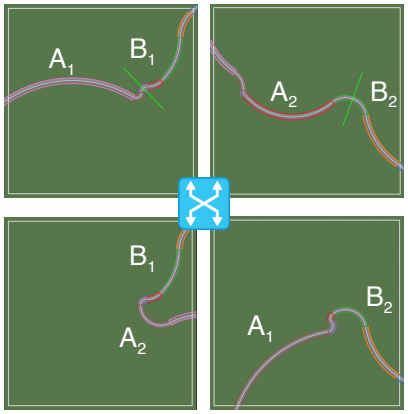

Figure 7: Join crossover

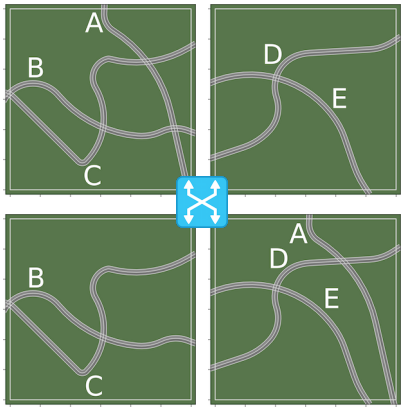

Figure 8: Merge crossover

The figures show the application of the genetic operators defined by AsFault: road network mutation (Figure 6), which randomly replaces road segments in a road; join (Figure 7), which recombines road segments from parent roads to form new roads; and, merge (Figure 8), which recombines roads in parent road networks to form new road networks.

Mutating and recombining road networks might also generate *similar* tests, that is, tests defined over navigation paths which have similar geometrical properties. We expect that the similar tests stress lane keeping in similar ways; hence, executing those tests would not likely provide any additional insight on the lane keeping behavior, while it would considerably slow down the generation process. So, to improve the efficiency of test generation, AsFault identifies and filters out similar tests before executing them. Our expectation is grounded in the fact that AsFault is black-box and randomized, hence, it cannot rely on information about the internals of the system under test to systematically guide the search towards identifying those very few *adversarial* tests among the multitude of similar tests that can be generated via mutation and crossover.

AsFault computes the similarity between tests by means of the Jaccard Index of their road segments. Formally, given tests T_1 and T_2, we define their similarity by means of Eq. 2:

$$\text{similarity}(T1, T2) = \frac{|C_{T_1} \cap C_{T_2}|}{|C_{T_1} \cup C_{T_2}|} \quad (2)$$

where C_{T_i} refers to sequences of consecutive road segments of a given size. Values of similarity close to 1 indicates that the tests contains many common sub-sequences of road segments, while values close to 0 suggest that tests differ in many regards.

As consequence of filtering out invalid and similar test cases, the number of offspring in newer generations might be smaller than the configured population size. In this case, AsFault pads the new generations with the fittest individuals from the previous one, and the process continues by executing *only* the newly generated tests.

5 EVALUATION

To assess the benefits of using procedural content generation for testing lane keeping systems and to understand how AsFault's main configuration parameters affect the quality of both the test generation process and the tests it generates, we investigate the following main research questions:

RQ1 **Can we expose safety-critical problems in lane keeping systems by procedurally generating roads?** To the best of our knowledge, we are the first using procedural road generation for testing lane keeping systems. Hence, we are interested to understand if this leads to effective tests.

RQ2 **Does search-based testing improve the effectiveness of testing with procedural content generation?** Generating road networks using search is more complex than generating them randomly. This raises the question whether using a genetic algorithm leads to more OBEs than generating tests randomly.

RQ3 **Does generating tests which feature intersecting roads improve testing effectiveness?** Compared to single-road networks, multi-road networks offer the possibility to create more scenarios, but might be harder to generate. This raises the question whether generating tests from multi-road networks actually pays off.

RQ4 **How does the size of the map impact testing?** Using large maps allows us to generate long roads which contain many road segments. This increases the chances of observing OBEs, but also slows down both the test generation and the test execution, which raises the question whether and how the map size affects testing.

5.1 Experimental Settings

To address the research questions, we implemented AsFault in a prototype [16], and conducted a large number of experiments by executing AsFault and the driving simulator on a commodity "gaming" PC running Windows 10 and equipped with an AMD Ryzen 7 1700X 8-Core CPU (3.4 GHz), 64 GB of memory, and an NVIDIA Geforce GTX 1080 GPU.

Driving simulator. In the current implementation, AsFault relies on the state of art driving simulator BeamNG.research [7], a freely available research-oriented version of the commercial game BeamNG.drive. We opted for BeamNG.research instead of other available simulators used for testing autonomous vehicles (e.g., [13], [35]) for two main reasons: first, BeamNG.research exposes an intuitive API for programmatically configuring virtual roads and controlling the simulations, which the other simulators currently do not provide; and, second, it features a very accurate driving physics engine.

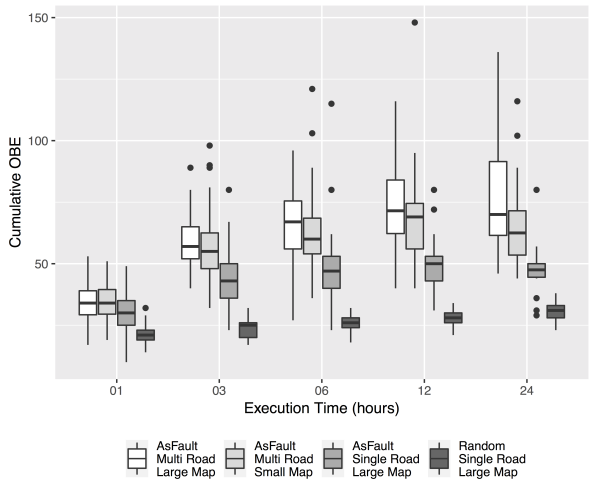

Figure 9: Test suite effectiveness — BeamNG.AI

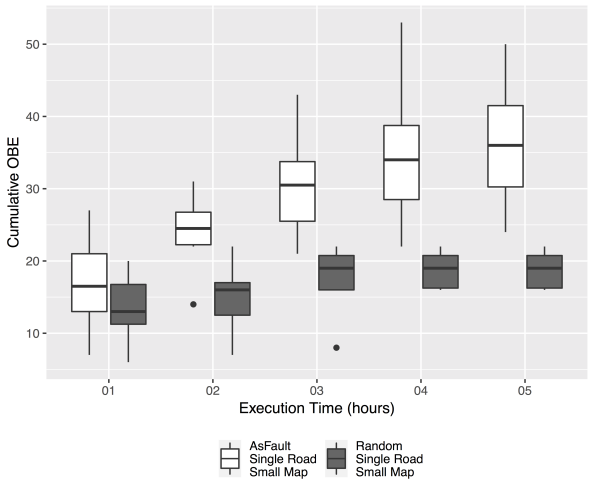

Figure 10: Test suite effectiveness — DeepDriving

Design of Experiments. Although we can generate roads with multiple lanes per traffic direction, we configured the procedural road generation to generate only roads with one lane per traffic direction. This choice is motivated by two reasons: First, roads with only one lane per direction are the simplest to generate which enable us to test lane keeping with respect to different unsafe behaviors such as driving off the road and invading the opposite traffic lane. Second, single lane roads simplify the definition of a test oracle for lane keeping; in case of roads with multiple lanes for each direction, if the ego-car switches lane, it might be hard to tell if that is the expected behavior or the manifestation of a bug. We configured AsFault to generate *single-* and *multi-* road networks, over *small* (1 Km^2) and *large* maps (4 Km^2). Additionally, we configured AsFault to filter similar test cases when the similarity value computed over pairs of road segments scores a value of 0.9 or above. For generating single-road networks, AsFault always used the join crossover operator, while multi-road networks selected join or crossover with equal probability (50%). In both settings, AsFault mutates offspring with a chance of 5%.

5.2 Test Subjects

We consider two test subjects, both implementing all the required functionality to drive the ego-car autonomously and in the lane.

BeamNG.AI. BeamNG.research ships with a driving AI that we refer to as BeamNG.AI. It has perfect knowledge of the virtual roads and drives the ego-car by computing an ideal driving trajectory to stay in the center of the lane. Then, it derives the actual driving actions (i.e., acceleration, braking, and steering) to follow this trajectory. The behavior of BeamNG.AI can be parameterized with a so-called "aggression" factor which controls the risk the driver is willing to take in order to decrease the time to reach the designated destination. As explained by BeamNG.research developers, low aggression values (e.g., 0.7) result in a smooth driving, while high aggression values (e.g., 1.2 and above) result in an edgy driving, which might even "cut corners." For this evaluation, we set the aggression value to 0.75, which corresponds to a rather *conservative* driver that always favors safe travels over quick ones.

DeepDriving. DeepDriving is a vision-based self-driving car software developed by Chen et al. [11]. It implements the *direct perception* paradigm: first, it uses a Convolutional Neural Network (CNN) to predict a number of driving affordance indicators, such as the current position of the car on the road, from the images captured by a forward facing camera attached to the front of the ego-car; then, it uses a standard rule-based driving controller to compute the driving actions from those indicators. For this evaluation, we used the open-source implementation of DeepDriving based on TensorFlow[2] provided by A. Netzeband [30], and we extended it by including a safety logic which mimics the setup of current autonomous vehicles. Our extension monitors the quality of DeepDriving predictions and disengages DeepDriving when the quality of its predictions is not satisfactory. At this point, the control of the ego-car passes to BeamNG.AI until the quality of DeepDriving predictions raises back to acceptable values, and the ego-car regains the center of the lane. During the test execution, the driving simulator renders the images required by DeepDriving, and pauses the simulation waiting for DeepDriving to compute the driving controls. As soon as the driving controls are ready, they are applied to the ego-car and the simulation resumes until the next control cycle. As described by Chen and co-authors, we configured DeepDriving to operate at a frequency of $10Hz$.

5.3 RQ1. Can We Expose Safety-Critical Problems in Lane Keeping by Procedurally Generating Roads?

To answer the first research question, we procedurally generated single-road road networks at random, and used those for testing BeamNG.AI and DeepDriving. In particular, for testing BeamNG.AI, we evolved populations of 25 individuals, used the large map, and stopped the generation after 24 hours; we repeated this experiment for $n = 40$ times. At the end of each experiment, AsFault generated test suites containing the best 25 tests.

[2] https://www.tensorflow.org/

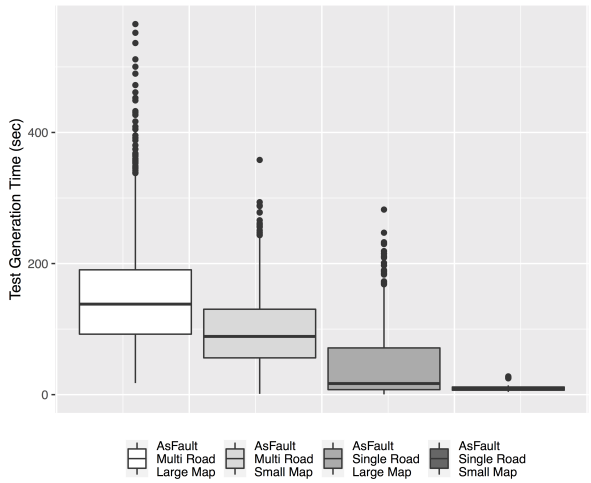

Figure 11: Test generation efficiency

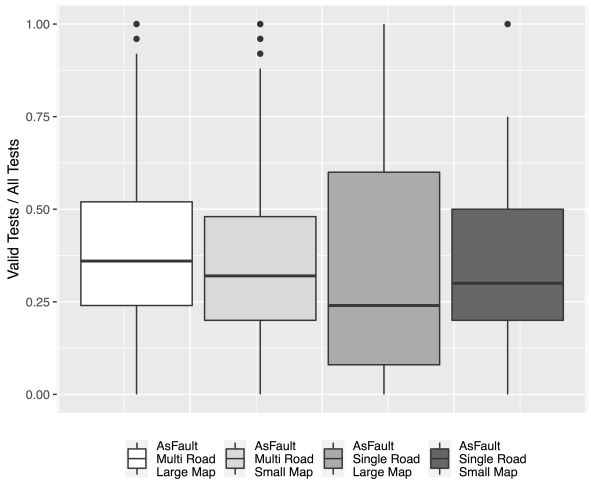

Figure 12: Test generation effectiveness

DeepDriving requires to run simulations in *synchronous* mode for suitably controlling the ego-car.[3] This is considerably slower than the setup we used to test BeamNG.AI, so, to reduce the execution time for running the experiments we reduced the number of tests executed at each iteration. For testing DeepDriving, we therefore generated test suites composed of 20 tests using the small map; we repeated this experiments for $n = 10$ times. Additionally, we limit the generation budget to 5 hours, because the empirical evidence collected while testing BeamNG.AI suggested that around that time AsFault converged and improved only marginally afterwards.

Figure 9 and Figure 10 plot the overall amount of OBEs (*Cumulative OBE*) identified by the best randomly generated test suites under the labels "Random Single Road Large Map" and "Random Single Road Small Map". From these results we observe that procedurally generating roads effectively was able to cause a multitude of OBEs in both test subjects already at the beginning of the execution; for example, after the first hour, Random caused on average 21.4 OBEs in BeamNG.AI and 13.6 OBEs in DeepDriving. Random kept causing more OBEs as the test generation progressed and, by the end of the execution, it caused on average 29.1 OBEs in BeamNG.AI, and 18.1 OBEs in DeepDriving. These results let us conclude that:

RQ1: In our experiments, procedurally generated road networks found 29 *and* 18 *OBE on average in the two test subjects.*

5.4 RQ2. Does Search-Based Testing Improve the Effectiveness of Testing with Procedural Content Generation?

To answer the second research question, we configured AsFault to generate single-road road networks over the large map. As before, for testing BeamNG.AI we evolved populations composed of 25 individuals using the large map and a generation budget of 24 hours, and repeated the experiment 40 times. For testing DeepDriving we evolved populations of 20 individuals using the small map and a generation budget of 5 hours, and repeated the experiment 10 times.

[3] In synchronous mode, the simulation is paused while the driving controller computes the driving commands and resumed afterwards for the duration of a control period.

Comparing the results under the labels "AsFault Single Road Large Map" and "Random Single Road Large Map" in Figure 9 we observe that AsFault produced more OBEs on BeamNG.AI than Random during the entire generation ($\hat{A}_{12}$ ranges from 0.85 after 1 hour, to 0.96 after 24 hours, and p is always less than 0.005).

We also observe that the number of OBEs identified by AsFault quickly increased from an average amount of 30.1 OBEs after 1 hour to an average number of 42.1 after 3 hours, and then stabilized to reach an average amount of 47.5 OBEs after 24 hours. Compared to this, the test effectiveness achieved by Random increased slower passing from an average of 21.4 OBEs after 1 hour to an average of 25.2 OBEs after 3 hours; moreover, Random did not catch up with AsFault over the observation period, and after 24 the average number of OBEs it discovered was 29.1.

The results reported in Figure 10 show a similar trend also for DeepDriving. In this case, AsFault always caused more OBEs than Random, and the difference between the amount of OBEs caused by the two techniques increased with execution time. Between hours 1 and 5, the average number of OBEs caused by AsFault almost doubled, passing from 16.9 to 36, the average number of OBEs caused by Random during the same period increased only marginally, passing from 13.6 to 18 ($\hat{A}_{12}$ increased from 0.68 to 1, while the p-value passed from 0.18 to less than 0.005). In light of these results we conclude that:

RQ2: In our experiments, the genetic algorithm found 47.5 *and* 36 *OBEs on average, twice as many OBE as random search.*

5.5 RQ3. Does Generating Tests which Feature Intersecting Roads Improve Testing Effectiveness?

To answer the third research question, we configured AsFault to generate multi-road road networks. As before, we tested BeamNG.AI by generating test suites composed of 25 tests using the large map and a generation budget of 24 hours, and repeated the experiment

for 40 times. However, since DeepDriving cannot handle intersecting roads, we were not able to test it in this configuration.

From Figure 9, comparing the results achieved by AsFault using single roads ("AsFault Single Road Large Map") and multi-roads ("AsFault Multi Road Large Map"), we observe that the number of OBEs exposed by AsFault follows the same trend as before. At the beginning of the generation, both configurations achieved comparable results, but the multi-road configuration was slightly better than single-road ($\hat{A}_{12} = 0.65$ and $p = 0.01$): AsFault in single-road configuration exposed an average of 30.1 OBEs, while AsFault in multi-road exposed an average of 34.6 OBEs. During the generation, the results of both configurations improved, but "AsFault Multi Road Large Map" improved at a faster pace and always achieved better results than "AsFault Single Road Large Map" ($\hat{A}_{12}$ passed from 0.65 to 0.88, while p value dropped from 0.01 to less than 0.005).

From Figure 11, we can observe that AsFault took less time to generate single-road tests compared to generate multi-road tests ($\hat{A}_{12} = 0.1$ and $p < 0.005$): On average, AsFault took 41.1 sec to generate a single-road tests, while it took 152.6 sec to generate a multi-road one. We expect this result as for generating and evolving multi-road road networks AsFault did process more data and did perform more checks to establish road networks validity.

From the results in Figure 12, instead, we can make an interesting observation: compared to the multi-road case, evolving single-road tests took less time but generated fewer valid configurations ($\hat{A}_{12} = 0.59$, $p < 0.005$). The average value of the ratio of valid configurations generated by AsFault decreased from 0.39% for the multi-road to 0.34% for single-road, which suggests that evolving single-road road networks might be more difficult than evolving multi-road road networks. To explain this, we consider that the merge crossover operation used in the multi-road setting works with roads already considered valid and cannot produce, for example, self-intersecting ones, whereas the join operation the single-road setting is restricted to can. Neither crossover operator nor mutation can guarantee valid results. Given the restrictions imposed on road networks, it is challenging to create valid configurations, but more so using the join and mutation operators which alter road geometry and can thus cause self-intersections or roads that do not start or end at the boundary. Hence, the single-road configuration, which only had join and mutation operator available, generated a larger number of invalid configurations than the multi-road configuration. All in all, these results let us conclude that:

RQ3: Although single-road tests are three time as fast to generate, they revealed fewer OBEs and their generation is more difficult.

5.6 RQ4. How Does the Size of the Map Impact Testing?

To answer the last research question, we configured AsFault to generate multi-road road networks over the small map and tested BeamNG.AI only. As before, we generated test suites composed of 25 tests, used a generation budget of 24 hours, and repeated the experiment for 40 times.

From Figure 9, comparing the results achieved by AsFault generating multi-road tests over the small map ("AsFault Multi Road Small Map") and the large map ("AsFault Multi Road Large Map"), we observe once again the same trend: at the beginning of the execution, both configurations achieved very similar results, but, as we expected, the large map enabled AsFault to expose more OBEs (34.6 on average) than the small map (34.4 on average). During the execution, the results of both configurations improved, however, this time the difference between the OBEs found by AsFault in the two configurations was not as large as before ($\hat{A}_{12} = 0.65$, $p = 0.045$), and by the end of the execution, AsFault using the small map found 61.4 OBEs on average, while AsFault using the large map found 71.1 OBEs on average. From Figure 11, we observe that compared to using the large map, generating tests using the small map was faster both for generating single-road tests ($\hat{A}_{12} = 0.31$, $p < 0.005$) and multi-road tests ($\hat{A}_{12} = 0.3$, $p < 0.005$). On average, to generate single-road tests, AsFault took 10.5 seconds using the small map and 41.1 seconds using the large map, while to generate multi-road tests it took on average 97.0 seconds using the small map and 152.6 using the large map. Additionally, using the small map led to the generation of fewer valid multi-road tests ($\hat{A}_{12} = 0.46$, $p = 0.055$), but more valid single-road tests ($\hat{A}_{12} = 0.54$, $p < 0.005$). This suggests that intersecting multiple roads into a smaller map was slightly more difficult than putting them into a larger one, while using a smaller map was easier to generate roads which intersect the map boundary. In summary, we can conclude that:

RQ4: Large maps led to slightly more effective tests, but small maps allow for substantially faster search.

5.7 Threats to Validity

Internal validity To ensure that our integration of DeepDriving into BeamNG.research is correct, we used the author's guidelines, manually tested the integration, and validated DeepDriving by having it successfully drive along randomly generated roads. While AsFault does not recreate all the elements that can be found in real roads, i.e., trees, weather conditions, all the elements required by DeepDriving are correctly generated. To increase internal validity, we use BeamNG.AI as test subject. Being tightly integrated with the driving simulator, BeamNG.AI has perfect knowledge of the roads, hence it does not suffer from the traditional limitations of vision-based lane keeping systems, and is less sensitive to misconfigurations of the procedurally generated roads. In conducting our experimental evaluation, we followed the guidelines for comparing randomized test generation algorithms presented by Arcuri and Briand [4]; hence, to support our conclusions we repeated each experiment several times, performed significance tests (i.e., Mann–Whitney U-test p-values), and considered the *effect size* measures (i.e., non-parametric Vargha–Delaney's $\hat{A}_{12}$ statistic).

External validity Our evaluation, like any empirical investigation, considered a limited number of test subjects in its experimental setup, so results might not generalize. We only used a single vision-based lane keeping system for evaluation, because most lane keeping systems are not publicly available, and the available ones which are used in similar research (e.g., [40, 50]) implement only a subset of the required functionalities, i.e., steering model. While we therefore cannot say if effectiveness results generalize, our evaluation nevertheless demonstrates that AsFault is able to generate effective test cases under different execution conditions (i.e., types of road networks and maps of different size).

6 RELATED WORK

Self-driving car testing is still a fairly open field, and current approaches to it span from N-FOT to simulation-based tests [20]. In this paper, we propose to combine procedural road generation and genetic algorithms for systematically testing lane keeping software.

Testing vision-based lane keeping software Several works test vision-based lane keeping software by means of adversarial image generation. For example, Müller et al. [27] and Zhang et al. [50] test steering prediction models by simulating the effects of various weather conditions on the images elaborated by lane keeping systems; Tian et al. [40], instead, alter those images by applying transformations which simulate possible distortions introduced by the cameras pointing towards the road. Differently than those approaches, we do not focus on testing only the prediction components of lane keeping systems, but we target the entire system instead. Moreover, we define test cases in term of driving tasks which the ego-car must complete; hence, we can stress the interactions between the prediction components and the driving controllers, and study the closed-loop properties of the ego-car, which is not possible by only considering single images during testing.

Using procedural content generation for testing In the content of testing unmanned vehicles and robots, various works advocate the use of procedural content generation. Arnold and Alexander [5] generate random bi-dimensional bit maps, while Sotiropoulos et al. [38] generate three-dimensional terrains. Compared to those algorithms, which test navigation algorithms, we target a different type of systems under test, i.e., vision-based lane keeping, hence we procedurally generate more realistic environments, i.e. roads. In the context of virtual testing of self-driving cars, Schuldt et al. [33] developed a construction kit for generating PCG-based tests; however, in this work, generating the virtual tests and setting up the environment remain manually activities. AsFault, at the contrary, is fully automated. Similarly to us, Kendall et al. [19] also automatically generate roads; however, Kendall and co-authors focus on developing reinforcement learning algorithms and generate only random roads, while we aim at test generation and follow a systematic approach based on search.

Search-based procedural road generation AsFault generates roads automatically by combining a genetic algorithm and PCG. In this area, the focus is usually the generation of racing tracks, and the goal is to generate tracks which entertain the players. For example, Loiacono et al. [22] automatically generate racing tracks with various curvature and speed profiles which require different driving skills to be completed, while Georgiou et al. [17] and Cardamone et al. [10] used live-feedback from the gameplay to study player reactions and use those information to drive the generation of new tracks which maximize player enjoyment. Those works are our inspiration, hence, we adopt a similar strategy in generating roads. However, our procedural road generation aims at generating realistic roads and not racing track, it uses a different geometrical representations of the roads, i.e., polylines instead of Bézier curves, and, most importantly, targets completely different test subjects, i.e., self-driving cars instead of human players.

Search-based testing of self-driving car software AsFault uses search to systematically generate simulation-based tests. Simulation-based tests require the execution of time-consuming computer simulations, which strongly affects the efficiency of the overall test generation process. To cope with that, some authors optimize the search process in order to minimize the amount of test executions required to generate effective tests. Abdessalem et al.[1, 2] combine genetic algorithms and machine learning to test a pedestrian detection system. The machine learning components speed up the search by predicting the value of the fitness function without executing the tests and drive the test generation towards regions in the input space which most likely expose safety-critical problems. Similarly, Mullins et al. [28] use Gaussian Processes to drive the search towards yet unexplored regions of the input space. We also improve the efficiency of test generation by limiting the amount of executed tests; however, we do so by means of using a different technique based on the similarity of roads.

7 CONCLUSIONS AND FUTURE WORK

Simulation-based testing has emerged as an alternative to the dangerous and costly practice of testing self-driving cars by means of N-FOT; however, simulation-based testing comes with the main challenge of systematically generating tests which expose safety-critical behavior of self-driving cars among the multitude of driving scenarios which can be simulated.

In this paper, we presented AsFault, a novel approach for systematically testing lane keeping systems. AsFault addresses the challenges of simulation-based testing by combining search-based testing and procedural content generation to generate virtual roads which cause self-driving cars to depart from their lane. Our extensive evaluation showed that AsFault can generate effective test suites and expose many safety-critical problems related to lane keeping in a multitude of configurations.

Despite the positive results achieved by AsFault, our work on combining procedural content generation and search-based testing for testing self-driving cars is preliminary; hence, we plan to address some relevant aspects of this research in the future. Our ongoing work includes (i) extending the road generation algorithm to generate roads with varying lanes and width, which follows upland and lowland terrains; (ii) using existing terrains and real roads as seeds for the generation; (iii) enlarging the scope of the test generation algorithm to generate obstacles and traffic on the roads; (iv) extending the procedural content generation algorithm to generate more realistic virtual environments, possibly drawing inspiration from data collected in the real world [48]; (v) investigating alternative search-based techniques, such as novelty search, to generate effective simulation-based tests.

In the future, we also plan to work on improving the efficiency of the test generation by employing surrogate models and machine learning components, and extend AsFault to work with multi-objective fitness functions which can capture additional aspects of driving, such as passenger comfort, in addition to safety.

ACKNOWLEDGMENT

This work is supported by EPSRC project EP/N023978/2.

REFERENCES

[1] Abdessalem, R. B., Nejati, S., Briand, L. C., and Stifter, T. Testing advanced driver assistance systems using multi-objective search and neural networks. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)* (Sept. 2016), pp. 63–74.

[2] Abdessalem, R. B., Nejati, S., Briand, L. C., and Stifter, T. Testing vision-based control systems using learnable evolutionary algorithms. In *Proceedings of the 40th International Conference on Software Engineering* (New York, NY, USA, 2018), ICSE '18, ACM, pp. 1016–1026.

[3] Althoff, M., Urban, S., and Koschi, M. Automatic conversion of road networks from opendrive to lanelets. In *Proceedings of the IEEE International Conference on Service Operations and Logistics, and Informatics* (2018 - to appear), SOLI'18.

[4] Arcuri, A., and Briand, L. A Hitchhiker's Guide to Statistical Tests for Assessing Randomized Algorithms in Software Engineering. *Softw. Test. Verif. Reliab. 24*, 3 (May 2014), 219–250.

[5] Arnold, J., and Alexander, R. Testing Autonomous Robot Control Software Using Procedural Content Generation. In *Computer Safety, Reliability, and Security* (Sept. 2013), Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, pp. 33–44.

[6] Bansal, P., and Kockelman, K. M. Forecasting Americans' long-term adoption of connected and autonomous vehicle technologies. *Transportation Research Part A: Policy and Practice 95* (Jan. 2017), 49–63.

[7] BeamNG GmbH. BeamNG.research. https://beamng.gmbh/research/, 2018.

[8] Bertoncello, M., and Wee, D. Ten ways autonomous driving could redefine the automotive world | McKinsey & Company.

[9] Boissonnat, J.-D., and Teillaud, M., Eds. *Effective Computational Geometry for Curves and Surfaces*. Mathematics and Visualization. Springer-Verlag, Berlin Heidelberg, 2006.

[10] Cardamone, L., Lanzi, P. L., and Loiacono, D. TrackGen: An interactive track generator for TORCS and Speed-Dreams. *Applied Soft Computing 28* (Mar. 2015), 550–558.

[11] Chen, C., Seff, A., Kornhauser, A., and Xiao, J. DeepDriving: Learning Affordance for Direct Perception in Autonomous Driving. In *Proceedings of the International Conference on Computer Vision* (2015), ICCV '15, pp. 2722–2730.

[12] Davies, A. Google's Self-Driving Car Caused Its First Crash. *Wired* (Feb. 2016).

[13] Dosovitskiy, A., Ros, G., Codevilla, F., Lopez, A., and Koltun, V. CARLA: An Open Urban Driving Simulator. *arXiv:1711.03938 [cs]* (Nov. 2017). arXiv: 1711.03938.

[14] Dupuis, M., Strobl, M., and Grezlikowski, H. Opendrive 2010 and beyond–status and future of the de facto standard for the description of road networks. In *Proceedings of the Driving Simulation Conference Europe* (2010), pp. 231–242.

[15] Fritsch, J., Kühnl, T., and Geiger, A. A new performance measure and evaluation benchmark for road detection algorithms. In *16th International IEEE Conference on Intelligent Transportation Systems (ITSC 2013)* (Oct. 2013), pp. 1693–1700.

[16] Gambi, A., Mueller, M., and Fraser, G. AsFault: Testing self-driving car software using search-based procedural content generation. In *Proceedings of the 41th International Conference on Software Engineering* (2019), ICSE '19.

[17] Georgiou, T., and Demiris, Y. Personalised track design in car racing games. In *2016 IEEE Conference on Computational Intelligence and Games (CIG)* (Sept. 2016), pp. 1–8.

[18] Kalra, N., and Paddock, S. M. Driving to safety: How many miles of driving would it take to demonstrate autonomous vehicle reliability? *Transportation Research Part A: Policy and Practice 94* (Dec. 2016), 182–193.

[19] Kendall, A., Hawke, J., Janz, D., Mazur, P., Reda, D., Allen, J.-M., Lam, V.-D., Bewley, A., and Shah, A. Learning to Drive in a Day. *arXiv:1807.00412 [cs, stat]* (July 2018). arXiv: 1807.00412.

[20] Khastgir, S., Birrell, S., Dhadyalla, G., and Jennings, P. Identifying a gap in existing validation methodologies for intelligent automotive systems: Introducing the 3xd simulator. In *2015 IEEE Intelligent Vehicles Symposium (IV)* (June 2015), pp. 648–653.

[21] Koopman, P., and Wagner, M. Challenges in Autonomous Vehicle Testing and Validation. *SAE Int. J. Trans. Safety 4*, 1 (Apr. 2016), 15–24.

[22] Loiacono, D., Cardamone, L., and Lanzi, P. L. Automatic Track Generation for High-End Racing Games Using Evolutionary Computation. *IEEE Transactions on Computational Intelligence and AI in Games 3*, 3 (Sept. 2011), 245–259.

[23] Masuda, S. Software Testing Design Techniques Used in Automated Vehicle Simulations. In *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)* (Mar. 2017), pp. 300–303.

[24] Mauritz, M., Howar, F., and Rausch, A. Assuring the Safety of Advanced Driver Assistance Systems Through a Combination of Simulation and Runtime Monitoring. In *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications* (Oct. 2016), Lecture Notes in Computer Science, Springer, Cham, pp. 672–687. Alessio Marc.

[25] McMinn, P. Search-based software testing: Past, present and future. In *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops* (Washington, DC, USA, 2011), ICSTW '11, IEEE Computer Society, pp. 153–163.

[26] Minnerup, P., and Knoll, A. Testing Automated Vehicles Against Actuator Inaccuracies in a Large State Space. *IFAC-PapersOnLine 49*, 15 (Jan. 2016), 38–43.

[27] Müller, S., Hospach, D., Bringmann, O., Gerlach, J., and Rosenstiel, W. Robustness Evaluation and Improvement for Vision-Based Advanced Driver Assistance Systems. In *2015 IEEE 18th International Conference on Intelligent Transportation Systems* (Sept. 2015), pp. 2659–2664.

[28] Mullins, G. E., Stankiewicz, P. G., and Gupta, S. K. Automated generation of diverse and challenging scenarios for test and evaluation of autonomous vehicles. In *Robotics and Automation (ICRA), 2017 IEEE International Conference on* (2017), IEEE, pp. 1443–1450.

[29] Nentwig, M., Miegler, M., and Stamminger, M. Concerning the applicability of computer graphics for the evaluation of image processing algorithms. In *2012 IEEE International Conference on Vehicular Electronics and Safety (ICVES 2012)* (July 2012), pp. 205–210. Marc.

[30] Netzeband, A. Deepdriving for tensorflow V1.0. https://bitbucket.org/Netzeband/deepdriving/src, 11 2017.

[31] Richter, S. R., Vineet, V., Roth, S., and Koltun, V. Playing for Data: Ground Truth from Computer Games. In *Computer Vision – ECCV 2016* (Oct. 2016), Lecture Notes in Computer Science, Springer, Cham, pp. 102–118.

[32] SAE. Automated Driving Levels. http://www.sae.org/misc/pdfs/automated_driving.pdf, 2013.

[33] Schuldt, F. *Ein Beitrag für den methodischen Test von automatisierten Fahrfunktionen mit Hilfe von virtuellen Umgebungen, Towards testing of automated driving functions in virtual driving environments*. PhD thesis, Technischen Universitaet Braunschweig, 2017.

[34] Sedgewick, R. *Algorithms in C, Part 5: Graph Algorithms, Third Edition*, third ed. Addison-Wesley Professional, 2001.

[35] Shah, S., Dey, D., Lovett, C., and Kapoor, A. AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles. *arXiv:1705.05065 [cs]* (May 2017). arXiv: 1705.05065.

[36] Sippl, C., Bock, F., Wittmann, D., Altinger, H., and German, R. From Simulation Data to Test Cases for Fully Automated Driving and ADAS. In *Testing Software and Systems* (Oct. 2016), Lecture Notes in Computer Science, Springer, Cham, pp. 191–206.

[37] Solon, O. Tesla that crashed into police car was in 'autopilot' mode, California official says. *The Guardian* (May 2018).

[38] Sotiropoulos, T., Guiochet, J., Ingrand, F., and Waeselynck, H. Virtual Worlds for Testing Robot Navigation: A Study on the Difficulty Level. In *Proceedings of the European Dependable Computing Conference* (2016), EDCC '16, pp. 153–160.

[39] Tesla, I. Autopilot. https://www.tesla.com/autopilot.

[40] Tian, Y., Pei, K., Jana, S., and Ray, B. DeepTest: Automated testing of deep-neural-network-driven autonomous cars. *arXiv preprint arXiv:1708.08559* (2017).

[41] Tibba, G., Malz, C., Stoermer, C., Nagarajan, N., Zhang, L., and Chakraborty, S. Testing automotive embedded systems under x-in-the-loop setups. In *Proceedings of the 35th International Conference on Computer-Aided Design* (New York, NY, USA, 2016), ICCAD '16, ACM, pp. 35:1–35:8.

[42] Togelius, J., Champandard, A. J., Lanzi, P. L., Mateas, M., Paiva, A., Preuss, M., and Stanley, K. O. Procedural Content Generation: Goals, Challenges and Actionable Steps. In *Artificial and Computational Intelligence in Games*, S. M. Lucas, M. Mateas, M. Preuss, P. Spronck, and J. Togelius, Eds., vol. 6 of *Dagstuhl Follow-Ups*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2013, pp. 61–75.

[43] Togelius, J., Kastbjerg, E., Schedl, D., and Yannakakis, G. N. What is Procedural Content Generation?: Mario on the Borderline. In *Proceedings of the 2Nd International Workshop on Procedural Content Generation in Games* (New York, NY, USA, 2011), PCGames '11, ACM, pp. 3:1–3:6.

[44] Tuncali, C. E., Fainekos, G., Ito, H., and Kapinski, J. Simulation-based adversarial test generation for autonomous vehicles with machine learning components. In *2018 IEEE Intelligent Vehicles Symposium (IV)* (June 2018), pp. 1555–1562.

[45] UberATG. Self-driving cars return to Pittsburgh roads in manual mode, July 2018.

[46] Waymo. Waymo. https://waymo.com/.

[47] Wehner, M. Video shows Tesla Model S slamming into a wall while driving on Autopilot, Mar. 2017.

[48] Yang, L., Liang, X., Wang, T., and Xing, E. Real-to-virtual domain unification for end-to-end autonomous driving. In *Computer Vision - ECCV 2018 - 15th European Conference, Munich, Germany, September 8-14, 2018, Proceedings, Part IV* (2018), V. Ferrari, M. Hebert, C. Sminchisescu, and Y. Weiss, Eds., vol. 11208 of *Lecture Notes in Computer Science*, Springer, pp. 553–570.

[49] Yenikaya, S., Yenikaya, G., and Düven, E. Keeping the vehicle on the road: A survey on on-road lane detection systems. *ACM Computing Surveys (CSUR) 46*, 1 (2013), 2.

[50] Zhang, M., Zhang, Y., Zhang, L., Liu, C., and Khurshid, S. DeepRoad: GAN-based Metamorphic Autonomous Driving System Testing. *arXiv:1802.02295 [cs]* (Feb. 2018). arXiv: 1802.02295.