A Survey of Algorithms for Black-Box Safety Validation

Anthony Corso ACORSO@STANFORD.EDU
Aeronautics and Astronautics, Stanford University, Stanford, CA 94305, USA

Robert J. Moss MOSSR@CS.STANFORD.EDU
Computer Science, Stanford University, Stanford, CA 94305, USA

Mark Koren MKOREN@STANFORD.EDU
Aeronautics and Astronautics, Stanford University, Stanford, CA 94305, USA

Ritchie Lee RITCHIE.LEE@NASA.GOV
NASA Ames Research Center, Moffett Field, CA 94035, USA

Mykel J. Kochenderfer MYKEL@STANFORD.EDU
Aeronautics and Astronautics, Stanford University, Stanford, CA 94305, USA

Abstract

Autonomous and semi-autonomous systems for safety-critical applications require rigorous testing before deployment. Due to the complexity of these systems, formal verification may be impossible and real-world testing may be dangerous during development. Therefore, simulation-based techniques have been developed that treat the system under test as a black box during testing. Safety validation tasks include finding disturbances to the system that cause it to fail (falsification), finding the most-likely failure, and estimating the probability that the system fails. Motivated by the prevalence of safety-critical artificial intelligence, this work provides a survey of state-of-the-art safety validation techniques with a focus on applied algorithms and their modifications for the safety validation problem. We present and discuss algorithms in the domains of optimization, path planning, reinforcement learning, and importance sampling. Problem decomposition techniques are presented to help scale algorithms to large state spaces, and a brief overview of safety-critical applications is given, including autonomous vehicles and aircraft collision avoidance systems. Finally, we present a survey of existing academic and commercially available safety validation tools.

Keywords: Safety Validation, Falsification, Optimization, Path Planning, Reinforcement Learning, Cyber-Physical Systems

1. Introduction

Increasing levels of autonomy promise to revolutionize industries such as automotive transportation [1] and aviation [2], [3] by improving convenience and efficiency while lowering cost. Innovations have been driven by recent progress in artificial intelligence, particularly in machine learning [4], [5] and planning [6], [7]. Machine learning has recently achieved human-competitive performance in board games [8], [9], video games [10], [11], and visual perception [12], [13]. However, applying machine learning technologies to safety-critical domains has been challenging. Safety-critical systems differ from other systems in that their failures can have serious consequences, such as loss of life and property. As a result, these systems must undergo extensive validation and testing prior to certification and deployment.

Safety validation is the process of ensuring the correct and safe operation of a system operating in an environment. The desired system behavior is defined through a specifica-

tion, and a failure is any violation of that specification. Typically, simulation is used to find failures of a system caused by disturbances in the the environment, and a model of the disturbances can then be used to determine the probability of failure. A system is deemed safe if no failure has been found after adequate exploration of the space of possible disturbances, or if the probability of failure is found to be below an acceptable threshold. The procedure of proving that a system is safe to all disturbances is known as verification [14]–[18] and is outside the scope of this survey.

In this paper, we focus on cyber-physical systems, which involve software and physical systems interacting over time. This broad definition includes systems such as robots, cars, aircraft, and planetary rovers. There are several reasons why validating cyber-physical systems is challenging. First, many of these systems contain complex components, including those produced by machine learning. The safety properties of these systems may not be well-understood and subtle and emergent failures can go undetected [19]. Second, many systems, such as autonomous cars and aircraft, interact with complex and stochastic environments that are difficult to model. Third, safety properties are generally defined over both the system under test and its environment. For example, the requirement that "the test vehicle shall not collide with pedestrians" involves both the system under test (test vehicle) and actors in its environment (pedestrians). As a result, safety validation must be performed over the combined system. Fourth, sequential interactions between the system and the environment means that failure scenarios are trajectories over time. Naively searching over trajectories leads to an exponential analysis space in time. Finally, safety validation is often applied to mature safety-critical systems in the late stages of development, where failures can be extremely rare.

Traditional methods for ensuring safety, through safety processes, engineering analysis, and conventional testing, though necessary, do not scale to the complexity of next-generation systems. Advanced validation techniques are needed to build confidence in these systems. Many validation approaches have been proposed in the literature. They can be broadly categorized by the information they use for analysis. White-box methods use knowledge of the internals of the system. For example, formal verification in the form of model checking [14], [15] and automated theorem proving [16]–[18], represents the system using mathematical models. Because the model is known, formal verification methods can find failure examples when they exist or prove the absence of failures when they do not. However, because formal verification considers all execution possibilities, it often has difficulty scaling to large problems.

In contrast to white-box methods, black-box techniques do not assume that the internals of the system are known. They consider a general mapping from input to output that can be sampled. Owing to the generality of the system description, black-box methods can be applied to analyze a much broader class of systems. However, the reliance on sampling means that black-box methods can generally never (in a finite number of samples) provide complete coverage and prove the absence of failures. Instead, black-box methods generally aim to quickly and efficiently find failure examples. While one can never conclude that a failure cannot exist, confidence in the system can increase with additional sampling. Due to its flexibility and scalability, black-box validation is often the only feasible option for large complex systems, and is the focus of this survey.

We consider three safety validation tasks for a system with specifications. First, falsification aims to find an example disturbance in the environment that causes the system to violate the specification. This formulation is useful for discovering previously unknown failure modes and finding regions where the system can operate safely. The second safety validation task is to find the most-likely failure according to a probabilistic model of the disturbances. The model can be created through expert knowledge or data to reflect the probabilities in the real environment. The third safety validation task is to estimate the probability that a failure will occur. Failure probability estimation is important for the acceptance and certification of autonomous systems.

There are many algorithms that have been used for these safety validation tasks. This survey categorizes and presents many of them. Falsification and most-likely failure analysis are related tasks, in that they involve finding failures of an autonomous system. Categories of algorithms that are suited for these tasks include optimization, path planning, and reinforcement learning. In optimization, we seek to find a trajectory of disturbances that cause the system to fail. In path planning, we use the environment's state to aid in the exploration of possible failure modes. In reinforcement learning, we frame the problem as a Markov decision process and search for a policy that maps environment states to disturbances that cause the system to fail. When the goal is to estimate the probability of failure and failures are rare, we can use techniques such as importance sampling to generate scenarios and translate their results into a probability estimate. For all of the safety validation tasks, a major challenge is scalability. We present several problem-decomposition techniques that can allow for better scalability of the presented algorithms.

This paper is organized as follows. Section 2 introduces notation and the problem formulations of the three safety validation tasks. Section 3 demonstrates how to frame falsification and most-likely failure analysis as optimization problems and presents several algorithms for solving them efficiently. Section 4 describes how to use the environment state to find failures of the system through several path-planning algorithms. Section 5 shows how to frame falsification and most-likely failure analysis as a Markov decision processes and presents several reinforcement learning algorithms to solve them. Section 6 introduces importance sampling and describes some algorithms that are used to estimate the probability of failure. Section 7 presents several ways to address problem scalability through decomposition. Section 8 surveys the various applications and discusses common strategies and adaptations of approaches for each domain. Finally, section 9 surveys existing tools in the literature and compares their basic features.

2. Preliminaries

This section first presents notation used to define the safety validation tasks and describe the safety validation algorithms. We then define three different safety validation tasks: falsification, most-likely failure analysis, and failure probability estimation.

2.1 Notation

Suppose we have a system under test (referred to as the *system*) that takes actions $a \in A$ in an *environment* with state $s \in S$. Actions are selected based on observations $o \in O$ of the environment. The system interacts with the environment over discrete time $t \in$

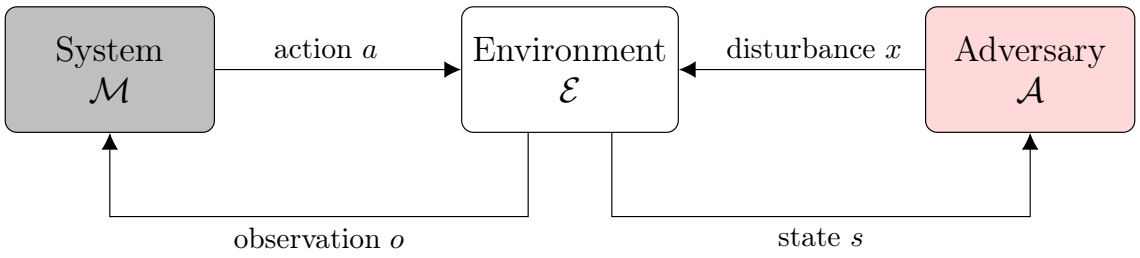

Figure 1: Model of the safety validation problem.

$\{1, \ldots, t_{\max}\}$. We denote a state trajectory up to time t as $\mathbf{s}_{1:t} = [s_1, \ldots, s_t]$, an observation trajectory $\mathbf{o}_{1:t} = [o_1, \ldots, o_t]$, and an action trajectory $\mathbf{a}_{1:t} = [a_1, \ldots, a_t]$. The system may be modeled by a function $\mathcal{M}$ that maps an observation trajectory to an action

$$a_t = \mathcal{M}(\mathbf{o}_{1:t}). \tag{1}$$

An *adversary* $\mathcal{A}$ produces disturbances $x \in X \subseteq \mathbb{R}^n$ in the environment with the goal of causing the system to fail. A disturbance trajectory is denoted $\mathbf{x}_{1:t}$ or just $\mathbf{x}$. The adversary can use full or partial knowledge of the environment state to determine the next disturbance such that

$$x_t = \mathcal{A}(\mathbf{s}_{1:t}). \tag{2}$$

The environment state evolves over time and is influenced by the actions of the system and the disturbances from the adversary. The environment is modeled by $\mathcal{E}$, where

$$s_{t+1}, o_{t+1} = \mathcal{E}(\mathbf{a}_{1:t}, \mathbf{x}_{1:t}). \tag{3}$$

The interaction between the system, the environment, and the adversary is depicted in fig. 1.

Some safety validation tasks require that the environment has a model of the disturbances. Let $p(\mathbf{x})$ be the probability density of the disturbance trajectory $\mathbf{x}$. If the disturbances are independent across time, then it is sufficient to define the density over a single disturbance $p(x)$, or if the disturbances depend only on the state it is sufficient to define $p(x \mid s)$. The disturbance model can be constructed through expert knowledge or learned from data.

In this work, we assume that the environment and the system are fixed, making disturbances the only way to affect the system. Therefore, from the point of view of the adversary, the system and environment can be combined into a single function f that maps disturbance trajectories into state trajectories

$$\mathbf{s} = f(\mathbf{x}). \tag{4}$$

Some algorithms require the ability to simulate a disturbance x_t for a single timestep from a state s_t. We denote the simulation step

$$s_{t+1} = f(s_t, x_t). \tag{5}$$

The desired operation of the system is described by one or more specifications ψ which are written in a formal specification language such as temporal logic [20] or designed ad hoc.

We overload the notation ψ to be the set of state trajectories that satisfy the specification and write $\mathbf{s} \in \psi$ when the state trajectory satisfies the specification and $\mathbf{s} \notin \psi$ when $\mathbf{s}$ violates the specification.

2.2 Problem Formulation

The focus of this work is the safety validation of black-box systems. *Safety validation* is the process of discovering which disturbances $\mathbf{x}$ cause the system to behave improperly and how frequently they occur. Answering these questions gives insight into the safety of the system. A system is a *black box* if the function $\mathcal{M}$ is not known or too complex to explicitly reason about. A *white-box* system can be described analytically or specified in a formal modeling language. Some white-box systems may be treated as a black box if knowledge of their design does not help the validation process. For example, while small neural networks can have properties formally verified [21], large neural networks with millions or billions of parameters are generally too large for white-box verification techniques. *Gray-box* systems usually provide some knowledge of their underlying structure without describing it entirely. Some gray-box safety validation techniques are included in this survey due to their broad applicability.

Three safety validation tasks are considered in this work and are defined below.

Falsification. Falsification is the process of finding a disturbance trajectory that causes the outputs to violate a specification ψ. Such a trajectory is known as a counterexample, failure trajectory, or falsifying trajectory. We want to find

$$\mathbf{x} \text{ s.t. } f(\mathbf{x}) \notin \psi. \tag{6}$$

Most-Likely Failure Analysis. Most-likely failure analysis tries to find the failure trajectory with maximum likelihood. We want to solve

$$\arg\max_{\mathbf{x}} p(\mathbf{x}) \text{ s.t. } f(\mathbf{x}) \notin \psi. \tag{7}$$

Failure Probability Estimation. Failure probability estimation tries to compute the probability that a specification will be violated. We want to compute the expectation

$$P_{\text{fail}} = \mathbb{E}\left[\mathbb{1}\{f(\mathbf{x}) \notin \psi\}\right], \tag{8}$$

3. Optimization

A naive approach to falsification is to search randomly over disturbance trajectories until a counterexample is discovered. If counterexamples are rare, this process can be inefficient. To help guide the search for counterexamples, we can define a cost function $c_{\text{state}}(\mathbf{s})$ that measures the level of safety of the system over the state trajectory $\mathbf{s}$. A well-designed cost function will help bias the search over disturbance trajectories toward those that are less safe. Additionally, if the goal is to find the most-likely failure, then the cost function can incorporate the likelihood of the disturbance trajectory.

Once a cost function is defined, falsification and most-likely failure analysis become optimization problems where we seek to minimize the cost with respect to the disturbance

trajectory $\mathbf{x}$. We define the optimization problem as

$$\mathbf{x}^* = \arg\min_{\mathbf{x}} c(\mathbf{x}), \tag{9}$$

where $c(\mathbf{x}) = c_{\text{state}}(f(\mathbf{x}))$. The cost function is designed such that $c(\mathbf{x}) \geq \epsilon \iff f(\mathbf{x}) \in \psi$, where ϵ is a safety threshold. Therefore, if a disturbance $\mathbf{x}$ causes $c(\mathbf{x}) < \epsilon$, then $\mathbf{x}$ is a counterexample.

The design of the cost function c is specific to the application. Consider the safety validation of an autonomous vehicle that has a specification to not collide with other vehicles. The cost function could be the distance at the point of closest approach (i.e. miss distance) between the system and any other vehicle. The smaller the value, the closer the system is to a collision. If the distance is less than a safety threshold, then a collision has occurred [22].

When the specification is represented by a temporal logic expression, a common choice for c is the temporal logic robustness $\rho(\mathbf{s}, \psi)$. The robustness is a measure of the degree to which the trajectory $\mathbf{s}$ satisfies the specification ψ. Large values of robustness mean that at no point does the trajectory come close to violating the specification, while low but positive values of robustness mean that the trajectory is close to violating the specification. A robustness value less than 0 means that the specification has been violated and gives an indication of by how much. The robustness for space-time signals can be computed from a recursive definition [23]–[25]. Upper and lower bounds on robustness can be computed for incomplete signals [26], which is useful when constructing a trajectory sequentially [26], [27]. The derivative of the robustness with respect to the state trajectory can be approximated, which may help derive gradient-driven optimization algorithms [28]. Causal information in the form of a Bayesian network can be incorporated into the robustness for improved falsification [29].

One drawback to using robustness as a cost is that it depends on the scale of the state values. If the units of one state variable cause it to be much larger than another (i.e. engine RPMs versus angular rotation in radians), then the contribution to the robustness will favor the larger variable. The state variables may be normalized, but this requires the range of the state variables which may not be known. To mitigate this problem, Zhang, Hasuo, and Arcaini [30] proposed measuring the robustness of each state variable independently and using a multi-armed bandit algorithm to decide which robustness value to optimize on each iteration.

When the safety validation task is to find the most-likely failure, we can modify the cost function to include the likelihood of the disturbance trajectory $p(\mathbf{x})$. One option is to define a multiobjective cost function $c'(\mathbf{x})$ defined as

$$c'(\mathbf{x}) = c(\mathbf{x}) - \lambda p(\mathbf{x}), \tag{10}$$

where $\lambda > 0$ is user-defined. Alternatively, we can consider a piecewise objective that only considers the likelihood when the disturbance trajectory leads to a failure:

$$c'(\mathbf{x}) = \begin{cases} c(\mathbf{x}) & \text{if } c(\mathbf{x}) \geq \epsilon \\ -p(\mathbf{x}) & \text{if } c(\mathbf{x}) < \epsilon. \end{cases} \tag{11}$$

Formulating safety validation as an optimization problem allows for the use of many existing optimization algorithms (see Kochenderfer and Wheeler [31] for an overview). Due

to the complexity of many autonomous systems and environments, the optimization problem is generally non-convex and can have many local minima. Therefore, algorithms that can escape local minima and have adequate exploration over the space of disturbance trajectories are preferred. Two algorithms that have been used in falsification software [32], [33] (see section 9 for more details) are covariance matrix adaptation evolution strategy (CMA-ES) [34] and globalized Nelder-Mead (GNM) [35], both of which are effective for global optimization. Another way to escape local minima is to combine global and local search [36]–[38], where one optimization routine is used to explore the space of disturbances trajectories to find regions of interest, and another algorithm does local optimization to find the best disturbance trajectory in a region.

This section describes algorithms that have had success for falsification. The algorithms included are those that have been designed explicitly for, or have been modified to work with, black-box falsification. The basic version of each algorithm is described, and where applicable, the variations of the algorithm are discussed.

3.1 Simulated Annealing

An approach to stochastic global optimization known as simulated annealing (SA) uses a random walk around the disturbance space to minimize a cost function c. A temperature parameter β is used to control the amount of stochasticity in the method over time and a transition function $T(\mathbf{x}' \mid \mathbf{x})$ describes the probability distribution over the next disturbance trajectory $\mathbf{x}'$. Due to the stochastic nature of simulated annealing, it is often a suitable algorithm for solving a global optimization problem with many local minima and can therefore be effective for falsification [39]–[41].

The basic approach is presented in algorithm 1. It begins by selecting a random starting disturbance trajectory $\mathbf{x}$ (line 2). At each iteration, a new disturbance trajectory $\mathbf{x}'$ is sampled from $T(\mathbf{x}' \mid \mathbf{x})$ (line 4). If the new trajectory is a counterexample, then it is returned (line 6), otherwise it is subjected to an acceptance test with probability $\exp(-\beta(c(\mathbf{x}')-c(\mathbf{x})))$ (line 7). If $\mathbf{x}'$ is accepted, then it replaces $\mathbf{x}$, otherwise $\mathbf{x}$ is left unchanged. The procedure repeats until the computational budget is exhausted.

Algorithm 1 Simulated Annealing

```
1: function SimulatedAnnealing(β, T, c, ϵ)
2:     Sample initial x
3:     loop
4:         x′ ∼ T(x′ | x)
5:         if c(x′) < ϵ
6:             return x′
7:         if UniformRand() < exp(−β(c(x′) − c(x)))
8:             x ← x′
```

The main design choices when using simulated annealing are the annealing temperature β the choice of the transition function T. The annealing temperature can be adjusted based on the number of accepted and rejected disturbance trajectories. A typical choice is to adjust β so that the next point is accepted approximately half of the time [39].

A common choice for transition function is to use a Gaussian distribution around the current point $\mathbf{x}$ with a standard deviation that is adjusted based on the ratio of accepted points [31]. This approach may not work well when the disturbance space has constraints that must be satisfied, such as lower and upper bounds on the possible disturbances [39]. Abbas *et al.* [39] proposes the use of a *hit and run* approach to transitioning that respects constraints [39]. It follows 3 steps:

1. Sample a random direction $\mathbf{d}$ in the disturbance trajectory space.
2. Perform a line search in the direction of $\mathbf{d}$ to determine the range of α such that $\mathbf{x}+\alpha\mathbf{d}$ does not violate any constraints.
3. Sample α from this range according to a chosen distribution. The standard deviation of this distribution can be adjusted using the acceptance ratio to improve convergence.

Aerts *et al.* [40] improved the hit and run scheme by suggesting that α be chosen for each disturbance dimension separately so that highly constrained dimensions do not restrict the step size of less constrained dimensions [40].

In the basic SA algorithm, the size of $\mathbf{x}$ remains fixed throughout the optimization, meaning the temporal discretization of the disturbance trajectory is never adjusted. Aerts *et al.* [40] note that the frequency content of the disturbance trajectory is salient for some falsification problems, and therefore the temporal discretization of the disturbance trajectory should itself be optimized. An approach called input-signal-space optimization uses a two-layered approach, where an outer loop uses SA to select the length of the disturbance trajectory, and an inner loop finds the lowest cost trajectory for that length. This approach is able to adapt the fidelity of the time domain, getting improved results for some falsification problems [40].

3.2 Genetic Algorithms

Genetic algorithms (GA) approach global optimization by mimicking the biological process of evolution and have been previously applied to falsification [42], [43]. The algorithm starts by sampling a population of individuals (disturbance trajectories) $\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}\}$ and then iterates over the following steps:

1. *Selection:* Pairs of individuals are chosen to produce the next generation. The pairs are typically chosen to be among the lowest-cost individuals.
2. *Crossover:* The selected pairs produce new trajectories by combining their features using a crossover function.
3. *Mutation:* Some of the new individuals are randomly modified to incorporate new traits into the population.

Zhao, Krogh, and Hubbard [42] applied a genetic algorithm to generate test cases for embedded control systems. Each individual is represented by a real-valued vector for all continuous variables and a binary encoding for all discrete variables. Selection occurs with a probability inversely related to the cost function. The crossover between $\mathbf{x}^{(1)}$ and $\mathbf{x}^{(2)}$ is

done using one of three mechanisms. The first is simple crossover where a random time t is chosen and the crossover disturbance trajectory is

$$\mathbf{x}_{\text{new}} = \left[\mathbf{x}^{(1)}_{1:t} \ \mathbf{x}^{(2)}_{(t+1):t_{\max}}\right], \tag{12}$$

where the square brackets indicate concatenation. The second is an arithmetic crossover where

$$\mathbf{x}_{\text{new}} = (1-\lambda)\mathbf{x}^{(1)} + \lambda\mathbf{x}^{(2)} \tag{13}$$

for some value of $\lambda \in [0, 1]$. Zhao, Krogh, and Hubbard [42] perform crossover with

$$\mathbf{x}_{\text{new}} = \min(\max(\mathbf{x}^{(1)} \pm \lambda\mathbf{x}^{(2)}, \mathbf{x}_{\text{low}}), \mathbf{x}_{\text{high}}), \tag{14}$$

where $\mathbf{x}_{\text{low}}$ and $\mathbf{x}_{\text{high}}$ are the lower and upper bounds of the disturbance space. Mutation is performed by selecting a random position in $\mathbf{x}_{\text{new}}$ and filling in a new value from $x \in X$.

3.3 Bayesian Optimization

In Bayesian optimization [44], we maintain a probabilistic surrogate model of the cost function over the space of disturbance trajectories. We use the model to select disturbance trajectories that are likely to lower the cost function. Maintaining a surrogate model can be beneficial when evaluations of $c(\mathbf{x})$ are costly. Since the surrogate model is probabilistic, it naturally handles stochastic objective functions and uncertainty. For these reasons, Bayesian optimization is a natural choice for falsification [29], [45]–[47].

A common choice of surrogate model is the Gaussian process (GP), which is a distribution over functions. GPs are popular because they can model complex functions and infer the costs of unexplored regions of the disturbance space based on previous sample points [31]. Gaussian processes are parameterized by a mean function, a kernel function, and a model of the noise in the cost function. For notational convenience, all of these parameters will be grouped into θ, which can be specified or learned from data using maximum likelihood estimation [31].

The basic approach is outlined in algorithm 2. It takes as input the initial GP parameters θ. A set of previous disturbance trajectories and their costs are each initialized to the empty set (line 2). On each iteration, the Gaussian process model is updated based on the parameters and all of the trajectories and costs seen so far (line 4). The GP is used to get estimates of the mean and uncertainty of the cost function over the disturbance space (line 5) and those estimates are used to determine the next sample point (line 6). If a counterexample is found, return it, otherwise add the new trajectory and associated cost to dataset (line 9). Lastly, the parameters can optionally be fit using the data seen so far (line 10).

The OptimalDisturbance function searches with respect to a particular metric. Examples of metrics include [31]:

- *Prediction-based exploration:* The value of the predicted mean $\hat{\mu}$ is minimized.
- *Error-based exploration:* The value of the standard deviation $\hat{\sigma}$ is maximized.
- *Lower confidence bound exploration:* The value of the lower confidence bound ($\hat{\mu} - \alpha\hat{\sigma}$ for some $\alpha > 0$) is minimized.

Algorithm 2 Bayesian Optimization

```
function BayesianOptimization(θ, c, ϵ)
    data ← ∅
    loop
        GP ← ConstructGP(θ, data)
        μ̂(x), σ̂(x) ← Model(GP)
        x_new ← OptimalDisturbance(μ̂, σ̂)
        if c(x_new) < ϵ
            return x_new
        data ← data ∪ {(x_new, c(x_new))}
        θ ← Fit(GP, data)
```

- *Probability of improvement:* If $c_{\min}$ is the lowest cost so far then $P(c < c_{\min})$ is maximized.
- *Expected improvement exploration:* The expected improvement $\mathbb{E}_{\hat{\mu},\hat{\sigma}}[I(c)]$ is maximized, where the improvement is defined as:

$$I(c) = \begin{cases} c_{\min} - c & \text{if } c < c_{\min} \\ 0 & \text{otherwise.} \end{cases} \tag{15}$$

The search for the optimal disturbance can be conducted using an optimization routine such as simulated annealing [29] or a sampling plan with good coverage of the space of disturbance trajectories [47].

One drawback to using Gaussian processes is their inability to scale to large dimensions and many sample points. To improve scalability for falsification, Mathesen *et al.* [36] introduced an algorithm called stochastic optimization with adaptive restarts (SOAR) that uses a two-layered Bayesian optimization approach combined with simulated annealing. A Gaussian process model is maintained over the global search space, and a stochastic method is used to select regions for further exploration. Local Gaussian process models are used for refined searching of local regions. When a local region is no longer seeing improvement, a new region is selected.

3.4 Extended Ant-Colony Optimization

Ant colony optimization is a probabilistic technique initially used for finding optimal paths through graphs [48]. The algorithm is inspired by the exploration strategy used by some ant species, where ants leave traces of pheromone when searching for food. The presence of pheromone makes it more likely for an ant to follow a path, but the pheromone evaporates over time.

Ant colony optimization was extended to continuous spaces by Fainekos and Giannakoglou [49] and later applied to falsification [50]. Extended ant-colony optimization is presented in algorithm 3. Finding the optimal disturbance trajectory $\mathbf{x}$ is formulated as a graph-traversal problem. Edges in the graph exist between adjacent temporal points (x_t, x_{t+1}). At each time t, the space of disturbances is discretized into N equally spaced

cells, each with a center point μ_{it} for integer $i \in \{1, \ldots, N\}$. Ants traverse the graph (line 5) by selecting their next cell i at time t based on the amount of pheromone present τ_{it} (initialized to a constant τ_0 in line 2). The probability of selecting a cell is

$$P(i \mid t) = \frac{\tau_{it}}{\sum_j \tau_{jt}}. \tag{16}$$

Once a cell is selected, a random point x_{it} is selected uniformly at random from within it. The process continues for each ant until it reaches the end of the trajectory at $t_{\max}$. At the end of a disturbance trajectory, if it is a counterexample then it can be returned (line 6). After each of M ants has completed a full disturbance trajectory, the pheromones are updated (line 8) through evaporation with rate ρ, and deposition as

$$\tau_{it} \leftarrow \underbrace{(1-\rho)\tau_{it}}_{\text{evaporation}} + \underbrace{\sum_{k=1}^{M} \frac{\exp(-\|x_t^{(k)} - \mu_{it}\|^2/\alpha)}{\beta + c(\mathbf{x}^{(k)})}}_{\text{deposition}} \tag{17}$$

where $x_t^{(k)}$ is the disturbance of ant k at time t, $c(\mathbf{x}^{(k)})$ is the cost of the full disturbance trajectory of ant k, and α and β are user-defined constants.

Algorithm 3 Extended Ant Colony Optimization

```
1: function ExtendedAntColony(τ_0, μ_it, ρ, α, β, c, ϵ)
2:     τ_it ← τ_0 for all i and t
3:     loop
4:         for k ∈ 1 : M
5:             x^(k) ← RunAnt(τ_it)
6:             if c(x^(k)) < ϵ
7:                 return x^(k)
8:         τ_it ← UpdatePheromone(τ_it, μ_it, x^(k), ρ, α, β)
```

3.5 Interpretable Falsification with Genetic Programming

Corso and Kochenderfer [51] approach falsification and most-likely failure analysis with the idea that counterexamples can often be described by simple temporal logic specifications on the disturbance trajectory. A specifications φ is sampled from a grammar with a bias toward simpler statements. It is optimized with the goal of finding a specification such that if a disturbance trajectory satisfies it, then that trajectory causes the system to fail, i.e.

$$\mathbf{x} \in \varphi \implies c(\mathbf{x}) < \epsilon. \tag{18}$$

The approach is outlined in algorithm 4. It starts by sampling a set of M disturbance specifications from a temporal logic grammar $\mathcal{G}_{\text{TL}}$ (line 2). Specifications are represented by a tree, and the sampling procedure involves iteratively expanding each non-terminal node in the tree with a production rule from the grammar, chosen uniformly at random. This sampling scheme induces a distribution over specifications where more complex specifications are significantly less likely to be sampled.

For each specification we estimate the average cost of disturbance trajectories that satisfy it by taking the mean of N samples (lines 5 and 6). This sampling is done by converting the specification into a set of inequality constraints on the disturbance trajectory [51], and then sampling from $p(\mathbf{x})$ subject to those constraints. Inequality constraints can easily be applied when the components of $\mathbf{x}$ are independent or when the joint distribution is multivariate Gaussian [51]. For more complex distributions, rejection sampling could be used, but may not scale to high dimensions.

If a specification has a sufficiently small average cost, then return it (line 8), otherwise, update the specifications using genetic programming by performing selection, crossover, and mutation (line 9 to 11). In selection, the highest performing specifications are chosen to produce the next generation. In crossover, a new specification is constructed from two selected parent specifications. The subtree from a random node in the first parent replaces the subtree at a random node of the second parent. In mutation, the specification is altered by randomly replacing a subtree with a new rule, or permuting the subtrees already present [31]. The result of algorithm 4 is a specification that describes the types of disturbances that cause the system to fail. It has been shown to be effective at finding counterexamples as well as giving engineering insight into the failure modes of the system.

Algorithm 4 Interpretable Falsification with Genetic Programming

1: **function** InterpretableFalsification($\mathcal{G}_{\text{TL}}$, c, ϵ)
2: Sample $\{\varphi_1, \ldots, \varphi_M\}$ from $\mathcal{G}_{\text{TL}}$
3: **loop**
4: **for** each specification φ_j
5: Sample $\{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ from $p(\mathbf{x})$ s.t. $\mathbf{x}_i \in \varphi_j$
6: $c_{\varphi_j} \leftarrow \frac{1}{N} \sum_{i=1}^{N} c(\mathbf{x}_i)$
7: **if** any $c_{\varphi_j} < \epsilon$
8: **return** φ_j
9: Select new specifications $\{\varphi_1, \ldots, \varphi_M\}$ based on c_{φ_j}
10: Crossover specifications
11: Mutate specifications

4. Path Planning

Falsification may be framed as a path planning problem through the state space of the environment using the disturbances as control inputs. In path planning, there is an initial state s_0 and a set of failure states S_{fail} that we seek to reach by sequentially constructing a disturbance trajectory $\mathbf{x}$. The benefits to a path planning approach are the use of state information to guide the choice of disturbance and the ability to reuse partial trajectory segments. For a discussion of general path planning algorithms see the overview by LaValle [52]

The first path planning algorithm covered is rapidly exploring random tree (RRT) which has many variants that have been used for falsification. The RRT algorithm attempts to cover the state space by building a space-filling tree, making it a natural approach for finding counterexamples. The next algorithm discussed is a multiple shooting method that frames

falsification as a graph traversal problem over a discretized state space. The discretization is iteratively refined so computation is only spent in promising regions. Lastly, we present Las Vegas tree search which constructs a tree of disturbances from a predefined set of trajectory segments.

4.1 Rapidly Exploring Random Tree

Rapidly-exploring random tree (RRT) is a path planning technique for efficiently finding failure trajectories [53]. A space-filling tree is iteratively constructed by sampling the state space and growing in the direction of unexplored regions. RRT has been applied to the falsification of black-box systems [26], [54]–[61].

Algorithm 5 Rapidly-exploring random tree

```
1: function RRT(s_0, S_fail)
2:     T ← InitializeTree(s_0)
3:     loop
4:         s_goal ← SampleState()
5:         s_near ← NearestNeighbor(T, s_goal)
6:         x_new ← GetDisturbance(s_near, s_goal)
7:         s_new ← f(s_near, x_new)
8:         AddNode(T, s_near, s_new)
9:     return CounterExamples(T, S_fail)
```

The basic approach is presented in algorithm 5. On each iteration, a random point in the state space s_{goal} is generated, which acts as the goal state for the next node to be added (line 4). The tree is searched for the node that is closest to the goal state s_{near} (line 5) based on some distance metric d. This node will act as the starting point when attempting to reach the goal. A disturbance x_{new} is generated that drives s_{near} toward s_{goal} (line 6). Since the system is a black-box, we cannot generally determine an x_{new} that causes $s_{\text{near}} = s_{\text{goal}}$ exactly. Instead we can use random sampling of x_{new} or a more advanced optimization procedure to get as close as possible. Lastly, the disturbance x_{new} is simulated, starting from s_{near}, resulting in a new state s_{new} (line 7) which is then added to the tree as a child of s_{near} (line 8). Note that if the simulator cannot be initialized to any state, then the trace can be simulated by starting at the root and simulating the disturbances through the branch containing s_{near}. The algorithm stops when the maximum number of iterations is reached, a suitable falsifying trajectory is found, or tree coverage reaches a specified threshold. Variants of RRT [26], [54]–[58], [60], [61] typically differ in their approach to state space sampling, choice of distance metric for nearest neighbor selection, or by adding additional steps that reconfigure the tree for improved performance.

Coverage Metrics and Stopping Criteria. Informally, the tree coverage $C(T)$ in the state space S is the degree to which T represents S. One of the first coverage metrics to be used with RRT falsification is based on the notion of dispersion [54]. For a distance metric between states d, the dispersion is the radius of the largest ball in S that contains no points in T, i.e. $\max_{s \in S} \min_{s_i \in T} d(s, s_i)$. Dispersion is challenging to compute in many dimensions and can be overly conservative [54]. Instead, a coverage metric based on average dispersion

can be computed on a grid with n points and a spacing of δ as

$$C_{\text{disp}}(T) = 1 - \frac{1}{\delta} \sum_{i=1}^{n} \frac{\min\left(d_j(T, \delta)\right)}{n}, \tag{19}$$

where $d_j(T)$ is the shortest distance from grid point j to any node in T according to the metric d [54]. The value $\min(d_j, \delta)$ is therefore the radius of the largest ball that can sit at grid point j and not touch any nodes in the tree or another grid point. Note that $C_{\text{disp}} \to 1$ when the tree contains nodes at each grid point. The coverage is therefore related to the fidelity of the grid, with a finer grid giving a more comprehensive value at greater computational expense.

Another coverage metric that has been used more recently is star discrepancy, which measures how evenly a set of points is distributed [26], [57], [58]. The discrepancy D of a tree over a region $B \subseteq S$ is

$$D(T, B) = \frac{|T \cap B|}{|T|} - \frac{\text{vol}\left(B\right)}{\text{vol}\left(S\right)}, \tag{20}$$

where vol is the volume of a region and $|T|$ is the number of nodes in the tree. The first term is the fraction of the tree nodes that are in B and the second term is the ratio of the volume of B to the volume of S. The star discrepancy D^* is the largest discrepancy over all possible subregions of S

$$D^*(T) = \max_B D(T, B). \tag{21}$$

Note that $D^* \to 0$ when all subregions of S have their "fair share" of the sample points based on volume, and $D^* \to 1$ when all the sample points overlap. To turn discrepancy into a coverage metric, we may define

$$C_{\text{disc}}(T, B) = 1 - D(T, B) \tag{22}$$

and

$$C_{\text{star}}(T) = 1 - D^*(T). \tag{23}$$

Since S is continuous and we must maximize over all possible subregions, C_{star} is challenging to compute. Instead, it is possible to approximate it using a finite partitioning of the state space into regions $\{B_1, \ldots, B_n\}$ such that $\cup_{i=1}^{n} B_i = S$. The discrepancy coverage can be computed for each box $C_{\text{disc}}(T \cap B_i, B_i)$ [26] or a lower and upper bound of the star discrepancy coverage can be computed based on B_i [58], [62].

For path planning problems in robotics (for which the RRT algorithm was first developed), the system is generally *controllable* meaning the reachable set of states is the entire state space and the growth of the tree can be terminated when a coverage metric exceeds a chosen threshold. When applying RRT to the problem of falsification, it is often the case that the system is not controllable by the disturbances so the reachable set is a subset of S [54]. In this case, a coverage metric may never exceed the needed threshold. To mitigate this problem, Esposito, Kim, and Kumar [54] proposed a growth metric on the tree defined as

$$g(T) = dC(T)/d|T|. \tag{24}$$

The tree growth measures how much the coverage metric increases with the increase in nodes in the tree. In addition to stopping based on adequate coverage, the tree can be terminated if the growth is below a specified threshold, suggesting that adding more nodes to the tree will not improve the coverage.

Adaptive Sampling. When the reachable set is only a subset of the state space a uniform sampling of goal states may be inefficient, leading to slow tree growth and low coverage values. There are several techniques for biasing goal samples toward the reachable set.

An approach known as guided-RRT (g-RRT) [58] biases the selection of s_{goal} to regions with low coverage. In g-RRT, the state space is segmented into n regions $\{B_1, \ldots, B_n\}$ and each region is assigned a weight $w(B_i)$. To sample s_{goal}, we start by sampling a region according to

$$P(B_{\text{goal}} = B_i) = \frac{w(B_i)}{\sum_j w(B_j)}, \tag{25}$$

and then s_{goal} is sampled uniformly from B_{goal}. In the original version of g-RRT [57], [58], the weight $w(B_i)$ is related to the increase in the lower and upper bounds of C_{star} when a new point is added to B_i. In later work [26], the weight is $w(B_i) = \sigma(D(T, B_i))$, where σ is the sigmoid function. The goal in both cases is to sample points that increase the coverage of the tree.

Kim, Esposito, and Kumar [55] maintain a distribution over s_{goal} that has a mean biased toward low values of the cost function and a standard deviation that adaptively changes to maximize sampling in the reachable set. As the algorithm progresses, they keep track of the ratio of successful expansions of the tree β. A successful expansion is one where

$$d(s_{\text{new}}, s_{\text{goal}}) > d(s_{\text{near}}, s_{\text{goal}}), \tag{26}$$

meaning the tree was able to grow in the direction of s_{goal}. The ratio β is updated after a user-specified number of iterations and is used to update the standard deviation between bounds $[\sigma_{\min}, \sigma_{\max}]$ as

$$\sigma = (1 - \beta)(\sigma_{\max} - \sigma_{\min}) + \sigma_{\min}. \tag{27}$$

Thus, when the number of successful expansions is large, the standard deviation is reduced to continue sampling in that region, but when the tree frequently cannot grow toward s_{goal} the range of values is increased to better search for the reachable set.

Neighbor Selection. The choice of distance metric for finding the nearest neighbor s_{near} is critical for the performance of RRT. In some applications, it is acceptable to minimize the Euclidean distance between s_{near} and s_{goal} [61], but this approach may be insufficient for some problems. Firstly, Euclidean distance does not take into account whether s_{goal} can be reached from s_{new} based on the dynamics of the system. Secondly, if the reachable set is only a small subset of the state space, points on the boundary of the reachable set will frequently be chosen as s_{near}, limiting the coverage of the tree. Lastly, if there is a cost associated with each trajectory, it should be considered when picking which node to expand.

To encode reachability in the selection of node neighbors, Kim, Esposito, and Kumar [55] uses an estimation of the time to go from s_{near} to s_{goal} defined by

$$t(s_{\text{near}}, s_{\text{goal}}) = \begin{cases} d(s_{\text{near}}, s_{\text{goal}})/v & \text{if } v > 0 \\ \infty & \text{otherwise} \end{cases}, \tag{28}$$

where the velocity is

$$v = \max_{x \in X} \left(-\frac{\partial d(s, s_{\text{goal}})}{\partial s} f(s_{\text{near}}, x) \Big|_{s=s_{\text{near}}} \right). \tag{29}$$

To compute the derivative $\partial d(s, s_{\text{goal}})/\partial s$ exactly requires white-box knowledge of the system, but it could be estimated with domain knowledge or from simulations [55].

Another modification to neighbor selection is known as history-based selection, which penalizes the selection of nodes that fail to expand [55]. A node fails to expand if it is selected as s_{near} and creates an output s_{new} that is already in the tree. Failure to expand typically occurs when s_{near} is on the boundary of the reachable set. The number of times a node has failed to expand is stored as n_f and the distance between nodes is modified as

$$d_h(s_{\text{near}}, s_j) = d(s_{\text{near}}, s_j) + \lambda n_f \tag{30}$$

for any choice of distance metric d and a user specified constant $\lambda > 0$. Kim, Esposito, and Kumar [55] chose λ to balance the contribution of d and n_j.

To incorporate a state-dependent cost function $c(s)$ into the RRT algorithm, Karaman and Frazzoli [63] developed the approach known as RRT*. In RRT*, the neighbor selection is done by finding a set S_{near} of nodes in an ϵ-radius of s_{goal} and then selecting the node in S_{near} that has the lowest cost, i.e.

$$s_{\text{near}} = \operatorname*{arg\,min}_{s_i \in S_{\text{near}}} c(s_i) \tag{31}$$

where

$$S_{\text{near}} = \{s \mid d(s, s_{\text{goal}}) < \epsilon\}. \tag{32}$$

For the cost function, Dreossi *et al.* [26] use the partial robustness of the system specification while Tuncali and Fainekos [60] uses a heuristic cost for autonomous driving.

Other RRT Variants. In the basic version of RRT, a single tree is maintained and grown until termination, but under some circumstances multiple trees may be used. When searching over a space of static parameters θ, a different tree can be grown for each choice of parameter in an approach called rapidly exploring forest of trees (RRFT) [54]. In RRFT, any tree that has reached a threshold coverage value or has stopped growing will be terminated, and a new tree (with a new value of θ) is added to the forest. This process continues until a counterexample is found or until the parameter space has been adequately covered with fully grown trees. Another example of maintaining multiple trees is the approach of Koschi *et al.* [61] where a fixed number of nodes K are added at each iteration, including the first (so K trees are maintained). Each new node is added to the tree that has the closest node. Trees that are not being grown can be removed for memory efficiency [61].

Tuncali and Fainekos [60] combined stochastic global optimization techniques with RRT by including a transition test (similar to algorithm 1) and a similarity test for the addition of new nodes. New nodes are only added if they pass the transition test (based on their cost function) and if they are sufficiently different from all of the other nodes in the tree. These two techniques enhance exploration and coverage of the state space.

Koschi *et al.* [61] introduced the backwards algorithm for RRT which connects nodes in the tree backward in time without breaking the black-box assumption. The algorithm starts by sampling a state s_0 in the failure set. At each iteration, a state is randomly sampled as s_{goal}, and the nearest neighbor in the tree s_{near} is determined. A disturbance is selected that drives s_{goal} toward s_{near} (notice this is opposite from the basic algorithm), and a new state s_{new} is generated by simulating $f(s_{\text{goal}}, x_{\text{new}})$. To maintain continuity in the tree, s_{new} replaces s_{near} and the entire branch connecting s_{near} to s_0 is re-simulated with the same disturbances. If the resulting state (the new s_0) is no longer in the failure set, then the branch is terminated and the algorithm repeats. This approach, although more computationally expensive, showed improved performance for finding rare failure events of an adaptive cruise control system [61]. Note, however, that unlike the basic RRT algorithm, this approach requires a simulator that can be initialized based only on the observed state of the environment.

4.2 Multiple Shooting Methods

Multiple shooting methods [64] solve non-linear initial value problems and have been used for robotic motion planning [65] and falsification [66], [67]. The idea is to sample trajectory segments using random initial states and random disturbances. A shortest path problem is solved to find a *candidate trajectory* through these segments that connects initial states and failure states, where two segments are connected if the terminal state of one segment is sufficiently close to the starting state of another. The discovered path may not be feasible due to the gaps between segments, so an optimization routine is used to find disturbances that minimize those gaps and find an actual failure trajectory. In most multiple-shooting algorithms, the procedure to connect trajectory segments involves using white-box information such as dynamical equations or gradients [66]. Zutshi *et al.* [67] developed a version of multiple shooting that is applicable to black-box falsification that we present here.

There are two ideas that make multiple shooting methods applicable for black-box falsification in large state spaces [67]:

1. The state space should only be explored around the reachable set.
2. State space discretization and refinement can be used to connect segments.

To achieve this, the state space is implicitly discretized into disjoint cells C, each of size δ. The cells are not explicitly stored, but they are constructed so it is efficient to find which cell contains a given state. One efficient representation is a Cartesian grid with a fixed cell size. The reachable set is estimated by running simulations forward in time with random disturbances and recording which cells are connected together. The cell connections form a graph that can be searched for candidate falsifying trajectories. Cells in these trajectories are refined until an actual counterexample is found. Since the algorithm relies on a graph search over a discretized grid, it may be useful to provide a set of initial states S_0 to have more than one starting cell.

The approach is presented in algorithm 6. It takes as input a set of initial states S_0 and a set of failure states S_{fail}, a discrete cell size δ, a refinement factor γ, a maximum number of segments per iteration $N_{\max}$, and the number of samples per cell K. A set of candidate trajectories T is initialized to the empty set (line 2). On each iteration a graph

Algorithm 6 Multiple Shooting Method

function MultipleShooting(S_0, S_{fail}, δ, γ, $N_{\max}$, K)
    $T \leftarrow \emptyset$
    loop
        $G \leftarrow \text{SampleSegments}(T, \delta, N_{\max}, K)$
        $T \leftarrow \text{FindCandidateTrajectories}(G, S_0, S_{\text{fail}})$
        if $T = \emptyset$
            return $\emptyset$
        if $\text{ActualTrajectories}(T) \neq \emptyset$
            return $\text{ActualTrajectories}(T)$
        $\delta \leftarrow \gamma\delta$
function SampleSegments(S_0, T, δ, $N_{\max}$, K)
    $G \leftarrow \emptyset$
    $Q \leftarrow \text{SampleCells}(S_0, \delta)$
    while $Q \neq \emptyset$ **and** $|G| < N_{\max}$
        $C \leftarrow \text{pop}(Q)$
        Sample $\{s_1, \ldots s_K\}$ uniformly from C
        Sample $\{x_1, \ldots, x_K\}$ from $p(x)$
        for $i \in 1 : K$
            $s_{\text{new}} = f(s_i, x_i)$
            $C_{\text{new}} \leftarrow \text{FindCell}(s_{\text{new}}, \delta)$
            if $C_{\text{new}} \in T$ **or** $T = \emptyset$
                $G \leftarrow G \cup (C, C_{\text{new}}, x_i)$
    return G

G is constructed with edges (C_i, C_j, x_{ij}), where x_{ij} is the disturbance that transformed the system from a state $s_i \in C_i$ to a state $s_j \in C_j$ (line 2). An initially empty graph is constructed as follows:

- Some starting cells are sampled by sampling states in the initial set S_0, finding the corresponding cell, and adding the cell to a queue Q (line 13).
- On each iteration, a cell C is popped off the queue and K states are sample uniformly at random from within it (line 16). Then, a disturbance is randomly sampled for each state (line 17).
- Each sampled state is simulated forward one timestep based on the random disturbance x (line 19), resulting in a new state in cell C_{new} (line 20).
- To ensure the search focuses on promising regions of the state space, the edge (C, C_{new}, x) is only added to the graph if C_{new} is in the set of candidate trajectories T (line 21). On the first iteration, when the set of candidate trajectories is empty, all edges are added.

Once the graph is constructed, it is searched for candidate trajectories that connect cells in the initial set to cells in the failure set (line 5). If no candidate trajectories are found then the algorithm failed. Each candidate trajectory in T is simulated from its starting state using the disturbance applied at each segment to get an actual trajectory (line 8). The actual trajectory will not match that candidate trajectory, since the candidate trajectory is made up of disjoint segments. If the actual trajectory is a counterexample, then we return it. Otherwise, we reduce the grid size by a factor of γ (line 10) and repeat the procedure.

4.3 Las Vegas Tree Search

Las Vegas tree search (LVTS) [27] is a tree-based falsification algorithm based on two ideas:

1. The disturbance trajectory $\mathbf{x}$ should only be discretized in time as finely as it needs to be.
2. The system is more likely to be falsified by extreme values of the disturbance space than less extreme values.

To achieve the first goal, the disturbance trajectory is constructed piecewise from disturbances of different durations. Longer duration disturbances can be implemented as a repetition of a disturbance. Let the notation $x^{[\ell]}$ indicate that x is applied ℓ times. In LVTS, the set of possible disturbances at each step is discretized and represented by the set Y and the probability of selecting disturbances $x^{[\ell]}$ is $P(x^{[\ell]})$. To achieve the second goal, P is defined to favor distrubances with more extreme. LVTS (algorithm 7) takes as input Y, P, cost function c, and safety threshold ϵ.

The algorithm proceeds by growing a tree of disturbance trajectories $\mathbf{x}$. Each unfinished disturbance trajectory $\mathbf{x}$ has associated with it a set of explored disturbances (initialized to the empty set in line 2) and a set of unexplored disturbances (initialized to the set of all possible segments Y in line 3). For each iteration of the algorithm, a trajectory is stochastically expanded by sampling a new disturbance from P (line 7) and then concatenating

Algorithm 7 Las Vegas Tree Search

```
1: function LVTS(Y, P, c, ϵ)
2:     explored(x) ← ∅ for all x
3:     unexplored(x) ← Y for all x
4:     loop
5:         x ← ∅
6:         while unexplored(x) ≠ ∅ or explored(x) ≠ ∅
7:             Sample x from P(x)
8:             x_new ← [x x]
9:             if x ∈ unexplored(x)
10:                unexplored(x) ← unexplored(x) \ {x}
11:                c_, c̄ ← CostBounds(f(x_new))
12:                if c̄ < ϵ
13:                    return x_new
14:                if c_ > ϵ
15:                    break 4
16:                explored(x) ← explored(x) ∪ {x}
17:            x ← x_new
```

it to the current trajectory (line 8). If the new disturbance has not been explored before, it is removed from the unexplored set (line 10) and a lower and upper bound on the cost is computed from the function CostBounds. If the cost function is the robustness, then the bounds can be computed by the method of Dreossi *et al.* [26]. If the upper bound is lower then the safety threshold then a counterexample is found and returned (line 13). If the lower bound is greater than the safety threshold, then a counterexample is not possible for this disturbance trajectory and the algorithm restarts (line 15).

Disturbance segments are grouped by a integer $\ell \in \{1, \ldots, \ell_{\max}\}$ that controls the duration of the segment. For duration ℓ, the set Y_ℓ is given by

$$Y_\ell = \{x^{[\ell]} \mid x_i = x_{i,\min} + \alpha_{i,j}(x_{i,\max} - x_{i,\min})\}, \tag{33}$$

where x_i is the ith component of the disturbance space and $x_{i,\min}$ and $x_{i,\max}$ are the minimum and maximum values of that component. The factor

$$\alpha_{i,j} = (2j+1)/2^{b_i} \tag{34}$$

for any integer $j < (2^{\ell_{\max}-\ell} - 1)/2$ and $b_1 + \ldots + b_n = 1$. The construction of these sets ensures that segments with larger values of ℓ have longer duration and more extreme disturbance values. The full set of segments Y is the union of all the Y_ℓ up to a maximum level of refinement $\ell_{\max}$:

$$Y = \bigcup_{\ell=1}^{\ell_{\max}} Y_\ell. \tag{35}$$

The segment sampling distribution P ensures that disturbances with larger values of ℓ and lower costs are chosen more often. The distribution is constructed implicitly by assigning

a weight to each level of refinement

$$w_\ell = \frac{|\text{unexplored}(\mathbf{x}) \cap Y_\ell| + |\text{explored}(\mathbf{x}) \cap Y_\ell|}{2^{\ell_{\max}-\ell}|Y_\ell|} \tag{36}$$

and choosing ℓ with probability $w_\ell / \sum_{k=1}^{\ell_{\max}} w_k$. The weight is constructed to favor lower values of ℓ (due to the exponential factor in the denominator) until the number of unexplored edges goes to zero without finding many plausible trajectories (note that the explored set only increases if the trajectory remains plausible for falsification). Once the refinement level is selected, one of the the following four options is selected uniformly at random:

1. Sample x from $\text{unexplored}(\mathbf{x}) \cup Y_l$.
2. Sample x from $\text{explored}(\mathbf{x}) \cup Y_l$.
3. Choose the x from $\text{explored}(\mathbf{x}) \cup Y_l$ that minimizes the cost of $[\mathbf{x}\ x]$.
4. Choose the x from $\text{explored}(\mathbf{x}) \cup Y_l$ that minimizes the cost of $[\mathbf{x}\ x\ x']$ for all $x' \in \text{explored}([\mathbf{x}\ x])$.

Option 1 ensures exploration over the unexplored set, while options 2–4 increasingly exploit knowledge of trajectories with low costs.

5. Reinforcement Learning

One of the drawbacks to casting falsification as an optimization problem is the need to search over the entire space of disturbance trajectories, which scales exponentially with the number of simulation timesteps. This scalability problem may be alleviated if the environment and the system are modeled as a Markov decision process (MDP) [6]. In an MDP, the next state only depends on the previous state and the choice of disturbance. We can therefore find a policy (a function that maps states to disturbances) that can generate counterexamples when simulated. The policy generates a disturbance based on the state so we do not need to explicitly represent the entire disturbance trajectory, allowing for the solution of long time horizon falsification problems [22]. For an overview of MDPs and their solvers see the texts by Kochenderfer [6] or Sutton and Barto [68].

An MDP is defined by a transition model $P(s' \mid s, x)$ that gives the probability of arriving in state s' given the current state s, a disturbance (or action) x, a reward $R(s, x)$, and a discount factor γ that decreases the value of future rewards. In keeping with the notation of MDP literature, the reward serves the same purpose as the cost function, but is maximized rather than minimized, and can therefore be thought of as a measure of risk rather than safety. The goal of an MDP solver is to determine a mapping from states to disturbances $x = \pi(s)$, called a policy, that maximizes the expected discounted sum of rewards

$$\mathbb{E}\left[\sum_t \gamma^t R(s_t, x_t)\right]. \tag{37}$$

There are several concepts from the MDP literature that are useful for understanding the algorithms in this section. The first is the notion of the value function $V^\pi(s)$, which is

the expected discounted sum of future rewards when in the state s and following the policy π. The value function can be computed as the solution to the Bellman equation

$$V^\pi(s) = \mathbb{E}\left[R\left(s, \pi(s)\right) + \gamma V^\pi(s')\right]. \tag{38}$$

The optimal policy π^* is defined as

$$\pi^*(s) = \arg\max_\pi V^\pi(s). \tag{39}$$

The state-action value function for a policy is $Q^\pi(s, x)$ which is the value of being in state s, applying disturbance x, and then following the policy π. It is defined by the Bellman equation

$$Q^\pi(s, x) = \mathbb{E}\left[R(s, x) + \gamma Q^\pi\left(s', \pi(s')\right)\right]. \tag{40}$$

If the size of the state space and disturbance space is discrete then Q^π can be computed exactly with matrix inversion or dynamic programming [6]. If the state or action space is large or continuous (as is often the case for cyber-physical systems), Q^π can be estimated through random sampling and bootstrapping [68].

A major challenge in formulating an MDP is designing the reward function. Akazaki *et al.* [69] uses a reward that is a convex function of the temporal logic robustness

$$R(\mathbf{x}) = \exp\left(-\rho(\mathbf{x})\right) - 1, \tag{41}$$

which bounds the negative reward to -1. An approach known as adaptive stress testing (AST) [70], [71] defines an MDP reward that leads to the discovery of the most-likely counterexample. If the specification ψ is for the system to avoid reaching a set of failure states S_{fail}, then the reward is defined as:

$$R(s_t, x_t) = \begin{cases} 0 & \text{if } s_t \in S_{\text{fail}} \\ -\lambda & \text{if } s_t \notin S_{\text{fail}},\ t \geq t_{\max} \\ \log p(x_t \mid s_t) & \text{if } s_t \notin S_{\text{fail}},\ t < t_{\max}. \end{cases} \tag{42}$$

A large positive number λ is chosen to penalize disturbances that do not end in a failure state. The log probability of the disturbance is rewarded at each time step to encourage the discovery of likely failures.

Often, the reward requires some amount of domain knowlege. For example, Qin *et al.* [72] penalize disturbances that do not follow a domain-specific set of constraints, which is useful for getting adversarial agents in a driving scenario to follow traffic laws. Other examples of domain-specific heuristics are described in section 8.

We consider two types of reinforcement learning algorithms. First, we discuss Monte Carlo tree search algorithms which use online planning to maximize reward. Then, we present deep reinforcement learning algorithms which use offline learning and neural networks to estimate the optimal policy or value function.

5.1 Monte Carlo Tree Search

Monte Carlo tree search (MCTS) is an online planning algorithm for sequential decision making that has seen success in the games of go [73], chess [74], shogi [74], and falsification [70], [75]–[78]. Monte Carlo tree search uses online planning to determine the best actions to take from a starting state to maximize reward. A search tree is maintained over all of the paths tried as well as an estimate of the value function at each step. Each iteration in MCTS consists of four steps:

1. *Selection*: Starting from the root of the search tree, disturbances are selected sequentially until reaching a leaf node, with the goal of choosing disturbances that are more likely to lead to high reward. The most common selection heuristic is based on upper confidence bounds (UCB) [79], which provides a principled trade-off between exploration and exploitation. The selected disturbance is the one with the highest UCB given by

$$Q(s_t, x_t) + c\sqrt{\frac{\log N(s_t)}{N(s_t, x_t)}},$$

   where $Q(s_t, x_t)$ is the estimated state-action value at s_t, $N(s_t)$ is the number of visits to state s_t, $N(s_t, x_t)$ is the number of times x_t has been selected from s_t, and c is a hyperparameter that controls exploration.

2. *Expansion*: Once the algorithm arrives at a leaf node, a new disturbance is chosen, and a node is added to the tree with zero visits.

3. *Rollout*: The value of the new node is estimated by running a simulation from the new node while choosing disturbances according to a rollout policy until the episode terminates or the finite planning horizon is reached. The rollout estimate is denoted q.

4. *Backpropagation*: The value of the new node is used to update the value of all of its ancestors in the tree according to

$$Q(s_t, x_t) \leftarrow Q(s_t, x_t) + \frac{q - Q(s_t, x_t)}{N(s_t, x_t)}. \tag{43}$$

   The visit counts are also updated $N(s_t, x_t) \leftarrow N(s_t, x_t) + 1$.

These four steps are repeated until a stopping criterion is met such as computational budget, wall-clock time limit, or a threshold for the reward of the best solution found so far. At that point, the best trace can be returned, or, for longer time-horizon problems, the best candidate disturbance is selected and the process restarts, possibly retaining the subtree associated with the selected disturbance.

In problems where there is a discrete state or observation space, it makes sense to maintain separate state and disturbance nodes so if a state is repeated, the corresponding node in the tree can be reused, increasing the ability of the algorithm to reuse prior information. In most safety validation problems, however, the state space of the simulator will be continuous (or unavailable to the algorithm entirely), in which case the same state will rarely be sampled twice. To save memory, it is common to only include disturbance nodes in the search

tree, which is equivalent to setting the state equal to the concatenation of all disturbances that led to it ($s_t = \mathbf{x}_{1:t}$).

The discrete nature of nodes in the search tree are incompatible with a continuous disturbance space X. Delmas *et al.* [77] discretizes the disturbance space into a small number of discrete disturbances that are representative of the continuous space. As an alternative to discretization, when adding a new node to the tree, Lee *et al.* [70] samples a new disturbance x_t from a known distribution over disturbances by selecting a random seed uniformly at random and using it to produce disturbances from $p(x)$.

The UCB ensures that all disturbances are tried at least once from each node. When the disturbance space is large, however, this requirement restricts the depth of tree. An extension used to mitigate this problem is known as progressive widening [80], [81], where a new disturbance node is added as a child of node s_t when

$$|C(s_t)| \leq kN(s_t)^\alpha, \tag{44}$$

where $C(s_t)$ is the number of children, and k and α are hyperparameters.

Zhang *et al.* [75] combined MCTS with global optimization, using MCTS for exploration of the disturbance trajectory space and global optimization for refinement of the disturbance trajectories. The disturbance space is first discretized into $L_1 \times \ldots \times L_n$ equal-sized regions. Each node in the tree represents one of the regions in the disturbance space denoted B_t. When a new node is added to the tree at depth d, its value is estimated by solving the following constrained optimization problem:

$$\begin{aligned} \max_{\mathbf{x}} \quad & \mathbb{E}\left[\sum_t \gamma^t R(s_t, x_t)\right] \\ \text{s.t.} \quad & x_1 \in B_1, \quad \ldots, \quad x_d \in B_d. \end{aligned} \tag{45}$$

This approach can be combined with progressive widening when the disturbance space is large. Instead of randomly sampling a new region, optimization can be used to find the optimal region to expand. When adding a new disturbance at depth $d+1$, solve eq. (45) with the added constraint that $\mathbf{x}_{d+1}$ exists in the set of regions that have yet to be expanded. For ease of optimization, that subset may be further restricted to its convex subset [75]. Once the MCTS budget is spent, traces with high reward can be found by solving the constrained optimization problem of the highest performing branch of the tree.

The use of optimization for each new node of the search tree increases the computational cost of the algorithm. Zhang *et al.* [75] note that balancing the computational budget between optimization iterations and tree expansions is critical to the success of this approach. They compare simulated annealing (SA) [39], Globalized Nelder-Mead (GNM) [35], and covariance matrix adaptation evolution strategy (CMA-ES) [34] for the global optimization algorithms. It is noted that since CMA-ES has a built-in exploration strategy, it sees less improvement when combined for MCTS than SA and GNM. GNM, on the other hand, does not have an exploration strategy (beyond probabilistic restarts) and therefore sees a significant improvement with MCTS.

5.2 Deep Reinforcement Learning

Deep reinforcement learning (DRL) is a category of reinforcement learning that uses deep neural networks to represent the value function $V(s)$, the state-action value function $Q(s, x)$, or the policy $\pi(s)$. DRL has shown state-of-the art results in playing Atari games [82], playing chess [74], and robot manipulation from camera input [83]. In recent years, different DRL techniques have been applied to falsification and most-likely failure analysis [22], [69], [72], [84]–[88].

DRL algorithms are broadly split between value-function approaches, where the neural network is used to represent the value function, and policy search approaches, where the neural network represents the policy. There are advantages and disadvantages to both, and several algorithms are discussed below. The reader is referred to the original papers for implementation details.

If the disturbance space X is discrete, then the deep Q-network (DQN) [82] algorithm can be used for falsification [69], [72]. In DQN, the optimal Q function is estimated by a neural network that takes as input the state s and outputs a value for each discrete disturbance. Disturbances are selected greedily as

$$x = \arg\max_x Q(s, x) \tag{46}$$

with probability $1 - \delta$ and chosen at random with probability δ to encourage exploration. The parameters of the Q network are updated to minimize the mean squared error between the current value and a target

$$Q_{\text{target}}(s, x) = r(s, x) + \gamma \max_{x'} Q(s', x'). \tag{47}$$

Since the target value greedily selects the highest value disturbance, the Q-network can be trained on any sample of a state, disturbance, and reward (a feature known as off-policy learning). Many implementations of DQN therefore maintain a replay buffer that stores previous (s, x, r) tuples which are repeatedly used to update the network parameters. This reuse of data makes DQN a relatively sample-efficient algorithm. The main drawback of DQN is that it is incompatible with large or continuous disturbance spaces.

For large or continuous disturbance spaces, the policy itself is represented by a neural network (with parameters θ) that takes the state as input and either outputs a disturbance directly (e.g. $x = \pi_\theta(s)$) or outputs parameters of a distribution from which a disturbance can be sampled (e.g. for a normal distribution $[\mu, \sigma^2] = \pi_\theta(s)$ and $x \sim \mathcal{N}(\mu, \sigma^2)$). The policy is optimized to produce higher rewards using the policy gradient method [89]. Policy gradient methods can suffer from high variance and can be unstable during optimization. To improve optimization stability, an approach known as trust region policy optimization (TRPO) [90] restricts the amount a policy can change at each step. TRPO has previously been used for falsification of autonomous vehicles [22], [84], [85].

Another drawback of policy gradient methods is their inability to learn off-policy. Without data reuse, these methods can require a large number of simulations to converge. Newer approaches combine policy gradient methods with value function methods to create the actor-critic paradigm, which can perform well on problems with continuous disturbance spaces while also using previous simulation data to improve sample efficiency. Actor-critic

methods [91] use two neural networks, one for the policy (the actor network) and one for the value function (the critic network) and come in several varieties. Advantage actor critic (A2C) was used for falsification by Kuutti, Fallah, and Bowden [87]. Its more scalable counterpart, asynchronous advantage actor critic (A3C), was used by Akazaki *et al.* [69]. Behzadan and Munir [86] use another actor-critic method known as deep deterministic policy gradient (DDPG) combined with Ornstein-Uhlenbeck exploration [92].

One potential drawback of using DRL for black-box falsification is the requirement for a simulator that can be observed after each disturbance is applied. Koren and Kochenderfer [85] developed a DRL technique (also used by Corso *et al.* [84]) that does not require access to the simulator state. The technique uses a recurrent neural network (RNN) with long-short term memory (LSTM) layers as the policy [93]. The RNN maintains a hidden state akin to the state of the simulator that is used to make future decisions. The input to each layer is the previous disturbance and the initial state of the simulation, allowing the approach to generalize across initial conditions.

DRL can be combined with tree search in an approach called go-explore (GE) [94], which has been effective for hard-exploration problems with long time horizons. The first stage of GE is a random tree search over the disturbance space with heuristics that are designed to encourage efficient exploration. The heuristics bias the search towards unseen states, and away from states that have not resulted in productive rollouts. The second stage uses the backwards algorithm (BA) [95] to learn a policy that mimics the best disturbance trajectory found from the tree search. In expectation, rollouts from this policy obtain the same or better reward as the original trajectory.

While the original version of GE uses the state of the simulator when building the tree and training the robust policy, Koren and Kochenderfer [88] modified the algorithm to use the history of disturbances instead, reducing the access requirements of the simulator. GE was used for the falsification of an autonomous vehicle and was able to find counterexamples more reliably than MCTS on problems with long horizons and BA was able to find more probable failures than MCTS, or DRL. Furthermore, GE was able to find failures when no guiding reward heuristic was used, reducing the expert domain knowledge needed to perform falsification.

6. Importance Sampling Algorithms

For many cyber-physical applications, it is impossible to design an autonomous system that never violates a specification. In that case, we may wish to know how likely it is for a system to fail. To estimate the probability of failure, we can sample disturbance trajectories from the distribution with density $p(\mathbf{x})$ and compute the empirical probability of failure. A hypothesis test can be used to check if the probability failure is less than a desired threshold. Hypothesis testing for cyber-physical systems is the domain of statistical model checking [96], [97] and is not covered in this survey.

If the probability of failure is very small, i.e. a failure event is rare, then the Monte Carlo approach will require a large number of samples before converging to the true probability of failure [98]. To address this problem, we would like to artificially make failures more likely, and then weight them accordingly, to get an unbiased estimate of the probability of failure with fewer samples. This is the idea behind importance sampling.

Suppose we wish to estimate the probability that a system violates a specification. A failure occurs when the safety evaluation metric $c(\mathbf{x})$ is less than a safety threshold ϵ, represented here as the indicator function $\mathbb{1}\{c(\mathbf{x}) < \epsilon\}$. As in section 3, $c(\mathbf{x})$ is defined such that $c(\mathbf{x}) < \epsilon$ implies $f(\mathbf{x}) \notin \psi$. The probability of failure P_{fail} is the expectation over the probability density $p_{\mathbf{x}}$, i.e.

$$P_{\text{fail}} = \mathbb{E}_p \left[\mathbb{1}\{c(\mathbf{x}) < \epsilon\}\right]. \tag{48}$$

In importance sampling, we choose a proposal distribution $q(\mathbf{x})$ that makes failures more likely but has the property that $q(\mathbf{x}) > 0$ everywhere $p(\mathbf{x}) > 0$ (so all disturbances are still possible). The proposal distribution is sometimes referred to as the biased, sampled, or importance distribution. The importance sampling estimate of the probability of failure is done by taking N samples drawn from q and computing the weighted average

$$\hat{P}_{\text{fail}} = \frac{1}{N} \sum_{i=1}^{N} \frac{p(\mathbf{x}_i)}{q(\mathbf{x}_i)} \mathbb{1}\{c(\mathbf{x}_i) < \epsilon\}. \tag{49}$$

The variance of the importance sampling estimate is given by

$$\text{Var}(\hat{P}_{\text{fail}}) = \frac{1}{N} \mathbb{E}_q \left[\frac{(p(\mathbf{x}) \mathbb{1}\{c(\mathbf{x}_i) < \epsilon\} - q(\mathbf{x}) P_{\text{fail}})^2}{q(\mathbf{x})} \right]. \tag{50}$$

The goal of a good importance sampling distribution is to minimize the variance of the estimator $\hat{P}_{\text{fail}}$ so that fewer samples are needed for a good estimate. From eq. (50), we can see that a zero variance estimate can be obtained if we use the optimal importance sampling distribution

$$q^*(\mathbf{x}) = \frac{p(\mathbf{x}) \mathbb{1}\{c(\mathbf{x}) < \epsilon\}}{P_{\text{fail}}}. \tag{51}$$

Generating this distribution is not possible in practice because $c(\mathbf{x})$ is a black box and the normalization constant P_{fail} is the very quantity we would like to estimate. The algorithms in this section seek to estimate the optimal importance sampling distribution $q^*(\mathbf{x})$.

The choice of a proposal distribution can significantly affect the performance of importance sampling algorithms and a bad choice can lead to estimates with larger variance than the basic Monte Carlo approach. For example, if $q(x)$ is very small, where $p(\mathbf{x})$ is relatively large, then the weight $p(\mathbf{x})/q(\mathbf{x}) \gg 1$ and some samples will dominate the probability estimate (eq. (49)). To identify if a bad proposal distribution is chosen consider the size of the weights or compute the effective sample size. When samples with large weights are being drawn, Kim and Kochenderfer [99] suggest limiting the maximum weight by clipping the proposal distribution in regions with large weights.

6.1 Cross-Entropy Method

The cross-entropy method [100], [101] iteratively learns the optimal importance sampling distribution from a family of distributions $q(\mathbf{x}; \theta)$ parameterized by θ. The optimal distribution parameters θ^* are found by minimizing the KL-divergence between a proposal distribution $q(\mathbf{x}; \theta)$ and the optimal distribution $q^*(\mathbf{x})$, i.e.

$$\theta^* = \arg\min_{\theta} D_{\text{KL}}(q^*(\mathbf{x}) \parallel q(\mathbf{x}; \theta)), \tag{52}$$

where D_{KL} calculates the KL-divergence. Substituting the definitions, we can arrive at a stochastic optimization program

$$\begin{aligned}\theta^* &= \arg\min_\theta -\int_{\mathbf{x}} q^*(\mathbf{x}) \log q(\mathbf{x};\theta) d\mathbf{x} \\ &= \arg\max_\theta \int_{\mathbf{x}} \frac{\mathbb{1}\{c(\mathbf{x}) < \epsilon\} p(\mathbf{x})}{P_{\text{fail}}} \log q(\mathbf{x};\theta) d\mathbf{x} \\ &= \arg\max_\theta \mathbb{E}_\varphi \left[\mathbb{1}\{c(\mathbf{x}) < \epsilon\} \frac{p(\mathbf{x})}{q(\mathbf{x};\varphi)} \log q(\mathbf{x};\theta) \right] \\ &\approx \arg\max_\theta \frac{1}{N} \sum_{i=1}^{N} \left[\mathbb{1}\{c(\mathbf{x}_i) < \epsilon\} \frac{p(\mathbf{x}_i)}{q(\mathbf{x}_i;\varphi)} \log q(\mathbf{x}_i;\theta) \right],\end{aligned} \tag{53}$$

where φ is any set of parameters and $\mathbf{x}_i$ are sampled from $q(\mathbf{x};\varphi)$. eq. (53) can be solved analytically when the family of algorithms is in the natural exponential family (i.e. normal, exponential, Poisson, gamma, binomial, and others), and the solution corresponds to the maximum likelihood estimate of the parameters [101].

For an iterative solution to finding θ^*, we start by choosing a set of starting parameters θ_0 so that $q(\mathbf{x};\theta_0)$ is close to $p(\mathbf{x})$. Then, we iterate $k = 0, 1, \ldots$

1. Set $\varphi = \theta_k$.
2. Draw samples $\{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ from $q(\mathbf{x};\varphi)$.
3. Solve eq. (53) for θ_{k+1}.

One major challenge to this approach is the rarity of failure events. If all samples have $c(\mathbf{x}) > \epsilon$, then $\hat{P}_{\text{fail}} = 0$ and the algorithm may not converge to the optimal proposal distribution. One solution is to adaptively update the safety threshold ϵ at each iteration. At iteration k, a safety threshold ϵ_k and a rarity parameter ρ is chosen so that the fraction of samples that have $c(\mathbf{x}) < \epsilon_k$ is ρ. The parameter ρ is also known as the quantile level and is often set in the range $\rho = [0.01, 0.2]$ [99], [102]. The entire algorithm is shown in algorithm 8. Note that we assume ρN is an integer used for indexing on line 7.

The cross-entropy method has been used to estimate the probability of failure for aircraft collision avoidance systems [99] and autonomous vehicles [102]–[104]. Typically, probability distributions in the natural exponential family are used [99], [102], [103] so that cross-entropy updates can be performed analytically. Huang *et al.* [104] propose a method for using piecewise exponential distributions for more flexibility while retaining the ability to compute updates analytically. Sankaranarayanan and Fainekos [105] discuss piecewise uniform distributions over the disturbance space, and techniques for factoring the space to reduce the number of parameters needed.

6.2 Multilevel Splitting

Multilevel splitting [106] is a non-parametric approach to estimating the optimal importance sampling distribution. Rather than relying on an explicit probability distribution (as in the cross-entropy method), multilevel splitting relies on Markov chain Monte Carlo (MCMC) estimation and scales better to larger dimensions [107]. It has been applied to the

Algorithm 8 Cross-Entropy Method

1: **function** CrossEntropy(p, q, θ_0, ρ, c, ϵ)
2: $\theta \leftarrow \theta_0$
3: $k \leftarrow 0$
4: **loop**
5: Sample $\{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ from $q(\mathbf{x}; \theta_k)$
6: Sort $\{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ by $c(\mathbf{x}_i)$
7: $\epsilon_k \leftarrow \max(c(\mathbf{x}_{\rho N}), \epsilon)$
8: $\theta_{k+1} \leftarrow \arg\max_\theta \frac{1}{N} \sum_{i=1}^{N} \left[\mathbb{1}\{c(\mathbf{x}_i) < \epsilon\} \frac{p(\mathbf{x}_i)}{q(\mathbf{x}_i; \theta_k)} \log q(\mathbf{x}_i; \theta)\right]$
9: **if** $\epsilon_k < \epsilon$
10: **break**
11: $k \leftarrow k + 1$
12: **return** EstimateProbability(p, q_{θ_k})

estimation of probability of failure of autonomous driving policies with a large number of parameters [108].

The idea of multilevel splitting is to define a set of threshold levels $\infty = \epsilon_0 > \epsilon_1 > \ldots > \epsilon_K = \epsilon$ and assume that the probability of failure can be computed as a Markov chain of the form

$$P(c(\mathbf{x}) < \epsilon) = \prod_{k=1}^{K} P(c(\mathbf{x}) < \epsilon_k \mid c(\mathbf{x}) < \epsilon_{k-1}). \tag{54}$$

Given enough levels (i.e. a large enough K), each conditional probability

$$P_k = P(c(\mathbf{x}) < \epsilon_k \mid c(\mathbf{x}) < \epsilon_{k-1}) \tag{55}$$

is much larger than $P(c(\mathbf{x}) < \epsilon)$ and can therefore be computed efficiently with basic Monte Carlo methods. Algorithm 9 outlines the algorithm. At iteration k, N samples of $\mathbf{x}$ are sorted by their cost function (line 8) and used to estimate the conditional probability P_k (line 10). The total probability is updated based on the Markov assumption (line 11). All samples with $c(\mathbf{x}) > \epsilon_k$ are discarded, and the remaining samples are resampled to get back up to N samples (line 12). Those samples perform a random walk of M steps using a kernel $T(\mathbf{x}' \mid \mathbf{x})$ (line 13). The process repeats until the level reaches the true safety threshold. The choice of levels ϵ_k can be done adaptively [109] by selecting a rarity parameter ρ such that we keep ρN samples per iteration (line 9).

6.3 Classification Methods

Supervised learning can be used to classify disturbances as safe or unsafe. That classification can then be combined with importance sampling to estimate the probability of failure of the system. Supervised learning often requires observing more than one example of a failure, so these approaches are most applicable when failures can be found relatively easily. One approach is to use a space-filling sampling plan for the disturbance [110] to ensure good coverage of the disturbance space, but this fails to scale to high dimensions. Another approach is to use previous versions of the system that are far less safe [111]. Known as a *continuation*

Algorithm 9 Adaptive Multilevel Splitting Method

function MulitlevelSplitting(p, N, M, T, c, ϵ)
    $P_{\text{fail}} \leftarrow 1$
    $k \leftarrow 1$
    $\epsilon_0 = \infty$
    Sample $\{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ from $p(\mathbf{x})$
    while $\epsilon_k > \epsilon$
        $k \leftarrow k + 1$
        Sort $\{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ by $c(\mathbf{x}_i)$
        $\epsilon_k \leftarrow \max(c(\mathbf{x}_{\rho N}), \epsilon)$
        $P_k \leftarrow \frac{1}{N} \sum_{i=1}^{N} \mathbb{1}\{c(\mathbf{x}_i) < \epsilon_k\}$
        $P_{\text{fail}} \leftarrow P_{\text{fail}} P_k$
        Resample $\{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ from $\{\mathbf{x}_1, \ldots, \mathbf{x}_{\rho N}\}$
        Random walk M steps using $T(\mathbf{x}' \mid \mathbf{x})$ for each $\{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$
    return P_{fail}

approach, this technique is well suited to black-boxes that have learned behavior. During the learning process, the system will fail more easily (but in related ways) to the version that is ultimately being tested. Therefore, earlier versions of the system can be used for classification of disturbances.

One way to combine supervised learning with importance sampling was explored by Huang *et al.* [110]. They built a proposal distribution centered on the boundary between safe and unsafe disturbances. If $q(\mathbf{x}; \theta)$ is a proposal distribution with parameters θ, then they search for the point $\mathbf{x}^*$ in the set of failures that maximizes the probability of the proposal distribution:

$$\mathbf{x}^* = \arg\max_{\mathbf{x}} \mathbb{1}\{c(\mathbf{x}) < \epsilon\} q(\mathbf{x}; \theta). \tag{56}$$

Then, they adjust the parameters of the proposal distribution such that the mean is located at $\mathbf{x}^*$. This approach has shown to construct an efficient proposal distribution when the unsafe set is convex [112]. In practice, it is unlikely that the set is convex, but we can find a linear boundary between safe and unsafe examples using the support vector machine (SVM) algorithm [113] if the disturbance space is lifted to a higher dimension through a mapping $\phi(\mathbf{x})$.

Once a boundary is found, the choice of distribution q determines the feasibility of computing $\mathbf{x}^*$. One approach is to represent q by a mixture of K Gaussians with weights α as

$$q_{\text{GMM}}(\mathbf{x}; \mu_i, \sigma_i) = \sum_{i=1}^{K} \alpha_i \mathcal{N}(\mathbf{x}; \mu_i, \sigma_i). \tag{57}$$

The optimal mean for each Gaussian model $\mathbf{x}_i^*$ is determined separately and the proposal distribution is reconstructed by

$$q^*_{\text{GMM}}(\mathbf{x}; \mathbf{x}_i^*, \sigma_i) = \sum_{i=1}^{K} \alpha_i \mathcal{N}(\mathbf{x}; \mathbf{x}_i^*, \sigma_i). \tag{58}$$

Note that a Gaussian mixture model (GMM) can be obtained in the lifted space $\tilde{q}_{\text{GMM}}(\phi(\mathbf{x}))$ and then reduced to the true disturbance space by integrating over the extra dimensions. The approach is outlined in algorithm 10.

Algorithm 10 Classification-based Importance Sampling

function ClassificationImportanceSampling(p, M, ϕ, c, ϵ)
  Sample $\{\mathbf{x}_1, \ldots, \mathbf{x}_M\}$ from $p(\mathbf{x})$ and fit a GMM $\tilde{q}_{\text{GMM}}(\phi(\mathbf{x}); \mu_i, \sigma_i)$
  Sample $\{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ and compute $\{\mathbb{1}\{c(\mathbf{x}_1) < \epsilon\}, \ldots, \mathbb{1}\{c(\mathbf{x}_N) < \epsilon\}\}$
  Lift the disturbances by computing $\{\phi(\mathbf{x}_1), \ldots, \phi(\mathbf{x}_N)\}$
  Apply SVM on pairs $\{(\phi(\mathbf{x}_1), \mathbb{1}\{c(\mathbf{x}_1) < \epsilon\}), \ldots, (\phi(\mathbf{x}_N), \mathbb{1}\{c(\mathbf{x}_N) < \epsilon\})\}$
  Determine the optimal points ϕ_i^*
  Construct $\tilde{q}^*_{\text{GMM}}(\phi(\mathbf{x}); \phi_i^*, \sigma_i)$
  Marginalize to $q^*(\mathbf{x})$
  return EstimateProbability(p, q^*)

An alternative approach by Uesato *et al.* [111] uses supervised learning to estimate the probability of failure $\hat{P}_{\text{fail}}(\mathbf{x})$ for a disturbance $\mathbf{x}$. The function $\hat{P}_{\text{fail}}(\mathbf{x})$ can be represented using a neural network or some other model and is trained on failure examples. The optimal importance sampling distribution was proven to be

$$q^*(\mathbf{x}) = \frac{\sqrt{\hat{P}_{\text{fail}}(\mathbf{x})} p(\mathbf{x})}{\mathbb{E}_p\left[\sqrt{\hat{P}_{\text{fail}}(\mathbf{x})}\right]}, \tag{59}$$

which can be sampled from using rejection sampling.

6.4 Approximate Dynamic Programming

While most importance sampling approaches focus on the entire space of disturbance trajectories, the approach of Corso, Lee, and Kochenderfer [114] uses the framework of sequential decision making to find the optimal importance sampling policy $q(x \mid s)$ for a simulation state s. It is shown in that work that for a Markovian system and simulator, the optimal importance sampling policy is given by

$$q(x \mid s) = \frac{p(x \mid s) P_{\text{fail}}(s')}{P_{\text{fail}}(s)}, \tag{60}$$

where s' is deterministically reached after disturbance x is applied in state s. The probability of failure $P_{\text{fail}}(s)$ can be estimated through the approximate solution of the Bellman equation

$$P_{\text{fail}}(s) = \begin{cases} 1 & \text{if } s \in S_{\text{fail}} \\ 0 & \text{if } s \notin S_{\text{fail}},\ s \in S_{\text{term}} \\ \sum_a p(x \mid s) P_{\text{fail}}(s') & \text{otherwise} \end{cases}, \tag{61}$$

where S_{term} is the set of terminal states that are not failure states. Local approximation dynamic programming and Monte Carlo estimations are two successful approaches for solving eq. (61) [114].

7. Problem Decomposition Techniques

One of the most significant challenges to overcome for the safety validation of autonomous systems is that algorithms often scale poorly to large disturbance spaces and state spaces. This section will address two approaches for decomposing the falsification problem into more manageable subproblems to accelerate finding counterexamples.

7.1 State Space Decomposition

The approach presented by Corso, Lee, and Kochenderfer [114] involves decomposing the simulator state and disturbance spaces into independent components, and finding failures for each component. Originally done in the context of multiple distinct actors in a simulated driving environment, this approach can be used with any simulation that has components that can be simulated individually. To separate the state space into M different components, define a decomposition operator D such that

$$\{s^{(1)}, \ldots, s^{(M)}\} = D(s) \tag{62}$$

and the disturbance space $X^{(i)}$ associated with the state $s^{(i)}$ is smaller than the full disturbance space (i.e. $|X^{(i)}| < |X|$). For each subproblem, solve for a policy that finds counterexamples

$$x^{(i)} = \pi^{(i)}(s^{(i)}) \tag{63}$$

and then combine the policies with a fusion function F such that

$$x = F(\pi^{(1)}, \ldots, \pi^{(M)})(s). \tag{64}$$

An approach for solving the subproblems that is amenable to policy fusion is approximate dynamic programming (see section 6.4). In that case, the subproblem policy is defined by computing the probability of failure for each state component $P^{(i)}_{\text{fail}}$. The fusion function can then apply simple arithmetic operations (like mean, max, or min) to arrive at a joint policy

$$\tilde{P}_{\text{fail}}(s) = F\left(P^{(1)}_{\text{fail}}(s^{(1)}), \ldots, P^{(1)}_{\text{fail}}(s^{(m)})\right) \tag{65}$$

$$x = \frac{p(\mathbf{x} \mid s) P_{\text{fail}}(s')}{P_{\text{fail}}(s)}, \tag{66}$$

where s' is the deterministic state reached after applying disturbance x at state s.

If the fusion function is simple, then it may not capture the joint interactions between multiple disturbance components [114]. To address this problem, a global correction factor can be learned from the full simulation. We have

$$P_{\text{fail}}(s) \approx \tilde{P}_{\text{fail}}(s) + \delta P_\theta(s). \tag{67}$$

The correction factor δP_θ is trained using rollouts $\{s_1, \ldots, s_N\}$ from the full simulation and minimizing the difference between the estimate $\tilde{P}_\theta(s_i)$ and the actual discounted return $G(s_i)$ where

$$G(s_i) = \mathbb{1}\{s_N \in S_{\text{fail}}\} \frac{\prod_{t=i}^{N} p(x_t \mid s_{t-1})}{\pi(x_t \mid s_{t-1})}. \tag{68}$$

This approach has been shown to increase the number of failures found in a complex driving environment with a large disturbance space [114].

7.2 Compositional Approach with Machine Learning Components

The approach of Dreossi, Donzé, and Seshia [115] is applicable when the system contains a machine-learned component that performs classification (such as a neural network based perception system). Knowing something about the structure of the system makes this approach gray-box, but due to the prevalence of ML components, it is still widely applicable.

The algorithm is performed with the following steps:

- The ML component of interest is partitioned from the rest of the system and environment $f(\mathbf{x})$. The ML component, which has a large disturbance space, is replaced with an idealized abstraction with a much smaller input space.
- The simulator with the idealized ML component is separated into two versions: 1) $f^+(\mathbf{x})$ where the ML component behaves as well as possible (i.e. classifying all inputs correctly), and 2) $f^-(\mathbf{x})$ where the ML component behaves poorly.
- For each version of the simulator, a traditional falsification algorithm is used to partition the disturbance space into the regions that satisfy the specification safe($\mathbf{x}$ and regions of failures fail($\mathbf{x}$). The notation safe$(\mathbf{x})^{\pm}$ indicates the safe disturbance set for $f^{\pm}(\mathbf{x})$. Similar notation is used for the failure set.
- To find counterexamples that were caused by the ML component, find falsifying examples in the set safe$(\mathbf{x})^+ \setminus$ safe$(\mathbf{x})^-$. The set safe$(\mathbf{x})^-$ is removed because no failure is possible in this region even with the ML component functioning as poorly as possible. This part of the disturbance space is referred to as the region of interest.
- Finally, use a separate analyzer (possibly white-box) to identify the high-dimensional inputs that lead to failures (i.e. misclassifications) in the region of interest.

This approach has been able to find counterexamples in a neural network perception system used by an autonomous vehicle [115]. A similar approach by Julian, Lee, and Kochenderfer [78] uses a white-box neural network analyzer combined with Monte Carlo tree search to find failure trajectories of an image-based control system.

8. Applications

Autonomous cars and aircraft are two major application domains of black-box safety validation methods. While there are a variety of scenarios within these application domains, a common underlying principle is that of *miss distance* as a reward heuristic. For most scenarios, it is possible to use some sort of distance metric to measure how close the system came to a failure during the scenario. This distance is then used to allow optimization to find counterexamples. The autonomous vehicles and aircraft application domains are covered in more detail below.

8.1 Autonomous Driving

Within the field of autonomous driving, there are multiple scenarios that are commonly used for testing:

Lane-Following. Lane-following is one of the most common examples in the autonomous vehicle field, and has been used to test systems ranging from full-stack systems [102] to systems with ideal perception [86] to systems that are not fully autonomous like advanced driver-assistance systems (ADAS) [61]. The system under test tries to maintain a desired speed in the current lane and may [60] or may not [87] have the ability to change lanes. The goal is often to find collisions, most commonly rear-end collisions. Scenario variations include both highway driving [108] and local road driving [116].

Intersection Scenarios. In an intersection scenario, the system under test approaches a stoplight [116], stop sign [45], crosswalk [22], or other form of intersection and must proceed through without a failure. Failures can include collisions with pedestrians [88] or other vehicles [116]. Failures may also include violations of traffic laws, although we found no previous work in this area.

Lane Change Scenarios. Lane change scenarios can involve the test vehicle initiating a lane change or reacting to one [72], [103]. Existing work has considered ADAS and other driver aid systems [110]. Failure modes include rear-end collisions as well as side collisions caused by turning into an occupied lane.

Platooning Vehicles. Hu, Lygeros, and Sastry [117] present a platooning scenario, where a number of cars are following a lead car while keeping a specific trailing distance. The vehicles are subjected to various stochastic disturbances arising from road conditions, wind conditions, or the presence of human operators. The goal of the system is to minimize the time that platoon vehicles spend in "chasing" mode, where they attempt to catch up to the vehicle ahead. The system fails when any of the vehicles gets too far from or too close to the leading vehicle.

Within these applications, researchers have developed domain specific techniques to improve performance. The most common approach is to use miss distance, the physical distance between the test vehicle and other vehicles in the simulation, as a reward heuristic to allow easier optimization [88]. Miss distance is also the underlying principle for robust semantics of temporal formulas when applied to autonomous vehicles [75]. Some work has been done on identifying situations where a collision is imminent instead of the collision itself, sometimes called "unsafe states" [61] or "boundary cases" [60] since these regions may be easier to find.

8.2 Autonomous Flying and Aircraft Collision Avoidance

Black-box validation techniques have also been applied to aircraft in multiple ways:

Flight Control Software. Delmas *et al.* [77] present an application where the controller must keep an aircraft in steady flight in response to disturbances such as wind or pilot inputs. Failures include reduced flight quality, autopilot disengagements, and overshoots of expert-defined thresholds. A second application is presented by Ernst *et al.* [118], where an F-16 controller performs automatic maneuvers to avoid ground collisions. Validation algorithms search for violations of a minimum altitude from various initial conditions. Julian, Lee, and Kochenderfer [78] perform safety validation on a neural network controller with camera inputs for aircraft taxiing.

Collision Avoidance. There are several examples of applications to collision avoidance problems for aircraft. Lee *et al.* [70] validate the next-generation aircraft collision avoidance system (ACAS X), which makes recommendations to pilots to avoid mid-air collisions. The system may or may not coordinate its recommendations with the other aircraft. A failure occurs when the aircraft pass too close to one another, known as a near mid-air collision (NMAC). Esposito, Kim, and Kumar [54] validate a control system that guides a group of planes from an initial location to a final destination in the presence of stochastic wind disturbances. A failure in this case is a collision between any two aircraft.

Flight Path-Planning. Systems are given a mission, which is a series of waypoints that the system must navigate [119], or a target location and keep-out zones along the path [120]. Falsifying trajectories may be different mission parameters or disturbances during mission execution, such as wind and sensor noise. Failures occur when the system enters areas defined as off-limits, collides with an obstacle, or produces a software error.

8.3 General Systems

Hybrid Systems. Algorithms for black-box safety validation have also been applied to a number of hybrid systems outside of autonomous driving and autonomous aircraft. Hoxha, Abbas, and Fainekos [121] present an automatic transmission system model, which is widely used as a benchmark problem in the literature. The system under test selects a gear based on throttle and brake inputs as well as state information such as current engine load and car speed. Failure events include violations of speed thresholds and changing gears too often. Jin *et al.* [122] present another widely used benchmark problem from the automotive field. The system is an abstract fuel controller for an automotive powertrain. The controller receives inputs such as fuel-flow measurements and throttle and must output a fuel command. Failures may be either steady-state, such as the violation of an air-to-fuel ratio, or transient, such as pulses that violate a settling-time constraint.

Non-automotive systems have also been considered as well. Kim, Esposito, and Kumar [55] analyze a thermostat model under various environmental conditions and test whether the heater will be active for longer than a certain proportion of time. Schuler, Adegas, and Anta [123] present a simplified model of a wind turbine, which attempts to generate as much power as possible based on the current wind speed. Failures are violations of safety criteria, which include violations of thresholds on tower base moment and rotor speed.

Controllers. Systems with neural network controllers have also become common case studies for validation techniques. Yaghoubi and Fainekos [41] test a steam condenser controlled by a recurrent neural network (RNN). The system modulates steam flow rate based on energy balance and cooling water mass balance. A failure occurs when the pressure falls outside the acceptable range. Yaghoubi and Fainekos [41] also present a generalized non-linear system controlled by a feed-forward neural network. A failure is defined as a constraint on the reference signal value. Ernst *et al.* [118] study a neural network controlled magnet levitation system. The controller takes a reference position as input and attempts to move the magnet to track the given reference position. A failure occurs if the magnet does not stabilize to a position close enough to the reference position within some time limit.

Planning Modules. Falsification of planning modules is common among the autonomous vehicle and aircraft applications, but there are examples in other domains as well. Kim, Esposito, and Kumar [55] present a hovercraft navigation application. The validation task tests whether the hovercraft can successfully navigate to some goal region while subjected to stochastic wind disturbances. Similarly, Zhang *et al.* [75] validate a free-floating robot system that must navigate to a desired target location. Fehnker and Ivančić [124] present a generalization of this task, where an agent is moving in a discrete 2D environment, sometimes called a *gridworld* task. In their formulation, there are states that must be reached and states that must be avoided. A violation of either constraint is considered a failure.

9. Existing Tools

Implementations of various safety validation algorithms have been made available as tools for others to use. The existing tools range from open-source academic-based toolboxes to closed-source commercial software. Each of these tools creates falsifying disturbance trajectories to a system given a set of system requirements that should be satisfied. Certain tools also perform most-likely failure analysis and failure probability estimation. Many of the existing tools interface with the MATLAB programming language/environment to stress industry standard Simulink models. The tools surveyed in this section focus on black or gray-box testing of cyber-physical systems and although some of these tools include additional functionality, this section focuses on features of the safety validation components. A brief overview of the existing safety validation tools will be discussed and their benefits and restrictions will be highlighted.

9.1 Academic Tools

Many of the existing safety validation tools are products of academic research and released as experimental prototypes. Although these tools are prototypes, several have matured enough to gain wide-spread usage and acceptance [26], [30], [67], [119], [125]. Two particular falsification tools have become benchmark standards in the field: S-TaLiRo [126] and Breach [33]. Both are open-sourced MATLAB toolboxes for optimization-based falsification. While most of the tools are open-sourced, two other tools referenced in the falsification literature are not publicly available but still discussed. Although their respective papers indicate how to recreate their work, none of those tools have code or executables available online. The following collection is organized into optimization-based and reinforcement learning-based tools.

9.2 Optimization-based Tools

The tools in this section employ a standard optimization-based technique to search for counterexamples. Section 3 outline the approaches implemented by the following tools.

S-TaLiRo. S-TaLiRo (Systems Temporal Logic Robustness) [32], [126] is a simulation-based MATLAB toolbox for temporal logic falsification of non-linear hybrid systems. S-TaLiRo parameterizes the disturbance space to reformulate the falsification problem as an optimization problem. S-TaLiRo instructs the user to specify system requirements as metric temporal logic (MTL) formulas and then constructs the optimization cost function

to minimize a global robustness metric. Various optimization techniques are included in the S-TaLiRo toolbox, such as simulated annealing (described in section 3.1), genetic algorithms (described in section 3.2), stochastic optimization with adaptive restarts (described in section 3.3), the cross-entropy method (described in section 6.1), and uniform random sampling. S-TaLiRo is designed to analyze arbitrary MATLAB functions or Simulink/Stateflow models. S-TaLiRo is open-source and available under the GNU General Public License (GPL).[1]

Specific add-ons to S-TaLiRo have been implemented that extend the core falsification functionalities. These add-ons generally provide other solution methods or provide additional simulation environments that interface with S-TaLiRo. Two applicable add-ons are described as follows.

DP-TaLiRo. DP-TaLiRo (Dynamic Programming Temporal Logic Robustness) [25] is an algorithmic add-on to S-TaLiRo that can be used as the robustness computation engine. A dynamic programming algorithm is implemented for the robustness metric. DP-TaLiRo is included as a standard part of S-TaLiRo.

Sim-ATAV. Sim-ATAV (Simulation-based Adversarial Testing for Autonomous Vehicles) [116] is an add-on to S-TaLiRo that provides a simulation framework for autonomous vehicle testing. Sim-ATAV is an open-source Python framework and is available under the MIT license.[2]

Breach. Breach [33], [115] is a simulation-based MATLAB toolbox for falsification of temporal logic specifications for hybrid dynamical systems, similar to S-TaLiRo. Breach uses optimization-based techniques including simulated annealing (described in section 3.1), genetic algorithms (described in section 3.2), globalized Nelder-Mead [35], and CMA-ES [34]. The user-defined system requirements are input using signal temporal logic (STL) formulas. These requirements, i.e. specifications, are used to construct a cost function to be minimized based on a robustness metric. Breach is designed to test arbitrary MATLAB functions and Simulink models and includes a MATLAB graphical user interface (GUI) that gives the user access to the input parameter sets, temporal logic formulas, and trajectory visualizations. Breach is open-source and available under the BSD license.[3]

Rrt-Rex. Rrt-Rex (Rapidly-exploring Random Tree Robustness-guided Explorer) [26] is a MATLAB falsification tool that focuses on coverage given a computational budget. A Simulink model and user-defined STL requirements are taken as input. RRT path planning algorithms are used to search the disturbance space for falsifying cases, guided by a combined state space coverage metric and a robustness satisfaction metric. Section 4 discusses the RRT approach in detail. Rrt-Rex is not currently publicly available.

9.3 Reinforcement Learning-based Tools

As a direct replacement to optimization algorithms, reinforcement learning can be used as the central idea behind searching for falsifying trajectories. The following tools imple-

1. `https://sites.google.com/a/asu.edu/s-taliro`
2. `https://sites.google.com/a/asu.edu/s-taliro/sim-atav`
3. `https://github.com/decyphir/breach`

ment reinforcement learning algorithms as solvers for the falsification problem. Sections 5.1 and 5.2 describe the reinforcement learning algorithms implemented in the tools.

FalStar. FalStar [75] is a prototype Scala tool for falsification of cyber-physical systems that interfaces with MATLAB through a Java API. FalStar uses reinforcement learning combined with optimization techniques to generate counterexamples. Techniques include Monte Carlo tree search with stochastic optimization as described in section 5.1 and adaptive Las Vegas tree search (aLVTS) as described in section 4.3. FalStar requires a Simulink model as input and uses the above techniques to generate counterexamples to the STL specifications. FalStar can also interface directly with the Breach toolbox to use the available solvers implemented in Breach. FalStar is open-source and available under the BSD license.[4]

falsify. falsify [69] is a prototype simulation-based falsification tool that uses deep reinforcement learning. Common among the academic tools, falsify can interface directly with MATLAB functions and Simulink models. Implementing a robustness-guided approach, falsify defines the reward as a convex function of the robustness, as described in section 5. This robustness-guided reward function is used by two deep reinforcement learning algorithms implemented in falsify: asynchronous advantage actor critic (A3C) and double deep-Q network (double DQN), both described in section 5.2. The falsify tool is not currently publicly available.

AST Toolbox. The AST Toolbox (Adaptive Stress Testing Toolbox) [22] is a Python toolbox for safety validation, which includes falsification, most-likely failure analysis, and failure probability estimation. The AST Toolbox uses reinforcement learning to find the most likely failures of black-box systems. Two reinforcement learning techniques used as solvers are included: Monte Carlo tree search (described in section 5.1) and deep reinforcement learning (described in section 5.2). The AST Toolbox is built on top of two popular reinforcement learning packages, namely, OpenAI Gym [127] and Garage [5]. Building off of these packages gives the user access to widely used reinforcement learning benchmarking problems.

To test a system, users must provide the definitions for three basic interfacing functions. These interfacing functions allow the tool to interact with the black-box system or simulator. The interface defines how to *initialize* the system, how to *step* the system forward (returning indications of found failures or a real-valued measure of distance to a failure), and finally a means to determine if the system is in a *terminal* state. While the implementation of the interface is restricted to Python, the user can call out to existing executables or other languages from within Python. As an example, Python can interface with MATLAB through the MATLAB Engine API for Python.[6] The AST Toolbox is open-source and available under the MIT license.[7]

A related toolbox, called AdaptiveStressTesting.jl [70], follows a similar paradigm as the AST Toolbox but is implemented in the Julia programming language [128]. Julia can

4. `https://github.com/ERATOMMSD/falstar`
5. https://github.com/rlworkgroup/garage
6. `https://www.mathworks.com/help/matlab/matlab-engine-for-python.html`
7. `https://github.com/sisl/AdaptiveStressTestingToolbox`

interface directly with many other programming languages,[8] and notably, Julia can interface with MATLAB through the MATLAB.jl package.[9] AdaptiveStressTesting.jl is open-source and available under the Apache License Version 2.0.[10]

9.4 Commercial Tools

Certain techniques for safety validation of cyber-physical systems have become available as commercial toolboxes. This section briefly describes these commercially available tools relating to black-box safety validation.[11]

Reactis. Reactis [129] is a simulation-based MATLAB tool from Reactive Systems for falsification of Simulink models. Reactis has three components: Reactis Tester, Reactis Simulator, and Reactis Validator. The Reactis Tester controls the generation of falsifying trajectories. It uses a patented technique called *guided simulation* which uses proprietary algorithms and heuristics to generate trajectories to maximize coverage for falsification. Then, the Reactis Simulator runs the system under test given the purposed falsifying tajectory. The last component, the Reactis Validator, uses proprietary techniques to search for violations of user-defined model specifications. Reactive Systems also has a version of Reactis for C [130] that has analogous Reactis components for systems developed in the C programming language. Reactis is commercially available through Reactive Systems, Inc.[12]

TestWeaver. TestWeaver [131] is a simulation-based falsification tool from Synopsys. TestWeaver uses proprietary search algorithms to generate falsifying trajectories while maximizing coverage. User-defined system requirements and worst-case quality indicators are used to guide the search. Extensive knowledge of the underlying system may be required to provide useful worst-case quality indicators, which may limit the application. TestWeaver can interface with other simulation frameworks to control the disturbance trajectories via libraries in Simulink, Modelica, Python, and C. TestWeaver is commercially available through Synopsys, Inc.[13]

TrustworthySearch API. TrustworthySearch API [108] is a risk-based falsification and probability estimation framework from Trustworthy AI. TrustworthySearch API is specifically designed for black-box autonomous vehicle safety validation. The tool implements a probabilistic-based approach through adaptive importance sampling techniques described in section 6. Importance sampling is a stand in for traditional optimization algorithms used to search the disturbance space for falsification. The use of a proposal distribution biased towards rare failure events ensures that these low probability events are sampled more frequently. Adaptive multilevel splitting (AMS), described in section 6.2, is implemented to estimate this biased distribution from data. Along with falsification, the tool can also perform failure event probability estimation, unlike other commercially available products and most academic tools. To estimate the failure probability, the probability from

8. `https://github.com/JuliaInterop`
9. `https://github.com/JuliaInterop/MATLAB.jl`
10. `https://github.com/sisl/AdaptiveStressTesting.jl`
11. The authors are not affiliated with any of the companies.
12. `https://www.reactive-systems.com/`
13. `https://www.synopsys.com/verification/virtual-prototyping/virtual-ecu/testweaver.html`

the biased distribution is reweighted according to the likelihood from the original unbiased distribution. Although specific to autonomous vehicle safety validation, we include TrustworthySearch API to highlight recent advancements of black-box safety validation tools. TrustworthySearch API is commercially available through Trustworthy AI, Inc.[14]

9.5 Toolbox Competition

An academically-driven friendly competition for systems verification, called ARCH-COMP, has been held annually since 2017 [118]. One category within the competition is the falsification of temporal logic specifications for cyber-physical systems. Falsification researchers compare their tools against common benchmark problems and use the competition to track state-of-the-art falsification tools.

As detailed in their 2019 report [118], S-TaLiRo, Breach, FalStar, and falsify participated in the most recent competition. Six benchmark problems from the literature were used to evaluate each tool. The tools were evaluated based on their falsification rate and statistics on number of simulations required to find a falsifying trajectory. The outcome of the 2019 competition showed that falsify had the most success, only requiring a single simulation in certain benchmark problems. A notable emphasis on repeatability of results was made during this recent competition. For future competitions, we suggest a comparison metric for tool runtime to help assess relative computational timing complexities. Continuing to mature the competition as more falsification methods arise will help drive discussion around falsification tool design decisions. The competition's benchmarks and results are available online.[15]

9.6 Tools Discussion

The available tools can be categorized based on which aspects of the safety validation process they implement, as described in section 2.2: falsification, most-likely failure analysis, and failure probability estimation. Note that all of the available tools preform falsification as their core task. Regarding academic tools, S-TaLiRo, Breach, FalStar, Rrt-Rex, and falsify are falsification-only tools. As for commercial tools, Reactis and TestWeaver also fall into the falsification-only category. TrustworthySearch API is the only commercial tool that also preforms failure probability estimation and the AST Toolbox is the only academic tool that preforms most-likely failure analysis. A full comparison is outlined in table 1.

As for solution techniques, the tools S-TaLiRo and Breach take a standard optimization-based approach, while Rrt-Rex reformulates the falsification problem to solve it using path planning algorithms. The AST Toolbox and falsify are based on reinforcement learning and FalStar combines reinforcement learning and global optimization techniques for further refinement of the disturbance trajectory search. Common among all of the tools is an interface to MATLAB, with a particular emphasis on testing Simulink models. This is evident in the survey and can be attributed to an industrial emphasis on the use of MATLAB/Simulink models for prototyping. A common component specific to academic falsification-only tools is the use of temporal logic to specify system requirements—encoded in signal temporal logic (STL) or metric temporal logic (MTL). Although expressive, requiring the strict usage of a

14. `http://trustworthy.ai/`
15. `https://gitlab.com/goranf/ARCH-COMP`

Table 1: Black-box Safety Validation Tools

	Safety Validation Problem				
Tool	Falsification	Most-Likely Failure	Probability Estimation	Technique	Source
S-TaLiRo [32] †*	✓	·	·	Optimization	Open
Breach [33] *	✓	·	·	Optimization	Open
Rrt-Rex [26] †	✓	·	·	Path Planning	Closed
FalStar [75] † *	✓	·	·	Optimization, RL	Open
falsify [69] † *	✓	·	·	Reinforcement Learning	Closed
AST Toolbox [22]	✓	✓	·	Reinforcement Learning	Open
Reactis [129]	✓	·	·	Proprietary	Commercial
TestWeaver [131]	✓	·	·	Proprietary	Commercial
TrustworthySearch API [108]	✓	·	✓	Importance Sampling	Commercial

* Competed in ARCH-COMP 2019 [118].
† Accepts system specification in temporal logic (STL or MTL).

temporal logic to encode system requirements could also limit the applicability of these tools. With a common goal of ensuring that safety-critical systems are in fact safe, the availability of these tools allows users to provide feedback given their specific use-cases and experience. Continued tool and technique development is further encouraged by academically-driven competitions such as ARCH-COMP [118].

10. Conclusion

With the rapid increase of safety-critical autonomous systems operating with humans, it is imperative that we develop robust testing procedures that can ensure the safety of these systems. Due to the high level of system complexity, a comprehensive testing procedure must involve black-box validation strategies that seek to find failures of the autonomous system. This work describes the problems of falsification (where a single failure example is searched for), most-likely failure analysis (where the most likely failure is searched for), and failure probability estimation (where we seek a good estimate of the likelihood of failure).

With these goals defined, we outline a wide array of algorithms that have been used to accomplish these tasks. Global optimization, path planning, and reinforcement learning algorithms have been used to find falsifying examples, while importance sampling methods can be used to estimate the probability of failure even when it is close to zero. To address the problem of scalability, we describe approaches for decomposing the safety validation problem into more manageable components. We give a brief overview of the main applications for black-box safety validation including autonomous driving, and autonomous flight. Finally, we provide an overview of the existing tools that can be used to tackle these validation tasks.

References

[1] "Automated vehicles 3.0: Preparing for the future of transportation," U.S. Department of Transportation, Tech. Rep., 2018.

[2] "FAA aerospace forecast fiscal years 2020–2040," Federal Aviation Administration, Tech. Rep., 2019.

[3] M. J. Kochenderfer, J. E. Holland, and J. P. Chryssanthacopoulos, "Next-generation airborne collision avoidance system," *Lincoln Laboratory Journal*, vol. 19, no. 1, 2012.

[4] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, 2015.

[5] S. Russell and P. Norvig, *Artificial intelligence: A modern approach*, 4th ed. Prentice Hall, 2020.

[6] M. J. Kochenderfer, *Decision making under uncertainty: Theory and application*. MIT Press, 2015.

[7] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT Press, 2018.

[8] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, *et al.*, "Mastering the game of Go without human knowledge," *Nature*, vol. 550, no. 7676, 2017.

[9] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, *et al.*, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, 2016.

[10] O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev, *et al.*, "Grandmaster level in starcraft ii using multi-agent reinforcement learning," *Nature*, vol. 575, no. 7782, 2019.

[11] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing Atari with deep reinforcement learning," *ArXiv*, no. 1312.5602, 2013.

[12] S. Pillai, R. Ambruş, and A. Gaidon, "Superdepth: Self-supervised, super-resolved monocular depth estimation," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2019.

[13] K. He, G. Gkioxari, P. Dollár, and R. Girshick, "Mask R-CNN," in *International Conference on Computer Vision (ICCV)*, 2017.

[14] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, *Handbook of model checking*. Springer, 2018.

[15] J.-P. Katoen, "The probabilistic model checking landscape," in *ACM/IEEE Symposium on Logic in Computer Science (LICS)*, 2016.

[16] A. Platzer and J.-D. Quesel, "KeYmaera: A hybrid theorem prover for hybrid systems (system description)," in *International Joint Conference on Automated Reasoning (IJCAR)*, 2008.

[17] M. Fitting, *First-order logic and automated theorem proving*. Springer Science & Business Media, 2012.

[18] J. M. Schumann, *Automated theorem proving in software engineering*. Springer Science & Business Media, 2001.

[19] D. Yeh, "Autonomous systems and the challenges in verification, validation, and test," *IEEE Design & Test*, vol. 35, no. 3, 2018.

[20] C. Baier and J.-P. Katoen, *Principles of model checking*. MIT Press, 2008.

[21] G. Katz, C. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer, "Reluplex: An efficient smt solver for verifying deep neural networks," in *International Conference on Computer Aided Verification (CAV)*, 2017.

[22] M. Koren, S. Alsaif, R. Lee, and M. J. Kochenderfer, "Adaptive stress testing for autonomous vehicles," in *IEEE Intelligent Vehicles Symposium (IV)*, 2018.

[23] G. E. Fainekos and G. J. Pappas, "Robustness of temporal logic specifications for continuous-time signals," *Theoretical Computer Science*, vol. 410, no. 42, 2009.

[24] A. Donzé and O. Maler, "Robust satisfaction of temporal logic over real-valued signals," in *International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS)*, 2010.

[25] H. Yang, G. Fainekos, H. Sarjoughian, and A. Shrivastava, "Dynamic programming algorithm for computing temporal logic robustness," PhD thesis, Arizona State University, 2013.

[26] T. Dreossi, T. Dang, A. Donzé, J. Kapinski, X. Jin, and J. V. Deshmukh, "Efficient guiding strategies for testing of temporal properties of hybrid systems," in *NASA Formal Methods Symposium (NFM)*, 2015.

[27] G. Ernst, S. Sedwards, Z. Zhang, and I. Hasuo, "Fast falsification of hybrid systems using probabilistically adaptive input," in *International Conference on Quantitative Evaluation of Systems (QEST)*, 2019.

[28] Y. V. Pant, H. Abbas, and R. Mangharam, "Smooth operator: Control using the smooth robustness of temporal logic," in *IEEE Conference on Control Technology and Applications (CCTA)*, 2017.

[29] T. Akazaki, Y. Kumazawa, and I. Hasuo, "Causality-aided falsification," *Electronic Proceedings in Theoretical Computer Science*, vol. 257, 2017.

[30] Z. Zhang, I. Hasuo, and P. Arcaini, "Multi-armed bandits for boolean connectives in hybrid system falsification," in *Computer Aided Verification (CAV)*, 2019.

[31] M. J. Kochenderfer and T. A. Wheeler, *Algorithms for optimization*. MIT Press, 2019.

[32] Y. Annapureddy, C. Liu, G. Fainekos, and S. Sankaranarayanan, "S-TaLiRo: A Tool for Temporal Logic Falsification for Hybrid Systems," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2011.

[33] A. Donzé, "Breach, a toolbox for verification and parameter synthesis of hybrid systems," in *International Conference on Computer Aided Verification (CAV)*, 2010.

[34] N. Hansen and A. Ostermeier, "Adapting arbitrary normal mutation distributions in evolution strategies: The covariance matrix adaptation," in *IEEE International Conference on Evolutionary Computation*, 1996.

[35] M. A. Luersen and R. Le Riche, "Globalized nelder–mead method for engineering optimization," *Computers & Structures*, vol. 82, no. 23-26, 2004.

[36] L. Mathesen, S. Yaghoubi, G. Pedrielli, and G. Fainekos, "Falsification of cyber-physical systems with robustness uncertainty quantification through stochastic optimization with adaptive restart," English (US), in *International Conference on Automation Science and Engineering (CASE)*, Aug. 2019.

[37] J. Deshmukh, X. Jin, J. Kapinski, and O. Maler, "Stochastic local search for falsification of hybrid systems," in *International Symposium on Automated Technology for Verification and Analysis (ATVA)*, 2015.

[38] A. Adimoolam, T. Dang, A. Donzé, J. Kapinski, and X. Jin, "Classification and coverage-based falsification for embedded control systems," in *International Conference on Computer Aided Verification (CAV)*, 2017.

[39] H. Abbas, G. Fainekos, S. Sankaranarayanan, F. Ivančić, and A. Gupta, "Probabilistic temporal logic falsification of cyber-physical systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 12, no. 2s, 2013.

[40] A. Aerts, B. T. Minh, M. R. Mousavi, and M. A. Reniers, "Temporal logic falsification of cyber-physical systems: An input-signal-space optimization approach," in *IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2018.

[41] S. Yaghoubi and G. Fainekos, "Gray-box adversarial testing for control systems with machine learning components," in *ACM International Conference on Hybrid Systems: Computation and Control (HSCC)*, 2019.

[42] Q. Zhao, B. H. Krogh, and P. Hubbard, "Generating test inputs for embedded control systems," *IEEE Control Systems Magazine*, vol. 23, no. 4, 2003.

[43] X. Zou, R. Alexander, and J. McDermid, "Safety validation of sense and avoid algorithms using simulation and evolutionary search," in *International Conference on Computer Safety, Reliability, and Security (SafeComp)*, 2014.

[44] J. Mockus, *Bayesian approach to global optimization: Theory and applications*. Springer Science & Business Media, 2012, vol. 37.

[45] Y. Abeysirigoonawardena, F. Shkurti, and G. Dudek, "Generating adversarial driving scenarios in high-fidelity simulators," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2019.

[46] G. E. Mullins, P. G. Stankiewicz, R. C. Hawthorne, and S. K. Gupta, "Adaptive generation of challenging scenarios for testing and evaluation of autonomous vehicles," *Journal of Systems and Software*, vol. 137, 2018.

[47] S. Silvetti, A. Policriti, and L. Bortolussi, "An active learning approach to the falsification of black box cyber-physical systems," in *International Conference on Integrated Formal Methods (iFM)*, 2017.

[48] M. Dorigo, V. Maniezzo, and A. Colorni, "Ant system: Optimization by a colony of cooperating agents," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 26, no. 1, 1996.

[49] G. E. Fainekos and K. C. Giannakoglou, "Inverse design of airfoils based on a novel formulation of the ant colony optimization method," *Inverse Problems in Engineering*, vol. 11, no. 1, 2003.

[50] Y. S. R. Annapureddy and G. E. Fainekos, "Ant colonies for temporal logic falsification of hybrid systems," in *Annual Conference on IEEE Industrial Electronics Society (IECON)*, 2010.

[51] A. Corso and M. J. Kochenderfer, "Interpretable safety validation for autonomous vehicles," *ArXiv*, no. 2004.06805,

[52] S. M. LaValle, *Planning algorithms*. Cambridge University Press, 2006.

[53] S. M. Lavalle, "Rapidly-exploring random trees: A new tool for path planning," Tech. Rep., 1998.

[54] J. M. Esposito, J. Kim, and V. Kumar, "Adaptive RRTs for validating hybrid robotic control systems," in *Algorithmic Foundations of Robotics VI*, Springer, 2004, pp. 107–121.

[55] J. Kim, J. M. Esposito, and V. Kumar, "An RRT-based algorithm for testing and validating multi-robot controllers," Moore School of Electrical Engineering GRASP Lab, Tech. Rep., 2005.

[56] M. S. Branicky, M. M. Curtiss, J. Levine, and S. Morgan, "Sampling-based planning, control and verification of hybrid systems," *IEEE Proceedings - Control Theory and Applications*, vol. 153, no. 5, 2006.

[57] T. Dang, A. Donzé, O. Maler, and N. Shalev, "Sensitive state-space exploration," in *IEEE Conference on Decision and Control (CDC)*, 2008.

[58] T. Nahhal and T. Dang, "Test coverage for continuous and hybrid systems," in *Computer Aided Verification*, 2007.

[59] E. Plaku, L. E. Kavraki, and M. Y. Vardi, "Hybrid systems: From verification to falsification by combining motion planning and discrete search," *Formal Methods in System Design*, vol. 34, no. 2, 2009.

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

[61] M. Koschi, C. Pek, S. Maierhofer, and M. Althoff, "Computationally efficient safety falsification of adaptive cruise control systems," in *IEEE International Conference on Intelligent Transportation Systems (ITSC)*, 2019.

[62] E. Thiémard, "An algorithm to compute bounds for the star discrepancy," *Journal of Complexity*, vol. 17, no. 4, 2001.

[63] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *The International Journal of Robotics Research*, vol. 30, no. 7, 2011.

[64] H. G. Bock and K.-J. Plitt, "A multiple shooting algorithm for direct solution of optimal control problems," *IFAC Proceedings Volumes*, vol. 17, no. 2, 1984.

[65] M. Diehl, H. G. Bock, H. Diedam, and P.-B. Wieber, "Fast direct multiple shooting algorithms for optimal robot control," in *Fast Motions in Biomechanics and Robotics*, Springer, 2006, pp. 65–93.

[66] A. Zutshi, S. Sankaranarayanan, J. V. Deshmukh, and J. Kapinski, "A trajectory splicing approach to concretizing counterexamples for hybrid systems," in *IEEE Conference on Decision and Control (CDC)*, 2013.

[67] A. Zutshi, J. V. Deshmukh, S. Sankaranarayanan, and J. Kapinski, "Multiple shooting, CEGAR-based falsification for hybrid systems," in *International Conference on Embedded Software (ICESS)*, 2014.

[68] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[69] T. Akazaki, S. Liu, Y. Yamagata, Y. Duan, and J. Hao, "Falsification of cyber-physical systems using deep reinforcement learning," in *International Symposium on Formal Methods (FM)*, 2018.

[70] R. Lee, M. J. Kochenderfer, O. J. Mengshoel, G. P. Brat, and M. P. Owen, "Adaptive stress testing of airborne collision avoidance systems," in *Digital Avionics Systems Conference (DASC)*, 2015.

[71] M. Koren, A. Corso, and M. J. Kochenderfer, "The adaptive stress testing formulation," *ArXiv*, no. 2004.04293, 2020.

[72] X. Qin, N. Aréchiga, A. Best, and J. Deshmukh, "Automatic testing and falsification with dynamically constrained reinforcement learning," *ArXiv*, no. 1910.13645, 2019.

[73] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, *et al.*, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, 2016.

[74] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, *et al.*, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," *ArXiv*, no. 1712.01815, 2017.

[75] Z. Zhang, G. Ernst, S. Sedwards, P. Arcaini, and I. Hasuo, "Two-layered falsification of hybrid systems guided by Monte Carlo tree search," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, 2018.

[76] M. Wicker, X. Huang, and M. Kwiatkowska, "Feature-guided black-box safety testing of deep neural networks," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2018.

[77] R. Delmas, T. Loquen, J. Boada-Bauxell, and M. Carton, "An evaluation of Monte Carlo tree search for property falsification on hybrid flight control laws," in *International Workshop on Numerical Software Verification*, 2019.

[78] K. D. Julian, R. Lee, and M. J. Kochenderfer, "Validation of image-based neural network controllers through adaptive stress testing," *ArXiv*, no. 2003.02381, 2020.

[79] L. Kocsis and C. Szepesvári, "Bandit based Monte-Carlo planning," in *European Conference on Machine Learning (ECML)*, 2006.

[80] R. Coulom, "Computing 'ELO ratings' of move patterns in the game of go," *ICGA Journal*, vol. 30, no. 4, 2007.

[81] G. M. J. Chaslot, M. H. Winands, H. J. V. D. HERIK, J. W. Uiterwijk, and B. Bouzy, "Progressive strategies for Monte Carlo tree search," *New Mathematics and Natural Computation*, vol. 4, no. 03, 2008.

[82] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, 2015.

[83] S. Gu, E. Holly, T. Lillicrap, and S. Levine, "Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2017.

[84] A. Corso, P. Du, K. Driggs-Campbell, and M. J. Kochenderfer, "Adaptive stress testing with reward augmentation for autonomous vehicle validation," in *IEEE International Conference on Intelligent Transportation Systems (ITSC)*, 2019.

[85] M. Koren and M. J. Kochenderfer, "Efficient autonomy validation in simulation with adaptive stress testing," in *IEEE International Conference on Intelligent Transportation Systems (ITSC)*, 2019.

[86] V. Behzadan and A. Munir, "Adversarial reinforcement learning framework for benchmarking collision avoidance mechanisms in autonomous vehicles," *IEEE Intelligent Transportation Systems Magazine*, 2019.

[87] S. Kuutti, S. Fallah, and R. Bowden, "Training adversarial agents to exploit weaknesses in deep control policies," *ArXiv*, no. 2002.12078, 2020.

[88] M. Koren and M. J. Kochenderfer, "Adaptive stress testing without domain heuristics using go-explore," *ArXiv*, no. 2004.04292, 2020.

[89] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Advances in Neural Information Processing Systems (NIPS)*, 2000.

[90] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in *International Conference on Machine Learning (ICML)*, 2015.

[91] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International Conference on Machine Learning (ICML)*, 2016.

[92] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *ArXiv*, no. 1509.02971, 2015.

[93] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, 1997.

[94] A. Ecoffet, J. Huizinga, J. Lehman, K. O. Stanley, and J. Clune, "Go-explore: A new approach for hard-exploration problems," *ArXiv*, no. 1901.10995, 2019.

[95] T. Salimans and R. Chen, "Learning Montezuma's Revenge from a single demonstration," *ArXiv*, no. 1812.03381, 2018.

[96] A. Legay, B. Delahaye, and S. Bensalem, "Statistical model checking: An overview," in *International Conference on Runtime Verification (RV)*, 2010.

[97] G. Agha and K. Palmskog, "A survey of statistical model checking," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 28, no. 1, 2018.

[98] G. Hahn, "Sample sizes for Monte Carlo simulation," *IEEE Transactions on Systems, Man, and Cybernetics*, 1972.

[99] Y. Kim and M. J. Kochenderfer, "Improving aircraft collision risk estimation using the cross-entropy method," *Journal of Air Transportation*, vol. 24, no. 2, 2016.

[100] R. Y. Rubinstein and D. P. Kroese, *The cross-entropy method: A unified approach to combinatorial optimization, Monte Carlo simulation and machine learning*. Springer Science & Business Media, 2013.

[101] P.-T. De Boer, D. P. Kroese, S. Mannor, and R. Y. Rubinstein, "A tutorial on the cross-entropy method," *Annals of Operations Research*, vol. 134, no. 1, 2005.

[102] M. O'Kelly, A. Sinha, H. Namkoong, R. Tedrake, and J. C. Duchi, "Scalable end-to-end autonomous vehicle testing via rare-event simulation," in *Advances in Neural Information Processing Systems (NIPS)*, 2018.

[103] D. Zhao, H. Lam, H. Peng, S. Bao, D. J. LeBlanc, K. Nobukawa, and C. S. Pan, "Accelerated evaluation of automated vehicles safety in lane-change scenarios based on importance sampling techniques," *IEEE Transactions on Intelligent Transportation Systems*, vol. 18, no. 3, 2016.

[104] Z. Huang, H. Lam, D. J. LeBlanc, and D. Zhao, "Accelerated evaluation of automated vehicles using piecewise mixture models," *IEEE Transactions on Intelligent Transportation Systems*, vol. 19, no. 9, 2017.

[105] S. Sankaranarayanan and G. Fainekos, "Falsification of temporal properties of hybrid systems using the cross-entropy method," in *ACM International Conference on Hybrid Systems: Computation and Control (HSCC)*, 2012.

[106] H. Kahn and T. E. Harris, "Estimation of particle transmission by random sampling," *National Bureau of Standards Applied Mathematics Series*, vol. 12, 1951.

[107] Z. I. Botev and D. P. Kroese, "An efficient algorithm for rare-event probability estimation, combinatorial optimization, and counting," *Methodology and Computing in Applied Probability*, vol. 10, no. 4, 2008.

[108] J. Norden, M. O'Kelly, and A. Sinha, "Efficient black-box assessment of autonomous vehicle safety," *ArXiv*, no. 1912.03618, 2019.

[109] F. Cérou and A. Guyader, "Adaptive multilevel splitting for rare event analysis," *Stochastic Analysis and Applications*, vol. 25, no. 2, 2007.

[110] Z. Huang, Y. Guo, M. Arief, H. Lam, and D. Zhao, "A versatile approach to evaluating and testing automated vehicles based on kernel methods," in *American Control Conference (ACC)*, 2018.

[111] J. Uesato, A. Kumar, C. Szepesvari, T. Erez, A. Ruderman, K. Anderson, N. Heess, P. Kohli, *et al.*, "Rigorous agent evaluation: An adversarial approach to uncover catastrophic failures," *ArXiv*, no. 1812.01647, 2018.

[112] J. S. Sadowsky and J. A. Bucklew, "On large deviations theory and asymptotically efficient Monte Carlo estimation," *IEEE Transactions on Information Theory*, vol. 36, no. 3, 1990.

[113] J. A. Suykens and J. Vandewalle, "Least squares support vector machine classifiers," *Neural Processing Letters*, vol. 9, no. 3, 1999.

[114] A. Corso, R. Lee, and M. J. Kochenderfer, "Scalable autonomous vehicle safety validation through dynamic programming and scene decomposition," *ArXiv*, no. 2004.06801,

[115] T. Dreossi, A. Donzé, and S. A. Seshia, "Compositional falsification of cyber-physical systems with machine learning components," *Journal of Automated Reasoning*, vol. 63, no. 4, Dec. 2019.

[116] C. E. Tuncali, G. Fainekos, D. Prokhorov, H. Ito, and J. Kapinski, "Requirements-driven test generation for autonomous vehicles with machine learning components," *IEEE Transactions on Intelligent Vehicles*, 2019.

[117] J. Hu, J. Lygeros, and S. Sastry, "Towards a theory of stochastic hybrid systems," in *International Workshop on Hybrid Systems: Computation and Control (HSCC)*, 2000.

[118] G. Ernst, P. Arcaini, A. Donze, G. Fainekos, L. Mathesen, G. Pedrielli, S. Yaghoubi, Y. Yamagata, and Z. Zhang, "Arch-comp 2019 category report: Falsification," *EPiC Series in Computing*, vol. 61, 2019.

[119] C. E. Tuncali, B. Hoxha, G. Ding, G. Fainekos, and S. Sankaranarayanan, "Experience report: Application of falsification methods on the uxas system," in *NASA Formal Methods Symposium (NFM)*, 2018.

[120] R. Lee, O. J. Mengshoel, A. K. Agogino, D. Giannakopoulou, and M. J. Kochenderfer, "Adaptive stress testing of trajectory planning systems," in *AIAA Scitech Intelligent Systems Conference (IS)*, 2019.

[121] B. Hoxha, H. Abbas, and G. Fainekos, "Benchmarks for temporal logic requirements for automotive systems," in *International Workshop on Applied Verification for Continuous and Hybrid Systems (ARCH)*, vol. 34, 2015.

[122] X. Jin, J. V. Deshmukh, J. Kapinski, K. Ueda, and K. Butts, "Powertrain control verification benchmark," in *ACM International Conference on Hybrid Systems: Computation and Control (HSCC)*, 2014.

[123] S. Schuler, F. D. Adegas, and A. Anta, "Hybrid modelling of a wind turbine (benchmark proposal)," in *International Workshop on Applied Verification for Continuous and Hybrid Systems (ARCH)*, ser. EPiC Series in Computing, vol. 43, 2017.

[124] A. Fehnker and F. Ivančić, "Benchmarks for hybrid systems verification," in *International Workshop on Hybrid Systems: Computation and Control (HSCC)*, 2004.

[125] R. D. Diwakaran, S. Sankaranarayanan, and A. Trivedi, "Analyzing neighborhoods of falsifying traces in cyber-physical systems," in *International Conference on Cyber-Physical Systems (ICCPS)*, 2017.

[126] G. Fainekos, B. Hoxha, and S. Sankaranarayanan, "Robustness of specifications and its applications to falsification, parameter mining, and runtime monitoring with S-TaLiRo," in *International Conference on Runtime Verification (RV)*, 2019.

[127] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "OpenAI gym," *ArXiv*, no. 1606.01540, 2016.

[128] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," *SIAM Review*, vol. 59, no. 1, 2017.

[129] "Testing and Validation of Simulink Models with Reactis," Reactive Systems Inc., Tech. Rep. RSITR 1.11, Oct. 2013.

[130] "Finding Bugs in C Programs with Reactis for C," Reactive Systems Inc., Tech. Rep. RSITR 2.5, Aug. 2011.

[131] A. Junghanns, J. Mauss, and M. Tatar, "Tatar: Testweaver-a tool for simulation-based test of mechatronic designs," in *International Modelica Conference*, 2008.