arXiv:1804.06760v3 [cs.SY] 25 Jul 2018

Simulation-based Adversarial Test Generation for Autonomous Vehicles with Machine Learning Components *

Cumhur Erkan Tuncali[1] Georgios Fainekos[2] Hisahiro Ito[1] James Kapinski[1]
[1]Toyota Research Institute of North America, Ann Arbor, MI, USA
[2] Arizona State University, Tempe, AZ, USA
cumhur.tuncali@toyota.com, fainekos@asu.edu, {hisahiro.ito, jim.kapinski}@toyota.com

Many organizations are developing autonomous driving systems, which are expected to be deployed at a large scale in the near future. Despite this, there is a lack of agreement on appropriate methods to test, debug, and certify the performance of these systems. One of the main challenges is that many autonomous driving systems have machine learning components, such as deep neural networks, for which formal properties are difficult to characterize. We present a testing framework that is compatible with test case generation and automatic falsification methods, which are used to evaluate cyber-physical systems. We demonstrate how the framework can be used to evaluate closed-loop properties of an autonomous driving system model that includes the ML components, all within a virtual environment. We demonstrate how to use test case generation methods, such as covering arrays, as well as requirement falsification methods to automatically identify problematic test scenarios. The resulting framework can be used to increase the reliability of autonomous driving systems.

1 Introduction

Many groups are developing autonomous driving systems. These systems are on the road now and are expected to have a significant impact on the vehicle market and the broader economy in the near future; however, no generally agreed upon testing or verification methods have arisen for these systems. One reason for this is that the current designs usually include some machine learning (ML) components, such as deep neural networks (DNNs), which are notoriously difficult to test and verify. We present a framework for Simulation-based Adversarial Testing of Autonomous Vehicles (Sim-ATAV), which can be used to check closed-loop properties of autonomous driving systems that include ML components. We describe a testing methodology, based on a test case generation method, called covering arrays, and requirement falsification methods to automatically identify problematic test scenarios. The resulting framework can be used to increase the reliability of autonomous driving systems.

Autonomous driving system designs often use ML components such as DNNs to classify objects within CCD images and to determine their positions relative to the vehicle, a process known as *semantic segmentation* [12]. Other designs use neural networks (NNs) to perform *end-to-end* control of the vehicle, meaning that the NN takes in the image data and outputs actuator commands, without explicitly performing an intermediate step to do the semantic segmentation [30]. Still other approaches use end-to-end learning to do intermediate decisions like risk assessment [33].

The ML system components are problematic from an analysis perspective, as it is difficult or impossible to characterize all of the behaviors of these components under all circumstances. One reason for this is that the complexity of these systems, in terms of number of parameters, can be high. For example, AlexNet, a pre-trained, DNN, which is used for classification of CCD images, has 60 million

*This work was partially funded by NSF awards CNS 1446730, 1350420

This is a modified version of a paper presented at the 29th IEEE Intelligent Vehicles Symposium (IV 2018)

parameters [23]. Another reason for the difficulty in characterizing behaviors of the ML components is that the parameters are learned based on training data. Characterizing the ML behaviors is, in some ways, as difficult as the task of characterizing the training data. Again using the AlexNet example, the number of training images used was 1.2 million. While a strength of DNNs is their ability to generalize from training data; the challenge for analysis is that we do not well understand how they generalize for all possible cases.

There has been significant interest recently on verification and testing for ML components (see Sec. 2). For example, adversarial testing approaches seek to identify perturbations in image data that result in misclassifications. By contrast, our work focuses on methods to determine perturbations in the configuration of a testing scenario, meaning that we seek to find scenarios that lead to unexpected behaviors, such as misclassifications and ultimately collisions. The framework that we present allows this type of testing in a virtual environment. By utilizing advanced 3D models and image rendering tools, such as the ones used in game engines or film studios, the gap between testing in a virtual environment and the real world can be minimized.

Most of the previous work to test and verify systems with ML components focuses only on the ML components themselves, without consideration of the closed-loop behavior of the system. For autonomous driving applications, we observe that the ultimate goal is to evaluate the closed-loop performance, and so the testing methods used to evaluate these systems should reflect this.

The closed-loop nature of a typical autonomous driving system can be described as follows. A *perception* system processes data gathered from various sensing devices, such as cameras, LIDAR, and radar. The output of the perception system is an estimation of the principal (*ego*) vehicle's position with respect to external obstacles (e.g., other vehicles, called *agent* vehicles, and pedestrians). A *path planning* algorithm uses the output of the perception system to produce a short-term plan for how the ego vehicle should behave. A *tracking controller* then takes the output of the path planner and produces actuation outputs, such as accelerator, braking, and steering commands. The actuation commands affect the vehicle's interaction with the environment. The iterative process of sensing, processing, and actuating is what we refer to as closed-loop behavior.

Our contributions can be summarized as follows. We provide a new algorithm to perform falsification of formal requirements for an autonomous vehicle in a closed-loop with the perception system, which includes an efficient means of searching over discrete and continuous parameter spaces. Our method represents a new way to do adversarial testing in scenario *configuration space*, as opposed to the usual method, which considers adversaries in *image space*. Additionally, we demonstrate a new way to characterize problems with perception systems in *configuration space*. Lastly, we extend software testing notions of *covering arrays* to closed-loop cyber-physical system (CPS) applications based on ML.

2 Related work

Verification and testing for NNs is notoriously difficult since NNs correspond to complex nonlinear and non-convex functions. Currently, the methods developed for the analysis of NN components can be classified into two main categories.

The first category concerns output range verification for a given bounded set of input values [9, 17, 20, 31]. The methods in this category use a combination of Satisfiability Modulo Theory (SMT) solvers, Linear Programing (LP), gradient-based local search, and Mixed Integer Linear Programming (MILP). The second category deals with adversarial sample generation ([29, 34, 28, 27, 14, 4]), which addresses the problem of how to minimally perturb the input to the NN so that the classification decision of the

NN changes. If these perturbations are very small, then in a sense, the NN is not robust. The methods typically employed to solve this problem primarily use some form of gradient ascent to adjust the input so that the output of the NN changes. The differences between the various approaches relate to whether the system is black-box or white-box and how the gradient is computed or approximated. Under this category, we could also potentially include generative adversarial networks [13].

All of the aforementioned methods deal with the verification and testing of NNs at the component level; however, our work targets the NN testing problem at the system level. The line of research that is the closest in spirit to our work is [7, 6, 8]. The procedure described in [7, 6, 8] analyzes the performance of the perception system using static images to identify candidate counterexamples, which they then check using simulations of the closed-loop behaviors to determine whether the system exhibits unsafe behaviors. We, on the other hand, provide a new method to search for unsafe behaviors directly on the closed-loop behaviors of the system, meaning our search uses a global optimizer guided by a cost function that is defined based on the closed-loop behaviors.

The goals of our approach are similar to those of [26], though their framework is not capable of simulating the autonomous vehicle system in a closed-loop, including the perception system. In contrast to our work, their framework utilize a game engine only to visualize the results after the testing and analysis is done on well defined, but simpler, vehicle dynamic models like a bicycle model, which may not well represent real behaviors.

Beyond NN-focused testing and analysis, our work borrows ideas from robustness guided falsification for autonomous vehicles [35], where the goal is to detect boundary-case failures. Finally, our work falls under the broader context of automatic test generation methods for autonomous vehicles and driver assist systems [22, 21], but our methods and testing goals are different.

3 Preliminaries

This section presents the setting used to describe the testing procedures performed using our framework. The purpose of our framework is to provide a mechanism to test, evaluate, and improve on an autonomous driving system design. To do this, we use a simulation environment that incorporates models of a vehicle (called the *ego* vehicle), a perception system, which is used to estimate the state of the vehicle with respect to other objects in its environment, a controller, which makes decisions about how the vehicle will behave, and the environment in which the ego vehicle exists. The environment model contains representations of a wide variety of objects that can interact with the ego vehicle, including roads, buildings, pedestrians, and other vehicles (called *agent* vehicles). The behaviors of the system are determined by the evolution of the model states over time, which we compute using a *simulator*.

Formally, the framework implements a model of the system, which is a tuple

$$M = (\mathscr{X}, \mathscr{U}, \mathscr{P}, sim),$$

where $\mathscr{X}$ is a set of system states, $\mathscr{U}$ is a set of inputs, and *sim* is a simulation function

$$sim : \mathscr{X} \times \mathscr{U} \times \mathscr{P} \times T \to \mathscr{X},$$

where T is a discrete set of sample times $t_0, t_1, \ldots, t_N$, with $t_i < t_{i+1}$. $\mathscr{P} = \mathscr{W} \times \mathscr{V}$ is a combination of continuous-valued and discrete-valued parameters, where $\mathscr{W} = \mathscr{W}_1 \times \cdots \times \mathscr{W}_W$ and each $\mathscr{W}_i \subseteq \mathbb{R}$, and $\mathscr{V} = \mathscr{V}_1 \times \cdots \times \mathscr{V}_V$ and each $\mathscr{V}_i$ is some finite domain, such as Boolean or a finite list of agent car colors.

Given $\mathbf{x} \in \mathscr{X}$, $\hat{\mathbf{x}} = sim(\mathbf{x}, \mathbf{u}, \mathbf{p}, t)$ is the state reached starting from state $\mathbf{x}$ after time $t \in T$ under input $\mathbf{u} \in \mathscr{U}$ and parameter value $\mathbf{p} \in \mathscr{P}$. We call a sequence

$$U = (\mathbf{u}_0, t_0)(\mathbf{u}_1, t_1) \cdots (\mathbf{u}_N, t_N),$$

where each $\mathbf{u}_i \in \mathscr{U}$ and $t_i \in T$, an input trace of M. Given a model M, an input trace of M, U, and a $\mathbf{p} \in \mathscr{P}$, a simulation trace of M under input U and parameters $\mathbf{p}$ is a sequence

$$T = (\mathbf{x}_0, \mathbf{u}_0, t_0)(\mathbf{x}_1, \mathbf{u}_1, t_1) \cdots (\mathbf{x}_N, \mathbf{u}_N, t_N),$$

where $sim(\mathbf{x}_{i-1}, \mathbf{u}_{i-1}, \mathbf{p}, t_{i-1}) = \mathbf{x}_i$ for each $1 \leq i \leq N$. For a given simulation trace T, we call $X = (\mathbf{x}_0, t_0)(\mathbf{x}_1, t_1) \cdots (\mathbf{x}_N, t_N)$ the state trace. We denote the set of all simulation traces of M by $\mathscr{L}(M)$.

3.1 Signal Temporal Logic

Signal Temporal Logic (STL) was introduced as an extension to Metric Temporal Logic (MTL) [3] to reason about real-time properties of signals (simulation traces). STL formulae are built over predicates on the variables of a signal using combinations of Boolean and temporal operators. The temporal operators include *eventually* ($\Diamond_{\mathscr{I}}$), *always* ($\Box_{\mathscr{I}}$) and *until* ($\mathscr{U}_{\mathscr{I}}$), where $\mathscr{I}$ encodes timing constraints.

In this work, we interpret STL formulas over the observable simulation traces of a given system. STL specifications can describe the usual properties of interest in system design such as (bounded time) **reachability**, e.g., *between time 1 and 5,* $\mathbf{x}$ *should drop below* -10: $\Diamond_{[1,5)}$ $(\mathbf{x} \leq -10)$, and **safety**, e.g., *after time 2,* $\mathbf{x}$ *should always be greater than* 10: $\Box_{[2,+\infty)}(\mathbf{x} \geq 10)$. Beyond the usual properties, STL can capture sequences of events, e.g., $\Diamond_{\mathscr{I}_1}(\pi_1 \wedge \Diamond_{\mathscr{I}_2}(\pi_2 \wedge \Diamond_{\mathscr{I}_3}\pi_3))$, and infinite behaviors, e.g., **periodic** behaviors : $\Box(\pi_1 \rightarrow \Diamond_{[0,2]}\pi_2)$, where π_i are predicates over signal variables.

Informally speaking, we allow predicate expressions to capture arbitrary constraints over the state variables, inputs and parameters of the system. In other words, we assume that predicates π are expressions built using the grammar $\pi ::= f(\mathbf{x}, \mathbf{u}, \mathbf{p}) \geq c \mid \neg\pi_1 \mid (\pi) \mid \pi_1 \vee \pi_2 \mid \pi_1 \wedge \pi_2$, where f is a function and c is a constant in $\mathbb{R}$. Effectively, each predicate π represents a subset in the space $\mathscr{X} \times \mathscr{U} \times \mathscr{P}$. In the following, we represent that set that corresponds to the predicate π using the notation $\mathscr{O}(\pi)$. For example, if $\pi = (x^{(1)} \leq -10) \vee (x^{(1)} + x^{(2)} \geq 10)$ and we represent by $x^{(i)}$ the i-th component of the vector $\mathbf{x}$, then $\mathscr{O}(\pi) = (\infty, -10] \times \mathbb{R} \cup \{x \in \mathbb{R}^2 \mid x^{(1)} + x^{(2)} \geq 10\}$.

Definition 1 (STL Syntax) *Assume* Π *is the set of predicates and* $\mathscr{I}$ *is any non-empty interval of* $\mathbb{R}_{\geq 0}$. *The set of all well-formed STL formulas is inductively defined as*

$$\varphi \ ::= \ \top \mid \pi \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \bigcirc \phi \mid \phi_1 U_{\mathscr{I}} \phi_2,$$

where π *is a predicate,* $\top$ *is* true, $\bigcirc$ *is Next and* $U_{\mathscr{I}}$ *is Until operator.*

For STL formulas ψ, ϕ, we define

$$\begin{aligned}
\wedge\phi &\equiv \neg(\neg\psi \vee \neg\phi), \\
\bot &\equiv \neg\top && \text{(False)}, \\
\psi \rightarrow \phi &\equiv \neg\psi \vee \phi && (\psi \text{ Implies } \phi), \\
\Diamond_I \psi &\equiv \top U_I \psi && \text{(Eventually } \psi), \\
\Box_I \psi &\equiv \neg\Diamond_I \neg\psi && \text{(Always } \psi), \\
\psi R_I \phi &\equiv \neg(\neg\psi U_I \neg\phi) && (\psi \text{ Releases } \phi)
\end{aligned}$$

using syntactic manipulation.

In our previous work [11], we proposed robust semantics for STL formulas. Robust semantics (or robustness metrics) provide a real-valued measure of satisfaction of a formula by a trace in contrast to the Boolean semantics that just provide a *true* or *false* valuation. In more detail, given a trace T of the system, its robustness w.r.t. a temporal property φ, denoted $[\![\varphi]\!]_{\mathbf{d}}(T)$ yields a positive value if T satisfies φ and a negative value otherwise. Moreover, if the trace T satisfies the specification ϕ, then the robust semantics evaluate to the radius of a neighborhood such that any other trace that remains within that neighborhood also satisfies the same specification. The same holds for traces that do not satisfy ϕ.

Definition 2 (STL Robust Semantics) *Given a metric* $\mathbf{d}$*, trace* T *and* $\mathscr{O} : \Pi \to 2^{\mathscr{X}\times\mathscr{U}\times\mathscr{P}}$*, the robust semantics of any formula* ϕ *w.r.t* T *at time instance* $i \in N$ *is defined as:*

$$
\begin{aligned}
[\![\top]\!]_{\mathbf{d}}(T,i) &:= +\infty \\
[\![\pi]\!]_{\mathbf{d}}(T,i) &:= \begin{cases} -\inf\{\mathbf{d}((\mathbf{x}_i,\mathbf{u}_i,\mathbf{p}_i),\mathbf{y}) \mid \mathbf{y} \in \mathscr{O}(\pi)\} & \text{if } (\mathbf{x}_i,\mathbf{u}_i,\mathbf{p}_i) \notin \mathscr{O}(\pi) \\ \inf\{\mathbf{d}((\mathbf{x}_i,\mathbf{u}_i,\mathbf{p}_i),\mathbf{y}) \mid \mathbf{y} \in \overline{\mathscr{O}(\pi)}\} & \text{if } (\mathbf{x}_i,\mathbf{u}_i,\mathbf{p}_i) \in \mathscr{O}(\pi) \end{cases} \\
[\![\neg\phi]\!]_{\mathbf{d}}(T,i) &:= -[\![\phi]\!]_{\mathbf{d}}(T,i) \\
[\![\phi_1 \vee \phi_2]\!]_{\mathbf{d}}(T,i) &:= \max\left([\![\phi_1]\!]_{\mathbf{d}}(T,i), [\![\phi_2]\!]_{\mathbf{d}}(T,i)\right) \\
[\![\bigcirc\phi]\!]_{\mathbf{d}}(T,i) &:= \begin{cases} [\![\phi]\!]_{\mathbf{d}}(T,i+1) & \text{if } i+1 \in N \\ -\infty & \text{otherwise} \end{cases} \\
[\![\phi_1 U_{\mathscr{I}} \phi_2]\!]_{\mathbf{d}}(T,i) &:= \max_{j \text{ s.t. } (t_j - t_i)\in\mathscr{I}} \left(\min\left([\![\phi_2]\!]_{\mathbf{d}}(T,j), \min_{i\le k<j} [\![\phi_1]\!]_{\mathbf{d}}(T,k) \right) \right)
\end{aligned}
$$

A trace T satisfies an STL formula ϕ (denoted by $T \models \phi$), if $[\![\phi]\!]_{\mathbf{d}}(T,0) > 0$. On the other hand, a trace T' falsifies ϕ (denoted by $T' \not\models \phi$), if $[\![\phi]\!]_{\mathbf{d}}(T',0) < 0$. Algorithms to compute $[\![\varphi]\!]_{\mathbf{d}}$ have been presented in [11, 10, 5].

Example 3.1 *Consider the following dynamical system:*

$$
\begin{aligned}
\dot{x}_1 &= x_1 - x_2 + 0.1t \\
\dot{x}_2 &= x_2\cos(2\pi x_2) - x_1\sin(2\pi x_1) + 0.1t
\end{aligned}
$$

Sample system trajectories with initial conditions over a grid of 0.05 intervals in each dimension over the set of initial conditions $[-1,1]^2$ *are presented in Fig. 1. Using the specification*

$$
\begin{aligned}
\varphi = &\Box\neg(x \in [-1.6,-1.4]\times[-1.1,-0.9]) \\
&\wedge \Box\neg(x \in [3.4,3.6]\times[-1.2,-0.8]),
\end{aligned}
$$

we can construct the robustness surface in Fig. 2. Informally, the specification states that all system trajectories must always not enter any of the red boxes in Fig. 1. Therefore, any falsifying system behavior will enter either of the red boxes. Note that the regions of the initial conditions that initiate falsifying trajectories correspond to negative values in the robustness landscape in Fig. 2.

3.2 Robustness-Guided Model Checking (RGMC)

The goal of a *model checking* algorithm is to ensure that all traces satisfy the requirement. The robustness metric can be viewed as a fitness function that indicates the degree to which individual executions of the

system satisfy the requirement φ, with positive values indicating that the execution satisfies φ. Therefore, for a given system M and a given requirement φ, the model checking problem is to ensure that for all $T \in \mathscr{L}(M)$, $[\![\varphi]\!]_{\mathbf{d}}(T) > 0$.

Let φ be a given STL property that the system is expected to satisfy. The robustness metric $[\![\varphi]\!]_{\mathbf{d}}$ maps each simulation trace T to a real number r. Ideally, for the STL verification problem, we would like to prove that $\inf_{y \in \mathscr{L}(\Sigma)} \mathscr{R}_{\varphi}(y) > \varepsilon > 0$ where ε is a desired robustness threshold.

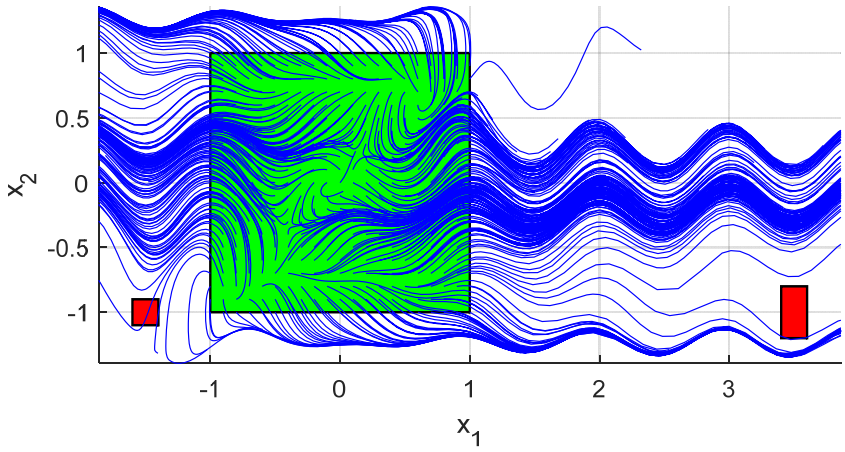


Figure 1: System trajectories.

3.3 Falsification and Critical System Behaviors

In this work, we focus on identifying critical counterexamples, which we refer to as *glancing* examples. For the autonomous driving system, and its corresponding requirement, described below, robustness values that are negative and large in magnitude will correspond to traces where the ego vehicle collides with an object at high velocity. Glancing counterexamples are those that correspond to robustness values that are close to 0, where the ego vehicle collides with an object at very low speeds. Glancing counterexamples are valuable for designers, as they provide examples of behaviors at the boundary between satisfying and falsifying the given property [35]. As discussed in [32], it is important to extract the cases where the collision is avoidable, instead of failures caused by a pedestrian or an agent vehicle making a maneuver leading to an unavoidable collision.

To identify critical system behaviors, we leverage existing work on *falsification*, which is the process of identifying system traces T that do not satisfy φ. For the STL falsification problem, falsification attempts to solve the problem: Find $T \in \mathscr{L}(\Sigma)$ s.t. $[\![\varphi]\!]_{\mathbf{d}}(T) < 0$. This is done using best effort solutions to the following optimization problem: $T^{\star} = \arg\min_{T \in \mathscr{L}(\Sigma)} [\![\varphi]\!]_{\mathbf{d}}(T)$. If $[\![\varphi]\!]_{\mathbf{d}}(T^{\star}) < 0$, then a counterexample (adversarial sample) has been identified, which can be used for debugging or for training. In order to solve this non-linear non-convex optimization problem, a number of stochastic search optimization methods can be applied (e.g., [2] – for an overview see [16, 19]).

We use falsification methods to identify glancing examples for an autonomous driving system, where

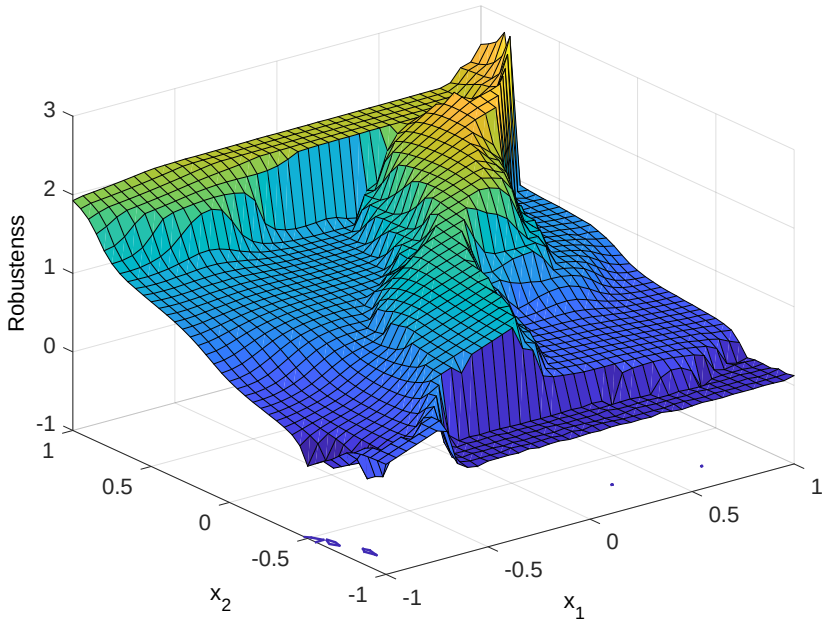


Figure 2: The resulting robustness landscape for specification φ.

we identify

$$\hat{T}^{\star} = \arg \min_{T \in \mathscr{L}(\Sigma)} |[\![\varphi]\!]_{\mathbf{d}}(T)| \tag{1}$$

as the glancing instance. The minimum of $|[\![\varphi]\!]_{\mathbf{d}}(T)|$ for the autonomous driving system, which is usually 0, can be designed to correspond to a trace where a collision occurs with a small velocity.

3.4 Covering Arrays

In software systems, there can often be a large number of discrete input parameters that affect the execution path of a program and its outputs. The possible combinations of input values can grow exponentially with the number of parameters. Hence, exhaustive testing on the input space becomes impractical for fairly large systems. A fault in such a system with k parameters may be caused by a specific combination of t parameters, where $1 \leq t \leq k$. One best-effort approach to testing is to make sure that all combinations of any t-sized subset (i.e., all t-way combinations) of the inputs is tested.

A *covering array* is a minimal number of test cases such that any t-way combination of test parameters exist in the list [15]. Covering arrays are generated using optimization-based algorithms with the goal of minimizing the number of test cases. We denote a t-way covering array on k parameters by $CA(t,k,(v_1,...,v_k))$, where v_i is the number of possible values for the i^{th} parameter. The size of covering array increases with increasing t, and it becomes an exhaustive list of all combinations when $t = k$. Here, t is considered as the *strength* of the covering array. In practice, t can be chosen such that the generated tests fit into the testing budget. Empirical studies on real-world examples show that more than 90 percent of the software failures can be found by testing 2 to 4-way combinations of inputs [24].

Despite the t-way combinatorial coverage guaranteed by covering arrays, a fault in the system possibly may arise as a result of a combination of a number parameters larger than t. Hence, covering arrays

are typically used to supplement additional testing techniques, like uniform random testing. We consider that because of the nature of the training data or the network structure, NN-based object detection algorithms may be sensitive to a certain combination of properties of the objects in the scene. In Sec. 5, we describe how Sim-ATAV combines covering arrays to explore discrete and discretized parameters with falsification on continuous parameters.

4 Framework

We describe Sim-ATAV, a framework for performing testing and analysis of autonomous driving systems in a virtual environment. The simulation environment used in Sim-ATAV includes a vehicle perception system, a vehicle controller, a model of the physical environment, and a mechanism to render 2D images from 3D models. The framework uses freely available and low cost tools and can be run on a standard desktop PC. Later, we demonstrate how Sim-ATAV can be used to implement traditional testing approaches, as well as advanced automated testing approaches that were not previously possible using existing frameworks.

Fig. 3 shows an overview of the simulation environment. The simulation environment has three main components: the perception system, the controller, and the environment modeling framework, which also performs image rendering.

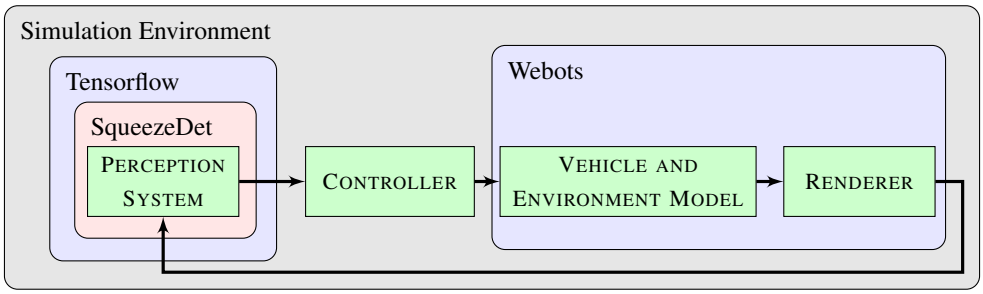


Figure 3: Overview of the simulation environment.

For the perception system, we feed the vehicle front camera images to a lightweight Deep NN, SqueezeDet, which performs object detection and classification [36]. SqueezeDet is implemented in TensorFlow™[1], and it outputs a list of object detection boxes with corresponding class probabilities. This network was originally trained on real image data from the KITTI dataset [12] to achieve accuracy comparable to the popular AlexNet classifier [23]. We further train this network on the virtual images generated in our framework.

For our experiments, we use only camera information and neglect other sensors that would typically be available on autonomous automobile systems, such as LIDAR and radar. In principle, this means that the controller that we use is likely more sensitive to the performance of the image-based perception systems, as compared to a typical autonomous driving system, since the computer vision system is essentially the only sensor information that we use. This assumption was made for practical considerations, but we argue that this is acceptable for our prototype framework, as this allows us to focus our attention on the NN-based computer vision system, which is often considered to be the most difficult aspect of autonomous driving systems to test. Having said that, we realize that this assumption is significant; future work will include adding models of other typical sensing systems, such as LIDAR and radar.

For our evaluations of the proposed testing framework, we implemented a simple *collision avoidance* controller in Python. The controller takes the detection box and object class information from SqueezeDet and estimates the actual positions of the detected objects (e.g., pedestrians or vehicles) by a linear mapping of the detection box pixel positions and sizes to the 3D space. The controller also utilizes the Median Flow tracker [18] implementation in OpenCV to track and estimate future positions of the objects. A collision is predicted if an object is, or in the future will be, in front of the controlled vehicle at a distance shorter than a threshold. When there is no collision risk, the controller drives the car with a constant throttle. When a future collision is predicted, it applies the brakes at the maximum level, and if the predicted time to collision is less than a threshold, it also steers away from the object to avoid a possible collision. We note that Sim-ATAV does not restrict the controller; our controller can be easily exchanged with an alternative.

The environment modeling framework is implemented in Webots [25], a robotic simulation framework that models the physical behavior of robotic components, such as manipulators and wheeled robots, and can be configured to model autonomous driving scenarios. In addition to modeling the physics, a graphics engine is used to produce images of the scenarios. In Sim-ATAV, the images rendered by Webots are configured to correspond to the image data captured from a virtual camera that is attached to the front of a vehicle.

The process used by Sim-ATAV for test generation and execution for discrete and discretized continuous parameters is illustrated by the flowchart shown in Fig. 4 (left). Sim-ATAV first generates test cases that correspond to scenarios defined in the simulation environment using covering arrays as a combinatorial test generation approach. The scenario setup is communicated to the simulation interface using TCP/IP sockets. After a simulation is executed, the corresponding simulation trace is received via socket communication and evaluated using a cost function. Among all discrete test cases, the most promising one is used as the initial test case for the falsification process shown in Fig. 4 (right). For falsification, the result obtained from the cost function is used in an optimization setting to generate the next scenario to be simulated. For this purpose, we used S-TaLiRo [10], which is a MATLAB® toolbox for falsification of CPSs. Similar tools, such as Breach [5], can also be used in our framework for the same purpose.

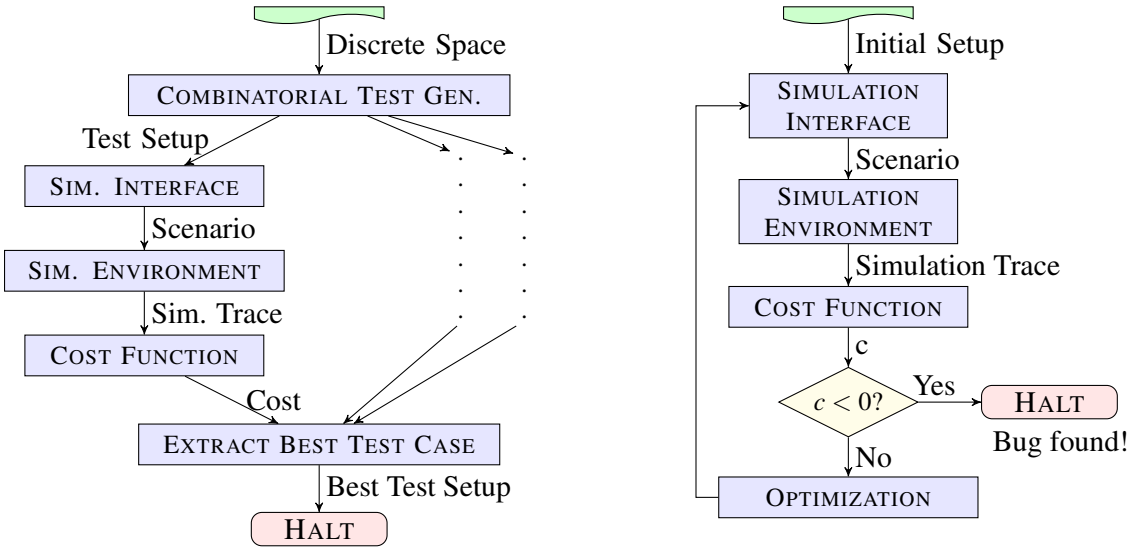


Figure 4: Flowcharts illustrating the combinatorial testing (left) and falsification (right) approaches.

5 Testing Application

In this section, we present the results from a case study using an autonomous driving scenario to evaluate Sim-ATAV. We describe a specific driving scenario and provide the corresponding STL specification. Then, we describe the testing procedure that we use to identify critical (glancing) system behaviors, which is based on covering arrays and robustness-guided falsification. Finally, we analyze the results.

5.1 Scenario Setup

In this section, we describe the driving scenario that we use to evaluate Sim-ATAV. The scenario is selected to be both challenging for the autonomous driving system and also analogous to plausible driving scenarios experienced in real world situations. In general, the system designers will need to identify crucial driving scenarios, based on intuition about challenging situations, from the perspective of the autonomous vehicle control system. A thorough simulation-based testing approach will include a wide array of scenarios that exemplify critical driving situations.

We use the scenario depicted in Fig. 5, which we call system M_d. The scenario includes the ego car on a one-way road, with parked cars in lanes on the left and right of the vehicle. In the Figure, agent Vehicles 1, 3, 4, 5, and 6 are parked. Also, Vehicle 2 is stopped due to the two pedestrians in the crosswalk. The Jay-walking Pedestrian is crossing the street between Vehicles 4 and 5. This pedestrian crosses in front of the Ego car until it reaches the leftmost lane, at which time it changes direction and returns to the right Sidewalk.

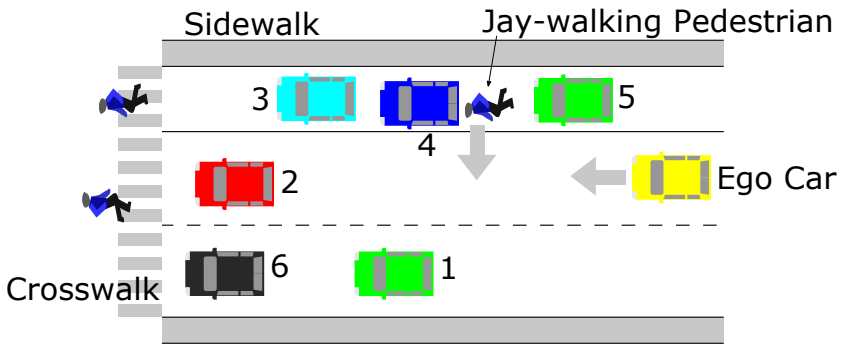


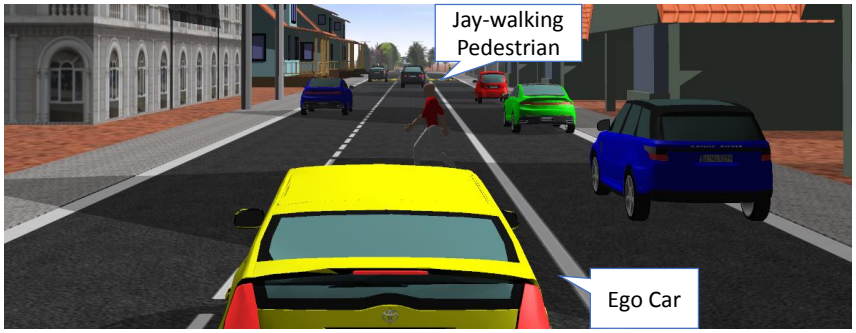
Figure 5: Overview of the scenario used for evaluations.

Several aspects of the scenario depicted in Fig. 5 are parameterized, meaning that their values are fixed for any given simulation by appropriately selecting the model parameters $\mathbf{p}_d$. The longitudinal position of Vehicle 1 is parameterized. Also the colors of Vehicles 1 through 5 and the shirt and pants of the Jay-walking Pedestrian are parameterized and can each take one of 5 values: red, green, blue, white, or black. Additionally, Vehicles 1 through 5 can be one of 5 different car models. Further, the walking speed of the Jay-walking Pedestrian is parameterized, along with the initial longitudinal position of the Ego car. Fog may be present in the scene, and this corresponds to a Boolean variable that determines whether the fog is present or not. In all, M_d has 13 discrete ($V = 13$) and 3 continuous parameters ($W = 3$).

Figure 6 shows examples of the scenario described above. The images are generated by the Webots framework, which implements the physics model and provides the image rendering of the scene.

Figure 6a shows an example of the scene during a simulation run performed using the Sim-ATAV framework. The Ego car is in the bottom-center of the image, and the Jay-walking Pedestrian can be seen just in front of the Ego car. The parked cars can be seen in front of the Ego car on the left and right sides.

Figure 6b shows an example of a camera image that is processed by the DNN, SqueezeDet. This image corresponds to the scene shown in Fig. 6a. The object detection and classification performed by SqueezeDet is used to estimate where vehicles and pedestrians appear in the image, and the corresponding estimates are shown with red bounding boxes. The solid blue boxes indicate vehicles or pedestrians detected that the perception system determines are in danger of being collided with, unless action is taken by the controller.



(a) Overview of testing scenario

(b) Camera view and object detection and classification on the image

Figure 6: Webots images from testing scenario.

5.2 STL Specifications

Below is the STL specification used in our experiments:

$$\phi_d = \square(\pi_{v_{ego},mov.} \implies (\neg\pi_{0,coll} \wedge \ldots \wedge \neg\pi_{l,coll} \wedge \neg\pi_{0,\overline{coll}} \wedge \ldots \wedge \neg\pi_{l,\overline{coll}}))$$

where, for all $i \in \{1,..,l\}$,

$$\begin{aligned}\pi_{v_{ego},mov.} &= v_{ego} > \varepsilon_{speed}\\ \pi_{i,coll} &= dist(i,ego) < \varepsilon_{dist} \wedge front(i,ego) = \top\\ \pi_{i,\overline{coll}} &= dist(i,ego) < 0.\end{aligned}$$

In the above specification, the Ego vehicle is indexed with *ego*, and all other objects in the environment are indexed from 1 to l. Position of an object is represented with $\mathbf{x}$. $front(i, ego)$ evaluates to true ($\top$) if object i is partially in the front corridor of the Ego vehicle and to false ($\bot$) otherwise. $dist(i, ego)$ gives the minimum Euclidean distance between front bumper of the Ego vehicle and the object i. The specification basically describes that the Ego vehicle should not hit another vehicle or pedestrian. The predicates $\pi_{\overline{i,coll}}, \forall i \in \{1, ..., l\}$ are added so that the robustness corresponds to the Euclidean distance of the corresponding object when it is not in front corridor of the Ego vehicle.

5.3 Testing Strategies

In this section, we describe three testing strategies that we implement in Sim-ATAV. We compare their performance for the scenario described in Sec. 5.1. Because of the simple logic of our controller and the nature of the test scenario, it is relatively easy to find test cases that result in a bad behavior with respect to the system requirement; however, finding test cases that correspond to behaviors that are on the boundary between safe and unsafe, that is, glancing cases, is a challenging task. In our tests, we search for glancing cases (*i.e.,* $\hat{T}^\star$, as defined in Eq. 1 in Sec. 3.3), where a collision occurs at very low speeds or a collision is nearly missed.

5.3.1 Global Uniform Random Search (Global UR)

One naive approach to testing the autonomous driving system M_d against property ϕ_d is to select a collection of J test cases (parameters $\mathbf{p}_{d_j} \in \mathscr{P}_d$, where $1 \leq j \leq J$) at random, run simulations to obtain the corresponding traces $T_{d_j} \in \mathscr{L}(M_d)$, and then check whether the traces satisfy the specification, by checking whether $[\![\phi_d]\!]_{\mathbf{d}}(T_{d_j}) > 0$ for all $1 \leq j \leq J$. To indentify the critical (glancing) cases using Global UR, we use $\arg\min_{1 \leq j \leq J} |[\![\phi_d]\!]_{\mathbf{d}}(T_{d_j})|$. Global UR may not be efficient, as it is not guided towards critical areas. The alternative methods below address this issue.

5.3.2 Covering Arrays and Uniform Random Search (CA+UR)

To provide guidance towards behaviors that demonstrate critical behaviors for system M_d, we use a covering array, which is generated by the ACTS tool [24], for the discrete variables $\mathscr{V}_d$, and we combine this with a falsification approach that we use to explore the continuous variables $\mathscr{W}_d$.

The CA+UR method first creates a list of test cases T_{d_j}, where $1 \leq j \leq J$, and evaluates against the specification ϕ_d. The covering array is generated on the discrete variables $\mathscr{V}_d$ and discretized version of the continuous variables $\hat{\mathscr{W}}_d$.

To identify a glancing behavior, the discrete parameters $\mathbf{v}_{d_j}$ from the covering array that correspond to the T_{d_j} that minimizes $|[\![\phi_d]\!]_{\mathbf{d}}(T_{d_j})|$ over $1 \leq j \leq J$ are used to create the test cases using the uniform random method over the continuous variables. T_{d_k} that corresponds to the minimum value of $|[\![\phi_d]\!]_{\mathbf{d}}(T_{d_j})|$ is then taken as a glancing behavior.

The CA+UR method provides some benefits over the Global UR method, as it first uses covering arrays to identify promising valuations for the discrete variables. One problem with CA+UR is that the search over the continuous variables is still performed in an unguided fashion. Next, we describe another approach that addresses this problem.

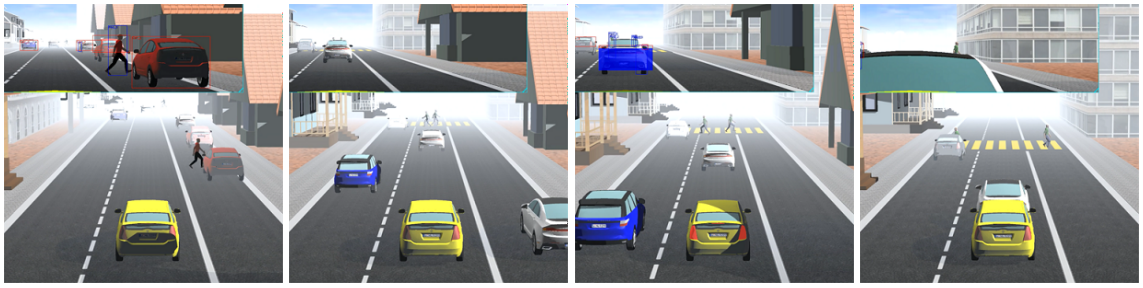
5.3.3 Covering Arrays and Simulated Annealing (CA+SA)

The third method that we evaluate combines covering arrays to evaluate the discrete variables $\mathcal{V}_d$ and uses simulated annealing to search over the continuous variables $\mathcal{W}_d$.

The CA+SA method is identical to the CA+UR method, except that, instead of generating continuous parameters $\mathbf{w}_{d_k}$ offline, a cost function is used to guide a search of the continuous variables. To identify glancing behaviors, we use $|[\![\phi_d]\!]_{\mathbf{d}}(T_{d_j})|$ as the cost function.

Fig. 7a illustrates a test run from a non-failing but close to failing covering array test. In this example, even though there had been detection failures in the perception system in detecting the white vehicle ahead, they did not result in a collision, as the Ego vehicle was able to take a steering action and avoid the collision. By utilizing Simulated Annealing over the continuous parameters, while keeping the discrete parameters unchanged, we can search for a collision and find a behavior like the one illustrated in Fig. 7b, where the detection failures in the perception for the white vehicle ahead leads to a rear-end collision.

(a)



(b)

Figure 7: Time-ordered images from (a) a non-failing test from the covering array and (b) a failure detected by falsification.

5.4 Experiment Results

We applied the testing strategies described in Sec. 5.3 to M_d. Fig. 8 illustrates how the glancing-case robustness values fit to a truncated normal distribution for different test approaches, together with the mean robustness values. The robustness values closer to zero represent a better performance for the related test approach, as they correspond to behaviors where a collision either nearly occurred or nearly did not occur. For each test approach we perform 20 trials, each with a total budget of 200 simulations.

For Global UR, the average robustness values reported are the mean of the robustness values with minimum absolute value over the 20 trials.

For the CA+UR and CA+SA approaches, we first generate a 2-way covering array:

$$CA(2, 16, (v_1, ..., v_{16}))$$

where $v_1, ..., v_{16}$ are each the number of possible discrete values a parameter can take. The number of possible values is $v_1 = ... = v_5 = 5$ for the colors of the vehicles, $v_6 = ... = v_{10} = 5$ for the models of the vehicles, $v_{11}, v_{12} = 5$ for the Jay-walking Pedestrian shirt and pants colors, $v_{13} = 2$ for the existence (True/False) of fog, $v_{14} = 4$ for the discretized space of the Ego vehicle position, $v_{15} = 4$ for the discretized space of the agent Vehicle 1 position, and $v_{16} = 4$ for the discretized space of the Pedestrian speed. The size of that covering array is 47, and it covers all possible 2,562 2-way combinations, while the number of all possible combinations of all parameters would be $5^{12} \cdot 2 \cdot 4^3$.

Each of the CA+UR and CA+SA runs begins with the 47 covering array test cases (i.e., these runs account for 47 of the 200 simulations allotted to each trial). Then, for the test cases that have the closest robustness value to their target (robustness values closest to 0), the search continues over the real-valued space for 153 additional simulations. Hence, each CA+UR and CA+SA trial has a total budget of 200 simulations, which is equal to the simulation budget for each Global UR trial. Also, for the CA+UR and CA+SA test cases, we limit the maximum number of search iterations (simulations) starting from a given discrete case ($\mathbf{v}_{d_j}$) to 50, at which time we restart the search at the *next best* discrete case and perform another search; this repeats until the total simulation budget of 200 is spent. For CA+UR and CA+SA, as we target finding a glancing case (close to zero robustness value), we take the absolute value of the robustness.

Fig. 8 shows that, compared to the Global UR method, the CA+UR method does a better job of identifying glancing cases and also has the advantage of guaranteeing that any 2-way combination among the parameters are covered in our search space, while still using an unguided approach to explore the real-valued space.

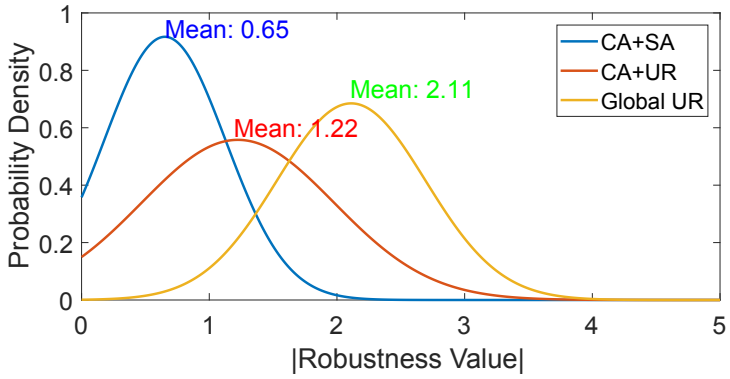


Figure 8: Probability distribution of the robustness values.

The CA+SA method does better than both the Global UR and CA+UR method in identifying glancing behaviors. As can be seen in Fig. 8, CA+SA, on average, can find the test cases that generate a behavior closer to the target (i.e., close to 0). The CA+SA approach, while maintaining the coverage guarantees inherited from covering arrays, also applies falsification in the continuous space, and so CA+SA is expected to perform better than either of the other two methods. The results in Fig. 8 confirm this analysis.

We believe that these results indicate that an approach combining covering arrays and optimization-based falsification, such as CA+SA, can be useful in testing autonomous driving systems, especially if specific combinations of some discrete parameters that cannot be directly discovered by intuition may generate challenging scenarios for perception systems.

6 Conclusions

We demonstrated a simulation-based adversarial test generation framework for autonomous vehicles. The framework works in a closed-loop fashion where the system evolves in time with the feedback cycles of the autonomous vehicle's controller, including its perception system, which is assumed to be based on a deep neural network (DNN). We demonstrated a new effective way of finding a critical vehicle behavior by using 1) covering arrays to test combinations of discrete parameters and 2) simulated annealing to find corner-cases.

Future work will include using identified counterexamples to retrain and improve the DNN-based perception system. Also, the framework can be further improved by incorporating not only camera data but also other types of sensors. The scene rendering may also be made more realistic by using other scene rendering tools such as those based on state-of-the-art game engines.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin et al. (2016): *Tensorflow: Large-scale machine learning on heterogeneous distributed systems*. arXiv preprint:1603.04467.

[2] Houssam Abbas, Georgios E. Fainekos, Sriram Sankaranarayanan, Franjo Ivancic & Aarti Gupta (2013): *Probabilistic Temporal Logic Falsification of Cyber-Physical Systems*. ACM Transactions on Embedded Computing Systems 12(s2).

[3] Ezio Bartocci, Jyotirmoy Deshmukh, Alexandre Donzé, Georgios Fainekos, Oded Maler, Dejan Nickovic & Sriram Sankaranarayanan (2018): *Specification-based Monitoring of Cyber-Physical Systems: A Survey on Theory, Tools and Applications*. In: Lectures on Runtime Verification - Introductory and Advanced Topics, LNCS 10457, Springer, pp. 128–168.

[4] Pin-Yu Chen, Huan Zhang, Yash Sharma, Jinfeng Yi & Cho-Jui Hsieh (2017): *ZOO: Zeroth Order Optimization based Black-box Attacks to Deep Neural Networks without Training Substitute Models*. arXiv:1708.03999v2.

[5] Alexander Donze & Oded Maler (2010): *Robust satisfaction of Temporal Logic over Real-valued signals*. In: Formal Modelling and Analysis of Timed Systems, LNCS 6246, Springer.

[6] T. Dreossi, A. Donze & S. A. Seshia (2017): *Compositional Falsication of Cyber-Physical Systems with Machine Learning Components*. In: NASA Formal Methods (NFM), LNCS 10227, Springer, pp. 357–372.

[7] T. Dreossi, S. Ghosh, A. Sangiovanni-Vincentelli & S. A. Seshia (2017): *Systematic Testing of Convolutional Neural Networks for Autonomous Driving*. In: Reliable Machine Learning in the Wild (RMLW).

[8] Tommaso Dreossi, Somesh Jha & Sanjit A. Seshia (2018): *Semantic Adversarial Deep Learning*. arXiv:1804.07045v2.

[9] Souradeep Dutta, Susmit Jha, Sriram Sanakaranarayanan & Ashish Tiwari (2017): *Output Range Analysis for Deep Neural Networks*. arXiv:1709.09130.

[10] Georgios Fainekos, Sriram Sankaranarayanan, Koichi Ueda & Hakan Yazarel (2012): *Verification of Automotive Control Applications using S-TaLiRo*. In: Proceedings of the American Control Conference.

[11] Georgios E. Fainekos & George J. Pappas (2006): *Robustness of Temporal Logic Specifications*. In: *Formal Approaches to Testing and Runtime Verification*, *LNCS* 4262, Springer, pp. 178–192.

[12] Andreas Geiger, Philip Lenz & Raquel Urtasun (2012): *Are we ready for autonomous driving? the kitti vision benchmark suite*. In: *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, IEEE.

[13] Ian Goodfellow (2017): *NIPS 2016 Tutorial: Generative Adversarial Networks*. arXiv:1701.00160v3.

[14] Ian J Goodfellow, Jonathon Shlens & Christian Szegedy (2015): *Explaining and harnessing adversarial examples*. In: *3rd International Conference on Learning Representations (ICLR)*.

[15] Alan Hartman (2005): *Software and hardware testing using combinatorial covering suites*. *Graph theory, combinatorics and algorithms* 34, pp. 237–266.

[16] B. Hoxha, H. Bach, H. Abbas, A. Dokhanchi, Y. Kobayashi & G. Fainekos (2014): *Towards Formal Specification Visualization for Testing and Monitoring of Cyber-Physical Systems*. In: *International Workshop on Design and Implementation of Formal Tools and Systems*.

[17] Xiaowei Huang, Marta Kwiatkowska, Sen Wang & Min Wu (2017): *Safety Verification of Deep Neural Networks*. In: *29th International Conference on Computer Aided Verification (CAV)*, *LNCS* 10426, Springer, pp. 3–29.

[18] Zdenek Kalal, Krystian Mikolajczyk & Jiri Matas (2010): *Forward-backward error: Automatic detection of tracking failures*. In: *Pattern recognition (ICPR), 20th international conference on*, IEEE.

[19] James Kapinski, Jyotirmoy V Deshmukh, Xiaoqing Jin, Hisahiro Ito & Ken Butts (2016): *Simulation-Based Approaches for Verification of Embedded Control Systems: An Overview of Traditional and Advanced Modeling, Testing, and Verification Techniques*. *IEEE Control Systems Magazine* 36(6), pp. 45–64.

[20] Guy Katz, Clark Barrett, David L. Dill, Kyle Julian & Mykel J. Kochenderfer (2017): *Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks*. In: *International Conference on Computer Aided Verification (CAV)*, Springer.

[21] BaekGyu Kim, Akshay Jarandikar, Jonathan Shum, Shinichi Shiraishi & Masahiro Yamaura (2016): *The SMT-based automatic road network generation in vehicle simulation environment*. In: *International Conference on Embedded Software (EMSOFT)*, ACM, pp. 18:1–18:10.

[22] BaekGyu Kim, Yusuke Kashiba, Siyuan Dai & Shinichi Shiraishi (2017): *Testing Autonomous Vehicle Software in the Virtual Prototyping Environment*. *Embedded Systems Letters* 9(1), pp. 5–8.

[23] Alex Krizhevsky, Ilya Sutskever & Geoffrey E Hinton (2012): *Imagenet classification with deep convolutional neural networks*. In: *Advances in neural information processing systems*, pp. 1097–1105.

[24] D Richard Kuhn, Raghu N Kacker & Yu Lei (2013): *Introduction to combinatorial testing*. CRC press.

[25] Olivier Michel (2004): *Cyberbotics Ltd. Webots: professional mobile robot simulation*. *International Journal of Advanced Robotic Systems* 1(1), p. 5.

[26] M. O'Kelly, H. Abbas & R. Mangharam (2017): *Computer-aided design for safe autonomous vehicles*. In: *2017 Resilience Week (RWS)*.

[27] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z Berkay Celik & Ananthram Swami (2017): *Practical black-box attacks against machine learning*. In: *ACM Asia Conference on Computer and Communications Security*, ACM.

[28] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z Berkay Celik & Ananthram Swami (2016): *The limitations of deep learning in adversarial settings*. In: *IEEE European Symposium on Security and Privacy (EuroS&P)*, IEEE.

[29] Kexin Pei, Yinzhi Cao, Junfeng Yang & Suman Jana (2017): *DeepXplore: Automated Whitebox Testing of Deep Learning Systems*. In: *Symposium on Operating Systems Principles (SOSP)*.

[30] Dean A Pomerleau (1989): *Alvinn: An autonomous land vehicle in a neural network*. In: *Advances in neural information processing systems*, pp. 305–313.

[31] Luca Pulina & Armando Tacchella (2010): *An Abstraction-Refinement Approach to Verification of Artificial Neural Networks*, pp. 243–257. *LNCS* 6174, Springer Berlin Heidelberg, Berlin, Heidelberg.

[32] Shai Shalev-Shwartz, Shaked Shammah & Amnon Shashua (2017): *On a Formal Model of Safe and Scalable Self-driving Cars*. arXiv:1708.06374.

[33] Mark Strickland, Georgios Fainekos & Heni Ben Amor (2017): *Deep Predictive Models for Collision Risk Assessment in Autonomous Driving*. arXiv preprint arXiv:1711.10453.

[34] Yuchi Tian, Kexin Pei, Suman Jana & Baishakhi Ray (2017): *DeepTest: Automated Testing of Deep-Neural-Network-driven Autonomous Cars*. arXiv:1708.08559v1.

[35] Cumhur Erkan Tuncali, Theodore P. Pavlic & Georgios Fainekos (2016): *Utilizing S-TaLiRo as an Automatic Test Generation Framework for Autonomous Vehicles*. In: IEEE Intelligent Transportation Systems Conference.

[36] Bichen Wu, Forrest Iandola, Peter H Jin & Kurt Keutzer (2016): *SqueezeDet: Unified, small, low power fully convolutional neural networks for real-time object detection for autonomous driving*. arXiv preprint arXiv:1612.01051.